CliqueSquare: Flat Plans for Massively Parallel RDF Queries

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

CliqueSquare: Flat Plans for Massively Parallel RDF Queries

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

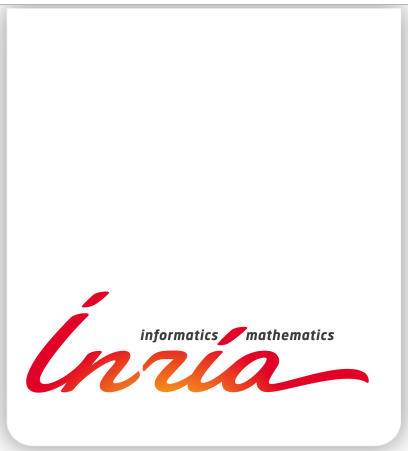
RESEARCH REPORT

N° 8612

October 2014

Project-Teams OAK

ISSN 0249-6399 ISRN INRIA/RR--8612--FR+ENG



CliqueSquare: Flat Plans for Massively Parallel RDF Queries

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

Project-Teams OAK

Research Report n° 8612 — October 2014 — 37 pages

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.

Key-words: Query optimizations, parallel databases, MapReduce, RDF, n-ary joins, Semantic Web, SPARQL

* Institute for the Management of Information Systems
† Qatar Computing Research Institute

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

CliqueSquare: Plans Massivement Parallèles pour l'Evaluation Efficace de Requêtes RDF

Résumé : Pour faire face à l'explosion du volume de données RDF produites et analysées quotidiennement, de nombreux systèmes de stockage et d'interrogation de données RDF massivement distribués ont été développés. Ces architectures sont caractérisées par leur méthode de partitionnement et de stockage de données RDF d'une part, et, d'autre part, par la façon dont elles optimisent les requêtes, c'est-à-dire la manière dont les calculs sont distribués entre les différents nœuds afin de calculer les réponses.

Cet article présente CliqueSquare, une nouvelle approche d'optimisation pour l'évaluation de requêtes RDF conjonctives dans un environnement massivement parallèle. Notre but principal est de réduire le temps de réponse des requêtes; pour cela, nous nous intéressons aux plans d'éxecution "plats" (de faible hauteur), dans lesquels le nombre de jointures successives sur un chemin allant de la racine du plan d'exécution jusqu'à l'un de ses opérateurs feuilles est minimisé. Nous présentons une famille d'algorithmes d'optimisation, basés sur des jointures d'égalité n-aires (en "étoile"), pour construire des plans plats et comparons leurs capacités à trouver les plans les plus plats possibles. Nous avons implémenté nos algorithmes dans une plate-forme RDF basée sur MapReduce; nos expériences démontrent l'intérêt des plans plats construits par nos meilleurs algorithmes d'optimisation.

Mots-clés : Optimisation de requêtes, bases de données parallèle, MapReduce, RDF, jointures n-aires, Web Sémantique, SPARQL

Contents

1 Introduction

The Resource Description Framework (RDF) [14] 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 [3, 30] and scientific applications [34, 37] to Web 2.0 platforms [19, 36] and databases [6]. While its query language, SPARQL [28], 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 [21]. 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. Accordingly, many previous works on distributed RDF query evaluation, such as [17, 23, 8, 16], have placed an important emphasis on the data partitioning process (workload-driven in the case of [8, 16]), 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 re-distribution 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 [26], 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 [8], n-ary joins (with $n > 2$) only in the first level of the plans and binary joins in the next levels [17, 23, 16], or n-ary joins at all levels [27] but organized in left-deep plans. Such methods lead to high (non-flat) plans and hence high response times. HadoopRDF [18] 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 [13]; 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 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 [25, 10, 32]) 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 2. Section 3 introduces the logical model used in CliqueSquare for queries and query plans and describes our generic logical optimization algorithm. In Section 4, we present our algorithm variants, their search spaces, and analyze them from the viewpoint of their ability to produce flat query plans. Section 5 shows how to translate and execute our logical plans to MapReduce jobs, based on a generic RDF partitioning strategy. Section 6 experimentally demonstrates the effectiveness and efficiency of our logical optimization approach and Section 7 concludes our findings.

2 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[1], 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 o-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 [28] 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 [25] 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 [32], DP is avoided and plans are heuristically built relying solely on the shape of the query, without using cardinality estimations etc. In [11], 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 [24] and RDF [21] data.

Early works on relational data mainly focus on selection and projection push-down [31], while [9] relies on other classical distributed database techniques [26]. The authors in [33] propose a cost-based approach for deciding how to split a query into a set of *fragments*; they use an n-ary repartition join [2] to join each fragment. Then, the authors consider 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 [33], our approach relies on n-ary joins at all levels and hence it develops some logical plans that [33] 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 [17, 23, 39, 8, 16], with the goal of making the

[1] 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.

first-level joins (those applied directly on the input data) PWOC. Independently, aggressive indexing and compression of RDF data has been studied in [27]. 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 [8], n-ary joins (with $n > 2$) only in the first level of the plans and binary joins in the next levels [17, 23, 16], or n-ary joins at all levels [27] 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 [18], 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.

```
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"}
```

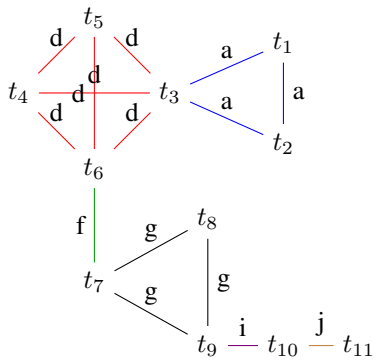


Figure 1: Query Q_1 and its variable graph $\mathbf{G_1}$.

3 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 3.1 and present the CliqueSquare optimization algorithm in Section 3.2.

3.1 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) [35], 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* $v \in V$*.*

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.

3.2 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

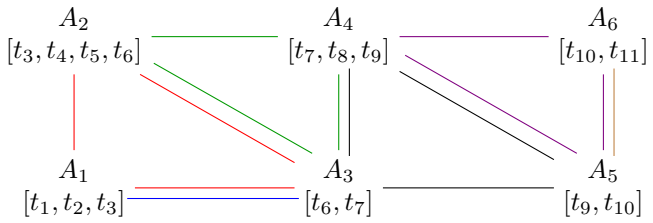


Figure 2: Clique reduction $\mathbf{G_2}$ of Q_1's variable graph (shown in Figure 1).

a logical plan. In the sequel of Section 3, we detail the graph transformation process, and delegate plan building to Section 4.

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 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 $v \in V$.*

Algorithm 1: CliqueSquare algorithm

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

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.

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.

4 Query Planning

We describe CliqueSquare's logical operators, plans, and plan spaces (Section 4.1) and how logical plans are generated by Algorithm 1 (Section 4.2). We then consider a set of alternative concrete clique decomposition methods to use within the CliqueSquare algorithm, and describe the resulting search spaces (Section 4.3). 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 4.4). Finally, we present a complexity analysis of our optimization algorithm (Section 4.5).

4.1 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.

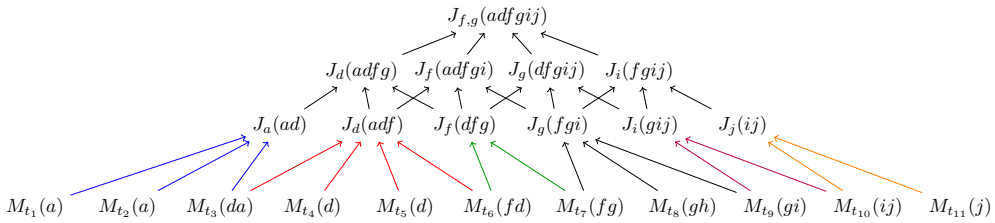


Figure 3: Sample logical plan built by CliqueSquare for Q1 (Figure 1).

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

- **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.

4.2 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.

4.3 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 [22].

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 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 4.2; the final plan is shown in Figure 4.

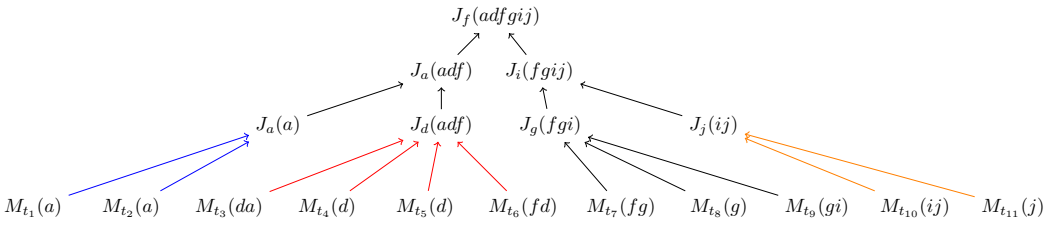


Figure 4: Logical plan built by CliqueSquare-MSC for Q1 (Figure 1).

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.

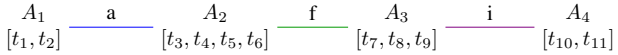
Relationships between plan spaces. We have completely characterized the set inclusion relationships holding between the plan spaces of the eight CliqueSquare variants. Figure 7 summarizes them: an arrow from option $\mathcal{A}$ to option $\mathcal{A}'$ indicates that the plan space of option $\mathcal{A}$ *includes* the one of option $\mathcal{A}'$. For instance, CliqueSquare-SC (partial cliques, all set covers) has the biggest search space $\mathcal{P}_{\text{SC}}$ which includes all the others. Formally:

Theorem 4.1 (Plan spaces relationships)**.** *All the inclusion relationships shown in Figure 7 hold.*

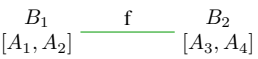
Proof. Recall that a decomposition is determined by three choices: (a) maximal or partial cliques; (b) exact or simple set cover; (c) minimum-size versus all covers. Therefore, we use a triple (o_1, o_2, o_3) where $o_i \in \{<, >, =\}$, $1 \leq i \leq 3$, to encode three relationships between two options, option i and option j:

- The symbol o_1 represents the relationship between the types of cliques used: since maximal cliques are a special case of partial cliques, o_1 is $<$ iff i uses maximal cliques while j uses partial cliques, $>$ if the opposite holds, and $=$ otherwise.
- The symbol o_2 represents the relationship between the types of cover used. Similarly, since exact covers are particular cases of set covers, o_2 is $<$ iff Option i relies on exact covers and Option j on general set covers; o_2 is $>$ in the opposite case, and $=$ otherwise.
- Finally, o_3 encodes the relationship between the size of the covers which are retained from the cover algorithms: minimum set covers being more restrictive, o_3 is $<$ iff Option i uses only the minimum covers while Option j uses them all, $>$ in the opposite case, and $=$ otherwise.

For example, comparing MXC$^+$ with XC$^+$ leads to the triple $(=, =, <)$: they both use maximal cliques and exact cover; MXC$^+$ considers only minimum covers while XC$^+$ considers them all. We say a symbol $s \in \{<, >\}$ *dominates* a triple (o_1, o_2, o_3) if the triple contains s and does not contain



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



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

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

Figure 5: Variable graphs after each call of CliqueSquare-MSC.

	XC$^+$	MSC$^+$	SC$^+$	MXC	XC	MSC	SC
MXC$^+$	$(=,=,<)$	$(=,<,=)$	$(=,<,<)$	$(<,=,=)$	$(<,=,<)$	$(<,<,=)$	$(<,<,<)$
XC$^+$		$(=,<,>)$	$(=,<,=)$	$(<,=,>)$	$(<,=,=)$	$(<,<,>)$	$(<,<,=)$
MSC$^+$			$(=,=,<)$	$(<,>,=)$	$(<,>,<)$	$(<,=,=)$	$<,=,<)$
SC$^+$				$(<,>,>)$	$(<,>,=)$	$(<,=,>)$	$(<,=,=)$
MXC					$(=,=,<)$	$(=,<,=)$	$(=,<,<)$
XC						$(=,<,>)$	$(=,<,=)$
MSC							$(=,=,<)$
SC							

Figure 6: Detailed relationships between decomposition options.

the opposite-direction symbol. For instance, $<$ dominates $(=,<,=)$ as well as $(<,<,=)$, but does not dominate $(<,>,=)$; $>$ does not dominate the latter, either. The following simple property holds:

Proposition 4.1 (Option domination)**.** *Let i, j be two options, $i, j \in$ {MXC$^+$, XC$^+$, MSC$^+$, SC$^+$, MXC, XC, MSC, SC} and (o_1, o_2, o_3) be the comparison triple of the options i and j. If $<$ (respectively, $>$) dominates (o_1, o_2, o_3), then the plan space of CliqueSquare using option i is included (respectively, includes) in the plan space of CliqueSquare using option j.*

The reason for the above is that each comparison symbol encodes the relationship between the alternatives available to each algorithm. If neither $<$ nor $>$ dominates the comparison triple, it can be easily shown that the search spaces are incomparable (not included in one another). Figure 6 shows the comparison triples for all pairs of decomposition options. The cell (row, col) corresponds to the comparison of option row and option col. The comparisons dominated by $<$ or $>$ (which entail a relationship between the respective search spaces) are highlighted. □

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. We say two plans $p, p' \in \mathcal{P}(q)$ are *similar*, denoted $p \sim p'$, if p' can be obtained from p by moving σ

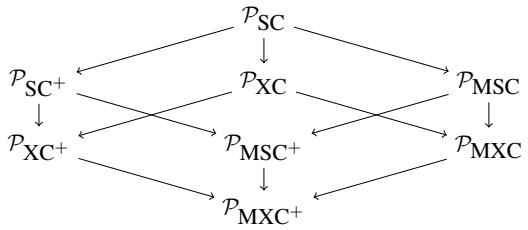


Figure 7: Inclusions between the plan spaces of CliqueSquare variants.

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

Figure 8: Upper bounds on the complexity of CliqueSquare variants on a query of n nodes.

and π up and down. We denote by $\mathcal{P}^{\sim}(q)$ the space of equivalence classes obtained from $\mathcal{P}(q)$ and the equivalence relation $\sim$. By a slight abuse of notations, we view $\mathcal{P}^{\sim}(q)$ to be a set of *representative* plans, one (arbitrarily chosen) from each equivalence class.

Based on this discussion, and to keep notations simple, *in the sequel we use $\mathcal{P}(q)$ to refer to $\mathcal{P}^{\sim}(q)$*, and we say an algorithm CliqueSquare-$\mathcal{A}$ is *correct* iff it is both *sound*, i.e., it produces only representatives of some equivalence classes from $\mathcal{P}^{\sim}(q)$, and *complete*, i.e., it produces a representative from every equivalence class of $\mathcal{P}^{\sim}(q)$.

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: $\mathcal{P}_{\mathbf{SC}}(q) = \mathcal{P}(q)$.*

Proof. We first show that $\mathcal{P}_{\mathbf{SC}}(q) \subseteq \mathcal{P}(q)$ holds, before proving $\mathcal{P}(q) \subseteq \mathcal{P}_{\mathbf{SC}}(q)$.

Soundness. $\mathcal{P}_{\mathbf{SC}}(q) \subseteq \mathcal{P}(q)$ directly follows from our plan generation method starting from a sequence of variable graphs produced within CliqueSquare-SC (Section 4.2), by recursive $\mathbf{SC}$-clique decompositions/reductions (Section 3.2).

Completeness. For proving $\mathcal{P}(q) \subseteq \mathcal{P}_{\mathbf{SC}}(q)$, consider any plan $p \in \mathcal{P}(q)$ and let us show that CliqueSquare-SC builds a plan $p' \in \mathcal{P}_{\mathbf{SC}}(q)$ similar to p ($p \sim p'$), i.e., disregarding the projection and selection operators. That is, we use p to refer to its *subplan* consisting of all its leaves (match operators) up to the last join operator, assuming all the σ and π operators have been pushed completely up.

The proof relies on three notions: (i) the height of a plan, (i) the subplans at a given level of a plan, and (iii) the equality of two plans up to a level.

For a given plan p, the *height* of p, denoted $h(p)$, is the largest number of successive join operators encountered in a root-to-leaf path of p; a *level* l of p is an integer between 0 and $h(p)$.

A subplan of p is a sub-DAG of p. For any subplan p' of p whose root is the node n, p' *is at level* l, for $0 \leq l \leq h(p)$ iff the longest n-to-leaf path is of size at most l, *and* the longest path from a direct parent of n (if any) to a leaf is of size at least $l+1$. In particular, the match operator leaves of a plan p are all the subplans of p at level 0, while p is its only subplan at level $h(p)$.

Finally, two plans are *equal up to a level* iff they have the same subplans at that level.

With the above notions in place, showing completeness amounts to proving the property:

$(*)$ *for any plan $p \in \mathcal{P}(q)$, CliqueSquare-SC produces a plan equal to p up to level $h(p)$.*

We prove this by induction on the level l of a sub-plan of p, as follows.

(Base) For $l = 0$, i.e., we consider p's leaves only, which are necessarily **match** operators, one for every triple pattern in q. Since CliqueSquare-SC is initially called with the variable graph G of q having a single triple per graph node, and with the empty queue *states*, any plan generated by CliqueSquare-SC using `createQueryPlans` has the same leaves as p. Therefore, any plan produced by CliqueSquare-SC is equal to p up to $l = 0$.

(Induction) Suppose that the above property $(*)$ holds up to level n, and let us show it also holds up to level $n + 1$.

At level $n+1$, consider the new join operators that are not at level n. These operators correspond to the roots of subplans, i.e., of sub-DAGs of p, whose children are roots of sub-DAGs of p at level n. For any such new join operator $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$, consider a plan p' produced by CliqueSquare-SC that is equal to p up to level n (p' exists thanks to the induction hypothesis).

By construction, p' has been produced from the $states$ variables of CliqueSquare-SC in which the n^{th} variable graph G_q^n has one node per root of subplan of p at level n. Any new join operator $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$ introduced in p at level $n + 1$ has as children $op_1, \ldots, op_m$ operators at level n. Since every operator $op_1, \ldots, op_m$ outputs the set of attributes A, the nodes corresponding to these operators form some cliques (as many as there are variables in A) in G_q^n. As, by definition, any such clique can be found by a SC clique decomposition, there exists a plan $p'' \in \mathcal{P}_{\mathbf{SC}}(q)$ generated by CliqueSquare-SC from the $states$ variable whose first n variable graphs are equal to those from which p' has been generated, and whose $n + 1^{th}$ graph has a node corresponding to $J_A(op_1, \ldots, op_m)(a_1, \ldots, a_n)$. Therefore, there exists a plan produced by CliqueSquare-SC that is equal to p up to $n + 1$. □

4.4 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 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 4.1 (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 4.2 (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 4.3 (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:

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

Figure 9: HO properties of CliqueSquare algorithm variants.

Figure 9 classifies the eight CliqueSquare variants we mentioned, from the perspective of these properties.

Theorem 4.3. *The properties stated in Figure 9 hold.*

Proof. **CliqueSquare-SC is HO-complete.** This is a direct corollary of Proposition 4.2. Since for any query q CliqueSquare-SC computes $\mathcal{P}(q)$, it therefore computes all the optimal plans for q.

CliqueSquare-SC$^+$ is HO-partial. First let us show that CliqueSquare-SC$^+$is not HO-complete.

We show that SC$^+$ is not SO-complete based on the example of the query in Figure 10. SC$^+$ can produce only one plan for this query, joining $\{t_1, t_2\}$, and $\{t_2, t_3\}$ in first level and then joining the resulting two intermediate relations in the next level. In contrast, SC is allowed to consider partial cliques, thus it may also build another SO plan as follows: choose as first cover $\{\{t_1, t_2\}, \{t_3\}\}$, and in the subsequent stage join the result of $t_1 \bowtie t_2$ with t_3. SC$^+$ cannot build this plan.

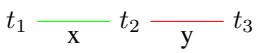


Figure 10: Query for which CliqueSquare-SC$^+$fails to find all SO plans.

CliqueSquare-SC is HO-partial. Now, let us show that for any stage optimal plan for a query q, CliqueSquare-SC$^+$computes a plan for q with same height, therefore it is HO-partial. This follows from the HO-completeness of CliqueSquare-SC. Let p be any stage optimal plan for a query q, built by CliqueSquare-SC. Consider the plan p' resulting from applying successfully to p the following changes:

1. pushing completely up the selection and projection operators;
2. starting from level 1 of p up to $h(p)$, replace *each* join operator resulting from a non-maximal clique for a given variable by the join operator obtained from the maximal version of this clique. Then add a projection operator on top of this maximal-clique join, to restrict its output to exactly the attributes that the original join used to return, and move this projection operator completely up.

Observe that p' also computes the answer to q (because no matter how much larger the newly introduced joins are, all the extra predicates that they bring were also going to be enforced in p) and that p and p' have the same height. Since p' is obtained from decompositions made of maximal-cliques only, p' is in the output of CliqueSquare-SC$^+$.

CliqueSquare-MSC is not HO-complete. Consider the query depicted in Figure 11. The only plan MSC produces for this query is shown in Figure 12. However, the plan shown in Figure 13 is also SO. This counterexample demonstrates that MSC is SO-partial.

CliqueSquare-MSC is HO-partial. Now, let us show that for any stage optimal plan for a query q, CliqueSquare-MSC computes a plan for q with same height, therefore it is HO-partial.

We first introduce the following notions for a Join operator op at level l in a plan p:

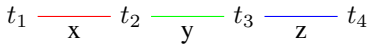


Figure 11: Query QX illustrating that minimum covers may lead to missing plans.

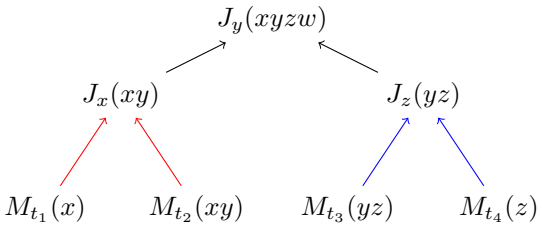


Figure 12: Logical plan for the query QX using minimum-cover decompositions.

- 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.

Let p be an HO plan produced by CliqueSquare-SC. (Recall that CliqueSquare-SC is HO-complete, thus it computes all the HO plans.) We next show that if *up to level* l, for $1 < l < h(p)$, the nodes of p result from an MSC decomposition based on the nodes one level below, then we can build from p a plan p' of the same height, computing the same output as p (thus computing q), and whose nodes are obtained from MSC clique decompositions *up to level* $l+1$. Applying this process repeatedly on p, then on p' etc. eventually leads to an MSC plan computing q.

Observe that at the level $l = 1$, p has only Match operators for the input relations. Thus, at $l = 1$, any p produced by CliqueSquare-SC coincides with any HO plan, because the leaf operators are the same in all plans. Similarly, at $l = h(p)$, the plan p consists of a single Join operator, thus the level $h(p)$ *is* indeed obtained by an MSC clique decomposition.

Now, let us consider a level l, $1 < l < h(p)$, the first level of p from 2 up to $h(p) - 1$ *not* resulting from an MSC clique decomposition based on the operators at the previous level $l-1$.

We build from p a plan p' computing q, with the same height, and resulting from MSC clique decompositions up to its level $l+1$, as follows.

Let d be one of the MSC clique decompositions corresponding to level $l-1$ of p. Let p' be a copy of the plan p up to $l-1$, and having at level l the Join operators corresponding to d. Observe that the connections between the operators from level $l-1$ and l in p' are determined by d, as they are built by CliqueSquare's function `createQueryPlans`.

Now, let us show how to connect the level l of p' to (a copy of) the level $l+1$ of p, so as to make p' identical to p level-by-level between $l+1$ and $h(p)$.

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'. Observe that the operators in $par(op)$ in p' are guaranteed to contain all the variables in op, because (i) any node has at most the variable present in all its parents (thus, grandparents etc.)[2] and (ii) op and $gp(op)$ were

[2]We say "at most" because in p, some variables present in the parent may have been projected away at an upper level. However, for simplicity, we ignore projections throughout the proof; it is easy to see that one can first pull up all projections from p, then build p' out of p as we explain, and finally push back all necessary projections on p', with no impact on the number of stages.

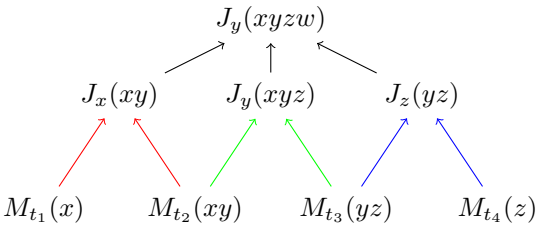


Figure 13: SO plan for the query QX obtained from non-minimum decompositions.

connected in p. Because all the input variables of op are provided by $par(op)$ in p', the join predicates encoded by op can be applied in p' exactly as in p (all operators at level $l+1$, are unchanged between p and p'). If the nodes in $par(op)$ in p' bring some variables not in p, we project them away prior to connecting the $par(op)$ operators to op in p'.

The plan p' satisfies the following: (i) it is syntactically correct, i.e., all operators have legal inputs and outputs, (ii) it computes q because, by construction, it is a CliqueSquare-SC HO-plan for q, and (iii) it is based on MSC decompositions up to level l (thus, one step higher than p). This concludes our proof.

CliqueSquare-MSC$^+$ is not HO-complete. CliqueSquare-MSC$^+$ is not HO-complete because (i) CliqueSquare-MSC is not HO-complete and (ii) CliqueSquare-MSC$^+$ outputs a subset of the plans produced by CliqueSquare-MSC (Proposition 4.1).

CliqueSquare-MSC$^+$ is HO-partial. Now, let us show that for any stage optimal plan for a query q, CliqueSquare-MSC$^+$ computes a plan for q with same height, therefore it is HO-partial. This follows from the fact that CliqueSquare-MSC is HO-partial. Let p be any stage optimal plan for a query q that is computed by CliqueSquare-MSC. Consider the plan p' resulting from applying successfully to p the following changes:

1. pushing completely up the selection and projection operators,
2. Starting from level 1 of p up to $h(p)$, replace each join operator resulting from a non-maximal clique for a given variable by the join operator resulting from the maximal version of this clique. Then add a projection operator on top of it to output the same relation as the previous join operator, and move this projection operator completely up.

Observe that p' computes the answer to the query, too, because no matter how much larger the newly introduced joins are, all predicates they bring are enforced at some point in p, too. Further, p and p' have the same height. Since p' is obtained from minimum decompositions (picked by CliqueSquare-MSC) now made of maximal-cliques only, p' is in the output of CliqueSquare-MSC$^+$.

MXC$^+$, XC$^+$, MXC and XC are HO-lossy. Consider the query shown in Figure 14. An exact cover 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 the number of stages. For instance, using MSC$^+$, one can evaluate the query in Figure 14 in two stages: in the first stage, the cliques $\{t_1, t_2\}$, $\{t_2, t_3\}$, $\{t_2, t_4\}$ are processed; in the second stage, all the results are joined together using the common variables xyz. On the other hand, any plan built only from exact covers requires an extra stage: 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. This requires an extra stage in order to finish processing the query. □

When MXC$^+$ and XC$^+$ fail. It turns out that the CliqueSquare algorithm based on the MXC$^+$ and

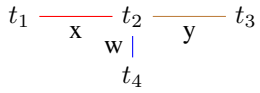


Figure 14: Query on which XC CliqueSquare variants are HO-lossy.

XC$^+$ may fail to find *any* plan for some queries, such as the one shown in Figure 10. For this query, the maximal clique decomposition returns the cliques $\{t_1, t_2\}$, $\{t_2, t_3\}$, out of which no exact cover of the query nodes can be found. In turn, this translates into CliqueSquare-MXC$^+$ and CliqueSquare-XC$^+$ failing to find a query plan! Thus, we do not consider MXC$^+$ and XC$^+$ further.

4.5 Time complexity of the optimization algorithm

Through some simplification, we study the complexity of the CliqueSquare algorithm by focusing only on the total number of clique reductions performed. While there are many other operations involved, the decompositions applied by the algorithm are the main factor determining how many computations are overall performed.

Let n be the number of nodes in the variable graph, and let $T(n)$ denote the number of clique reductions. The size of the problem (number of nodes) reduces at every stage (every recursive call); the reduction rate largely depends on the chosen decomposition method. Thus, we analyse the complexity of each of our eight option separately.

Recall that our decomposition methods can be classified in two major categories: (i) those based on minimum set covers only, and (ii) those using minimum or non-minimum covers.

Decompositions based on minimum covers. The decompositions using minimum set covers, namely MXC$^+$, MSC$^+$, MXC, and MSC, reduce the size of the graph by a factor of at least 2 at each call, as we explain below:

- Since we only consider connected query graphs, a graph of n nodes has at least $n-1$ edges.
- This graph admits at least one clique; if it has exactly one, the graph size is divided by n in one stage of decomposition.
- At the other extreme, assuming each edge is labeled with a different variable; in this case, selecting $\lceil (n-1)/2 \rceil$ edges is guaranteed to lead to a minimum cover. Thus, in the next (recursive) call, the graph will have at most $\lceil (n-1)/2 \rceil$ nodes, and the reduction divides the size of the problem by a factor of 2.

We denote by $D(n)$ the number of possible decompositions. Given that at each step the algorithm performs as many reduction as there are possible decompositions, the following recurrence relation can be derived:

$$T(n) \leq D(n)T(\lceil (n-1)/2 \rceil) \tag{1}$$

where $T(1) = 1$.

Decompositions based on any covers. For the decompositions that are not using minimum covers, namely XC$^+$, SC$^+$, XC, SC, the size of the graph in the worst case is smaller by 1 (this follows from the Definition 3.3). In this case, the recurrence relation is:

$$T(n) \leq D(n)T(n-1) \tag{2}$$

where $T(1) = 1$.

The number of decompositions $D(n)$ depends on the graph and the chosen method. The first parameter affecting the number of decompositions $D(n)$ is the **set of cliques**.

Given a query q and its variable graph G_V, the join variables JV of q determine the number of maximal/partial cliques of the graph (we only consider non-trivial queries with at least one join variable, $|JV| \geq 1$).

Counting maximal cliques. The number of maximal cliques in the graph is equal to the number of join variables. When $|JV| = 1$, there is exactly one maximal clique.

Lemma 4.1. *A variable graph G_V has at most $2n + 1$ maximal cliques.*

The proof is trivial since any conjunctive query q cannot have more than $2n + 1$ distinct variables. Thus, the variable graph at any stage does not have nodes that contain new variables so the number of maximal cliques is bound by the number of distinct variables existing in the query.

Counting partial cliques. Let $c\ell_u$ be the maximal clique corresponding to a variable $u \in JV$; from the definition of partial cliques it follows that the number of all non-empty partial cliques of $c\ell_u$ is equal to $2^{|cl_u|} - 1$. For two variables v_1, $v_2 \in JV$, the maximal cliques $c\ell_{v_1}$ and $c\ell_{v_2}$ may have some common nodes; in this case, some partial cliques for v_1 are also partial cliques for v_2.

To count the clique overlappings the following factor is introduced:

$$OF = \sum_{v_1,v_2 \in JV : c\ell_{v_1} \cap c\ell_{v_2} \neq \emptyset} 2^{|c\ell_{v_1} \cap c\ell_{v_2}|}$$

Thus, the total *number of partial cliques* is given by:

$$\sum_{u \in JV} (2^{|cl_u|} - 1) - OF \tag{3}$$

Lemma 4.2. *A variable graph G_V has at most $2^n - 1$ partial cliques.*

The proof is trivial since even in the case where we can take all combinations of nodes as partial cliques this number cannot exceed the power set.

The second parameter that affects the number of decompositions $D(n)$ is the decomposition method. Below we establish the complexity of CliqueSquare algorithm for each decomposition method, considering the worst case scenario for $D(n)$.

Complexity of CliqueSquare-SC. Out of $2^n - 1$ partial cliques, we search for set covers of size at most $n - 1$. The maximum number of decompositions (the maximum number of covers) satisfies:

$$D(n) \leq \sum_{k=1}^{n-1} \binom{2^n - 1}{k} \tag{4}$$

The equation above corresponds to the total number of sets that will be examined in CLIQUEDECOMPOSITION function. Only some of them are valid covers, thus the above is a very rough bound. This is easy to see for example considering the case where $k = 1$. The candidate covers according to the equation will be $2^n - 1$, but only one of them is really a cover.

Complexity of CliqueSquare-MSC. A variable graph of n nodes is sure to contain a cover of size $\lceil n/2 \rceil$. MSC decomposition searches for minimum covers in the graph, thus if there exists a cover with size $\lceil n/2 \rceil$ there is no need to consider bigger covers. Furthermore, a cover smaller than $\lceil n/2 \rceil$ may exist, but since we consider the worst case (where the number of decompositions is maximized), we rely on $\lceil n/2 \rceil$ as the size of the minimum set cover. Based on this, the number of decompositions satisfies:

$$D(n) \leq \binom{2^n - 1}{\lceil n/2 \rceil} \tag{5}$$

The worst case is constructed by taking the worst case for the number of cliques and the worst case for the size of the minimum set cover. These two worst cases, though, might never appear together in practice. For example, the $2^n - 1$ partial cliques appear when we have a single clique query, but in that case the minimum set cover is exactly one. On the other hand, a minimum set cover of size $n/2$ is typical for chain queries, for which the number of partial cliques is $2n - 1$.

Complexity of CliqueSquare-XC. When there are $2^n - 1$ partial cliques, there is exactly 1 maximal clique; in this case the exact cover problem is equivalent with the problem of finding the partitions of a set. The number of ways to partition a set of n objects into k non-empty subsets is described by *Stirling partition numbers of the second kind* and is denoted as $\left\{ {n \atop k} \right\}$. The total number of decompositions satisfies:

$$D(n) \leq \sum_{k=0}^{n-1} \left\{ {n \atop k} \right\} \tag{6}$$

Complexity of CliqueSquare-MXC. Given a variable graph G_V with n nodes, in the worst case the size of the minimum set cover is equal with $\lceil n/2 \rceil$ (the graph is connected). In addition we have seen that for exact cover decompositions the maximum number cannot exceed the number given by Equation 6. Using again the equivalence of the exact cover problem with set partitioning, we are interested in non-empty partitions of size $k = \lceil n/2 \rceil$. The total number of decompositions satisfies:

$$D(n) = \left\{ {n \atop \lceil n/2 \rceil} \right\} \tag{7}$$

Again, the above upper bound is based on two mutually exclusive worst cases: 1-clique queries for which we the number of exact covers is given by the Stirling numbers, and chain queries which maximize the size of the set cover ($n/2$).

Complexity of CliqueSquare-SC$^+$. A variable graph of n nodes has at most $2n + 1$ maximal cliques (Lemma 4.1). Similarly to SC decompositions, we search for set covers of size at most $n - 1$. The total number of decompositions satisfies:

$$D(n) \leq \sum_{k=1}^{n-1} \binom{2n+1}{k} \tag{8}$$

Complexity of CliqueSquare-MSC$^+$. As before the maximum number of maximal cliques is $2n + 1$ (Lemma 4.1). Since the graph is connected, in the worst case, the size of the minimum set cover is $\lceil n/2 \rceil$. Thus, the total number of decompositions satisfies:

$$D(n) \leq \binom{2n+1}{\lceil n/2 \rceil} \tag{9}$$

Complexity of CliqueSquare-XC$^+$. Recall again Lemma 4.1. Due to the clique overlap, for every clique that is chosen, at least one other clique becomes ineligible. In practice the cliques may overlap even more but we are interested only in the worst case (the ones that generates the most decompositions). Assuming that there are $2n + 1$ maximal cliques, we can only pick a maximum of $n + 1$ [3] cliques, since any extra selection will make some clique ineligible. From the above, we conclude that the total number of decompositions satisfies:

$$D(n) \leq \sum_{k=1}^{n-1} \binom{n+1}{k} \tag{10}$$

[3] Notice that $\lceil (2n+1)/2 \rceil = \lceil n + 1/2 \rceil = n + 1$ since $n \in \mathbb{N}^+$

Complexity of CliqueSquare-MXC$^+$. Similar with XC$^+$ there are at most $n+1$ eligible cliques. Given that we are looking for a minimum cover, the size of the cover is constant (similar with the other options supporting minimum covers). Again the binomial coefficient is used to estimate the worst case and the total number of decompositions satisfies:

$$D(n) \leq \binom{n+1}{\lceil n/2 \rceil} \tag{11}$$

Figure 8 summarizes the number of decompositions for each option when considering the worst case scenarios. Note that the worst cases are not reached by all algorithms on the same queries, therefore Figure 8 does not provide an easy way to compare the efficiency of the optimization variants. We have obtained interesting comparison results experimentally by testing all variants against a large set of synthetic queries (see Section 6.2).

5 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 5.1), based on which queries are evaluated. We then present the translation of logical plans into physical plans (Section 5.2), then show how a physical plan is mapped to MapReduce jobs (Section 5.3) and finally introduce our cost model (Section 5.4).

5.1 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 [29]), 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 [5]. Triples that share the same value in any position (s, p, o) are located within the same compute node.
(2) Then, unlike [5], 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 [18] and also resembles the vertical RDF partitioning proposed in [1] 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 [7], 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.

5.2 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 [4] that joins its n inputs on their common attribute set A.
• **Map Shuffler**, $MF_A(op)$, is the repartition phase of a repartition join [4] 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 [4]. 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

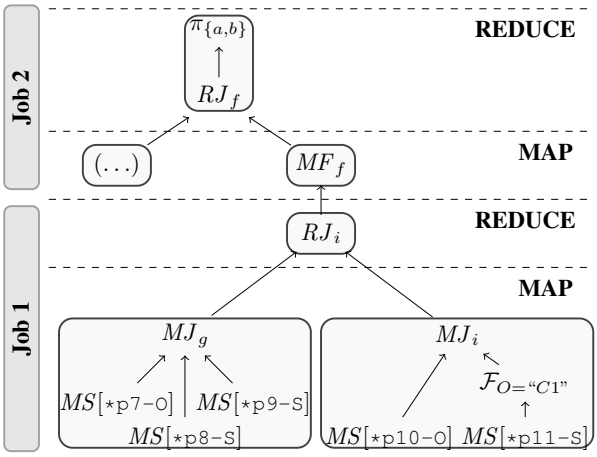


Figure 15: Part of Q1 physical plan and its mapping to MapReduce jobs.

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 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 15 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.

5.3 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 15 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.

5.4 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).

To estimate $tw(p)$, we introduce the *own costs* of each operator o as follows: $c_{io}(o)$ is the I/O cost of operator o, $c_{cpu}(o)$ is its CPU incurred cost, while $c_{net}(op)$ is its data transfer cost.

The cost of a scan operator, MS, is mainly the I/O operations for reading the corresponding file from HDFS, the cost of a filter operator, $\mathcal{F}_c$, is mainly the CPU cost for checking whether condition c is satisfied. π_A operator involves only CPU processing for removing the appropriate attributes in A. Regarding the cost of a map shuffler operator, MF_A, the I/O cost for reading intermediate results from the HDFS is measured, as well as the I/O cost for forwarding the results to the reducer (writing the results to disk). The cost of a map join operator, MJ_A, occurs from CPU operations for joining locally the input relations on attributes A and from I/O writes to disk (the joining results are written to disk before shuffling). Finally, a reduce join operator, RJ_A, entails network load for transferring the intermediate results to the reducers for attributes A, CPU cost for the computation of the join results and I/O cost for writing the results to disk. To sum up, the cost of each operator consists of the following individual costs:

- $c(MS) = c_{io}(MS)$
- $c(\mathcal{F}_{con}) = c_{cpu}(\mathcal{F}_{con})$
- $c(\pi_A) = c_{cpu}(\pi_A)$
- $c(MF_A) = c_{io}(MF_A)$
- $c(MJ_A) = c_{cpu}(MJ_A) + c_{io}(MJ_A)$
- $c(RJ_A) = c_{net}(RJ_A) + c_{cpu}(RJ_A) + c_{io}(RJ_A)$

The individual costs can be estimated as follows:

- $c_{io}(MS[FS]) = \sum_{f \in FS} |f| \times c_{read}$
- $c_{io}(MF_A(op)) = |op| \times c_{read} + |op| \times c_{write}$
- $c_{io}(MJ_A[op_1, \ldots, op_n]) = |op_1 \bowtie_A \ldots \bowtie_A op_n| \times c_{write}$
- $c_{io}(RJ_A[op_1, \ldots, op_n]) = |op_1 \bowtie_A \ldots \bowtie_A op_n| \times c_{write}$
- $c_{cpu}(\mathcal{F}_{con}(op)) = |op| \times c_{check}$
- $c_{cpu}(\pi_A(op)) = |op| \times c_{check}$
- $c_{cpu}(MJ_A(op_1, \ldots, op_n]) = c_{join}(op_1 \bowtie_A \ldots \bowtie_A op_n)$
- $c_{cpu}(RJ_A(op_1, \ldots, op_n)) = c_{join}(op_1 \bowtie_A \ldots \bowtie_A op_n)$
- $c_{net}(RJ_A(op_1, \ldots, op_n)) = (|op_1| + \ldots + |op_N|) \times c_{shuffle}$

where $|R|$ denotes the cardinality of R and R is a set of tuples. c_{read} and c_{write} the time to read and write one tuple from and to disk, respectively. $c_{shuffle}$ represents the time to transfer one tuple from one node to another through the network and $c_{join}(op_1 \bowtie_A ... \bowtie_A op_N)$ the cost of the join. Finally c_{check} is the time spend on performing a simple comparison on a part of the tuple.

While MapReduce program performance can be modeled at much finer granularity [20, 15], the simple model above has been sufficient to guide our optimizer well, as our experiments demonstrate next.

6 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 4.3 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.

6.1 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 [12] benchmark, since it has been extensively used in similar works such as [18, 17, 38, 27]. 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 [17, 27, 23], 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 [27] 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 Appendix A.

6.2 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.

Setup. We use the generator of [10] 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 16 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 4.3, 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 17 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

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

Figure 16: 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%

Figure 17: 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

Figure 18: 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%

Figure 19: Average uniqueness ratio per algorithm and query shape.

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 18. 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 19 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 **??** 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.

6.3 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 5.4. We translate all logical plans into MapReduce jobs as described in Section 5 and execute them on our CSQ prototype.

Comparison. Figure 20 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 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.

6.4 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 [13, 38], 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.

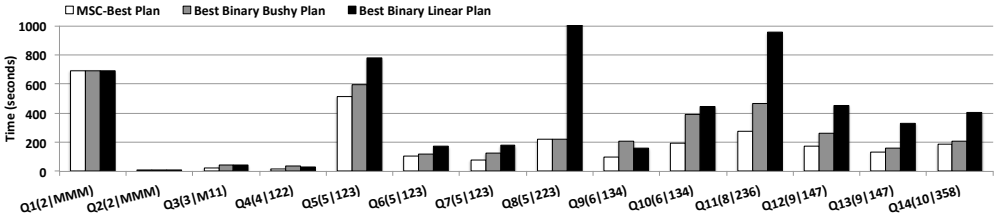


Figure 20: Plan execution time (in seconds) comparison between MSC-plans, bushy-plans, and linear plans for LUBM10k.

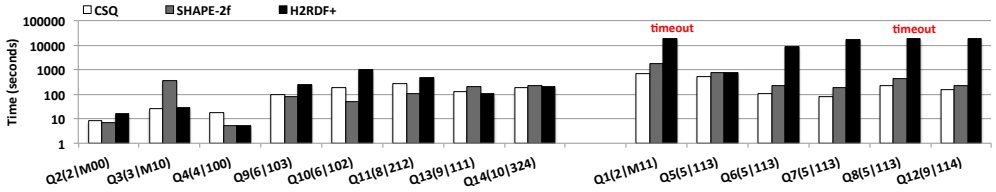


Figure 21: Query evaluation time comparison: CSQ, SHAPE and H_2RDF+.

Systems. We pick SHAPE [23] and H_2RDF+ [27], since they are the most efficient RDF platforms based on MapReduce; the previous HadoopRDF [18] is largely outperformed by H_2RDF+ [27] and [17] is outperformed by [23]. 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 [25]. 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 21 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 5).

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 thus, lower query response times.

7 Conclusion

Numerous distributed platforms have been proposed to handle large volumes of RDF data [21], 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. 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.

A SPARQL Queries

For completeness, we include the SPARQL queries used in our evaluation. For the sake of simplicity some constants appear abbreviated. The characteristics of the queries are summarized in Figure 22: number of triple patterns ($\#tps$), number of join variables ($\#jv$) and result cardinality for LUBM10k ($|Q|_{10k}$. The indicator (*original*) appears next to the query name when the query belongs to the default LUBM benchmark.

Queries	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
$\#tps$	2	2	3	4	5	5	5
$\#jv$	1	1	1	2	3	3	3
$\|Q\|_{10k}$	3.7B	1900	282.2K	93	56.1M	7.9M	25.1M
Queries	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**
$\#tps$	5	6	6	8	9	9	10
$\#jv$	3	3	3	4	4	4	5
$\|Q\|_{10k}$	504.3M	2528	439.9K	1647	12.5M	871	1413

Figure 22: Characteristics of the LUBM queries used in the experiments.

Q1: SELECT ?P ?S WHERE { ?P ub:worksFor ?D . ?S ub:memberOf ?D . }

Q2(original): SELECT ?X WHERE { ?X rdf:type ub:AssistantProfessor . ?X ub:doctoralDegreeFrom <http://www.University0.edu> }

Q3: SELECT ?P ?S WHERE { ?P ub:worksFor ?D . ?S ub:memberOf ?D . ?D ub:subOrganizationOf <University0> }

Q4(original): SELECT ?X ?Y WHERE { ?X rdf:type ub:Lecturer . ?Y rdf:type ub:Department . ?X ub:worksFor ?Y . ?Y ub:subOrganizationOf <University0> }

Q5: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:UndergraduateStudent . ?Y rdf:type ub:FullProfessor . ?Z rdf:type ub:Course . ?X ub:takesCourse ?Z . ?Y ub:teacherOf ?Z }

Q6: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:UndergraduateStudent . ?Y rdf:type ub:FullProfessor . ?Z rdf:type ub:Course . ?X ub:advisor ?Y . ?Y ub:teacherOf ?Z }

Q7: SELECT ?X ?Y ?Z WHERE { ?X a ub:GraduateStudent . ?Z ub:subOrganizationOf ?Y . ?X ub:memberOf ?Z . ?Z a ub:Department . ?Y a ub:University . }

Q8: SELECT ?X ?Y ?Z WHERE { ?X a ub:GraduateStudent . ?X ub:undergraduateDegreeFrom ?Y. ?Z ub:subOrganizationOf ?Y . ?Z a ub:Department . ?Y a ub:University . }

Q9(original): SELECT ?X ?Y ?Z WHERE { ?X a ub:GraduateStudent . ?X ub:undergraduateDegreeFrom ?Y. ?Z ub:subOrganizationOf ?Y . ?X ub:memberOf ?Z . ?Z a ub:Department . ?Y a ub:University . }

Q10(original): SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:Undergraduate Student . ?Y rdf:type ub:FullProfessor . ?Z rdf:type ub:Course . ?X ub:advisor ?Y . ?X ub:takesCourse ?Z . ?Y ub:teacherOf ?Z }

Q11: SELECT ?X ?Y ?E WHERE { ?X rdf:type ub:Undergraduate Student . ?X ub:takesCourse ?Y . ?X ub:memberOf ?Z . ?X ub:advisor ?W . ?W rdf:type ub:FullProfessor . ?W ub:emailAddress ?E . ?Z ub:subOrganizationOf ?U . ?U ub:name "University3" }

Q12: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:FullProfessor . ?X ub:teacherOf ?Y . ?Y rdf:type ub:GraduateCourse . ?X ub:worksFor ?Z . ?W ub:advisor ?X . ?W rdf:type ub:GraduateStudent . ?W ub:emailAddress ?E . ?Z rdf:type ub:Department . ?Z ub:subOrganizationOf ?U }

Q13: SELECT ?X ?Y ?Z WHERE ?X rdf:type ub:FullProfessor . ?X ub:teacherOf ?Y . ?Y rdf:type ub:GraduateCourse . ?X ub:worksFor ?Z . ?W ub:advisor ?X . ?W rdf:type ub:GraduateStudent . ?W ub:emailAddress ?E . ?Z rdf:type ub:Department . ?Z ub:subOrganizationOf <University0>

Q14: SELECT ?X ?Y ?Z WHERE { ?X rdf:type ub:FullProfessor . ?X ub:teacherOf ?Y . ?Y rdf:type ub:GraduateCourse . ?X ub:worksFor ?Z . ?W ub:advisor ?X . ?W rdf:type ub:GraduateStudent . ?W

ub:emailAddress ?E . ?Z rdf:type ub:Department . ?Z ub:subOrganizationOf ?U . ?U ub:name “University3” }

References

[1] Daniel J. Abadi, Adam Marcus, and Barton Data. Scalable Semantic Web Data Management using Vertical Partitioning. In *VLDB*, 2007.

[2] Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC*, 2007.

[4] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, 2010.

[5] Min Cai, Martin R. Frank, Baoshi Yan, and Robert M. MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics*, 2(2):109–130, 2004.

[6] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, 2005.

[7] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 2011.

[8] Luis Galarraga, Katja Hose, and Ralf Schenkel. Partout: A Distributed Engine for Efficient RDF Processing. Technical Report: CoRR abs/1212.5636, 2012.

[9] Alan Gates, Jianyong Dai, and Thejas Nair. Apache Pig’s optimizer. *IEEE Data Eng. Bull.*, 36(1), 2013.

[10] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in semantic web databases. *PVLDB*, 5(1), 2012.

[11] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, 2014.

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

[13] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: a Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In *SIGMOD*, 2014.

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

[15] Herodotos Herodotou, Fei Dong, and Shivnath Babu. MapReduce Programming and Cost-based Optimization? Crossing this Chasm with Starfish. *PVLDB*, 4(12), 2011.

[16] Katja Hose and Ralf Schenkel. WARP: Workload-Aware Replication and Partitioning for RDF. In *DESWEB*, 2013.

[17] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[18] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE TKDE*, 23(9), September 2011.

[19] David Huynh, Stefano Mazzocchi, and David R. Karger. Piggy Bank: Experience the Semantic Web inside your web browser. *J. Web Sem.*, 5(1), 2007.

[20] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1), 2010.

[21] Zoi Kaoudi and Ioana Manolescu. RDF in the Clouds: A Survey. The VLDB Journal, 2014.

[22] Richard Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. 1972.

[23] Kisung Lee and Ling Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14), September 2013.

[24] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):31, 2014.

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

[26] M. Tamer Özsu and Patrick Valduriez. *Distributed and Parallel Database Systems (3rd. ed.)*. Springer, 2011.

[27] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In *IEEE BigData*, 2013.

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

[29] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3rd. ed.)*. McGraw-Hill, 2003.

[30] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A Core of Semantic Knowledge. In *WWW*, 2007.

[31] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, 2010.

[32] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter A. Boncz. Heuristics-based query optimisation for SPARQL. In *EDBT*, 2012.

[33] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query Optimization for Massively parallel Data Processing. In *SOCC*, 2011.

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

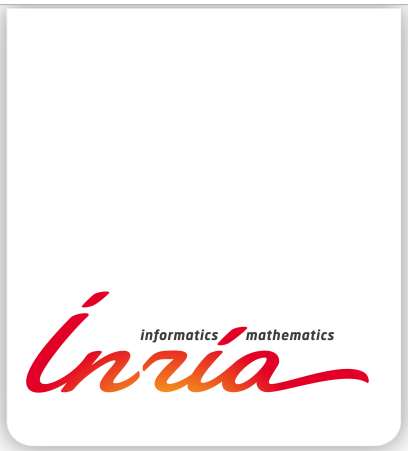
[35] Query graph model. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp`, 2010.

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

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

[38] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine for Web Scale RDF Data. In *PVLDB 2013*.

[39] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In *ICDE*, 2013.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399