5-19-2017

A Distributed Graph Approach for Pre-processing Linked RDF Data Using Supercomputers

Michael J. Lewis
The University of Illinois at Chicago, mlewis3@uic.edu

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Venkatram Vishwanath
Argonne National Laboratory

Michael J. Papka
Argonne National Laboratory and Northern Illinois University

Andrew Johnson
The University of Illinois at Chicago

Recommended Citation

Michael J. Lewis, George K. Thiruvathukal, Venkatram Vishwanath, Michael E. Papka, and Andrew Johnson, A Distributed Graph Approach for Pre-Processing Linked Data Using Supercomputers, In Proceedings of International Workshop on Semantic Big Data 2017 (SBD 2017) at ACM SIGMOD 2017.

A Distributed Graph Approach for Pre-processing Linked RDF Data Using Supercomputers*

Michael J. Lewis
University of Illinois at Chicago
Chicago, IL 60607
mlewis3@uic.edu

George K. Thiruvathukal
Loyola University Chicago
Chicago, IL 60660
gkt@cs.luc.edu
Argonne National Laboratory
Argonne, IL 60439
gkt@anl.gov

Venkatram Vishwanath
Argonne National Laboratory
Argonne, IL 60439
venkat@anl.gov

Michael E. Papka
Argonne National Laboratory
Argonne, IL 60439
papka@anl.gov
Northern Illinois University
Dekalb, IL 60115
papka@niu.edu

Andrew Johnson
University of Illinois at Chicago
Chicago, IL 60607
ajohnson@uic.edu

ABSTRACT

Efficient RDF, graph based queries are becoming more pertinent based on the increased interest in data analytics and its intersection with large, unstructured but connected data. Many commercial systems have adopted distributed RDF graph systems in order to handle increasing dataset sizes and complex queries. This paper introduces a distribute graph approach to pre-processing linked data. Instead of traversing the memory graph, our system indexes pre-processed join elements that are organized in a graph structure. We analyze the Dbpedia data-set (derived from the Wikipedia corpus) and compare our access method to the graph traversal access approach which we also devise. Results show from our experiments that the distributed, pre-processed graph approach to accessing linked data is faster than the traversal approach over a specific range of linked queries.

CCS CONCEPTS

• **Computing methodologies → Distributed algorithms**;

KEYWORDS

RDF; High Performance Computing; Distributed Algorithms

ACM Reference format:
Michael J. Lewis, George K. Thiruvathukal, Venkatram Vishwanath, Michael E. Papka, and Andrew Johnson. 2017. A Distributed Graph Approach for Pre-processing Linked RDF Data Using Supercomputers. In *Proceedings of SBD'17, Chicago, IL, USA, May 19-19, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3066911.3066913

*Produces the permission block, and copyright information

SBD'17, Chicago, IL, USA
 978-1-4503-4987-1/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3066911.3066913

1 INTRODUCTION

RDF query retrieval systems have been used to extract information from data ontologies over areas covering pharmaceutical, biomedicine, social media, and network security just to name a few. With larger datasets and the range of query complexities, it is important for query systems to keep up with the demanding workload, not only in system architecture but improvements in query retrieving algorithms.

Much of the evolution of RDF systems have focused on the improvement of fast data access over increasingly large datasets, and complex queries. Vertical partitioning applied in [11], [25], [19], [2] to access groups of stored triple data. Compression techniques have been used in systems RDF-3X [19], [2]. RDF store systems [12] [1] [21] allow fast access to triple stores by outsourcing to large scale key-value based database systems. Data scalable systems [9] [28] use the map-reduce algorithm to query to scale large RDF datasets. Graph partitioning RDF systems [5] [8] utilize graph partitioning algorithms in order to create highly coupled subgraphs for the purpose of reducing node to node communication type. RDF graph retrieval systems [16] [17] [23] utilize a distributed memory graph and traverse through connected nodes in order to retrieve query results. Path representation models [6] [14] provide techniques to represent and access paths of linked data within an index form.

1.1 Resource Descriptive Framework

RDF [13] is a language/data model used in the Semantic Web community to extract contextual relational and hierarchical data. The core data unit is composed of a three term (subject,predicate,object-value). Each term is a resource and can represent a URL. A literal can only be used within the the object term. Statements, also referred to as *triples*, are able to link to each other like Lego blocks over matching terms: subject-subject (s-s), predicate-predicate (p-p), (object-object) (o-o), subject-object (s-o), and object-subject (o-s). From these connections a dataset of triples can be transformed into an *RDF-graph* as shown in Figure 1.

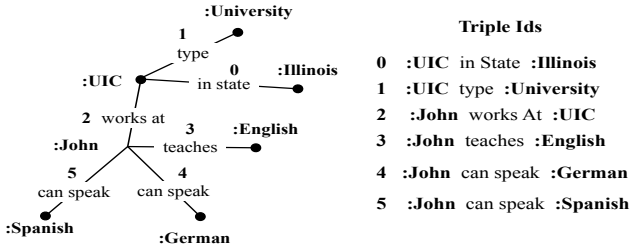


Figure 1: An RDF graph and its join connections.

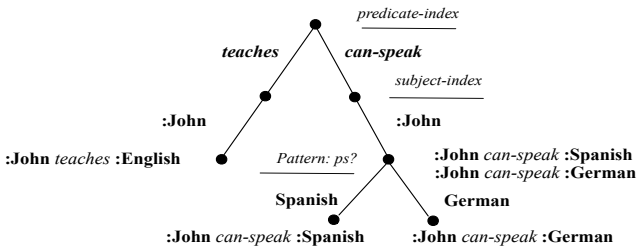


Figure 2: PS? patterns stored within a PSO hierarchy.

1.2 Queries

A typical RDF query is a collection of query statements. A query statement can represent a URL, value, variable *?var* or blank term *?*. URL is a unique resource name that represents a query term. A blank term: *?* can represent any value as long as there is a exist a triple for it. A query statement that has least one variable or blank term is referred to as a pattern, to signify that multiple triples can belong to it. For example in Figure 1, a query pattern :John *can-speak* ? (sp?) would include the triples: :John *can-speak* :Spanish and :John *can-speak* :German. Queries containing blank terms are more complex in terms of its low selectivity. The extraction net is bigger meaning that query results sizes are larger, however it does not necessarily mean the query extractions came from a broad range of locations within the dataset. Longer queries with low selectivity have another type of complexity where large groups of intermediate data have to be joined. Our query generation is focused on these two aspects of complexity where other query generation systems such as LUBM [7] also take in account the quality and completeness of knowledge base extractions, which is not the focus of this paper.

1.3 SPARQL

SPARQL [20], an RDF compliant query language offers expressions to satisfy graphical based extractions from linked triples.SPARQL allows the user to use variables to represent unknown triple components, as a way to provide an intersecting point on overlapping triple patterns. A SPARQL compliant query system should handle graph extractions from like terms (subject-subject, predict-predicate, object-object) links as well from linked terms, subject-object (s-o) and object-subject (o-s) connections. The script below shows an example query s-s linked query using the rdf-graph from Figure 1.

```
Select ?workers
Where {
        (?workers work at :UIC ).
        (?workers teach :English ).
}
Results:  :John
```

1.4 Our Contribution

This paper introduces Mantona, an RDF query processing system, written in C++ using the Message Passing Interface (MPI). Mantona is able to pre-processes conjunctive triple-triple connections, and utilize these store joins to expedite query retrievals. Our contribution is the following.

(1) Our unique graph-generation algorithm, designed to pre-processes conjunctive s-s, o-s, and s-o joins into a graph structure.
(2) A graph-retrieval algorithm, designed to retrieve conjunctive pattern based queries by indexing the join data from the graph structure that match the query patterns.
(3) A graph traversal query matching algorithm to mimic the query retrieval methods on RDF graph traversal access based systems.
(4) A random query generator. This is used to generate random pattern based queries of different complexity (based on the number of blank nodes).

We compare query retrieval performances with different query types using the Mantona graph-retrieval algorithm and the graph-traversal algorithm. Results show faster retrieval times using the Mantona graph-retrieval algorithm. We also compare build times for graph construction and neighbor construction.

2 RELATED SYSTEMS

2.1 Vertical Partitioning

Vertical partitioning is used as a technique to facilitate data access to a minimum of tables; grouping triple data from a common key(s). If the key is based on a subject term for example, then all the triple data from that index will be stored and sorted based on the subject terms. RDF-3X [19], HexaStore [25], BitMat [2] and the key-store DBMS based system [12] provide vertical access to triple stores based on set of key(s) combinations over a query term(s). RDF-3X and Hexastore use a B-tree index to access pattern data over 6 types of hierarchies (SPO,SOP,OSP, OPS,PSO,POS).

2.2 Data Communication Tools

Hadoop [3] provides a communication efficient, scalable environment where a user can develop their own RDF query system. Hadoop allows users to tap into the map-reduce [4] algorithm in order to create data-scalable jobs. Shard [22] and Scalable Sparql [9] incorporates the Hadoop map-reduce as the data-flow framework in order to process queries to scale big data. Spark [15], allows users to transparently utilize memory components across processors, and provides a graph engine and API: GraphX [26] that provide users the capability to create their own data-graph. Mantona uses the Message Passage Interface (MPI) for data communication. MPI is a message communication systems for distributed systems, commonly used with supercomputers and highly coupled systems. It provides function calls for node to node communication, 1 to many

communication, many to many communication where each processor can receive data information from every other processor , and a many (all) to many (all) communication where each process knows what data every other processor has.

2.3 Graph Partitioning Systems

Cluster based RDF systems [24] [8] [5] use partitioning algorithms to store regions of triples based on its community of neighbor nodes. Partout [5] uses the partitioning tool METIS [10] to find the k number of partitions that created the least amount of edges among each other. Linked queries however can be unpredictable to predict, unlikely triples can be found to be connected to each other through a series of s-o,o-s connections. An overlapping strategy of duplicating triple nodes over processors is used in [9]. These nodes represent a shared link of connections across processors, but this technique can only offer a short range solution for pattern linked queries.

2.4 Graph Access Systems

Distributed parallel RDF data systems: Trinity [23], Neo4J [16], Cray Graph Engine [17] use their own tailored highly coupled communication environment and effectively utilize memory storage to reduce the data communication cost. Trinity has shown through large scale experimentation using LUBM and DBpedia [18] generated datasets to outperform RDF-3X and BitMat. Trinity stores its data in a memory graph where nodes are the individual triple terms. Each node has an adjacency list of incoming and outgoing neighbor nodes. The collection of graph nodes residing on an individual processor are grouped together based on a SPO or OPS index.

2.5 Path Based Indices

The connection between path indices and pre-processed joins is in the creation of an organizational structure to index paths of connected data. Early research initially covered by Yamamotoa et al. [27] created structures for generating path indices from XML documents. Matono et al. [14] proposed a technique for translating RDF path expressions into suffix arrays using Directed Acyclic graphs extracted from an RDF dataset and/or schema. In Groppe et al. [6] joins were indexed in a hash-map over s-s,s-p,s-o,p-p,p-o and o-o connections for one join , two triple patterns and multiple joins over multiple triple patterns. With Grin [24] the RDF-graph is partitioned over center nodes, that adhere to a particular index. Using a center indexing formula queries can be determined if component lies within the radius of any center node.

3 MANTONA SYSTEM

Mantona name comes from the Sotho word meaning chiefs, where a chief can be viewed as the implemented code within a processor. Each *chief* governs their realm (graph) of linked RDF data. Mantona pre-processes RDF data in the form of paths within the RDF-graph. Mantona first processes an RDF-graph based on s-s,o-s,and s-o links and partitions node assignments to each processor. Each of these triples are referred to as a *root-id*. Each processor generates its own set of sub-graphs we term *root-graphs* for each of its assigned *root-ids*. A *root-graph* is composed of nodes termed *path-nodes*. Each

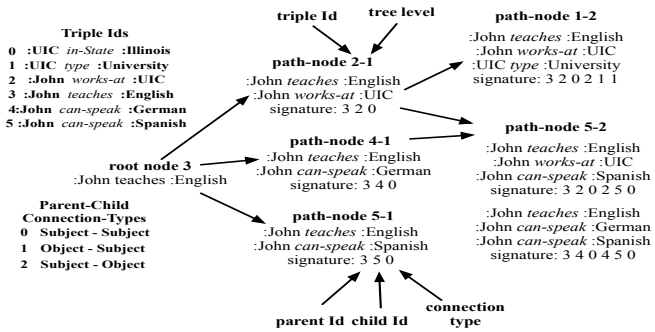


Figure 3: Root-graph from *root-id* 3, from Figure 1.

path-node contains a list of connected triples : *triple-product* that are generated from the resulting join operations stemming from from the path of intermediate path nodes up to the ending path-node starting from the root node. Figure 3 shows a root-graph from root pattern id *:john* teaches *:English*.

3.1 Path-signature

Each list of triple-products coming from a path node is labeled based on its connection signature. A connection signature is composed of a series of id that specifies the triples that are being connected and its type of link connection: <*connected triple id*> <*in-coming triple id*> <*connectionType*>. A connection type : 0 specifies a s-s connection, 1, o-s connection and 2 s-o connection. The triple-product under path-node 1-2 in Figure 3, has the connection signature *3 2 0 2 1 1*. Root id 3 *:john teaches :English* connects with triple id 2*:john works at :UIC* based on the subject-subject type specification 0. The next connection has id 1 *UIC type University* connecting with id 2 *:john works at :UIC* based on a object-subject specification 1. Path nodes are labeled by the ending connecting triple id and the graph depth. Path-node 1-2 in Figure 3; the 2 specifies the depth and 1 is the end connecting triple id. Every triple-product within that path-node will have the last connection to be *:UIC type :University*.

3.2 Graph-Cache Generation Algorithm

Here we show the root-graph generation algorithm and explain the variables and basic functions within the algorithm. Each processor has a set of root-graphs (*rootGraphList*). For each depth of the growing root-graph the total list (*tripleList*) of triple ids (to be potentially connected to the graph) are checked at the leaf nodes *fringe-nodes*. The *isIn* function determines if there are any s-s, o-s, s-o connections between the incoming *id* and the ids within each of the triple products residing within the fringe-node. If there is a connection, a join (*applyJoin*) is made at that connecting triple within the triple-product to create the new linked triple product and is added (*insertInPathNode*) to a new path-node. This path-node will become the newest addition to the root-graph and it contains all the the linked triple products of the common ending *id*. All new path-nodes are put on a temporary fringe list *addToList*. When all the fringe nodes have been visited, the new path-nodes become the fringe nodes *swapNodes* and the same procedure continues at the next depth (Algorithm 1).

```
procedure GRAPH-CACHE GENERATOR ;
    for depth ← 1 to maxDepth do
     foreach rootGraph in rootGraphList do
        foreach fringeNode in rootGraph do
            foreach id in tripleList do
                tId = IsIn(id, fringeNode) ;
                if tId > 0 then
                    tp = applyJoin(tId, id) ;
                    insertInPathNode(tp);
                    addToGraph(pathNode);
                    addToList(pathNode);
                end
            end
        end
        swapNodes(pathNode, fringeNodes) ;
     end
    end
end procedure
```

Algorithm 1: Graph-Cache Generation Algorithm

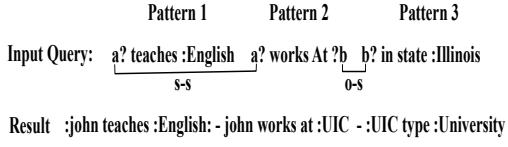


Figure 4: A sample input string to a Mantona job.

3.3 Node-Traversal Algorithm

Mantona has a node traversal algorithm that traverses through all the paths that are represented in a linked query and returns the results only from the matched paths. This is a recursive algorithm, starting at the root-id from *MatchedGraphList*, in which the root-id matches the the first pattern within the query pattern. A tripleProduct *tp* at depth 0 is created from root-id, is inserted in a list of triple products *tpList* and sent to *traversePath(depth,tpList)*. At each call, the depth is checked to see if it is at *maxDepth*. If so the resultant output (triple matches) is printed out, otherwise the traversal algorithm continues to expand the set of triple products (like newly grown branches of a tree) *newTpList* that match with the query pattern at the current depth. The list of neighbors are retrieved from the last id of triple product which represents the previous depth. The generatetps function generates a set of triple products resulting from the join of the neighbor id to any of the ids within the triple product (Algorithm 2).

3.4 Mantona Query Processing

Query processing starts with each processor taking from the Mantona random query generator, a linked query pattern string as shown in Figure 4. Mantona parses this string to produce the list of query patterns at each depth and determine the bounded and unbounded terms in each of the patterns. Each process finds if their root-ids match the first query pattern. MatchedRootGraphs represents all root-graphs that have the matching root-id.

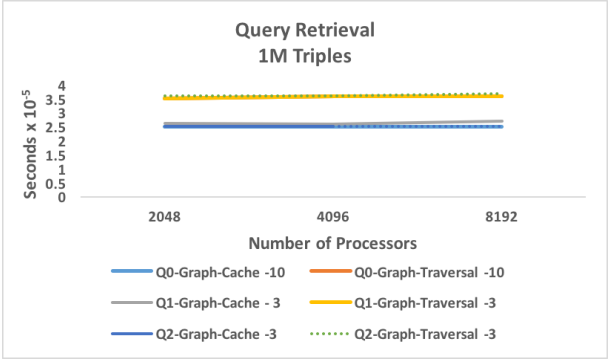
```
procedure NODE TRAVERSAL ALGORITHM ;
    foreach root-id in matchedGraphList do
     tp = generate(root − id) ;
     insert(tp, tpList) ;
     traversePath(1, tpList)
    end
end procedure
procedure TRAVERSEPATH(depth, tpList) ;
    if depth == queryDepth then
     printResult(pathNodes) ;
     return ;
    end
    instantiate(newtpList) ;
    foreach tp in tpList do
     foreach neighbor from tp[depth − 1] do
        generatetps(neighbor, tp, newtpList) ;
     end
    end
    deletetpList ;
    traversePath(depth+1,newtpList) ;
end procedure
```

Algorithm 2: Mantona Node Traversal Algorithm

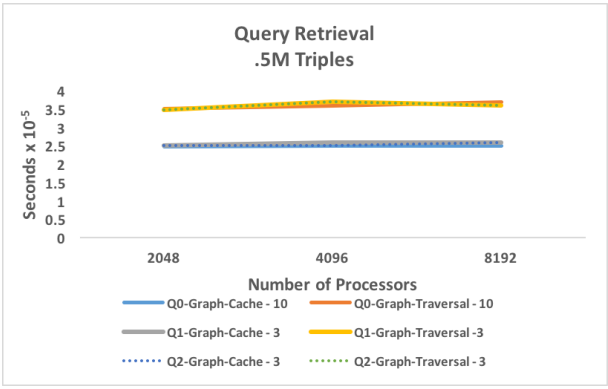
The *getNodes* function retrieves all qualifying path-nodes at the *queryDepth* level. So if the the input string consists of 5 linked patterns , Mantona will check all the path-nodes at tree level 4 , and will only accept the path-nodes where its ending connected triple id matches the 5th pattern. Mantona iterates over all the triple-products *tp* within the path-node(s) and compares each connecting triple id and link type to the correlating pattern. If the triple product matches all the patterns in the query in the right order, then its results are printed out (Algorithm 3).

```
procedure GRAPH RETRIEVAL ;
    foreach rootGraph in MatchedRootGraphs do
     pathNodeList = getNodes(queryDepth) ;
     foreach pathNode in pathNodeList do
        foreach tp in pathNode do
            matchingTp = true;
            foreach id,type,index in tp do
                if id,type not in pattern[index] then
                    matchingTriple = false ;
                end
            end
            if matchingTp == true then
                printOutput(tp);
            end
        end
     end
    end
end procedure
```

Algorithm 3: Mantona Graph-Cache Retrieval Algorithm

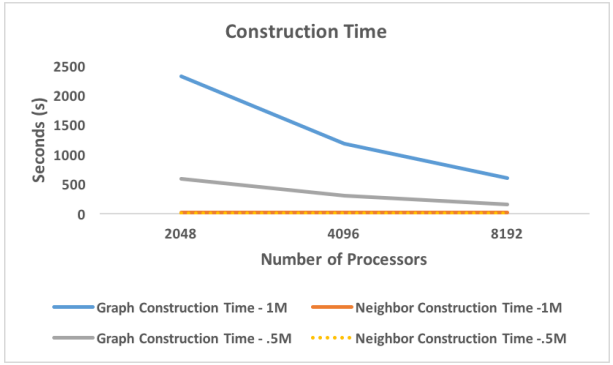


(a) 1000000 Triples



(b) 500000 Triples

Figure 5: Query results a) 1M triples b) .5M triples. Index on legend show result sizes.



(a) Graph-store algorithm

Figure 6: Graph construction and neighbor times over 1M, .5M triple dataset.

4 RESULTS

4.1 Experimental Setup

We ran Mantona query jobs on the Argonne Supercomputer: Cetus and Mira. Cetus was used as the preliminary test-bed. Cetus has 4096 nodes with 16 cores per node. Each core has a 1GB memory capacity. Mira was used for larger scale experiments. Mira has 49,152 nodes with 16 cores per node. Each core has a 1GB memory capacity. Both Cetus and Mira use PowerPC A2 1600 MHz processor and are connected to the same GPFS file system that has a 24 PB file storage capacity.

Our RDF data comes from the 2016 wiki-DBpedia datasets at http://wiki.dbpedia.org at 5.8 GB. From this data-set we extracted 3 files that produced 500,000 and 1,000,000 triples respectively. We used Cetus to test the 500,000 triples and Mira, the 1,000000 triple using 2048, 4096, and 8192 processors. For each job we recorded graph build time (for the graph-caching algorithm), neighbor build time for the node traversing algorithm and query retrieval time for both algorithms.

For each run we generated four types of queries based on ranges of query selectivity. Q0: a two pattern two blank nodes query. Q1: a two pattern, four blank nodes query. Q2: a three pattern, six blank nodes query. Query Q3 used the Mantona graph cache algorithm up to depth one, then used the node traversing algorithm for the last depth.

Results show (Figure 5) from from both the 1M triple dataset and the .5M triple dataset that the graph-cache algorithm has lower retrieval times from every type of query. Query complexity did not hold too much significance in retrieval timings. This mainly has to do with the result size from the query types being small. Q0 revealed 10 results,Q1 and Q2 produced 3 results as shown on the legend. With very fast results for both algorithms, the increase in processor times did not affect the query times in comparison to the extra time generated from processor synchronization.

Neighbor construction times (Figure 6) were significantly lower than graph construction times. However graph construction times scaled in direct proportion processor times.

5 CONCLUSIONS

The Mantona's graph-cache retrieval achieves better query times as compared to retrieving queries through path traversals within a cached memory RDF-graph. There are limitations based on memory size and triple count to how much depth of the graph can be pre-processed, but the Q2 results show that there can be mix of the two algorithms and still achieve better query times than the traversal algorithm.

More experiments need to be done on a large scale triple level, with varying processor sizes and query complexities to further understand what types of query patterns give better retrieval results for either algorithm. Even though the cache algorithm shows better results from all the queries, the queries did not cover the breadth and depth of the dataset. Queries that have OR cases and not just the conjunctive AND should be considered in order to increase the complexity level.

Further consideration of this work is to expand Mantona to include query planning algorithms, based on dynamic programming of triple binding sizes or frequency of terms, to determine what query ordering produces a smaller amount of joins and thus reduce query retrieval timings.

6 ACKNOWLEDGMENTS

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. The author Lewis would like to give acknowledgements to Dr. Vishwanath, Dr. Thiruvathukal, and Dr. Papka for their mentor-ship and support.

REFERENCES

[1] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. 2012. AMADA: web data repositories in the amazon cloud. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2749–2751.

[2] Medha Atre, Vineet Chaoji, Mohammed J Zaki, and James A Hendler. 2010. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In *Proceedings of the 19th international conference on World wide web*. ACM, 41–50.

[3] Milind Bhandarkar. 2010. MapReduce programming with apache Hadoop. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–1.

[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[5] Luis Galárraga, Katja Hose, and Ralf Schenkel. 2014. Partout: a distributed engine for efficient RDF processing. In *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 267–268.

[6] Sven Groppe, Jinghua Groppe, and Volker Linnemann. 2007. Using an index of precomputed joins in order to speed up SPARQL processing.. In *ICEIS (1)*. 13–20.

[7] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2 (2005), 158–182.

[8] Katja Hose and Ralf Schenkel. 2013. WARP: Workload-aware replication and partitioning for RDF. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*. IEEE, 1–6.

[9] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1123–1134.

[10] George Karypis and Vipin Kumar. 1995. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

[11] Dave Kolas, Ian Emmons, and Mike Dean. 2009. Efficient linked-list rdf indexing in parliament. *SSWS* 9 (2009), 17–32.

[12] Günter Ladwig and Andreas Harth. 2011. CumulusRDF: linked data management on nested key-value stores. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*. 30.

[13] Frank Manola, Eric Miller, and B McBride. 2004. Rdf primer w3c recommendation 10 february 2004. (2004).

[14] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. 2003. An indexing scheme for RDF and RDF schema based on suffix arrays. In *Proceedings of the First International Conference on Semantic Web and Databases*. CEUR-WS. org, 140–157.

[15] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7. http://jmlr.org/papers/v17/15-237.html

[16] Justin J Miller. 2013. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, Vol. 2324. 36.

[17] David Mizell. 2016. How the Cray Graph Engine Manages Graph Databases. (2016). http://www.cray.com/blog/how-cray-graph-engine-manages-graph-databases/

[18] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. 2011. DBpedia SPARQL benchmark–performance assessment with real queries on real data. In *International Semantic Web Conference*. Springer, 454–469.

[19] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases* 19, 1 (2010), 91–113.

[20] Eric Prud, Andy Seaborne, and others. 2006. SPARQL query language for RDF. (2006).

[21] Roshan Punnoose, Adina Crainiceanu, and David Rapp. 2012. Rya: a scalable RDF triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*. ACM, 4.

[22] Kurt Rohloff and Richard E Schantz. 2010. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*. ACM, 4.

[23] Bin Shao, Haixun Wang, and Yatao Li. 2012. The trinity graph engine. *Microsoft Research* (2012), 54.

[24] Octavian Udrea, Andrea Pugliese, and VS Subrahmanian. 2007. GRIN: A graph based RDF index. In *AAAI*, Vol. 1. 1465–1470.

[25] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1008–1019.

[26] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.

[27] Yohei Yamamoto, Masatoshi Yoshikawa, and Shunsuke Umeura. 1999. On indices for xml documents with namespaces. In *Conference Proceedings of Markup Technologies*, Vol. 99. 127–135.

[28] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. 2013. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Data engineering (ICDE), 2013 ieee 29th international conference on*. IEEE, 565–576.