Efficient query evaluation techniques over large amount of distributed linked data

Eleftherios Kalogeros\textsuperscript{a}, Manolis Gergatsoulis\textsuperscript{a}, Matthew Damigos\textsuperscript{a}, Christos Nomikos\textsuperscript{b}

\textsuperscript{a} Database \\& Information Systems Group (DBIS), Laboratory on Digital Libraries and Electronic Publishing, Department of Archives, Library Science and Museology, Ionian University Ioannou Theotoki 72, 49100 Corfu, Greece
\{kalogero, manolis\}@ionio.gr mgdamig@gmail.com
\textsuperscript{b} Department of Computer Science and Engineering, University of Ioannina, Greece P.O Box 1186, 45110 Ioannina, Greece cnomikos@cs.uoi.gr

Abstract
As RDF becomes more widely established and the amount of linked data is rapidly increasing, the efficient querying of large amount of data becomes a significant challenge. In this paper, we propose a family of algorithms for querying large amount of linked data in a distributed manner. These query evaluation algorithms are independent of the way the data is stored, as well as of the particular implementation of the query evaluation. We then use the MapReduce paradigm to present a distributed implementation of these algorithms and experimentally evaluate them, although the algorithms could be straightforwardly translated into other distributed processing frameworks. We also investigate and propose multiple query decomposition approaches of Basic Graph Patterns (subclass of SPARQL queries) that are used to improve the overall performance of the distributed query answering. A deep analysis of the effectiveness of these decomposition algorithms is also provided.

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

1. Introduction
Linked data has become a widely-established approach for publishing and sharing semantically-meaningful information through distributed and interrelated data. RDF is the standard model Linked data is built upon. As RDF data is rapidly increasing, the efficient querying of large amount of Linked data becomes a significant challenge in many business and research areas, such as bioinformatics and cheminformatics, and digital libraries [11, 52].

Both centralized (e.g., [65, 66]) and distributed (e.g., [35, 58]) processing of RDF data has been extensively investigated in the past, where SPARQL [64, 55]
is mainly used as query language. To process large amount of RDF data in a
distributed manner, parallel processing frameworks are considered [11, 16, 19].
Apache Hadoop [5] (the open source alternative of Google’s MapReduce [25]),
Spark [72, 8] and Flink [4] are three widely-used programming frameworks for
distributed processing. Although Apache Spark and Flink typically improve and
outperform Hadoop/MapReduce, mainly due to in-memory processing, from al-
gorithmic perspective, they all handle distributed processing in a similar man-
ner; i.e., define workflows of tasks running in parallel and determine the way
the data is reshuffled in order to be properly and efficiently joined.

In addition to the query evaluation approaches, a variety of effective storage
schemes has been used to improve query answering over RDF data, such as
the use of relational databases (e.g., [22, 66]) and NoSQL databases (e.g., [71]).
In distributed environments, the proper partitioning of the RDF data into a
distributed repository (either file system or distributed NoSQL database) can
significantly improve the performance of query answering [21, 27, 38]. Following
this approach, most of the distributed based methods and systems utilize
efficient partitioning of the data across a cluster of machines in order to ensure
efficient query processing through minimizing the communication cost and im-
proving parallel execution [21]. To take advantage of the selected partitioning
during query answering, certain approaches for decomposing the given query,
and creating a proper query plan (consisting of multiple steps of distributed
processing) are proposed [26, 29].

In this work, we present three distributed evaluation algorithms for querying
large amount of RDF data. The main idea behind these algorithms is described
as follows: a) The data graph is decomposed into a set of (possibly overlap-
ning) data graph segments stored in different nodes of a cluster of commodity
machines. b) The query graph Q is also decomposed into a set of (possibly
overlapping) subqueries. c) Subqueries are applied to each data graph segment,
in isolation, and intermediate results are computed. d) The intermediate results
are appropriately combined to obtain the answers of the query Q. Note that
the algorithms are independent of the way the data is stored as well as of the
particular implementation of the query evaluation. We then use the MapReduce
paradigm to present a distributed implementation of these algorithms and ex-
perimentally evaluate them, although the algorithms could be straightforwardly
translated into Spark jobs and/or Flink dataflows.

This paper consolidates our previous work presented in [37, 28, 50] into
a single unified framework for distributed evaluation of Basic Graph Pattern
queries (subclass of SPARQL queries), and extend this framework by proposing
multiple query decomposition algorithms that could be used by a wide-variety
of query evaluation approaches. In particular, we investigate decomposition
approaches a) which are based on producing subqueries of special forms (with
or without replication of query triples), and b) that take into account certain
replication of the distributed data.

The paper is organized as follows. In Section 2, related work is presented and
discussed. In Section 3, the framework of our work is defined. More specifically,
after presenting some preliminary definitions in Subsection 3.1, we introduce
the concept of (data and query) graph decomposition in Subsection 3.2. Then, we define the concept of partial embeddings in Subsection 3.3 and distinguish special forms of queries in Subsection 3.4.

In Section 4, we presented three query evaluation approaches. More specifically, in Subsection 4.1, we present a query evaluation approach which is based on the concept of partial embeddings. In Subsection 4.2, we present an approach which is based on the decomposition of queries into subqueries of a specific form called generalized star queries. Finally, in Subsection 4.3, we present an approach which is based on the idea that replication in data decomposition can be taken into account to efficiently answer queries. In Subsection 4.4, we present a set of query decomposition algorithms.

In Section 5, we present a set of query evaluation algorithms, which implement the approaches presented in Section 4. Experimental evaluation results of the algorithms are presented in Section 6. Finally, Section 7 concludes the paper.

2. Related work

The problem of efficiently querying linked data has been widely investigated, for both centralized (single-machine) \[65, 66\] (e.g., systems such as RDF-3X \[47, 49\] and Hexastore \[70\]) and distributed environments (e.g., \[35, 58\]). Processing large amount of linked data into a single machine has significant limitations, since it lacks scalability \[21\]. To handle this problem, a variety of distributed methods for storing and processing linked data has been proposed \[71\]. Most of the approaches proposed in the literature to handle scalability of answering SPARQL queries over big linked datasets \[11\] focus on two aspects, distributed storage of linked data and distributed processing of SPARQL queries. Typically, the proper partitioning of the data into a distributed repository (either file system or distributed NoSQL database) can significantly improve the performance of query answering \[21, 27, 38\] (e.g., Random Partitioning \[28, 50\], Hash Partitioning \[58, 26, 23\], Graph Partitioning \[34, 39\], and Semantic Partitioning \[43\]).

To process and query large amount of linked data, distributed processing frameworks, such as MapReduce \[5\] or Apache Spark \[8\], are used. Apart from these approaches, there is a noteworthy amount of related work focusing on utilizing distributed NoSQL databases \[57, 18, 60, 42\] to store the linked data graph and answer the given queries. In these cases, the query evaluation is achieved either through translating the given SPARQL query into the query language supported by the NoSQL database \[44\], or by using a distributed processing framework to implement the overall query execution plan \[29\] (in such a case, the NoSQL database mainly used as a storage layer ensuring proper data partitioning).

In this context, Afrati et al. \[15\] proposed an approach for optimizing joins in MapReduce by choosing the appropriate map-key and shares. This approach is extended in \[12\] to data graphs, where the cost of evaluating queries on data graphs using one round of Map-Reduce is investigated, and an approach
of translating the query patterns into conjunctive queries is proposed. The communication cost is minimized using the techniques of the approach proposed in [13]. Such an approach could be used for answering conjunctive SPARQL queries [48, 56, 68].

A method of answering SPARQL Basic Graph Pattern using traditional multi-way join into MapReduce, instead of multiple individual joins, is also presented in [45], where certain joining keys are selected to avoid unnecessary iterations. This approach can be used for every type of partitioning of the RDF data.

SHARD [58] is built on top of Hadoop, and uses the Hadoop distributed files system (HDFS) to store data in native text files. It uses subject hash partitioning to decompose the RDF data graph; all the triples with the same subject are stored in the same line of the text file. For the execution, one MapReduce job is created for every query triple, while an additional job is used, at the end, to remove duplicated results and apply the required projection. Hence, assuming an $n$-triple query pattern, $n + 1$ jobs are required, and all the data graph is scanned $n$ times.

HadoopRDF [35] uses predicate hash partition method to distribute the data graph; similar to the vertical partitioning approach applied by SW-Store [10]. In general, the number of the data fragments is equal to the number of the distinct predicates. The query evaluation is performed through a sequence of MapReduce jobs and is optimized using a heuristic and a greedy approach.

CliqueSquare [29] presents a method that generates highly parallelizable query plans for BGP queries, which rely on n-ary equality joins with minimum amount of MapReduce stages. CliqueSquare uses a data partitioning scheme that permits first-level joins can be evaluated locally at each node. The triples that share the same value in subject, predicate or object are placed on the same node. This partition ensures that queries sharing the same variable (like star queries) can be evaluated locally.

H2RDF [54] uses the Apache HBase [6] to store data triples. Three RDF indices on subject, predicate and object (spo, pos and osp combinations) are materialized and stored to HBase in the form of key-value pairs. Different strategies are used to execute joins and answer the given query. H2RDF+ [53] extends H2RDF by considering three more indices (ops, osp and sop). Furthermore, MapReduce Merge Join algorithm is used to join query triples that share the same variable and the MapReduce Sort-Merge Join algorithm is used for joining the intermediate results.

PigSPARQL [61] is yet another approach which uses Hadoop-based implementation of vertical partitioning of the data stored into HDFS. It implements a translation from SPARQL to Pig Latin [51]. In the system RAPID+ [41], an alternative query algebra, called the Nested Triple Group Algebra, is used as an extension of Apache Pig, to improve the performance of SPARQL query processing over MapReduce.

The authors in [34] proposed a graph partitioning schema, which resembles the s-decomposition partitioning defined in this work. In particular, the data is partitioned in such a way that the vertices that are relatively close to each
other are included in the same segment. In this context, the following main methodologies are investigated and proposed: the undirected and the directed n-hop guarantee. The former focuses on initially partitioning the vertices and then assigning the triple-paths of length $n$ that start from a vertex that is already included in the segment. The latter is similar to the directed one but considers any undirected path of length $n$. In both cases, a graph practitioner tool which is based on METIS is used for partitioning the vertices of the RDF graph into disjoint partitions so that the minimum number of edges is cut. The queries are also decomposed in such a way that the subqueries generated can be computed locally, in each cluster node. MapReduce is used for the joins of the intermediate results of subqueries. Although s-decomposition partitioning approach is similar to 1-hop undirected guarantee (or hash partitioning), the n-hop guarantee of the data graph may cause data explosion especially in coherent data graphs if $n > 2$. SHAPE [43] proposes a semantic hash partitioning which is based on the similarity of the URI hierarchy of the vertices. The vertices with same URI prefixes are placed in the same partition. After a simple hash partitioning is used, a replication of only a set of necessary triples is performed, using a k-hop semantic hash partitioning and context-aware filters. The system also uses a RDF-3X triple store in each data node. Query processing and the joins of the intermediate results is based in MapReduce.

The papers [37, 50, 28] focus on both decomposing queries and partition the RDF data, where the data is stored into MySQL and the framework used to evaluate the queries is MapReduce. SPARQL to SQL translations is used for query processing, and MapReduce is used to apply the joins.

D-SPARQ [44] uses the document database MongoDB [9] to store and index data using subject hash partition. A single MapReduce job is then used to import data into the document database and to collect statistical information for query optimization process based on join reordering. All triples sharing the same subject value are stored in the same document (JSON) file.

Another approach which is based on the MapReduce is Sempala [62], which applies SPARQL-to-SQL translation on top of Hadoop. It uses Impala [7] as a distributed SQL processing engine. Sempala uses a unified vertical partitioning (single property table) in order to boost the star-shaped queries.

In [69], the authors proposed a MapReduce algorithm, called StarMR, which is based on star decomposition for answering subgraph matching queries. The StarMR algorithm is improved with two optimization strategies. The first applies an RDF property filtering approach and the second one postpones any Cartesian product operation. RDF graph is stored in a distributed adjacency list.

[36] uses a partitioning method over the predicate value and the type of objects to store the RDF data. Query processing is performed using MapReduce and the algorithm proposed applies a number of the jobs that depends on the form of the given query.

Apache Spark and Flink have been used to improve the performance of SPARQL query evaluation over big RDF data [16]. SPARQLGX [31] uses a
vertical partition approach, where the triples are partitioned according to their predicate values. The query evaluation is performed by initially filtering the triples matching a query triple, in each segment, and then, by applying a sequence of join operations through a query plan which is generated according to predefined statistics. The authors in [19] propose an approach for translating SPARQL queries into Apache Flink [4] programs for querying RDF data, as well as investigate the semantic correspondence between Apache Flink’s subset transformations and the SPARQL Algebra operators.

S2RDF [63] also proposes a vertical-like partitioning, called Extended Vertical Partitioning (ExtVP), which is based on semi-joins reductions (i.e., a certain number of semi-joins are applied between the vertical partitioning tables and their results are materialized for improving the overall performance). To evaluate queries over ExtVP, an approach of applying a certain partitioning of the query triples (in order to achieve parallel/local computation) and utilizing Spark SQL is followed.

HAQWA [23] proposes a hash-based partitioning over the subject values of the RDF triples. This ensures local computation of subject-centric star queries (a subclass of generalized star queries). To extend the supported queries, the query is decomposed into subqueries and missing triples of each subquery are replicated. The overall computation process is managed through a Spark application.

The authors in [46] analyze the query evaluation plan of a BGP expression on Spark and proposes a joins plan for efficiently evaluating BGPs over a large RDF graph. Considering an initially hash-based partitioning of the data (e.g., the triples are partitioned by their subject), the authors propose a hybrid method to find a query plan. The approach uses a cost-driven combination of partitioned/cascade and broadcast joins over Spark.

Apart from the previous approaches, it’s worth mentioning the approaches S2X [59], Spar(k)ql [30] and [40], which focus on evaluating SPARQL queries using the Spark GraphX library. Sparklify [67] applies a SPARQL-to-SQL rewriter for translating SPARQL queries into Spark executable code.

In [33], a property table scheme is built on top of HBase storage system and a vertical partitioning scheme on top of Cassandra storage system. Query processing is based on SPARQL query translation to SparkSQL for both HBase and Cassandra storage schemas.

As mentioned previously, multiple approaches that use NoSQL platforms to store RDF data and answer SPARQL queries have been proposed in the literature [38]. Representative examples include the distributed systems Rya [57], AMADA [18], MAPSIN [60] and CumulusRDF [42] which use NoSQL [24] databases to store RDF data and provide efficient query processing using three different indices SPO, POS and OSP (S for subject, P for predicate and O for object values). More specifically, Rya uses Apache Accumulo [2], AMADA use Amazon DynamoDB [1], MAPSIN Apache HBase and CumulusRDF Apache Cassandra [3]. MAPSIN (Map-Side Index Nested Loop Join) joins are performed in the map phase, so shuffle and reduce phase are not required. The proposed algorithm is optimized for the efficient processing of multiway joins.
3. Definition of the framework

3.1. Preliminaries

Let $U_{so}$ and $U_p$ be two countably infinite disjoint sets of URI references, $L$ be a countably infinite set of (plain) literals\(^1\) and $V$ be a countably infinite set of variables. In the following, we define two types of graphs, data graphs and query graphs. The former describes the data model and the latter determines the form of the query expressions over the stored data.

**Definition 1.** A triple $(s, p, o) \in U_{so} \times U_p \times (U_{so} \cup L)$ is called a data triple. In a data triple $t = (s, p, o)$, $s$ is called the subject, $p$ the predicate and $o$ the object of $t$. A data graph $G$ 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.** A triple $(s, p, o) \in (U_{so} \cup V) \times U_p \times (U_{so} \cup L \cup V)$ is called a query triple. In a query triple $q = (s, p, o)$, $s$ is called the subject, $p$ the predicate and $o$ the object of $q$. A query graph (or simply a query) $Q$ is a nonempty set of query triples. The output pattern $O(Q)$ of a query graph $Q$ is the tuple $(X_1, \ldots, X_n)$, with $n \geq 0$, of all the variables appearing in $Q$. A query $Q$ is said to be a Boolean query if $n = 0$. A query graph $Q'$ is a subquery of a query graph $Q$ if $Q' \subseteq Q$.

**Definition 3.** Let $G$ be a data or query graph. A directed path (or simply path) in $G$ is a sequence of triples $(v_0, p_1, v_1), (v_1, p_2, v_2), \ldots, (v_{n-1}, p_n, v_n)$ in $G$, where $n \geq 1$. The length of the path is the number $n$ of triples in the path. A finite directed path always has a start node which corresponds to the subject of its first triple, and an end node which is the object of its last triple. A cycle is a path in which the start node and the end node are the same. A path with no repeated nodes (i.e. without cycles) is called a simple path.

The set of nodes of a data graph $G$ (resp. a query graph $Q$), denoted by $N(G)$ (resp. $N(Q)$), is the set of elements of $U_{so} \cup L$ (resp. $U_{so} \cup L \cup V$) that occur in the triples of $G$ (resp. $Q$). The set of edge labels of a data graph $G$ (resp. a query graph $Q$), denoted by $E(G)$ (resp. $E(Q)$), is the set of elements of $U_p$ that occur in the triples of $G$ (resp. $Q$). Finally, the set of variables in a query $Q$ is denoted by $V(Q)$.

Notice that the data graphs defined above correspond to ground RDF graphs defined in [32]. Notice also that query graphs correspond to Basic Graph Patterns (BGP) SPARQL queries. In this paper, we do not allow queries with variables in the place of predicates. However, the query evaluation algorithms proposed in this paper can be easily extended to allow such variables.

Data and query graphs are graphically represented as follows: A node (subject or object), which is a URI or a variable, is represented as a rounded rectangle, while an object which is a literal is represented by a rectangle. Each triple $(s, p, o)$ is represented by a labeled edge $s \xrightarrow{p} o$ connecting the nodes $s$ and $o$.

---

\(^1\)In this paper we do not consider typed literals
In this paper, we use strings with initial lowercase letters to represent elements in \( U_p \) (i.e., URIs corresponding to predicates), while strings with initial uppercase letters denote elements in \( U_o \) (i.e., URIs corresponding to objects and subjects). Literals are represented as strings enclosed in double quotes. Finally, we assume that variables are represented by strings whose first symbol is the question mark symbol (?).

**Example 1.** Fig. 1(a) depicts a data graph showing information about three journal papers, their authors and the relationships between the authors. Fig. 1(b) shows a query graph.

![Figure 1: (a) A data graph, and (b) a query graph](image)

**Definition 4.** A (total) embedding of a query graph \( Q \) in a data graph \( G \) is a total mapping \( e : \mathcal{N}(Q) \rightarrow \mathcal{N}(G) \) with the following properties:

1. For each node \( v \in \mathcal{N}(Q) \), if \( v \) is not a variable then \( e(v) = v \).
2. For each triple \((v_1, p, v_2) \in Q\), the triple \( (e(v_1), p, e(v_2)) \) 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. 2 depicts an embedding of the query graph \( Q \) in data graph \( G \), where \( Q \) and \( G \) are the graphs appearing in Fig. 1. The answer obtained by this embedding is \(?A, ?W, ?T = (Article2, Person3, “Title2”)\). Notice that a second embedding exists giving the answer \(?A, ?W, ?T = (Article2, Person2, “Title2”)\).

### 3.2. Data and query graph decomposition

A crucial problem, when we use a cluster of computer nodes to evaluate queries, is how to distribute the data in the computers of the cluster as well as how to compute the queries on the distributed data. In this section we define the concept of data and query graph decomposition.
Definition 5. A data (resp. query) graph decomposition of a data (resp. query) graph $F$ is an $m$-tuple of data graphs $D_F = (F_1, \ldots, F_m)$, where $m \geq 1$, such that:

1. $F_i \subseteq F$, for $i = 1, \ldots, m$, and
2. $\bigcup_i F_i = F$.

Each data (resp. query) graph $F_i$ in a data (resp. query) graph decomposition is called a data (resp. query) graph segment. When, in a data/query graph decomposition, for all the pairs $i, j$, with $1 \leq i < j \leq m$, it also holds $F_i \cap F_j = \emptyset$, i.e. data (resp. query) graph segments are disjoint of each other, then the data (resp. query) graph decomposition is said to be non-redundant and the graph (resp. query) segments obtained form a partition of the triples of data (resp. query) graph $F$, called $m$-triple partition of $F$.

Notice that, $F_i \neq \emptyset$ for $i = 1, \ldots, m$, since, because of Definitions 1 and 2, a data/query graph is nonempty.

It should be noted that, in a data or a query graph decomposition, a triple is (in general) allowed to participate in multiple data or query graph segments. At first sight, this redundancy seems to burden the system with the extra cost of storing and managing or evaluating more data. However, as it is shown in subsequent sections (see for example Section 4.3), if it is used appropriately it may lead to more efficient computation of the query answers, due to proper parallelization of the query execution.

Definition 6. Let $D_F = (F_1, \ldots, F_m)$, with $m \geq 1$, be a data (resp. query) graph decomposition of a data graph $F$, and $F_i$, $F_j$, with $i \neq j$, be two data (resp. query) graph segments in $D_F$. A border node $v$ of $F_i$ and $F_j$, is a node that belongs to $\mathcal{N}(F_i) \cap \mathcal{N}(F_j) - L$. By $B(F_i, F_j)$ we denote the set of border nodes of
$F_i$ and $F_j$, while, by $B(F_i)$, we denote the set $\bigcup_{1 \leq j \leq m \wedge j \neq i} B(F_i, F_j)$. Finally, by $B(F)$ we denote the set of all border nodes of $F$ i.e. $B(F) = \bigcup_{1 \leq i \leq m} B(F_i)$.

Notice that, according to the above definition, literals that occur in more than one segments in $D_F$, are not considered to be border nodes.

**Definition 7.** Let $D_Q = (Q_1, \ldots, Q_n)$, where $n \geq 1$, be a query decomposition of a query graph $Q$. A node $n \in B(Q)$ is said to be a common border node if $n \in B(Q_i)$ for each $Q_i$ in $D_Q$. The set of common border nodes in $Q$ is denoted as $CB(Q)$.

**Example 3.** A data graph decomposition $D_G$ (more specifically a 3-triple partition) of the data graph $G$ of Fig. 1(a) appears in Fig. 3. The dark nodes correspond to the border nodes between the data graph segments, that is:

$B(G_1) = \{\text{Person4, Article1}\}$
$B(G_2) = \{\text{Person4, Article1, Article2}\}$
$B(G_3) = \{\text{Article1, Article2}\}$

A decomposition $D_Q$ of a query $Q$ into 3 query graph segments (subqueries) $Q_1$, $Q_2$, and $Q_3$ is illustrated in Fig. 4. The border nodes between the query graph segments are:

$B(Q_1) = \{n1, n2\} = \{?P1, ?A\}$
Example 4. Consider the query graph $Q$ appearing in the left part of Fig. 4. $Q$ represents the query: “Find an article (variable $?A$) and its title (variable $?T$) published in Journal1, which has as authors a person (variable $?P1$) and his supervisor (variable $?P2$).” It is easy to see that the evaluation of this query on the data graph $G$ depicted in Fig. 3 returns the answers:

Answer 1: $(?P1, ?A, ?P2, ?T) = (Person4, Article1, Person1, "Title1").$

Answer 2: $(?P1, ?A, ?P2, ?T) = (Person2, Article2, Person3, "Title2").$

Notice, however, that, we cannot evaluate $Q$ on a single data graph segment in $DG$ depicted in Fig. 3. Instead, all these graph segments are needed in order to compute the answers to this specific query $Q$ as each of them contains part of the data needed to answer the query $Q$.

Query decomposition will be proved very useful in the subsequent sections in query evaluation. The general idea behind the algorithms that will be presented is that, in order to find the answers to a query $Q$, it suffices to decompose $Q$ into a tuple of subqueries $(Q_1, \ldots, Q_m)$, find the answers (or partial answers) of $Q_1, \ldots, Q_m$ and then combine appropriately these answers to construct the answers to the query $Q$.

3.3. Partial embeddings

When a query $Q$ is evaluated over a data graph segment $G_i$ of a data graph $G$, it is likely that no embedding of $Q$ in $G_i$ exists. However, this does not necessarily mean that there is no embedding of $Q$ in $G$, at all. Instead, it
is possible that "part" of an embedding of \( Q \) in \( G \) has images in \( G_i \), while other "parts" of the embeddings have images in other data graph segments of \( G \). Then, to obtain the embedding of \( Q \) in \( G \), we have to combine appropriately these "partial embeddings". This situation is formulated as follows:

**Definition 8.** A partial embedding of a query graph \( Q \) in a data graph \( G \) is a partial mapping \( e : \mathcal{N}(Q) \rightarrow \mathcal{N}(G) \) such that for every node \( v \in \mathcal{N}(Q) \) for which \( e(v) \) is defined, the following properties hold:

1. if \( v \) is not a variable, then \( e(v) = v \).
2. if \( v \) is a variable, then there exists a node \( u \in \mathcal{N}(Q) \) for which \( e(u) \) is defined and an edge label \( p \in \mathcal{E}(Q) \), such that \((v, p, u) \in Q\) and \((e(v), p, e(u)) \in G\) or \((u, p, v) \in Q\) and \((e(u), p, e(v)) \in G\).

A partial embedding is said to be non-trivial if there exists a triple \((v_1, p, v_2) \in Q\) such that both \(e(v_1)\) and \(e(v_2)\) are defined and the triple \((e(v_1), p, e(v_2))\) belongs to \( G \). In other words, a non-trivial partial embedding is a partial embedding that maps at least one edge of \( Q \) in \( G \).

In essence, a partial embedding represents a mapping from a subset of nodes and edges of \( Q \) to a given data graph \( G \). In other words, partial embeddings represent partial answers to \( Q \), provided that, they can be appropriately “combined” with other “compatible” partial embeddings to give complete answers to the query \( Q \).

The intuition behind Condition (2) is that when \( e(v) \) is defined for a variable \( v \) of a query \( Q \), then there is a triple \( t \) in \( Q \) such that the variable is either the subject or the object of \( t \), and \( t \) is mapped, through \( e \), to a triple in the data graph \( G \). Notice that, as we will prove in the next section, no answers are lost by imposing this condition to the definition of partial embeddings, while it substantially restricts the search space for computing partial embeddings.

It is easy to see that a total embedding \( e \) of \( Q \) in \( G \) is also a partial embedding of \( Q \) in \( G \). Moreover, a total embedding of a subquery of \( Q \), corresponds to a partial embedding of \( Q \).

**Definition 9.** Two partial mappings \( e_1 : D_1 \rightarrow R_1 \) and \( e_2 : D_2 \rightarrow R_2 \) are said to be compatible if for every node \( v \in D_1 \cap D_2 \) such that \( e_1(v) \) and \( e_2(v) \) are defined, it is \( e_1(v) = e_2(v) \).

**Definition 10.** Let \( e_1 : D_1 \rightarrow R_1 \) and \( e_2 : D_2 \rightarrow R_2 \) be two compatible partial mappings. The join of \( e_1 \) and \( e_2 \) is the partial mapping \( e : D_1 \cup D_2 \rightarrow R_1 \cup R_2 \) defined as follows:

\[
e(v) = \begin{cases} 
e_1(v) & \text{if } e_1(v) \text{ is defined} \\ e_2(v) & \text{if } e_2(v) \text{ is defined and } e_1(v) \text{ is undefined} \\ \text{undefined} & \text{if both } e_1(v) \text{ and } e_2(v) \text{ are undefined} \end{cases}
\]

Note that, the above definitions apply also to total embeddings as they are partial mappings. Notice also that in the first case of Definition 10, \( e_2(v) \) may
be defined or not. If it is defined, then the compatibility of the two partial mappings (embeddings) implies that \( e_2(v) = e_1(v) \). It is trivial to prove that the join of two compatible partial embeddings is a partial embedding and that the join operation is commutative and associative. Therefore, we can refer to the partial embedding resulting by the join of \( n \) mutually compatible partial embeddings without ambiguity.

It should also be noted that if \( Q' \) is a subquery of a query \( Q \) and \( e \) is a total embedding of \( Q' \) in a data graph \( G \), then \( e \) is a partial embedding of \( Q \) in \( G \).

**Example 5.** The mappings \( e_1 \), with \( e_1(?A) = \text{Article2} \) and \( e_1(?T) = \text{Title2} \), and \( e_2 \), with \( e_2(?A) = \text{Article2} \) and \( e_2(?W) = \text{Person3} \), are partial embeddings of the query graph of Fig. 1(b) in the data graph of Fig. 1(a). \( e_1 \) and \( e_2 \) are compatible and their join is the partial embedding \( e_3 \), with \( e_3(?A) = \text{Article2} \), \( e_3(?T) = \text{Title2} \) and \( e_3(?W) = \text{Person3} \).

### 3.4. Special forms of queries

In this subsection we define several forms of queries. We begin by defining the path queries:

**Definition 11.** A query \( Q \) is said to be a path query of length \( n \), with \( n \geq 1 \), if it is of the form \((v_0, p_1, v_1), (v_1, p_2, v_2), \ldots, (v_{n-1}, p_n, v_n)\).

We now define the generalized star queries as follows:

**Definition 12.** A query \( Q \) is called a generalized star query if there exists a node \( c \in N(Q) \), called the central node of \( Q \) and denoted as \( C(Q) \), such that for every triple \( t = (u, p, v) \in Q \) it is either \( u = c \) or \( v = c \).

We now define three special forms of generalized star queries, namely subject star queries, object star queries and subject-object star queries (\( s \)-query, \( o \)-query, and \( so \)-query for sort respectively).

**Definition 13.** A generalized star query \( Q \) is said to be a subject star query (resp. object star query) if for every triple \( t \in Q \) the central node \( c \) of \( Q \) is the subject (resp. object) of \( t \).

**Definition 14.** A generalized star query \( Q \) is said to be a subject-object star query if for every triple \( t \in Q \), the central node \( c \) of \( Q \) is either the subject or the object of \( t \) and there is a triple \( t' \in Q \), such that \( c \) is the subject of \( t' \).

The interest in the above special forms of queries lies in that these queries are easier to evaluate (are evaluated more efficiently and are amenable to parallelization) than the general query graphs. Besides, as we will see in the next section, we can easily decompose a huge data graph into a set of graph segments such that a query (as those defined above) can be evaluated independently on each graph segment. These special classes of queries have the following property: for every query graph \( Q \) there exist a (non-redundant) decomposition into
s-queries (or o-queries, or so-queries, or path queries of length 1). This follows trivially from the fact that every query that consists of a single triple belongs to each of these classes of queries.

The algorithms for query evaluation proposed in this paper are based on these observations. In the following sections we will use two of the above classes of queries, namely, generalized star queries and subject-object star queries (so-star queries).

4. Query evaluation approaches

In this section we present three procedures for query evaluation. All procedures are based on the idea that the query is decomposed on a set of subqueries which are evaluated on data segments that are obtained from decomposing the data set using various approaches. Finally, we present algorithms for decomposing a query into so-queries.

4.1. Query evaluation using partial embeddings

The query evaluation approach presented below, called QEJPE-algorithm, is based on the idea of computing (possibly in a distributed manner) partial embeddings of subqueries (query graph segments) of a query $Q$ over data segments of a decomposed data graph $G$ and combining these partial embeddings to obtain (total) embeddings of the initial query $Q$ in the data graph $G$. To narrow down the search space for finding partial embeddings we introduce the concept of useful partial embeddings:

**Definition 15.** Let $\mathcal{D}_G = (G_1, \ldots, G_m)$, with $m \geq 1$, be a data graph decomposition of a data graph $G$ and let $e$ be a partial embedding of a query graph $Q$ in some $G_i$. Then $e$ is called a useful partial embedding of $Q$ in $G_i$ if the following conditions hold:

1. $e$ is non-trivial.
2. $e$ is defined for all the nodes in $\mathcal{N}(Q) \cap \mathcal{N}(G_i)$.
3. for each triple $(v, p, u) \in Q$, if $e(v)$ is defined and $e(v) \notin (B(G_i) \cup L)$, then $e(u)$ is also defined and $(e(v), p, e(u))$ is a triple in $G_i$.
4. for each triple $(v, p, u) \in Q$, if $e(u)$ is defined and $e(u) \notin (B(G_i) \cup L)$, then $e(v)$ is also defined and $(e(v), p, e(u))$ is a triple in $G_i$.

Notice that, according to the above definition, if $v$ is a non-variable node of the query graph $Q$ that maps to a non-border node of $G_i$, then the second property implies that $e(v)$ is defined, and the third and fourth properties enforce every triple that contains $v$ to be mapped in $G_i$. More generally, the edges which start from or end to a node that maps to a non-border node in a data graph segment $G_i$ should also have images that belong entirely to $G_i$ otherwise the partial embedding cannot be used to construct a query answer.
Lemma 1. Let $\mathcal{D}_G = (G_1, \ldots, G_m)$, with $m \geq 1$, be a (redundant or non-redundant) data graph decomposition of a data graph $G$ and let $Q$ be a query graph. Then the following statements are equivalent:

1. $e$ is a total embedding of $Q$ in $G$.
2. there exist mutually compatible useful partial embeddings $e_1, \ldots, e_k$ of $Q$ in $G_{i_1}, \ldots, G_{i_k}$, respectively, for some $i_1, \ldots, i_k$ with $1 \leq i_1 < \ldots < i_k \leq m$, that satisfy the following properties:
   (a) for every triple $(v, p, u) \in Q$ there exists some $j$ for which $e_j(v), e_j(u)$ are defined and $(e_j(v), p, e_j(u)) \in G_{i_j}$.
   (b) the join of $e_1, \ldots, e_k$ is $e$.

Proof. Assume that (1) holds, that is, $e$ is an embedding of $Q$ in $G$. Let $Q_i = \{(v, p, u) \in Q : (e(v), p, e(u)) \in G_i\}$, $1 \leq i \leq m$, and let $I$ be the set of indices for which $Q_i$ is non-empty, that is, $I = \{i \mid Q_i \neq \emptyset\}$. Since the query graph $Q$ is non-empty, $I$ must be also non-empty. Suppose that $|I| = k$ and let $i_1, \ldots, i_k$ be the elements of $I$ in increasing order.

For every $j$, $1 \leq j \leq k$, define the following mapping $e_j$ from $Q$ to $G_{i_j}$:

$$e_j(v) = \begin{cases} v & \text{if } v \text{ is a non-variable node in } \mathcal{N}(Q) \cap \mathcal{N}(G_{i_j}) \\ e(v) & \text{if } v \text{ is a variable node in } \mathcal{N}(Q_{i_j}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is not hard to see that $e_j$ is a partial embedding of $Q$ in $G_{i_j}$ and that the join of $e_1, \ldots, e_k$ is exactly $e$. Thus, property (2b) holds. In order to prove that property (2a) holds, consider a triple $(v, p, u) \in Q$. Then, $(v, p, u) \in Q_i$, for some $j$, which implies that $v, u \in \mathcal{N}(Q_{i_j})$ and $e(v), e(u) \in \mathcal{N}(G_{i_j})$. From the definition of $e_j$ it follows that $e_j(v) = e(v)$ and $e_j(u) = e(u)$, which implies that $(e_j(v), p, e_j(u)) = (e(v), p, e(u))$, which is in $G_{i_j}$ by the definition of $Q_{i_j}$.

It remains to prove that $e_j$ is useful. The fact that $e_j$ is non-trivial is straightforward, since $Q_{i_j}$ is non-empty. Moreover, it obviously satisfies condition (2) of Definition 15.

In order to prove that $e_j$ satisfies condition (3) of Definition 15, consider a triple $(v, p, u) \in Q$ such that $e_j(v)$ is defined and $e_j(v) \notin (\mathcal{B}(G_{i_j}) \cup L)$. Since $e$ is an embedding of $Q$ in $G$, it must be $(e(v), p, e(u)) \in G$. Moreover, $e(v) = e_j(v)$ which implies that $e(v)$ is not a border node of $G_{i_j}$ nor an element of $L$. Therefore, $e(v)$ appears only in $G_{i_j}$, which implies that $(e(v), p, e(u))$ must be a triple in $G_{i_j}$. Hence, $(v, p, u) \in Q_{i_j}$, which implies that $u \in \mathcal{N}(Q_{i_j})$ and by the definition of $e_j$ it is $e_j(u) = e(u)$ (i.e. $e_j(u)$ is defined). The fact that $(e_j(v), p, e_j(u))$ is a triple in $G_{i_j}$ is now clear, since it equals $(e(v), p, e(u))$. The proof for condition (4) of Definition 15, is similar.

For the other direction, assume that (2) holds. We first show that $e$ (the join of $e_1, \ldots, e_k$) is a total mapping from $\mathcal{N}(Q)$ to $\mathcal{N}(G)$. Notice that $e_1, \ldots, e_k$ are compatible. Suppose that $v \in \mathcal{N}(Q)$, that is, $v$ appears in some triple of the form $(u, p, v)$ or $(v, p, u)$ in $Q$. Then, $e_j(v)$ is defined for some $j$ (by property (2a)), which implies (using the definition of join) that $e(v)$ is also defined.
We next show that $e$ is an embedding of $Q$ in $G$. Let $v$ be a non-variable element in $N(Q)$. From the definition of join, it must be $e(v) = e_j(v)$ for some $j$. Since $e_j$ is a useful partial embedding, it is $e_j(v) = v$. Therefore, it holds $e(v) = v$.

Finally, consider a triple $(v, p, u) \in Q$. By property (2a), there exists some $j$ such that $(e_j(v), p, e_j(u)) \in G_{i_j}$, which implies that $(e(v), p, e(u)) \in G$ (since $e(v) = e_j(v)$, $e(u) = e_j(u)$, and $G_{i_j} \subseteq G$).

**Lemma 2.** Let $D_Q = (Q_1, \ldots, Q_n)$, with $n \geq 1$, be a query decomposition of a query graph $Q$ and $G$ be a data graph. Then $e$ is a total embedding of $Q$ in $G$ if and only if there exist mutually compatible total 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 $e$.

**Proof.** For the one direction, assume that $e$ is a total embedding of $Q$ in $G$. For every $i$ define $e_i$ to be the restriction of $e$ in $N(Q_i)$ (that is, $e_i : N(Q_i) \to N(G)$, with $e_i(v) = e(v)$). Obviously $e_i$ is a total mapping. Furthermore, for every non-variable element $v \in N(Q_i)$ it is $e_i(v) = e(v) = v$ and for every triple $(v_1, p, v_2) \in Q_i$ it is $(e(v_1), p, e(v_2)) = (e(v_1), p, e(v_2)) \in G$, which implies that $e_i$ is actually an embedding of $Q_i$ in $G$.

Moreover, for every $i, j$ with $i \neq j$, if $v \in N(Q_i) \cap N(Q_j)$ then it is $e_i(v) = e_j(v) = e(v)$, which implies that $e_i$ and $e_j$ are compatible. Therefore, the join $e'$ of $e_1, \ldots, e_n$ exists. It remains to show that $e' = e$. Consider an arbitrary $v \in N(Q)$. Then $v$ appears in some triple $t \in Q$. Since $D_Q$ is a decomposition of $Q$, there exists some $i$ such that $t \in Q_i$. Thus, $v \in N(Q_i)$, which implies that $e_i(v)$ is defined. From the definition of join, it follows that $e'(v) = e_i(v)$, which implies $e'(v) = e(v)$.

For the other direction, assume that $e_1, \ldots, e_n$ are compatible total embeddings of $Q_1, \ldots, Q_n$ in $G$ and let $e$ be their join. Using the same argument as above, we can prove that for every $v \in N(Q)$ there exists some $i$ such that $e_i(v)$ is defined, which implies that $e(v)$ is also defined. Thus, $e$ is a total mapping.

We next show that $e$ is an embedding of $Q$ in $G$. Consider any non-variable element $v \in N(G)$. Since $e$ is total, $e(v)$ is defined. From the definition of join, there must be some $i$ such that $e(v) = e_i(v)$, which implies $e(v) = v$ (since $e_i$ is an embedding).

Finally, let $(v_1, p, v_2)$ be a triple in $Q$. Since $D_Q$ is a decomposition of $Q$, $(v_1, p, v_2)$ belongs to some $Q_i$. Since $e_i$ is a total embedding of $Q_i$ in $G$, it holds $(e_i(v_1), p, e_i(v_2)) \in G$, which implies $(e(v_1), p, e(v_2)) \in G$. □

**Theorem 3.** Let $D_Q = (Q_1, \ldots, Q_n)$, with $n \geq 1$, be a query decomposition of a query graph $Q$ and $D_G = (G_1, \ldots, G_m)$, with $m \geq 1$, be a data graph decomposition of a data graph $G$. Then the following statements are equivalent:

1. $e$ is a total embedding of $Q$ in $G$.
2. for every $j$, with $1 \leq j \leq n$, there exist useful partial embeddings $e_{j,1}, \ldots, e_{j,k_j}$ of $Q_j$ in $G_{i_{j,1}}, \ldots, G_{i_{j,k_j}}$ for some $i_{j,1}, \ldots, i_{j,k_j}$, with $1 \leq i_{j,1} < \ldots < i_{j,k_j} \leq m$ that satisfy the following properties:
(a) for every \( j \), with \( 1 \leq j \leq n \), and every triple \((v, p, u) \in Q_j\) there exists some \( \ell \) such that \( e_{j,\ell}(v) \), \( e_{j,\ell}(u) \) are defined and \((e_{j,\ell}(v), p, e_{j,\ell}(u)) \in G_{i,\ell}\).

(b) for every \( j, j_1, j_2, \ell_1, \ell_2 \), with \( 1 \leq j_1 \leq j_2 \leq n \), \( 1 \leq \ell_1 \leq k_{j_1} \), \( 1 \leq \ell_2 \leq k_{j_2} \), the partial embeddings \( e_{j_1,\ell_1} \) and \( e_{j_2,\ell_2} \) are compatible.

(c) the join of \( e_{j,\ell} \) for all \( j \in \{1, \ldots, n\} \) and all \( \ell \in \{1, \ldots, k_j\} \) is \( e \).

**Proof.** For the one direction, assume that (1) holds, that is, \( e \) is an embedding of \( Q \) in \( G \). From Lemma 2 we conclude that there are mutually compatible total 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 \( e \). Now, from Lemma 1 we conclude that, for each \( Q_j \), there exist mutually compatible useful partial embeddings \( e_{j,1}, \ldots, e_{j,k_j} \) of \( Q_j \) in \( G_{i,1}, \ldots, G_{i,k_j} \) such that property (a) holds and the join of \( e_{j,1}, \ldots, e_{j,k_j} \) is \( e_j \). In order to show that property (b) holds, suppose for the sake of contradiction, that \( e_{j_1,\ell_1} \) and \( e_{j_2,\ell_2} \) are not compatible, for some \( j_1, j_2, \ell_1, \ell_2 \). Then, there exists some \( v \) such that \( e_{j_1,\ell_1}(v) \neq e_{j_2,\ell_2}(v) \). Since \( e_{j_1}(v) = e_{j_1,\ell_1}(v) \) and \( e_{j_2}(v) = e_{j_2,\ell_2}(v) \), the total embeddings \( e_{j_1} \) and \( e_{j_2} \) must also be incompatible, which is a contradiction. Therefore, property (b) holds. Finally, property (c) holds since for all \( j \), the join of \( e_{j,1}, \ldots, e_{j,k_j} \) is \( e_j \) and the join of \( e_1, \ldots, e_n \) is \( e \).

For the other direction, assume that (2) holds. From Lemma 1, it follows that for every \( j \), the join \( e_j \) of the partial embeddings \( e_{j,1}, \ldots, e_{j,k_j} \) is a total embedding of \( Q_j \) in \( G \). Moreover, the resulting embeddings \( e_1, \ldots, e_n \) are mutually compatible, since we have assumed that \( e_{j_1,\ell_1} \) and \( e_{j_2,\ell_2} \) are compatible for all \( j_1, j_2, \ell_1, \ell_2 \). Now, from Lemma 2 it follows that the join \( e \) of \( e_1, \ldots, e_n \) is a total embedding of \( Q \) in \( G \).

**Theorem 3** implies a generic query evaluation strategy, named **Query Evaluation by Joining Partial Embeddings (QEJPE) strategy** consisting of four steps. The algorithm assumes an arbitrary decomposition of the data graph \( G \) into a tuple \( D_G \) of data graph segments \( G_1, \ldots, G_m \), with \( m \geq 1 \), stored into a cluster of computer nodes.

**Step 1:** Decompose the query \( Q \) into a tuple \( D_Q \) of subqueries \( Q_1, \ldots, Q_n \), with \( n \geq 1 \).

**Step 2:** Compute all possible useful partial embeddings of each subquery \( Q_j \) over each data graph segment \( G_i \) of \( G \).

**Step 3:** For each subquery \( Q_j \), collect all the partial embeddings of \( Q_j \) obtained in Step 2 and join them to get total embeddings of \( Q_j \).

**Step 4:** To construct the total embeddings (i.e. answers) of \( Q \), join the total embeddings obtained in Step 3 by using one embedding for each subquery, in all possible ways.

Notice that the above generic query evaluation strategy has several interesting properties: a) it is independent of the way the data graph is decomposed and the way the data graph segments obtained by this decomposition are stored
in the nodes of the cluster, b) it is independent of the way the query graph is decomposed, and c) it is independent of the algorithm used to compute (partial) embeddings.

In Subsection 5.4, we present an implementation of this strategy on a cluster of commodity computers based on the Map-Reduce programming framework.

4.2. Query evaluation by decomposing queries into generalized stars

In this section we present another approach, called eval-STARS algorithm, for evaluating queries over linked data. The algorithm is based on assumptions similar to these on which the QEJPE-algorithm, presented in Section 4.1, is based. The main difference is that we now impose subqueries obtained from the decomposition of a user query \( Q \) to be in the form called generalised star queries. Besides, the algorithm is based on evaluation of total embeddings of the subqueries instead of partial embeddings.

Recall that, as we proved in Lemma 2, in order to compute the answers to a given query in a data graph \( G \), we can decompose the query into a tuple of subqueries, compute the embeddings of the subqueries in \( G \) (which may be a more efficient task due to the simpler or 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, if \( C \) is the class of path queries of length 3, in other words if the subqueries must be of the form \( \{(u,p,v),(v,p',w),(w,p'',z)\} \), then it can be proved that it is not possible to decompose every user query into a set of path queries of length 3. Nevertheless, if the target class \( C \) is the class of generalised star queries, then for every query \( Q \) there exist a (non-redundant) decomposition of \( Q \) into a tuple of generalised star subqueries. This trivially follows from the fact that every query that consists of a single triple is also a generalised star query (with either the subject or the object being the central node). We next present a more general result, relating the decomposition of a query graph \( Q \) into generalised star subqueries, to the node covers of this query graph.

**Definition 16.** Let \( Q \) be a query graph. A set of nodes \( V \subseteq N(Q) - L \) is called a **node cover** of \( Q \) if for every triple \( (s,p,o) \in Q \), it holds either \( s \in V \) or \( o \in V \).

**Lemma 4.** Let \( Q \) be a query graph and \( V = \{v_1,\ldots,v_k\} \) be a node cover of \( Q \). For each \( v_i \in V \) define the generalised star query \( Q_{v_i} = \{ t \in Q \mid t = (s,p,v_i) \} \cup \{ t \in Q \mid t = (v_i,p,o) \text{ and } o \notin V \} \). Then \( D_Q = (Q_{v_1},\ldots,Q_{v_k}) \) is a non-redundant decomposition of \( Q \).

**Proof.** It easy to see that \( (Q_{v_2},\ldots,Q_{v_k}) \) forms a decomposition of \( Q \) since:

1. by construction \( Q_{v_i} \subseteq Q \), for \( i = 1,\ldots,k \); and
2. \( \bigcup_i Q_{v_i} = Q \), since for every triple \( t = (s,p,o) \in Q \), either \( s \in V \) or \( o \in V \). If \( o \in V \) then, by construction \( t \in Q_o \). Otherwise (i.e. if \( s \in V \) and \( o \notin V \)) then \( t \in Q_s \).

We will now prove (by contradiction) that \( D_Q = (Q_{v_1},\ldots,Q_{v_k}) \) is non-redundant. Assume that \( D_Q \) is redundant. Then there exists a triple \( t = (s,p,o) \in Q \) such that \( t \notin \bigcup_i Q_{v_i} \).
\((s, p, o) \in Q\) such that \(t\) belongs to two different subqueries in \(D_Q\). It is easy to see that these subqueries should be \(Q_s\) and \(Q_o\) and \(s, o \in V\). However, since \(t \in Q_s\), then, by construction, \(o \not\in V\), which contradicts with the fact that \(s, o \in V\).

Therefore, if a set of nodes is a node cover of a query \(Q\), then its elements are the central nodes of the generalized star subqueries in a non-redundant decomposition of \(Q\). It turns out that the converse also holds.

**Lemma 5.** Let \(Q\) be a query graph, let \(D_Q = (Q_1, \ldots, Q_k)\) be a decomposition of \(Q\) such that \(Q_1, \ldots, Q_k\) are generalized star queries and let \(c_1, \ldots, c_k\) be their central nodes. Then \(\{c_1, \ldots, c_k\}\) is a node cover of \(Q\).

**Proof.** It immediately follows from Definitions 5, 12 and 16.

**Example 6.** In Fig. 5 we see a decomposition of the query \(Q\) into three generalized star queries \(Q_1, Q_2\) and \(Q_3\), which is obtained by the construction of Lemma 4, using the node cover \(\{n_4, n_2, n_5\}\) of \(Q\).

Figure 5: Query decomposition into star queries.

Following the discussion above we can be specialize the generic query evaluation strategy (QEJPE strategy) presented in Subsection 4.1, obtaining in this way a new algorithm called eval-STARS algorithm. As in the case of QEJPE strategy we assume an arbitrary decomposition of the data graph \(G\) into a tuple \(D_G\) of data graph segments \(G_1, \ldots, G_m\), with \(m \geq 1\), stored into a cluster of computer nodes.

The **eval-STARS algorithm** algorithm consists of the following steps:

**Step 1:** Decompose the query \(Q\) into a tuple of generalized star subqueries \(D_Q = (Q_1, \ldots, Q_n)\), with \(n \geq 1\).

**Step 2:** Compute all possible embeddings of each triple in \(Q\) over each data graph segment \(G_i\) of \(G\).

**Step 3:** For each subquery \(Q_j\), collect the embeddings of all the triples in \(Q_j\) and join compatible embeddings in all possible ways to compute the total embeddings of \(Q_j\) in \(G\).
Step 4: To construct the total embeddings (i.e. answers) of \( Q \), join the total embeddings obtained in Step 3 by using one embedding for each subquery, in all possible ways.

Note that eval-STARS algorithm applies two query decomposition processes. Initially, the given query is decomposed into generalized star queries (Step 1 of eval-STARS) and each star query is further decomposed (Step 2 of eval-STARS) into its triples. On the contrary, QEJPE applies a single decomposition (Step 1 of QEJPE). Following this stepwise approach of two decompositions, in fact, we achieve the construction of the total embeddings in two phases, where each phase gathers the compatible partial embeddings and join them together (i.e., it applies Lemma 2 twice). Although this extra decomposition could be thought of as a redundant step, in parallel computation (see Section 5), such an approach brings a significant performance improvement and facilitates the distribution of both the intermediate data and the computation.

4.3. Query evaluation by data decomposition using replication

In this section we propose a query evaluation approach, called \( QE\text{-}with\text{-}Redundancy \), which uses a specific form of replication in the data graph decomposition to efficiently answer queries. More specifically:

(a) Data graphs are decomposed into data graph segments in which replication of the data triples is allowed. Data triples are replicated in such a way that all the answers to a special form of queries, namely subject-object star queries, can be obtained from a single data segment. The partition of the data graph is specified by an arbitrary partition of the data nodes, while data segments consist of the in- and out-edges of each block of nodes. Therefore, triples containing nodes that are in two different blocks occur in both segments of the data graph corresponding to these blocks. This redundancy, as we show, ensures that the subject-object star subqueries can be easily evaluated over each segment, independently.

(b) Each query posed by the user is decomposed into a tuple of subject-object star subqueries.

In the evaluation strategy presented in this section, our aim is to construct the embeddings of a query \( Q \) in a data graph \( G \), by appropriately combining embeddings (i.e. joining compatible embeddings) of so-subqueries of \( Q \) over the proper subgraphs of \( G \).

The following lemma refers to the compatibility of embeddings:

**Lemma 6.** Let \( D_Q = (Q_1, \ldots, Q_n) \), with \( n \geq 1 \), be a query decomposition of a query graph \( Q \) and \( D_G = (G_1, \ldots, G_m) \), with \( m \geq 1 \), be a data graph decomposition of a data graph \( G \). Let \( e_{Q_i} \) and \( e_{Q_j} \) be two embeddings of the subqueries \( Q_i \) and \( Q_j \) respectively, with \( 1 \leq i \neq j \leq n \), on two (not necessarily different) graph segments \( D_k \) and \( D_l \) in \( D_G \). Let \( B(Q_i, Q_j) \) be the border nodes of \( Q_i, Q_j \). Then \( e_{Q_i} \) and \( e_{Q_j} \) are compatible if and only if for each node \( v \in B(Q_i, Q_j) \), it holds that \( e_{Q_i}(v) = e_{Q_j}(v) \).
Proof. It immediately follows from Definitions 9 and 6. □

In the following definition we present a decomposition scheme for a data graph $G$, called star-oriented decomposition (or simply s-decomposition).

Definition 17. A star-oriented decomposition (or s-decomposition for short) of a data graph $G$ is a tuple of graphs $D_G = (G_1, \ldots, G_m)$, with $m \geq 1$, if $N_P = (N_1, \ldots, N_m)$ is a partition of the nodes in $N(G) - L$ and for each $i$, with $1 \leq i \leq m$, $G_i = \{ t \mid t = (s, p, o) \text{ and } t \in G \text{ and } s \in N_i \text{ or } o \in N_i \}$. Subgraphs $G_1, \ldots, G_m$ are called s-graph segments. A node in $N(G_i) - L - N_i$ is called a replicated node in $G_i$. A replicated triple $t = (s, p, o)$ in a s-graph segment $G_i$ is a data triple in $G_i$ such that either $s$ or $o$ is a replicated node.

In the following, the set of replicated nodes in a s-graph segment $G_i$ is denoted by $R_N(G_i)$. The replicated nodes of a data graph $G$ is $R_N(G) = \bigcup_i R_N(G_i)$. Similarly, the set of replicated triples in a s-graph segment $G_i$ is denoted by $R_t(G_i)$. Finally, replicated triples of a data graph $G$ is $R_t(G) = \bigcup_i R_t(G_i)$.

Example 7. Fig. 6 shows an s-decomposition of the data graph $G$ of Fig. 1(a), which is based on the following partition of the set of nodes in $N(G) - L$: $N_1 = \{ \text{Article1, Article3, Journal2, Person4} \}$ $N_2 = \{ \text{Person1, Person2, Person3} \}$ $N_3 = \{ \text{Article2, Journal1} \}$.

The grey colored nodes in the segments $G_1$, $G_2$, and $G_3$ correspond to the nodes in $N_1$, $N_2$, and $N_3$, respectively, while the pink colored nodes are the...
replicated nodes. Finally, the dashed lines in the graph segments correspond to replicated data triples. Consider now the query graph \( Q \) appearing in the right part of Fig. 1. It is easy to see that we cannot obtain the solution described in Example 1 by finding an embedding of \( Q \) in a single graph segment of \( G \) appearing in Fig. 6 (as such an embedding does not exist).

The following lemma presents some interesting properties of the star-oriented decomposition of a data graph.

**Lemma 7.** Let \( \mathcal{D}_G = (G_1, \ldots, G_m) \), with \( m \geq 1 \), be an \( s \)-decomposition of a data graph \( G \) based on the partition \( \mathcal{N}_s = (N_1, \ldots, N_m) \) of the nodes in \( \mathcal{N}(G) - L \). Then the following hold:

1. \( (\mathcal{N}(G_i) - L) \supseteq N_i \), for each \( i \), with \( 1 \leq i \leq m \).
2. \( \bigcup_{i \leq m} \mathcal{N}(G_i) = \mathcal{N}(G) \)
3. \( \bigcup_{i \leq m} G_i = G \)
4. Consider a node \( s \in \mathcal{R}_N(G_i) \). Then for each triple \( t = (s, p, o) \in G_i \) it holds that \( o \in N_i \) and \( t \in \mathcal{R}_t(G_i) \).
5. Consider a node \( o \in \mathcal{R}_N(G_i) \). Then for each triple \( t = (s, p, o) \in G_i \) it holds that \( s \in N_i \) and \( t \in \mathcal{R}_t(G_i) \).
6. For each node \( v \in \mathcal{R}_N(G_i) \), with \( 1 \leq i \leq m \) there exists an index \( j \), with \( 1 \leq j \leq m \) and \( i \neq j \), such that \( v \in N_j \).
7. For each triple \( t \in \mathcal{R}_t(G_i) \), with \( 1 \leq i \leq m \), there exists an index \( j \), with \( 1 \leq j \leq m \) and \( i \neq j \), such that \( t \in G_j \).

**Proof.**

Proof of 1: It immediately follows from Definition 17.

Proof of 2: Let \( v \) be a node in \( \bigcup_{i \leq m} \mathcal{N}(G_i) \). Then, there exist a triple \( (s, p, o) \in G_i \), for some \( i \), such that \( v = s \) or \( v = o \). Since \( G_i \subseteq G \), we have \( (s, p, o) \in G \), which implies that \( s, o \in \mathcal{N}(G) \). Therefore, \( v \in \mathcal{N}(G) \).

Now let \( v \) a node in \( \mathcal{N}(G) \). Then, there exist a triple \( (s, p, o) \in G \), such that \( v = s \) or \( v = o \). Since \( s \) is the subject of this triple, it must be \( s \in \mathcal{N}(G) - L \), which implies that \( s \in N_i \), for some \( i \). Therefore, \( (s, p, o) \in G_i \) and thus \( s, o \in \mathcal{N}(G_i) \). Consequently, \( v \in \bigcup_{i \leq m} \mathcal{N}(G_i) \).

Proof of 3: From Definition 17 we conclude that \( \bigcup_{i \leq m} G_i \subseteq G \). To prove the inverse let \( t = (s, p, o) \) be a triple in \( G \). Then \( s \in (\mathcal{N}(G) - L) \). Thus \( s \in N_i \) for some \( i \) with \( 1 \leq i \leq m \). Hence, by construction of the s-segments, \( t \in G_i \) and therefore \( t \in \bigcup_{i \leq m} G_i \). Therefore, \( G \subseteq \bigcup_{i \leq m} G_i \).

Proof of 4: It immediately follows form Definition 17.

Proof of 5: It immediately follows form Definition 17.

Proof of 6: As \( v \in \mathcal{R}_N(G_i) \), from Definition 17 we conclude that \( v \notin N_i \). But as \( v \) is a node in \( \mathcal{N}(G) - L \), the node \( v \) should belong to another set \( N_j \), with \( j \neq i \), of the partition of the nodes in \( \mathcal{N}(G) - L \).

Proof of 7: Assume that \( t \) is of the form \( t = (s, p, o) \). As \( t \in \mathcal{R}_t(G_i) \), from Definition 17 we see that either \( s \) or \( o \) is a replicated node in \( \mathcal{R}_N(G_i) \). Assume that the replicated node is \( s \). Then, from (6) we conclude that there is an index
Theorem 8. Let $D_Q = (Q_1,\ldots,Q_n)$, with $n \geq 1$, be a query decomposition of a query graph $Q$, such that each $Q_i$, with $1 \leq i \leq n$, is an so-query. Let also $D_G = (G_1,\ldots,G_m)$, with $m \geq 1$, be an s-decomposition of a data graph $G$. Then $e$ is a total embedding of $Q$ in $G$ if and only if $e$ is the join of $e_1,\ldots,e_n$, where $e_1,\ldots,e_n$ are mutually compatible embeddings such that for each $i$, with $1 \leq i \leq n$, $e_i$ is a total embedding of $Q_i$ in some segment $G_j$, where $1 \leq j \leq m$.

Proof. For the one direction, assume that $e$ is a (total) embedding of $Q$ in $G$. For every $i$, with $1 \leq i \leq n$, define $e_i$ to be the restriction of $e$ in $Q_i$ (that is, $e_i : \mathcal{N}(Q_i) \rightarrow \mathcal{N}(G)$, with $e_i(v) = e(v)$ for every node $v \in \mathcal{N}(Q_i)$). Obviously $e_i$ is a total mapping. Furthermore, for every non-variable element $v \in \mathcal{N}(Q_i)$ it is $e_i(v) = e(v) = v$ and for every triple $(v_1,p,v_2) \in Q_i$ it is $(e_i(v_1),p,e_i(v_2)) = (e(v_1),p,e(v_2)) \in G$, which implies that $e_i$ is actually an embedding of $Q_i$ in $G$.

As $Q_i$ is an so-query, let $C(Q_i)$ be the central node of $Q_i$ and $e(C(Q_i)) \in \mathcal{N}(G)$ the image of $C(Q_i)$ in $G$. From Definition 17 we conclude that $e(C(Q_i)) \in N_j$, for some $N_j$ with $1 \leq j \leq m$ and that $e_i$ is an embedding of $Q_i$ in $G_j$.

We next prove that the embeddings $e_i$, with $1 \leq i \leq n$ are mutually compatible and their join is $e$. By construction, for every $i,j$ with $i \neq j$, if $v \in \mathcal{N}(Q_i) \cap \mathcal{N}(Q_j)$ then it is $e_i(v) = e_j(v) = e(v)$, which implies that $e_i$ and $e_j$ are compatible. Therefore, the join $e'$ of $e_1,\ldots,e_n$ exists. It remains to show that $e' = e$. Consider an arbitrary $v \in \mathcal{N}(Q)$. Then $v$ appears in some triple $t \in Q$. Since $D_Q$ is a decomposition of $Q$, there exists some $i$ such that $t \in Q_i$. Thus, $v \in \mathcal{N}(Q_i)$, which implies that $e_i(v)$ is defined. From the definition of join, it follows that $e'(v) = e_i(v)$, which implies $e'(v) = e(v)$.

For the other direction, assume that for each $i$, with $1 \leq i \leq m$, there is an embedding $e_i$ for the subquery $Q_i$ in some graph segment $G_j$. Assume also that $e_1,\ldots,e_n$ are mutually compatible embeddings and let $e$ be their join. We will prove that $e$ is an embedding of $Q$ in $G$. It is easy to see that for every $v \in \mathcal{N}(Q)$ there exists some $i$ such that $e_i(v)$ is defined, which implies that $e(v)$ is also defined. Thus, $e$ is a total mapping.

We next show that $e$ is an embedding of $Q$ in $G$. Consider any non-variable element $v \in \mathcal{N}(G)$. Since $e$ is total, $e(v)$ is defined. From the definition of join, there must be some $i$ such that $e(v) = e_i(v)$, which implies $e(v) = v$ (since $e_i$ is an embedding).

Finally, let $(v_1,p,v_2)$ be a triple in $Q$. Since $D_Q$ is a decomposition of $Q$, $(v_1,p,v_2)$ belongs to some $Q_i$. Since $e_i$ is a total embedding of $Q_i$ in an s-segment of $G$, it is also a total embedding of $Q_i$ in $G$. Thus, $(e_i(v_1),p,e_i(v_2)) \in G$, which implies $(e(v_1),p,e(v_2)) \in G$. □

23
The above theorem suggests the following strategy for the evaluation of a query \( Q \) on a data graph \( G \), called QE-with-Redundancy. QE-with-Redundancy strategy assumes a star-oriented decomposition of the data graph \( G \). To obtain such a decomposition we assume an arbitrary partition \( \mathcal{N}_P = (N_1, \ldots, N_m) \), with \( m \geq 1 \) of the nodes in \( \mathcal{N}(G) \). Then we decompose the data graph \( G \) into a tuple of graph segments \( D_G = (G_1, \ldots, G_m) \), such that \( D_G \) is a star-oriented decomposition of \( G \) (as defined in Definition 17).

The QE-with-Redundancy strategy consists of the following steps:

**Step 1:** Decompose the query \( Q \) into a tuple of queries \( D_Q = (Q_1, \ldots, Q_n) \), with \( n \geq 1 \), such that each query in \( D_Q \) is a subject-object star query.

**Step 2:** Compute all possible embeddings of each subquery in \( D_Q \) on every segment in \( D_G \).

**Step 3:** Compute the embeddings of \( Q \) on \( G \) by joining compatible embeddings of the subqueries \( Q_1, \ldots, Q_n \).

It is important to note that the algorithm is independent of the choice of the specific partition \( N_P \) of the nodes in \( \mathcal{N}(G) \) used for the data graph decomposition, as well as of the specific query decomposition strategy (employed in Step 1).

### 4.4. Query decomposition algorithms

In this section, we present and analyze algorithms for decomposing queries into a set of so-subqueries. In the previous subsections, we assumed that the queries are decomposed into a set of subqueries, but we have not typically discussed any algorithm for finding such a decomposition, so far. Although the algorithms presented in Sections 4.1, 4.2, and 4.3 can be used to evaluate a query over a single machine, they are designed to be efficiently applied on a distributed environment, as we will see in the next sections. In this context, we focus on decomposition algorithms that can boost parallelization. Furthermore, although the QEJPE algorithm (Sections 4.1) is quite generic and can support every query decomposition, the algorithms presented in this section aim to take advantage of the special, so-queries decomposition, which can be used in the evaluation algorithms eval-STARS (Section 4.2) and QE-with-Redundancy (Section 4.3).

Intuitively, the decomposition approach followed can affect the efficiency of the overall query evaluation process, since an appropriate algorithm can significantly reduce the amount of the data transferred through the network (i.e., the amount of intermediate results). For example, in the extreme scenario that we decompose the query in Figure 2 so that each edge defines a different subquery, it is easy to see that all the 6 edges with predicate “hasAuthor” are mapped by the edge-subquery \( \{(?A, \text{hasAuthor, } ?W)\} \); hence, 6 embeddings are resulted by Step 2 of the QE-with-Redundancy algorithm and passed to Step 3. If, however, we decompose the query \( Q \) in such a way that at least one constant (i.e., non-variable node) is included in each subquery, the number of embeddings found
in Step 2 and used in Step 3 is significantly reduced; e.g., consider the query \{ (?A, hasAuthor, ?W), (?A, year, 2008) \}, or \( Q \) itself. Practically, the more the number of constants each query has, the less the embeddings that are found, since the constants filter out useless embeddings (i.e., partial embeddings that surely cannot be used in Step 3 to construct a total embedding). The Steps 3 and 4 of the eval-STARS algorithm operate similarly. The following proposition proves this statement.

**Proposition 9.** Let \( Q_1 \) and \( Q_2 \) be two generalized star queries, such that \( Q_2 \subseteq Q_1 \) and each triple in the set \( (Q_1 - Q_2) \) is either of the form \((C, p, c)\) or of the form \((c, p, C)\), where \( C = C(Q_1) = C(Q_2) \), \( p \) is a predicate, and \( c \) is not a variable. Then, for every data graph \( G \) the set of answers of \( Q_1 \) over \( G \) is a subset of the set of answers of \( Q_2 \) over \( G \), and \( n_1^i \leq n_2^i \), where \( n_i^i \) is the number of embeddings of \( Q_i \) over \( G \), with \( i = 1, 2 \).

The proof of the previous proposition is straightforwardly given by expressing both queries as conjunctive queries and checking containment of the corresponding conjunctive queries \([20, 14]\).

To maximize the number of constants in each subquery, one could come up with the following simple decomposition algorithm (called naive algorithm). Let \( Q \) be a query.

**Step 1:** For each node \( n \) in \( N(Q) \), we construct the star query \( Q_n \) such that \( Q_n \) includes all the edges in \( Q \) of either the form \((n, p, m)\) or the form \((m, p, n)\), where \( m \in N(Q) \). Let \( S_Q \) be the set including all the subqueries constructed by this process.

**Step 2:** We, then, remove from \( S_Q \) the subqueries that are not so-queries.

It is easy to see that the remaining subqueries in \( S_Q \) form a decomposition \( D_Q \) of \( Q \) that can be used in both QE-with-Redundancy and eval-STARS algorithms.

**Proposition 10.** Considering a query \( Q \), the naive algorithm results a decomposition \( D_Q \) of \( Q \) such that each query in \( D_Q \) is an so-query.

The proof of the Proposition 10 is straightforward since each edge of \( Q \) will be at least in the subquery centered by its subject. Furthermore, it is easy to see that the naive algorithm results a query decomposition that maximizes the number of constants in each star subquery, since each subquery is constructed by a query node along with all of its adjacent edges. This algorithm, however, results a quite large number of subqueries as at the worst-case scenario one subquery for each query edge is obtained (the subject of each query edge, may introduce a new subquery), and does not limit the number of variables in each subquery, which might impact the overall performance of the query evaluation, as we will see in the next sections.

In the following sections, we study two additional parameters, the number of variables into each subquery and the number of subqueries in the decomposition, in order to provide an effective decomposition approach. In particular,
in Section 4.4.1, we present an algorithm that decomposes a query in a way that the number of variables do not exceed a given threshold. Section 4.4.2 discusses multiple algorithms that aim to reduce the number of subqueries into the decomposition.

4.4.1. Subqueries with a limited number of variables

In this section, we present a decomposition algorithm that aims to keep the number of the variables in each subquery as low as possible. As Proposition 9 shows, by decomposing into so-subqueries with large number of constants we can achieve significant improvement in the overall performance of both QE-with-Redundancy and eval-STARS algorithms. However, we might not have the same result if we choose star subqueries with large number of variables. To see which is the impact of the number of variables into the overall query evaluation process we start our analysis with an example.

Consider the simple query

\[ Q = \{(c, p_1, ?X), (c, p_2, ?Y), (?X, p_3, ?Y)\} \]

and the data graph

\[ G = \{(c, p_1, c_{11}), (c, p_1, c_{12}), (c, p_2, c_{21}), (c, p_2, c_{22}), (c, p_2, c_{23})\} \]

where \( p_1, p_2, p_3 \) are two predicates and \( c, c_{ij} \) are either URIs or literals. Suppose now a decomposition

\[ D = \{Q_1, Q_2\} \]

of \( Q \), such that

\[ Q_1 = \{(c, p_1, ?X), (c, p_2, ?Y)\} \]

and

\[ Q_2 = \{(?X, p_3, ?Y)\} \]

Notice that although the answers of both \( Q \) and \( Q_2 \) are empty, there are 9 total embeddings from \( Q_1 \) to \( G \), giving 9 answers. Looking at the algorithms eval-STARS (Steps 4 and 5) and QE-with-Redundancy (Steps 3 and 4), it is worth further decomposing \( Q_1 \) into two subqueries \( Q_{11}, Q_{12} \), one for each edge, instead of keeping \( Q_1 \) into \( D \). In particular, if we replace \( Q_1 \) in \( D \) with \( Q_{11} \) and \( Q_{12} \), the number of embeddings found in Step 4 of eval-STARS and passed to Step 5 (resp., found in Step 3 of QE-with-Redundancy and passed to Step 4) is 6, instead of 9 in the case we use \( Q_1 \).

In the previous example, we saw that the presence of multiple variables in a subquery might increase the number of embeddings of this subquery in both QE-with-Redundancy and eval-STARS algorithms. Especially, if we apply these algorithms into a distributed environment, as we will see in the next sections, we might have significant impact on the performance of each algorithm, since the communication cost might be increased tremendously from the large number of embeddings transferred through the network.

In this context, we present the \textit{min-res} decomposition algorithm, which finds a decomposition by keeping the number of variables into each subquery at most 2. One could wonder how we come up with the threshold number 2. Typically, we want to keep the number of variables in each subquery as low as possible. If we set such a threshold to one variable, we miss edges consisting of two variables, i.e., we cannot find a valid decomposition of any given query.

\textit{Min-res} algorithm decomposes a query \( Q \) into a set of so-subqueries, such that each subquery has at most two variables. It also allows replication of triples that contains at most one variable, and maximizes the number of “constraints” (triples that do not increase the number of variables in the query) in each subquery containing variables. As for the subqueries that do not contain
any variable, the algorithm constructs maximal subqueries without redundant constraints. The min-res decomposition algorithm, in detail, is given as follows.

\[\text{min-res}(Q)\]

// Q a query.
// The min-res function returns a decomposition R of Q consisting of so-subqueries of Q
begin
  \[T_{\text{sub-obj}} = \{ t \in Q \mid t = (s, p, o) \text{ and } s, o \in V(Q) \}; \] // subject and object are variables
  \[T_{\text{obj}} = \{ t \in Q \mid t = (s, p, o) \text{ and } s \in V(Q) \text{ and } o \notin V(Q) \}; \] // only subject is variable
  \[T_e = \{ t \in Q \mid t = (s, p, o) \text{ and } s, o \notin V(Q) \}; \] // subject and object are nonvariables
  \[R = \emptyset;\]
foreach \( t = (s, p, o) \in T_{\text{sub-obj}} \) do // select a maximal so-query centered at subject of \( t \)
  \[Q_s = \{ t \mid |t| = \{ (s, p', c) \} \}; \]
  // \( Q_s \) is an so-query with central node \( s \)
  \[S = \{ t' \mid t' \in T_{\text{sub}} \text{ and } t' = (\{ s, p', c \}); \]
  \[Q_o = \{ t \mid t \in T_{\text{obj}} \text{ and } o \in V(Q) \}; \]
end

\[T_{\text{sub}} = T_{\text{sub}} - \{ t \mid t \in Q' \text{ and } Q' \in R \}; \] // Remove from \( T_{\text{sub}} \) the triples used so far
\[T_{\text{obj}} = T_{\text{obj}} - \{ t \mid t \in Q' \text{ and } Q' \in R \}; \] // Remove from \( T_{\text{obj}} \) the triples used so far
while \( T_{\text{obj}} \neq \emptyset \) do // For each member of \( T_{\text{obj}} \), construct an so-query
begin
  \[Q' = \{ t \mid t \in T_{\text{sub}} \text{ and } t' = (s, p', c) \} \cup \{ t'' \mid t'' \in T_{\text{obj}} \text{ and } t''' = (c', p'', s) \}; \]
end

\[T_{\text{sub}} = T_{\text{sub}} - \{ t \mid t \in Q' \}; \] // Remove from \( T_{\text{sub}} \) the triples used in \( Q' \)
\[T_{\text{obj}} = T_{\text{obj}} - \{ t \mid t \in Q' \}; \] // Remove from \( T_{\text{obj}} \) the triples used in \( Q' \)
end
foreach \( t = (s, p, o) \in T_{\text{obj}} \) do // For each member of \( T_{\text{obj}} \), construct an so-query
begin
  \[Q = \{ t \mid t \in T_{\text{sub}} \text{ and } t' = (s, p', c) \} \cup \{ t'' \mid t'' \in T_{\text{obj}} \text{ and } t''' = (c', p'', s) \}; \]
end
\[T_e = T_e - \{ t \mid t \in Q' \text{ and } Q' \in R \}; \] // Remove from \( T_e \) the triples used in \( Q' \)
end
return \( R; \)
end.

Intuitively, the algorithm performs as follows. Let \( Q \) be a query. Initially, for each edge \( t \) of two variables in \( Q \), it constructs an so-query \( Q_s \) having the subject \( s \) of \( t \) as central node. All the adjacent edges of \( s \) in \( Q \) such that \( s \) is their only variable are added into the subquery. In each construction step,
the possibility to get an so-query \( Q_o \), whose central node is the object of \( t \), is also considered, and the query with maximum number of edges between \( Q_s \) and \( Q_o \) is finally selected. It is easy to see that the subqueries constructed in this step include 2 variables. Next, the algorithm constructs the subqueries that have a single variable as central code. These subqueries have at least one edge whose subject is the central variable. Then, the remaining query triples give so-subqueries whose central node is not a variable. Each of the subqueries constructed in this step have at most a single variable which is not the central node. Notice here that the min-res algorithm constructs subqueries with two variables only if those variables are used by an edge in \( Q \). Note also that in each of the aforementioned steps, we build an so-subquery by initially selecting an edge from a set (e.g., the set of edges having two variables). The order the edges are selected might give different decompositions. Here, we consider an arbitrary ordering of the edges included in each set. The following proposition shows that the min-res algorithm results a decomposition of a query into a set of so-queries.

**Proposition 11.** Considering a query \( Q \), the min-res algorithm results a decomposition \( D_Q \) of \( Q \) such that each query in \( D_Q \) is an so-query.

**Proof.** Consider the sets \( T_{\text{sub-obj}}, T_{\text{sub}}, T_{\text{obj}}, T_c \) of edges as defined in the min-res algorithm. To prove that \( R \) is a decomposition of \( Q \), we need to show that (1) each query \( Q_i \) in \( R \) is a subquery of \( Q \), and (2) \( \bigcup_{Q_i \in R} Q_i = Q \). The first condition is straightforward since each edge of \( Q_i \) is an edge in \( T_{\text{sub-obj}} \cup T_{\text{sub}} \cup T_{\text{obj}} \cup T_c \), which equals \( Q \). To show the second condition, we need to prove that each edge \( t \) of \( Q \) is included in at least one subquery in \( R \). Since the algorithm uses all the edges in \( T_{\text{sub-obj}} \cup T_{\text{sub}} \cup T_{\text{obj}} \cup T_c \) to construct the subqueries, we have that \( t \) is included in at least one subquery in \( R \). Besides, it is easy to see that, by construction, all queries in \( D_Q \) are so-queries.

**Example 8.** Consider the query \( Q \) depicted in Figure 5. Figure 7 illustrates a decomposition of \( Q \) resulted by the min-res algorithm. In particular, we initially select the edge \( t_2 \) and construct \( Q_1 \). Similarly, the queries \( Q_2 \), \( Q_3 \) and \( Q_4 \) are given by selecting the edges \( t_3, t_4 \) and \( t_5 \), respectively, having two variables, as well. \( Q_5 \) and \( Q_6 \) are then constructed by selecting the corresponding edges of \( Q \). Note here that the edge \( t_6 \) is replicated to multiple subqueries, as min-res algorithm requires, since it can reduce the number of intermediate answers through the constant “2008”.

4.4.2. Reducing the number of subqueries

Unlike the min-res algorithm which minimizes the number of variables in each subquery, in this section, we investigate algorithms that keep the number of so-subqueries as low as possible, as well as select so-subqueries with high degree. As we will see in the following, there are settings where the number of queries in the decomposition affects the overall performance of the query evaluation, since large number of subqueries might increase the amount of intermediate results. The following example presents such a case.

28
Example 9. Consider the query $Q$ and the data graph $G$ depicted in Figure 5 and Figure 2, respectively. It is easy to see that there is a single total embedding from $Q$ to $G$. Suppose two decompositions $D^1_Q$ and $D^2_Q$ illustrated in Figure 7 and Figure 8, respectively. As we saw in Example 8, $D^1_Q$ is resulted by min-res algorithm. Counting now the embeddings found for the subqueries of each decomposition over $G$, we have that there are 12 embeddings, in total, from queries in $D^1_Q$ to $G$, while $D^2_Q$ gives 10 embeddings. Hence, we can see that although each subquery in $D^1_Q$ has minimum number of variables, the total number of embeddings is high, due to the large number of subqueries. Box

To construct a decomposition with minimum number of subqueries, we follow an approach based on the naive algorithm. In particular, considering a query $Q$, a simple algorithm, called min-subquery decomposition algorithm, computing a decomposition with minimum number of so-subqueries is given as follows.

**Step 1:** We initially apply the naive algorithm and get a decomposition $D_N$.

**Step 2:** Then, we construct the set $S$ including all the subsets $D$ of $D_N$ such that the queries in $D$ cover all the edges of $Q$; i.e., $\bigcup_{Q_i \in D} Q_i = Q$.

**Step 3:** Finally, we find the sets in $S$ with the minimum number of subqueries and output one of them.

It is easy to see that the min-subquery algorithm returns a decomposition with minimum number of so-subqueries. Note that there might be multiple decompositions that minimize the number of so-subqueries.
Proposition 12. Considering a query $Q$, the min-subquery algorithm results a decomposition $D_Q$ of $Q$ such that each query in $D_Q$ is an so-query and $D_Q$ has the minimum number of so-subqueries, among all the decompositions of $Q$ including so-subqueries.

The proof of the previous proposition follows by the Proposition 10. The last step of the algorithm also ensures that the output has the minimum number of so-subqueries.

Although the min-subquery algorithm returns a minimal decomposition, it applies an exhaustive search over the search space and the resulted decomposition has high redundancy (i.e., there are triples that are included in two subqueries). Especially, if the replicated edges include variables, as we saw in the previous section, the amount of the intermediate results could affect the overall evaluation time. To overcome these issues, we focus on an efficient approach that constructs a decomposition based on the nodes’ degree. In particular, we focus on selecting first the subqueries containing as many triples as possible. In addition, each query triple is included in a unique so-subquery (i.e., redundancy is not allowed in query decomposition). The decomposition algorithm, called max-degree, that follows this approach is given below.

```plaintext
max-degree(Q)
// Q a query.
// max-degree function returns a decomposition R of Q consisting of so-subqueries
begin
R = ∅;
N = N(Q) − L; // The non-literal nodes.
SQ = FindInitMaxSoQueries(N)
Tcovered = ∅;
while SQ ≠ ∅ do
```

Figure 8: Min-subquery decomposition.
\begin{verbatim}
begin
\text{old}SQ, R, \text{TCovered} = \text{ReconstructSQ}(SQ, R, \text{TCovered});
\text{old}SQ = \emptyset;
\text{foreach} (m, S) \in \text{old}SQ do
begin
  S' = S - \text{TCovered};
  if there is a triple with subject \(m\) in \(S'\) then
  \text{SQ} = \text{SQ} \cup \{(m, S')\};
end
end
return \(R\);
end.

\text{FindInitMaxSoQueries}(N)
// \(N\) is a set of the non-literal nodes of a query
begin
\text{SQ} = \emptyset;
\text{foreach} \(n \in N\) do // \(\text{SQ}\) contains all pairs \((n, S(n))\) where \(n \in N\) and
begin
// \(S(n)\) is the maximal so-query with \(n\) as central node.
if there is a triple \((n, p, o)\) \(\in Q\) then
begin
  \(S(n) = \{t | t \in Q\text{ and } t = (n, p, o)\text{ or } t = (s, p', n)\};
\text{SQ} = \text{SQ} \cup \{(n, S(n))\};
end
end
return \(\text{SQ}\);
end.

\text{ReconstructSQ}(\text{SQ}, R, \text{TCovered})
//Find next so-subquery and update both the result \(R\) and the set \(\text{TCovered}\) of covered edges.
begin
//select a maximal so-query in \(\text{SQ}\)
select a \((n, Q') \in \text{SQ}\) such that \(|Q'|\) is maximal among all elements in \(\text{SQ}\).
\(R = R \cup \{Q'\};\) // ... add \(Q'\) to the result and ... \(\text{TCovered} = \text{TCovered} \cup Q'\); // ... add its triples to \(\text{TCovered}\).
\text{old}SQ = \text{SQ} - \{(n, Q')\};
\text{return} \text{old}SQ, R, \text{TCovered};
end.
\end{verbatim}

Intuitively, the max-degree algorithm performs similarly to the min-subquery algorithm. In particular, in each step, it finds an so-query with max degree and removes its edges from the remaining so-stars. The algorithm stops once all the query edges are covered. Note that the max-degree algorithm does not aim to minimize the number of subqueries in the decomposition. An example describing such a case is illustrated in Figure 9. Notice that \(D_1\) and \(D_2\) are two decompositions of \(Q\), where \(D_1\) is the result of the max-degree algorithm and \(D_2\) is the result of min-subquery algorithm. On the other hand, there are queries where the results of both algorithms match. Such an example is illustrated in Figure 8.

As we mentioned above, the max-degree decomposition algorithm does not apply any edge replication (no redundant edges are allowed). Lack of replication might improve the performance of finding the embeddings of a subquery, since less edges are checked in order to find an embedding. However, as we saw in Proposition 9, replicating edges that add more constraints in the subquery might decrease the total number of embeddings of subqueries; hence, it may also decrease the communication cost in a distributed execution. Taking this into account, we present a modification of the max-degree algorithm, called max-degree-with-redundancy, which replicates edges with constants. The max-degree-with-redundancy algorithm is given by replacing the function \text{ReconstructSQ} with \text{ReconstructSQW}.
in max-degree algorithm with the function \textit{Reconstruct}S\textsubscript{Q}Redundancy, which is defined as follows.

\begin{align*}
\text{Reconstruct}S\textsubscript{Q}Redundancy(S_Q, R, T\text{Covered})
& \quad \text{//Find next so-subquery and update both the result } R \text{ and the set } T\text{Covered of covered edges.} \\
& \begin{array}{l}
\text{begin} \\
\quad \text{//select a maximal so-query in } S_Q \\
\quad \text{select a } (n, Q') \in S_Q \text{ such that } |Q'| \text{ is maximal among all elements in } S_Q, \\
\quad \text{//add triples that have already covered and do not add any new variable to the subquery found} \\
\quad Q'' = Q' \cup \{t = (n_1, p, n_2) | t \in (Q - Q'), \text{ and either } n_1 = n \text{ and } n_2 \notin V(Q) \text{ or } n_2 = n \text{ and } n_1 \notin V(Q)\}; \\
\quad R = R \cup \{Q'\}; \quad // ... add } Q'' \text{ to the result and ...} \\
\quad T\text{Covered} = T\text{Covered} \cup Q'; \quad // ... add its triples to } T\text{Covered.} \\
\quad S''_Q = S_Q - \{(n, Q')\}; \\
\quad \text{return } S''_Q, R, T\text{Covered}; \\
\text{end.}
\end{array}
\end{align*}

Comparing the max-degree and max-degree-with-redundancy algorithms, we can easily see that the function \textit{Reconstruct}S\textsubscript{Q}Redundancy used in the max-degree-with-redundancy algorithm to construct each subquery and add it into the resulting set \( R \), constructs each subquery \( Q'' \) from \( Q' \) (which is similar to the query \( Q' \) constructed by \textit{Reconstruct}S\textsubscript{Q} in the max-degree algorithm) and all the query triples in \( Q \) that either start or end to the central node of \( Q' \) and do not include a variable in the other node; i.e., the triples having constants in the non-central node are replicated and reused. In the contrary, the max-
degree algorithm (function $\text{ReconstructS}_Q$) does not replicate any edge during construction of the result.

**Proposition 13.** Considering a query $Q$, the results of both the max-degree and max-degree-with-redundancy algorithms are decompositions of $Q$ that include only so-queries.

**Proof.** By construction all subqueries produced in both algorithms are so-queries. Besides, as all query triples are used the algorithms produce decompositions of $Q$. □

As we mentioned above the main difference between max-degree and max-degree-with-redundancy algorithms is that in the latter, we replicate edges that have constants in the adjacent nodes of the central node. As Proposition 14 shows, the decomposition resulted by the max-degree-with-redundancy algorithm might reduce the number of embeddings exchanged between the last two steps of the evaluation algorithms eval-STARS and QE-with-Redundancy, comparing to the corresponding decomposition resulted by the max-degree algorithm.

**Proposition 14.** Let $Q$ be a query and $D_M$ be a decomposition of $Q$ resulted by the max-degree. Then, there is a decomposition $D_R$ of $Q$ resulted by the max-degree-with-redundancy algorithm such that the following hold:

- there is an one-to-one mapping $\mu$ from $D_M$ to $D_R$ such that $\mu(P_R) = P_M$, if $P_M \subseteq P_R$, where $P_R \in D_R$ and $P_M \in D_M$; and
- for each data graph $G$ and every query $P$ in $D_R$, the number of embeddings of $P$ over $G$ is less than or equal to the number of embeddings of $\mu(P)$ over $G$.

**Proof.** Let $Q$ be a query and $D_M$ be a decomposition of $Q$ resulted by the max-degree. We now need to prove that the max-degree-with-redundancy algorithm can result a decomposition of $Q$ which satisfies the aforementioned properties.

Each subquery $Q'$ in $D_M$ is constructed by the function $\text{ReconstructS}_Q$, and specifically, once it is constructed it is inserted into the resulting set $R$ (which eventually equals $D_M$). Let’s consider that instead of returning the subquery $Q'$ into the result $R$, we return the subquery $Q'' = Q' \cup \{t = (n_1, p, n_2) | t \in (Q - Q') \}$ and either $n_1 = n$ and $n_2 \notin V(Q)$ or $n_2 = n$ and $n_1 \notin V(Q)$, where $n$ is the central node of $Q'$. Since $R$ simply stores the resulting subqueries and is not used in any other step of the algorithm, such a modification does not affect the construction of the subqueries. It is easy to verify that the modified function is given by the function $\text{ReconstructS}_{Q, \text{Redundancy}}$, and the modified algorithm is the max-degree-with-redundancy. Let also $D_R$ be the result of the modified algorithm (i.e., the final set $R$ returned by the algorithm); hence, $D_R$ is the result of the max-degree-with-redundancy.

According to the previous modification, for each query $Q'$ in $D_M$, there is a query $Q''$ in $D_R$, such that $Q'' = Q' \cup \{t = (n_1, p, n_2) | t \in (Q - Q') \}$, and either
n₁ = n and n₂ \notin V(Q) or n₂ = n and n₁ \notin V(Q), where n is the central node of Q’. Hence, there is an one-to-one mapping \( \mu \) from \( D_M \) to \( D_R \) such that \( Q'' = \mu(Q') \) and \( Q' \subseteq Q'' \); which proves the first condition of the proposition.

Furthermore, Proposition 9 and the construction of \( Q'' \) from \( Q' \) imply that for each data graph \( G \), the number of embeddings of \( Q'' \) over \( G \) is less than or equal to the number of embeddings of \( Q' \) over \( G \); which means that the second property is also satisfied. Hence, the decomposition \( D_R \) satisfies both properties of the Proposition 14. □

As we have seen, both max-degree and max-degree-with-redundancy algorithms iterate over the maximal so-subqueries found by the \textit{FindInitMaxSoQueries} function, from the queries of max degree to the queries with min degree, and remove triples covered in the previous iterations. In each iteration, if the query resulted by removing the covered triples is not an so-query, then both the algorithms ignore this query and continue to the next iteration. Let \( Q_R \) be the set of the remaining triples, in such cases. Note that \( Q_R \) will be covered in the next iterations, but the number of iterations might increase due to the triples that do not form an so-query in some iterations. To reduce the number of iterations, we can construct so-queries from the triples in \( Q_R \) by adding to \( Q_R \) a triple that makes it so-query. Such a triple \( t \) is found in the set of covered triples. In addition, to avoid replicating triples that add variables to a query, we remove the triple \( t \) from the so-query that was constructed in the previous iterations. Such an approach might reshape the so-queries constructed in the previous iterations. A decomposition algorithm following this approach is given as follows, and called \textit{max-degree-with-reshaping}.

\textit{max-degree-with-reshaping}(Q)

// \( Q \) a query.
// The function returns a decomposition \( R \) of \( Q \) consisting of so-subqueries of \( Q \\
begin
\text{R} = 0; \\
N = N(Q) - L; // The non-literal nodes. \\
S_Q = \text{FindInitMaxSoQueries}(N); \\
T_{\text{covered}} = 0; \\
while S_Q \neq 0 do \\
begin
select a \((n, Q') \in S_Q \) s.t. \( V(m, Q'') \in (S_Q - \{(n, Q')\}) \) it holds \( NC(Q') \geq NC(Q'') \); \\
// \( NC(Q) \) function returns the number of Not Covered triples in \( Q \) \\
\( S' = Q' - \{t \mid t = (s, p, n) \in Q' \text{ and } t \in T_{\text{covered}} \text{ and } s \in V(Q')\} \); \\
// i.e. remove covered triples whose object is \( n \) that add variable in \( Q' \) \\
\( T = \{t \mid t = (n, p, o) \in Q' \text{ and } t \in T_{\text{covered}} \text{ and } o \in V(Q')\} \);
If \( S' - T \) is so-query then \( S = S' - T \) else
begin
\( S = S' - T \cup \{t'\} \) where \( t' \) is a triple in \( T \); \\
replace \( F \) by \( F - \{t'\} \) in \( R \) where \( F \) is the query in \( R \) containing \( t' \); \\
// notice that \( F - \{t'\} \) is also an so-query 
end
\text{R} = R \cup \{S\}; // ... add \( S \) to the set of subqueries of \( Q \) ... \\
T_{\text{covered}} = T_{\text{covered}} \cup S; // ... add the triples of \( S \) to \( T_{\text{covered}} \) \\
S_Q'' = S_Q - \{(n, Q')\}; \\
S_Q = 0; \\
foreach \((m, S) \in S_Q'' \) do //Reconstruct \( S_Q \) by removing the queries whose triples are ...
begin
// ... completely covered by the so-queries already constructed ... \\
end \\
end
Proposition 15. Considering a query $Q$, the results of max-degree-with-reshaping algorithm are decompositions of $Q$ that include only so-queries.

Proof. By construction all subqueries produced in the algorithm are so-queries. Besides, as all query triples are used the algorithm produces decompositions of $Q$. □

5. Distributed query evaluation algorithms using MapReduce

In this section, we present a set of distributed algorithms implementing the query evaluation approaches presented in Section 4. These algorithms take advantage of the commutation power provided by the MapReduce computation framework.

5.1. The MapReduce framework

MapReduce is a programming model for processing large datasets in a distributed manner. It is based on the definition of two functions, the Map and the Reduce function. The storage layer for the MapReduce framework is a Distributed File System (DFS), such as Hadoop Distributed File System (HDFS), and is characterized by the block/chuck size (the chunk size, which is larger than the chuck size in conventional file systems, is typically 16-128MB in most of DFSs) and the replication of chunks in relatively independent locations to ensure availability. Creating a MapReduce job is straightforward. Briefly, the user defines the functions, which run in each cluster node, in isolation. The map function is applied on one or more files, in DFS, and results $[\text{key}, \text{value}]$ pair. 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 on the pairs) so that all pairs with the same key initialize a single reduce process, called reduce task. The reduce tasks apply the reduce function on the input pairs and result $[\text{key}, \text{value}]$ pairs; which are stored in the DFS. This procedure describes one MapReduce step. Furthermore, the output of the reducer can be set as the input of a map function, which gives to the user the flexibility to create pipelines of multiple steps.

5.2. Overall methodology

Before describing the query MapReduce query evaluation algorithms, we focus on presenting the main patterns used to construct these algorithms. In particular, the algorithms presented in the upcoming sections are based on the following patterns:
1. **Data graph decomposition**: The data graph $G$ is decomposed into a set of data segments according to a given decomposition approach. The data graph segments are stored in the nodes of a cluster of commodity computers.

2. **Storage of the data graph segments**: A generic methodology for storing the data graph segment is used. Such an approach focuses on storing the RDF data into simple text files in N-triple format. Each file also includes the set of border nodes of the segment represented by the triples in the file. Although the segments are stored in simple text files, relational, NoSQL and graph databases could be used, instead, for storing the corresponding segments. Especially the use of multiple relational databases to store the data segments can facilitate the implementation of certain algorithms, but it has a significant impact on the scalability, and fault tolerance.

3. **Query graph decomposition**: The query graph $Q$ is decomposed into a tuple of subqueries $(Q_1, \ldots, Q_n)$, with $n \geq 1$, according to the principles specified in the definition of the corresponding algorithm.

4. **Implementing the query evaluation algorithm**: The proposed query evaluation algorithms are implemented in the MapReduce programming framework. In general, the implementation of each algorithm consists of a preprocessing phase followed by two MapReduce phases (see next section).

5.3. **Preprocessing Phase**

As mentioned earlier, all the query evaluation algorithms presented in the subsequent sections consider a preprocessing phase, where the setting is prepared. In particular, the pre-processing phase accepts a query $Q$ which is posed by the user and decomposes it into a tuple of subqueries $(Q_1, \ldots, Q_n)$, with $n \geq 1$, following the decomposition principles determined by the specific query evaluation algorithm. These subqueries broadcasted or distributed to the mappers of the first MapReduce phase of the query evaluation algorithm.

Preprocessing phase also constructs some auxiliary structures and emits them to the mappers/reducers that implement the algorithm. To define these structures we assume an enumeration $n_1, n_2, \ldots, n_{|N(Q)|}$ of the nodes of the query $Q$, so that $n_1, n_2, \ldots, n_{|B(Q)|}$ are the border nodes of $Q$ and $n_{|B(Q)|+1}, \ldots, n_{|N(Q)|}$ are the non-border nodes of $Q$. We denote by $I$ the function that gives the index of a node in $N(Q)$ with respect to the above enumeration (that is, for every $x \in N(Q)$ it holds $x = n_{I(x)}$). We also denote by $I_{nb}$ the function from $N(Q) - B(Q)$ to $\{1, \ldots, |N(Q) - B(Q)|\}$, with $I_{nb}(x) = I(x) - |B(Q)|$. Similarly, we assume an enumeration $t_1, t_2, \ldots, t_{|Q|}$ of the triples in $Q$. Using the above enumeration functions we now define the concept of query prototype. A query prototype is a triple of tuples of the form:

\[(\text{BorderNodeFlags}, \text{NonBorderNodeFlags}, \text{TripleFlags})\]

where \text{BorderNodeFlags} is a tuple of $|B(Q)|$ items, one item for each border node in $B(Q)$. Similarly, the \text{NonBorderNodeFlags} is a tuple of $|N(Q) - B(Q)|$ items, one for each non border node in $N(Q) - B(Q)$. Finally, the tuple \text{TripleFlag} has $|Q|$ items, one for each triple in $Q$.  

36
Consider now that a query prototype is assigned to each (sub)query $Q_i$. Each item in the tuples of the prototype has either the value ‘+’ to denote the presence of the corresponding border node/non-border node/triple, in $Q_i$, or the value ‘-’ to denote the absence of that node or triple.

We also construct a set\(^2\) called **Missing Border Nodes (MBN)** as follows: $\text{MBN} = \{(b_i, Q_j) | b_i \in B(Q) \text{ and } b_i \not\in N(Q_j)\}$. An element $(b_i, Q_j)$ in MBN denotes that the border node $b_i$ of $Q$ does not appear among the nodes of the subquery $Q_j$ of $Q$.

Based on the idea of query prototype we can represent a partial or total embedding $e$ of a (sub)query in a similar way; i.e. as a triple of tuples of the form $(\text{BorderNodeValues}, \text{NonBorderNodeValues}, \text{TriplesMatched})$. More specifically, $\text{BorderNodeValues}$ stores the images of the border nodes of the query through the (partial)embedding, while $\text{NonBorderNodeValues}$ stores the images of non border nodes of the query. The star symbol (‘*’) is placed in the corresponding node place if no image of that node is defined in $e$. Finally, $\text{TriplesMatched}$ keeps track of the triples of the query that have images on the data graph through the (partial) embedding $e$ (by putting a ‘+’ sign or a ‘-’ sign in the corresponding place of $\text{TriplesMatched}$).

**5.4. QEJPE-algorithm**

In this section we present an implementation of the query evaluation algorithm (QEJPE-algorithm) presented in Subection 4.1. The implementation is based on the MapReduce programming framework. Besides the general assumption on which all algorithms are based, we have the following specific assumptions of the present algorithm:

1. In this algorithm both the decomposition of the data graph $G$ and the query graph $Q$ may be redundant or non-redundant. The query graph $Q$ is decomposed into a tuple of arbitrary subqueries $(Q_1, \ldots, Q_n)$, with $n \geq 1$.

2. The implementation of the algorithm consists of a preprocessing phase followed by two map-reduce phases:

   (a) In the first map-reduce phase the subqueries are applied to each graph segment, in isolation, and intermediate results are computed. More specifically, the mappers of phase 1 compute useful (partial or total) embeddings of the subqueries, by applying each subquery to each specific graph segment. Then the reducers of phase 1 combine (i.e. join) the partial embeddings to compute the total embeddings of each subquery. Notice that the total embeddings of the subqueries are, in general, partial embeddings of the query $Q$ to the graph $G$.

   (b) In the second map-reduce phase, the embeddings of the subqueries are combined appropriately to produce the embedding of the query $Q$ on the graph $G$. More specifically, the mapper of phase 2 fills

\(^2\)In the algorithms presented in this section we represent the MBN set as list.
the missing border nodes in each sub-query embeddings using the values obtained from the embeddings of the other subqueries. Then, reducers of phase 2 construct the embeddings of the query $Q$ by combining compatible embeddings, one for each subquery.

5.4.1. The preprocessing phase

In the preprocessing phase the users’ query $Q$ is decomposed into a tuple of subqueries $(Q_1, \ldots, Q_n)$, with $n \geq 1$ and the auxiliary structures presented in Subsection 5.2 are constructed. Preprocessing phase emits these structures with key the pair $(\text{subqueryID}, \text{SegmentID})$ to the mappers of Phase 1, except of $\text{MBN}$ list, which is emitted directly to the reducers of Phase 1.

Example 10. Consider the query $Q$ appearing in Fig. 4 and assume that the subqueries $Q_1$, $Q_2$, and $Q_3$ are constructed in the preprocessing phase. Assume also that the numbering functions has numbered the nodes and the edges of $Q$ as shown in Fig. 4. Then, it is easy to see that \( B(Q) = \{n_1, n_2, n_3\} \) while \( N(Q) - B(Q) = \{n_4, n_5\} \). Finally, the lists of triples is \( (t_1, t_2, t_3, t_4, t_5) \). It is thus easy to see that the query prototypes for the subqueries $Q_1$, $Q_2$ and $Q_3$ are:

$Q_1$: (\( <+,+>, <+,>- > \))

$Q_2$: (\( <+,+>, <+,>- > \))

$Q_3$: (\( <+,+>, <+,>- > \))

while the list of missing border nodes is $\text{MBN} = [(n_1, Q_2), (n_2, Q_3), (n_3, Q_1)]$.

5.4.2. Phase 1 of the QEIPJE-algorithm

The mapper of phase 1 gets as input a subquery $Q_i$ and a graph segment $G_j$ and evaluates $Q_i$ on $G_j$ obtaining in this way all useful (total and partial) embeddings. These embeddings are emitted to the reducers of Phase 1 with key the subquery ID $Q_i$. The procedure for the Mapper of Phase 1 is given below:

mapper1 $((Q_i, G_j), (G_jData, subqueryInfo))$  
//($Q_i$, $G_j$): $Q_i$ is the ID of a subquery, $G_j$ is the ID of a data segment  
// $G_jData$: the content of the data graph segment $G_j$  
// $SubqueryInfo$: prototypes/border & non-border nodes/triples of $Q$  
begin  
compute $E = \{ e \mid e$ is a useful partial embedding of $Q_i$ in $G_jData\}$;  
for each $e \in E$ do emit $([Q_i, e])$;  
end.

Example 11. (Continued from Example 10). Some embeddings of the subqueries $Q_1$, $Q_2$ and $Q_3$ (see Fig. 4) in the segments $G_1$, $G_2$ and $G_3$ (see Fig. 3) computed by the corresponding mappers and emitted with key the subquery ID, appear below. More specifically, a total embedding evaluated and emitted by the mapper working on $(Q_1, G_1)$ is\(^3\):

\[^3\text{Notice that we can check if an embedding is total or partial by comparing it with the}\]
The Mapper working on \((Q_1, G_2)\) computes and emits the partial embedding:

\[
key = Q_1, \text{ value } = (<\text{Person2}, \text{Article2}, \ast>, <\ast, \ast>, <+>, <+>.)
\]

Among the embeddings obtained and emitted by the Mapper working on \((Q_2, G_2)\) are the (partial) embeddings:

\[
(4) \text{ key } = Q_2, \text{ value } = (<\ast, \text{Article1}, \text{Person1}>, <\ast, \ast>, <+>, <+>.)
\]

\[
(5) \text{ key } = Q_2, \text{ value } = (<\ast, \text{Article2}, \text{Person3}>, <\ast, \ast>, <+>, <+>.)
\]

The Mapper working on \((Q_3, G_2)\) computes and emits the total embeddings:

\[
(6) \text{ key } = Q_3, \text{ value } = (<\text{Person4}, \ast, \text{Person1}>, <\ast, \ast>, <+>, <+>.)
\]

\[
(7) \text{ key } = Q_3, \text{ value } = (<\text{Person2}, \ast, \text{Person3}>, <\ast, \ast>, <+>, <+>.)
\]

The Mapper working on \((Q_1, G_3)\) emits the partial embedding:

\[
(8) \text{ key } = Q_1, \text{ value } = (<\ast, \text{Article2}, \ast>, <\ast, \text{"Title2"}>, <+>, <+>.)
\]

The Mapper working on \((Q_2, G_3)\) emits the partial embeddings:

\[
(9) \text{ key } = Q_2, \text{ value } = (<\ast, \text{Article1}, \ast>, <\ast, \text{Journal1}, \ast>, <+>, <+>.)
\]

\[
(10) \text{ key } = Q_2, \text{ value } = (<\ast, \text{Article2}, \ast>, <\ast, \text{Journal1}, \ast>, <+>, <+>.)
\]

Finally, the mappers working on \((Q_3, G_3)\) and \((Q_3, G_1)\) return no (partial or total) embeddings.

It is important to note that the procedure for the \textit{mapper1} does not determine a specific method for the computation of the useful (partial) embeddings of the subqueries. This means that, every algorithm that can compute all partial embeddings can be used in a specific implementation of the \textit{mapper1}. Moreover, \textit{mapper1} is independent of the way the data graph is stored.

A \textit{Reducer of Phase 1} receives all useful (partial) embeddings of a subquery \(Q_i\) whose ID is the key of the reducer, in all graph segments \(G_1, \ldots, G_m\) of \(G\). A reducer: (a) computes all total embeddings of \(Q_i\) in \(G\) and emits them to the mappers of Phase 2 with key the subquery ID, and (b) it finds all border node values from the total embeddings of \(Q_i\) that are missing from the total embeddings of other subqueries and emits them with the appropriate subquery IDs as keys. The reducer is defined as follows\(^4\):

\begin{verbatim}
reducer1(Q_i, values)
// Q_i: a subquery ID.
// values: contains the list of the embeddings for Q_i and the MBN list
begin
    collect in a list \(F_i\) the total embeddings of \(Q_i\) appearing in values or obtained by joining compatible partial embeddings in values;
    if \(F_i\) is empty then EXIT; // there is no solution for the subquery \(Q_i\)
end
\end{verbatim}

\(^4\)In the presentation of the procedures the following abbreviations are used: \textit{bnv} stands for \textit{BorderNodeValues}, \textit{nbnv} for \textit{NonBorderNodeValues}, and \textit{tm} for \textit{TriplesMatched}.

---

\(39\)
and thus for the original query $Q$

extract the MBN list from values;

foreach embedding $e = (bnv, nbnv, tm)$ in $F_i$ do
begin
emit($[Q_i, (bnv, nbnv)]$); // emits total embedding with key the subquery ID $Q_i$
for $i = 1$ to $|bnv|$ do
if ($bnv[i] != \ast$) then
for each ($n_i, Q_j$) in MBN do
emit($[Q_j, (n_i, bnv[i])]$);
end
end
end.

Example 12. (Continued from Example 11). Among the total embeddings of $Q_1$ that constructs and emits reducer with key $Q_1$ are:

(1)\(\Rightarrow\) key = $Q_1$, value = ($\langle$Person4,Article1,\ast, $\langle$*,Title1\rangle$)

(3)+(8)\(\Rightarrow\) key = $Q_1$, value = ($\langle$Person2,Article2,\ast, $\langle$*,Title2\rangle$)

Taking into account the contents of the MBN list: $MBN = [(n_1, Q_2), (n_2, Q_3), (n_3, Q_1)]$ the reducer also emits the following missing border node values:

key = $Q_2$, value = (1,Person2) key = $Q_2$, value = (1,Person4)
key = $Q_3$, value = (2,Article1) key = $Q_3$, value = (2,Article2), ...

The Reducer for key $Q_2$ constructs and emits the total embeddings for $Q_2$:

(4)+(9)\(\Rightarrow\) key = $Q_2$, value = ($\langle$*,Article1,Person1>, $\langle$Journal1,\ast$)$)

(5)+(10)\(\Rightarrow\) key = $Q_2$, value = ($\langle$*,Article2,Person3>, $\langle$Journal1,\ast$)$)

and the following values for missing border nodes:

key = $Q_3$, value = (2,Article1) key = $Q_3$, value = (2,Article2)
key = $Q_1$, value = (3,Person1) key = $Q_1$, value = (3,Person3), ...

Reducer for key $Q_3$ emits:

(6)\(\Rightarrow\) key = $Q_3$, value = ($\langle$Person4,\ast,Person1>, $\langle$*,\ast$)$)

(7)\(\Rightarrow\) key = $Q_3$, value = ($\langle$Person2,\ast,Person3>, $\langle$*,\ast$)$)
key = $Q_2$, value = (1,Person2) key = $Q_2$, value = (1,Person4)
key = $Q_1$, value = (3,Person1) key = $Q_1$, value = (3,Person3)

5.4.3. Phase 2 of the QEJPE-algorithm

Each Mapper in Phase2 manipulates the embeddings of a specific subquery. It fills in their missing border node values using values from the embeddings of other subqueries that have been emitted by the reducer1 based in the information in MBN list and emits the resulted embeddings to the reducers of Phase 2 (the key is the tuple of the border node values). The mapper of Phase 2 is given below:

mapper2($Q_i$, values)

// $Q_i$: the ID of a subquery
// values: a list $E$ of the parts $(bnv, nbnv)$ of the total embeddings of $Q_i$ and
// a list $V$ of pairs $(i, v)$, where $v$ is a candidate value for $bnv[i]$
begin
for each embedding $e = (bnv, nbnv)$ in $E$ do
for each instance $bnv'$ of $bnv$ using the values in $V$ do

40
emit([bnv', (Qi, nbnv)]);

end.

Example 13. (Continued from Example 12). Mapper with key Q1 receives:

\( E = [\langle \text{Person4, Article1, *}, \langle *, "Title1" \rangle \rangle, \langle \text{Person2, Article2, *}, \langle *, "Title2" \rangle \rangle, \ldots ] \)

\( V = [(3, \text{Person1}), (3, \text{Person3}), \ldots] \)

This mapper produces instances of the border node tuples in \( E \) by replacing the "*" in the 3rd place with a value in \( V \). Among the key-value pairs obtained and emitted in this way are:

- key = (Person4, Article1, Person1), value = (Q1, \langle *, "Title1" \rangle)
- key = (Person2, Article2, Person3), value = (Q1, \langle *, "Title2" \rangle)

The input of the Mapper with key Q2 is:

\( E = [\langle *, Article1, Person1 \rangle, \langle *, "Journal1,"*\rangle, \langle *, Article2, Person3 \rangle, \langle Journal1,"*\rangle, \ldots] \)

\( V = [(1, \text{Person2}), (1, \text{Person4}), \ldots] \)

Some of the instances that this mapper produces and emits are:

- key = (Person4, Article1, Person1), value = (Q2, \langle Journal1, *\rangle)
- key = (Person2, Article2, Person3), value = (Q2, \langle Journal1, *\rangle)

The input of the Mapper with key Q3 is:

\( E = [\langle *, *, Person1 \rangle, \langle *, *, "Journal1,"*\rangle, \langle *, *, Person3 \rangle, \langle *, *, "Journal1,"*\rangle] \)

\( V = [(2, \text{Article1}), (2, \text{Article2})] \)

Some key-value pairs produced and emitted (as above) by this mapper are:

- key = (Person4, Article1, Person1), value = (Q3, \langle *, *\rangle)
- key = (Person2, Article2, Person3), value = (Q3, \langle *, *\rangle)

In each reducer of phase 2, the embeddings (one for each subquery in \((Q_1, \ldots, Q_n)\)) are joined¹ to construct the final answers of \( Q \). The reducer of phase 2 is given below:

\[
\text{reducer2(key, values)}
\]

// key: a tuple of border node values
// values: pairs of the form \((Q_i, \text{partial embedding for non-border nodes})\)

begin

for each join of compatible embeddings

obtained by using one embedding for each subquery do

Emit the result produced by this join;

end.

Example 14. (Continued from Example 13). The Reducer with key (Person4, Article1, Person1) receives the list:

\[ [(Q1, \langle *, "Title1" \rangle), (Q2, \langle Journal1,"*\rangle), (Q3, \langle *, *\rangle)] \]

Combining (i.e. joining) these embeddings the reducer returns the answer:

\[ \langle \text{Person4, Article1, Person1, Journal1, "Title1"} \rangle \]

The reducer with key (Person2, Article2, Person3) receives the following list:

\[ [(Q1, \langle *, "Title2" \rangle), (Q2, \langle Journal1, *\rangle), (Q3, \langle *, *\rangle)] \]

¹Notice that the joined embeddings are, by construction, compatible.
which joins giving the answer:
\[ \langle \text{Person2,Article2,Person3,Journal1, } \text{"Title2"} \rangle \]

Notice that no other reducer returns solution (as they do not receive embeddings for all subqueries). This can be verified by considering all possible embeddings of all subqueries, which do not appear, for space reasons, in the examples of this subsection.

5.4.4. Discussion
QEJPE-algorithm computes the answers to the given query \( Q \) correctly, independently of a) the data graph partitioning, b) the way the graph segments are stored, c) the query graph decomposition, and d) the algorithm used for calculating intermediate (partial) results.

Some improvements to the proposed algorithm are as follows:

1. Note that, in order to obtain all total embeddings of a subquery \( Q_i \) in \( \text{reducer1} \) it suffices to combine partial embeddings obtained from different data graph segments. However, such provenance information is not emitted from \( \text{mapper1} \) in its present form. It is, however, easy to adapt the QEJPE-algorithm, so as the \( \text{mapper1} \) emits this information to \( \text{reducer1} \) and the \( \text{reducer1} \) takes it into account to construct more efficiently total embeddings of the subqueries.

2. Notice that, as we can see in Example 12, several instances of \( \text{reducer1} \) may emit the same values, either embeddings or missing node values, to the \( \text{mapper2} \). This is due to the fact that the same embedding or the same candidate missing node value may be found and emitted by several reducers. Thus, a specific instance of \( \text{mapper2} \) may receive multiple times the same value which may lead in the construction of the same embedding several times. A possible optimization is to eliminate redundant values from the lists \( E \) of embeddings and the list \( V \) of candidate values for missing nodes that an instance of a \( \text{mapper2} \) receives before the computation of embeddings that will be emitted to \( \text{reducer2} \).

5.5. eval-STARS algorithm
In this section we present a MapReduce based implementation of the eval-STARS query evaluation algorithm presented in Subsection 4.2. The algorithm is based on similar assumptions on which the QEJPE-algorithm is based. The main difference is that in eval-STARS algorithm, a queries \( Q \) posed by the user is decomposed into a tuple of queries \((Q_1, \ldots, Q_n)\), with \( n \geq 1 \), of a specific form called generalized star queries. The query decomposition may be redundant or non-redundant.

The implementation of the algorithm consists of a preprocessing phase followed by two map-reduce phases: The first map-reduce phase takes advantage of the generalized star form of the sub-queries and focuses on evaluating the generalized 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.

42
In the preprocessing phase the users’ query $Q$ is decomposed into a tuple of
generalized star subqueries, with $n \geq 1$ and the auxiliary structures presented
in Subsection 5.2 are constructed. Preprocessing phase emits these structures
with key the pair $(subqueryID, SegmentID)$ to the mappers of Phase 1.

Example 15. Consider the query graph appearing in the left part of Fig. 5
which is decomposed into three generalized star subqueries $Q_1$, $Q_2$, and $Q_3$ appearing in the right part of Fig. 5. The border nodes are $B(Q) = \{n_1, n_2, n_3\}$, while the non-border nodes are $N(Q) - B(Q) = \{n_4, n_5, n_6, n_7\}$.

The query prototypes are the following:

- $Q_1$: $(<+,+,+>, <+,+,+>, <+,+,+,+,+>)$
- $Q_2$: $(<+,+,+,+,+,+,+,+,+,+>)$
- $Q_3$: $(<+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,+,
the mapper are directly emitted to the mappers of Phase 2 (rather than to the reducers of Phase 1). Similarly, the values of missing border nodes are emitted to the mappers of Phase 2.

**mapper1**((Q, G1), (GjData, B(GjData), subqueryInfo, MBN))
// (Q, G1): Q1 is the ID of a subquery; G1 is the ID of a data segment
// GjData: the content of the data graph segment Gj
// B(GjData): the set of border nodes of Gj
// SubqueryInfo: prototypes/border & non-border nodes/triples of Q
// MBN: the list of missing border nodes
begin
Let c1 = C(Q1);
% Part 1
foreach triple t = (c1, p, o) in Q1 do
begin
compute E = {e | e is an embedding of t in GjData and e(c1) ∈ B(GjData) };
foreach embedding e in E do
emit([(Q1, e(c1)), (o, e(o))]);
end
foreach triple t = (s, p, c1) in Q1 do
begin
compute E = {e | e is an embedding of t in GjData and e(c1) ∈ (B(GjData) ∪ L) };
for each embedding e in E do
emit([(Q1, e(c1)), (s, e(s))]);
end
% Part 2
compute E = {e | e is a embedding of Q1 in GjData and e(c1) /∈ (B(GjData) ∪ L) };
for each embedding e = (bn, nbn) in E do
begin
emitToSecondPhase([(Q1, (bnv, nbnv))]; // i.e. to the mapper of phase 2
for k = 1 to |bnv| do
if (bnv[k] != "") then
for each (n_k, Qj) in MBN do
emitToSecondPhase([(Qj, (n_k, bnv[k]))];
end
end.
end.

**Example 16.** (Continued from Example 15). In this example, we assume that the query graph Q and its generalized star subqueries are those appearing in Fig. 5, while the data graph G and the graph segments obtained by decomposing G are those appearing in Fig. 3. Below, we see the application of **mapper1** on the pairs of subqueries and graph segments:

Applying **mapper1** on (Q1, G1) results in emission (see Part 1 of the procedure for **mapper1**) of the following (key, value) pairs to the **reducer1**:  
key = (Q1, Article1), value = (n1, Person4) (embedding of t1)  
key = (Q1, Article1), value = (n6, "Title1") (embedding of t7) 
No key value pairs are emitted to Phase 2 (see Part 2 of the procedure for **mapper1**).

44
Applying mapper1 on \((Q_2, G_1)\) results in emission (see Part 1) of the following key, value pair to the reducer1:

\[
\text{key} = (Q_2, \text{Article}1), \text{value} = (n_1, \text{Person}4) \text{ (embedding of } t_2) \\
\]

Besides, the following key, value pairs are emitted directly to the mapper2 (Mapper of Phase 2) (see Part 2):

\[
\begin{align*}
&\text{key} = Q_2, \text{value} = (<\text{Person}4, \text{Article}3, \text{Journal}2>, <*, *, "2008">) \\
&\text{key} = Q_1, \text{value} = (n_2, \text{Article}3) \\
&\text{key} = Q_3, \text{value} = (n_3, \text{Journal}2) \\
\end{align*}
\]

Notice that the last two emissions are conducted by the MBN list which, as we have seen in Example 15, is \(\text{MBN} = [\langle n_2, Q_1 \rangle, \langle n_3, Q_3 \rangle]\).

Applying mapper1 on \((Q_3, G_1)\) results in emission (see Part 1) of the following (key, value) pairs to reducer1:

\[
\begin{align*}
&\text{key} = (Q_3, \text{Person}4), \text{value} = (n_2, \text{Article}1) \text{ (embedding of } t_4) \\
&\text{key} = (Q_3, \text{Person}4), \text{value} = (n_2, \text{Article}3) \text{ (embedding of } t_4) \\
\end{align*}
\]

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 reducer1:

\[
\begin{align*}
&\text{key} = (Q_1, \text{Article}1), \text{value} = (n_1, \text{Person}1) \text{ (embedding of } t_1) \\
&\text{key} = (Q_1, \text{Article}1), \text{value} = (n_1, \text{Person}2) \text{ (embedding of } t_1) \\
\end{align*}
\]

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:

\[
\begin{align*}
&\text{key} = (Q_2, \text{Article}1), \text{value} = (n_1, \text{Person}1) \text{ (embedding of } t_2) \\
&\text{key} = (Q_2, \text{Article}1), \text{value} = (n_1, \text{Person}2) \text{ (embedding of } t_2) \\
&\text{key} = (Q_2, \text{Article}2), \text{value} = (n_1, \text{Person}2) \text{ (embedding of } t_2) \\
&\text{key} = (Q_2, \text{Article}2), \text{value} = (n_1, \text{Person}3) \text{ (embedding of } t_2) \\
\end{align*}
\]

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 reducer1 (see Part 1). However, the following (key, value) pairs are emitted (see Part 2) to mapper2:

\[
\begin{align*}
&\text{key} = Q_3, \text{value} = (<\text{Person}4, \text{Article}1, *>, <*, *, *, "2008">) \\
&\text{key} = Q_3, \text{value} = (<\text{Person}2, \text{Article}2, *>, <*, *, *, *>) \\
&\text{key} = Q_1, \text{value} = (n_2, \text{Article}1) \\
&\text{key} = Q_1, \text{value} = (n_2, \text{Article}2) \\
\end{align*}
\]

Applying mapper1 on \((Q_1, G_3)\) results in emission of the following (key, value) pair to reducer1:

\[
\begin{align*}
&\text{key} = (Q_1, \text{Article}1), \text{value} = (n_3, \text{Journal}1) \text{ (t8)} \\
\end{align*}
\]

No key value pairs are emitted to Phase 2 (see Part 2).

Applying mapper1 on \((Q_2, G_3)\) results in emission of the following (key, value) pair to reducer1:

\[
\begin{align*}
&\text{key} = (Q_2, \text{Article}1), \text{value} = (n_3, \text{Journal}1) \text{ (t5)} \\
&\text{key} = (Q_2, \text{Article}2), \text{value} = (n_3, \text{Journal}1) \text{ (t5)} \\
&\text{key} = (Q_2, \text{Article}2), \text{value} = (n_7, "2008") \text{ (t6)} \\
\end{align*}
\]

No key value pairs are emitted to Phase 2.

Applying mapper1 on \((Q_3, G_3)\) results in no emission of any (key, value) pair.

\[\square\]
Concerning the Reducer of Phase 1 For each key \((Q_i, v)\) the corresponding reducer computes all the embeddings of \(Q_i\) that map the 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 value 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 \(Q_i\). Then, for every \(j = 1, \ldots, m\), the reducer constructs a set \(L[k_j]\) of all possible values for node \(n_{k_j}\). More specifically, for each element \((x_1, x_2, \ldots, x_m)\) of the cartesian product \(L[k_1] \times L[k_2] \times \ldots \times L[k_m]\), it constructs an embedding \(e = (bnv, nbnv)\) of \(Q_i\) in \(G\), such that \(e(c_i) = v\) and \(e(n_{k_j}) = x_j\) and emits \((Q_i, (bnv, nbnv))\) (see Subsection 5.3 for the representation of an embedding). Moreover, if at least one embedding of \(Q_i\) has been found, Reducer 1 emits the values of missing border nodes.

**reducer1**((\(Q_i, v\), values))
// \(Q_i\): a subquery ID
// \(v\): the value of the central node of \(Q_i\)
// values: contains (i) a list of pairs \((x, u)\), with \(x \neq C(Q_i)\) and \(u\) is a candidate image of \(x\) and (ii) the MBN list.

```
begin
% Part 1
allNonEmpty = true;
foreach non-central node \(x\) in \(Q_i\) do
begin
  \(L[I(x)] = \{u \mid (x, u) \in \text{values}\}\);
  if \(L[I(x)] = \emptyset\) then allNonEmpty = false;
end
% Part 2
if allNonEmpty = true then // i.e. there are values for all non-central nodes of \(Q_i\)
begin
  create an embedding with undefined values;
  \((bnv, nbnv) = ((*, \ldots, *), (*, \ldots, *))\);
  \(c_i = C(Q_i)\);
  \(L[I(c_i)] = \{v\}\);
  if \(c_i\) is a border node then
    \(bnv[I(c_i)] = v\);
  else \(nbnv[I_{nb}(c_i)] = v\);
  \(E = \{(bnv, nbnv)\}\);
  for each non-central node \(x\) in \(Q_i\) do
    begin
      \(E' = \emptyset\);
      foreach \(e\) in \(E[I(x)]\) do
        begin
          create a copy \(e' = (bnv', nbnv')\) of \(e\);
          if \(x\) is a border node then
            \(bnv'[I(x)] = u\);
          else \(nbnv'[I_{nb}(x)] = u\);
          insert \((bnv', nbnv')\) in \(E'\);
        end
    end
end
```

46
Example 17. (Continued from Example 16).
The reducer with key \((Q_1, \text{Article1})\) receives the following list of values:
\([(n_1, \text{Person4}), (n_6, \text{"Title1"}), (n_1, \text{Person1}), (n_1, \text{Person2}), (n_3, \text{Journal1})]\).
Notice that, as we can conclude from the sub-query prototypes appearing in Example 15, the border nodes of \(Q\) is \(B(Q) = \{n_1, n_2, n_3\}\), while the non-border are \(N(Q) - B(Q) = \{n_4, n_5, n_6, n_7\}\). Besides, from Fig. 5, we see that the central node of \(Q_1\) is \(n_4\) while its non-central nodes are \(n_1, n_3\) and \(n_6\). Finally, the MBN list is \(MBN = [(n_2, Q_1), (n_3, Q_3)]\). Taking into account the above, the reducer with key \((Q_1, \text{Article1})\), concludes by applying Part 1 of the procedure that it has received values for all non-central nodes of \(Q_1\). More specifically, \(\text{reducer1}\) constructs the following lists:
- \(L[1] = [\text{Person1}, \text{Person2}, \text{Person4}]\)
- \(L[3] = [\text{Journal1}]\)
- \(L[6] = [\text{"Title1"}]\)
which contain the values for the non-central nodes \(n_1, n_3\) and \(n_6\) respectively. Combining these values, as well as the value \(\text{Article1}\) of the central node \(n_4\), \(\text{reducer1}\) in Part 2 constructs and emits the following \((key, value)\) pairs (that represent embeddings of \(Q_1\)):
- \(key = Q_1, value = (<\text{Person1}, *, \text{Journal1}>, <\text{Article1}, *, \text{Title1}, *>))\)
- \(key = Q_1, value = (<\text{Person2}, *, \text{Journal1}>, <\text{Article1}, *, \text{Title1}, *>))\)
- \(key = Q_1, value = (<\text{Person4}, *, \text{Journal1}>, <\text{Article1}, *, \text{Title1}, *>))\)

Besides, \(\text{reducer1}\), based on the MBN list, emits the following:
\(Q_3, (n_3, \text{Journal1})\)

The reducer with key \((Q_2, \text{Article1})\) receives the following list of values:
\([(n_1, \text{Person4}), (n_1, \text{Person1}) (n_1, \text{Person2}), (n_3, \text{Journal1})]\).
Based on this values it constructs the following lists (corresponding to the values of the non-central nodes \(n_1, n_3\) and \(n_7\) of the subquery \(Q_2\)):
- \(L[1] = [\text{Person1}, \text{Person2}, \text{Person4}]\)
- \(L[3] = [\text{Journal1}]\)
- \(L[7] = []\)
From the above we see that the list for the non-central node \(n_7\) is empty. Thus, these values cannot be used to construct a valid embedding for the query \(Q_2\). Therefore, nothing is emitted to the next phase from this reducer.

The reducer with key \((Q_2, \text{Article2})\) receives the following list of values:
\([(n_1, \text{Person2}), (n_1, \text{Person3}), (n_3, \text{Journal1}), (n_7, \text{"2008"})]\).
It constructs the lists:
- \(L[1] = [\text{Person2}, \text{Person3}]\)
Combining these values, as well as the value Article2 of the central node n2, Part 2 of reduc1 constructs and emits the following (key, value) pairs (that represent embeddings of Q_3):

- key = Q2, value = (\langle Person2, Article2, Journal1 \rangle, \langle *, *, *, "2008" \rangle)
- key = Q2, value = (\langle Person3, Article2, Journal1 \rangle, \langle *, *, *, "2008" \rangle)

Besides, reduc1, based on the MBN list, emits the following:

- key = Q1, value = (n2, Article2)
- key = Q3, value = (n3, Journal1)

The reducer with key (Q_3, Person4) receives the following list of values:

- [(n2, Article1), (n2, Article3)].

Based on this values it constructs the following lists (corresponding to the values of the non-central nodes n_1 and n_2 of the subquery Q_3):

- L[1] = []
- L[2] = [Article1, Article3]

From the above we see that the list for the non-central node n_1 is empty. Thus, these values cannot be used to construct a valid embedding for the query Q_3. Therefore, nothing is emitted to the next phase from this reducer.

**5.5.2. Phase 2 of the algorithm**

Phase 2 of the algorithm is similar to the Phase 2 of the QEJPE-algorithm presented in Subsection 5.4.3. The input of each mapper of Phase 2, consist of all the embeddings of a specific subquery Q_i. Besides, for each border node that does not occur in Q_i, mapper gets as input, the values assigned to this node by the embeddings of the other queries. These values are sent by the mappers and reducers of Phase 1 based on the MBN list. Based on its input, the mappers of Phase 2 fills in their missing border node values using the corresponding input values, and emits the resulted embeddings to the reducers of Phase 2 using as key the tuple of the border node values. This means 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 (bnv, nbnv) of the total embeddings of Q_i
// and a set V of pairs (n_k, v), where v is a candidate value for bnv[k]
begin
  foreach embedding e = (bnv, nbnv) in E do
    foreach instance bnv' of bnv using the values in V do
      emit([bnv', (Q_i, nbnv)]);
  end.
end.

**Example 18. (Continued from Example 17).** The mapper that works for the subquery Q_1 (i.e. the key is Q_1), gets a list of values that contain the embeddings of Q_1 in G:

- (\langle Person1, *, Journal1 \rangle, \langle Article1, *, Title1, * \rangle)
- (\langle Person2, *, Journal1 \rangle, \langle Article1, *, Title1, * \rangle)
The mapper sends the following pairs of embeddings to the reducer of Phase 2, by completing the missing node values:

```
key = (<Person1, Article1, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person1, Article2, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person1, Article3, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person2, Article1, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person2, Article2, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person2, Article3, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person4, Article1, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person4, Article2, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person4, Article3, Journal1>,), value = (Q1, Article1, *, Title1, *)
key = (<Person4, Article1, Journal2>,), value = (Q1, Article1, *, Title1, *)
```

The mapper that works for the subquery $Q_2$ (i.e., the key is $Q_2$), receives a list of values that contain the embeddings of $Q_2$ in $G$:

```
(<Person4, Article3, Journal2>, *, *, *, "2008")
(<Person2, Article2, Journal1>, *, *, *, "2008")
(<Person3, Article2, Journal1>, *, *, *, "2008")
```

Notice that $Q_2$ has no missing border nodes. The mapper emits the following pairs to the reducers of Phase 2:

```
key = (<Person4, Article3, Journal2>,), value = (Q2, *, *, *, "2008")
key = (<Person2, Article2, Journal1>,), value = (Q2, *, *, *, "2008")
key = (<Person3, Article2, Journal1>,), value = (Q2, *, *, *, "2008")
```

The mapper that works for the subquery $Q_3$ (i.e., the key is $Q_3$), receives a list of values that contain the embeddings of $Q_3$ in $G$:

```
(<Person4, Article1, *, Journal1>, *, *, *, *)
(<Person2, Article2, *, Journal1>, *, *, *, *)
```

and the values of missing border nodes:

```
(n3, Journal1), (n3, Journal2)
```

The mapper emits the following pairs to the reducer of Phase 2, by completing the missing node values:

```
key = (<Person4, Article1, Journal1>,), value = (Q3, *, *, *, *)
key = (<Person4, Article1, Journal2>,), value = (Q3, *, *, *, *)
key = (<Person2, Article2, Journal1>,), value = (Q3, *, *, *, *)
key = (<Person2, Article2, Journal2>,), value = (Q3, *, *, *, *)
```

Concerning the reducers 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 border 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 border node values
// values: pairs of the form (Qi, partial embedding for non-border nodes)
begin
```
**foreach** join obtained by using one embedding for each subquery **do**

Emit the result produced by this join;

**end.**

**Example 19.** (Continued from Example 18).

The reducer with key \(<\text{Person2}, \text{Article2}, \text{Journal1}>\) receives the following list of values:

\[
[(Q1, <\text{Article1}, *, \text{Title1}, *>),
(Q2, <*, *, "2008">),
(Q3, <*, \text{Person3}, *, *>)]
\]

and constructs the unique embedding of \(Q\) in \(G\):

\[
(<\text{Person2}, \text{Article2}, \text{Journal1}>, <\text{Article1}, \text{Person3}, \text{Title1}, "2008">)
\]

The remaining 11 reducers do not return any answer (they don’t receive values for at least one subquery).

### 5.5.3. Discussion

Due to the specific form in which the user query \(Q\) is decomposed, namely the generalized star queries, the eval-STARS algorithm can compute embeddings more efficiently than QEJPE-algorithm computes partial embeddings of the subqueries of \(Q\).

Notice also that **mapper1** computes and emits directly to **mapper2** total embeddings of the subqueries that map their central nodes to non-border nodes of the data graph segment. This is also an advantage of the eval-STARS algorithm compared with the QEJPE-algorithm.

### 5.6. QE-with-Redundancy algorithm

In this section we present an implementation of the QE-with-Redundancy query evaluation algorithm presented in Subsection 4.3 based on the MapReduce programming framework. Recall that, for the implementation of the algorithm we assume a star-oriented decomposition (s-decomposition) of the data graph \(G\) and a (possibly redundant) decomposition of the query \(Q\) posed by the user into a set of subject-object star subqueries \(\{Q_1, \ldots, Q_n\}\), with \(n \geq 1\). The implementation of the algorithm consists of a preprocessing phase followed by one and a half Map-Reduce phase. 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.

#### 5.6.1. The preprocessing phase

In the preprocessing phase the users’ query \(Q\) is decomposed into a set of so-queries \(\{Q_1, \ldots, Q_n\}\), with \(n \geq 1\), and the auxiliary structures presented in Subsection 5.2 are constructed. Preprocessing phase emits the above to the mappers of Phase 1 with key the pair \((\text{subqueryID}, \text{SegmentID})\).
Example 20. To present the QE-with-redundancy algorithm, we will use again the query \( Q \) and its decomposition into three so-queries presented in Fig. 5. The query prototypes, the MBN list and the tuple of common border nodes of appearing in these subqueries are the same as in Example 15.

Concerning the data graph decomposition, to present the algorithm we will use the data graph segments (s-segments) obtained by decomposing the data graph \( G \) as presented in Fig. 6.

5.6.2. Phase 1 of the algorithm

The first phase of the algorithm computes the embeddings of the so-queries \( Q_1, \ldots , Q_n \) in \( G \) locally in each star graph segment \( G_j \) of \( G \).

Each mapper in phase 1 gets as input an s-graph segment \( G_j \), an so-query \( Q_i \), the MBN list, and the tuple \( CB(Q) \) and computes the embeddings of \( Q_i \) into \( G_j \). The embeddings computed are directly emitted to the mappers of Phase 2. Similarly, the values of the nodes in MBN are emitted to the mappers of Phase 2. Notice that the instances of the nodes in \( CB(Q) \) take part in the keys of the (key, value) pairs emitted to the mappers of Phase 2.

\[
\text{mapper1}: (Q_i, G_j), (G_j, \text{Data}, \text{SubqueryInfo}, \text{MBN}, \text{CB}(Q))
\]

\[
\begin{aligned}
&\text{//}(Q_i, G_j) : Q_i/G_j \text{ is the ID of a subquery/data segment} \\
&\text{// } G_j, \text{Data}: the content of the data graph segment G_j \\
&\text{// } \text{SubqueryInfo}: \text{prototypes of the subqueries of } Q \\
&\text{// } \text{MBN}: \text{the list of missing border nodes} \\
&\text{// } \text{CB}(Q) \text{ is the tuple of common border nodes of } Q
\end{aligned}
\]

\[
\begin{aligned}
\text{begin} & \quad \text{compute } E = \{ e \mid e \text{ is an embedding of } Q_i \text{ in } G_j \text{Data} \} \\
& \quad \text{for each embedding } e = (bnv, nbnv) \text{ in } E \ do \\
& \quad \begin{aligned}
& \text{if } (\text{MBN} = \emptyset) \text{ then} \\
& \quad \text{emitToMapper2}((Q_i, e(\text{CB}(Q))), (bnv, nbnv))); \\
& \quad \text{for } k = 1 \text{ to } |bnv| \ do \\
& \quad \begin{aligned}
& \text{if } (bnv[k] \neq '*') \text{ then} \\
& \quad \text{foreach } (n_k, Q_j) \text{ in } \text{MBN} \ do \\
& \quad \begin{aligned}
& \text{emitToMapper2}((Q_j, e(\text{CB}(Q))), (n_k, bnv[k])); \\
& \text{else} \\
& \quad \text{emitToReducer2}([bnv, (Q_i, nbnv)]); \\
& \quad \end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\]

Example 21. (Continued from Example 20). This example shows the results obtained by the application of \text{mapper1} on the pairs \((Q_i, G_j)\), where \(Q_i\) is an so-query and \(G_j\) is a graph segment.

The following three embeddings of \(Q_1\) into \(G_1\) are computed by the algorithm:

\[
\begin{aligned}
e_1 &= (<\text{Person1}, *, \text{Journal1}>, <\text{Article1}, *, \text{Title1}, *> ) \\
e_2 &= (<\text{Person2}, *, \text{Journal1}>, <\text{Article1}, *, \text{Title1}, *> ) \\
e_3 &= (<\text{Person4}, *, \text{Journal1}>, <\text{Article1}, *, \text{Title1}, *> )
\end{aligned}
\]

For \(e_1\) the algorithm emits the following (key,value) pair to \text{mapper2}:
Besides, based on the MBN list and the CB(Q), which, in the preprocessing phase have been computed to MBN = \{ \langle n_2, Q_1 \rangle, \langle n_3, Q_3 \rangle \} and CB(Q) = \{ n_1 \}, mapper1 also emits to mapper2 the following key value pair:

key = (Q_3, Person1), value = (n_3, Journal1)

Similarly, mapper1 also emits the following (key, value) pairs based on the embeddings e_2 and e_3:

key = (Q_1, Person2), value = (\langle Person2, *, Journal1, * \rangle, \langle Article1, *, Title1, * \rangle)
key = (Q_3, Person2), value = (n_3, Journal1)
key = (Q_1, Person4), value = (\langle Person4, *, Journal1, * \rangle, \langle Article1, *, Title1, * \rangle)
key = (Q_3, Person4), value = (n_3, Journal1)

Concerning query Q_1 there are no embeddings in segments G_2 and G_3. Thus nothing is emitted by the corresponding mappers.

The following embedding of Q_2 into G_1 is computed (among others) by the algorithm:

\( e'_{1} = (\langle Person4, Article3, Journal2, *, *, *, "2008"\rangle) \)

For \( e'_{1} \) the algorithm emits the following (key, value) pair to mapper2:

key = (Q_2, Person4), value = (\langle Person4, Article3, Journal2, *, *, *, "2008"\rangle)

Besides, based on the MBN list and the CB(Q), mapper1 also emits to mapper2 the following key value pair:

key = (Q_1, Person4), value = (n_2, Article3)
key = (Q_3, Person4), value = (n_3, Journal2)

Query Q_2 has no embeddings in segment G_2; hence nothing is emitted in this case.

The following two embedding of Q_2 into G_3 are computed by the algorithm:

\( e'_{2} = (\langle Person2, Article2, Journal1, *, *, *, "2008"\rangle) \)
\( e'_{3} = (\langle Person3, Article2, Journal1, *, *, *, "2008"\rangle) \)

As above, based on these embeddings as well as on the content of the MBL list and the CB(Q), mapper1 also emits to mapper2 the following key value pairs:

key = (Q_2, Person2), value = (\langle Person2, Article2, Journal1, *, *, *, "2008"\rangle)
key = (Q_1, Person2), value = (n_2, Article2)
key = (Q_3, Person2), value = (n_3, Journal1)
key = (Q_2, Person3), value = (\langle Person3, Article2, Journal1, *, *, *, "2008"\rangle)
key = (Q_1, Person3), value = (n_2, Article2)
key = (Q_3, Person3), value = (n_3, Journal1)

The following embedding of Q_3 into G_1 is computed by the algorithm:

\( e"_{1} = (\langle Person4, Article1, *, *, Person1, *, *\rangle) \)

For \( e"_{1} \) the algorithm emits the following (key, value) pair to mapper2:

key = (Q_3, Person4), value = (\langle Person4, Article1, *, *, Person1, *, *\rangle)

Besides, based on the MBN list and the CB(Q), mapper1 also emits to mapper2 the following key value pair:

key = (Q_1, Person4), value = (n_2, Article1)

The following embeddings of Q_3 into G_2 are computed by the algorithm:

\( e"_{2} = (\langle Person2, Article2, *, *, Person3, *, *\rangle) \)
\( e"_{3} = (\langle Person2, Article2, *, *, Person3, *, *\rangle) \)
As above, based on these embeddings as well as on the content of the MBL list and the CB(Q), mapper1 also emits to mapper2 the following key value pairs:

key = (Q3, Person4), value = (<Person4, Article1, *>, <*, Person1, *>)
key = (Q1, Person4), value = (n2, Article1)
key = (Q3, Person2), value = (<Person2, Article2, *>, <*, Person3, *>)
key = (Q1, Person2), value = (n2, Article2)

Query Q₃ has no embeddings in segment G₃; hence nothing is emitted by this mapper. □

5.6.3. Phase 2 of the algorithm

Phase 2 of the algorithm is similar to the Phase 2 of the eval-STARS algorithm.

Each mapper in Phase 2 gets as input all the embeddings of a specific subquery Qᵢ which have the same values for the nodes in CB(Q); moreover for each border node that does not occur in Qᵢ it gets as input the values assigned to this node by the embeddings of the other subqueries. It fills in their missing border 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 border node values, which implies that two embeddings are emitted to the same reducer if and only if they are compatible.

mapper2((Qᵢ,e(CB(Qᵢ))), values)
// Qᵢ: the ID of a subquery
// values: a set E of the parts (bvn, nbvn) of the embeddings
// of Qᵢ,e(CB(Qᵢ)) and a set V of pairs (nᵦ, v),
// where v is a candidate value for bvn[k]
begin
  foreach embedding e = (bvn, nbvn) in E do
    foreach ground instance bvn′ of bvn using the values in V do
      emit([bvn′, (Qᵢ, nbvn)]);
  end.

Example 22. (Continued from Example 21). This example shows the application of mapper2.

The mapper applied for the key (Q₁, Person1) gets the value:

(<Person1, *, Journal1>, <Article1, *, Title1, *>)

but it does not get any value for the missing border nodes. As V = ∅ no ground instances of (<Person1, *, Journal1>) can be found. Therefore this mapper does not emit (key, value) pairs to reducer2.

The mapper applied for the key (Q₁, Person2) gets the value:

(<Person2, *, Journal1>, <Article1, *, Title1, *>)

(n2, Article2)

This mapper constructs the instance <Person2, Article2, Journal1> of bvn and emits the following (key, value) pair to reducer2:

key = <Person2, Article2, Journal1>, value = (Q₁, <Article1, *, Title1, *>)

The mapper applied for the key (Q₁, Person3) gets the values:
As $E = \emptyset$ this mapper emits nothing to $\text{reducer2}$.

The mapper applied for the key $(Q_1, \text{Person}4)$ gets (after eliminating duplicates) the values:
- $(<\text{Person}4, \ast, \text{Journal}1>, <\text{Article}1, \ast, \text{Title}1, \ast>)$
- $(n2, \text{Article}3)$
- $(n2, \text{Article}1)$

This mapper emits the following (key, value) pairs to $\text{reducer2}$:
- $key = <\text{Person}4, \text{Article}3, \text{Journal}1>, value = (Q1, <\text{Article}1, \ast, \text{Title}1, \ast>)$
- $key = <\text{Person}4, \text{Article}1, \text{Journal}1>, value = (Q1, <\text{Article}1, \ast, \text{Title}1, \ast>)$

The mapper applied for the key $(Q_2, \text{Person}2)$ gets the value:
- $(<\text{Person}2, \text{Article}2, \text{Journal}1>, <\ast, \ast, \ast, 2008>)$

This mapper emits the following (key, value) pair to $\text{reducer2}$:
- $key = <\text{Person}2, \text{Article}2, \text{Journal}1>, value = (Q2, <\ast, \ast, \ast, 2008>)$

The mapper applied for the key $(Q_2, \text{Person}3)$ gets the value:
- $(<\text{Person}3, \text{Article}2, \text{Journal}1>, <\ast, \ast, \ast, 2008>)$

This mapper emits the following (key, value) pair to $\text{reducer2}$:
- $key = <\text{Person}3, \text{Article}2, \text{Journal}1>, value = (Q2, <\ast, \ast, \ast, 2008>)$

The mapper applied for the key $(Q_3, \text{Person}1)$ gets the value:
- $(n3, \text{Journal}1)$

As $E = \emptyset$ this mapper emits nothing to $\text{reducer2}$.

The mapper applied for the key $(Q_3, \text{Person}2)$ gets the value:
- $(<\text{Person}2, \text{Article}2, \ast>, <\ast, \text{Person}3, \ast, \ast>)$
- $(n3, \text{Journal}1)$

This mapper emits the following (key, value) pair to $\text{reducer2}$:
- $key = <\text{Person}2, \text{Article}2, \text{Journal}1>, value = (Q3, <\ast, \text{Person}3, \ast, \ast>)$

The mapper applied for the key $(Q_3, \text{Person}3)$ gets the value:
- $(n3, \text{Journal}1)$

As $E = \emptyset$ this mapper emits nothing to $\text{reducer2}$.

The mapper applied for the key $(Q_3, \text{Person}4)$ gets the values:
- $(<\text{Person}4, \text{Article}1, \ast>, <\ast, \text{Person}1, \ast, \ast>)$
- $(n3, \text{Journal}1)$
- $(n3, \text{Journal}2)$

This mapper emits the following (key, value) pairs to $\text{reducer2}$:
- $key = <\text{Person}4, \text{Article}1, \text{Journal}1>, value = (Q3, <\ast, \text{Person}1, \ast, \ast>)$
- $key = <\text{Person}4, \text{Article}1, \text{Journal}2>, value = (Q3, <\ast, \text{Person}1, \ast, \ast>)$

Concerning the $\text{Reducer of Phase 2}$, each reducer gets as input embeddings for each subquery that are compatible as the key for the reducer is a tuple of values for all border nodes of the query $Q$. The embeddings (one for each subquery $Q_1, \ldots, Q_n$) are joined to construct the final answers of $Q$: 54
reducer2(key, values)
// key: a tuple of values for the border nodes of Q
// values: pairs of the form (Qi, partial embeddings of non-border nodes of Qi)
begin
    foreach join obtained by using one embedding for each subquery do
        Emit the result produced by this join;
    end.
end.

Example 23. (Continued from Example 22). The reducer with key: <Person2, Article2, Journal1> receives the list:
[(Q1, <Article1, *, Title1, *>)
, (Q2, <*, *, *, 2008>)
, (Q3, <*, Person3, *, *>)]
As this list contains an embedding for each subquery, we join them and obtain the following embedding of the query Q:
(<Person2, Article2, Journal1>,
, <Article1, Person3, Title1, 2008>)
This embedding corresponds to the answer:
(?P1, ?A, ?J, ?P2, ?T) = (Person2, Article2, Journal1, Person3, Title1)
Note that, the remaining reducers do not return any answer (they don’t receive values for at least one subquery).

5.6.4. Discussion
QE-with-Redundancy algorithm has several advantages compared with QEJPE-algorithm and eval-STARS algorithm. Notice that QE-with-Redundancy algorithm is implemented using one and a half Map-Reduce phases while QEJPE-algorithm and eval-STARS algorithm are implemented using two Map-Reduce phases. Another advantage of QE-with-Redundancy algorithm is that, due to the replication of the data triples in the decomposition of the data graph, and the special form of subqueries in which the user query Q is decomposed, namely subject-object star queries, all the answers to a subject-object star queries can be obtained from a single data segment.

On the other hand, due to the replication of the data triples in the decomposition of the data graph, multiple occurrences of the same embedding as well as multiple instances of member of MBL list may be produced and emitted in Phase 1 of the algorithm.

6. Experimental results
In this section, we present a set of experiments performed over a cluster of 10 virtual machines, and analyze the outcomes. Each cluster node has the following characteristics: Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz (8 Cores) with 16GB RAM, 60GB HD, Ubuntu 16.04 LTS, 64-bit Operating System. We used Apache Hadoop v3.1 with HDFS (1 NameNode, 1 Secondary NameNode, 10 DataNodes each one 30GB) and YARN (1 ResourceManager, 10 NodeManagers). The 10 virtual machines were connected through external IP addresses.

To perform the experiments we used four different datasets (D1, D2, D3, D4) in N-Triples format from the Waterloo SPARQL Diversity Test Suite (WatDiv) [17] to evaluate the algorithms proposed in this paper. The number of triples
of each dataset, as well as the scale factors used to generate the datasets, are illustrated in Table 1.

| Dataset/Query | Scale Factor | Number of triples | Number of files |
|---------------|--------------|-------------------|-----------------|
| D1            | 25           | 2,731,510         | 7               |
| D2            | 50           | 5,486,199         | 13              |
| D3            | 100          | 10,979,566        | 25              |
| D4            | 200          | 21,961,070        | 49              |

Table 1: Description of the Datasets

The data graph was partitioned using three approaches, random edge partition, vertex partition, and METIS. In particular, the random edge partition was implemented by randomly adding each edge into a file such that each file had approximately 450,000 triples. We also stored information about the border nodes in each file. The vertex partition essentially describes the implementation of s-decomposition approach defined in Subsection 4.3. The last partitioning approach used is METIS [39], in order to minimize the number of border nodes in each file. Note that the random edge and METIS partitioning approaches were used to evaluate queries using the QEJPE-algorithm and eval-STARS algorithms, while the vertex partition was used to evaluate the QE-with-Redundancy algorithm.

In the implementation of each algorithm, we used the library RDFLib\(^6\) to pose the subqueries over data segments in each MapReduce task. In particular, in order to find the partial embeddings in each MapReduce task, we parse the data segment and load it into certain structure using RDFLib. Then, we use the query evaluation mechanism of the library to query the loaded segment and find the corresponding partial embeddings. Although the usage of RDFLib facilitates the evaluation of subqueries and provides an efficient evaluation tool in each task, there is an overhead due to loading of each data segment, which is around 40 seconds for each data segment. Note that the loading time does not include the transfer time of each segment as well as the time that takes each task to be initialized.

We conducted several types of experiments to investigate both the performance of the query evaluation algorithms proposed in this paper and the impact of the query decomposition algorithms on the overall query evaluation. In the following, we initially analyze the scalability of each query evaluation algorithm, in terms of both the size of the dataset and the number of cluster nodes. Then, we analyze how the query evaluation algorithms perform in widely-used pattern types of queries and different partitions of the data graph (the ones mentioned above). Finally, we analyze how the overall performance of query evaluation is affected by the type of query decomposition selected.

\(^6\)RDFLib documentation: [https://rdflib.readthedocs.io/](https://rdflib.readthedocs.io/)
6.1. Scalability

In this section, we investigate the scalability of the QEJPE-algorithm, eval-STARS and QE-with-Redundancy algorithms. In particular, we conducted a set of experiments to analyze how the query evaluation algorithms perform in terms of both the size of the input dataset and the number of compute nodes in the cluster.

Initially, we selected three queries of different types from the WatDiv Benchmark and evaluated them using each of the algorithms over each of the D2-D4 datasets. The queries selected are illustrated in the Table 2, along with the number of subqueries generated per algorithm. For each query, the type of the query and the number of the resulting tuples for each dataset are included in the table, as well. Table 3 summarizes the execution time of each query per evaluation algorithm and dataset, where $L$, $S$ and $F$ represent the Linear, Star and Snowflake queries selected, respectively. Figures 10a, 10b and 10c graphically show the execution time per dataset and evaluation algorithm, for each query. Figure 10d illustrates the average execution time for each dataset and each algorithm. Looking at the experimental results, we can see that although the amount of the data in each dataset is doubled (i.e., $D_3$ and $D_4$ have around 100% more triples than $D_2$ and $D_3$, respectively), the growth rate of the execution time remains less than 30%, in average; which shows that each algorithm scales well in terms of the size of the dataset.

Table 2: Description of the queries evaluated over datasets of different sizes

| Query type   | Number of subqueries | QEJPE-algorithm | eval-STARS | QE-with-Redundancy |
|--------------|----------------------|-----------------|------------|--------------------|
| Linear (L2)  | 2                    | 2               | 2          | 2                  |
| Star (S5)    | 2                    | 1               | 1          | 2                  |
| Snowflake (F1)| 3                   | 2               | 2          | 2                  |
| Results D2   | 36                   | 9               | 432        |
| D3           | 28                   | 33              | 64         |
| D4           | 0                    | 10              | 4          |

Table 3: Query evaluation in datasets of different sizes (in seconds)

| Dataset | L | S | F | L | S | F |
|---------|---|---|---|---|---|---|
| $D_2$   | 1079 | 1106 | 1109 | 1086 | 931 | 893 |
| $D_3$   | 1140 | 1207 | 1696 | 1134 | 1135 | 1156 |
| $D_4$   | 1205 | 1358 | 2874 | 1205 | 1270 | 1288 |

To evaluate the scalability in terms of the size of the cluster (i.e., the number of compute nodes), we performed as follows. We evaluated over 3 cluster settings the 3 queries described in Table 2 using each algorithm over the dataset D4. In particular, the first setting had 4 compute nodes (NodeManagers), the second had 7 compute nodes and the last one utilized all the 10 available compute nodes. Then, we executed the evaluation algorithms in each setting. The execution

---

Note that the scalability of each algorithm is limited by the capacity of the cluster resources (i.e., memory, disk space).
Figure 10: Query evaluation in terms of the size of dataset
times are summarized into the Table 4. Figures 11a, 11b and 11c illustrate the execution time in terms of the size of the cluster per algorithm for each type of query. As we can see, the algorithms scale well in terms of the number of compute nodes; i.e., the execution time is decreasing by increasing the number of compute nodes.

|          | QEJPE-algorithm | eval-STARS | QE-with-Redundancy |
|----------|-----------------|------------|--------------------|
|          | L               | S          | F                  | L               | S          | F                  | L               | S          | F                  |
| 4 Nodes  | 1929            | 2214       | -1970              | 1914            | 2073        | 1957               | 1537            | 1993       |
| 7 Nodes  | 1421            | 1738       | 3967               | 1417            | 1437        | 1323               | 969             | 1334       |
| 10 Nodes | 1265            | 1358       | 2874               | 1205            | 1288        | 1103               | 749             | 1102       |

Table 4: Query evaluation in D4 dataset for different compute nodes size (in seconds)

6.2. Comparison of query evaluation algorithms

In this section, we present the outcomes of the experiments performed in order to compare the performance of the query evaluation algorithms proposed in this work; i.e., the QEJPE-algorithm, eval-STARS and QE-with-Redundancy algorithms. The evaluation performed by applying all the algorithms for a variety of queries over the dataset D4 described in the previous sections. We used multiple queries from WatDiv Benchmark, from all the proposed query
types (Linear, Star, Snowflake, and Complex). Table 5 summarizes the queries used in this experiment, along with corresponding characteristics of each query; e.g., number of triples, number of variables, number of resulting tuples over the dataset D4, and the number of subqueries generated by the query decomposition.

The QEJPE-algorithm and eval-STARS algorithms were also tested over two data partitioning approaches, random partitioning and METIS, while the QE-with-Redundancy algorithm was only evaluated over vertex-partitioned data (due to the requirements of the algorithm). The execution time for each query is included in the Table 6, where the execution time is given in minutes, followed by seconds (i.e., Minutes:Seconds). The average execution time for each algorithm and each query type, per data partitioning approach, is illustrated in Table 7 and graphically presented in Figure 12. Note that evaluating the majority of Snowflake and Complex queries using QEJPE-algorithm the cluster reached the memory limits (14.5GB for all YARN containers on a node) and did not manage to provide any result.

As we can see in the experimental results, QEJPE is more efficient for Linear queries than Star queries. In addition, METIS outperforms Random partition for both Star and Linear queries.

Table 5: Description of Watdiv queries

| Parameters | Linear | Star | Snowflake | Complex |
|------------|--------|------|-----------|---------|
| Triples | L2 | L4 | S3 | S5 | F1 | F4 | C3 |
| Variables | 2 | 4 | 4 | 4 | 5 | 5 | 5 |
| Results | 432 | 109 | 677 | 64 | 4 | 73 | 763924 |
| Subqueries QEJPE-algorithm | 2 | 1 | 2 | 2 | 3 | 5 | 3 |
| Subqueries eval-STARS and QE-with-Redundancy | 2 | 1 | 1 | 1 | 2 | 2 | 1 |
| Subqueries QE-with-Redundancy | 2 | 1 | 1 | 2 | 2 | 1 |

Table 6: Execution time per query

| Query | QEJPE METIS | Random | eval-STARS METIS | Random | QE-with-Redundancy |
|-------|-------------|--------|------------------|--------|-------------------|
| L2    | 21:06       | 21:05  | 21:11            | 20:55  | 18:23             |
| L4    | 19:56       | 21:06  | 21:09            | 21:03  | 12:43             |
| S3    | 21:47       | 34:59  | 18:51            | 21:18  | 12:37             |
| S5    | 20:43       | 22:38  | 18:14            | 21:10  | 13:12             |
| F1    | -           | -      | 22:22            | 21:28  | 18:22             |
| F4    | -           | -      | 21:22            | 21:38  | 18:23             |
| C3    | -           | -      | 30:09            | 25:30  | 17:48             |

Table 7: Average execution time per query type (seconds)

| Query Type | Data Partitioning | Linear | Star | Snowflake | Complex |
|------------|-------------------|--------|------|-----------|---------|
| QEJPE-algorithm | METIS | 1231,0 | 1275,0 | - | - |
| | Random | - | 1265,5 | 1728,5 | - | - |
| eval-STARS | METIS | 1270,0 | 1112,5 | 1312,0 | 1809,0 |
| | Random | 1234,0 | 1274,0 | 1293,0 | 1530,0 |
| QE-with-Redundancy | s-decomposition | 333,0 | 753,0 | 1102,5 | 1068,0 |
Queries L2 and L4 from WatDiv Benchmark (Linear query type) evaluated in all the algorithms and the mean execution times in seconds of these queries are presented in Figure 12a and in Table 7. QE-with-Redundancy algorithm performs better than QEJPE-algorithm and eval-STARS algorithms. eval-STARS algorithm perform better than QEJPE-algorithm using both METIS and Random partition.

Queries S3 and S5 were used to evaluate star type queries. QE-with-Redundancy algorithm performs better than QEJPE-algorithm and eval-STARS algorithms while eval-STARS perform better than QEJPE-algorithm. Both eval-STARS and QEJPE-algorithm perform better for METIS partition than random partition.

In case of Snowflake queries, queries F1 and F4 executed. Experimental results prove that QEJPE-algorithm is not efficient for this type of queries. QE-with-Redundancy algorithm performs again better results from eval-STARS algorithm. eval-STARS algorithm performed almost the same results for random and METIS partition.

Similar behavior with Snowflake queries had the Complex type queries. C3 query executed and QEJPE-algorithm was not efficient, QE-with-Redundancy algorithm performs better results than eval-STARS algorithm. In this type of query, eval-STARS algorithm performed better using random partition rather than METIS partition.

6.3. Query Decomposition Algorithms Evaluation

In this section, we experimentally analyze how the selection of the query decomposition algorithm can affect the overall query evaluation performance. We focus on the three main query decomposition algorithms proposed in Subsection 4.4; i.e., min-res, max-degree, and max-degree-with-reshaping. To perform this experiment, we decomposed multiple queries using the aforementioned decomposition algorithms and evaluate them using the QE-with-Redundancy algorithm and over a single dataset.

In particular, we initially used a query template (i.e., query graph structure) over the Watdiv data model and generated six different queries by setting variables and constants to the nodes. The queries Q1-Q6 that were constructed are depicted in Figure 13, where the white-colored nodes represent the variables and dark-colored nodes represent constants. We also constructed an additional complex query Q7, over the Watdiv data model, asking for certain edges of the data graph multiple times. We then decomposed the queries Q1-Q6 using different decomposition algorithms and evaluated them using the QE-with-Redundancy algorithm and the dataset D4. For Q7 query, the smaller dataset D1 was used to overcome memory limitation due to the large number of results. The execution time for each query and each decomposition algorithm is illustrated in Table 8, along with the number of subqueries resulted by each decomposition algorithm and the number of resulting tuples. The execution time per query and algorithm is graphically presented in Figure 14.

Analyzing the execution time of the queries per decomposition algorithm (Table 8 and Figure 14), we can easily see that for queries Q1 and Q2, the
Figure 12: Comparison of query evaluation algorithms for a variety of query types
decompositions of the max-degree and max-degree-with-reshaping perform better than the ones given by min-res. Queries Q4-Q6 perform similarly, for all the three algorithms. This can be explained by the fact that the queries Q1 and Q2 give decompositions with more subqueries using min-res than the de-
compositions given by the two other algorithms, as well as the results are few. For queries Q4-Q6 the performance of min-res algorithm is improved compared with the max-degree and max-degree-with-reshaping algorithms. Query Q3 is an exception since the decomposition resulted by all the three algorithms is the same.

Comparing now the execution time of the max-degree and the max-degree-with-reshaping, these algorithms resulted similar decompositions. Hence, as we can see in Table 8 and Figure 14, their execution time for the majority of the queries is very close.

Looking however the execution time of the query Q7, the decomposition resulted by min-res outperforms the decompositions given by max-degree and max-degree-with-reshaping. To analyze this result in more detail, we can easily see that since the node ?x1 is high degree, gives a subquery with multiple variables in the decompositions given by max-degree and max-degree-with-reshaping. In addition, the variables ?x2, ?x3, and ?x4 in both subqueries map to the same data nodes and increase significantly the number of intermediate results (comparing with the number of the corresponding data edges mapped by these variables in all the embeddings). On the other hand, min-res handles such a case better, since it does not allow subqueries having more than 2 variables to be generated.

7. Conclusions

In this paper, we presented a set of distributed query evaluation algorithms that are independent of the storage and data distribution approaches. These
algorithms could also be implemented in various distributed processing frameworks. We also presented a set of query decomposition approaches and analysed their advantages and disadvantages. Evaluating the proposed algorithms, we showed that each problem instance (data and query graph) might benefit from different decomposition algorithm and/or evaluation approach.

As future work, we aim to investigate the proper methods for storing data in order to further improve our algorithms. Investigation of the usage of certain NoSQL databases with the appropriate indices is also considered, in order to optimize the query plans used to combine the results of the generalized star subqueries in the last approach presented. Furthermore, we aim to analyze additional query decomposition approaches, focusing on finding an optimal query decomposition for every different setting. An additional topic for further investigation is how our approach could be extended to support query evaluation over dynamic RDF data. Finally, improvements of our algorithms using in-memory processing frameworks, such as Apache Spark and Flink, are also considered for further investigation.

Bibliography

[1] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.
[2] Apache Accumulo. https://accumulo.apache.org/.
[3] Apache Cassandra. http://cassandra.apache.org/.
[4] Apache Flink. https://flink.apache.org/.
[5] Apache Hadoop. https://hadoop.apache.org/.
[6] Apache HBase. https://hbase.apache.org/.
[7] Apache Impala. https://impala.apache.org/.
[8] Apache Spark. https://spark.apache.org/.
[9] MongoDB, NoSQL Document Database. https://www.mongodb.com/.
[10] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. The VLDB Journal, 18(2):385–406, 2009.
[11] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proceedings of the VLDB Endowment, 10(13):2049–2060, 2017.
[12] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using Map-Reduce. In 2013 IEEE 29th International Conference on Data Engineering (ICDE), pages 62–73, April 2013.
[13] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a Map-Reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, Sept 2011.

[14] Foto N. Afrati and Rada Chirkova. *Answering Queries Using Views, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[15] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a Map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT ’10, pages 99–110, New York, NY, USA, 2010. ACM.

[16] G. Agathangelos, G. Troullinou, H. Kondylakis, K. Stefanidis, and D. Plexousakis. RDF query answering using Apache Spark: Review and assessment. In *34th IEEE International Conference on Data Engineering, ICDE Workshops*, pages 54–59. IEEE Computer Society, 2018.

[17] G"uneş Aluç, Olaf Hartig, M. Tamer Özu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble, editors, *The Semantic Web – ISWC 2014*, pages 197–212, Cham, 2014. Springer International Publishing.

[18] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. AMADA: web data repositories in the Amazon cloud. In *21st ACM International Conference on Information and Knowledge Management, CIKM’12, Maui, HI, USA, October 29 - November 02, 2012*, pages 2749–2751, 2012.

[19] Oscar Ceballos, Carlos Alberto Ramírez Restrepo, María Constanza Pabón, Andres M. Castillo, and Oscar Corcho. SPARQL2Flink: Evaluation of SPARQL queries on Apache Flink. *Applied Sciences*, 11(15), 2021.

[20] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, 1977.

[21] Tanvi Chawla, Girdhari Singh, Emmanuel S. Pilli, and M.C. Govil. Storage, partitioning, indexing and retrieval in big RDF frameworks: A survey. *Computer Science Review*, 38:100309, 2020.

[22] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000, 2009.

[23] O. Curé, H. Naacke, M. Baazizi, and B. Amann. HAQWA: a hash-based and query workload aware distributed RDF store. In *Proceedings of the
[24] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on NoSQL stores. *ACM Computing Surveys*, 51(2):40:1–40:43, 2018.

[25] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] Jin-Hang Du, Hao-Fen Wang, Yuan Ni, and Yong Yu. *HadoopRDF: A Scalable Semantic Data Analytical Engine*, pages 633–641. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[27] David C. Faye, Olivier Curé, and Guillaume Blin. A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 15:11–35, 2012.

[28] M. Gergatsoulis, C. Nomikos, E. Kalogeros, and M. Damigos. An algorithm for querying linked data using Map-Reduce. In A. Hameurlain, J. W. Rahayu, and D. Taniar, editors, *Data Management in Cloud, Grid and P2P Systems - 6th International Conference, Globe 2013, Prague, Czech Republic, August 28-29, 2013. Proceedings*, volume 8059 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2013.

[29] François Goasdoué, Zoï Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare: Flat Plans for Massively Parallel RDF Queries. In *International Conference on Data Engineering*, Seoul, South Korea, 2015.

[30] G. Gombos, G. Rácz, and A. Kiss. Spar(k)ql: SPARQL evaluation method on Spark GraphX. In M. Younas, I. Awan, and J. E. Haddad, editors, *4th IEEE International Conference on Future Internet of Things and Cloud Workshops, FiCloud Workshops 2016, Vienna, Austria, August 22-24, 2016*, pages 188–193. IEEE Computer Society, 2016.

[31] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. SPARQLGX: efficient distributed evaluation of SPARQL with Apache Spark. In P. T. Groth et al., editors, *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016. Proceedings, Part II*, volume 9982 of *Lecture Notes in Computer Science*, pages 80–87. Springer, 2016.

[32] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of semantic web databases. *Journal of Computer and System Sciences*, 77(3):520–541, 2011.

[33] Mahmudul Hassan and Srividya K. Bansal. Semantic data querying over NoSQL databases with Apache Spark. In *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*, pages 364–371. IEEE, 2018.
[34] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[35] 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 Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.

[36] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. In *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*, pages 1–10. IEEE Computer Society, 2010.

[37] E. Kalogeros, M. Gergatsoulis, and M. Damigos. Redundancy in linked data partitioning for efficient query evaluation. In I. Awan, M. Younas, and M. Mecella, editors, *3rd International Conference on Future Internet of Things and Cloud, FiCloud 2015, Rome, Italy, August 24-26, 2015*, pages 497–504. IEEE Computer Society, 2015.

[38] Zoi Kaoudi and Ioana Manolescu. Rdf in the clouds: A survey. *The VLDB Journal*, 24(1):67–91, 2015.

[39] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[40] B. Kassaie. SPARQL over GraphX. *CoRR*, abs/1701.03091, 2017.

[41] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. From SPARQL to MapReduce: The journey using a nested triplegroup algebra. *Proceedings of the VLDB Endowment*, 4(12):1426–1429, 2011.

[42] G. Ladwig and A. Harth. CumulusRDF: linked data management on nested key-value stores. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, volume 30, 2011.

[43] Kisung Lee and Ling Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of the VLDB Endowment*, 6(14):1894–1905, September 2013.

[44] R. Mutharaju, S. Sakr, A. Sala, and P. Hitzler. D-SPARQ: distributed, scalable and efficient RDF query engine. In E. Blomqvist and T. Groza, editors, *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*, volume 1035 of *CEUR Workshop Proceedings*, pages 261–264, 2013.

[45] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. SPARQL basic graph pattern processing with iterative MapReduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud, MDAC ’10*, pages 6:1–6:6, New York, NY, USA, 2010. ACM.
[46] H. Naacke, B. Amann, and O. Curé. SPARQL graph pattern processing with Apache Spark. In P. A. Boncz and Josep-Lluís Larriba-Pey, editors, Proceedings of the Fifth International Workshop on Graph Database Management Experiences & Systems, SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017, pages 1:1–1:7. ACM, 2017.

[47] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment, 1(1):647–659, 2008.

[48] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pages 627–640, 2009.

[49] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. The VLDB Journal, 19(1):91–113, February 2010.

[50] C. Nomikos, M. Gergatsoulis, E. Kalogeratos, and M. Damigos. A MapReduce algorithm for querying linked data based on query decomposition into stars. In K. Selçuk Candan et al., editors, Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014, volume 1133 of CEUR Workshop Proceedings, pages 224–231, 2014.

[51] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD ’08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[52] M Tamer Özsu. A survey of RDF data management systems. Frontiers of Computer Science, 10(3):418–432, 2016.

[53] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziaris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In 2013 IEEE International Conference on Big Data, pages 255–263, Oct 2013.

[54] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziaris. H2RDF: Adaptive query processing on RDF data in the cloud. In WWW ’12 Companion: Proceedings of the 21st International Conference on World Wide Web, WWW ’12 Companion, pages 397–400, New York, NY, USA, 2012. ACM.

[55] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In David et. al. Hutchison, editor, The Semantic Web - ISWC 2006, volume 4273, pages 30–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.
[56] François Picalausa and Stijn Vansummeren. What are real SPARQL queries like? In *Proceedings of the International Workshop on Semantic Web Information Management*, pages 1–6, 2011.

[57] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: a scalable RDF triple store for the clouds. In *1st International Workshop on Cloud Intelligence (colocated with VLDB 2012), Cloud-I ’12, Istanbul, Turkey, August 31, 2012*. ACM, 2012.

[58] Kurt Rohloff and Richard E. Schantz. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, DIDC ’11*, pages 35–44, New York, NY, USA, 2011. ACM.

[59] A. Schützle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: graph-parallel querying of RDF with GraphX. In F. Wang et al., editors, *Biomedical Data Management and Graph Online Querying - VLDB 2015 Workshops, Big-O(Q) and DMAH, Waikoloa, HI, USA, August 31 - September 4, 2015, Revised Selected Papers*, volume 9579 of *Lecture Notes in Computer Science*, pages 155–168. Springer, 2016.

[60] Alexander Schützle, Martin Przyjaciel-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading map-side joins over HBase for scalable join processing. In Achille Fokoue, Thorsten Liebig, Eric L. Goodman, Jesse Weaver, Jacopo Urbani, and David Mizell, editors, *Proceedings of the Joint Workshop on Scalable and High-Performance Semantic Web Systems, Boston, USA, November 11, 2012*, volume 943 of *CEUR Workshop Proceedings*, pages 59–74. CEUR-WS.org, 2012.

[61] Alexander Schützle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proceedings of the International Workshop on Semantic Web Information Management, SWIM ’11*, pages 4:1–4:8, New York, NY, USA, 2011. ACM.

[62] Alexander Schützle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. Sempala: Interactive SPARQL query processing on Hadoop. In Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble, editors, *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, volume 8796 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2014.

[63] Alexander Schützle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF Querying with SPARQL on Spark. *Proceedings of the VLDB Endowment*, 9(10):804–815, 2016.

[64] Andy Seaborne and Eric Prud’hommeaux. SPARQL Query Language for RDF. W3C Recommendation, January 2008.
[65] Juan F Sequeda and Daniel P Miranker. Ultrawrap: SPARQL execution on relational data. *Journal of Web Semantics*, 22:19–39, 2013.

[66] Dimitrios-Emmanuel Spanos, Periklis Stavrou, and Nikolas Mitrou. Bringing relational databases into the semantic web: A survey. *Semantic Web*, 3(2):169–209, 2012.

[67] Claus Stadler, Gezim Sejdiu, Damien Graux, and Jens Lehmann. Sparkify: A scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2019.

[68] Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently joining group patterns in sparql queries. In *Extended Semantic Web Conference*, pages 228–242. Springer, 2010.

[69] Xin Wang, Lele Chai, Qiang Xu, Yajun Yang, Jianxin Li, Junhu Wang, and Yumpen Chai. Efficient subgraph matching on large RDF graphs using MapReduce. *Data Science and Engineering*, 4(1):24–43, 2019.

[70] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[71] I. Yaqoob, I. A. T. Hashem, A. Gani, S. Mokhtar, E. Ahmed, N. B. Anuar, and A. V. Vasilakos. Big data: From beginning to future. *International Journal of Information Management*, 36(6):1231–1247, 2016.

[72] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In E. M. Nahum and D. Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*, 2010.