A Map-Reduce algorithm for querying linked data based on query decomposition into stars[*]

Christos Nomikos
Department of Computer Science and Engineering
University of Ioannina, Ioannina, Greece
cnomikos@cs.uoi.gr

Manolis Gergatsoulis, Eleftherios Kalogeros, Matthew Damigos
Database & Information Systems Group (DBIS)
Department of Archives, Library Science and Museology
School of Information Science and Informatics
Ionian University, Corfu, Greece
manolis@ionio.gr

ABSTRACT

In this paper, we investigate the problem of efficient querying large amount of linked data using Map-Reduce framework. We assume data graphs that are arbitrarily partitioned in the distributed file system. Our technique focuses on the decomposition of the query posed by the user, which is given in the form of a query graph into star subqueries. We propose a two-phase, scalable Map-Reduce algorithm that efficiently results the answer of the initial query by computing and appropriately combining the subquery answers.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Query Processing*; C.2.4 [**Distributed Systems**]: Distributed Databases; D.1.3 [**Concurrent Programming**]: Distributed programming

General Terms

Algorithms, Theory, Experimentation

Keywords

Linked Data, Graph Querying, Map-Reduce, Distributed Processing, Cloud Computing, Semantic Web

[*]This research was supported by the project "Handling Uncertainty in Data Intensive Applications", co-financed by the European Union (European Social Fund - ESF) and Greek national funds, through the Operational Program "Education and Lifelong Learning", under the research funding program THALES.

1. INTRODUCTION

Linked data is a widespread method for publishing interlinked data, built upon a standard model used for data interchange, called RDF. As the amount of linked data is rapidly increasing the efficient management, analysis, and hence querying of large amount of linked data has become a significant challenge in many areas, such as learning analytics, digital libraries, and other applications analyzing linked open data. The efficient querying of large amount of data is also a significant challenge in many other information management areas, where parallel processing has been proved particularly effective in manipulating such an amount of data.

A well-established programming framework used for processing and managing large amount of data, in parallel, using a cluster of commodity machines is Map-Reduce [8]. A popular, open-source implementation is Apache Hadoop [1]. Boasting a simple, fault-tolerant and scalable paradigm, Hadoop has been established as dominant in the area of massive data analysis.

In this paper, we investigate the problem of efficient querying large amount of linked data using Map-Reduce framework, and extend our approach presented in [10]. In particular, we focus on the decomposition of the query posed by the user, which is given in the form of a query graph, into star subqueries and propose a two-phase, scalable Map-Reduce algorithm that efficiently results the answer of the initial query. Furthermore, we assume data graphs that are arbitrarily partitioned in the distributed file system. The first phase of our algorithm takes advantage of the star form of the sub-queries and focuses on evaluating the star subqueries over the input segments. The results of the sub-queries are emitted to the second phase, which combines them properly in order to produce the answers of the initial query.

2. RELATED WORK

During the last decade, the problem of efficiently querying large data graphs using Map-Reduce framework has been investigated in many research areas related to information management [9, 2, 12, 11, 16, 15, 13, 18, 5, 6, 19].

Evaluating SPARQL queries over RDF graphs by parallelizing the processing of the join and selection operators has received much attention [14, 12, 16, 15, 18]. The RDF triples can be either directly stored in files in the distributed file system (DFS) or stored in a distibuted database (e.g., HBase [16]). In [14], the authors propose an approach where the query plan is evaluated using a sequence of Map-Reduce phases. Initially, the relevant RDF triples are selected and then a sequence of joins is evaluated. Each operator in the plan is performed within a Map-Reduce phase, while the evaluation of the query requires multiple Map-Reduce phases, according to the number of operators of the given query. In [12] the RDF triples are stored in multiple DFS files, according to their predicates and objects, while only the relevant files are read from the DFS for each query. In H_2RDF system [16] data triples are stored in HBase. In order to answer a query, a sequence of joins is executed, which is obtained in a greedy manner. Other approaches have been proposed, which translate SPARQL queries to other query languages, such as JACL ([15]) and PigLatin ([13], [18]). In addition, [7] provides a detailed experimental analysis and comparison of the main NoSQL data stores for RDF processing. The systems that are used for storing the data are the following: Apache HBase, CumulusDB (a RDF data store built upon Cassandra) and Couchbase; while the 4store was used as a baseline, native and distributed RDF DBMS. Jena was used as a SPARQL query engine over HBase and Couchbase, Hive over HBase and Sesame for CumulusDB.

In [11] and [9], the RDF data has been initially partitioned in a predefined manner. [11] assumes that the RDF graph is vertex partitioned and its parts are stored with some triple replication, to ensure that for small queries, each answer can be computed using a single part of the graph. Larger queries are decomposed and the answers to the subqueries are joined by MapReduce jobs. In HadoopRDF [9], RDF triples with the same predicate are placed in the same part of the data graph, which is stored in a traditional triple store, such as Sesame. Queries are divided in the same way, so that each subquery can be answered by a single computer-node. The answers to the subqueries are merged by MapReduce jobs. The decomposition of the given query is also one of the main characteristics of our Map-Reduce algorithm presented in [10]. In that approach, however, the triples could be arbitrarily partitioned in the DFS and the queries are decomposed into paths. The proposed Map-Reduce algorithm evaluates the given query in two phases; one for evaluating the path subqueries and one for finding the total answers by combining the results of the subqueries.

The problem of querying large data graphs using Map-Reduce is also investigated in [2, 19, 6]. J. Cohen in [6] presents different approaches of finding subgraphs in large data graphs using Map-Reduce. In [19], the authors focus on the problem of querying triangles and propose one single-phase and one two-phase Map-Reduce algorithm for finding triangles in a graph whose edges have been arbitrarily distributed in DFS. Afrati et al. [2] investigate the cost of evaluating query graphs on data graphs using Map-Reduce, and proposed an approach of translating the graphs into conjunctive queries which in turn are evaluated in Map-Reduce using the approach proposed in [3].

Systems for quering RDF data that are distributed over the web, which adopt query decomposition, have also been proposed ([17], [4]). DARK [17] uses a service description language to maintain information about the data stored in various hosts, and uses these service descriptions for query planning and optimization. Avalanche [4] has a preprocessing phase that selects a set of candidate host, which are queried for statistical information. Next, the query execution phase brakes down the given query into subqueries (called molecules), which are bound to physical hosts by a plan generator. This phase terminates, when an adequate number of solutions have been found.

3. DATA AND QUERY GRAPHS

In this section we introduce the basic notions regarding our data model. We start with the definition of data and query graphs:

Definition 1. Let U_{so} and U_p be two disjoint infinite sets of URI references and let L be an infinite set of (plain) literals[1]. An element $(s, p, o) \in U_{so} \times U_p \times (U_{so} \cup L)$ is called a *data triple*. In a data triple (s, p, o), s is called the *subject*, p the *predicate* and o the *object*. A *data graph* is a non-empty set of data triples. A data graph G' is a *subgraph* of a data graph G if $G' \subseteq G$.

Definition 2. Let U_{so} and U_p be two disjoint sets of URI references, let L be a set of (plain) literals and let V be a set of variables. An element $(s, p, o) \in (U_{so} \cup V) \times U_p \times (U_{so} \cup L \cup V)$ is called a *query triple*. A *query graph* (or simply a *query*) is a non-empty set of query triples. The *output pattern* $O(Q)$ of a query graph Q is the sequence $(X_1, \ldots, X_n)$ of the variables appearing in Q. A query graph Q' is a *subquery* of a query graph Q if $Q' \subseteq Q$.

Notice that, queries with variables in the place of predicates are not allowed. The set of *nodes* $nd(G)$ of a data graph G consists of all the elements of $U_{so} \cup L$ that occur in the triples of G. The set of *arc labels* $al(G)$ of a data graph G consist of all the elements of U_p that occur in the triples of G. The set of nodes $nd(Q)$ and the set of arc labels $al(Q)$ of a query Q are defined in an analogous way.

A class of queries of a special form (namely star queries) play an important role in this paper.

Definition 3. A query Q is called a *star query* if there exists a node $c \in nd(Q)$ such that for every triple $t \in Q$ it is either $t = (c, p, v)$ or $t = (v, p, c)$ for some node $v \in nd(Q)$ and some predicate $p \in al(Q)$. The node c is called the *central node* of Q.

It is convenient to use a graphical representation for data and query graphs. A node (subject or object) which is either a URI or a variable is represented as a rounded rectangle while an object which is a literal is represented by a rectangle. A triple (s, p, o) is represented by an arc from s to o, labeled with p. Moreover, we adopt the following conventions: strings with initial lowercase letters represent predicates, while strings with initial uppercase letters denote URIs. Literals are represented as strings enclosed in double quotes. Finally, variable names begin with the question mark symbol (?).

Example 1. Fig. 1 depicts a data graph G and a query graph Q. $\square$

[1]In this paper we do not consider typed literals

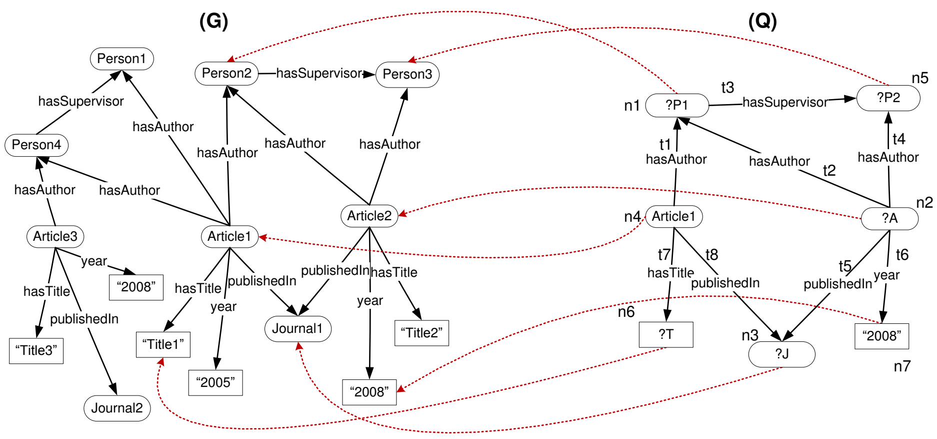

Figure 1: An embedding of the query graph Q in the data graph G.

In order to compute the answers to a query Q for a given data graph G, we need to find an appropriate correspondence between the nodes of Q and the nodes of G. This is formalized by the notion of embedding, defined as follows:

Definition 4. An *embedding* of a query graph Q in a data graph G is a total mapping $e : nd(Q) \rightarrow nd(G)$ with the following properties:

1. For each non-variable node $v \in nd(Q)$, it is $e(v) = v$.
2. For each triple $(s, p, o) \in Q$, $(e(s), p, e(o))$ is in G.

The tuple $(e(X_1), \ldots, e(X_n))$, where $(X_1, \ldots, X_n)$ is the output pattern of Q, is said to be an *answer* to the query Q.

Example 2. Fig. 1 shows an embedding of the query Q in data graph G. The answer obtained is $(?P1, ?A, ?J, ?P2, ?T)$ $= (Person2, Article2, Journal1, Person3, \text{“}Title1\text{”})$. □

4. DATA GRAPH PARTITIONING

Data graphs may consist of a huge number of data triples, stored in numerous computer nodes. In this section we define the notion of the partition of a data graph.

Definition 5. A *triple partition* of a data graph G is a tuple $\mathcal{P} = (G_1, \ldots, G_m)$ where $G_1, \ldots, G_m \subseteq G$, such that $\bigcup_i G_i = G$ and $G_i \cap G_j = \emptyset$, for all i, j, with $1 \leq i < j \leq m$. Subgraphs $G_1, \ldots, G_m$ are called the *graph segments*.

From the above definition it follows that graph segments in a triple partition of a graph G cannot share data triples; however, they may have common nodes.

Definition 6. Let $\mathcal{P} = (G_1, \ldots, G_m)$ be a triple partition of a data graph G. Then, a *border node* v of G_i, is a node that belongs to $nd(G_i) \cap nd(G_j) \cap U_{so}$ for some $j \neq i$. We denote by $B(G_i)$ the set of border nodes of G_i.

Example 3. (Continued from Example 2). In Fig. 2 we see a triple partition of the data graph G of Fig. 1. The shaded nodes correspond to the border nodes between the graph segments. Consider now the query graph Q appearing in the right part of Fig. 2. It is easy to see that we cannot obtain the solution appearing in Example 2 by finding an embedding of Q in a segment of G appearing in Fig. 2 (as such an embedding does not exist). □

5. QUERY DECOMPOSITION

In this section we define the notion of query decomposition and we show how it can be used in order to compute all the answers to a given query.

Definition 7. A *query decomposition* of a query graph Q is a tuple $\mathcal{F} = (Q_1, \ldots, Q_n)$ such that $Q_1, \ldots, Q_n \subseteq Q$ and $\bigcup_i Q_i = Q$. A query decomposition is *non-redundant* if $Q_i \cap Q_j = \emptyset$ for each pair i, j such that $1 \leq i < j \leq n$. A *branching node* in Q is a node that belongs to $nd(Q_i) \cap nd(Q_j)$ for a pair i, j, where $i \neq j$. By $B(Q)$ we denote all branching nodes of Q.

In this paper, our aim is to construct the embeddings of a query Q in G, by appropriately combining embeddings of its subqueries in G. Two embeddings can be combined only in the case that they agree in the values of nodes that are common in the corresponding subqueries. The above requirement is formalized by the following definition:

Definition 8. Let Q_1, Q_2 be two query graphs and let e_1, e_2 be embeddings of Q_1, Q_2 respectively in a data graph G. We say that e_1 and e_2 are *compatible* if for every $v \in nd(Q_1) \cap nd(Q_2)$ it is $e_1(v) = e_2(v)$. The *join* of e_1 and e_2 is the embedding e of $Q_1 \cup Q_2$ in G defined as follows:

$$e(v) = \begin{cases} e_1(v) & \text{if } v \in nd(Q_1) \\ e_2(v) & \text{otherwise} \end{cases}$$

It is not hard to see that the join operation is commutative and associative. Therefore, we can refer to the embedding resulting by the join of n mutually compatible embeddings without ambiguity. The following theorem can be easily proved by an induction on the number n of subqueries in the decomposition of a query Q.

THEOREM 1. *Let $\mathcal{F} = (Q_1, \ldots, Q_n)$ be a query decomposition of a query graph Q and let G be a data graph. Then e is an embedding of Q in G if and only if there exist mutually compatible embeddings $e_1, \ldots, e_n$ of $Q_1, \ldots, Q_n$ in G such that the join of $e_1, \ldots, e_n$ is e.*

The above theorem implies that in order to compute the answers to a given query for a data graph G, we can decompose the query into subqueries that belong to a certain class

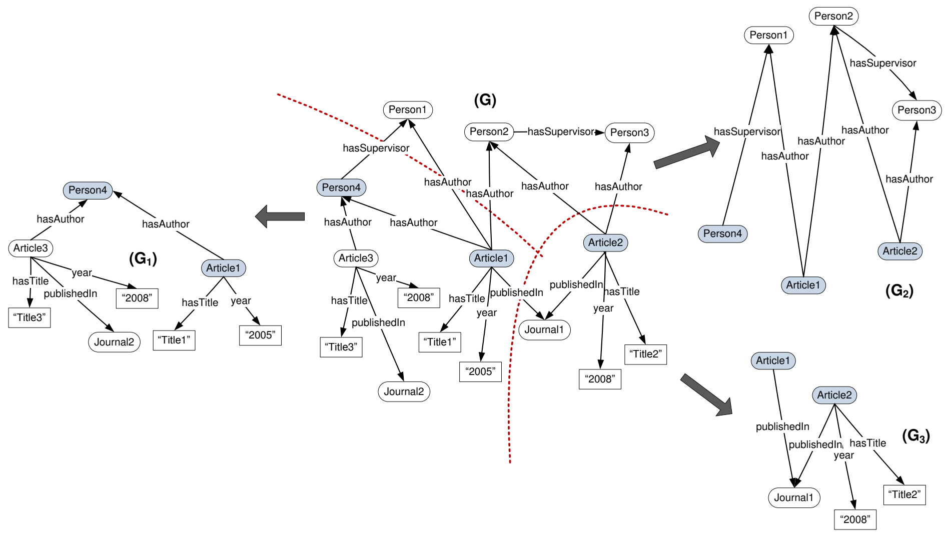

Figure 2: triple partition of the data graph G of Fig. 1.

C, compute the embeddings of the subqueries in G (which may be an easier task due to the special form of the subqueries) and then join these embedding to obtain the desired result. However, given a target class of queries C, it may not be always possible to decompose an arbitrary query Q into subqueries that belong to C (for example this is the case if C is the class of path queries of length 3). Nevertheless, for every query there exist a non-redundant decomposition into star subqueries. This follows trivially from the fact that every query that consists of a single triple is a star query (with either the subject or the object being the central node). We next present a more general result, relating the decompositions of a query graph into star subqueries to the node covers of this query graph.

Definition 9. Let Q be a query graph. A set of nodes $V \subseteq nd(Q)$ is called a *node cover* of Q if for every triple $(s, p, o) \in Q$, it is either $s \in V$ or $o \in V$. A node cover V is *minimal* if no proper subset of V is a node cover.

LEMMA 1. *Let Q be a query graph and let $V = \{v_1, \dots v_k\}$ be a minimal node cover of Q. For each v_i define the star query $Q_{v_i} = \{t \in Q \mid t = (s, p, v_i)\} \cup \{t \in Q \mid t = (v_i, p, o)$ and $o \notin V\}$. Then $\mathcal{F} = (Q_{v_1}, \dots, Q_{v_k})$ is a non-redundant decomposition of Q.*

Therefore, if a set of nodes is a minimal node cover of a query Q, then its elements are the central nodes of the star subqueries in some non-redundant decomposition of Q.

Conversely, in any decomposition (redundant or not) of a query into stars, the set of the central nodes is a node cover (not necessarily minimal).

LEMMA 2. *Let Q be a query graph, let $\mathcal{F} = (Q_1, \dots, Q_k)$ be a decomposition of Q such that $Q_1, \dots, Q_k$ are star queries and let $c_1, \dots, c_k$ be their central nodes. Then $\{c_1, \dots, c_k\}$ is a node cover of Q.*

Example 4. In Fig. 3 we see a decomposition of the query Q into three star queries Q_1, Q_2 and Q_3, which is obtained by the construction of Lemma 1, using the minimal node cover $\{n_4, n_2, n_5\}$ of Q. □

To summarize, suppose that a data graph G is partitioned into m segments $G_1, \dots, G_m$, that are stored in different computer nodes. The above discussion suggests the following query evaluation strategy:
Step 1: Decompose query Q into star subqueries $Q_1, \dots, Q_n$.
Step 2: Compute all possible embeddings of each triple in Q in every segment G_i of G.
Step 3: For each subquery Q_j, collect the embeddings of all triples in Q_j and join compatible embeddings in all possible ways to compute the embeddings of Q_j in G.
Step 4: Join compatible embeddings $Q_1, \dots, Q_n$ in all possible ways to compute the embeddings of Q in G.

6. A MAP-REDUCE ALGORITHM

We start this section with a brief presentation of the Map-Reduce framework. Then, we give a detailed description of our algorithm for quering linked-data using Map-Reduce.

6.1 The MapReduce framework

Map-Reduce is a programming framework for processing large datasets in a distributed manner, using a cluster of commodity machines. The storage layer for the Map-Reduce framework is a Distributes File System (DFS), such as the Hadoop Distributed File System (HDFS). The DFSs differ from conventional file systems in three main aspects. First, the data files are distributed across the nodes of the cluster. Second, their block/chuck size (typically 16-128MB in most of DFSs) is much larger than those in conventional file systems. Third, replication of chunks in relatively independent locations ensures availability.

The framework is based on the definition of two functions, the *Map* and the *Reduce* function. In particular, the user defines the two functions, which run in each cluster node, in isolation. The map function is applied to one or more files, in DFS, and results `[key,value]` pairs. This process is called *Map process/task*. The nodes that run the Map processes are called *Mappers*, and may run multiple tasks over different input files. The *master controller* is responsible to route the pairs to the *Reducers* (i.e., the nodes that apply the reduce function to the pairs) such that all pairs with the same key

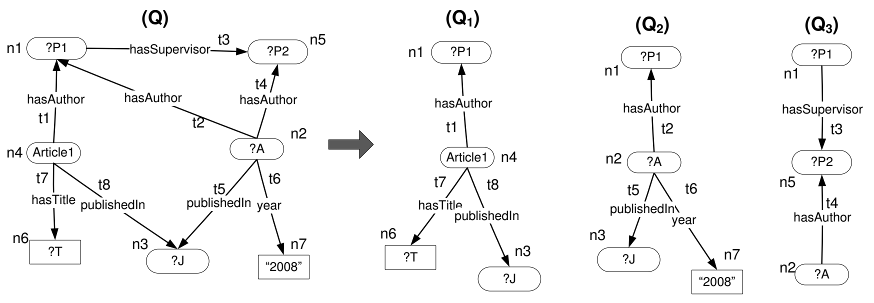

Figure 3: Query decomposition.

initialize a single reduce process, called *reduce task*. The reduce tasks apply the reduce function to the input pairs and also results `[key,value]` pairs, which are eventually stored in the DFS. This procedure describes one *MapReduce phase*. Furthermore, the output of the reducer can be set as the input of a map function, which gives the user the flexibility to create workflows consisting of Map-Reduce phases.

6.2 The preprocessing phase

In this phase query Q is decomposed into a set of star subqueries $Q_1, \ldots, Q_n$. For each subquery Q_i, a *query prototype* of the form $(bnFlags, nbnFlags, tFlags)$ is constructed, where $bnFlags$, $nbnFlags$ and $tFlags$ have one place for each branching node, non-branching node, and triple in Q, respectively. If an element (node or triple) occurs in Q_i, then the corresponding place in the query prototype contains a "+", otherwise it contains a "-". Moreover, an auxiliary list $NBL = [(b_i, Q_j) \mid b_i \in B(Q) \text{ and } b_i \notin nd(Q_j)]$ is constructed.

Preprocessing phase emits the above to the mappers of Phase 1 with key the pair $(subqueryID, SegmentID)$. NBL is also emitted to all reducers of Phase 1.

For the presentation of the algorithm we assume an enumeration $n_1, n_2, \ldots, n_{|nd(Q)|}$ of the nodes of a query Q, such that $n_1, n_2, \ldots, n_{|B(Q)|}$ are the branching nodes of Q and $n_{|B(Q)|+1}, \ldots, n_{|nd(Q)|}$ are the non-branching nodes of Q. We will denote by I the function that gives the index of a node in $nd(Q)$ with respect to the above enumeration (that is, for every $x \in nd(Q)$ it holds $x = n_{I(x)}$). We also denote by I_{nb} the function from $nd(Q) - B(Q)$ to $\{1, \ldots, |nd(Q) - B(Q)|\}$, with $I_{nb}(x) = I(x) - |B(Q)|$. Similarly, we assume a an enumeration $t_1, t_2, \ldots, t_{|Q|}$ of the triples in Q.

An embedding e of a (sub)query is represented as a pair of tuples (bn, nbn). If x is a branching (resp. non-branching) node, then $bn[I(x)]$ (resp. $nbn[I_{nb}(x)]$) contains the value $e(x)$. If e is an embedding of a subquery Q_i and x is a node that does not occur in Q_i, then an asterisk ('*') is stored in the place of $e(x)$.

Example 5. Consider the query graph appearing in Fig. 3 decomposed into three star subqueries. The branching nodes are n_1, n_2 and n_3, while the non-branching nodes are n_4, n_5, n_6 and n_7. The query prototypes are the following:
Q_1: $(\langle +,-,+ \rangle, \langle +,-,+,- \rangle, \langle +,-,-,-,-,-,+,+ \rangle)$
Q_2: $(\langle +,+,+ \rangle, \langle -,-,-,+ \rangle, \langle -,+,-,-,+,+,-,- \rangle)$
Q_3: $(\langle +,+,- \rangle, \langle -,+,-,- \rangle, \langle -,-,+,+,-,-,-,- \rangle)$
The list $NBL = [(n_2, Q_1), (n_3, Q_3)]$ is also constructed. □

6.3 Phase 1 of the algorithm

The first phase of the algorithm computes the embeddings of the star subqueries $Q_1, \ldots, Q_n$ in G.

6.3.1 *Mapper of Phase 1*

Each mapper gets as input a graph segment G_j, a star subquery Q_i and the NBL list. We denote by c_i the central node of Q_i (recall that this node appears in every triple of Q_i). The operation of the mapper is divided into two parts. Observe that if for some embedding e of Q_i in G the value of c_i is a non-border node of G_j (i.e., is $e(c_i) \in (nd(G_i) - B(G_i)))$, then it holds $e(v) \in G_j$ for every node $v \in nd(Q_i)$. This means that e is an embedding of Q_i into G_j and it can be computed locally. This computation is performed by the second part of the mapper.

The first part of the mapper handles the information that is relevant to the remaining embeddings: it computes all the embeddings of each triple of Q_i in G_j that map the central node c_i to a *border node*, and emits the results to appropriate reducers. More specifically, let $t = (s, p, o)$ be a triple that belongs to subquery Q_i and let e be an embedding of t into G_j such that $e(c_i)$ is a border node of G_j. If the central node of Q_i is s then the mapper emits a pair $(key, value)$, where $key = (Q_i, e(s))$ and $value = (o, e(o))$. Otherwise (i.e., the central node of Q_i is o) then $key = (Q_i, e(o))$ and $value = (s, e(s))$.

Notice that embeddings of triples in Q_i that map c_i to different nodes of the data graph are incompatible and cannot be joined to obtain an embedding of Q_i into G. Since the value of c_i is included in the key, incompatible embeddings of triples are emitted to different reducers, while compatible embeddings are emitted to the same reducer.

The second part of the mapper, computes all the embeddings of Q_i in G_j, which map c_i to a non-border node of G_j. This can be achieved either by adding an appropriate conjunct to Q_i, or by computing all the embeddings of Q_i in G_j and then removing those that assign border nodes to c_i. The embedings computed in the second part of the mapper are directly emitted to the mappers of Phase 2 (rather than to the reducers of Phase 1). Similarly, the values of branching nodes are emitted to the mappers of Phase 2.

```
mapper1(($Q_i$, $G_j$), (GjData, B(GjData), subqueryInfo, NBL))
//($Q_i$,$G_j$): $Q_i$/$G_j$ is the ID of a subquery/data segment
// GjData: the content of the data graph segment $G_j$
// B(GjData): the set of border nodes of $G_j$
// SubqueryInfo: prototypes/branching & non-branching nodes
    /triples of Q
// NBL: the list of missing branching nodes
begin
    - $c_i$ := the central node of $Q_i$
    % Part 1
    - for each triple $t = (c_i, p, o)$ in $Q_i$ do
        begin
            - compute $E = \{e \mid e$ is an embedding of $\{t\}$ in GjData
            and $e(c_i) \in B(GjData)$ };
            - **for each** embedding $e$ in $E$ **do**
                emit([$(Q_i, e(c_i))$,$(o, e(o))$])
        **end**
    - **for each** triple $t = (s, p, c_i)$ in $Q_i$ **do**
        **begin**
            - compute $E = \{e \mid e$ is an embedding of $\{t\}$ in GjData
            and $e(c_i) \in B(GjData)$ };
            - **for each** embedding $e$ in $E$ **do**
                emit([$(Q_i, e(c_i))$,$(s, e(s))$])
        **end**
    % Part 2
    - compute $E = \{e \mid e$ is a embedding of $Q_i$ in GjData
        and $e(c_i) \notin B(GjData)$ }
    - **for each** embedding $e = (bn, nbn)$ in $E$ **do**
        **begin**
            - emitToSecondPhase([$Q_i$, (bn, nbn)]);
            - **for** k = 1 to |bn| **do**
                - **if** (bn[k] != '*') **then**
                    - **for each** $(n_k, Q_j)$ in $NBL$ **do**
                        - emitToSecondPhase([$Q_j, (n_k, bn[k])$]);
        **end**
**end.**
```

Example 6. (Continued from Example 5). In this example we see the application of the *mapper1* on the pairs of subqueries and graph segments:

Applying mapper1 on (Q_1, G_1) results in emission (see Part 1) of the following $key, value$ pairs to the *reducer1*:
$(Q_1, Article1), (n_1, Person4)$ (embedding of $t1$)
$(Q_1, Article1), (n_6, \text{"}Title1\text{"})$ (embedding of $t7$)
No key value pairs are emitted to Phase 2.
Applying mapper1 on (Q_2, G_1) results in emission (see Part 1) of the following $key, value$ pairs to the *reducer1*:
$(Q_2, Article1), (n_1, Person4)$ (embedding of $t2$)
Besides, the following $key, value$ pairs are emitted directly (see Part 2) to the *mapper2* (mapper of Phase 2):
$Q_2, (\langle Person4, Article3, Journal2\rangle, \langle *, *, *, \text{"2008"}\rangle)$
$Q_1, (n_2, Article3)$
$Q_3, (n_3, Journal2)$
Applying mapper1 on (Q_3, G_1) results in emission (see Part 1) of the following $key, value$ pairs to the *reducer1*:
$(Q_3, Person4), (n_2, Article1)$ (embedding of $t4$)
$(Q_3, Person4), (n_2, Article3)$ (embedding of $t4$)
No key value pairs are emitted to Phase 2.
Applying mapper1 on (Q_1, G_2) results in emission (see Part 1) of the following $key, value$ pairs to the *reducer1*:
$(Q_1, Article1), (n_1, Person1)$ (embedding of $t1$)
$(Q_1, Article1), (n_1, Person2)$ (embedding of $t1$)
No key value pairs are emitted to Phase 2.
Applying mapper1 on (Q_2, G_2) results in emission (see Part 1) of the following $key, value$ pairs to the *reducer1*:
$(Q_2, Article1), (n_1, Person1)$ (embedding of $t2$)
$(Q_2, Article1), (n_1, Person2)$ (embedding of $t2$)
$(Q_2, Article2), (n_1, Person2)$ (embedding of $t2$)
$(Q_2, Article2), (n_1, Person3)$ (embedding of $t2$)
No key value pairs are emitted to Phase 2.
Applying mapper1 on (Q_3, G_2) results in no emission of any $key, value$ pair to the *reducer1.* However, the following $key, value$ pairs are emitted (see Part 2) to the *mapper2*:
$Q_3, (\langle Person4, Article1, *\rangle, \langle *, Person1, *, *\rangle)$
$Q_3, (\langle Person2, Article2, *\rangle, \langle *, Person3, *, *\rangle)$
$Q_1, (n_2, Article1)$
$Q_1, (n_2, Article2)$
Applying mapper1 on (Q_1, G_3) results in emission of the following $key, value$ pair to the *reducer1*:
$(Q_1, Article1), (n_3, Journal1)$ $(t8)$
No key value pairs are emitted to Phase 2.
Applying mapper1 on (Q_2, G_3) results in emission of the following $key, value$ pair to the *reducer1*:
$(Q_2, Article1), (n_3, Journal1)$ $(t5)$
$(Q_2, Article2), (n_3, Journal1)$ $(t5)$
$(Q_2, Article2), (n_7, \text{"2008"})$ $(t6)$
No key value pairs are emitted to Phase 2.
Applying mapper1 on (Q_3, G_3) produces no results. □

6.3.2 Reducer of Phase 1

For each key (Q_i, v) the corresponding reducer computes all the embeddings of Q_i that map central node c_i of Q_i to v. The input to this reducer is a list of pairs of the form (n_k, u), where n_k is a node of Q_i different from c_i and u is a possible values for n_k in an embedding of Q_i in G.

Suppose that $n_{k_1}, n_{k_2}, \ldots, n_{k_m}$ are the non-central nodes in Q_i. The reducer constructs for every $j = 1, \ldots, m$ a set L_{k_j} of all possible values for node n_{k_j}. Then for each element $(x_1, x_2, \ldots, x_m)$ of the cartesian product $L_{k_1} \times L_{k_2} \times \cdots \times L_{k_m}$ it constructs an embedding $e = (bn, nbn)$ of Q_i in G, such that $e(c_i) = v$ and $e(n_{k_j}) = x_j$ and emits $(Q_i, (bn, nbn))$ (see Subsection 6.2 for the representation of an embedding).

Moreover, if every list $L_{k_1}, L_{k_2}, \ldots, L_{k_m}$ is non-empty (that is, at least one embedding of Q_i has been found), *reducer1* emits the values of missing branching nodes.

```
**reducer1**($(Q_i, v)$, values)
// $Q_i$: a subquery ID
// $v$: the value of the central node of $Q_i$
// values: contains a list of pairs $(x, u)$ and the $NBL$ list.
**begin**
    - $allNonEmpty := true$
    - **for each** non-central node $x$ in $Q_i$ **do**
        **begin**
            - $L[x] := \{u \mid (x, u) \in values\}$
            - **if** $L[x]$ is empty **then** $allNonEmpty := false$
        **end**
    - **if** $allNonEmpty$ **then**
        **begin**
            - create an embedding with undefined values
                $(bn, nbn) := (\langle *, \ldots, * \rangle, \langle *, \ldots, * \rangle)$
            - $c_i$ := the central node of $Q_i$
            - $L[c_i] := \{v\}$
            - **if** $c_i$ is a branching node
                **then** $bn[I(c_i)] := v$
                **else** $nbn[I_{nb}(c_i)] := v$
            - $E := \{(bn, nbn)\}$
            - **for each** non-central node $x$ in $Q_i$ **do**
                **begin**
                    - $E' := \emptyset$
                    - **for each** $e$ in $E$ **do**
                        - **for each** $u$ in $L[x]$ **do**
                            **begin**
                                - create a copy $e' = (bn', nbn')$ of $e$
                                - **if** $x$ is a branching node
                                    **then** $bn'[I(x)] := u$
                                    **else** $nbn'[I_{nb}(x)] := u$
                                - insert $(bn', nbn')$ in $E'$
                            **end**
                    - $E := E'$
                **end**
            - **for each** embedding $e = (bn, nbn)$ in $E$ **do**
                - emit([$Q_i$, (bn, nbn)])
            - **for each** $(x, Q_j)$ in $NBL$ **do**
                - **if** $x$ is a node in $Q_i$ **then**
                    - **for each** $u$ in $L[x]$ **do** emit([$Q_j, (x, u)$])
        **end**
**end.**
```

Example 7. (Continued from Example 6).
The *reducer with key* $(Q_1, Article1)$ receives the list of values

$[(n_1, Person4), (n_6, \text{“}Title1\text{”}), (n_1, Person1), (n_1, Person2), (n_3, Journal1)]$. It constructs the lists: $L_{n_1} = \{Person1, Person2, Person4\}$, $L_{n_3} = \{Journal1\}$, $L_{n_6} = \{\text{“}Title1\text{”}\}$.
It emits the following $key, value$ pairs:
$Q_1, (\langle Person1, *, Journal1\rangle, \langle Article1, *, Title1, *\rangle)$
$Q_1, (\langle Person2, *, Journal1\rangle, \langle Article1, *, Title1, *\rangle)$
$Q_1, (\langle Person4, *, Journal1\rangle, \langle Article1, *, Title1, *\rangle)$
$Q_3, (n_3, Journal1)$
The *reducer with key* $(Q_2, Article1)$ receives the list of values $[(n_1, Person4), (n_1, Person1), (n_1, Person2), (n_3, Journal1)]$.
It constructs the lists: $L_{n_1} = \{Person1, Person2, Person4\}$, $L_{n_3} = \{Journal1\}$, $L_{n_7} = \{\}$. Nothing is emitted.
The *reducer with key* $(Q_2, Article2)$ receives the list of values $[(n_1, Person2), (n_1, Person3), (n_3, Journal1), (n_7, \text{“}2008\text{”})]$.
It constructs the lists: $L_{n_1} = \{Person2, Person3\}$, $L_{n_3} = \{Journal1\}$, $L_{n_7} = \{\text{“}2008\text{”}\}$.
It emits the following $key, value$ pairs:
$Q_2, (\langle Person2, Article2, Journal1\rangle, \langle *, *, *, \text{“}2008\text{”}\rangle)$
$Q_2, (\langle Person3, Article2, Journal1\rangle, \langle *, *, *, \text{“}2008\text{”}\rangle)$
$Q_1, (n_2, Article2)$
$Q_3, (n_3, Journal1)$
The *reducer with key* $(Q_3, Person4)$ receives the list of values $[(n_2, Article1), (n_2, Article3)]$. It constructs the lists: $L_{n_1} = \{\}$, $L_{n_2} = \{Article1, Article3\}$. Nothing is emitted. □

6.4 Phase 2 of the algorithm

Phase 2 of the algorithm is similar to the Phase 2 of the algorithm proposed in [10].

6.4.1 Mapper of Phase 2

Each mapper gets as input all the embeddings of a specific subquery Q_i; moreover for each branching node that does not occur in Q_i it gets as input the values assigned to this node by the embeddings of the other queries. It fills in their missing branching node values using the corresponding values in the input, and emits the resulted embeddings to the reducers of Phase 2. The key is the tuple of the branching node values, which implies that two embeddings are emitted to the same reducer if and only if they are compatible.

```
mapper2($Q_i$, values)
// $Q_i$: the ID of a subquery
// values: a set E of the parts (bn, nbn) of the embeddings
//      of $Q_i$ and a set V of pairs $(n_k, v)$,
//      where v is a candidate value for bn[k]
begin
    - for each embedding $e = (bn, nbn)$ in E do
        - for each instance $bn'$ of bn using the values in V do
            - emit($[bn', (Q_i, nbn)]$)
end.
```

Example 8. (Continued from Example 7). The mapper that works for the subquery Q_1, gets a list of values that contain the embeddings of Q_1 in G:
$Q_1, (\langle Person1, *, Journal1\rangle, \langle Article1, *, Title1, *\rangle)$
$Q_1, (\langle Person2, *, Journal1\rangle, \langle Article1, *, Title1, *\rangle)$
$Q_1, (\langle Person4, *, Journal1\rangle, \langle Article1, *, Title1, *\rangle)$
and the values of missing branching nodes:
$(n_2, Article1)$, $(n_2, Article2)$, $(n_2, Article3)$
It emits the following $key, value$ pairs to the reducers of Phase 2:
$\langle Person1, Article1, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person1, Article2, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person1, Article3, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person2, Article1, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person2, Article2, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person2, Article3, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person4, Article1, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person4, Article2, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
$\langle Person4, Article3, Journal1\rangle$, $(Q_1, \langle Article1, *, Title1, *\rangle)$
The mapper that works for the subquery Q_2, receives a list of values containing the following embeddings
$Q_2, (\langle Person4, Article3, Journal2\rangle, \langle *, *, *, \text{“}2008\text{”}\rangle)$
$Q_2, (\langle Person2, Article2, Journal1\rangle, \langle *, *, *, \text{“}2008\text{”}\rangle)$
$Q_2, (\langle Person3, Article2, Journal1\rangle, \langle *, *, *, \text{“}2008\text{”}\rangle)$
Notice that Q_2 has no missing branching nodes. The mapper emits the following $key, value$ pairs to the reducers:
$\langle Person4, Article3, Journal2\rangle$, $(Q_2, \langle *, *, *, \text{“}2008\text{”}\rangle)$
$\langle Person2, Article2, Journal1\rangle$, $(Q_2, \langle *, *, *, \text{“}2008\text{”}\rangle)$
$\langle Person3, Article2, Journal1\rangle$, $(Q_2, \langle *, *, *, \text{“}2008\text{”}\rangle)$
The mapper that works for the subquery Q_3, get a list of values that contain the embeddings of Q_3 in G:
$Q_3, (\langle Person4, Article1, *\rangle, \langle *, Person1, *, *\rangle)$
$Q_3, (\langle Person2, Article2, *\rangle, \langle *, Person3, *, *\rangle)$
and the values of missing branching nodes:
$(n_3, Journal1)$, $(n_3, Journal2)$
It emits the following $key, value$ pairs to the reducers:
$\langle Person4, Article1, Journal1\rangle$, $(Q_3, \langle *, Person1, *, *\rangle)$
$\langle Person4, Article1, Journal2\rangle$, $(Q_3, \langle *, Person1, *, *\rangle)$
$\langle Person2, Article2, Journal1\rangle$, $(Q_3, \langle *, Person3, *, *\rangle)$
$\langle Person2, Article2, Journal2\rangle$, $(Q_3, \langle *, Person3, *, *\rangle)$ □

6.4.2 Reducer of Phase 2

Each reducer gets as input embeddings for each sub-query that are compatible (each one of them assigns the values in the key of the reducer to the branching nodes of the query). The embeddings (one for each subquery in $(Q_1, \ldots, Q_n)$) are joined to construct the final answers of Q:

```
reducer2(key, values)
// key: a tuple of branching node values
// values: pairs of the form
    ($Q_i$, partial embedding for non-branching nodes)
begin
    - for each join obtained by using one
            embedding for each subquery do
        - Emit the result produced by this join
end.
```

Example 9. (Continued from Example 8). The reducer with key $\langle Person2, Article2, Journal1\rangle$ receives the list of values $[(Q_1, \langle Article1, *, Title1, *\rangle)$, $(Q_2, \langle *, *, *, \text{”}2008\text{”}\rangle)$, $(Q_3, \langle *, Person3, *, *\rangle)]$ and constructs the unique embedding of Q in G:
$(\langle Person2, Article2, Journal1\rangle,$
$\langle Article1, Person3, Title1, \text{“}2008\text{”}\rangle)$
The remaining 11 reducers do not return any answer (they don't receive values for at least one subquery). □

6.5 Implementation of the algorithm and experimental results

In this section, we present a set of preliminary experiments performed on a Hadoop cluster of 14 nodes of the following characteristics: Intel Pentium(R) Dual-Core CPU E5700 3.00GHz with 4GB RAM. We used five different datasets of sizes 113.6MB, 231.6MB, 491.5MB, 1.2GB, and 2.5GB obtained and adapted from the Lehigh University Benchmark (LUBM)[2]. These data sets are partitioned into

[2]http://swat.cse.lehigh.edu/projects/lubm/

four, nine or fourteen data segments in correspondence with the number of the nodes of the cluster used in the experiment. Data segments are stored in different nodes of the cluster, in relational MySQL databases, whose schema consists of two tables one containing the triples of the segment and the other containing the border nodes.

The results of the first experiment are depicted in Table 1, where we can see that the algorithm is scalable in terms of the dataset size. In addition, the degree of the star (i.e. the number of the triples in the star) also affects the execution time: the larger the degree the smaller the execution time. The results of the second experiment (depicted in Table 2) show that our algorithm scales well by increasing the number of nodes in the cluster. In this experiment, the same queries have been computed (using the same star-decompositions) in 4, 9 and 14 computer nodes, for a fixed data set of size 231.6MB. Finally, the results of third experiment, depicted in Table 3, show that the algorithm of this paper performs better in most cases (depending on the form of the given query) than the algorithm proposed in [10].

Dataset size	**stars of degree 6**	**Stars of degree 3**	**Stars of degree 2**	**Num of Answers**
113.6MB	190	219	254	93
231.6MB	212	258	280	189
491.5MB	310	369	459	402
1.2GB	650	689	727	999
2.5GB	1149	1200	1261	2007

Table 1: Execution times (in seconds) for three different star-decompositions of a query in three different datasets, using a cluster of 14 nodes.

# nodes	**stars of degree 6**	**stars of degree 3**	**stars of degree 2**
4	278	322	343
9	271	309	316
14	212	258	280

Table 2: Execution times (in seconds) for three different star-decompositions of a query in a dataset of size 231.6MB, using clusters of 4, 9, and 14 nodes.

Alg.	**query1**	**query2**	**query3**	**query4**	**query5**
[10]	8,24	9,44	9,03	10,37	5,06
this	4,27	5,45	5,25	11,53	5,59

Table 3: Execution times (in minutes) of two algorithms for five different queries, in a dataset of size 491,5MB, using a cluster of 14 nodes.

7. CONCLUSION

In this paper, we present a two-phase MapReduce algorithm, for querying large amount of linked data, that extends our approach in [10]. The input query is decomposed into star subqueries and the answers to these subqueries are computed and joined to obtain the answers to the given query. Experimental evaluation shows that the algorithm is scalable in terms of the size of the data graph as well as the number of nodes in the cluster. In the near future we plan to compare the performance of the algorithm for different methods for decomposition of the input query into stars.

8. REFERENCES

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, pp. 62–73, 2013.

[3] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[4] C. Basca and A. Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. In *CEUR Workshop*, 2010.

[5] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. A budget-based algorithm for efficient subgraph matching on huge networks. In *ICDE Workshops*, pp. 94–99, 2011.

[6] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11(4):29–41, 2009.

[7] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. T. Groth, A. Haque, A. Harth, F. L. Keppmann, D. P. Miranker, J. Sequeda, and M. Wylot. Nosql databases for rdf: An empirical evaluation. In *International Semantic Web Conference (2)*, pp. 310–325, 2013.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] J.-H. Du, H. Wang, Y. Ni, and Y. Yu. HadoopRDF: A scalable semantic data analytical engine. In *ICIC (2)*, pp. 633–641, 2012.

[10] M. Gergatsoulis, C. Nomikos, E. Kalogeros, and M. Damigos. An algorithm for querying linked data using map-reduce. In *Globe*, pp. 51–62, 2013.

[11] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[12] M. F. Husain, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *CLOUD*, pp. 1–10. IEEE, 2010.

[13] P. Mika and G. Tummarello. Web semantics in the clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[14] J. Myung, J. Yeon, and S. goo Lee. SPARQL basic graph pattern processing with iterative mapreduce. In *MDAC*, ACM, 2010.

[15] Z. Nie, F. Du, Y. Chen, X. Du, and L. Xu. Efficient SPARQL query processing in mapreduce through data partitioning and indexing. In *APWeb*, pp. 628–635, 2012.

[16] N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *WWW (Companion Volume)*, pp. 397–400. ACM, 2012.

[17] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In *ESWC*, pp. 524–538, 2008.

[18] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. Pigsparql: mapping SPARQL to pig latin. In *SWIM*, 2011.

[19] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pp. 607–614, 2011.