CliqueSquare: efficient Hadoop-based RDF query processing

François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge Quiané-Ruiz, Stamatis Zampetakis

CliqueSquare: efficient Hadoop-based RDF query processing

François Goasdoué[1,2] Zoi Kaoudi[2,1] Ioana Manolescu[2,1]
Jorge Quiané-Ruiz[3] Stamatis Zampetakis[2,1]
[1] Université de Paris-Sud, France firstname.lastname@lri.fr
[2] Inria Saclay–Île-de-France, France firstname.lastname@inria.fr
[3] Qatar Computing Research Institute, Qatar jquianeruiz@qf.org.qa

Abstract

Large volumes of RDF data collections are being created, published and used lately in various contexts, from scientific data to domain ontologies and to open government data, in particular in the context of the Linked Data movement. Managing such large volumes of RDF data is challenging due to the sheer size and the heterogeneity. To tackle the size challenge, a single isolated machine is not an efficient solution anymore. The MapReduce paradigm is a promising direction providing scalability and massively parallel processing of large-volume data.

We present CliqueSquare, an efficient RDF data management platform based on Hadoop, an open source MapReduce implementation, and its file system, Hadoop Distributed File System (HDFS). CliqueSquare relies on a novel RDF data partitioning scheme enabling queries to be evaluated efficiently, by minimizing both the number of MapReduce jobs and the data transfer between nodes during query execution. We present preliminary experiments comparing our system against HadoopRDF, the state-of-the-art Hadoop-based RDF platform. The results demonstrate the advantages of CliqueSquare not only in terms of query response times, but also in terms of network traffic.

Résumé

De grands volumes de données RDF sont créés, publiés et utilisés dans de nombreux contextes, allant des données scientifiques aux ontologies de domaine, en passant par les données ouvertes notamment avec l'essor des données liées. Gérer de telles données RDF est un challenge de par leur volume et leur hétérogénéité. En particulier, les solutions centralisées ne font plus face à la masse des données. Le paradigme MapReduce, offrant des traitements massivement parallèles à fort potentiel de passage à l'échelle, semble une voie prometteuse pour manipuler ces nouveaux ordres de grandeur de données.

Dans cet article, nous présentons CliqueSquare, une plateforme efficace de gestion de données RDF fondée sur Hadoop, une implémentation open-source de MapReduce, et son système de fichiers, Hadoop Distributed File System (HDFS), pour stocker et traiter de grands volumes de données. Nous proposons une méthode de partitionnement efficace des données RDF réduisant les transferts de données lors de l'évaluation des requêtes, ainsi qu'un algorithme fondé sur des cliques pour produire des plans de requêtes, minimiser le nombre d'étapes MapReduce, et exploiter notre schéma de partitionnement des

données. Enfin, nous présentons des résultats préliminaires en comparant notre système avec HadoopRDF, la référence de la littérature pour les solutions de stockage et interrogation de données RDF fondées sur Hadoop. Nous montrons notamment la supériorité de CliqueSquare en termes de temps de réponse et de trafic réseau.

Mots-clefs: RDF, MapReduce, Hadoop, query optimization

1 Introduction

The Resource Description Framework (RDF) [13] has been designed as a flexible data representation for the Semantic Web. In the recent years, the RDF data model has gained a lot of attention from both industry and academia. This is mainly because the RDF data model is quite general to express any type of data. As a result, a huge number of current applications use RDF as a first-class citizen or provide support for RDF data. These applications range from the Semantic Web [3, 31] and scientific applications [35, 38] to Web 2.0 platforms [17, 37] and databases [6].

Given the proliferation of the RDF data model, large volumes of RDF data collections are being created and published, in particular in the context of the Linked Data movement. Although the RDF data model is general and flexible, it can result in serious performance issues. This is because RDF queries are mainly composed of a set of joins over the RDF dataset. This issue becomes bigger and bigger as the proliferation of RDF-based applications continues and hence as the amount of RDF data increases.

Therefore, efficient and scalable management of RDF data is at the core of many applications. Several research efforts have been made in the context of RDF data management, resulting in different RDF engines for storing, indexing, and querying [1, 22, 34]. However, despite all these research efforts, efficiently processing big RDF datasets is still an open problem. The main challenge in managing big RDF datasets mainly resides in the sheer size of the data itself. Indeed, to tackle this size challenge, a single isolated machine is not an efficient solution anymore. Therefore, several researchers have proposed distributed systems, especially MapReduce-based, for RDF data management [16, 27, 28]. However, all these works still have to transfer a considerable amount of data through the network, which has a negative impact in query performance.

In this paper, we focus on providing an efficient and scalable approach for RDF data management. We propose CliqueSquare, an efficient Hadoop-based RDF data management platform for storing and processing big RDF datasets. In summary, we make the following main contributions:

(1) We propose an RDF data partitioning method that aims at reducing the amount of data to be transferred through the network at query processing time. For this, CliqueSquare exploits the existing data replication (three replicas by default) in the Hadoop Distributed File System (HDFS) to partition the RDF dataset based on the subject, the property, and the object of each triple in the RDF dataset.

(2) We propose a clique-based algorithm for query processing, which produces query plans that minimize the number of MapReduce stages. For this, CliqueSquare exploits the way it partitions RDF datasets. This allows CliqueSquare to perform most common types of RDF queries locally at each node, minimizing the data transfer through the network.

(3) We perform a series of experiments using our first CliqueSquare prototype and com-

pare it with HadoopRDF [16] (the state-of-the-art Hadoop-based framework for storing and querying RDF data). The results show the high superiority of CliqueSquare over HadoopRDF in terms of both job execution times and network traffic. CliqueSquare improves HadoopRDF for more than one order of magnitude (it is up to 67 faster in terms query execution times and up to 91 faster in terms of data transfers). In addition, we conducted a statistical study on real-world queries. The results show that CliqueSquare can answer more than 99% of the queries in one MapReduce job.

The remainder of this paper is structured as follows. We first give some background necessary for this paper in Section 2. We then present our RDF data partitioning approach in Section 3 and give the query model used by CliqueSquare in Section 4. In Section 5, we present the query processing techniques used by CliqueSquare to perform SPARQL queries in Hadoop MapReduce. In Section 6, we give the experimental results of CliqueSquare. Finally, we present related work in Section 7 and conclude in Section 8.

2 Background

In this section, we introduce the RDF data model and its SPARQL query language, as well as the Hadoop infrastructure on which we base our platform.

2.1 RDF

The Resource Description Framework (RDF) [19] is a graph-based data model recommended by W3C for publishing (linked) Web data on the Semantic Web.

RDF is based on the concept of *resource* which is everything that can be referred to through a Uniform Resource Identifier (URI). In particular, RDF builds on *triples* to relate URIs to others URIs, to constants called *literals* or to unknown values called *blank nodes* (which are similar to the notion of labelled nulls in incomplete databases). A *triple* is a statement $(s\ p\ o)$ meaning that the *subject* s is described using the *property* p (a.k.a. *predicate*) by having the *object* value o. Formally, given U, L and B denoting three (pairwise disjoint) sets of URIs, literals, and blank nodes respectively, a well-formed *triple* is a tuple $(s\ p\ o)$ from $(U \cup B) \times U \times (U \cup L \cup B)$. In the following, we only consider well-formed triples.

A set of triples is an RDF *graph*, in which every triple $(s\ p\ o)$ corresponds to a directed edge labelled with p from the node labelled with s to the node labelled with o.

Notation. The notion of *namespaces* are often used for writing URIs in a compact way. A namespace is a term mapped to a URI which serves as a prefix to build other URIs. For instance, the namespace `rdf` is usually associated to the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#` to conveniently refer to the URIs of RDF built-in terms like the `rdf:Literal` class of literals or the `rdf:type` property for typing resources. In the following, we omit the namespaces whenever they are not relevant to the discussion and denote a literal by a string between quotes. Figure 1 shows a set of triples following these notations, whose graphical representation is given in Figure 2.

2.2 SPARQL

SPARQL [25] is the W3C standard for querying RDF graphs. In this paper, we consider the Basic Graph Pattern (BGP) queries of SPARQL, i.e., its conjunctive fragment allowing to express the core Select-Project-Join database queries. In such queries, the notion of triple is

```
:stud1 :takesCourse :db   :stud1 :member :dept4
:stud2 :takesCourse :os   :stud2 :member :dept1
:prof1 :advisor :stud1    :prof2 :advisor :stud2
:prof1 :name "bob"        :prof2 :name "alice"
:stud1 :name "ted"        :dept1 rdf:type :Dept
```

Figure 1: Sample RDF data set.

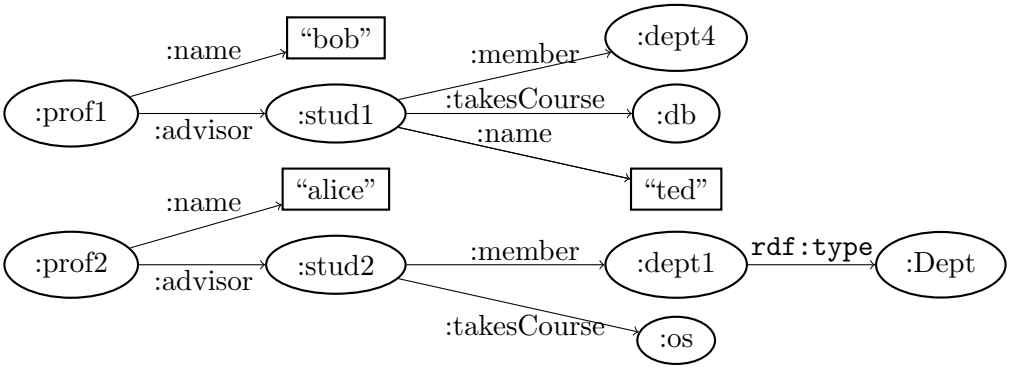


Figure 2: Graph representation of the example RDF data.

generalized to that of *triple pattern* $(s\ p\ o)$ from $(U \cup B \cup V) \times (U \cup V) \times (U \cup L \cup B \cup V)$, where V is a set of variables. The normative syntax of BGP queries is

$$\texttt{SELECT}\ ?v_1 \cdots ?v_m\ \texttt{WHERE}\ \{t_1 \cdots t_n\}$$

where $t_1, \ldots, t_n$ are triple patterns and $?v_1, \ldots, ?v_m$ are *distinguished* variables occurring in $\{t_1 \cdots t_n\}$ which define the output of the query. Observe that repeating a variable among triple patterns is the way of expressing joins. In the following, we assume BGP queries which do not contain cartesian products.

The *evaluation* of a query q, defined as $\texttt{SELECT}\ ?v_1 \cdots ?v_m\ \texttt{WHERE}\ \{t_1 \cdots t_n\}$, on an RDF graph G is: $eval(q) = \{\mu(?v_1 \cdots ?v_m) \mid \mu\colon varbl(q) \to val(G)$ is a function s.t. $\{\mu(t_1), \cdots, \mu(t_n)\} \subseteq G\}$, with $varbl(q)$ the set of variables and blank nodes occurring in q, $val(G)$ the set of URIs, literals and blank nodes occurring in G, and μ a function replacing any variable or blank node of q with its image in $val(G)$. By a slight abuse of notation, we denote by $\mu(t_i)$ the triple obtained by replacing the variables or blank nodes of the triple pattern t_i according to μ.

Observe that blank nodes do not play any particular role in queries, since (normative) query evaluation treats them as non-distinguished variables.

2.3 Hadoop

Hadoop[1] is a framework designed for data-intensive distributed applications, which mainly consists of the *Hadoop Distributed File System* (HDFS) and the Hadoop *MapReduce engine*. It provides the most popular open-source implementations of the Google File System [10] and Google MapReduce engine [8].

HDFS has been designed to store very large files in a distributed and robust fashion. In particular, it stores data in blocks of constant size (64 MB by default) which are replicated within the system (3 times by default).

[1] `http://hadoop.apache.org`

Data stored in HDFS can then be processed by the MapReduce engine through *jobs*. Each MapReduce job is typically a sequence of *map*, *shuffle*, and *reduce* phases. Data from HDFS to be processed is first chunked so as to be consumed in parallel by mapper nodes. These nodes extract data units from the chunks (words, tuples, triples, etc. depending on the application) to build and then shuffle key-value pairs. Any two pairs with a same key are routed to the same reducer node, usually through a hashing mechanism. In turn, reducer nodes consume the shuffled key-value pairs in parallel by grouping them based on their keys. Then, the reducer nodes compute the final key-value pairs, which are written in HDFS as part of the result of the distributed MapReduce job.

3 CliqueSquare storage

This section describes how CliqueSquare partitions and places RDF data in HDFS. We start from the observation that the performance of MapReduce jobs suffers from shuffling large amounts of intermediate data between the map and reduce phases. Therefore, our goal is to partition and place RDF data so that the largest number of joins are evaluated at the map phase itself. This kind of joins are known as *co-located* or *partitioned* joins [26, 23]. In the context of RDF, SPARQL queries involve various kind of joins, e.g., subject-subject, subject-object, or property-object joins. Co-locating such joins as much as possible is therefore an important step towards efficient query processing.

3.1 RDF partitioning

By default HDFS replicates each dataset three times for fault-tolerance reasons. CliqueSquare exploits this data replication to partition and store RDF data in three different ways. In detail, it proceeds as follows.

First, CliqueSquare partitions triples based on their subject, property, and object values. Given a value x occuring as a subject in at least one triple, we call *subject partition* of x the set of all triples having the subject value x. Similarly, we define the *property partitions* and *object partitions*. Like HDFS, CliqueSquare stores each triple three times. But, in contrast to HDFS, CliqueSquare stores one replica partitioned on the subject, one on the property, and another on the object.

Second, CliqueSquare stores all subject, property, and object partitions of the same value within the same node. Thus, for a given value x, the subject, property, and object partitions of x (if they exist) are stored on the same node. This placement of RDF triples allows CliqueSquare to perform as many joins as possible locally.

Finally, CliqueSquare groups all the subject partitions within a node by the value of the property in their triples. Similarly, it groups all object partitions based on their property values. Property-based partitioning has been first advocated in [16] and also resembles the vertical RDF partitioning proposed in [1] for centralized RDF stores. Then, CliqueSquare stores each resulting partition into an HDFS file, which we term *local property-based file*.

CliqueSquare reserves a special treatment to triples where the property is `rdf:type`. In many RDF datasets, such statements are very frequent which, in our context, translates into an unwieldy large property partition corresponding to the value `rdf:type`. To avoid the performance problems this may entail, CliqueSquare splits the property partition of `rdf:type` into several smaller partitions, according to their object value. This enables working with finer-granularity partitions.

3.2 MapReduce partitioning process

CliqueSquare partitions RDF data in parallel for performance reasons. For this, it leverages the MapReduce framework and partitions input RDF data using a single MapReduce job. We describe the *map*, *shuffle*, and *reduce* phases of this job in the following.

Map phase. For each input triple (s_1 p_1 o_1), the map function outputs three *key-value* pairs. CliqueSquare uses the triple itself as value of each output *key-value* pair and creates composite keys based on the subject, property, and object values. The first part of the composite key is used for routing the triples to the reducers, while the second part is used for grouping them in the property-based files. In specific, CliqueSquare computes the three keys as follows: one key is composed of the subject and property values (i.e., $s_1|p_1$); one key is composed of the object and property values (i.e., $o_1|p_1$); and one key is composed of the property value itself (i.e., p_1), but, if p_1 is `rdf:type`, CliqueSquare then concatenates the object value to this key (i.e., `rdf:type`$|o_1$).

Shuffle phase. CliqueSquare uses a customized partitioning function to shuffle the *key-value* pairs to reduce tasks based on the first part of the composite key. The reduce task (node) to which a key-value pair is routed is determined by hashing this part of the key. As a result, CliqueSquare sends any two triples having the same value x (as a subject, property, or object, irrespectively of where x appears in each of these two triples) to the same reduce task. Then, all triples belonging to the same reduce task are grouped by the second part of the composite key (the property value).

Reduce phase. The MapReduce framework then invokes the reduce function for each computed group of triples. The reduce function, in turn, stores each of these groups into a HDFS file (with a replication factor of one), whose file name is derived from the property value and a string token indicating if it is a subject, property, or object partition.

Algorithm 1 shows the pseudocode of the MapReduce job for partitioning data as explained above. Notice that since the property is included in the output key of the map function, we omit it from the value, in order to reduce the data transferred in the network and the data we store in HDFS.

Let us now illustrate our RDF partitioning approach (Algorithm `RDFPartitioner`) on the sample RDF graph of Figure 2 and a three-nodes cluster. Figure 3 shows the result after the routing of the shuffle phase. We underline the first part of the composite key used in the customized partitioning function. For example, the input triple (`:stud1 :takesCourse :db`) is sent: to node n_1 because of its subject value; to n_2 because of its object value; and to n_3 because of its property value. Next, each node groups the received triples based on the property part of the composite keys. Figure 4 shows the final result of the partitioning process assuming that the number of reduce tasks is equal to the number of nodes.

The advantage of our storage scheme is twofold. First, as many as possible joins can be performed locally during query evaluation. This is an important feature of our storage scheme as it reduces data shuffling during query processing and hence leads to improved query response times. Second, our approach strikes a good compromise between the generation of either too few or too many files. Indeed, one could have grouped all triples within a node (e.g., all triples on n_1 in Figure 4) into a single file. However, such files would have grown quite big and hence increase query response times. In contrast, the files stored by CliqueSquare have meaningful names, which can be efficiently exploited to load only the data relevant to any incoming query. Another alternative would be to omit the grouping by property values and create a separate file for each subject/property/object partition within a node. For instance, in our example, node n_2 has nine subject/property/object values (see underlined values in Figure 3) while only six files are located in this node (Figure 4). However, handling many small files would lead to a significant overhead within MapReduce jobs.

Algorithm 1: RDFPartitioner job

```
   Map(key, v)
       // key:  offset
       // v:  the value of a triple
1      String fName;//filename
2      String ov;//output value
3      if v.property == "rdf:type" then
4          fName = v.property + "#" + v.object;
5          ov = v.subject;
6      else
7          fName = v.property;
8          ov = v.subject + v.object;
9      emit((v.subject + "|" + fName + "-S"), ov) ;
10     emit((v.property + "|" + fName + "-P"), ov) ;
11     emit((v.object + "|" + fName + "-O"), ov) ;
   end
   Reduce(key, values)
       // key:  triple's attribute value|fileName
       // values:  triples
12     put values in file reducerID_key.fileName and store it in HDFS;
   end
```

3.3 Handling skewness in property values

In practice, the frequency distribution of property values in RDF datasets is higly skewed, i.e., some property values are much more frequent than others [20]. Hence, some property-based files created by CliqueSquare may be much larger than others, degrading the global partitioning time due to unbalanced parallel efforts: processing them may last a long time after the processing of property files for non-frequent properties.

To tackle this, map tasks in CliqueSquare keep track of the number of triples for each property file. When the number of triples reaches a predefined threshold, the map task decides to split the file and starts sending triples into a new property file. For example, when the size of the property file `takesCourse-P` reaches the threshold, the map task starts sending `takesCourse` triples into the new property file `takesCourse-P_02`, which may if necessary overflow into

node n_1	**node** n_2	**node** n_3
:stud1 :takesCourse :db	:stud1 :takesCourse :db	:prof1 :advisorOf :stud1
:stud1 :member :dept4	:stud1 :member :dept4	:prof1 :name "bob"
:stud1 :name "ted"	:dept1 type Dept	:prof2 :advisor s2
:prof1 :advisor :stud1	:stud2 :member :dept1	:prof2 :name "alice"
:stud2 :takesCourse :os	:prof1 :name "bob"	:stud1 :name "ted"
:prof2 :advisor :stud2	:prof1 :advisor :stud1	:stud1 :name "ted"
:stud2 :member :dept1	:prof2 :advisor :stud2	:prof1 :name "bob"
:dept1 type Dept	:stud2 :takesCourse :os	:prof2 :name "alice"
:stud1 :member :dept4	:prof2 :name "alice"	:stud1 :takesCourse :db
:stud2 :member :dept1	:dept1 type Dept	:stud2 :takesCourse :os

Figure 3: Data partitioning process: triples arriving at each node after the routing of the shuffle phase.

node n_1					
1_takesCourse-S	1_member-S	1_advisor-O	1_name-S	1_type#Dep-O	1_member-P
:stud1 :db :stud2 :os	:stud1 :dept4 :stud2 :dept1	:prof1 :stud1 :prof2 :stud2	:stud1 "ted"	:dept1	:stud1 :dept4 :stud2 :dept1

node n_2					
2_takesCourse-O	2_member-O	2_name-O	2_advisor-P	2_type#Dept-S	2_type#Dept-P
:stud1 :db :stud2 :os	:stud1 :dept4 :stud2 :dept1	:prof1 "bob" :prof2 "alice"	:prof1 :stud1 :prof2 :stud2	:dept1	:dept1

node n_3				
3_advisor-S	3_name-S	3_name-P	3_name-O	3_takesCourse-P
:prof1 :stud1 :prof2 :stud2	:prof1 "bob" :prof2 "alice"	:prof1 "bob" :prof2 "alice" :stud1 "ted"	:stud1 "ted"	:stud1 :db :stud2 ::os

Figure 4: Data partitioning process: triples in files at each node after the reduce phase.

another partition `takesCourse-P_03` etc. The new property files end up to different reduce tasks to ensure load balancing.

3.4 Fault-Tolerance

Fault-tolerance is one of the biggest strengths of HDFS as users do not have to take care of this issue for their applications. Fault-tolerance in HDFS is ensured through the replication of data blocks. If a data block is lost, e.g., because of a node failure, HDFS simply recovers the data from another replica of this data block. CliqueSquare also replicates RDF data three times. However, each replica is partitioned differently (based on the subject, property, and object). As a result, the copies of data blocks do not contain the same data. Consequently, some triples from the RDF data might be lost in case of a node failure. This is because such triples might belong to data blocks that were stored on the failing node.

Thus, fault-tolerance is a big challenge in this scenario. A simple solution to this problem is to partition a computing cluster into three groups of computing nodes. Each group is responsible of storing a different replica. This would avoid losing triples in case of node failures. However, this does not avoid CliqueSquare to read a large number of data blocks to recover the failed data blocks (stored on the failing node). The database community recognises this issue as a challenging and interesting problem. Hence, some research projects already started to deal with this problem, e.g., [36]. This is an interesting research direction that we would like to investigate in the future.

4 Query model

In this section we lay the foundations on which we base our query processing framework. We define the logical and physical (MapReduce-based) operators which we use for query evaluation.

4.1 Logical operators

Let Val be an infinite set of data values, A be a finite set of attribute names, and $R(a_1, a_2, \ldots, a_n)$, $a_i \in A$, $1 \leq i \leq n$, denote a relation over n attributes, such that each tuple $t \in R$ is of the form $(a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n)$ for some $v_i \in Val$, $1 \leq i \leq n$.

In our context, $Val \subseteq \{U \cup B \cup L\}$ and every mapping $\mu(tp)$ of a triple pattern tp from $V \cup B$ to $U \cup B \cup L$ is a tuple in a relation R_{tp} with $A = varbl(tp)$. For presentation purposes

and without loss of generality we simplify $varbl(tp)$ function to keep only those variables from tp which participate in a join.

Definition 4.1 (Logical operators). *The supported logical operators denoted as LOP are:*

- *The* match *operator, $M[tp][R_o]$, which takes as an input a triple pattern tp and outputs a relation R_o formed by the set of triples matching the specified triple pattern tp. The attributes of R_o are the variables of tp and the tuples are the values of the variables found in the matching triples.*
- *The* join *operator, $J_A[R_1, ..., R_n][R_o]$, which takes as an input a set of relations $R_1, ..., R_n$ and outputs a relation R_o which is the set of all combinations of tuples from the relations $R_1, ..., R_n$ which agree on the values of attributes A.*
- *The* project *operator, $\pi_A[R_i][R_o]$, which takes as an input a relation R_i and outputs a relation R_o having only the A attributes of R_i.*

Definition 4.2 (Logical plan graph). *A* logical plan graph *G_{LOP} is a rooted directed acyclic graph (DAG) where each node corresponds to a logical operator $lo \in LOP$ and there is a directed edge from lo_i to lo_j if the output of lo_i is used as an input of lo_j.*

A logical plan graph, G_{LOP} for the simple query shown in Figure 6(a) is depicted in Figure 6(c). The generation of this plan will be discussed later in Section 5.2.1. Notice that, we illustrate only the output relation of the join operator since the input relations are visible from the children of the operator. From now on, we only present the output relations with the attributes they contain.

4.2 Physical operators

We now define the physical operators we rely on for executing MapReduce jobs to evaluate the queries.

Definition 4.3 (Physical operators). *The supported physical operators denoted as POP are:*

- *The* map scan *operator, $MS[f][R_o]$, takes as an input a file from HDFS and outputs a relation R_o, each tuple of which is a line in the file f.*
- *The* filter *operator, $\mathcal{F}_{con}[R_i][R_o]$, which takes as an input a relation R_i and outputs a relation R_o whose tuples satisfy the condition con on the attributes.*
- *The* map join *operator, $MJ_A[R_1, ..., R_N][R_o]$, performed in the map phase, takes as an input a set of relations $R_1, ..., R_n$ and outputs a relation R_o which is the set of all combinations of tuples from the relations $R_1, ..., R_n$ that agree on the values of the attributes A.*
- *The* reduce join *operator, $RJ_{A|B}[R_1, ..., R_N][R_o]$, which takes as an input a set of relations $R_1, ..., R_n$ and performs a join on the attributes A by shuffling on the values of A in the reduce phase. Then, in the reduce function a join is performed on attributes B for the relations that contain them. The set of attributes B can be empty.*
- *The* project *operator, $\pi_A[R_i][R_o]$, which takes as an input a relation R_i and outputs a relation R_o which is the projection of R_i on the attributes specified in A.*

Definition 4.4 (Physical plan graph). *A* physical plan graph *G_{POP} is a rooted directed acyclic graph (DAG) where each node corresponds to a physical operator $po \in POP$ and there is a directed edge between two nodes $po_i \rightarrow po_j$ if the output relation of po_i is used as an input relation for po_j.*

An example of a physical plan graph corresponding to the query presented in Figure 6 is illustrated in Figure 10. The creation of the plan is discussed later in Section 5.4. Note that a leaf of a physical query plan is always an MS operator.

Cost model. We use a simplified cost model for a physical query plan G_{POP} taking into account the number of MapReduce *stages*. One stage is either a map or a reduce phase in a MapReduce job.

We distinguish two cases. If the root of G_{POP} is an $\mathcal{F}$, an MS or an MJ operator, the execution will only require 1 map phase for scanning and filtering matching triples and potentially evaluating an MJ. Otherwise, if the root of G_{POP} is an RJ operator, the number of MapReduce stages depends on the length of the longest path from the root to any MJ node in G_{POP}.

Thus, if r is the root node of G_{POP} and d_{max} the length of the longest path from the root node to any MJ node, the cost of evaluating G_{POP} will be:

$$Cost(G_{POP}) = \begin{cases} 1 \text{ stage} & \text{if r is } \mathcal{F}/\text{MS}/\text{MJ} \\ 2 \times d_{max} \text{ stages} & \text{if r is RJ} \end{cases}$$

5 Query processing framework

In this section, we explain how we evaluate queries through MapReduce jobs, on an RDF store partitioned as presented in Section 3.

At the core of BGP queries are joins. In a distributed/parallel environment such as the one we consider, joins can raise performance issues due to the data transfers across the network that they incur. As we will show, our RDF partitioning model enables us to reduce the amount of data shuffling by performing co-located joins (i.e., map-side joins). As a result, we are usually able to process incoming queries in a single MapReduce job, which translates to performance advantages over existing approaches.

We organize the presentation as follows. We start by presenting a set of preliminaries in Section 5.1, which we use in our query processing framework. Section 5.2 introduces two very common classes of queries, which we show can be answered with a single MapReduce job based on our RDF storage model. Section 5.3 provides a general algorithm for building logical plans, while Section 5.4 presents the translation of logical plans into MapReduce programs, completing the description of our query processing approach.

5.1 Preliminaries

For representing a BGP query we use the following form of graph.

Definition 5.1 (Variable graph). *A* variable graph G_V *of a BGP query q is a 4-tuple (N, E, V, ℓ), where N is the set of nodes, E is the set of labeled undirected edges, V is the set of variables occurring in the query q, and ℓ is a total function $\ell : E \to V$ assigning labels to the edges. Moreover:*

- *Each node $n \in N$ corresponds to a set of triple patterns from q.*
- *There is an edge e between n_1, n_2 from N (with $n_1 = n_2$ as a particular case) iff the triple patterns represented by these two nodes share a variable $v \in V$. This edge e is labeled after the shared variable v: $\ell(e) = v$.*

Observe that the above definition allows many edges between two nodes. Also note that, the variable graphs of the BGP queries considered in this paper are always connected, as these queries do not feature cartesian products.

In the following, we use the notion of *colors* to mark the edges of a query graph. One color is assigned to each unique edge label in the query graph (i.e., edges with the same label have the same color). Figure 5 shows a BGP query with its corresponding variable graph G_V, where each node is comprised from one triple pattern. The query is rather complex and abstract on purpose, to allow us presenting some useful notions based on the shape of its variable graph.

```
SELECT
?a ?b ?c ?d ?e
WHERE {
?a p1 ?b
?a p2 ?c
?d p3 ?a
?d p4 ?e
?l p5 ?d
?f p6 ?d
?f p7 ?g
?g p8 ?h
?g p9 ?i
?i p10 ?j
?j p11 ?k
}
```

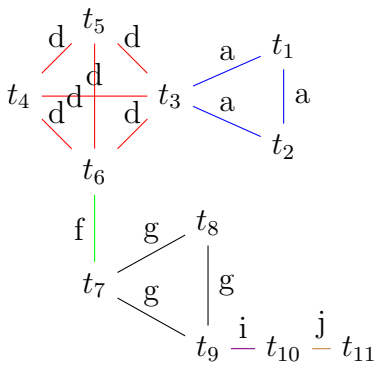


Figure 5: SPARQL query Q1 and its variable graph.

Definition 5.2 (Join variables). *Given a query q the* join variables *JV of q is the set of the variables which appear more than once in the triple patterns of query q.*

Note that the join variables of a query are the ones that appear as labels in its variable graph. For example, the join variables for the query depicted in Figure 5 are $\{a, d, f, g, i, j\}$.

In the following, we borrow the concept of a *clique* from graph theory and overload it as follows.

Definition 5.3 (Variable clique). *Let $G_V : (N, E, V, \ell)$ be a variable graph and $T \subseteq V$ a set of variables. The* variable clique *of T, denoted by $C\ell_T$, is the set of all nodes from N which are adjacent to an edge $e \in E$ such that $\ell(e) \in T$.*

Note that the definition of a variable clique concerns maximal cliques. This means that given a variable clique $C\ell_{\{x\}}$ in a graph G_V there exists no other variable clique for variable x in G_V. Consider the following examples. The variable clique $C\ell_{\{a\}}$ for the graph in Figure 5 is $\{t_1, t_2, t_3\}$, and $C\ell_{\{a,d\}}$ for the same graph is $\{t_1, t_2, t_3, t_4, t_5, t_6\}$.

Definition 5.4 (Clique subgraph). *Let G_V be a variable graph (N, E, V, ℓ), and x a join variable. A variable graph G'_V: (N', E', V', ℓ') is a* clique subgraph *of G_V with respect to the join variable x, denoted by $G'_V \sqsubseteq_x G_V$, if and only if:*

- *Nodes N' form a variable clique of $\{x\}$ in G_V.*
- *$E' = \{e \in E \mid \ell(e) = x\}$: G'_V contains all the edges of E labeled with the variable x.*
- *$V' \subseteq V$: the variables of G'_V are included in the variables of G_V.*
- *$\ell' = \ell|_{E'}$: ℓ' is the restriction of ℓ to E'.*

For instance, in Figure 5, the clique subgraph of variable d consists of the four nodes $\{t_3, t_4, t_5, t_6\}$ and the edges connecting them.

Definition 5.5 (Union of variable graphs). *Let $\{G_1 : (N_1, E_1, V_1, \ell_1), ..., G_k : (N_k, E_k, V_k, \ell_k)\}$ be a set of variable graphs. The union of the graphs $\bigcup_{1 \leq i \leq k} G_i$ is a variable graph $G : (N, E, V, \ell)$ such that $N = \bigcup_{1 \leq i \leq k} N_i$, $E = \bigcup_{1 \leq i \leq k} E_i$, $V = \bigcup_{1 \leq i \leq k} V_i$ and $\ell_{|E_i} = \ell_i$.*

Definition 5.6 (Clique decomposition). *Let $G_V : (N, E, V, \ell)$ be a variable graph. A* clique decomposition *of G_V is a set of clique subgraphs $\{G_1, ..., G_n\}$ of G_V whose union produces the original graph: $G_V = (\bigcup_{1 \leq i \leq n} G_i)$.*

An illustration is readily provided by Figure 5, which features one four-node clique ($\{t_3, t_4, t_5, t_6\}$ as mentioned above), two three-nodes cliques ($\{t_1, t_2, t_3\}$ and $\{t_7, t_8, t_9\}$) and three two-node cliques ($\{t_6, t_7\}$, $\{t_9, t_{10}\}$ and $\{t_{10}, t_{11}\}$). The decomposition of the graph in cliques is the set of these six cliques, one for each color appearing in the figure.

Proposition 5.1 (Unique decomposition). *Given a query q with variable graph $G_V : (N, E, V, \ell)$ and join variables JV there exists a unique clique decomposition of G_V into exactly $|JV|$ clique subgraphs.*

Proof. Since a variable clique is maximal (see Definition 5.3), for each join variable $i \in JV$ we can identify a unique variable clique $C\ell_{\{i\}}$. For each $C\ell_{\{i\}}$ there is a clique subgraph G_V^i such that $G_V^i \sqsubseteq_i G_V$. G_V^i has as nodes the variable clique $C\ell_{\{i\}}$, as edges the edges of G_V that are labeled by i and as variables the variables from V that appear in the triple patterns of $C\ell_{\{i\}}$. □

5.2 Smart plans for small queries

We consider two simple yet very popular classes of queries whose evaluation requires *at most* a full MapReduce job.

5.2.1 Single job - map phase

We identify the following class of queries based on their syntax:

Definition 5.7 (1-clique query). *Let G_V be a variable graph of a query q. Query q is 1-clique query iff there is a clique subgraph $G'_V \sqsubseteq_x G_V$ such that the nodes of G_V are the same with the nodes of G'_V.*

Intuitively, 1-clique queries are queries which share a variable among *all* their triple patterns.

To this category belong all star-shaped BGP queries, which share a variable typically in the subject position (it can also appear in the object position but this is more rare). However, the class of 1-clique is strictly larger than star queries, since it includes also those where some triple patterns share a variable across *distinct* positions in triple patterns. For instance, the query `SELECT ?x WHERE { ?x` $\mathtt{p}_1$ $\mathtt{o}_1$ `.` $\mathtt{s}_2$ $\mathtt{p}_2$ `?x}` is not a star since `?x` appears in different positions in the two triple patterns, yet it has only one clique.

Importantly for the problem we study, we have:

Proposition 5.2 (Map-only job for 1-clique queries). *1-clique queries can be evaluated in the map phase of one job.*

Proof. By definition 1-clique queries contain only one join variable, entailing that the triple patterns belonging to these queries are joined together using this variable. Recall our partitioning scheme, which places triples sharing a value in the same node regardless of whether it is subject, property or object. As a result the join for those triple patterns can be computed *locally* at

each node. To answer 1-clique queries it is sufficient to evaluate the query independently on each node and union the results afterwards, to form the final answer. The queries are processed using a single map-only job. □

This map-only join evaluation resembles the directed-join described in [4].

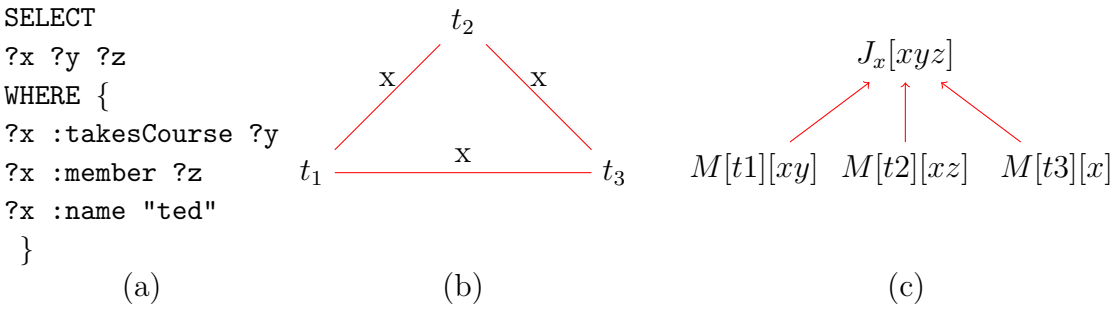


Figure 6: BGP query Q2 (a), its variable graph (b), and its logical plan (c).

An example of an 1-clique query with the shared variable in the subject position is shown in Figure 6(a). In the same figure we can see the variable graph for the query (b), and its corresponding logical plan (c). For each triple pattern a match operator scans the data, selects the triples that match the triple pattern and creates a relation with attributes the variables of the triple pattern and tuples the bindings of the variables. The join operator combines the outputs of the three match operators on their common variable x.

5.2.2 Single job - map & reduce phase

Continuing with our classification of queries, we introduce:

Definition 5.8 (Central clique). *Let $G_V : (N, E, V, \ell)$ a variable graph of a query q, and $\{G_1, ..., G_n\}$ its clique decomposition. There is a* central clique *in G_V, iff there is a clique subgraph G_i in the decomposition which overlaps with all other clique subgraphs in $\{G_1, ..., G_n\}$.*

In the above, the clique G_i must overlap (have one node in common) with any other clique G_j, $i \neq j$, $1 \leq i, j \leq n$, but this does not need to be a single (same) G_i node. We term *central-clique queries* those queries having a central clique; obviously, any 1-clique query is also central-clique, but the class of central-clique queries is larger since it allows more than one clique. Our interest in such queries stems from:

Proposition 5.3 (Map-reduce job for central-clique queries). *Queries having a central clique can be evaluated in one complete job.*

Proof. To answer this type of queries we decompose the query into 1-clique subqueries. We showed earlier (Proposition 5.2) that we can evaluate 1-clique queries in the map phase of one job. Based on Definition 5.8, all 1-clique subqueries have at least one common variable (from the central clique). Thus, the results of all 1-clique subqueries can be joined on this variable, during the reduce phase of the job. Thus, one complete map-reduce job suffices to answer these queries. □

Figure 7 shows a central-clique query with its variable graph, and the derived logical plan. The query can be decomposed into two 1-clique subqueries; the red one based on variable $?x$ and the blue one using variable $?w$. Both cliques can be considered central since they have

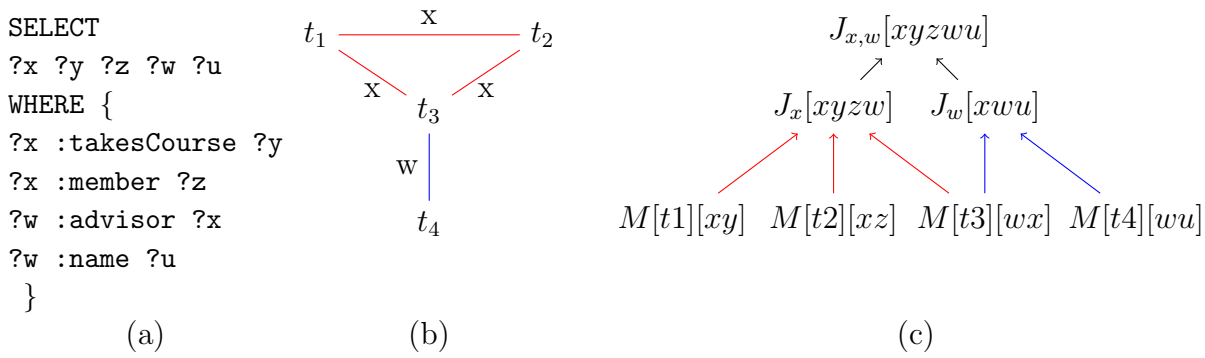


Figure 7: BGP query Q3 (a), its variable graph (b), and its logical plan (c).

the common node t_3. Therefore, based on Proposition 5.3, the query can be evaluated in one MapReduce job. The 1-clique subqueries are evaluated in the map phase and there is an extra join operator on top for combining the intermediate results which is evaluated in the reduce phase.

In practice, there are many real-world SPARQL queries which fall in this category. As we will show in Section 6.5, more than 99% of real queries taken from DBPedia's logs are central-clique queries and can, thus, be answered in a single MapReduce job.

5.3 CliqueSquare algorithm

Although many of the real-world queries are either 1-clique or central-clique queries, we propose an algorithm for the general case of queries that may be neither 1-clique nor central-clique. The evaluation of such queries on our RDF store needs more than one MapReduce job.

Based on Proposition 5.1 about unique decomposition, we present the CliqueSquare algorithm for producing logical query plans from BGP queries. CliqueSquare is based on the variable graph of a query and its decomposition into clique subgraphs. The algorithm works in an iterative way, identifying cliques and "collapsing" them successively, by evaluating the joins on the common variables of each clique. The process ends when the variable graph consists of only one node.

We start by introducing some definitions.

Definition 5.9 (Complete set of variable cliques). *Let $G_V : (N, E, V, \ell)$ be a variable graph of a query q, and JV the set of all its join variables. We define the* complete set of variable cliques *for G_V, denoted by $\mathcal{CVC}$, as the set of cliques $\{C\ell_{\{u\}} \mid u \in JV\}$.*

For example, the complete set of variable cliques for the graph in Figure 5 is shown below.

$$\{C\ell_{\{a\}}, C\ell_{\{d\}}, C\ell_{\{f\}}, C\ell_{\{g\}}, C\ell_{\{i\}}, C\ell_{\{j\}}\}$$

The iterative transformation of the variable graph in the algorithm may result to variable cliques which are either identical or included in one another. To eliminate such redundancies we introduce the two simplification transformations below:

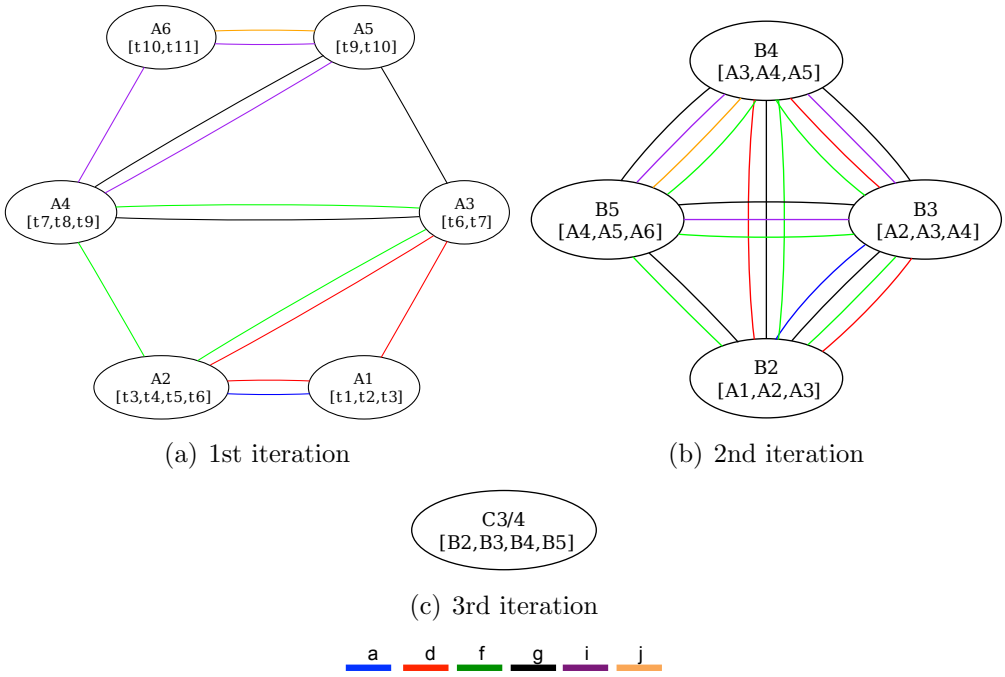
Definition 5.10 (Clique set simplifications). *Let $G_V : (N, E, V, \ell)$ be a variable graph and $\mathcal{VC}$ a set of variable cliques for G_V. We define the following two* simplification transformations (or simplification, in short) *of a clique set:*

- equivalence $\epsilon : \mathcal{VC} \rightarrow \mathcal{VC}$ *is defined as: for $C\ell_{\{i\}}, C\ell_{\{j\}} \in \mathcal{VC}$, if $C\ell_{\{i\}} = C\ell_{\{j\}}$ then $\epsilon(\mathcal{VC}) = (\mathcal{VC} \setminus \{C\ell_{\{i\}}, C\ell_{\{j\}}\}) \cup C\ell_{\{i,j\}}$ (we merge the equivalent cliques);*

- subset $\sigma : \mathcal{VC} \rightarrow \mathcal{VC}$ *is defined as: for* $C\ell_{\{i\}}, C\ell_{\{j\}} \in \mathcal{VC}$, *if* $C\ell_{\{i\}} \subset C\ell_{\{j\}}$ *then* $\sigma(\mathcal{VC}) = \mathcal{VC} \setminus C\ell_{\{i\}}$ *(if a clique* $C\ell_{\{i\}}$ *is contained into a clique* $C\ell_{\{j\}}$*).*

The pseudocode of CliqueSquare is shown in Algorithm 2. CliqueSquare takes as an input a BGP query q and outputs a logical plan graph G_{LOP}; at each iteration, some cliques are identified and their corresponding fragment of a MapReduce plan is build, while the variable graph is simplified accordingly by merging nodes belonging to the same clique. Each clique will correspond to a node in the rebuilt graph.

For illustration, consider query Q1 of Figure 5 to demonstrate the steps of the algorithm. The variable graph G_V for all intermediate steps of the algorithm for Q1 is shown in Figure 8. The initial query graph G_V is created from the query, according to Definition 5.1, at the algorithm line 2. For each node in G_V (i.e., each triple pattern tp of the query), a match (M) operator is created, where the input is the triple pattern tp and the output is a relation R_o with attributes the variables of tp and values the bindings of the variables for all the matching triples of tp. The operators are added in G_{LOP} (line 3).



(a) 1st iteration (b) 2nd iteration

(c) 3rd iteration

Figure 8: Variable graph after each iteration.

Then, the output of the M operators should be joined to produce the intermediate or final results. To decide on the precise operators to join and the choice of the join attributes, we explore and transform the variable graph, working on the variable cliques. We start by creating the complete set $\mathcal{CVC}$ for G_V (line 5). Since there may be more than one edge between nodes in G_V, there might be cliques in $\mathcal{CVC}$ that are subset of one another or which are identical. Such cases can be found for the graph in Figure 8(a) where $C\ell_a \subset C\ell_d$ and $C\ell_j \subset C\ell_i$ and in Figure 8(b), where $C\ell_f = C\ell_g = \{B2, B3, B4, B5\}$. For this reason, we apply the simplifications introduced in Definition 5.10, at line 6 of the algorithm. This simplification allows us to eliminate superfluous joins, since the same relations are already contained in another clique set. Avoiding this step would not harm the correctness of the algorithm but rather affect the efficiency since we would introduce more joins which are redundant. The same reasoning applies for clique sets that are equivalent.

Based on the remaining cliques after the simplifications, we build a new variable graph G_V, where each variable clique in the simplified $\mathcal{CVC}$, corresponds to a single node in G_V (line 7). A

Algorithm 2: CliqueSquare

```
CliqueSquare(q, G_LOP)
    Input  : Conjunctive SPARQL query q
    Output: Logical plan graph G_LOP
1   G_LOP ← ∅;
2   G_V ← createVarGraph(q);
3   G_LOP ← addMoperators(G_V);
4   repeat
5       CVC ← findVarCliques(G_V);
6       CVC ← simplifyVarCliques(CVC);
7       G_V ← createVarGraph(CVC);
8       G_LOP ← addJoperators(G_V);
9   until |G_V.nodes| = 1;
end
```

node in the new G_V corresponds to the result of joining several triple patterns from the original query. To refer to such new nodes, we give them ad-hoc names of the form $A1$, $A2$ etc. and to help the reader trace how such nodes were obtained, in Figure 8 we show in square brackets the names of the nodes *from which* the node was created. The nodes in the newly created G_V are connected by one edge for each join variable they share.

To record how each new G_V node is created out of nodes from the previous-level variable graphs, we introduce a J_A operator in G_{LOP}. The attribute list A of the join consists of the variables defining the clique to which this node corresponds, whereas the nodes belonging to the clique correspond to the input relations for J_A. For example consider node $A1$ in Figure 8(a) which corresponds to $C\ell_a$; for this node we introduce the operator $J_a[t_1, t_2, t_3][ad]$.

Finally, the join operators are added to G_{LOP} (line 8). The algorithm proceeds iteratively until the variable graph is transformed to a graph with a single node, which corresponds to a single relation, materializing the result of the complete join expression that is the body of the query.

The logical plan graph G_{LOP} for query Q1 produced by CliqueSquare is shown in Figure 9.

Proposition 5.4 (Number of CliqueSquare iterations). *Given a query q with variable graph $G_V : (N, E, V, \ell)$ and join variables set JV, the total number of iterations for CliqueSquare cannot exceed $|JV|$.*

Proof. Consider the worst case scenario of a path query q with n triple patterns and $n-1$ join variables. In this case, the first step of the algorithm creates $n-1$ relations consisting of exactly two triple patterns (e.g., $t_1 \bowtie t_2$, $t_2 \bowtie t_3$, ..., $t_{n-1} \bowtie t_n$). In each subsequent step we combine in the worst case two of the composite relations together resulting into relations with at least three triple patterns in the second step, four triple patterns in the third step, etc. The complete join expression over n triple patterns will be reached in the $|JV|$-th step of the algorithm.

It is easy to see that this is the worst case. Indeed, for all other shapes of queries, the variable graph has fewer cliques than $n-1$ and accordingly less join stages (iterations) are necessary. □

5.4 Query evaluation on Hadoop

In this section, we describe how a logical query plan G_{LOP} produced by CliqueSquare is translated to a physical query plan, and how this physical query plan is executed in Hadoop.

For each logical operator found in G_{LOP} we construct physical query operators as follows.

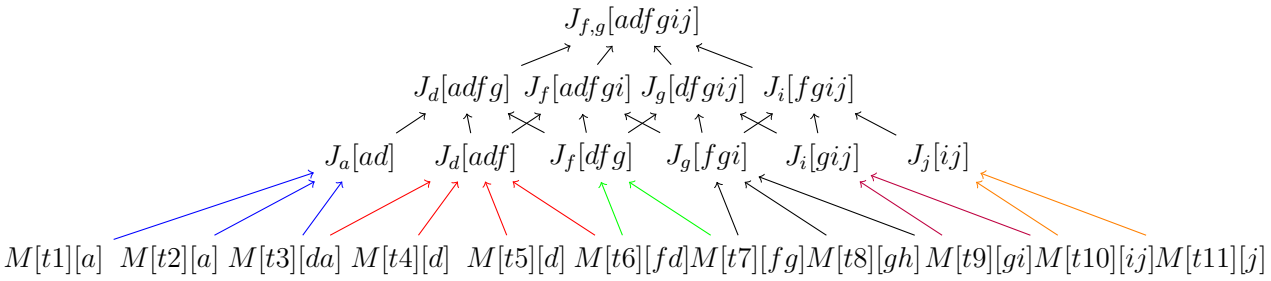


Figure 9: Logical plan for the query Q1 (shown in Figure 5).

Match operator. Let $M[tp]$ be a match operator in G_{LOP} having k edges. We create the following physical operators:

1. k *scan* operators $MS[f_j]$ $(1 \leq j \leq k)$, one for each edge of $M[tp]$. f_j is a local property-based file, as introduced in Section 3. The name of f_j is made of two parts: (i) the property of tp[2], and (ii) a string token indicating whether we have to scan the subject, property or object partition of the property file. The latter depends on the position of the join variable of tp. Since tp can have up to three join variables, we have to follow the current edge in G_{LOP}, from $M[tp]$ to its first J ancestor, to deduce the current join variable and thus, the partition that needs to be scanned.

2. If the triple pattern tp has a constant in the subject and/or object position, a *filter* operator, $\mathcal{F}_{con}$, is added on top of $MS[f_j]$, where *con* are the conditions on which we filter the matching tuples on the subject and/or object of tp. Note that the filter on the property is implicitly done by the scan operator and the name of the property-based files.

Join operator. Let J_V be a logical join operator in G_{LOP}. Three cases may occur:

1. If all children nodes of J_V are match operators, then J_V is transformed into a map join MJ_V.

2. Each logical join operator that is not the root of G_{LOP} is transformed into a reduce join $RJ_{V_1|V_2}$, where $V_1 = V$ and $V2 = \emptyset$.

3. If J_V is the root of G_{LOP} then we introduce a reduce join operator $RJ_{V_1|V_2}$, where $V_1 = V$ and $V_2 = JV \setminus V$. The first join on variables V_1 is done during the shuffle phase, while the join on variables V_2 is done as a post-processing step inside the last reduce phase. The latter ensures that any results that have reached the root and were not joined up to that point along some path in the query plan, will be joined locally by the reduce function of the last reduce.
   Consider, for example, the following two paths in Figure 9: (i) $M[t_3] \rightarrow J_d \rightarrow J_f \rightarrow J_{f,g}$ and (ii) $M[t_3] \rightarrow J_a \rightarrow J_d \rightarrow J_{f,g}$. In this case, there may be some values for the variable `?a` which have reached the top but they have not been joined together.

Project operator. The logical project operator, $\pi_A[R_i][R_o]$, is directly mapped to the physical project operator.

A physical query plan is mapped to a sequence of MapReduce jobs quite simply. In the map phase of the first MapReduce job all MS, $\mathcal{F}$ and MJ operators are evaluated. Initially, each mapper scans the appropriate files from HDFS one after the other, and passes to the map function a $(key, value)$ pair, where the *value* is the triple read from the file and the *key* indicates the file (and thus, triple pattern) from which the triple was read. Then, in the map function,

[2]If the property of tp is a variable, the wildcard "*" is used, meaning we have to scan all files.

the $\mathcal{F}$ operators eliminate triples that do not match the triple pattern; then, one hash-join for each MJ_V operator is performed.

The $RJ_{V_1|V_2}$ operators, if any, need to be evaluated in the reduce phase as they join intermediate results and data shuffling is imperative. The partitioning key in the shuffle phase is the concatenation of the values of the V_1 variables. If V_2 is non-empty, an extra join on the V_2 variables is performed in the reduce function.

The first level of RJ operators (whose children are all MJ nodes) are performed in the reduce phase of the first MapReduce job. Then, for each level of RJ operators a new MapReduce job is initiated.

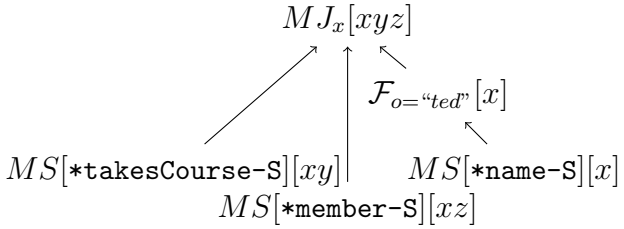


Figure 10: Physical plan for Q2.

Example (query Q2). We illustrate the above on the logical plan of query Q2, which is shown in Figure 6(c). The physical plan of Q2 is shown in Figure 10. A MapReduce job for answering this query over the data in Figure 1 works as follows. Each node scans its local files `*takesCourse-S`, `*member-S`, `*name-S`, one after the other, and passes to the map function a $(key, value)$ pair, where the $value$ is the triple read from the file and the key is a number indicating the file (and thus, triple pattern) from which the triple was read. Then, two hash-joins are performed in the map function, e.g., one for joining the triples having property `takesCourse` with the triples having property `member` on their subject, and another one combining the result of the first join, with the triples corresponding to the `name` triple pattern. From the file `*name-S`, only triples having as an object the value "*ted*" are kept. The final results is then written back into the HDFS. The actual join order inside the map phase can be decided with the help of a standard cost-based optimizer RDF query optimizer, e.g., cost-based as in [11] or heuristic-based as in [32].

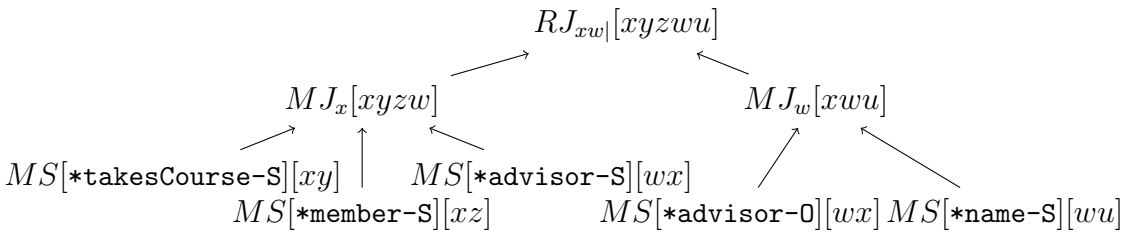


Figure 11: Physical plan for Q3.

Example (query Q3). Now recall query Q3 from Figure 7. Its logical plan appears in Figure 7(c) and its physical plan in Figure 11. As shown by the physical query plan there are two map joins. In the map phase, one map task (call it MT1) joins the first three triple patterns on variable `?x`, while another map task MT2 joins the last two triple patterns on variable `?w`. MT1 scans the files `*takesCourse-S`, `*member-S` and `*advisor-O` and performs a three-way join on `?x`, while MT2 scans the files `*advisor-S` and `*name-S` and performs a two-way join on `?w`. Then, both map tasks send the joined results to the reducers, using as key the concatenation

of the values of the common variables ?x and ?w, and as value the bindings of the other variables (in this case, ?y, ?z and ?u).

In the reduce phase, the intermediate results sharing the same values for ?x and ?w will be located in the same node. We join them on ?x and ?w locally at each node, in order to develop complete result tuples (with bindings for ?x, ?y, ?z, ?w and ?u). The final results are written back to the HDFS.

6 Experimental evaluation

We present preliminary experimental results evaluating our CliqueSquare prototype and comparing its performance with HadoopRDF [16], the state-of-the-art Hadoop-based RDF store, in terms of both data upload time and query processing time and network traffic. In addition, we provide some interesting statistics of real-world BGP queries with respect to our clique-based formalization.

Section 6.1 outlines the experimental setup and introduces the datasets and queries used in our experiments. Section 6.2 describes experimental results of loading data in our store, whereas Section 6.3 presents query performance and Section 6.4 focuses on data transfers incurred by query evaluation. Section 6.5 presents the real-world query statistics, then we conclude.

6.1 Experimental setup

In this section we first detail the specifications of the cluster on which we run our experiments as well as the datasets and queries we use. We also briefly describe the functionality of HadoopRDF [16], the system against which we compare our work.

Cluster. We use a cluster of 8 nodes, where each node has: eight 2.93GHz Quad Core Xeon processors; 4×4GB of main memory; two 600GB SATA hard disks configured in RAID 1; one Gigabit network card; Linux CentOS release 6.4.

Dataset and queries. For our experimental evaluation we use the Lehigh University benchmark (LUBM) [12] which has been extensively used in other works such as [16, 15]. The evaluation with real-world datasets and queries is the subject of our future work.

LUBM provides synthetic RDF datasets of arbitrary sizes. It consists of a university domain ontology modeling an academic setting and is widely used for testing RDF stores. Each dataset can be defined by the number of universities generated; for example, the dataset LUBM1 involves one university, while the dataset LUBM10 incorporates 10 universities. The more universities are involved in the data generation, the more triples are produced. We use two different datasets for our experiments: the LUBM10K and LUBM20K datasets. LUBM10K contains approximately one billion triples (216 GB), and LUBM20K about 2 billion triples (432 GB).[3]

LUBM benchmark contains 14 different queries. It is worth noticing that all of these queries can be mapped either to 1-clique queries (Section 5.2.1), or to central-clique queries (Section 5.2.2) and thus can be answered in a single MapReduce job using CliqueSquare algorithm. We use queries Q1, Q2, Q4, Q9 from LUBM which we have slightly modified so that RDFS reasoning is not necessary for returning a non-empty answer. In these queries we have only replaced the object of some of the rdf:type triple patterns keeping the structure of the query unchanged. We also add a new query Q15 to demonstrate that even non star-shaped queries can be answered in a single map phase. The rest of the LUBM queries exhibit similar characteristics and thus, we decided to omit them from our evaluation. In our future evaluation we plan to construct more complicated queries which require more than 1 job.

[3]We do not consider bigger datasets due to hard drive space limitations in our cluster.

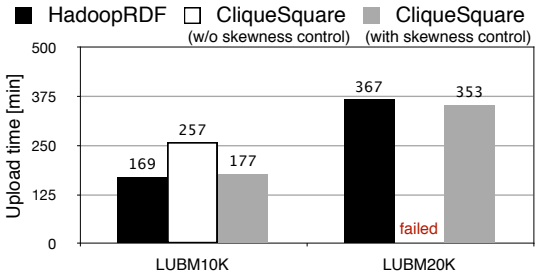


Figure 12: Data upload for different datasets.

The SPARQL queries we use can be found in Appendix A. Their characteristics are summarized in Table 1; number of triple patterns ($\#tps$), number of join variables ($\#JV$) and cardinality of results for both datasets ($card\text{-}LUBM10K$ and $card\text{-}LUBM20K$).

	Query1	**Query2**	**Query4**	**Query9**	**Query15**
$\#tps$	2	6	5	6	4
$\#JV$	1	3	1	3	1
$card-LUBM10K$	4	306	10	440,199	13,673,436
$card-LUBM20K$	4	629	10	879,422	27,352,179

Table 1: Query characteristics.

Systems. We use Oracle JDK v1.6.0_43 and Hadoop v1.0.4 for all experiments with default settings apart from the HDFS block size which we increase to 256MB. Notice that, we use one node to run the JobTracker, the NameNode, and the Secondary NameNode daemons in addition to the TaskTracker and DataNode daemons.

We compare our work against HadoopRDF [16] an open-source state-of-the-art system to store and query RDF data using Hadoop and HDFS. Although the source code of HadoopRDF is available online[4], we encountered a lot of bugs and thus, we used a debugged version provided to us by the authors of [24].

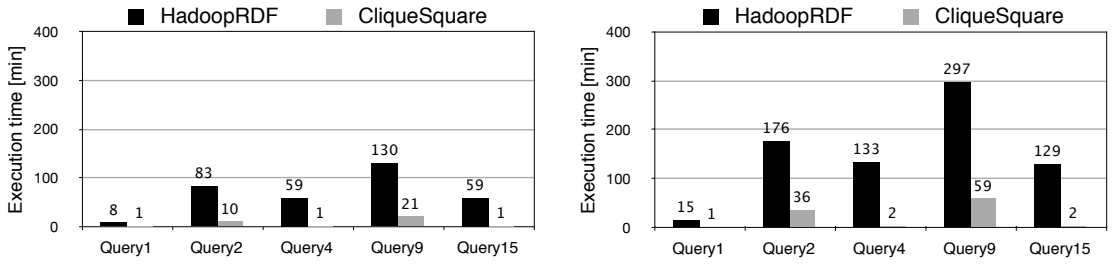
In HadoopRDF, in the upload phase of RDF data triples are firstly grouped based on their property value as we discussed in Section 3. Then, triples with the same property are further split and grouped based on the RDFS class their object belongs to (if such information exists). For example, for each triple $t =$ `(s p o)` if there is a triple (`o rdf:type c`) which states that `o` is of type `c`, then t is placed in a file named `p#c`. This is determined by inspecting all the triples having property value `rdf:type`. If such information is not available, t is stored in a file named `p`. Although we use a similar way for grouping triples based on their property, we do so at *each node* after the triples are placed to the appropriate nodes. In HadoopRDF the placement of the triples inside each property file is controlled by HDFS.

Query evaluation in HadoopRDF starts by selecting the HDFS files that need to be used for the query processing. Then, a heuristic approach which finds a query plan with the least number of MapReduce jobs is used. However, because HadoopRDF lacks data locality, much data is transferred in the network causing a big overhead during query evaluation as we will see in the following.

6.2 Data upload

We start by measuring the impact in data upload times of the data partitioning strategy used by CliqueSquare. For this, we upload the datasets LUBM10K and LUBM20K and compare CliqueSquare with HadoopRDF. We use two variants of CliqueSquare to better evaluate its skewness control described in Section 3.3.

[4] `https://code.google.com/p/hadooprdf/`



(a) LUBM10K (b) LUBM20K

Figure 13: Query evaluation time comparison.

Figure 12 illustrates the upload times for both frameworks. We observe that CliqueSquare (with skewness control) achieves the same upload times on average as HadoopRDF, even though CliqueSquare has a more elaborated partitioning mechanism. In particular, we observe that CliqueSquare (with skewness control) is faster than HadoopRDF for bigger datasets (i.e., for LUBM20K). This is because the skew handling mechanism used by CliqueSquare allows it to better balance the data upload process across computing nodes. This is not the case for CliqueSquare (w/o skewness control). CliqueSquare (with skewness control) is ∼1.5 times faster than CliqueSquare (w/o skewness control) for the LUBM10K dataset. Notice that CliqueSquare (w/o skewness control) fails for the LUBM20K dataset, because some computing nodes get overloaded. This shows the efficiency of the skew control used by CliqueSquare.

6.3 Query runtime

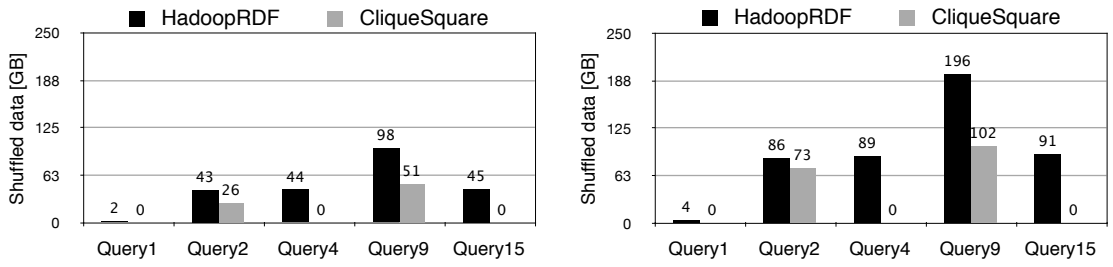
We now analyse the performance of CliqueSquare when running BGP queries. Our goal in these experiments is to show: (a) the efficiency of our system in comparison with HadoopRDF, and (b) the impact of the query structure in the execution time.

Figure 13(a) shows the performance of CliqueSquare for the LUBM10K dataset. We observe that CliqueSquare significantly outperforms HadoopRDF for all queries. CliqueSquare outperforms HadoopRDF by an improvement factor of 28 on average and up to 59 (for `Query4` and `Query15`). In particular, we observe that CliqueSquare can run all queries in 34 minutes while HadoopRDF can only run `Query1` within that time.

Figure 13(b) illustrates the results for LUBM20K. Similarly to the LUBM10K dataset, we observe that CliqueSquare outperforms HadoopRDF by more than one order of magnitude: an improvement factor of 31 on average and up to 67. We also observe that CliqueSquare runs all queries in 100 minutes while HadoopRDF runs `Query1` within that time.

Overall, we observe that users using CliqueSquare have to wait only a few minutes to get the results to most of their queries. This is not the case for users using HadoopRDF who have to wait for hours. The short execution times of CliqueSquare are mainly due to the fact the CliqueSquare does not have to transfer large amounts of data through the network. We study this aspect in detail in the next set of experiments.

The structure of the query plays an important role in the execution time. As shown by the results the number of join variables is the factor that significantly affects the efficiency of the queries in CliqueSquare, as opposed to the number of joins and thus, the number of triple patterns which affects HadoopRDF. Queries with fewer number of join variables are usually faster in CliqueSquare. In our experiments `Query1`, `Query4`, and `Query15`, which have only one join variable (?x) can be answered in less than 3 minutes. The rest of the queries (`Query2` and `Query9`) have three join variables and thus, higher execution times. The number of join variables is tightly connected with the number of MapReduce stages as shown in Section 5 which explains the execution times of the queries. The same does not hold for HadoopRDF, since the running times for `Query1` and `Query4` greatly vary, despite the fact that both have only one



(a) LUBM10K (b) LUBM20K

Figure 14: Size of the data transferred during the shuffle phase.

join variable.

6.4 Data transfer

One of the main goals of our framework is to reduce the amount of data transferred trough the network. We study this aspect of CliqueSquare in this section by measuring the number of bytes sent by map tasks to reduce tasks in each query we consider (i.e., we measure the shuffle phase cost).

Figure 14 shows the amount of data shuffled from map to reduce tasks. We observe in Figure 14(a) that CliqueSquare significantly outperforms HadoopRDF. In particular, we see that for `Query1`, `Query4`, and `Query15` CliqueSquare does not transfer any byte in the shuffle phase as it performs map-only jobs for these queries. This is in contrast to HadoopRDF, which transfers up to 45 GB for `Query15`. Still, for `Query2` and `Query9`, CliqueSquare sends ∼2 times less data than HadooRDF. The results in Figure 14(b) confirm this trend for the LUBM20K dataset. CliqueSquare significantly outperforms HadoopRDF in all queries and by almost one order of magnitude for two of the queries (`Query4` and `Query15`). Indeed, for queries like `Query1`, `Query4`, and `Query15`, CliqueSquare's improvement factor increases along with the size of the dataset. These type of queries do not incur any shuffling with CliqueSquare, whereas they considerably do in HadoopRDF.

6.5 Real-world query statistics

We have conducted a small study to investigate the form of real-world SPARQL queries with respect to our formalizations, based on query logs from DBPedia endpoint[5]. In order to parse them and create the variable graphs, we use Jena[6]. Among the 10 million queries existing in the log files, only half of them were valid and are included in our results.

Table 2 summarizes the collected statistics by classifying the valid queries based on the number of cliques they contain. We report: (*i*) the total number of queries belonging to each category (#queries), (*ii*) the total number of queries represented by a connected variable graph and for which we are concerned in this paper (#connected), (*iii*) the total number of central-clique queries (#central), and (*iv*) the average number of triple patterns (*tps*) that queries in each category have ($AVG(\#tps)$).

We observe that 1-clique queries as we defined them in Section 5.2.1 correspond to almost 99% of the total query log and we can answer them very efficiently in one single map-only job. Adding to these the central-clique queries (Section 5.2.2), we note that based on our partitioning scheme, a full MapReduce job is sufficient to answer more than 99% of real world queries.

Finally, observe that the class of central-clique queries includes some with complex structure and many triple patterns, such as queries with six triple patterns and as many as five cliques. All

[5] `ftp://download.openlinksw.com/support/dbpedia/`

[6] `http://jena.apache.org/`

#cliques	#queries	#connected	#central	$AVG(\#tps)$
0	4,111,964	4,100,276	4,100,276	1.00
1	963,257	963,103	963,103	2.00
2	13,930	13,876	13,876	3.18
3	9,647	9,613	9,613	4.04
4	18,771	18,761	98	5.01
5	3,169	3,169	3	6.05
6	19	19	0	8.73
7	12	12	0	11.75
10	1	1	0	18.00
Total	5,120,770	5,108,830	5,086,969	1.22

Table 2: DBPedia queries classified on #cliques.

of them can be efficiently answered in one MapReduce job following the CliqueSquare approach.

7 Related work

There is significant effort lately towards managing large volumes of RDF data in cloud environments using different architectures [18]. We classify such works into three distinct categories. The first, and most prominent one includes systems which are solely based on Hadoop and HDFS. The second one contains systems that depend on NoSQL key-value stores as their underlying store, while the third one includes proposals relying on other storage facilities, such as a set of independent single-site RDF stores, or data storage services supplied by the cloud providers. Obviously CliqueSquare belongs to the first category and, for this reason, we elaborate more on related works of this kind.

SHARD [28] was one of the first systems that proposed to use Hadoop and HDFS to store and query RDF data. In SHARD, RDF files provided by the user are simply uploaded and stored in HDFS. Query evaluation is done sequentially by processing one triple pattern at a time. One MapReduce job is used each time for joining one triple pattern with the previous created intermediate results. Query performance of SHARD is very poor with very large response times (in the magnitude of hundreds of minutes for LUBM-6000 on 20 nodes).

One of the state-of-the-art systems built on top of Hadoop, and against which we compare our work, is HadooRDF [16]. In HadoopRDF, RDF data triples are firstly grouped based on their property value. Triples with property `rdf:type` are further grouped based on their object value and then, triples with the same property are further split and grouped based on the RDFS class their object belongs to (if such information exists). Although we use a similar way for grouping triples based on their property, we do so at *each node* after the triples have been disseminated to the appropriate nodes. In HadoopRDF the placement of the triples inside each property file is controlled by HDFS. Query evaluation in HadoopRDF starts by selecting the HDFS files that need to be used for the query processing. Then, a heuristic approach which finds a query plan with the least number of MapReduce jobs is used. However, because HadoopRDF lacks data locality, much data is transferred in the network causing a big overhead during query evaluation as we demonstrated in our experimental evaluation.

As joins are the foundations of SPARQL query evaluation, in [27] the authors propose an intermediate nested algebra whose goal is to maximize the degree of parallelism during join evaluations and reduce the MapReduce cycles. This is achieved by interpreting star-joins as groups of triples (TripleGroups) and defining operators on these TripleGroups. Queries with n star-shaped subqueries are translated into a MapReduce flow with n MR cycles. The proposed

algebra is implemented in a system called RAPID+ which integrates the proposed nested algebra into Pig Latin, a high level language of MapReduce. However, in [27] they only leverage star-shaped joins (subject-subject) and, although not explicitly mentioned in the paper, they discard the case of predicate-subject or predicate-object joins. In addition, RAPID+ also partitions triples based on their property values as in [16]. Thus, their simple partitioning scheme does not allow for co-located joins and thus, even star-shaped queries require a complete MapReduce job.

Another recent work based on Hadoop and HDFS is [40], where the authors propose an RDF-based compression technique that enables an I/O efficient query evaluation suited especially for queries with range and order constraints. We consider this work as complementary to ours as it is focused on reducing I/O cost as opposed to ours which focuses on reducing network traffic.

In the second category of works we find systems that leverage various distributed key-value stores that are available nowadays. Key-value stores are used to index and store RDF triples. For example, Rya [7] uses Apache Accumulo, CumulusRDF [21] uses Apache Cassandra, Stratustore [30] uses Amazon's SimpleDB and H_2RDF [24] uses HBase. However, as key-value stores do not support joins, queries with joins in the above systems are either not allowed at all [21] or are performed in the client side. In [30], joins are performed centralized in one machine and in [7], query rewriting is used and multiple lookups to the key-value store compose the answer to the queries. Finally, in [24] the authors combine the two aforementioned methods together with executing parallel joins with MapReduce jobs depending on the query selectivity. Finally, a recent proposal is Trinity.RDF [39] which is based on a distributed in-memory key-value store designed for generic graphs [29]. Trinity.RDF takes advantage of the graph structure of RDF and evaluates SPARQL queries by exploring the distributed RDF graph in parallel.

[9, 14, 15] belong to the third category. [15] leverages single node RDF stores and Hadoop to parallelize the execution across the multiple nodes. Their main objective is to avoid the use of MapReduce jobs as much as possible, as it causes a lot of overhead, and send the whole query to be answered in parallel in different nodes. To achieve this, they use graph partitioning with replication and query decomposition to split the queries to parallelizable chunks. Although this approach seems suitable for some kinds of queries, data loading (partitioning and placement) is performed in a single machine and requires a large amount of time leading to a non-scalable solution. Similarly in [14, 9] RDF data is partitioned to single node RDF stores with the difference that the partitioning is mostly based on a query workload.

Finally, in [2, 5] a different architecture for storing and querying RDF data in commercial clouds is proposed. In this work, RDF data is stored in a storage service for raw data provided by the cloud provider and indices are built in the key-value store. Indices are used for routing a query to the smallest subset of RDF datasets that most probably contain answers to the query. Then, query evaluation is done by consulting the index to retrieve the appropriate datasets from the storage service, load them in an off-the-self RDF store and evaluate them against this RDF store in a virtual machine. Although the proposed architecture is suitable for very selective queries, it suffers from large query response times for queries that require large intermediate results.

8 Conclusion and Future Work

We presented CliqueSquare, an efficient RDF data management platform built on top of Hadoop for storing and processing large amounts of RDF data. In particular, we proposed an efficient RDF data partitioning strategy that significantly reduces the amount of data transferred through the network. We also proposed a greedy clique-based algorithm for producing query plans that minimize the number of MapReduce stages and exploits partitioning strategy used by CliqueSquare. We experimentally evaluate CliqueSquare using the LUBM benchmark and

compare it with HadoopRDF, a state-of-the-art Hadoop-based framework for big RDF data management. The results show the high superiority of CliqueSquare in terms of query execution times and network traffic. In particular, the results show that CliqueSquare improves HadoopRDF for more than one order of magnitude (it is up to 67 faster in terms of query execution times and up to 91 more efficient in terms of data transfers).

As future work, we plan to follow four main research directions. First, we plan to further evaluate CliqueSquare using more complex queries as well as real datasets and queries. Second, we plan to develop a fault-tolerance strategy that allows CliqueSquare to efficiently recover from node failures. Third, we aim at devising an optimization framework for the query processing performed by CliqueSquare. Finally, we plan to inject RDFS reasoning into CliqueSquare in the form of query reformulation, since our partitioning and query processing framework can be used as is if we consider that RDFS closure has been precomputed using MapReduce-based techniques like the one proposed in [33].

References

[1] D. J. Abadi, A. Marcus, and B. Data. Scalable Semantic Web Data Management using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.

[2] A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu. AMADA: Web Data Repositories in the Amazon Cloud (demo). In *21st International Conference on Information and Knowledge Management (CIKM)*, 2012.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC/ASWC*, 2007.

[4] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[5] F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu. RDF Data Management in the Amazon Cloud. In *Data Analytics in the Cloud (DanaC) Workshop (in conjunction with EDBT)*, 2012.

[6] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, 2005.

[7] A. Crainiceanu, R. Punnoose, and D. Rapp. Rya: A Scalable RDF Triple Store For The Clouds. In *1st International Workshop on Cloud Intelligence (in conjunction with VLDB 2012)*, 2012.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design and Implementation*, 2004.

[9] L. Galarraga, K. Hose, and R. Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. Technical Report: CoRR abs/1212.5636, 2012.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[11] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(1), 2012.

[12] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2-3), 2005.

[13] P. Hayes. RDF Semantics. W3C Recommendation, February 2004. `http://www.w3.org/TR/rdf-mt/`.

[14] K. Hose and R. Schenkel. WARP: Workload-Aware Replication and Partitioning for RDF. In *DESWEB Workshop (in conjunction with ICDE)*, 2013.

[15] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[16] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. on Knowl. and Data Eng.*, 23(9), Sept. 2011.

[17] D. Huynh, S. Mazzocchi, and D. R. Karger. Piggy Bank: Experience the Semantic Web inside your web browser. *J. Web Sem.*, 5(1):16–27, 2007.

[18] Z. Kaoudi and I. Manolescu. Triples in the clouds. In *ICDE seminars*, 2013.

[19] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004.

[20] S. Kotoulas, E. Oren, and F. van Harmelen. Mind the Data Skew: Distributed Inferencing by Speeddating in Elastic Regions. In *WWW*, 2010.

[21] G. Ladwig and A. Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *SSWS*, 2011.

[22] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1), 2010.

[23] M. T. Özsu and P. Valduriez. *Distributed and Parallel Database Systems (3rd. ed.)*. Springer, 2011.

[24] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H_2RDF: Adaptive Query Processing on RDF Data in the Cloud. In *Proceedings of the 21st international conference companion on World Wide Web (demo paper)*, 2012.

[25] E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF. W3C Recommendation, `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

[26] R. Ramakrishnan and J. Gehrke. *Database Management Systems (3rd. ed.)*. McGraw-Hill, 2003.

[27] P. Ravindra, H. Kim, and K. Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC*, pages 46–61, 2011.

[28] K. Rohloff and R. E. Schantz. High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: the SHARD Triple-Store. In *Programming Support Innovations for Emerging Distributed Applications*, 2010.

[29] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.

[30] R. Stein and V. Zacharias. RDF On Cloud Number Nine. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, May 2010.

[31] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A Core of Semantic Knowledge. In *WWW*, 2007.

[32] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for SPARQL. In *EDBT*, pages 324–335, 2012.

[33] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen. Scalable Distributed Reasoning using MapReduce. In *8th International Semantic Web Conference (ISWC)*, 2009.

[34] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 1(1), 2008.

[35] BioPAX: Biological Pathways Exchange. http://www.biopax.org.

[36] Las Vegas Project. http://database.cs.brown.edu/projects/las-vegas/.

[37] RDFizers. http://smile.mit.edu/wiki/RDFizers.

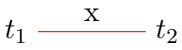
[38] Uniprot RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.

[39] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A Distributed Graph Engine for Web Scale RDF Data. In *PVLDB 2013*.

[40] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *ICDE*, 2013.

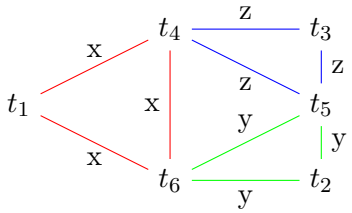
A LUBM Queries

```
SELECT
?x ?y
WHERE {
?x rdf:type ub:GraduateStudent
?x ub:takesCourse ?y
 }
```



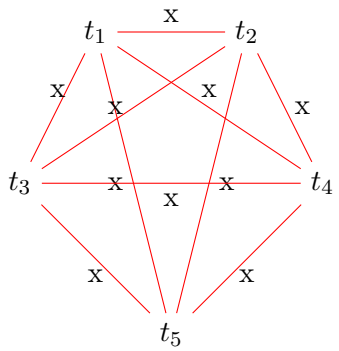
LUBM Query 1 and its variable graph

```
SELECT
?x ?y ?z
WHERE {
?x rdf:type ub:GraduateStudent
?y rdf:type ub:University
?z rdf:type ub:Department
?x ub:memberOf ?z
?z ub:subOrganizationOf ?y
?x ub:undergraduateDegreeFrom ?y
 }
```



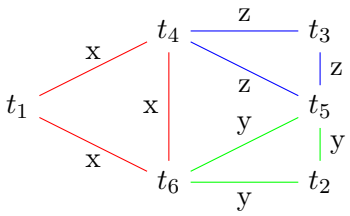
LUBM Query 2 and its variable graph

```
SELECT
?x ?y1 ?y2 ?y3
WHERE {
?x rdf:type ub:FullProfessor
?x ub:worksFor
  <http://www.Department0.University0.edu>
?x ub:name ?y1
?x ub:emailAddress ?y2
?x ub:researchInterest ?y3
 }
```



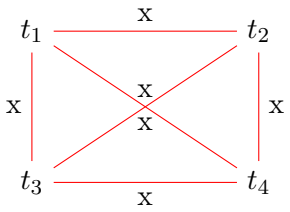
LUBM Query 4 and its variable graph

```
SELECT
?x ?y ?z
WHERE {
?x rdf:type ub:UndergraduateStudent
?y rdf:type ub:FullProfessor
?z rdf:type ub:Course
?x ub:takesCourse ?z
?y ub:teacherOf ?z
?x ub:advisor ?y
 }
```



LUBM Query 9 and its variable graph

```
SELECT
?x ?y ?z ?w
WHERE {
?x rdf:type ub:FullProfessor
?x ub:emailAddress ?y
?w ub:advisor ?x
?x ub:name ?z
 }
```



LUBM Query 15 and its variable graph