CliqueSquare: Flat Plans for Massively Parallel RDF Queries

François Goasdoué[1,2], Zoi Kaoudi[3*], Ioana Manolescu[2,4], Jorge-Arnulfo Quiané-Ruiz[5], Stamatis Zampetakis[2,4]

[1]*U. Rennes 1, France,* fg@irisa.fr [2]*INRIA, France* first.last@inria.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*—As increasing volumes of RDF data are being produced and analyzed, many massively distributed architectures have been proposed for storing and querying this data. These architectures are characterized first, by their RDF partitioning and storage method, and second, by their approach for distributed query optimization, i.e., determining which operations to execute on each node in order to compute the query answers.**

We present CliqueSquare, a novel optimization approach for evaluating conjunctive RDF queries in a massively parallel environment. We focus on reducing query response time, and thus seek to build *flat* plans, where the number of joins encountered on a root-to-leaf path in the plan is minimized. We present a family of optimization algorithms, relying on n-*ary (star) equality joins* to build flat plans, and compare their ability to find the flattest possibles. We have deployed our algorithms in a MapReduce-based RDF platform and demonstrate experimentally the interest of the flat plans built by our best algorithms.

I. Introduction

The Resource Description Framework (RDF) [1] is a flexible data model introduced for the Semantic Web. RDF is currently used in a broad spectrum of applications ranging from the Semantic Web [2], [3] and scientific applications (e.g., BioPAX[1], Uniprot[2]) to Web 2.0 platforms [4] and databases [5]. While its query language, SPARQL[3], comprises many powerful features such as aggregation and optional clauses, the most frequently used dialect is that of conjunctive queries, a.k.a. Basic Graph Pattern queries (or BGP, in short), typically featuring many equality joins.

Given the popularity of RDF, large volumes of RDF data are created and published, in particular in the context of the Linked Data movement. Thus, distributing the data and the computations across several nodes has been investigated in prior research, which has lead to large-scale, distributed systems for storing and querying RDF data [6]. Conceptually, each RDF database can be seen as a directed labelled graph. Thus, building a distributed RDF database requires addressing two main issues: how to distribute the graph data across the nodes; and how to split the query evaluation across the nodes.

Clearly, data distribution has an important impact on query performance. Many previous works on distributed RDF query evaluation, such as [7], [8], [9], [10], have placed an important emphasis on the data partitioning process (workload-driven in the case of [9], [10]), 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 redistribution across nodes, a.k.a. shuffling. 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 [11], 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 have 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 [7], [8], [10], or n-ary joins at all levels [12] but organized in left-deep plans. Such methods lead to high (non-flat) plans and hence high response times. HadoopRDF [13] is the only one building bushy plans of n-ary joins, but it cannot guarantee a plan as flat as possible.

In this paper, we focus on *the logical query optimization* of BGP queries, seeking to build *flat* query plans composed of *n-ary (star) equality joins*. Flat plans are most likely to lead to shorter response time in distributed/parallel settings, such as in MapReduce-like systems. The core of our study, thus, is independent of (and orthogonal to): the chosen partitioning model; storage and query facilities on each node; physical join algorithms; increasing the parallelism of join evaluation as in [14]; and the cost model characterizing execution performance. For validation, we implement concrete choices along each of these dimensions, but other options can be combined with our optimization algorithms to improve the

*Work partially done while the author was at INRIA.
[1]http://www.biopax.org
[2]http://www.uniprot.org/
[3]http://www.w3.org/TR/rdf-sparql-query/

overall performance of parallel RDF query evaluation.

Contributions. We present CliqueSquare, a novel approach for the logical optimization of BGP queries over large RDF graphs distributed in a massively parallel environment, such as MapReduce. We make the following contributions:

(1) We describe a search space of logical plans obtained by relying on *n-ary (star) equality joins*. The interest of such joins is that by aggressively joining many inputs in a single operator, they allow building flat plans.

(2) We provide a novel generic algorithm, called CliqueSquare, for exhaustively exploring this space, and a set of three algorithmic choices leading to eight variants of our algorithm. We present a thorough analysis of these variants, from the perspective of their ability to find one of (or all) the flattest possible plans for a given query. We show that the variant we call CliqueSquare-MSC is the most interesting one, because it develops a reasonable number of plans and is guaranteed to find some of the flattest ones.

(3) We have fully implemented our algorithms and validate through experiments their practical interest for evaluating queries on very large distributed RDF graphs. For this, we rely on a set of relatively simple parallel join operators and a generic RDF partitioning strategy, which makes no assumption on the kinds of input queries. We show that CliqueSquare-MSC makes the optimization process efficient and effective even for complex queries leading to robust query performance.

It is worth noting that our findings in (1)-(2) are not specific to RDF, but apply to any conjunctive query processing setting based on n-ary (star) equality joins. However, they are of particular interest for RDF, since (as noted e.g., in [15], [16], [17]) 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 atom has only three, leading to syntactically more complex queries.

The paper is organized as follows. We cover the necessary background and state-of-the-art in Section II. Section III introduces the logical model used in CliqueSquare for queries and query plans and describes our generic logical optimization algorithm. In Section IV, we present our algorithm variants, their search spaces, and analyze them from the viewpoint of their ability to produce flat query plans. Section V shows how to translate and execute our logical plans to MapReduce jobs, based on a generic RDF partitioning strategy. Section VI experimentally demonstrates the effectiveness and efficiency of our logical optimization approach and Section VII concludes our findings.

II. Background and state-of-the-art

We briefly recall RDF and SPARQL, before discussing works closely related to our query optimization approach.

RDF and SPARQL. RDF data is organized in *triples* of the form $(s\ p\ o)$, stating that the subject s has the property (a.k.a. predicate) p whose value is the object o. *Unique Resource Identifiers* (URIs) are central in RDF: one can use URIs in any position of a triple to uniquely refer to some entity or concept. Notice that literals (constants) are also allowed in the o position. Formally, given two disjoint sets of URIs and literals[4], U and L, a well-formed *triple* is a tuple $(s\ p\ o)$ from $U \times U \times (U \cup L)$. RDF admits a natural graph representation, with each $(s\ p\ o)$ triple seen as an p-labeled directed edge from the node identified by s to the node identified by o. A set of triples, i.e., an RDF dataset, is called an RDF graph.

SPARQL is the W3C standard for querying RDF graphs. We consider the BGP dialect 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 generalized to that of *triple pattern* $(s\ p\ o)$ from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$, 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. We consider BGP queries *with no cartesian products ($\times$)*. One can simply decompose a query with a cartesian product in $\times$-free subqueries, process them independently, and combine their results at the end.

The *evaluation* of a BGP query q: $\texttt{SELECT}\ ?v_1 \cdots ?v_m\ \texttt{WHERE}\ \{t_1 \cdots t_n\}$ on an RDF graph G is: $eval(q) = \{\mu(?v_1 \cdots ?v_m) \mid \mu\colon var(q) \rightarrow val(G)$ is a function s.t. $\{\mu(t_1), \cdots, \mu(t_n)\} \subseteq G\}$, with $var(q)$ the set of variables in q, $val(G)$ the set of URIs and literals occurring in G, and μ a function replacing any variable 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 of the triple pattern t_i according to μ.

Centralized RDF query optimization. Centralized RDF databases such as RDF-3X [15] typically rely on a Dynamic Programming (DP) algorithm to produce logical plans. This may lead to large plan spaces and thus long optimization time for large SPARQL queries. In more recent works such as [17], DP is avoided and plans are heuristically built relying solely on the shape of the query, without using cardinality estimations etc. In [18], a SPARQL query is decomposed into chain and star subqueries, and DP is applied on each subquery. Overall, designed for a centralized context, these approaches build only binary logical plans, and do not guarantee *flat* plans. As our experiments show, flat bushy plans built with n-ary joins bring important performance advantages in a parallel environment.

MapReduce-based query optimization. Many recent massively parallel data management systems leverage MapReduce in order to build scalable query processors for both relational [19] and RDF [6] data.

Early works on relational data mainly focus on selection and projection push-down [20], while [21] relies on other classical distributed database techniques [11]. The authors in [22] propose a cost-based approach for deciding how to split a query into a set of *fragments*; they use an n-ary repartition join [23] to join each fragment. Then, the authors consider

[4]RDF allows some form of incomplete information through *blank nodes*, standing for unknown constants or URIs. All our results apply in the presence of blank nodes; we omit them from the presentation for simplicity.

possible ways to combine the fragment results through *binary* joins. They consider both left-deep and bushy plans, and avoid a very large search space by cost-based pruning. In contrast with [22], our approach relies on n-ary joins at all levels and hence it develops some logical plans that [22] does not.

Most MapReduce-based RDF engines mainly focus on improving data access for optimizing query performance. Data access performance depends on how the data is partitioned across nodes and the data layout on each node (e.g., key-value representation, column layout, indexes). Previous works have focused on RDF data partitioning strategies, such as [7], [8], [24], [9], [10], with the goal of making the *first-level* joins (those applied directly on the input data) PWOC. Independently, aggressive indexing and compression of RDF data has been studied in [12]. However, none of these works focus on the logical query optimization nor on fully exploiting parallelism during query evaluation.

The performance of the joins *after* the first-level ones is determined by (i) the available physical operators, and (ii) the join plan built by the optimizer. Prior works have 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 [7], [8], [10], or n-ary joins at all levels [12] but organized in left-deep plans. Such methods lead to high (non-flat) plans and hence longer response times. HadoopRDF is the only one proposing some heuristics to produce flat plans [13], but it has two major disadvantages: (i) it produces a single plan that can be inefficient; (ii) it does not guarantee that the plan will be as flat as possible.

In this work, we focus on the *logical optimization of BGP queries* for massively parallel environments. In contrast to prior work, we use *n-ary star equi-joins* at all the levels of a query plan; we provide algorithms guaranteed to find at least some of *the flattest possible plans*. We show experimentally that our plans lead to efficient query evaluation even for large, complex queries.

III. Logical query model

This section describes the CliqueSquare approach for processing queries based on a notion of *query variable graphs*. We introduce these graphs in Section III-A and present the CliqueSquare optimization algorithm in Section III-B.

A. Query model

We model a SPARQL BGP query as a set of n-ary relations connected by joins. Specifically, we rely on a *variable (multi)graph representation*, inspired from the classical relational *Query Graph Model* (QGM[5]), and use it to represent incoming queries, as well as intermediary query representations that we build as we progress toward obtaining logical query plans. Formally:

Definition 3.1 (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 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 $v \in V$.

[5]http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp

```
SELECT ?a ?b
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 "C1"}
```

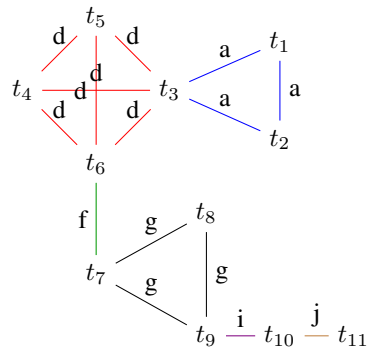


Fig. 1. Query Q_1 and its variable graph $\mathbf{G_1}$.

Figure 1 shows a query and its variable graph, where every node represents a single triple pattern. More generally, one can also use variable graphs to represent *(partially) evaluated queries*, in which some or all the joins of the query have been enforced. A node in such a variable graph corresponds to a *set of triple patterns* that have been joined on their common variables, as the next section illustrates.

B. Query optimization algorithm

The CliqueSquare process of building logical query plans starts from *the query variable graph* (where every node corresponds to a single triple pattern), treated as an *initial state*, and repeatedly applies *transformations* that decrease the size of the graph, until it is reduced to only one node; a one-node graph corresponds to having applied all the query joins. On a given graph (state), several transformations may apply. Thus, there are many possible sequences of states going from the query (original variable graph) to a complete query plan (one-node graph). Out of each such sequence of graphs, CliqueSquare creates a logical plan. In the sequel of Section III, we detail the graph transformation process, and delegate plan building to Section IV.

Variable cliques. At the core of query optimization in CliqueSquare lies the concept of *variable clique*, which we define as a set of variable graph nodes connected with edges having a certain label. Intuitively, *a clique corresponds to an n-ary (star) equi-join*. Formally:

Definition 3.2 (Maximal/partial variable clique): Given a variable graph $G_V = (N, E, V)$, a *maximal (resp. partial) clique* of a variable $v \in V$, denoted $c\ell_v$, is the set (resp. a non-empty subset) of *all* nodes from N which are incident to an edge $e \in E$ with label v.

For example, in the variable graph $\mathbf{G_1}$ of query Q_1 (see Figure 1), the maximal variable clique of d, $c\ell_d$ is $\{t_3, t_4, t_5, t_6\}$. Any non-empty subset is a partial clique of d, e.g., $\{t_3, t_4, t_5\}$.

Clique Decomposition. The first step toward building a query plan is to decompose (split) a variable graph into several cliques. From a query optimization perspective, *clique decomposition corresponds to identifying partial results to*

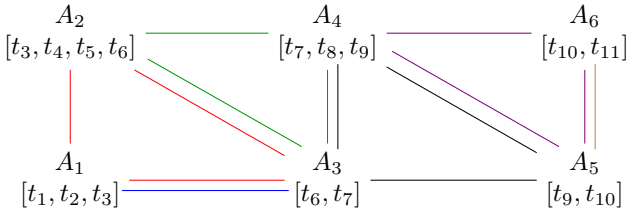


Fig. 2. Clique reduction $\mathbf{G_2}$ of Q_1's variable graph (shown in Figure 1).

be joined, i.e., for each clique in the decomposition output, exactly one join will be built. Formally:

Definition 3.3 (Clique decomposition): Given a variable graph $G_V = (N, E, V)$, a *clique decomposition* $\mathcal{D}$ 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 $|\mathcal{D}|$ is strictly smaller than the number of nodes $|N|$.

Consider again our query Q_1 example in Figure 1. One clique decomposition is $d_1 = \{\{t_1, t_2, t_3\}, \{t_3, t_4, t_5, t_6\}, \{t_6, t_7\}, \{t_7, t_8, t_9\}, \{t_9, t_{10}\}, \{t_{10}, t_{11}\}\}$; this decomposition follows the distribution of colors on the graph edges in Figure 1. A different decomposition is for instance $d_2 = \{\{t_1, t_2\}, \{t_3, t_4, t_5\}, \{t_6, t_7\}, \{t_8, t_9\}, \{t_{10}, t_{11}\}\}$; indeed, there are many more decompositions. We discuss the space of alternatives in the next section.

Observe that we do not allow a decomposition to have *more* cliques than there are nodes in the graph. This is because a decomposition corresponds to a step forward in processing the query (through its variable graph), and this advancement is materialized by the graph getting strictly smaller.

Based on a clique decomposition, the next important step is *clique reduction*. From a query optimization perspective, *clique reduction corresponds to applying the joins identified by the decomposition*. Formally:

Definition 3.4 (Clique Reduction): Given a variable graph $G_V = (N, E, V)$ and one of its clique decompositions $\mathcal{D}$, the *reduction of G_V based on $\mathcal{D}$* is the variable graph $G'_V = (N', E', V)$ such that: (i) every clique $c \in \mathcal{D}$ corresponds 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 query Q_1 in Figure 1 and the above clique decomposition d_1, CliqueSquare reduces its variable graph $\mathbf{G_1}$ into the variable graph $\mathbf{G_2}$ shown in Figure 2. Observe that in $\mathbf{G_2}$, the nodes labeled A_1 to A_8 each correspond to several triples from the original query: A_1 corresponds to three triples, A_2 to four triples, etc.

CliqueSquare algorithm. Based on the previously introduced notions, the CliqueSquare query optimization algorithm is outlined in Algorithm 1. CliqueSquare takes as an input a variable graph G corresponding to the query with *some* of the predicates applied (while the others are still to be enforced), and a list of variable graphs $states$ tracing the sequence of transformations which have lead to G, starting from the original query variable graph. The algorithm outputs a set of logical query plans QP, each of which encodes an alternative way to evaluate the query.

Algorithm 1: CliqueSquare algorithm

```
1  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
```

The initial call to CliqueSquare is made with the variable graph 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 G. Each decomposition is used by cliqueReduction (line 8) to reduce G into the variable graph G', where the n-ary joins identified by the decomposition have been applied. G' is in turn recursively processed, until it consists of a single node. When this is the case (line 2), CliqueSquare builds the corresponding logical query plan out of $states$ (line 3), as we explain in the next section. The plan is added to a global collection QP, which is returned when all the recursive calls have completed.

IV. Query Planning

We describe CliqueSquare's logical operators, plans, and plan spaces (Section IV-A) and how logical plans are generated by Algorithm 1 (Section IV-B). We then consider a set of alternative concrete clique decomposition methods to use within the CliqueSquare algorithm, and describe the resulting search spaces (Section IV-C). We introduce plan *height* to quantify its flatness, and provide a complete characterization of the CliqueSquare algorithm variants w.r.t. their ability to build the flattest possible plans (Section IV-D).

A. Logical CliqueSquare operators and plans

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, we take Val to be a subset of $U \cup L$, and $A = var(tp)$ to be the set of variables occuring in a triple pattern tp, $A \subseteq V$. Every mapping $\mu(tp)$ from $A = var(tp)$ into $U \cup L$ leads to a tuple in a relation which we denote R_{tp}. To simplify presentation and without loss of generality, we assume $var(tp)$ has only those tp variables which participate in a join.

We consider the following logical operators, where the output attributes are identified as $(a_1, \ldots, a_n)$:

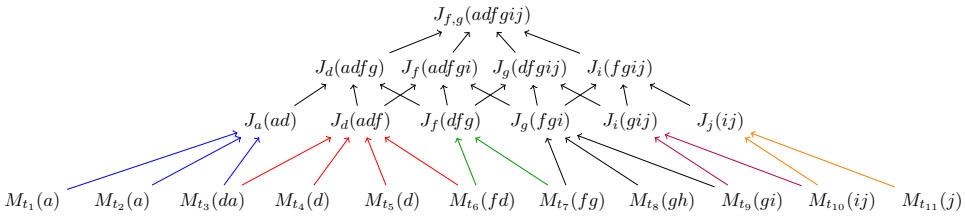


Fig. 3. Sample logical plan built by CliqueSquare for Q1 (Figure 1).

- **Match**, $M_{tp}(a_1, \ldots, a_n)$, is parameterized by triple pattern tp and outputs a relation comprising the triples matching tp in the store.
- **Join**, $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$, takes as input a set of m logical operators such that A is the intersection of their attribute sets, and outputs their join on A.
- **Select**, $\sigma_c(op)(a_1, \ldots, a_n)$, takes as input the operator op and outputs those tuples from op which satisfy the condition c (a conjunction of equalities).
- **Project**, $\pi_A(op)(a_1, \ldots, a_n)$, takes as input op and outputs its tuples restricted to the attribute set A.

A *logical query plan* p is a rooted directed acyclic graph (DAG) whose nodes are logical operators. Node lo_i is a parent of lo_j in p *iff* the output of lo_i is an input of lo_j. Furthermore, a subplan of p is a sub-DAG of p.

The *plan space* of a query q, denoted as $\mathcal{P}(q)$, is the set of *all* the logical plans computing the answer to q.

B. Generating logical plans from graphs

We now outline the CREATEQUERYPLANS function used by Algorithm 1 to generate plans. When invoked, the queue $states$ contains a list of variable graphs, the last of which (tail) has only one node and thus corresponds to a completely evaluated query.

First, CREATEQUERYPLANS considers the first graph in $states$ (head), which is the initial query variable graph; let us call it $\mathbf{G_q}$. For each node in $\mathbf{G_q}$ (query triple pattern tp), a match (M) operator is created, whose input is the triple pattern tp and whose output is a relation whose attributes correspond to the variables of tp. We say this operator is *associated to* tp. For instance, consider node t_1 in the graph $\mathbf{G_1}$ of Figure 1: its associated operator is $M_{t_1}(a, b)$.

Next, CREATEQUERYPLANS builds join operators out of the following graphs in the queue. Let $\mathbf{G_{crt}}$ be the current graph in $states$ (not the first). Each node in $\mathbf{G_{crt}}$ corresponds to a clique of node(s) from the previous graph in $states$, let's call it $\mathbf{G_{prev}}$.

For each $\mathbf{G_{crt}}$ node n corresponding to a clique made of a *single* node m from $\mathbf{G_{prev}}$, CREATEQUERYPLANS *associates to* n the operator already associated to m.

For each $\mathbf{G_{crt}}$ node n corresponding to a clique of *several* nodes from $\mathbf{G_{prev}}$, CREATEQUERYPLANS creates a J_A join operator and *associates it to* n. The attributes A of J_A are the variables defining the respective clique. The parent operators of J_A are the operators associated to each $\mathbf{G_{prev}}$ node m from the clique corresponding to n; since $states$ is traversed from the oldest to the newest graph, when processing $\mathbf{G_{crt}}$, we are certain that an operator has already been associated to each node from $\mathbf{G_{prev}}$ and the previous graphs. For example, consider node A_1 in $\mathbf{G_2}$ (Figure 2), corresponding to a clique on the variable a in the previous graph $\mathbf{G_1}$ (Figure 1); the join associated to it is $J_a(abcd)$.

Further, if there are query predicate which can be checked on the join output and could not be checked on any of its inputs, a selection applying them is added on top of the join.

Finally, a projection operator π is created to return just the distinguished variables part of the query result, then projections are pushed down etc. A logical plan for the query Q_1 in Figure 1, starting with the clique decomposition/reduction shown in Figure 2, appears in Figure 3.

C. Clique decompositions and plan spaces

The plans produced by Algorithm 1 are determined by variable graphs sequences; in turn, these depend on the clique decompositions returned by CLIQUEDECOMPOSITIONS. Many clique decomposition methods exist.

First, they may use partial cliques or only maximal ones (Definition 3.2); maximal cliques correspond to systematically building joins with as many inputs (relations) as possible, while partial cliques leave more options, i.e., a join may combine only some of the relations sharing the join variables.

Second, the cliques may form an exact cover of the variable graph (ensuring each node belongs to exactly one clique), or a simple cover (where a node may be part of several cliques). Exact covers lead to tree-shaped query plans, while simple covers may lead to DAG plans. Tree plans may be seen as reducing total work, given that no intermediary result is used twice; on the other hand, DAG plans may enable for instance using a very selective intermediary result as an input to two joins in the same plan, to reduce their result size.

Third, since every clique in a decomposition corresponds to a join, decompositions having as few cliques as possible are desirable. We say a clique decomposition for a given graph is minimum among all the other possible decompositions if it contains the lowest possible number of cliques. Finding such decompositions amounts to finding minimum set covers [25].

Decomposition and algorithm acronyms. We use the following short names for decomposition alternatives. **XC** decompositions are exact covers, while **SC** decompositions are simple covers. A $+$ superscript is added when only maximal cliques are considered; the absence of this superscript indicates covers made of partial cliques. Finally, **M** is used as a prefix when only minimum set covers are considered.

We refer to the CliqueSquare algorithm variant using a decomposition alternative $\mathcal{A}$ (one among the eight above) as CliqueSquare-$\mathcal{A}$.

CliqueSquare-MSC example. We illustrate below the working of the CliqueSquare-MSC variant (which, as we will show, is the most interesting from a practical perspective), on the

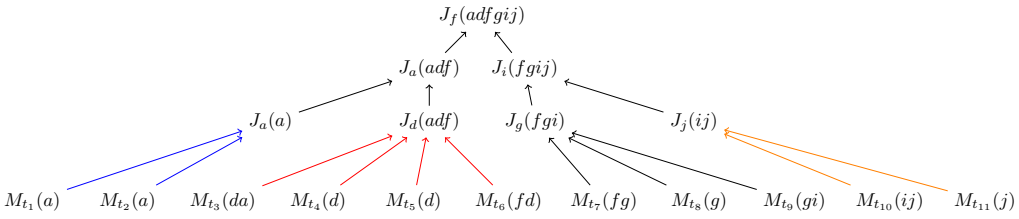
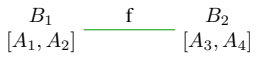
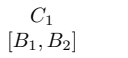


Fig. 4. Logical plan built by CliqueSquare-MSC for Q1 (Figure 1).

(a) 1st call: graph $\mathbf{G_3}$



(b) 2nd call: graph $\mathbf{G_4}$



(c) 3rd call: graph $\mathbf{G_5}$

Fig. 5. Variable graphs after each call of CliqueSquare-MSC.

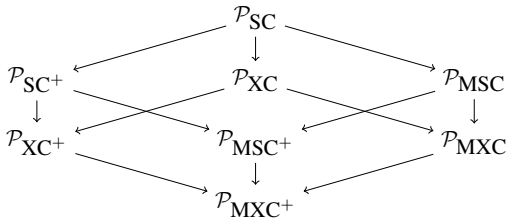


Fig. 6. Inclusions between the plan spaces of CliqueSquare variants.

query Q_1 of Figure 1. CliqueSquare-MSC builds out of the query variable graph $\mathbf{G_1}$ of Figure 1, successively, the graphs $\mathbf{G_3}$, then $\mathbf{G_4}$ and $\mathbf{G_5}$ shown in Figure 5. At the end of the process, *states* comprises $[\mathbf{G_1}, \mathbf{G_3}, \mathbf{G_4}, \mathbf{G_5}]$. CliqueSquare plans are created as described in Section IV-B; the final plan is shown in Figure 4.

The set of logical plans developed by CliqueSquare-$\mathcal{A}$ for a query q is termed *plan space of $\mathcal{A}$ for q* and we denote it $\mathcal{P}_{\mathcal{A}}(q)$; clearly, this must be a subset of $\mathcal{P}(q)$. We analyze the variants' plan spaces below.

Relationships between plan spaces. We have completely characterized the set inclusion relationships holding between the plan spaces of the eight CliqueSquare variants. Figure 6 summarizes them: an arrow from option $\mathcal{A}$ to option $\mathcal{A}'$ indicates that the plan space of option $\mathcal{A}$ *includes* the one of option $\mathcal{A}'$. For instance, CliqueSquare-SC (partial cliques, all set covers) has the largest search space $\mathcal{P}_{\text{SC}}$ which includes all the others. We have shown [26]:

Theorem 4.1 (Plan spaces inclusions): All the inclusion relationships shown in Figure 6 hold.

The inclusion relationships can be understood by noting that: (i) exact covers are a subset of simple covers, thus any plan space of the form $\mathcal{P}_{\alpha X \beta}$ is included in $\mathcal{P}_{\alpha S \beta}$, this corresponds to the four vertical arrows in Figure 6; (ii) the $+$ denotes a restriction to maximum cliques only, thus any $\mathcal{P}_{\alpha}$ includes $\mathcal{P}_{\alpha^+}$; this corresponds to the four parallel right-to-left arrows, and (iii) space $\mathcal{P}_{M\alpha}$ is included in $\mathcal{P}_{\alpha}$ for any α, since using minimum set covers only is a restriction. This is reflected by the four left-to-right arrows in Figure 6.

Optimization algorithm correctness. A legitimate question concerns the correctness of the CliqueSquare-SC, which has the largest search space: *for a given query q, does CliqueSquare-SC generate only plans from $\mathcal{P}(q)$, and all the plans from $\mathcal{P}(q)$?*

We first make the following remark. For a given query q and plan $p \in \mathcal{P}(q)$, it is easy to obtain a set of equivalent plans $p', p'', \ldots \in \mathcal{P}(q)$ by pushing projections and selections up and down. CliqueSquare optimization should not spend time enumerating p *and* such variants obtained out of p, since for best performance, σ and π should be pushed down as much as possible, just like in the traditional setting. Thus, without loss of generality, we focus on **match** and **join** operators only, assuming that two plans are equivalent if one can be obtained from the other by pushing up or down its σ and π operators. The following result holds:

Theorem 4.2 (CliqueSquare-SC correctness): For any query q, CliqueSquare-SC outputs the set of all the logical plans computing the answers to q: $\mathcal{P}_{\mathbf{SC}}(q) = \mathcal{P}(q)$.

The proof can be found in [26]. In short, *soundness* (the fact that CliqueSquare-SC only produces plans from $\mathcal{P}(q)$) directly follows from our plan generation method (Section IV-B) and the semantics of our clique decompositions.

To show *completeness*, we start by defining:

Definition 4.1 (Plan height): The *height* of a logical plan p, denoted $h(p)$, is the largest number of joins that can be found on a path from the root of p to one of its leaf operators.

The proof that any plan $p \in \mathcal{P}(q)$ is built by CliqueSquare-SC is by induction over the height of p. We show that for any level l between 0 and the height of p, CliqueSquare-SC builds a plan identical to p up to level l.

Complexity of the CliqueSquare algorithms. We study the complexity of the CliqueSquare algorithm variants by focusing on the total number of clique reductions performed, since this measure dictates the overall optimization effort. A clique reduction can be performed for each clique decomposition; thus, we count the decompositions available to each algorithm at each step, and sum them up over the successive invocations.

Proposition 4.1 (CliqueSquare complexity): The upper bounds on the number of decompositions performed by CliqueSquare variants shown in Figure 7 hold.

The detailed derivations can be found in [26]. In Figure 7, $\left\{ {n \atop k} \right\}$ is the number of ways to partition a set of n objects into k non-empty subsets, also known as the *Stirling partition number of the second kind*. The worst-case queries are not the same across variants, and in practice rarely occur; more insight is provided by our optimization experiments in Section VI-B.

D. Height optimality and associated algorithm properties

To decrease response time in our parallel setting, we are interested in flat plans, i.e., having few join operators on top of each other. First, this is because flat plans enjoy the known parallelism advantages of bushy trees. Second, while

MXC$^+$	MSC$^+$	MXC	MSC	XC$^+$	SC$^+$	XC	SC
$\binom{n+1}{\lceil n/2 \rceil}$	$\binom{2n+1}{\lceil n/2 \rceil}$	$\left\{ {n \atop \lceil n/2 \rceil} \right\}$	$\binom{2^n-1}{\lceil n/2 \rceil}$	$\sum_{k=1}^{n-1} \binom{n+1}{k}$	$\binom{2n+1}{\lceil n/2 \rceil}$	$\sum_{k=0}^{n-1} \left\{ {n \atop k} \right\}$	$\sum_{k=1}^{n-1} \binom{2^n-1}{k}$

Fig. 7. Upper bounds on the complexity of CliqueSquare variants on a query of n nodes.

HO-complete	SC
HO-partial	SC$^+$, MSC$^+$, MSC
HO-lossy	MXC$^+$, XC$^+$, MXC, XC

Fig. 8. HO properties of CliqueSquare algorithm variants.

the exact translation of logical joins into physical MapReduce-based ones (and thus, in MapReduce jobs) depends on the available physical operators, and also (for the first-level joins) on the RDF partitioning, it is easy to observe that overall, *the more joins need to be applied on top of each other, the more successive MapReduce jobs are likely to be needed by the query evaluation.* We define:

Definition 4.2 (Height optimal plan): Given a query q, a plan $p \in \mathcal{P}(q)$ is height-optimal (HO in short) iff for any plan $p' \in \mathcal{P}(q)$, $h(p) \leq h(p')$.

We classify CliqueSquare algorithm variants according to their ability to build *height optimal plans*. Observe that the height of a CliqueSquare plan is exactly the number of graphs (states) successively considered by its function CREATEQUERYPLANS, which, in turn, is the number of clique decompositions generated by the sequence of recursive CliqueSquare invocations which has lead to this plan.

Definition 4.3 (HO-completeness): CliqueSquare-$\mathcal{A}$ is *height optimal complete* (HO-complete in short) iff for any query q, the plan space $\mathcal{P}_{\mathcal{A}}(q)$ contains all the HO plans of q.

Definition 4.4 (HO-partial and HO-lossy): CliqueSquare-$\mathcal{A}$ is *height optimal partial* (HO-partial in short) iff for any query q, $\mathcal{P}_{\mathcal{A}}(q)$ contains *at least one* HO plan of q. An algorithm CliqueSquare-$\mathcal{A}$ which is not HO-partial is called HO-lossy.

An HO-lossy optimization algorithm may find *no* HO plan for a query q_1, *some* HO plans for another query q_2 and *all* HO plans for query q_3. In practice, an optimizer should provide uniform guarantees for any input query. Thus, only HO-complete and HO-partial algorithms are of interest.

The main result of our logical optimization study is:

Theorem 4.3: The properties stated in Figure 8 hold.

The proof appears in [26]; we sketch the main ideas below.

SC's *HO-completeness* follows from Theorem 4.2.

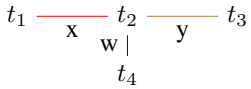


Fig. 9. Query on which XC CliqueSquare variants are HO-lossy.

The reason for the four *HO-lossy* claims is illustrated by the query shown in Figure 9. An exact cover (XC) algorithm cannot find an HO plan for this query. This is because the redundant processing introduced by considering simple (as opposed to exact) set covers may reduce plan height. For instance, using MSC$^+$, one can evaluate the query in Figure 9 with two join levels: in the first, the cliques $\{t_1, t_2\}$, $\{t_2, t_3\}$, $\{t_2, t_4\}$ are processed; in the second, the results are joined on the common variables xyz. In contrast, any plan built only from exact covers requires an extra level: t_2 is joined with the nodes of only one of its cliques, and thus, there is no common variable among the rest of the triple patterns, requiring an extra join level in order to finish processing the query.

The most interesting results are those concerning *HO-partial* algorithms; this is because, as Figure 7 and our experiments show, the complete space is sometimes too large to be explored. We start by showing why the algorithms are *not HO complete*; then we demonstrate that they are each *guaranteed to build some HO plans*.

That *SC$^+$ is not HO-complete* can be seen on the query in Figure 10. SC$^+$ finds only one plan for this query, joining $\{t_1, t_2\}$, and $\{t_2, t_3\}$ at the first level and then joining their results. It cannot build the HO plan based on the decomposition $\{\{t_1, t_2\}, \{t_3\}\}$, because $\{t_3\}$ is a partial clique.

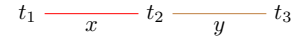


Fig. 10. Query for which CliqueSquare-SC$^+$ misses an HO plan.

That *MSC is not HO-complete* is illustrated in Figure 11. For this query, MSC only finds the plan shown at the bottom of the figure, *without* the node and edges shown within the shaded rectangle. However, the plan shown in Figure 11 *including* these node and edges is also HO, but cannot be built by MSC, because its first-level decomposition has three cliques (corresponding to its three first-level joins) whereas the minimum-size decomposition has just two.

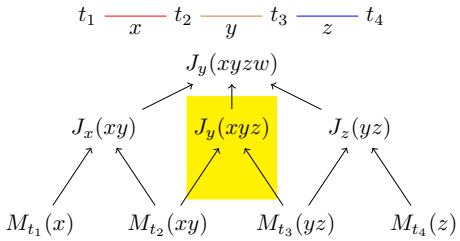


Fig. 11. Example of CliqueSquare-MSC missing an HO plan.

Given that MSC is not HO-complete, and that the search space of MSC$^+$ is included in that of MSC (Theorem 4.1), *MSC$^+$ is not HO-complete*, either.

Finally, we show that SC$^+$, MSC$^+$ and MSC *find some HO plan(s) for any query q*.

The proof for *SC$^+$* is based on the HO-completeness of SC: for any HO plan p produced by SC, we build a plan p' replacing each non-maximal clique decomposition with a maximal one, possibly introducing some projections. p' computes the same answer as p, has the same height, and is based on maximal clique decompositions only.

The proof for *MSC* also starts from an HO plan p built by SC. Observe that the *leaves* of all plans built by a CliqueSquare algorithm variant are the same, thus p is identical to any HO plan for the same query q at least at the level of the leaf (match) operators. Starting from the leaves and moving up, we group the join operators of p in *levels*, i.e., the Match operators are at level 0, the first-level joins at level 1 and so on. Now assume p's operators could have been obtained from MSC optimization *up to a level* l, $0 < l < h(p)$. We build a

plan p' as a copy of p in which the joins at level l have been replaced with a set of joins resulting from an MSC clique decomposition; there exists at least one, by reduction to the minimum set cover problem.

For a Join operator op at level l in a plan p:

- let $par(op)$ be the parents of op, for $1 < l \leq h(p)$, that is: the set of operators from level $l-1$ that beget op, i.e., that are reachable from op within p.
- let $gp(op)$ be the grandparents of op, for $2 < l \leq h(p)$, that is: the set of operators from level $l-2$ that beget op, i.e., that are reachable from op within p.

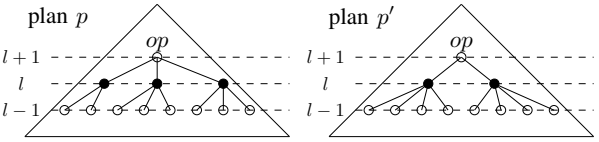


Fig. 12. Modified operators (black nodes) related to op, between p and p'.

For every operator op in p' at level $l+1$, which is identical to that of p, we connect op to a *minimal* subset of operators from level l in p', such that $gp(op)$ in p is a subset of $gp(op)$ in p' (Figure 12). It can be shown [26] that p' computes the same result as p; further, p' has the same height, and results from MSC decompositions *up to level* $l+1$. By repeating this procedure, we obtain an HO plan resulting completely from MSC decompositions. This completes the proof that *some HO plans are found by MSC for any query*.

The fact that *MSC*$^+$ is also guaranteed to find an HO plan for any query is based on MSC having this property; the proof is similar to that of SC$^+$ based on SC.

We end by noting that for some queries, CliqueSquare based on the MXC$^+$ and XC$^+$ *fails to find any plan*. The query in Figure 10 is an example: the only maximal clique decomposition is $\{t_1, t_2\}$, $\{t_2, t_3\}$, out of which no exact cover of the query nodes can be found. Thus, CliqueSquare-MXC$^+$ and CliqueSquare-XC$^+$ find no plan at all.

V. Plan evaluation on MapReduce

We now discuss the MapReduce-based evaluation of our logical plans. We first present the data storage scheme we adopt (Section V-A), based on which queries are evaluated. We then present the translation of logical plans into physical plans (Section V-B), then show how a physical plan is mapped to MapReduce jobs (Section V-C) and finally introduce our cost model (Section V-D).

A. Data partitioning

Our main goal is to split and place RDF data so that first-level joins can be evaluated locally at each node (PWOC, also termed *co-located* joins [27]), in order to reduce query response time. In the context of RDF, a single SPARQL query typically involves various joins types, e.g., subject-subject (s-s), subject-object (s-o), property-object (p-o) joins etc..

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 RDF data in three different ways and group the triples at each compute node to enable fine-granularity data access. In more detail, we proceed to store input RDF datasets in three main steps:

(1) We partition each triple and place it according to its subject, property and object values, as in [28]. Triples that share the same value in any position (s, p, o) are located within the same compute node.

(2) Then, unlike [28], we partition triples within each compute node based on their placement (s, p, 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 allows for accessing fewer triples.

(3) We further split each partition within a compute node by the value of the property in their triples. This property-based grouping has been first advocated in [13] and also resembles the vertical RDF partitioning proposed in [29] for centralized RDF stores. Finally, we store each resulting partition into an HDFS file. By using the value of the property as the filename, we benefit from a finer-granularity data access during query evaluation. It is worth noting that most RDF datasets contain many triples whose property is `rdf:type`, which in turn translates into a very large property partition. Thus, we further split the property partition of `rdf:type` into several smaller partitions, according to their object value. This enables working with finer-granularity partitions.

In contrast e.g., to Co-Hadoop [30], which considers a single attribute for co-locating triple, our partitioner co-locates them on the three attributes (one for each data replica). This allows us to perform *all first-level joins* in a plan (s-s, s-p, s-o etc.) *locally* in each compute node during query evaluation.

B. From logical to physical plans

We define a *physical plan* as a rooted DAG such that (i) each node is a physical operator and (ii) there is a directed edge from op_1 to op_2 iff op_1 is a parent of op_2. To translate a logical plan, we rely on the following physical MapReduce operators:

- **Map Scan**, $MS[FS]$, parameterized by a set of HDFS files FS, outputs one tuple for each line of every file in FS.
- **Filter**, $\mathcal{F}_{con}(op)$, where op is a physical operator, outputs the tuples produced by op that satisfy logical condition con.
- **Map Join**, $MJ_A(op_1, \ldots, op_n)$, is a directed join [31] that joins its n inputs on their common attribute set A.
- **Map Shuffler**, $MF_A(op)$, is the repartition phase of a repartition join [31] on the attribute set A; it shuffles each tuple from op on A's attributes.
- **Reduce Join**, $RJ_A(op_1, \ldots, op_n)$, is the join phase of a repartition join [31]. It joins n inputs on their common attribute set A by (i) gathering the tuples from $op_1, \ldots, op_n$ according to the values of their A attributes, (ii) building on each compute node the join results.
- **Project**, $\pi_A(op)$, is a simple projection (vertical filter) on the attribute set A.

We translate a logical plan p_l into a physical plan, operator by operator, from the bottom (leaf) nodes up, as follows.

match: Let M_{tp} be a match operator (a leaf in p_l), having $k \geq 1$ outgoing (parent-to-child) edges. (1) For each such outgoing edge e_j of M_{tp}, $1 \leq j \leq k$, we create a map scan

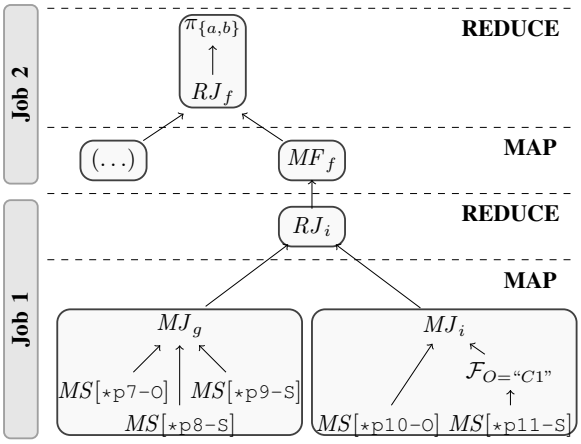


Fig. 13. Part of Q1 physical plan and its mapping to MapReduce jobs.

operator matching the appropriate files names f_j in HDFS. (2) If the triple pattern tp has a constant in the subject and/or object, a *filter* operator $\mathcal{F}_{con}$ is added on top of $MS[f_j]$, where con is a predicate constraining the subject and/or object as specified in tp. Observe the filter on the property, if any, has been applied through the computation of the f_j file name.

join: let J_A be a logical join; two cases may occur. (1) If all parent nodes of J_A are match operators, then J_A is transformed into a map join MJ_A. (2) Otherwise, we build a reduce join RJ_A. As a reduce join cannot be performed directly on the output of another reduce join, a map shuffler operator is added, if needed.

select: is mapped directly to the $\mathcal{F}$ physical operator.

project: is mapped directly to the respective physical operator.

For illustration, Figure 13 depicts the physical plan of Q1 built from its logical plan shown in Figure 4. Only the right half of the plan is detailed since the left side is symmetric.

C. From physical plans to MapReduce jobs

As a final step, we map a physical plan to 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 MapReduce task (either *map* or *reduce* task), (iii) any other operator is executed in a MapReduce task of its own. 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. Figure 13 shows how the physical plan of Q1 is transformed into a MapReduce program (i.e., a set of MapReduce jobs); rounded boxes show the grouping of physical operators into MapReduce tasks.

D. Cost model

We now define the cost $c(p)$ of a MapReduce query plan p, which allows us choosing a query plan among others, as an estimation of the total work $tw(p)$, required by the MapReduce framework, to execute p: $c(p) = tw(p)$. The total work accounts for (i) scan costs, (ii) join processing costs, (iii) I/O incurred by the MapReduce framework writing intermediary results to disk, and (iv) data transfer costs.

Observe that for full generality, our cost model takes into account many aspects (and not simply the plan height). Thus, while some of our algorithms are guaranteed to find plans as flat as possible, priority can be given to other plan metrics if they are considered important. In our experiments, the selected plans (based on this general cost model) were HO for all the queries but one (namely Q14).

While MapReduce program performance can be modeled at much finer granularity [32], [33], the simple model above has been sufficient to guide our optimizer well, as our experiments demonstrate next.

VI. Experimental evaluation

We have implemented the CliqueSquare optimization algorithms together with our partitioning scheme, and the physical MapReduce-based operators in a prototype we onward refer to as CSQ. First, we perform an in-depth evaluation of the different optimization algorithms presented in Section IV-C to identify the most interesting ones. We then time the execution of the best plans recommended by our CliqueSquare optimization algorithms, and compare it with the runtime of plans as created by previous systems: linear or bushy, but based on binary joins. Finally, we compare CSQ query evaluation times with those of two state-of-the-art MapReduce-based RDF systems and show the query robustness of CSQ.

A. Experimental setup

Cluster. Our cluster consists of 7 nodes, where each node has: one 2.93GHz Quad Core Xeon processor with 8 threads; 4×4GB of memory; two 600GB SATA hard disks configured in RAID 1; one Gigabit network card. Each node runs CentOS 6.4. We use Oracle JDK v1.6.0_43 and Hadoop v1.2.1 for all experiments with the HDFS block size set to 256MB.

Dataset and queries. We rely on the LUBM [34] benchmark, since it has been extensively used in similar works such as [13], [7], [35], [12]. We use the LUBM10k dataset containing approximately 1 billion triples (216 GB). The LUBM benchmark features 14 queries, most of which return an empty answer if RDF reasoning (inference) is not used. Since reasoning was not considered in prior MapReduce-based RDF databases [7], [12], [8], to evaluate these systems either the queries were modified, or empty answers were accepted; the latter contradicts the original benchmark query goal. We modified the queries as in [12] replacing generic types (e.g., <Student>, of which no explicit instance exists in the database) with more specific ones (e.g., <GraduateStudent> of which there are some instances). Further, the benchmark queries are relatively simple; the most complex one consists of only 6 triple patterns. To complement them, we devised other 11 LUBM-based queries with various selectivities and complexities, and present them next to a subset of the original ones to ensure variety across the query set. The complete workload can be found in [26].

B. Plan spaces and CliqueSquare variant comparison

We compare the 8 variants of our CliqueSquare algorithms w.r.t. : (i) the total number of generated plans, (ii) the number of height-optimal (HO) plans, (iii) their running time, and (iv) the number of duplicate plans they produce.

Option	Chain	Dense	Thin	Star
MXC$^+$	0.4	0.4	0.4	1
XC$^+$	0.4	0.4	0.4	1
MSC$^+$	**2.1**	**1.1**	**2.1**	**1**
SC$^+$	**764.6**	**1.2**	**764.6**	**1**
MXC	**5.4**	**6.47**	**5.4**	**1**
XC	52451.97	166944.57	51522.67	175273.80
MSC	**18.2**	**26**	**18.2**	**1**
SC	58948.33	23871.90	58394.27	54527.63

Fig. 14. Average number of plans per algorithm and query shape.

Option	Chain	Dense	Thin	Star
MXC$^+$	40%	40%	40%	100%
XC$^+$	40%	40%	40%	100%
MSC$^+$	**100%**	**100%**	**100%**	**100%**
SC$^+$	**71.9%**	**100%**	**71.9%**	**100%**
MXC	**100%**	**100%**	**100%**	**100%**
XC	34.8%	24.0%	34.8%	22.8%
MSC	**100%**	**100%**	**100%**	**100%**
SC	32.6%	21.5%	32.6%	21.5%

Fig. 15. Average optimality ratio per algorithm and query shape.

Option	Chain	Dense	Thin	Star
MXC$^+$	2.80	0.17	0.83	0.1
XC$^+$	0.63	0.07	0.20	0.13
MSC$^+$	**3.73**	**0.10**	**4.30**	**0.10**
SC$^+$	1836.47	0.17	1833.57	0.03
MXC	**42.03**	**1.77**	**40.77**	**0.43**
XC	13046.43	32023.50	12942.5	33442.73
MSC	**197.5**	**4.73**	**195.47**	**0.43**
SC	41095.07	53859.87	41262.33	61714.77

Fig. 16. Average optimization time (ms) per algorithm and query shape.

Option	Chain	Dense	Thin	Star
MXC$^+$	100%	100%	100%	100%
XC$^+$	100%	100%	100%	100%
MSC$^+$	**100%**	**100%**	**100%**	**100%**
SC$^+$	99.95%	98.89%	99.67%	100%
MXC	**100%**	**86.18%**	**100%**	**100%**
XC	97.80%	80.17%	98.63%	91.01%
MSC	**100%**	**91.50%**	**100%**	**100%**
SC	99.55%	62.89%	99.68%	93.81%

Fig. 17. Average uniqueness ratio per algorithm and query shape.

Setup. We use the generator of [16] to build 120 synthetic queries whose shape is either *chain*, *star*, or *random*, with two variants *thin* or *dense* for the latter: dense ones have many variables in common across triples, while thin ones have significantly less, thus they are close to chains. The queries have between 1 and 10 (5.5 on average) triple patterns. Each algorithm was stopped after a time-out of 100 seconds.

Comparison. Figure 14 shows the *search space size* for each algorithm variant and query type. The total number of generated plans is measured for each query and algorithm; we report the average per query category. As illustrated in Section IV-C, MXC$^+$and XC$^+$ fail to find plans for some queries (thus the values smaller than 1). SC and XC return an extremely large number of plans, whose exploration is impractical. For these reasons, MXC$^+$, XC$^+$, XC, and SC are not viable alternatives. In contrast, MSC$^+$, SC$^+$, MXC, and MSC produce a reasonable number of plans to choose from.

Figure 15 shows the average *optimality ratio* defined as the number of HO-plans divided by the number of all produced plans. We consider this ratio to be 0 for queries for which no plan is found. While the ratio for MSC$^+$, MXC, and MSC is 100% for this workload (i.e., they return *only* HO plans), this is not guaranteed in general. SC$^+$ has a smaller optimality ratio but still acceptable. On the contrary, although XC finds some optimal plans, its ratio is relatively small.

Options MSC$^+$, MXC, and MSC lead to the shortest *optimization time* as shown in Figure 16. MSC is the slowest among these three algorithms, but it is still very fast especially compared to a MapReduce program execution, providing an answer in less than $1s$.

Given that our optimization algorithm is not based on dynamic programming, it may end up producing the same plan more than once. In Figure 17 we present the average *uniqueness ratio*, defined as the number of unique plans divided by the total number of produced plans. Dense queries are the most challenging for all algorithms, since they allow more sequences of decompositions which, after a few steps, can converge to the same (and thus, build the same plan more than once). However, in practice, as demonstrated in the figure, our dominant decomposition methods, MSC$^+$, MXC, and MSC produce very few duplicate plans.

Summary. Based on our analysis, the optimization algorithms based on MSC$^+$, MXC, and MSC return sufficiently many HO plans to chose from (with the help of a cost model), and produce these plans quite fast (in less than one second, negligible in an MapReduce environment). However, Theorem 4.3 stated that MXC is HO-lossy; therefore, we do not recommend relying on it in general. In addition, recalling (from Theorem 4.1) that the search space of MSC is a superset of those of MSC$^+$, and given that the space of CliqueSquare-MSC is still of reasonable size, we consider it the best CliqueSquare algorithm variant, and we rely on it exclusively for the rest of our evaluation.

C. CliqueSquare plans evaluation

We now measure the practical interest of the flat plans with n-ary joins built by our optimization algorithm.

Setup. We compare the plan chosen by our cost model among those built by CliqueSquare-MSC, against the *best binary bushy* plan and the *best binary linear* plan for each query. To find the best binary linear (or bushy) plan, we build them all, and then select the cheapest using the cost function described in Section V-D. We translate all logical plans into MapReduce jobs as described in Section V and execute them on our CSQ prototype.

Comparison. Figure 18 reports the execution times (in seconds) for 14 queries (ordered from left to right with increasing number of triple patterns). In the x-axis, we report, next to the query name, the number of triples patterns followed (after the | character) by the number of jobs that are executed for each plan (where M denotes a map only job). For example, Q3(3|M11) describes query Q3, which is composed of 3 triple patterns, and for which MSC needs a map only job while the bushy and linear plans need 1 job each. The optimization time

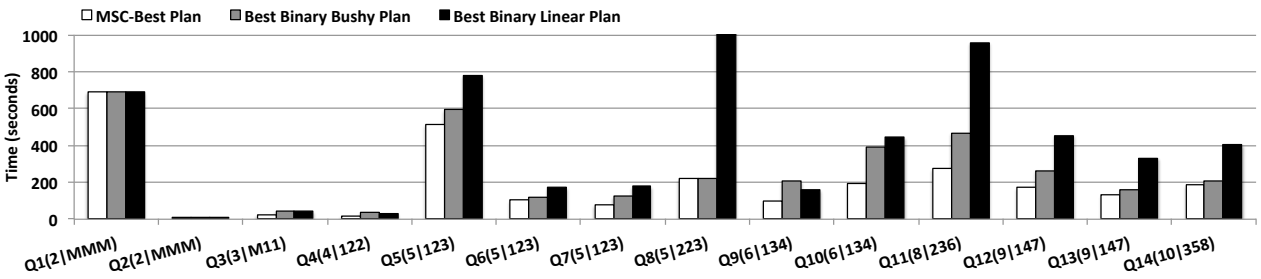


Fig. 18. Plan execution time (in seconds) comparison between MSC-plans, bushy-plans, and linear plans for LUBM10k.

is not included in the execution times reported. This strongly favors the bushy and linear approaches, because the number of plans to produce and compare is bigger than that for MSC.

For all queries, the MSC plan is faster than the best bushy plan and the best linear plan, by up to a factor of 2 (for query Q9) compared to the binary bushy ones, and up to 16 (for query Q8) compared to the linear ones. The three plans for Q1 (resp. Q2) are identical since the queries have 2 triple patterns. For Q8, the plan produced with MSC is the same as the best binary bushy plan, thus the execution times are almost identical. As expected the best bushy plans run faster than the best linear ones, confirming the interest of parallel (bushy) plans in a distributed MapReduce environment.

Summary. CliqueSquare-MSC plans outperform the bushy and linear ones, demonstrating the advantages of the n-ary star equality joins it uses.

D. CSQ *system evaluation*

We now analyze the query performance of CSQ with the MSC algorithm and run it against comparable massively distributed RDF systems, based on MapReduce. While some memory-based massively distributed systems have been proposed recently [14], [35], we chose to focus on systems comparable with CSQ in order to isolate as much as possible the impact of the query optimization techniques that are the main focus of this paper.

Systems. We pick SHAPE [8] and H_2RDF+ [12], since they are the most efficient RDF platforms based on MapReduce; the previous HadoopRDF [13] is largely outperformed by H_2RDF+ [12] and [7] is outperformed by [8]. H_2RDF+ is open source, while we used our own implementation of SHAPE. SHAPE explores various partitioning methods, each with advantages and disadvantages. We used their 2-hop forward partitioning ($2f$) since it has been shown to perform the best for the LUBM benchmark.

Comparison. While CSQ stores RDF partitions in simple HDFS files, H_2RDF+ uses HBase, while SHAPE uses RDF-3X [15]. Thus, SHAPE and H_2RDF+ benefit from index access *locally* on each compute node, while our CSQ prototype can only scan HDFS partition files. We consider two classes of queries: *selective* queries (which on this 1 billion triple database, return less than 0.5×10^6 results) and *non-selective* ones (returning more than 7.5×10^6 results).

Figure 19 shows the running times: selective queries at the left, non-selective ones at the right. As before, next to the query name we report the number of triple patterns followed by the number of jobs that the query needs in order to be executed in each system (M denotes one map only job). H_2RDF+ sometimes uses map-only jobs to perform first-level joins, but it performs each join in a separate MapReduce job, unlike CSQ (Section V).

Among the 14 queries of the workload, 4 (Q2, Q4, Q9, Q10) are PWOC for SHAPE (not for CSQ) and 1 (Q3) is PWOC for CSQ (not for SHAPE). These five queries are selective, and, as expected, perform better in the system which allows them to be PWOC. For the rest of the queries, where the optimizer plays a more important role, CSQ outperforms SHAPE for all but one query (Q11 has an advantage with $2f$ partitioning since a larger portion of the query can be pushed inside RDF-3X). The difference is greater for non-selective queries since a bad plan can lead to many MapReduce jobs and large intermediary results that affect performance. Remember that the optimization algorithm of SHAPE is based on heuristics without a cost function and produces *only one* plan. The latter explains why even for selective queries (like Q13 and Q14 which are more complex than the rest) CSQ performs better than SHAPE.

We observe that CSQ significantly outperforms H_2RDF+ for all the non-selective queries and for most of the selective ones, by 1 to more than 2 orders of magnitude. For instance, Q7 takes 4.8 hours on H_2RDF+ and only 1.3 minutes on CSQ. For queries Q1 and Q8 we had to stop the execution of H_2RDF+ after 5 hours, while CSQ required only 3.6 and 11 minutes, respectively. For selective queries the superiority of CSQ is less but it still outperforms H_2RDF+ by an improvement factor of up to 5 (for query Q9). This is because H_2RDF+ builds left-deep query plans and does not fully exploit parallelism; H_2RDF+ requires more jobs than CSQ for most of the queries. For example, for query Q12 H_2RDF+ initiates 4 jobs one after the other. Even if the first two jobs are map-only, H_2RDF+ still needs to read and write the intermediate results produced and pay the initialization overhead of these MapReduce jobs. In contrast, CSQ evaluates Q12 in a single job.

Summary. While SHAPE and H_2RDF+ focus mainly on data access paths techniques and thus perform well on selective queries, CSQ performs closely (or better in some cases), while it outperforms them significantly for non-selective queries. CSQ evaluates our complete workload in 44 minutes, while SHAPE and H_2RDF+ required 77 min and 23 hours, respectively. We expect that such systems can benefit from the logical query plans built by CliqueSquare to obtain fewer jobs and

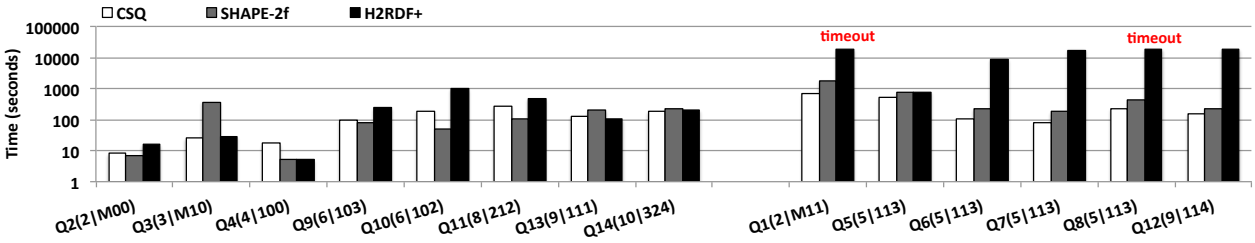


Fig. 19. Query evaluation time comparison: CSQ, SHAPE and H_2RDF+.

thus, lower query response times.

VII. Conclusion

Numerous distributed platforms have been proposed to handle large volumes of RDF data [6], in particular based on parallel processing frameworks such as MapReduce. In this context, our work focused on the *logical optimization of large conjunctive (BGP) SPARQL queries*, featuring many joins. We are interested in building *flat* logical plans to diminish query response time, and investigate the usage of n*-ary (star) equality joins* for this purpose.

We have presented CliqueSquare, a generic optimization algorithm and eight variants thereof, which build tree- or DAG-shaped plans using n-ary star joins. We have formally characterized their ability to find the flattest possible plans. Finally, we have put these algorithms to task in a complete MapReduce-based RDF data management platform [36]. Our experiments demonstrate that CliqueSquare-MSC is the most interesting alternative; it is guaranteed to find some of the flattest plans which, as shown in our experiments, outperform previous comparable systems, especially for complex queries where optimization plays an important role. More generally, our logical optimization approach can be used in any massively parallel conjunctive query evaluation setting, contributing to shorten query response time.

References

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

[2] 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*, 2007.

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

[4] D. Huynh, S. Mazzocchi, and D. R. Karger, "Piggy Bank: Experience the Semantic Web inside your web browser," *J. Web Sem.*, vol. 5, no. 1, 2007.

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

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

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

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

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

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

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

[12] 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.

[13] 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.

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

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

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

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

[18] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in SPARQL queries," in *EDBT*, 2014.

[19] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using MapReduce," *ACM Comput. Surv.*, vol. 46, no. 3, p. 31, 2014.

[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using Hadoop," in *ICDE*, 2010.

[21] A. Gates, J. Dai, and T. Nair, "Apache Pig's optimizer," *IEEE Data Eng. Bull.*, vol. 36, no. 1, 2013.

[22] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query Optimization for Massively parallel Data Processing," in *SOCC*, 2011.

[23] F. N. Afrati and J. D. Ullman, "Optimizing Joins in a Map-Reduce Environment," in *EDBT*, 2010.

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

[25] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, 1972, pp. 85–103.

[26] F. Goasdoué, Z. Kaoudi, I. Manolescu, J. Quiané-Ruiz, and S. Zampetakis, "CliqueSquare: Flat Plans for Massively Parallel RDF Queries," Research Report RR-8612, Oct. 2014. [Online]. Available: https://hal.inria.fr/hal-01071984

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

[28] M. Cai, M. R. Frank, B. Yan, and R. M. MacGregor, "A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management," *Journal of Web Semantics*, vol. 2, no. 2, pp. 109–130, 2004.

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

[30] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop," *PVLDB*, 2011.

[31] 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 *SIGMOD*, 2010.

[32] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *PVLDB*, vol. 3, no. 1, 2010.

[33] H. Herodotou, F. Dong, and S. Babu, "MapReduce Programming and Cost-based Optimization? Crossing this Chasm with Starfish," *PVLDB*, vol. 4, no. 12, 2011.

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

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

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