How to Deal with Cliques at Work

Benjamin Djahandideh[1] François Goasdoué[2,1] Zoi Kaoudi[3] Ioana Manolescu[1,4]
Jorge-Arnulfo Quiané-Ruiz[5] Stamatis Zampetakis[1,4]
[1] Inria, France firstname.lastname@inria.fr [2] Univ. Rennes 1, France fg@irisa.fr
[3] IMIS, Athena Research Center, Greece zoi@imis.athena-innovation.gr [4] Univ. Paris Sud, France firstname.lastname@lri.fr
[5] Qatar Computing Research Institute (QCRI) jquianeruiz@qf.org.qa

RÉSUMÉ

RDF est un modèle de données à la popularité croissante dans les applications réelles, menant ainsi à la création et l'exploitation de large volumes de données RDF. Des méthodes efficaces de gestion de données RDF sont cruciales pour permettre le passage à l'échelle des applications. Dans cette démonstration, nous mettons en avant la grande efficacité de CliqueSquare, un système de gestion de données RDF conçu au-dessus d'une infrastructure à la MapReduce. Nous montrons trois principaux aspects de CliqueSquare: (i) la réduction significative du traffic réseau pendant l'évaluation de requêtes, (ii) l'évaluation de requêtes potentiellement grandes en peu de jobs MapReduce et (iii) l'amélioration des performances de traitement des requêtes par la production de plans de type DAG. Dans tous les scénarios de démonstration, l'audience est invitée à interagir avec le système pour poser des requêtes, explorer et contrôler les fonctionnalités des algorithmes d'optimisation de la plateforme, et enfin de sélectionner et monitorer l'évaluation des plans de requête dans deux clusters.

1. INTRODUCTION

The *Resource Description Framework* (RDF, in short) [5] is a flexible data model for representing graph-structured data. In a nutshell, an RDF dataset consists of *triples* of the form (s, p, o), stating that a *subject* has a *property* whose value is *object*. Nowadays, many applications use RDF as a first-class citizen or provide support for RDF data, in areas ranging from the Semantic Web and scientific applications, such as BioPAX[1] and UniProt[2], to Web 2.0 platforms, such as RDFizers[3], and databases [2]. The RDF data model is accompanied by the SPARQL standard query language for querying RDF data. Efficient evaluation of SPARQL queries remains challenging though, due to the lack of structure and regularity in RDF graphs, and because SPARQL queries typically involve many joins over the RDF data set.

Many algorithms and architectures have been proposed to efficiently manage RDF data [1, 11, 14]. However, scalability in RDF data management is still an open problem. The main challenge resides in the sheer size of the data itself. Many distributed systems, especially MapReduce-based, have been proposed for RDF data management [6, 7, 8, 10, 13, 17]. However, performance problems still remain, essentially for one of these two reasons: (*i*) they transfer a considerable amount of data through the network or (*ii*) they do not fully exploit the parallelism offered in such an environment by joining intermediary results sequentially, both of which negatively impact query performance. Other systems leverage distributed key-value stores to store and index a large number of RDF triples such as [3, 12, 16]. However, as key-value stores do not support joins, these systems often perform the joins on a single node. Among the above mentioned systems, only [12] leverages MapReduce for queries with low selectivity, while Trinity.RDF [16], based on a distributed in-memory key-value store, evaluates SPARQL queries by navigating the distributed RDF graph and using the results from one triple pattern for the next one.

In this demo, we present CliqueSquare [15], an efficient highly-parallel RDF data management platform, on top of Hadoop, for storing and querying big RDF datasets. CliqueSquare introduces a novel RDF data storage scheme and query optimization algorithm (based on *query variable cliques*, thus the name) which both enable efficient and scalable query evaluation. Generally speaking, CliqueSquare aims at reducing both the number of MapReduce jobs and the amount of data transferred through the network. The demo highlights the following three key features: (*i*) *Data Partitioning* – we show how the data storage provided by CliqueSquare helps to perform many queries without any network traffic; (*ii*) *Job Minimization* – we show the benefits in query performance achieved by CliqueSquare as a result of having as few MapReduce jobs as possible; (*iii*) *Redundancy Exploration through DAG-shaped plans* – we show how DAG-shaped plans may help reduce the intermediary results and thus, lead to better query performance.

2. CLIQUESQUARE

We introduce CliqueSquare, a platform based on Hadoop. CliqueSquare relies on a novel RDF data partitioning scheme and a novel query optimization algorithm. These two components allow CliqueSquare to minimize the number of MapReduce jobs as well as the network traffic. Figure 1 outlines the *data partitioner* and the *query processor*, each of which is briefly outlined in this Section. A comprehensive description of CliqueSquare can be found in [15].

2.1 Data Partitioner

Most distributed file systems replicate input datasets for fault-tolerance reasons (Hadoop by default replicates the

[1] `http://www.biopax.org`
[2] `http://dev.isb-sib.ch/projects/uniprot-rdf/`
[3] `http://smile.mit.edu/wiki/RDFizers`

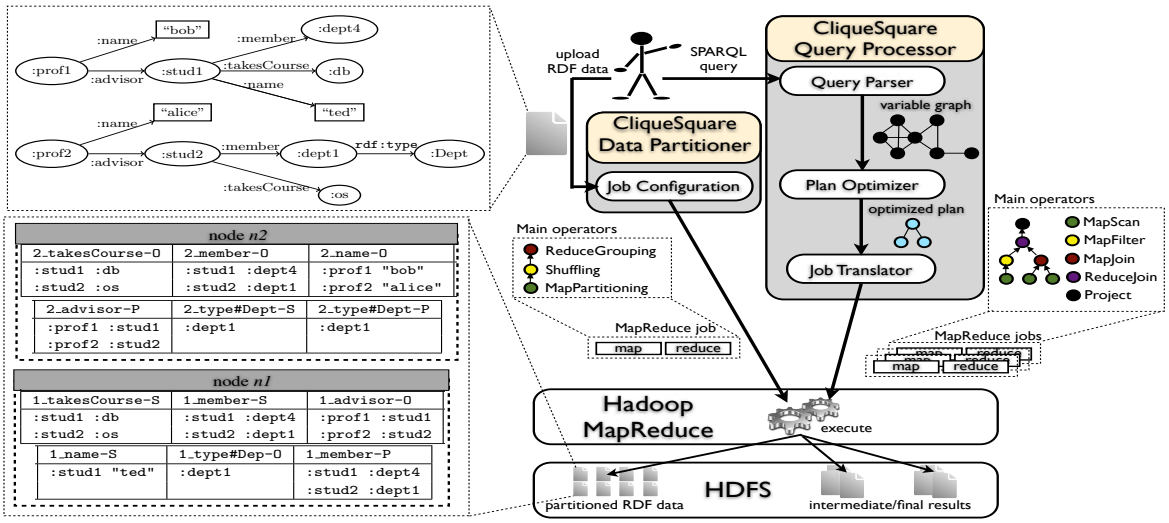


Figure 1: CliqueSquare architecture.

data three times). CliqueSquare exploits this data replication to partition and store RDF data three times, based on the subject, property, and object of triples. Overall, CliqueSquare proceeds to store RDF data in three main steps:

`MapPartitioning.` Triples are partitioned based on their subject, property, and object values, which generates three partitions. At the bottom-left of Figure 1 files with $-S$ extension belong to the subject partition, files with $-P$ to the property partition, and files with $-O$ to the object partition.

`Shuffling.` All partitions corresponding to the *same* value are placed on the same compute node. For instance, all the *stud*1 values, regardless of whether they appear as subjects, objects or properties, are located on node $n1$ (Figure 1). To cope with skew issues that may be raised by over-popular values in the data, CliqueSquare defines a special partition for the `rdf:type` property (which is part of the RDF standard and thus very popular) as in [9], and splits other partitions that are too large based on a predefined threshold [15].

`ReduceGrouping.` Finally, CliqueSquare groups all the subject, respectively property, and object, partitions inside a node based on the property value, and stores each resulting group into HDFS. Looking inside node $n1$ (Figure 1), we observe that we have two files for the property *member*; one based on the subject partition and one based on the object.

2.2 Query Processor

Query processing in CliqueSquare relies on a novel query optimization model and algorithm based on *cliques* in *query variable graphs*. The whole process is illustrated in the upper-right part of Figure 1. Given an incoming *Basic Graph Pattern* (BGP) SPARQL query q, the `Query Parser` outputs a *variable graph* representing q. This graph is then used by the `Plan Optimizer` to produce a logical query plan. Finally, the `Job Translator` transforms the logical plan to a sequence of MapReduce jobs utilizing CliqueSquare operators that can be executed in Hadoop. In the following we focus on the plan optimization procedure.

Variable graph. Besides representing an initial query, CliqueSquare also uses variable graphs to represent *(partially) evaluated queries*, in which some or all joins have been enforced. Formally:

Definition 2.1 (Variable graph). *A* variable graph

```
SELECT
?a ?b ?c ?d ?e
WHERE {
?a p1 ?b (t1)
?a p2 ?c (t2)
?d p3 ?a (t3)
?d p4 ?e (t4)
?l p5 ?d (t5)
?f p6 ?d (t6)
?f p7 ?g (t7)
?g p8 ?h (t8)
?g p9 ?i (t9)
?i p10 ?j (t10)
?j p11 ?k (t11) }
```

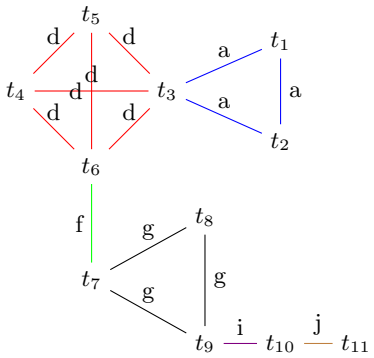


Figure 2: Query Q_1 and its variable graph G_1.

G_V of a BGP query q is a labeled multigraph (N, E, V), where V is the set of variables from q, N is the set of nodes, and $E \subseteq N \times V \times N$ is a set of labeled undirected edges such that: (i) each node $n \in N$ corresponds to a set of triple patterns in q; (ii) there is an edge $(n_1, v, n_2) \in E$ between two distinct nodes $n_1, n_2 \in N$ iff their corresponding sets of triple patterns join on the variable $v \in V$.

Figure 2 shows a query (Q_1) and its variable graph, where every node represents a single triple pattern.

Variable clique. At the core of the CliqueSquare query processing mechanism lies the concept of *variable clique* for representing n-way joins. A variable clique is a set of nodes connected with edges *all labeled by the same variable*. A *maximal clique* comprises all the nodes incident to an edge with the same variable. For example, in variable graph $\mathbf{G_1}$, the *maximal clique* of d is $\{t_3, t_4, t_5, t_6\}$. We term any non-empty subset of a maximal clique as *partial clique*.

Based on variable graphs and cliques, CliqueSquare uses two main operations during its optimization algorithm, namely *clique decomposition* and *clique reduction*.

Clique decomposition. As first step toward building a query plan, CliqueSquare decomposes a variable graph into several cliques. Formally:

Definition 2.2 (Clique decomposition). *Given a variable graph $G_V = (N, E, V)$, a* clique decomposition *of G_V is a set of variable cliques (maximal or partial) of G_V*

which covers all nodes of N, i.e., each node $n \in N$ appears in at least one clique, such that the size of the decomposition is strictly smaller than the number of nodes $|N|$.

For example, one clique decomposition in the variable graph $\mathbf{G_1}$ is $\mathbf{D}_1 = \{\{t_1, t_2, t_3\}, \{t_3, t_4, t_5, t_6\}, \{t_6, t_7\}, \{t_7, t_8, t_9\}, \{t_9, t_{10}\}, \{t_{10}, t_{11}\}\}$; this decomposition follows the distribution of colors on the graph edges in Figure 2. There are more than one ways to decompose the graph; we discuss all alternatives in [15]. From a query optimization perspective, *choosing a clique decomposition corresponds to selecting a set of n-way joins to evaluate.*

Clique reduction. After having identified a clique decomposition, CliqueSquare shrinks the variable graph into a smaller one. We formally define this process as follows:

DEFINITION 2.3 (CLIQUE REDUCTION). *Given a variable graph $G_V = (N, E, V)$ and one of its clique decompositions D, the* reduction of G_V based on D *is the variable graph $G'_V = (N', E', V)$ such that: (i) every clique $c \in D$ correspond to a node $n' \in N'$, whose set of triple patterns is the union of the nodes involved in $c \subseteq N$; (ii) there is an edge $(n'_1, v, n'_2) \in E'$ between two distinct nodes $n'_1, n'_2 \in N'$ iff their corresponding sets of triple patterns join on the variable $v \in V$.*

For example, given the above clique decomposition $\mathbf{D}_1$, CliqueSquare reduces $\mathbf{G_1}$ into a variable graph composed of one node for each maximal clique in $\mathbf{G_1}$ and its corresponding edges. From a query optimization perspective, *clique reduction corresponds to applying the joins identified by the given decomposition.*

CliqueSquare algorithm. CliqueSquare query optimization algorithms develops possible sequences of clique decompositions followed by clique reduction, until all the query predicates have been applied (Algorithm 1). Overall, CliqueSquare takes as an input a variable graph $\mathbf{G}$ of an incoming query and a list of variable graphs *states* modeling the successive evaluation steps that led to $\mathbf{G}$. As a result, CliqueSquare outputs a set of logical query plans QP, each of which is an alternative way to evaluate the incoming query. More in detail, CliqueSquare starts processing the variable graph $\mathbf{G}$ of the initial query, where each node consists of a single triple pattern, and the empty queue *states*. At each recursive call, CLIQUEDECOMPOSITIONS (line 6) returns a set of clique decompositions of $\mathbf{G}$, which is used by CLIQUEREDUCTION (line 8) to reduce $\mathbf{G}$ into $\mathbf{G}'$. $\mathbf{G}'$ is in turn recursively processed, until it consists of a single node. CliqueSquare builds the corresponding logical query plan out of the list of variable graphs comprised in *states* (line 3).

Depending on the clique decomposition alternative that is chosen, eight different instantiations of Algorithm 1 are possible [15]. Each instantiation results in a different plan space. In our demonstration, the user is allowed to choose between four of the most prominent ones and perceive the benefit of our winner algorithm `MSC` (minimum simple covers of partial cliques) in terms of efficiency (short optimization times) and effectiveness (rapidly evaluated plans). For more details on the different algorithms as well as their formal properties we point the reader to [15].

Figure 3 shows the runtimes for HadoopRDF [8] (the state-of-the art Hadoop-based RDF store) and CliqueSquare with two clique decompositions: MSC+ (minimum simple covers of maximal cliques) and MSC. We ran these experiments on 7 nodes of our cluster, detailed in Section 3.1, using five different queries over a 2 billion triples dataset from

Algorithm 1: CliqueSquare optimization algorithm

```
   CLIQUESQUARE (G, states)
   |  Input  : Variable graph G; queue of variable graphs
   |           states
   |  Output: Set of logical plans QP
1  |  states = states ∪ {G};
2  |  if |G| = 1 then
3  |  |  QP ← CREATEQUERYPLANS (states);
4  |  else
5  |  |  QP ← ∅;
6  |  |  D ← CLIQUEDECOMPOSITIONS(G);
7  |  |  foreach D ∈ D do
8  |  |  |  G' ← CLIQUEREDUCTION(G, D);
9  |  |  |  QP ← QP ∪ CLIQUESQUARE (G', states);
10 |  |  end
11 |  end
12 |  return QP;
   end
```

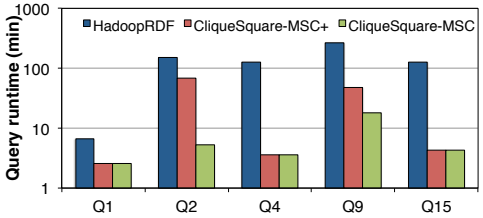


Figure 3: Query evaluation time for LUBM20K.

the LUBM benchmark. We observe in these results that CliqueSquare is more than one order of magnitude faster than HadoopRDF. We also see the high effectiveness of our algorithm MSC, producing plans up to one order of magnitude faster than MSC+ (our second best decomposition algorithm). More experiments are described in [15].

3. DEMONSTRATION

Our goal is to demonstrate CliqueSquare performing a variety of queries with different characteristics and illustrate its efficiency. The audience is invited to interact with the system to compose queries, explore the internals of different optimization algorithms, select and monitor the execution process of plans in two available clusters.

3.1 Setup

We use a cluster of 8 nodes, where each node has: one 2.93GHz Quad Core Xeon processors; 4×4GB of main memory; 2×600GB SAS hard disks configured in RAID 1; 1 Gigabit network; Linux CentOS release 6.4. We divide this cluster into two equal-size clusters to directly compare different techniques of CliqueSquare in some of the scenarios we consider. We use the popular RDF benchmark LUBM [4].

3.2 Demo Scenarios

To better showcase CliqueSquare to the demo attendees, we guide them through three different scenarios where they can see different aspects of CliqueSquare.

(1) Network traffic. We aim at showing how CliqueSquare significantly reduces the amount of data to transfer through the network by exploiting our sophisticated data partitioning scheme. In this scenario, the audience is invited to see how CliqueSquare efficiently executes queries using a map-only job incurring no network traffic at all. For this, the audience can choose to run one out of three predefined queries: one that retrieves all the graduate students and the courses

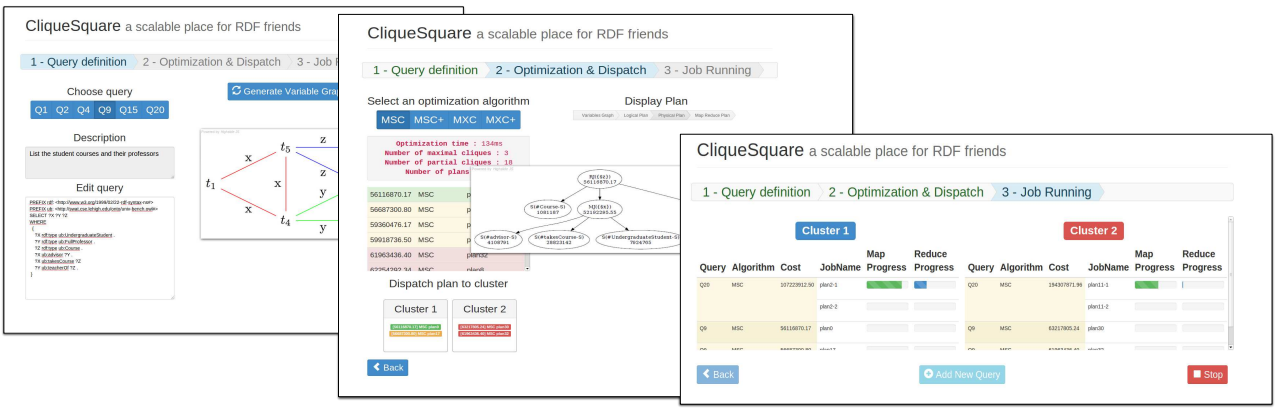


Figure 4: CliqueSquare graphical user interface.

that they participate in (Q1); one query that retrieves the name, the address, and the research interest of every full professor that works in a specific university (Q2); one query extending Q2 by having no restriction on the university (Q3).

(2) Number of MapReduce jobs. We show the audience the importance of plans with few jobs by letting them execute and compare query plans with different numbers of jobs, created by CliqueSquare for the same query. For the latter, the following two queries are provided: a query that returns the undergraduate students whose advisor is an associate professor and teaches a course they take (Q4); one query that asks for the undergraduate students of "University 3", whose advisor is a full professor, the courses they attend, and the email address of their advisor (Q5).

(3) Redundancy exploration. Last but not least, we aim at showing to the audience the benefits in query performance of having in some cases DAG-shaped, rather than tree-shaped, query plans. DAG-shaped plans result into reading at least one relation more than once, justifying the use of term "redundancy". The audience is able to compare the running time of DAG-shaped with tree-shaped query plans. For this, the audience is invited to run one query asking for graduate students of a department which belongs to the university they hold a degree from (Q6).

3.3 Demo Interaction

CliqueSquare comes with an intuitive and interactive graphical user interface to showcase each of the aforementioned scenarios. Figure 4 illustrates its three most important graphical interfaces. As a first step, in the left-side graphical interface, the audience is able to: (*i*) select one of the predefined queries to demonstrate one of the aforementioned scenarios; (*ii*) observe the variable graph CliqueSquare creates for each selected query; (*iii*) modify any of the predefined queries to adapt each demo scenario at will; Then, in the center graphical interface of Figure 4, the audience can: (*iv*) select one or two of the decomposition algorithms to run; (*v*) navigate through different visuals of a query plan (logical, physical, MapReduce plan) (*vi*) select different query plans to study and run; (*vii*) create their own personalized query workloads for realistic evaluations of CliqueSquare; (*viii*) visualize each of the steps of the query optimization process in CliqueSquare; Finally, in the right-side graphical interface, the audience is invited to: (*ix*) run their personalized query workloads; (*x*) monitor the progress of the MapReduce jobs resulting from their choices.

4. REFERENCES

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

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

[3] A. Crainiceanu, R. Punnoose, and D. Rapp. Rya: A Scalable RDF Triple Store For The Clouds. In *Cloud-I Workshop*, 2012.

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

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

[6] K. Hose and R. Schenkel. WARP: Workload-Aware Replication and Partitioning for RDF. In *DESWEB*, 2013.

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

[8] 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.

[9] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In *IEEE CLOUD*, 2010.

[10] H. Kim, P. Ravindra, and K. Anyanwu. From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra (demo). *PVLDB*, 4(12), 2011.

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

[12] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H_2RDF: Adaptive Query Processing on RDF Data in the Cloud. In *In WWW (demo)*, 2012.

[13] 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.

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

[15] Work under submission. Available at http://cloak.saclay.inria.fr/CliqueSquareExt.pdf, 2014.

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

[17] X. Zhang, L. Chen, and M. Wang. Towards Efficient Join Processing over Large RDF Graph Using MapReduce. In *SSDBM*, 2012.