CliqueSquare in Action: Flat Plans for Massively Parallel RDF Queries

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* first.last@inria.fr [2]*U. Rennes 1, France,* fg@irisa.fr
[3]*IMIS, Athena Research Center, Greece* zoi@imis.athena-innovation.gr [4]*U. Paris Sud, France* first.last@lri.fr
[5]*Qatar Computing Research Institute (QCRI), Qatar* jquianeruiz@qf.org.qa

***Abstract*—RDF is an increasingly popular data model for many practical applications, leading to large volumes of RDF data; efficient RDF data management methods are crucial to allow applications to scale.**

We propose to demonstrate CliqueSquare, an RDF data management system built on top of a MapReduce-like infrastructure. The main technical novelty of CliqueSquare resides in its logical query optimization algorithm, guaranteed to find a *logical plan as flat as possible* for a given query, meaning: a plan having the smallest possible number of join operators on top of each other. CliqueSquare's ability to build flat plans allows it to take advantage of a parallel processing framework in order to shorten response times. We demonstrate loading and querying the data, with a particular focus on query optimization, and on the performance benefits of CliqueSquare's flat plans.

I. Introduction

The *Resource Description Framework* (RDF, in short) [1] 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 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]. The RDF data model is accompanied by the SPARQL query language. The efficient evaluation of SPARQL queries is difficult though, due to the lack of structure and regularity in RDF datasets, and because SPARQL queries typically involve many joins between triple patterns.

Many algorithms and architectures have been proposed to efficiently manage RDF data [2], [3], [4]. However, scaling RDF query processing to very large data volumes is challenging. Prior research has also led to various distributed RDF systems, in particular based on MapReduce [5]. Some of these systems, such as [6], [7], have placed an important emphasis on the data partitioning process, with the goal of making the evaluation of certain shapes of queries *parallelizable without communications* (or *PWOC*, in short). In a nutshell, a PWOC query for a given data partitioning can be evaluated by taking the union of the query results obtained on each node.

However, it is easy to see that no single partitioning can guarantee that *all* queries are PWOC; in fact, most queries do require processing across multiple nodes and thus, data shuffling across nodes. The more complex the query is, the bigger will be the impact of evaluating the distributed part of the query plan. *Logical query optimization* – deciding how to decompose and evaluate an RDF query in a massively parallel context – has thus also a crucial impact on performance. As it is well-known in distributed data management [8], to efficiently evaluate queries one should maximize *parallelism* (both *inter-operator* and *intra-operator*) to take advantage of the distributed processing capacity and thus reduce the response time.

In a parallel RDF query evaluation setting, intra-operator parallelism relies on join operators that process chunks of data in parallel. To increase inter-operator parallelism one should aim at building *massively-parallel (flat) plans*, having as few (join) operators as possible on any root-to-leaf path in the plan; this is because the processing performed by such joins directly adds up into the response time. Prior works use binary joins organized in bushy plans [9], n-ary joins (with $n > 2$) only in the first level of the plans and binary joins in the next levels [6], [7], or n-ary joins at all levels [10] but organized in left-deep plans. Such methods lead to high (non-flat) plans and hence high response times. HadoopRDF [11] is the only one building bushy plans of n-ary joins, but it cannot guarantee the plan is as flat as possible.

In this demo we present CliqueSquare, a distributed RDF data management platform with a particular focus on *the logical optimization* of RDF queries, seeking to build *flat query plans* composed of *n-ary (star) equality joins*. In [12] we show that CliqueSquare's optimizer is *guaranteed* to build some of the flattest possible plans for any input query.

The benefits of flat plans can be combined with many orthogonal optimizations: e.g., the RDF partitioning model, the RDF storage and processing facilities on each node, the degree of parallelism of join evaluation as in [13]. Going beyond RDF, flat plans are beneficial in any conjunctive query processing setting based on n-ary (star) equality joins. However, flat plans are of particular interest for RDF, since (as also noted in [3], [14], [15]) RDF queries tend to involve more joins than a relational query computing the same result. This is because relations can have many attributes, whereas in RDF each query triple pattern has only three, leading to syntactically more complex queries.

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

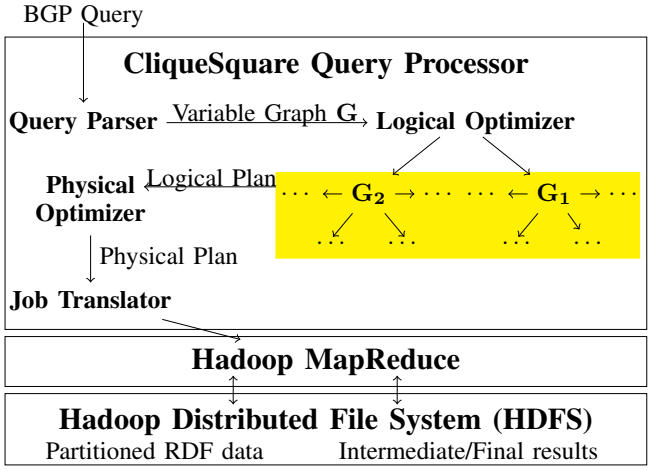

Fig. 1. Query evaluation workflow.

This demo will highlight the logical query optimization of CliqueSquare. The audience will explore the internals of different optimization algorithms by selecting and monitoring in real time the execution process of different plans.

II. CliqueSquare query processing

Figure 1 shows the query evaluation steps in CliqueSquare: The `Query Parser` takes a *Basic Graph Pattern* (BGP) query q, i.e., a conjunctive SPARQL query, as input and produces a *variable graph* $\mathbf{G}$ representing q. The `Logical Optimizer` explores a set of logical plans based on *graph decompositions* of this variable graph $\mathbf{G}$, as we explain shortly. Then, the `Physical Optimizer` translates a logical plan into a physical plan based on a cost model. Finally, for a physical plan, the `Job Translator` builds a sequence of MapReduce jobs whose execution is delegated to Hadoop. We explain these three main components of CliqueSquare in the following.

A. Logical Optimizer

CliqueSquare optimizes queries based on variable graphs, which it uses to encode both the *incoming query* and *intermediary query representations* that it builds as it progresses toward obtaining logical query plans. Formally:

Definition 2.1 (Variable graph): A *variable graph* 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 a set of *nodes* such that each $n \in N$ corresponds to a set of triple patterns in q; and $E \subseteq N \times V \times N$ is a set of labeled undirected edges. 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. In this example, every node represents a single triple pattern. Given a variable graph, CliqueSquare first identifies *variable cliques* within the variable graph. A variable clique is a set of nodes connected among themselves with edges that are all labeled by the same variable. We denote the set of *all* the nodes incident to an edge with the same variable as a *maximal* variable clique. For example, in variable graph $\mathbf{G_1}$, the *maximal clique* of x is $\{t_1, t_4, t_6\}$. We term any non-empty subset of a maximal clique as *partial clique*. Next, based on variable graphs and cliques, CliqueSquare uses two main operations during its optimization algorithm: *clique decomposition* and *clique reduction*.

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

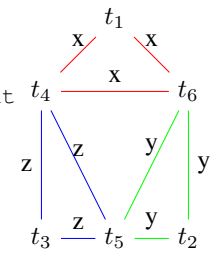

Fig. 2. Query Q_1 and its variable graph $\mathbf{G_1}$.

Clique decomposition. Intuitively, a clique decomposition is a way to "cover" the query with variable cliques; each clique corresponds to an n-way logical join on intermediary query results. 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 (partial or maximal) variable cliques 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 $\mathbf{D}_1 = \{\{t_1, t_4, t_6\}, \{t_3, t_4, t_5\}, \{t_5, t_6, t_2\}\}$; this decomposition follows the distribution of colors on the graph edges in Figure 2. Another decomposition for $\mathbf{G_1}$ is: $\mathbf{D}_2 = \{\{t_1, t_6\}, \{t_3, t_4\}, \{t_2, t_5\}\}$ etc.

It is worth noting that a variable graph has many decompositions. One may use partial cliques, or only maximal ones; we may seek *exact* covers (where each node appears in only one clique) or *simple* covers (where a node may be part of several cliques). Furthermore, we say a clique decomposition for a given graph is *minimum* if it has the lowest possible number of cliques; we may consider only minimum cliques, or also non-minimum ones.

Clique reduction. Given a clique decomposition, the optimizer then applies a *clique reduction*: this corresponds to shrinking the variable graph by collapsing all the nodes from every clique of the decomposition, into a single node. Formally:

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 of three nodes, one for each clique in $\mathbf{G_1}$.

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 ← CREATEQUERYPLAN (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
```

Query optimization. The CliqueSquare *query optimization algorithm* (Algorithm 1) develops from the initial query graph, possible sequences of clique decompositions followed by clique reductions, until all the query predicates have been applied. The algorithm takes as an input a variable graph $\mathbf{G}$ and a list of variable graphs $states$, modeling the successive evaluation steps that led to $\mathbf{G}$. The algorithm outputs a set of logical query plans QP, each of which is an alternative way to evaluate the incoming query. In the initial query graph $\mathbf{G}$, each node consists of a single triple pattern, and $states$ is empty. 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. The optimizer builds the corresponding logical query plan out of the list of variable graphs comprised in $states$ using CREATEQUERYPLAN function. A logical query plan p is a rooted directed acyclic graph (DAG) whose nodes can either scan triples from the store (*Match* operators), or be selections, projections, or n-way equi-joins.

Notice that, depending on the chosen clique decomposition method (maximal or partial cliques, exact or simple covers, restricted to minimum covers or not), eight different instantiations of Algorithm 1 are possible [12]. We have shown that the most interesting are those called MSC (*m*aximal *s*imple *c*overs), MSC+ (the same, restricted to minimum covers) and MXC (*m*aximal e*x*act *c*overs) are *guaranteed to find some of the flattest possible plans*, while exploring plan set of reasonable size and giving interesting plans to choose from, within a reasonable optimization time.

B. Physical Optimizer

The physical optimizer translates a logical plan into a physical one taking into account the physical storage (partitioning) of the data within the Hadoop Distributed File System (HDFS), and the available physical operators.

CliqueSquare storage. We use a simple partitioning scheme which allows all first level joins to be evaluated locally at each node (PWOC, also termed co-located joins), in order to reduce query response time. Our partitioner exploits the fact that most of the existing distributed file systems replicate a dataset at least three times for fault-tolerance reasons. Thus, we store three full partitions of the RDF data, each organized differently, and group the triples at each system node to enable fine-granularity data access. RDF data is stored in three steps:
(1) We place each triple in three partitions, based on its subject, property and object values. Triples with the same value of s, p, or o are located within the same compute node.
(2) Then, we partition triples within each compute node based on their placement (s, p, or o) attribute. We call these partitions subject, property, and object partition. Notice that given a type of join, e.g., subject-subject join, this local partitioning leads to directly reading the relevant triples only.
(3) We further split each partition within a compute node by the value of the property in their triples[4]. This property-based grouping resembles the well-known vertical RDF partitioning proposed for centralized RDF stores. Finally, we store each resulting partition into an HDFS file.

CliqueSquare physical operators. CliqueSquare relies on the following set of physical operators: (a) *Map Scan*; (b) *Filter*; (c) *Map Shuffler*; (d) *Map Join*; (e) *Reduce Join*; (f) *Project*. The translation from a logical plan to a physical plan is almost 1 to 1. An exception is the logical join operator which is translated either to a *Map Join*, if the join can be performed locally, or to a *Reduce Join* in any other case. As a *Reduce Join* cannot be performed directly on the output of another reduce join, a map shuffler operator is added, if needed.

C. Job Translator

The physical plan is mapped to a MapReduce programs as follows: (i) projections and filters are always part of the same MapReduce task as their parent operator; (ii) map joins along with all their ancestors are executed in the same (map or reduce) task; (iii) any other operator is executed in a task of its own. Then, the MapReduce tasks are grouped in MapReduce jobs in a bottom-up traversal of the task tree; each job has at least one map task, and zero or more reduce tasks.

III. DEMONSTRATION

During the demo, 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.

A. 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, and Linux CentOS release 6.4. We divide this cluster into two equal-size sub-clusters to directly compare different CliqueSquare's techniques in the following scenarios.

B. Demo Scenarios

We showcase the system by walking the demo attendees through the following scenarios based on the popular LUBM benchmark [16], featuring students, courses, universities etc.

[4]Special treatment is given to the rdf:type property; for details, see [12].

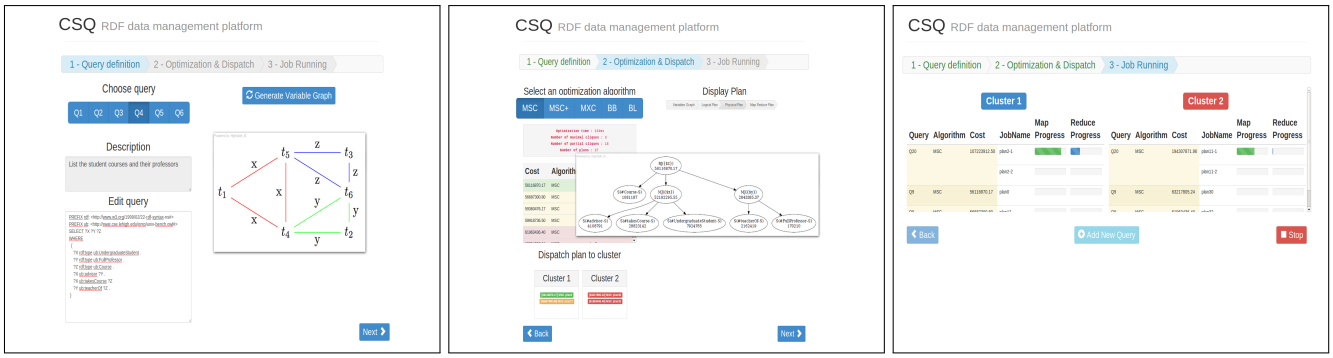

Fig. 3. Snapshots of the CliqueSquare graphical user interface.

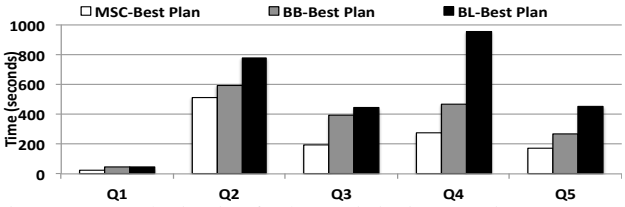

Fig. 4. Query evaluation time for three optimization strategies on LUBM10K.

1) Scenario 1: one-clique queries: First, we consider the following queries: (Q1) retrieves all the graduate students and their courses; (Q2) retrieves the name, the address, and the research interest of every full professor in a specific university; (Q3) is similar to Q2 but does not specify the university. These queries have only one variable clique, thus the optimization algorithm solves them with a single n-ary join and thus there is only one plan. Thanks to CliqueSquare's partitioning scheme, the queries run in a single map-only job.

2) Scenario 2: complex queries: Three queries are used. (Q4) returns the undergraduate students whose advisor is an associate professor and teaches a course they take. (Q5) requests (i) the undergraduate students of "University 3" whose advisor is a full professor, (ii) the courses they attend, and (iii) their advisors' email address. (Q6) asks for graduate students of a department, which belongs to the university from which the students hold a degree from. Answering these complex queries requires multiple jobs, whose exact number depends on the flatness of the plan. CliqueSquare's MSC optimization algorithm is guaranteed to find some of the flattest plans possible, thus leading to short response time even for such complex queries.

As to the system performance, Figure 4 (taken from [12]) shows the execution times for 5 representative queries, of the best plan generated with the CliqueSquare-MSC algorithm, together with the with BB (the best binary bushy plan) and BL (the best binary linear plan). We used the demo's proposed cluster set-up, and a 1 billion triples LUBM dataset. The results show the superiority of CliqueSquare-MSC in all cases.

C. Demo Interaction

CliqueSquare comes with an interactive graphical user interface, illustrated by the snapshots in Figure 3. Specifically, through the GUI, the leftmost snapshot in the Figure shows how a demo attendee may: (i) select and/or modify one of the predefined queries; (ii) inspect the initial variable graph created by CliqueSquare for each such query. The central snapshot demonstrates how users may (iii) select one clique decomposition option to use with CliqueSquare – among MSC, MSC+ and MXC – or the BB or BL algorithms; (iv) visualize the logical and physical plan built, as well as the resulting MapReduce script; (v) select a plan and explore all variable graphs produced by the recursive calls of the Algorithm 1 understanding how the logical plan is constructed from the subsequent graphs; (vi) review the estimated costs for the plans and manually select one to run on each cluster. Finally, the rightmost snapshot shows how demo attendees can monitor the progress of the MapReduce programs computing query results; in particular this will allow seeing the benefits of flat plans for reduced response times.

REFERENCES

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

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

[3] T. Neumann and G. Weikum, "The RDF-3X Engine for Scalable Management of RDF Data," *VLDBJ*, vol. 19, no. 1, 2010.

[4] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *PVLDB*, vol. 1, no. 1, 2008.

[5] Z. Kaoudi and I. Manolescu, "RDF in the Clouds: A Survey," The VLDB Journal, 2014.

[6] J. Huang, D. J. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, 2011.

[7] K. Lee and L. Liu, "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *PVLDB*, vol. 6, no. 14, Sep. 2013.

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

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

[10] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris, "H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs," in *IEEE BigData*, 2013.

[11] 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 TKDE*, vol. 23, no. 9, Sep. 2011.

[12] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis, "CliqueSquare: Flat Plans for Massively Parallel RDF Queries," in *ICDE*, 2015.

[13] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: a Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing," in *SIGMOD*, 2014.

[14] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View selection in semantic web databases," *PVLDB*, vol. 5, no. 1, 2012.

[15] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz, "Heuristics-based query optimisation for SPARQL," in *EDBT*, 2012.

[16] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *J. Web Sem.*, vol. 3, no. 2-3, 2005.