RDF Query Answering Using Apache Spark: Review and Assessment

Giannis Agathangelos[1], Georgia Troullinou[1], Haridimos Kondylakis[1], Kostas Stefanidis[2], Dimitris Plexousakis[1]
[1] ICS-FORTH, Greece {jagathan, troulin, kondylak, dp}@ics.forth.gr
[2] University of Tampere, Finland kostas.stefanidis@uta.fi

***Abstract*—The explosion of the web and the abundance of linked data demand for effective and efficient methods for storage, management and querying. More specifically, the ever-increasing size and number of RDF data collections raises the need for efficient query answering, and dictates the usage of distributed data management systems for effectively partitioning and querying them. To this direction, Apache Spark is one of the most active big-data approaches, with more and more systems adopting it, for efficient, distributed data management. The purpose of this paper is to provide an overview of the existing works dealing with efficient query answering, in the area of RDF data, using Apache Spark. We discuss on the characteristics and the key dimension of such systems, we describe novel ideas in the area, and the corresponding drawbacks, and provide directions for future work.**

I. Introduction

The prevalence of Open Linked Data, and the explosion of available information on the Web, have led to an enormous amount of widely available RDF datasets [6]. To store, manage and query these ever increasing RDF data, many systems have been developed by the research community and by commercial vendors. To this direction, distributed big data processing engines, like Hadoop, HBase and Impala [14], are exploited more and more for this purpose due to their ability to effectively handle mass amounts of data. Apache Spark, is one of the most active, big data approach with an ever increasing interest in using it for efficient query answering over RDF data. The platform uses in-memory data structures that can be used to store RDF data, offering increasing efficiency, and enabling effective distributed query answering.

As such, the goal of this work is to provide an overview of the works *dealing with efficient query answering, using Apache Spark, for RDF data*. Focusing on this specific field, we fill in the gap in the literature, providing a complete and detailed overview of the current research activities in the area. More specifically, our contributions are the following. Firstly, we present and discuss various dimensions of analysis, identifying key elements for such systems. Then, we classify the approaches according to the data model and the Apache Spark abstraction they use. We proceed further to perform an in depth overview of the approaches in each category, providing a unique perspective on the research in the area, and highlighting the novel ideas and the drawbacks of each one. Finally, we identify what is missing from the area and provide interesting directions for future work.

There are already surveys in the area of generic RDF storage [11] and on RDF data management systems in cloud environments [15]. However, distributed RDF query answering systems are beyond the scope of the former, as the authors claim in the first paper, whereas they both cover mainly works before the prevalence of Spark. As such, our work can be seen as complementary to the aforementioned surveys, shedding light to the area of RDF query answering, specifically, on works using Apache Spark as the underlying data management infrastructure. From a different perspective, [8] presents a preliminary experimental comparison, evaluating Spark implementations for RDF systems, focusing on techniques for distributing data. Specifically, the authors analyze five representative RDF data distribution approaches. The general goal is to examine the advantages of each distribution solution, and to identify the challenges of each approach when implemented in Spark. However, in our paper we do not attempt to do a comparative experimental evaluation to a limited number of approaches, but to identify and present an overview of the main research directions in the area.

The rest of this paper is structured as follows: In Section II, we present some background required when speaking about RDF data. Then, in Section III, we define the dimensions we use for describing the systems presented in Section IV. Finally, Section V concludes this paper, identifies gaps in the area and presents directions for future work.

II. Background

A. The Resource Description Framework (RDF)

The representation of knowledge in RDF is based on triples of the form of $(subject\ predicate\ object)$ which record that $subject$ is related to $object$ via $predicate$. Formally, representation of RDF data is based on three disjoint and infinite sets of resources, namely: URIs (U), literals (L) and blank nodes (B). RDF allows representing a form of incomplete information through blank nodes, standing for unknown constants or URIs. As such, a triple is a tuple $(subject\ predicate\ object)$ from $(U \cup B) \times U \times (U \cup L \cup B)$. In addition, to state that a resource r is of a type τ, the property $rdf{:}type$ is used.

RDF datasets have attached semantics through RDFS [1], a vocabulary description language. RDF Schema is a vocabulary description language that includes a set of inference rules used to generate new, implicit triples from explicit ones.

Finally, a collection of triples can be represented as a labeled directed graph, in which nodes represent subjects or objects and labeled directed edges represent predicates.

B. Querying

For querying RDF data, SPARQL is used. SPARQL [2] is currently the standard query language for the semantic web and has become an official W3C recommendation. Essentially, SPARQL is a graph-matching language. SPARQL queries contain a set of triples patterns, also called basic graph patterns. Triple patterns are like RDF triples that each of the subject, predicate and object may be a variable or a literal. Solutions to the variables are then found by matching the patterns in the query to triples in the dataset. Thus, SPARQL queries are pattern matching queries on triples, that compose an RDF data graph.

Specifically, a SPARQL query consists of three parts. The *pattern matching part*, which includes several features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering (or restricting) values of possible matchings. The *solution modifiers*, which once the output of the pattern has been computed (in the form of a table of values of variables), allows to modify these values applying classical operators, like projection, distinct, order, limit, and offset. Finally, the *output* of a SPARQL query can be of different types: yes/no answers, selections of values of the variables which match the patterns, construction of new triples from these values, and descriptions of resources.

According to the position of the variables in the triple patterns, a query can have different shapes that affect its performance. *Star-shaped* patterns/queries are characterized by subject-subject joins between triple patterns as the join variable is on the subject position. *Linear shaped* patterns/queries are made of subject-object (or object-subject) joins, for example, the join variable is on the object position in one triple pattern and on the subject position in the other. *Snowflake-shaped* patterns/queries are combinations of several *star-shaped* connections. Finally, more complex queries combine the above described patterns.

III. Evaluation Dimensions

Apache Spark [29] is an in-memory distributed computing platform designed for large-scale data processing. Spark was originally developed at UC Berkeley in 2009 and currently is one of the most active big-data Apache projects. It can be considered as a main-memory extension of the MapReduce model [10], since both of them enable parallel computations on comodity machines with locality-awareness scheduling, fault tolerance and load balancing. Because of Spark's main memory implementation, it can be up to 100 times faster than Hadoop. This level of efficiency is due to the two main data abstractions that Spark provides: RDDs (Resilient Distributed Dataset) and DataFrames. RDD was the primary user-facing API in Spark since its inception. At its core, an RDD is an immutable distributed collection of data elements, partitioned across nodes in a cluster that can be operated in parallel with a low-level API that offers transformations and actions. Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. By using DataFrames, Spark leverages this schema knowledge, and ends up in a much more efficient data encoding than java serialization.

On top of RDD and DataFrames, Spark proposes two higher-level data access models, GraphX and Spark SQL, for processing semi-structured data in general. Those data models can be used to handle RDF data and SPARQL queries. Spark GraphX [28] is a library enabling graph processing by extending the RDD abstraction and hence introduces a new feature called Resilient Distributed Graph or RDG. GraphX combines the benefits of graph-parallel and data-parallel systems, as it efficiently expresses graph computations within the framework of the data-parallel system. Spark SQL [3] is Spark's interface for working with structured and semi-structured data. It enables querying on data stored in DataFrames using SQL. It also provides an optimizer, Catalyst, which is claimed to improve the execution of queries.

As such, when studying the RDF processing approaches on Apache Spark, the key factors are: a) the data model that is selected in order to process the RDF data and b) the Spark data abstractions each work decided to rely the implementation on.

- **Data Model**: The model selected for the specific representation of the RDF data. It can be one of the following:
  a. The Triple Model. RDF data is stored and processed in their natural form, as triples that contain subject, predicate, object.
  b. The Graph Model. The RDF model is represented as a directed labeled graph in which, for example, the triple (*s hasProperty p*) can be interpreted as an edge labeled with *hasProperty* from node *s* to node *p*. This model is used mainly by systems that are built on top of the graph processing API of Spark.
- **Apache Spark Abstraction**: Spark provides various libraries and data abstractions each of them having several advantages and disadvantages.
  a. RDD. RDDs provide a low-level API that gives great control over the dataset. It lacks the schema control, but gives greater flexibility when it comes to storage and partition, as it gives the choice of implementing a custom partitioner.
  b. DataFrames. A Dataframe is an immutable distributed collection of data that is organized into named columns. Designed to make large datasets processing even easier, allowing developers to impose a structure onto a distributed collection of data.
  c. Spark SQL. It enables querying on structured data stored in DataFrames using SQL and provides an optimizer for improving execution times.
  d. GraphX. This is Spark's library for graph processing. By combining both graph-parallel and data-parallel processing, it can achieve great performance and flexibility. It also comes with well known graph processing algorithms, like pagerank, triangle counting and shortest paths computation.
  e. GraphFrames. This is the newest graph processing API that benefits from the scalability and high performance of DataFrames. In contrast with GraphX, it supports also queries over graphs. It is not yet an official part of Apache Spark, but comes as a side package.

Figure 1 summarizes the different dimensions based on which we study RDF query processing methods.

Besides the aforementioned dimensions for categorizing the works in the area, there are also a number of interesting

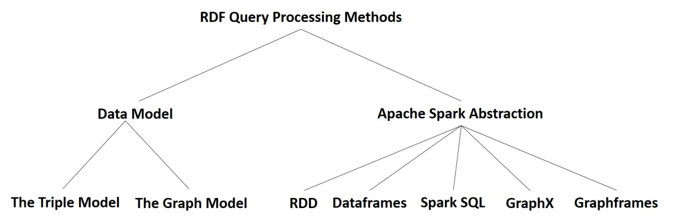

Fig. 1. A taxonomy presenting the dimensions for organizing RDF query processing methods.

dimensions according to which we could further study them. Specifically:

- **Query Processing**: This dimension identifies the procedure for translating a SPARQL query into a query compatible to the Spark format and how does the query get evaluated over the dataset. For example, a SPARQL query can be translated into SQL code, and execute this code using Spark SQL.
- **Query Processing Optimizations**: This dimension describes the optimization methods employed by a selected system. For example, a very common way for query optimization is to re-order the joins sequence based on data statistics.
- **Data Partitioning**: Choosing the right data partition strategy is essential in distributed systems. The goal is to maximize data locality and minimize network communication to achieve the desirable performance. Apache Spark uses by default a hash partitioning strategy, but this can be modified depending on the data abstraction that is used.
- **SPARQL Fragment**: SPARQL contains a huge set of operations and most of the systems do not provide full support for it. All systems start from evaluating simple blocks of triple patterns, called Basic Graph Patterns (BGP), and continue building on top of this, for more operations (BGP+), such as average (AVG) and filter operations (FILTER).
- **System Contribution**: The main focus of most of the systems is to improve query performance. Some systems focus on a particular query type, e.g., star queries, and others target at handling multiple or all query types.

IV. RDF Processing Approaches

In this section, we organize systems based on the way they model and process RDF data. Specifically, we distinguish between i) *triple processing systems* and ii) *graph processing systems*. In triple processing systems, data is loaded as triples and their raw form is used for further processing. Usually in such systems, a simple partitioning technique, like hash or vertical partitioning, is preferred whereas for the evaluation of the issued SPARQL queries, the RDD API or Spark SQL is used. In graph processing systems, RDF data is modeled as graphs, and queries are evaluated directly over them. Either GraphX or GraphFrames is used for query processing.

A. Triple Processing Systems

1) RDD Implementation: HAQWA [7] was the first approach that tries to process RDF data on Apache Spark. It proposes a trade-off between data distribution complexity and query answering efficiency. System's fragmentation and allocation is a two-step procedure. In the first step, a hash-based partitioning is performed on triple subjects. This fragmentation ensures that star-shaped queries are performed locally, but no guarantees are provided for other query types. In the second step, data are allocated according to the analysis of frequent queries executed over the dataset. Then, at query time, the system decomposes a query pattern into a set of local sub-queries that can be evaluated locally. Each of those sub-queries is a candidate to be the starting point (seed query) to evaluate the entire query pattern. To prevent network communication, the missing triples are replicated into the partitions that contain the triples of the seed. To do so, for each candidate and partition, HAQWA computes the cost of transferring missing triples into the current partition. HAQWA performs an encoding of string values to integer ones on data, which minimizes data volume and makes processing more efficient. Query processing is based on a mapping from SPARQL to RDDs API, like join, filter and count.

In SPARQLGX [13], RDF datasets are vertically partitioned. As such, a triple *(s p o)* is stored in a file named *p* whose content keeps only *s* and *o* entries. By following this approach, the memory footprint is reduced and the response time is minimized when queries have bounded predicates. The query translation is done by parsing one by one the triple patterns and mapping them to Spark's RDD API. In order to deal with a group of triple patterns, the result of each sub-query is joined with the next one having a common variable with it, using this common variable as a key (keyBy in Spark). If no common variable is found, between two triple patterns, then the cross procuct is computed. SPARQLGX is able to evaluate *Basic Graph Pattern* (BGP) queries and also operations like DISTINCT, SORT, UNION, OPTIONAL and FILTER. As an optimization, statistics on data are computed in order to reorder the join execution of each query. More specifically, the system counts all distinct subjects, predicates and objects of the given dataset.

2) Spark SQL: S2RDF [24] is an efficient and scalable system on top of Spark that aims to provide improvements for all query types. This work presents an extended version of the classic vertical partitioning technique, called ExtVP. Each ExtVP table is a set of sub-tables corresponding to a vertical partition (VP) table. The sub-tables are generated by using right outer joins between VP tables. More specifically, [24] pre-computes semi-join reductions for subject-subject (SS), object-subject (OS) and subject-object (SO). For SPARQL query execution, the triples are joined via shared variables. For example, for the triple patterns *?x likes ?y* and *?x follows ?z* the *?x* variable is used for joining them. Assuming that there are two tables containing 100 entries each, having only 10 entries in the same subject, we need 10,000 comparisons to join them. If we store data using ExtVP, only 10 comparisons are needed and as such, the efficiency of the query is enhanced. For query processing, S2RDF uses Jena ARQ to tranform a SPARQL query to an algebra tree and then it traverses this tree to produce a Spark SQL query. To reduce the storage overhead of the extra sub-tables a selectivity factor (SF) is being used. This SF defines the relative size of ExtVP of a table compared to the corresponding VP table size. So, S2RDF supports the definition of a threshold for SF such that all ExtVP tables above this threshold are not considered. As a query optimization, an algorithm that reorders sub-query execution based on the table size and the number of bounded variables is used. Sub-queries with the most bounded variables are executed first, and for those with same number of bounded variables the one that corresponds to the smallest table size is picked. S2RDF support SPARQL BGPs and also operations, like FILTER, UNION, OFFSET, LIMIT and ORDER BY.

3) Hybrid Approaches: [21] studies two distributed join algorithms, partitioned join and broadcast join, for the evaluation of BGP expressions on top of Apache Spark. In this work, we see what kind of join algorithm each data abstraction of Apache Spark uses, and how we can combine them to achieve better performance. For the purpose of this study, data is partitioned according to the value of the subject. For every data abstraction of Spark, the authors implement a translation from SPARQL to the corresponding API, in order to execute queries on RDF by exploiting the framework.

Spark SQL uses the embedded Catalyst optimizer to generate the execution plan of the query, using the Spark DataFrame and the broadcast join algorithm. A significant drawback of this approach is that when one query has more than one triple patterns (which is often the case) a Cartesian product is being used instead of a join, which is inefficient. The RDD approach translates each join into a partitioned join operator, following the order specified by the input logical query. This ends up with a sequence of (possibly n-ary) joins on different variables. This approach lacks efficiency when a broadcast join is cheaper e.g., join a small with a large data set. It is worth mentioning that RDDs always reads the entire data set for each triple pattern. DataFrames provide an important benefit which comes from the columnar compressed in-memory representation that is used. Up to 10 times larger data sets than RDD can be managed. It uses a cost-based join optimization approach by preferring a single broadcast join to a sequence of partitioned join if the dataset is smaller than a given threshold. For example, in the case of joining several small datasets with a large one this approach is more efficient. In cases that join expressions that are highly selective filtering over a large dataset, this approach will not use the most efficient join because it takes into account only the size. Also this approach does not consider data partitioning.

Trying to overcome the limitations of the previous approaches, [21] offers a hybrid strategy that combines broadcast joins with partitioned joins. More specifically, it takes into account an existing data partitioning scheme to avoid useless data transfer and use data compression from DataFrames to reduce the data access cost of self-join operations. The most efficient query plan for the combination of the two join algorithms is generated by a dynamic greedy optimization algorithm based on data statistics.

B. Graph Processing

1) GraphX: S2X [23] is a work that combines the graph-parallel abstraction of GraphX with the data-parallel computation of Spark to evaluate SPARQL queries in a distributed manner. GraphX is used to implement the graph pattern matching part of SPARQL and the data-parallel computaion of Spark to implement other SPARQL operators.

RDF data is being modeled as a property graph (for more in property graphs, see [26]). In a property graph each vertex has an ID and properties and edges have a property and two IDs of the corresponding vertices. Edge property stores the predicate URI. Vertex properties are used to store subject and object $URIs$, and a data structure for candidate query variables that could match this vertex. The basic idea of the proposed algorithm is that every vertex in the graph stores the variables of a query where it is a possible candidate for. The first step is to match all triple patterns of a BGP independently, and then exchange messages between adjacent vertices to validate the match candidates until they do not change anymore. The set of matches for each vertex is called local match and the matched sets of adjacent vertices are called remote matches.

More specifically, all possibly relevant vertices are determined by matching each edge with all triple patterns from the BGP. Match candidates are validated according to some validation rules, using local and remote match sets and invalid ones get discarded. Locally changed match sets are sent to their neighbors in the graph for validation in the next step. The same process is repeated until no changes occur. The final output is composed of the individual subgraphs of the previous steps. S2X can evaluate also SPARQL opertaros, like OPTIONAL, FILTER, ORDER BY, PROJECTION, LIMIT and OFFSET. These operators are implemented with the use of Spark API.

[16] introduces an approach that is based on subgraph matching on GraphX. Here, each vertex is assigned three properties: 1) a label that keeps the value of its corresponding subject or object, 2) a Match_Track table (M_T) that contains variables and constants, and 3) a flag that shows if a vertex is located at the end of a path (sequence of matched BGP triples). Edges have a property called edge label that keeps the predicate value.

The proposed algorithm iterates through all BGP triples of a SPARQL query. Graph matching is being implemented with the use of AggregateMessages operator of GraphX that provides two functions, sendMsg and a mergeMsg. SendMsg can be considered as a map function that matches the current BGP triple with all graph triples. If a match is found, the sendMsg prepares and sends different messages to the source and destination vertex of the triple. Then, using the mergeMsg as a reduce function, the received messages are aggregated at their destination vertex. At the last step in each iteration, the joinVertices function is used to evaluate the old property values and the new values in each vertex. After evaluating all BGP triples, we join the final M_T tables of the end vertices, which contain partial results, to generate the final query answer.

Spar(k)ql [12] also targets at evaluating SPARQL queries over GraphX. RDF data is modeled as a property graph. The node model adopted is pretty simple, object properties are the edges of the graph and data properties are stored in the nodes of the graph as node properties. An exception to this is the rdf:*type* property. Although, it is an object property, due to its popularity in SPARQL queries, it is stored in the node properties along with the data properties.

In order to implement the query answering via vertex programs, there is a need to store sub-results in tables in each node. The main idea is that each node get messages from its neighboors and calculates the sub-results based on the incoming messages and the stored information. For this reason, it performs a Map phase with the query variables as keys, and data tables as values, that contain possible sub-results. Furthermore, Spar(k)ql provides a message model that allows all edges to be active until they get processed, so that all type of queries will be able to be executed. A query plan is generated by exploiting a breadth-first search algorithm that uses object properties to create a tree. During the execution, the query plan is traversed bottom-up and, for each node, it iterates through the edges to find the corresponding matches.

2) GraphFrames: [4] is the first work that implements an efficient processing technique for RDF data over the GraphFrames API [9]. It is a new graph processing platform created over Apache Spark, using the concept of DataFrames. In this approach, the input dataset splits into two separate lists, a nodelist and an edgelist, which are used to generate the unweighted labeled graph. SPARQL queries are translated into query graphs which are then being optimized to improve performance. To determine an optimal order of the query, the algorithm takes into account the predicate frequency, and sorts sub-queries in non-descending order. In the next step, another optimization takes place called local search space pruning. In this procedure, for each query all triples in the dataset that do not match BGPs predicates get discarded. This technique results in a new graph created from this temporary dataset, which has a much smaller search space. Finally, query processing takes the optimized query and the locally pruned RDF data-graph, and performs subgraph matching to get the final query answer.

3) Hybrid Approaches: SparkRDF [5] is an elastic graph processing engine that is scalable, efficient, reduces I/O and intermediate communication and is built on top of Spark, without the use of a graph processing API. The SparkRDF approach presents a novel storage scheme for managing big RDF graphs in HDFS, an iterative graph model for processing SPARQL queries distributively and in-memory. Several optimization techniques were proposed, including an optimal query plan and a dynamic partitioning method.

Multi-layer Elastic Sub-graph (MESG) is the storage model created for this work. MESG consists of three level of indexes. In the first level, there is a class index and a relation index. The relation index is for triples that do not have an $rdf{:}type$ predicate and the class index is for those triples that have. Relation files are stored by predicate name and class files are stored by the name of object. In the second level, MESG uses more information for indexing than only the predicate. It divides predicate files according to the type of subjects and objects. So, there are CR (class-relation) and RC (relation-class) indexes. In the third level, it goes one step further and creates an index that combines every part of the triple. CRC (class-relation-class) index uses subjects class, predicate and objects class in order to exploit all the information that may be available for a triple.

The Memory Data Model, RDSG (Resilient Discreted Semantic SubGraph) is a distributed memory abstraction that enables in-memory query computations on large clusters. This model provides basic operations, like RDSG generation, filter,

TABLE I. A TAXONOMY OF THE RDF QUERY PROCESSING APPROACHES WITH RESPECT TO DATA MODEL AND APACHE SPARK ABSTRACTION.

		Data Model	
Apache Spark Abstraction		The Triple Model	The Graph Model
	RDD	[7], [13],[21]	[5]
	DataFrames	[21]	
	Spark SQL	[24]	
	GraphX		[23], [16], [12]
	GraphFrames		[4]

prepartition and join. These operations are based on the Spark API and are used to implement system's query processing.

Regarding query processing, every query is decomposed into an ordered sequence of variables and every query variable is made up of several triple patterns. For example, for the variable X, the authors compute the matches for each triple pattern for this variable, and the matched triples are used to find the matches on the next triple pattern that X exists, by joining them on the shared variable X. After this procedure finishes, the process continues with the next variable.

As query optimizations, variable's class is passed through a message to the corresponding triple patterns containing the variable. By following this method, the authors avoid reading many unnecessary data, and $rdf{:}type$ triple patterns can be removed. On-demand, dynamic pre-partitioning is applied to reduce the shuffling cost in the distributed join process. Specifically, this process pre-partitions the MESG only when it is on-demand loaded into the distributed memory. This pre-partitioning scheme guarantees that the records sharing the same variable value will be read into the same partition. Finally, an optimal query plan is generated that first determines the joining order of variables and then the order of triple patterns in a job.

V. DISCUSSION & CONCLUSIONS

In short, we can categorize the RDF query processing approaches on Apache Spark based on the following dimensions: how the data is modeled in order to process them (data model), and which is the Spark API that is used for the implementation of the approach (Spark Abstraction). RDF data is stored and processed in their natural form, as triples, or are represented as a directed labeled graph. RDDs, DataFrames, Spark SQL, GraphX and GraphFrames are the APIs provided by Spark. RDDs give great flexibility regarding storage and partitioning, while DataFrames offer an immutable distributed collection of data organized into named columns. When data is stored in DataFrames, Spark SQL can be used for optimized query processing. GraphX supports graph-parallel and data-parallel data processing, while, GraphFrames, in addition, support queries over graphs. Table I summarizes the various options in each dimension. Overall, the graph representation model is the one that is used mainly by systems that are built on top of the graph processing API of Spark.

Generally speaking, we observe that there is a trend around using Apache Spark when targeting at efficient RDF query processing approaches. Table II provides some additional characteristics of those approaches. The general goal of all those approaches is to improve query performance by exploiting data parallelization. However, to this purpose they neglect that data partitioning is a key element of efficient query processing

TABLE II. ADDITIONAL CHARACTERISTICS OF THE RDF QUERY PROCESSING APPROACHES.

System	Query Processing	Optimization	Partitioning	SPARQL
[7]	RDD API	No	Hash / Query Aware	BGP+
[13]	RDD API	Yes	Vertical	BGP+
[24]	Spark SQL	Yes	Extended Vertical	BGP+
[21]	Hybrid	Yes	Hash-sbj	BGP
[23]	Graph Iterations	No	Default	BGP+
[16]	Graph Iterations	Yes	Default	BGP
[12]	Graph Iterations	Yes	Default	BGP
[4]	Subgraph Matching	Yes	Default	BGP
[5]	Custom	Yes	Hash-sbj	BGP

that has a huge impact in query answering. As such they end up using simple partitioning techniques like vertical or hash partitioning. Although, some recent works have already started to recognize the importance of data partitioning (e.g. [24] that presents a sophisticated partitioning technique based on the classical vertical partitioning method or [27]), we argue that data partitioning is an essential part of efficient query processing and that further research is required in the area.

To this direction, exploiting knowledge about the queries previously submitted in a system, we can end up in a more efficient partitioning scheme. The goal of such a scheme would be to handle efficiently the query types that are mostly submitted to the system improving overall the efficiency of the system. [7] proposes a partitioning procedure towards this direction. Specifically, it exploits particular knowledge regarding the input queries in order to ensure data locality in frequent queries. Graph partitioning does not focus on load balancing rather than on minimizing the edge-cut between partitions. GraphX has not been exploited yet towards this direction and could be an option to build such algorithms, as it offers already an extensive amount of graph algorithms.

In a different direction, dynamicity is an important aspect of the RDF data, which are constantly evolving, typically without any warning, centralized monitoring, or reliable notification mechanism [18], [19], [17]. This raises the need to keep track of the different versions of the data, so as to be able to have access not only to the latest version, but also to previous ones [25], [22], [20]. As such an essential aspect, the next generation parallel RDF query answering systems should be able to handle evolving data in an uninterrupted manner.

ACKNOWLEDGMENT

This work was partially supported by the projects iManage-Cancer (H2020-643529), Bounce (H2020-777167) and through IKY scholarships programme and co-financed by the European Union and Greek national funds through the action entitled Reinforcement of Postdoctoral Researchers, in the framework of the Operational Programme Human Resources Development Program, Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) 20142020.

REFERENCES

[1] RDF Schema 1.1. Available online: http://www.w3.org/TR/rdf-schema/, (last accessed October 2017)

[2] SPARQL Query Language for RDF. Available online: https://www.w3.org/TR/rdf-sparql-query/

[3] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: SIGMOD (2015)

[4] Bahrami, R.A., Gulati, J., Abulaish, M.: Efficient processing of SPARQL queries over graphframes. In: WI, pp. 678–685. ACM (2017)

[5] Chen, X., Chen, H., Zhang, N., Zhang, S.: Sparkrdf: Elastic discreted RDF graph processing engine with distributed memory. In: WI-IAT (1), pp. 292–300. IEEE Computer Society (2015)

[6] Christophides, V., Efthymiou, V., Stefanidis, K.: Entity Resolution in the Web of Data. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers (2015)

[7] Curé, O., Naacke, H., Baazizi, M.A., Amann, B.: HAQWA: a hash-based and query workload aware distributed RDF store. In: International Semantic Web Conference (Posters & Demos), *CEUR Workshop Proceedings*, vol. 1486. CEUR-WS.org (2015)

[8] Curé, O., Naacke, H., Baazizi, M.A., Amann, B.: On the evaluation of RDF distribution algorithms implemented over apache spark. In: SSWS@ISWC (2015)

[9] Dave, A., Jindal, A., Li, L.E., Xin, R., Gonzalez, J., Zaharia, M.: Graphframes: an integrated API for mixing graph and relational queries. In: International Workshop on Graph Data Management Experiences and Systems, p. 2 (2016)

[10] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

[11] Faye, Cure, O., Blin: A survey of RDF storage approaches. ARIMA Journal **15**, 11–35 (2012)

[12] Gombos, G., Rácz, G., Kiss, A.: Spar(k)ql: SPARQL evaluation method on spark graphx. In: FiCloud Workshops (2016)

[13] Graux, D., Jachiet, L., Genevès, P., Layaïda, N.: SPARQLGX: efficient distributed evaluation of SPARQL with apache spark. In: ISWC (2016)

[14] Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL querying of large RDF graphs. PVLDB **4**(11), 1123–1134 (2011)

[15] Kaoudi, Z., Manolescu, I.: RDF in the clouds: a survey. VLDB J. **24**(1), 67–91 (2015)

[16] Kassaie, B.: SPARQL over graphx. CoRR **abs/1701.03091** (2017)

[17] Kondylakis, H., Despoina, M., et al., G.G.: Evordf: A framework for exploring ontology evolution. In: ESWC, pp. 104–108 (2017)

[18] Kondylakis, H., Plexousakis, D.: Ontology evolution in data integration: Query rewriting to the rescue. In: ER, pp. 393–401 (2011)

[19] Kondylakis, H., Plexousakis, D.: Ontology evolution: Assisting query migration. In: ER, pp. 331–344 (2012)

[20] Kondylakis, H., Plexousakis, D.: Ontology evolution without tears. J. Web Sem. **19**, 42–58 (2013)

[21] Naacke, H., Amann, B., Curé, O.: SPARQL graph pattern processing with apache spark. In: GRADES@SIGMOD/PODS, pp. 1:1–1:7. ACM (2017)

[22] Papakonstantinou, V., Flouris, G., Fundulaki, I., Stefanidis, K., Roussakis, Y.: Spbv: Benchmarking linked data archiving systems. In: BLINK2017 (2017)

[23] Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., Lausen, G.: S2X: graph-parallel querying of RDF with graphx. In: Big-O(Q)/DMAH (2015)

[24] Schätzle, A., Przyjaciel-Zablocki, M., Skilevic, S., Lausen, G.: S2RDF: RDF querying with SPARQL on spark. PVLDB **9**(10), 804–815 (2016)

[25] Stefanidis, K., Chrysakis, I., Flouris, G.: On designing archiving policies for evolving RDF datasets on the web. In: ER (2014)

[26] Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., Xie, G.T.: Sqlgraph: An efficient relational-based property graph store. In: SIGMOD, pp. 1887–1901 (2015)

[27] Troullinou, G., Kondylakis, H., Plexousakis, D.: Semantic partitioning for rdf datasets. Information Search, Integration, and Personlization. Communications in Computer and Information Science **760**, 11–35 (2017)

[28] Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: a resilient distributed graph system on spark. In: GRADES (2013)

[29] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: HotCloud (2010)