

Querying RDF Data: A Multigraph Based Approach

Vijay INGALALLI
Dino IENCO
Pascal PONCELET

April 27, 2018

Contents

| | |
|--|-----------|
| 0.1. Introduction | 7 |
| 0.2. Related Work | 8 |
| 0.3. Background and Preliminaries | 9 |
| 0.3.1. RDF Data | 9 |
| 0.3.1.1. Data Multigraph Representation | 9 |
| 0.3.2. SPARQL Query | 11 |
| 0.3.2.1. Query Multigraph Representation | 11 |
| 0.3.3. SPARQL Querying by Adopting Multigraph Homomorphism | 12 |
| 0.4. AMBER: A SPARQL Querying Engine | 14 |
| 0.5. Index Construction | 14 |
| 0.5.1. Attribute Index | 14 |
| 0.5.2. Vertex Signature Index | 15 |
| 0.5.3. Vertex Neighbourhood Index | 17 |
| 0.6. Query Matching Procedure | 19 |
| 0.6.1. Vertex Level Processing | 20 |
| 0.6.2. Processing Satellite Vertices | 21 |
| 0.6.3. Arbitrary Query Processing | 22 |
| 0.7. Experimental Analysis | 26 |
| 0.7.1. Experimental Setup | 26 |
| 0.7.2. Workload Generation | 26 |
| 0.7.3. Comparison with RDF Engines | 27 |
| 0.8. Conclusion | 30 |
| 0.9. Acknowledgment: | 31 |
| Bibliography | 33 |

0.1. Introduction

Resource Description Framework (RDF) is a standard for the conceptual description of knowledge. The RDF data is cherished and exploited by various domains such as life sciences, Semantic Web, social network, etc. Further, its integration at Web-scale compels RDF management engines to deal with complex queries in terms of both size and structure. Popular examples are provided by Google, that exploits the so called *knowledge graph* to enhance its search results with semantic information gathered from a wide variety of sources, or by Facebook, that implements the so called *entity graph* to empower its search engine and provide further information extracted, for instance by Wikipedia. Another example is supplied by recent question-answering systems [CAB 12, ZOU 14a] that automatically translate natural language questions in SPARQL queries and successively retrieve answers by considering the available information in the different Linked Open Data sources. In all these examples, complex queries (in terms of size and structure) are generated to ensure the retrieval of all the required information. Since the use of large knowledge bases that are commonly stored as RDF triplets is becoming a common way to ameliorate a wide range of applications, efficient querying of RDF data sources using SPARQL¹ (query language conceived to query RDF data) is becoming crucial for modern information retrieval systems.

With the ever increasing advantage of representing real world data of various domains in RDF format, two vital challenges have been faced by the RDF data management community: firstly, the automatically generated queries cannot be bounded in their structural complexity and size (e.g., the DBPEDIA SPARQL Benchmark [MOR 11] contains some queries having more than 50 triplets [ALU 14a]); secondly, the queries generated by retrieval systems (or by any other applications) need to be efficiently answered in a reasonable amount of time. Modern RDF data management, such as *x-RDF-3X* [NEU 10] and *Virtuoso* [ERL 12], are designed to address the scalability of SPARQL queries but they still have problems to answer big and structurally complex SPARQL queries [ALU 14b].

In order to address these challenges, in this chapter, we discuss about a graph-based RDF querying engine AMBER [ING 16] (Attributed Multigraph Based Engine for RDF querying), that involves two steps: an offline stage where RDF data is transformed into multigraph and indexed, and an online step where an efficient approach to answer SPARQL query is proposed. First of all RDF data is represented as a multigraph where subjects/objects constitute vertices and multiple edges (predicates) can appear between the same pair of vertices. Then, new indexing structures are conceived to efficiently access RDF multigraph information. Finally, by representing the SPARQL queries also as multigraphs, the query answering task can be reduced to the

1. <http://www.w3.org/TR/sparql11-overview/>

problem of subgraph homomorphism. To deal with this problem, AMBER employs an efficient approach that exploits structural properties of the multigraph query as well as the indices previously built on the multigraph structure. Experimental evaluation over popular RDF benchmarks show the quality in terms of time performances and robustness of our proposal. In this chapter, we focus only on the `SELECT/WHERE` clause of the SPARQL language², that constitutes the most important operation of any RDF query engines.

0.2. Related Work

In order to efficiently answer SPARQL queries, many stores and APIs inspired by relational model were proposed [ERL 12, BRO 02, NEU 10, CAR 04]. `x-RDF-3X` [NEU 10], inspired by modern RDBMS, represent RDF triples as a big three-attribute table. The RDF query processing is boosted using an exhaustive indexing schema coupled with statistics over the data. Also Virtuoso [ERL 12] heavily exploits RDBMS mechanism in order to answer SPARQL queries. Virtuoso is a column-store based systems that employs sorted multi-column column-wise compressed projections. Also these systems build table indexing using standard B-trees. Jena [CAR 04] supplies API for manipulating RDF graphs. Jena exploits multiple-property tables that permit multiple views of graphs and vertices which can be used simultaneously.

The database community has recently started to investigate RDF stores based on graph data management techniques [DAS 14, ZOU 14b, KIM 15]. The work in [DAS 14] addresses the problem of supporting property graphs as RDF, since majority of the graph databases are based on property graph model. The authors introduce a property graph to RDF transformation scheme and propose three models to address the challenge of representing the key/value properties of property graph edges in RDF. `gStore` [ZOU 14b] applies graph pattern matching techniques using filter-and-refinement strategy to answer SPARQL queries. It employs an indexing schema, named `VS*`-tree, to concisely represent the RDF graph. Once the index is built, it is used to find promising subgraphs that match the query. Finally, exact subgraphs are enumerated in the refinement step. `Turbo_Hom++` [KIM 15] is an adaptation of a state of the art subgraph isomorphism algorithm (`Turbo_ISO` [HAN 13]) to the problem of SPARQL queries. Exploiting the standard graph isomorphism problem, the authors relax the injectivity constraint to handle the graph homomorphism, which is the RDF pattern matching semantics. Unlike the proposed AMBER, `Turbo_Hom++` does not index the RDF graph, while `gStore` concisely represents RDF data through `VS*`-tree.

2. <http://www.w3.org/TR/sparql11-overview/>

0.3. Background and Preliminaries

In this section we provide basic definitions on the interplay between RDF and its multigraph representation. Later, we explain how the task of answering SPARQL queries can be reduced to multigraph homomorphism problem.

0.3.1. RDF Data

As per the W3C standards³, RDF data is represented as a set of triples $\langle S, P, O \rangle$, as shown in Figure 1a, where each triple $\langle s, p, o \rangle$ consists of three components: a *subject*, a *predicate* and an *object*. Further, each component of the RDF triple can be of any two forms; an *IRI* (Internationalized Resource Identifier) or a literal. For brevity, an *IRI* is usually written along with a prefix (e.g., $\langle \text{http://dbpedia.org/resource/isPartOf} \rangle$ is written as 'x:isPartOf'), whereas a literal is always written with double quotes (e.g., "90000"). While a subject s and a predicate p are always an *IRI*, an object o is either an *IRI* or a literal.

RDF data can also be represented as a directed graph where, given a triple $\langle s, p, o \rangle$, the subject s and the object o can be treated as vertices and the predicate p forms a directed edge from s to o , as depicted in Figure 1b. Further, to underline the difference between an *IRI* and a literal, we use standard rectangles and arc for the former while we use beveled corner and edge (no arrows) for the latter.

0.3.1.1. Data Multigraph Representation

Motivated by the graph representation of RDF data (Figure 1b), we take a step further by transforming it to a data multigraph G , as shown in Figure 1c.

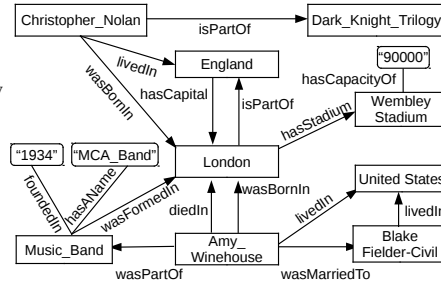
Let us consider an RDF triple $\langle s, p, o \rangle$ from the RDF tripleset $\langle S, P, O \rangle$. Now to transform the RDF tripleset into data multigraph G , we set four protocols: we always treat the subject s as a vertex; a predicate p is always treated as an edge; we treat the object o as a vertex only if it is an *IRI* (e.g., vertex v_2 corresponds to object 'x:London'); when the object is a literal, we combine the object o and the corresponding predicate p to form a tuple $\langle p, o \rangle$ and assign it as a vertex attribute to the subject s (e.g., $\langle \text{'y:hasCapacityOf', "90000"} \rangle$ is assigned to vertex v_4). Every vertex is assigned a null value $\{-\}$ in the vertex attribute set. However, to realize this in the realms of graph management techniques, we maintain three different dictionaries, whose elements are a pair of 'key' and 'value', and a mapping function that links them. The three dictionaries depicted in Table 1 are: a vertex dictionary (Table 1a), an edge-type dictionary (Table 1b) and a vertex attribute dictionary (Table 1c). In all the three dictionaries, an RDF entity represented by a 'key' is mapped to a corresponding

3. <http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>

10 RDF Querying

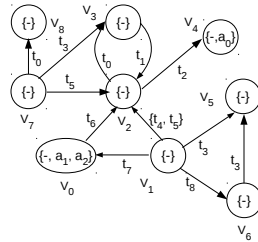
Prefixes: x= http://dbpedia.org/resource/ ; y=http://dbpedia.org/ontology/

| Subject | Predicate | Object |
|-----------------------|-----------------|-----------------------|
| x:London | y:isPartOf | x:England |
| x:England | y:hasCapital | x:London |
| x:Christophar_Nolan | y:wasBornIn | x:London |
| x:Christophar_Nolan | y:LivedIn | x:England |
| x:Christophar_Nolan | y:isPartOf | x:Dark_Knight_Trilogy |
| x:London | y:hasStadium | x:WembleyStadium |
| x:WembleyStadium | y:hasCapacityOf | "90000" |
| x:Amy_Winehouse | y:wasBornIn | x:London |
| x:Amy_Winehouse | y:diedIn | x:London |
| x:Amy_Winehouse | y:wasPartOf | x:Music_Band |
| x:Music_Band | y:hasName | "MCA_Band" |
| x:Music_Band | y:FoundedIn | "1994" |
| x:Music_Band | y:wasFormedIn | X:London |
| x:Amy_Winehouse | y:livedIn | x:United States |
| x:Amy_Winehouse | y:wasMarriedTo | x:Blake Fielder-Civil |
| x:Blake Fielder-Civil | y:livedIn | x:United States |



(a) RDF tripleaset

(b) Graph representation of RDF data



(c) Equivalent multigraph G

Figure 1 – (a) RDF data in n-triple format; (b) graph representation (c) attributed multigraph G

‘value’, which can be a vertex/edge/attribute identifier. Thus by using the mapping functions - \mathcal{M}_v , \mathcal{M}_e , and \mathcal{M}_a for vertex, edge-type and vertex attribute mapping respectively, we obtain a directed, vertex attributed data multigraph G (Figure 1c), which is formally defined as follows.

Definition 1. Directed, Vertex Attributed Multigraph. A directed, vertex attributed multigraph G is defined as a 4-tuple (V, E, L_V, L_E) where V is a set of vertices, $E \subseteq V \times V$ is a set of directed edges with $(v, v') \neq (v', v)$, L_V is a labelling function that assigns a subset of vertex attributes A to the set of vertices V , and L_E is a labelling function that assigns a subset of edge-types T to the edge set E .

To summarise, an RDF tripiaset is transformed into a data multigraph G , whose elements are obtained by using the mapping functions as already discussed. Thus, the set of vertices $V = \{v_0, \dots, v_m\}$ is the set of mapped subject/object *IRI*, and the labelling function L_V assigns a set of vertex attributes $A = \{-, a_0, \dots, a_n\}$ (mapped tuple of predicate and object-literal) to the vertex set V . The set of directed edges E is a set of pair of vertices (v, v') that are linked by a predicate, and the labelling function L_E assigns the set of edge types $T = \{t_0, \dots, t_p\}$ (mapped predicates) to the set of directed edges E . The edge set E maintains the topological structure of the RDF data. Further, mapping of object-literals and the corresponding predicates as a set of vertex attributes, results in a compact representation of the multigraph. As depicted in Figure 1c, all the object-literals and the corresponding predicates are reduced to a set of vertex attributes. For example, the pair $\langle y:\text{hasCapacityOf}, "90000" \rangle$ is mapped to the vertex attribute a_0 ; similarly, $\langle y:\text{wasFoundedIn}, "1994" \rangle$ and $\langle y:\text{hasName}, "MCA_Band" \rangle$ are mapped to attribute a_1 and a_2 respectively.

0.3.2. SPARQL Query

A SPARQL query usually contains a set of triple patterns, much like RDF triples, except that any of the subject, predicate and object may be a variable, whose bindings are to be found in the RDF data⁴. In the current work, we address the SPARQL queries with ‘SELECT/WHERE’ option, where the predicate is always instantiated as an *IRI* (Figure 2a). The SELECT clause identifies the variables to appear in the query results while the WHERE clause provides triple patterns to match against the RDF data.

0.3.2.1. Query Multigraph Representation

In any valid SPARQL query (as in Figure 2a), every triplet has at least one unknown variable $?X$, whose bindings are to be found in the RDF data. It should now be easy to observe that a SPARQL query can be represented in the form of a graph as in Figure 2b, which in turn is transformed into query multigraph Q (as in Figure 2c).

In the query multigraph representation, each unknown variable $?X_i$ is mapped to a vertex u_i that forms the vertex set U component of the query multigraph Q (e.g., $?X_6$ is mapped to u_6). Since a predicate is always instantiated as an *IRI*, we use the edge-type dictionary in Table 1b, to map the predicate to an edge-type identifier $t_i \in T$ (e.g., ‘isMarriedTo’ is mapped as t_8). When an object o_i is a literal, we use the attribute dictionary (Table 1c), to find the attribute identifier a_i for the predicate-object tuple $\langle p_i, o_i \rangle$ (e.g., $\{a_0\}$ forms the attribute for vertex u_4). Further, when a subject or an object is an *IRI*, which is a not a variable, we use the vertex dictionary (1a), to map it to an *IRI*-vertex u_i^{iri} (e.g., ‘x:United_States’ is mapped to u_0^{iri}) and maintain a set of *IRI* vertices R . Since this vertex is not a variable and a real vertex of the query,

4. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>

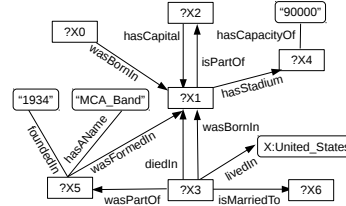
12 RDF Querying

```

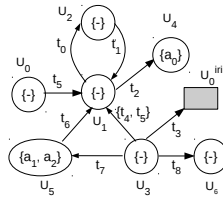
SELECT ?X0 ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 WHERE {
?X0 y:livedIn ?X1 .
?X1 y:isPartOf ?X2 .
?X2 y:hasCapital ?X1 .
?X1 y:hasStadium ?X4 .
?X3 y:wasBornIn ?X1 .
?X3 y:diedIn ?X1 .
?X3 y:isMarriedTo ?X1 .
?X3 y:wasPartOf ?X5 .
?X5 y:wasFormedIn ?X1 .
?X4 y:hasCapacity ?X4 .
?X5 y:hasName "MCA_Band" .
?X5 y:foundedIn "1934" .
?X3 y:livedIn x:United_States . }

```

(a) SPARQL Query



(b) Graph representation of SPARQL



(c) Equivalent Multigraph Q

Figure 2 – (a) SPARQL query representation; (b) graph representation (c) attributed multigraph Q

we portray it differently by a shaded square shaped vertex. When a query vertex u_i does not have any vertex attributes associated with it (e.g., u_0, u_1, u_2, u_3, u_6), a null attribute $\{-\}$ is assigned to it. On the other hand, an *IRI*-vertex $u_i^{iri} \in R$ does not have any attributes. Thus, a SPARQL query is transformed into a query multigraph Q .

In this work, we always use the notation V for the set of vertices of G , and U for the set of vertices of Q . Consequently, a data vertex $v \in V$, and a query vertex $u \in U$. Also, an incoming edge to a vertex is positive (default), and an outgoing edge from a vertex is labelled negative ('-').

0.3.3. SPARQL Querying by Adopting Multigraph Homomorphism

As we recall, the problem of SPARQL querying is addressed by finding the solutions to the unknown variables $?X$, that can be bound with the RDF data entities, so that the relations (predicates) provided in the SPARQL query are respected. In this work, to harness the transformed data multigraph G and the query multigraph Q , we reduce the problem of SPARQL querying to a sub-multigraph homomorphism problem. The RDF data is transformed into data multigraph G and the SPARQL query

| s/o | $\mathcal{M}_v(s/o)$ |
|------------------------|----------------------|
| x:Music_Band | v_0 |
| x:Amy_Winehouse | v_1 |
| x:London | v_2 |
| x:England | v_3 |
| x:WembleyStadium | v_4 |
| x:United States | v_5 |
| x:Blake Fielder-Civil | v_6 |
| x:Christopher_Nolan | v_7 |
| x:Dark_Knight_Triology | v_8 |

(a) Vertex Dictionary

| p | $\mathcal{M}_e(p)$ |
|----------------|--------------------|
| y:isPartOf | t_0 |
| y:hasCapital | t_1 |
| y:hasStadium | t_2 |
| y:livedIn | t_3 |
| y:diedIn | t_4 |
| y:wasBornIn | t_5 |
| y:wasFormedIn | t_6 |
| y:wasPartOf | t_7 |
| y:wasMarriedTo | t_8 |

(b) Edge-type Dictionary

| $\langle p, o \rangle$ | $\mathcal{M}_a(\langle p, o \rangle)$ |
|---|---------------------------------------|
| $\langle y:\text{hasCapacityOf}, "90000" \rangle$ | a_0 |
| $\langle y:\text{wasFoundedIn}, "1994" \rangle$ | a_1 |
| $\langle y:\text{hasName}, "MCA_Band" \rangle$ | a_2 |

(c) Attribute Dictionary

Table 1 – Dictionary look-up tables for vertices, edge-types and vertex attributes

is transformed into query multigraph Q . Let us now recall that finding SPARQL answers in the RDF data is equivalent to finding all the sub-multigraphs of Q in G that are homomorphic. Thus, let us now formally introduce homomorphism for a vertex attributed, directed multigraph.

Definition 2. Sub-multigraph Homomorphism. *Given a query multigraph $Q = (U, E^Q, L_U, L_E^Q)$ and a data multigraph $G = (V, E, L_V, L_E)$, the sub-multigraph homomorphism from Q to G is a surjective function $\psi: U \rightarrow V$ such that:*

- 1) $\forall u \in U, L_U(u) \subseteq L_V(\psi(u))$
- 2) $\forall (u_m, u_n) \in E^Q, \exists (\psi(u_m), \psi(u_n)) \in E$, where (u_m, u_n) is a directed edge, and $L_E^Q(u_m, u_n) \subseteq L_E(\psi(u_m), \psi(u_n))$.

Thus, by finding all the sub-multigraphs in G that are homomorphic to Q , we enumerate all possible homomorphic embeddings of Q in G . These embeddings contain the solution for each of the query vertex that is an unknown variable. Thus, by using the inverse mapping function $\mathcal{M}_v^{-1}(v_i)$ (referring to the vertex dictionary in Table 1a), we find the bindings for the SPARQL query. The decision problem of subgraph homomorphism is NP-complete. This standard subgraph homomorphism problem can be seen as a particular case of sub-multigraph homomorphism, where both the labelling functions L_E and L_E^Q always return the same subset of edge-types for all the edges in both Q and G . Thus the problem of sub-multigraph homomorphism is at least as hard as subgraph homomorphism. Further, the subgraph homomorphism problem is a generic scenario of subgraph isomorphism problem where, the injectivity constraints are slackened [KIM 15].

0.4. AMBER: A SPARQL Querying Engine

The proposed AMBER (Attributed Multigraph Based Engine for RDF querying) contains two different stages: (i) an offline stage during which, RDF data is transformed into multigraph G and then a set of index structures \mathcal{I} is constructed that captures the necessary information contained in G ; (ii) an online stage during which, a SPARQL query is transformed into a multigraph Q , and then by exploiting the sub-graph matching techniques along with the already built index structures \mathcal{I} , the homomorphic matches of Q in G are obtained.

Given a multigraph representation Q of a SPARQL query, AMBER decomposes the query vertices U into a set of core vertices U_c and satellite vertices U_s . Intuitively, a vertex $u \in U$ is a core vertex, if the degree of the vertex is more than one; on the other hand, a vertex u with degree one is a satellite vertex. For example, in Figure 2c, $U_c = \{u_1, u_3, u_5\}$ and $U_s = \{u_0, u_2, u_4, u_6\}$. Once decomposed, we run the sub-multigraph matching procedure on the query structure spanned only by the core vertices. However, during the procedure, we also process the satellite vertices (if available) that are connected to a core vertex that is being processed. For example, while processing the core vertex u_1 , we also process the set of satellite vertices $\{u_0, u_2, u_4\}$ connected to it; whereas, the core vertex u_5 has no satellite vertices to be processed. In this way, as the matching proceeds, the entire structure of the query multigraph Q is processed to find the homomorphic embeddings in G . The set of indexing structures \mathcal{I} is extensively used during the process of sub-multigraph matching. The homomorphic embeddings are finally translated back to the RDF entities using the inverse mapping function \mathcal{M}_v^{-1} as discussed in Section 0.3.

0.5. Index Construction

Given a data multigraph G , we build the following three different indices: (i) an inverted list \mathcal{A} for storing the set of data vertex for each attribute in $a_i \in A$ (ii) a trie index structure \mathcal{S} to store features of all the data vertices V (iii) a set of trie index structures \mathcal{N} to store the neighbourhood information of each data vertex $v \in V$. For brevity of representation, we ensemble all the three index structures into $\mathcal{I} := \{\mathcal{A}, \mathcal{S}, \mathcal{N}\}$. During the query matching procedure (the online step), we access these indexing structures to obtain the candidate solutions for a query vertex u . Formally, for a query vertex u , the *candidate solutions* are a set of data vertices $C_u = \{v | v \in V\}$ obtained by accessing \mathcal{A} or \mathcal{S} or \mathcal{N} , denoted as $C_u^{\mathcal{A}}$, $C_u^{\mathcal{S}}$ and $C_u^{\mathcal{N}}$ respectively.

0.5.1. Attribute Index

The set of vertex attributes is given by $A = \{a_0, \dots, a_n\}$ (Section 0.3), where a data vertex $v \in V$ might have a subset of A assigned to it. We now build the vertex

| Data vertex | Signature | Synopsis | | | | | | | |
|-------------|---|----------|---------|---------|---------|---------|---------|---------|---------|
| | | f_1^+ | f_2^+ | f_3^+ | f_4^+ | f_1^- | f_2^- | f_3^- | f_4^- |
| v_0 | $\{\{-t_6\}, \{t_7\}\}$ | 1 | 1 | -7 | 7 | 1 | 1 | -6 | 6 |
| v_1 | $\{\{-t_3\}, \{-t_7\}, \{-t_8\}, \{-t_4, -t_5\}\}$ | 0 | 0 | 0 | 0 | 2 | 5 | -3 | 8 |
| v_2 | $\{\{-t_0\}, \{t_1\}, \{-t_2\}, \{t_5\}, \{t_6\}, \{t_4, t_5\}\}$ | 2 | 4 | -1 | 6 | 1 | 2 | 0 | 2 |
| v_3 | $\{\{t_0\}, \{t_3\}, \{-t_1\}\}$ | 1 | 2 | 0 | 3 | 1 | 1 | -1 | 1 |
| v_4 | $\{\{t_2\}\}$ | 1 | 1 | -2 | 2 | 0 | 0 | 0 | 0 |
| v_5 | $\{\{t_3\}, \{t_3\}\}$ | 1 | 1 | -3 | 3 | 0 | 0 | 0 | 0 |
| v_6 | $\{\{t_8\}, \{-t_3\}\}$ | 1 | 1 | -8 | 8 | 1 | 1 | -3 | 3 |
| v_7 | $\{\{-t_0\}, \{-t_3\}, \{-t_5\}\}$ | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 5 |
| v_8 | $\{\{t_0\}\}$ | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 2 – Vertex signatures and the corresponding synopses for the vertices in the data multigraph G (Figure 1c)

attribute index \mathcal{A} by creating an inverted list where a particular attribute a_i has the list of all the data vertices in which it appears.

Given a query vertex u with a set of vertex attributes $u.A \subseteq A$, for each attribute $a_i \in u.A$, we access the index structure \mathcal{A} to fetch a set of data vertices that have a_i . Then we find a common set of data vertices that have the entire attribute set $u.A$. For example, considering the query vertex u_5 (Fig. 2c), it has an attribute set $\{a_1, a_2\}$. The candidate solutions for u_5 are obtained by finding all the common data vertices, in \mathcal{A} , between a_1 and a_2 , resulting in $C_{u_5}^A = \{v_0\}$.

0.5.2. Vertex Signature Index

The index \mathcal{S} captures the edge type information from the data vertices. For a lucid understanding of this indexing schema we formally introduce the notion of vertex signature that is defined for a vertex $v \in V$, which encapsulates the edge information associated with it.

Definition 3. Vertex signature. *For a vertex $v \in V$, the vertex signature σ_v is a multiset containing all the directed multi-edges that are incident on v , where a multi-edge between v and a neighbouring vertex v' is represented by a set that corresponds to the edge types. Formally, $\sigma_v = \bigcup_{v' \in N(v)} L_E(v, v')$ where $N(v)$ is the set of neighbourhood vertices of v , and \cup is the union operator for multiset.*

The index \mathcal{S} is constructed by tailoring the information supplied by the vertex signature of each vertex in G . To extract some interesting features, let us observe the vertex signature σ_{v_2} as supplied in Table 2. To begin with, we can represent the vertex signature σ_{v_2} separately for the incoming and outgoing multi-edges as $\sigma_{v_2}^+ = \{\{t_1\}, \{t_5\}, \{t_6\}, \{t_4, t_5\}\}$ and $\sigma_{v_2}^- = \{\{-t_0\}, \{-t_2\}\}$ respectively. Now we observe that $\sigma_{v_2}^+$ has four distinct multi-edges and $\sigma_{v_2}^-$ has two distinct multi-edges. Now, let us

assume that we want to find candidate solutions for a query vertex u . The data vertex v_2 can be a match for u only if the signature of u has at most four incoming ('+') edges and at most two outgoing ('-') edges; else v_2 can not be a match for u . Thus, more such features (e.g., maximum cardinality of a set in the vertex signature) can be proposed to filter out irrelevant candidate vertices. Thus, for each vertex v , we propose to extract a set of features by exploiting the corresponding vertex signature. These features constitute a *synopses*, which is a surrogate representation that approximately captures the vertex signature information.

The synopsis of a vertex v contains a set of features F , whose values are computed from the vertex signature σ_v . In this background, we propose four distinct features: f_1 - the maximum cardinality of a set in the vertex signature; f_2 - the number of unique dimensions in the vertex signature; f_3 - the minimum index value of the edge type; f_4 - the maximum index value of the edge type. For f_3 and f_4 , the index values of edge type are nothing but the position of the sequenced alphabet. These four basic features are replicated separately for outgoing (negative) and incoming (positive) edges, as seen in Table 2. Thus for the vertex v_2 , we obtain $f_1^+ = 2$, $f_2^+ = 4$, $f_3^+ = -1$ and $f_4^+ = 7$ for the incoming edge set and $f_1^- = 1$, $f_2^- = 2$, $f_3^- = 0$ and $f_4^- = 2$ for the outgoing edge set. Synopses for the entire vertex set V for the data multigraph G are depicted in Table 2.

Once the synopses are computed for all data vertices, an R-tree is constructed to store all the synopses. This R-tree constitutes the vertex signature index \mathcal{S} . A synopsis with $|F|$ fields forms a leaf in the R-tree. When a set of possible candidate solutions are to be obtained for a query vertex u , we create a vertex signature σ_u in order to compute the synopsis, and then obtain the possible solutions from the R-tree structure.

The general idea of using an R-tree is as follows. A synopsis F of a data vertex spans an axes-parallel rectangle in an $|F|$ -dimensional space, where the maximum co-ordinates of the rectangle are the values of the synopses fields $(f_1, \dots, f_{|F|})$, and the minimum co-ordinates are the origin of the rectangle (filled with zero values). For example, a data vertex represented by a synopsis with two features $F(v) = [2, 3]$ spans a rectangle in a 2-dimensional space in the interval range $([0, 2], [0, 3])$. Now, if we consider synopses of two query vertices, $F(u_1) = [1, 3]$ and $F(u_2) = [1, 4]$, we observe that the rectangle spanned by $F(u_1)$ is wholly contained in the rectangle spanned by $F(v)$ but $F(u_2)$ is not wholly contained in $F(v)$. Thus, u_1 is a candidate match while u_2 is not.

Lemma 1. *Querying the vertex signature index \mathcal{S} constructed with synopses, guarantees to output at least the entire set of candidate solutions.*

Proof. Consider the field f_1^\pm in the synopses that represents the maximum cardinality of the neighbourhood signature. Let σ_u be the signature of the query vertex u and $\{\sigma_{v_1}, \dots, \sigma_{v_n}\}$ be the set of signatures on the data vertices. By using f_1 we need to show that C_u^S has at least all the valid candidate matches. Since we are

looking for a superset of query vertex signature, and we are checking the condition $f_1^\pm(u) \leq f_1^\pm(v_i)$, where $v_i \in V$, a vertex v_i is pruned if it does not match the inequality criterion since, it can never be an eligible candidate. This analogy can be extended to the entire synopses, since it can be applied disjunctively. ■

Formally, the set of candidate solutions for a query vertex u can be written as $C_u^S = \{v | \forall_{i \in [1, \dots, |F|]} f_i^\pm(u) \leq f_i^\pm(v)\}$, where the constraints are met for all the $|F|$ -dimensions. Since we apply the same inequality constraint to all the fields, we negate the fields that refer to the minimal index value of the edge type (f_3^+ and f_3^-) so that the rectangular containment problem still holds. Further to respect the rectangular containment, we populate the synopses fields with ‘0’ values, in case, the signature does not have either positive or negative edges in it, as seen for v_1, v_3, v_4, v_5 and v_7 .

For example, if we want to compute the possible candidates for a query vertex u_0 in Figure 2c, whose signature is $\sigma_{u_0} = \{-t_5\}$, we compute the synopsis which is [0 0 0 0 1 1 5 5]. Now we look for all those vertices that subsume this synopsis in the R-tree, whose elements are depicted in Table 2, which gives us the candidate solutions $C_{u_0}^S = \{v_1, v_7\}$, thus pruning the rest of the vertices.

The S index helps to prune the vertices that do not respect the edge type constraints. This is crucial since this pruning is performed for the initial query vertex, and hence many candidates are cast away, thereby avoiding unnecessary recursion during the matching procedure. For example, for the initial query vertex u_0 , whose candidate solutions are $\{v_1, v_7\}$, the recursion branch is run only on these two starting vertices instead of the entire vertex set V .

0.5.3. Vertex Neighbourhood Index

The *vertex neighbourhood index* \mathcal{N} captures the topological structure of the data multigraph G . The index \mathcal{N} comprises of 1-neighbourhood trees built for each data vertex $v \in V$. Since G is a directed multigraph, and each vertex $v \in V$ can have both the incoming and outgoing edges, we construct two separate index structures \mathcal{N}^+ and \mathcal{N}^- for incoming and outgoing edges respectively, that constitute the structure \mathcal{N} .

To understand the index structure, let us consider the data vertex v_2 from Figure 1c, shown separately in Figure 3a. For this vertex v_2 , we collect all the neighbourhood information (vertices and multi-edges), and represent this information by a tree structure, built separately for incoming (+) and outgoing (-) edges. Thus, the tree representation of a vertex v contains the neighbourhood vertices and the corresponding multi-edges, as shown in Figure 3b, where the vertices of the tree structure are represented by the edge types.

In order to construct an efficient tree structure, we take inspiration from [TER 06] to propose the structure - Ordered Trie with Inverted List (OTIL). To construct the OTIL index as shown in Figure 3b, we insert each ordered multi-edge that is incident on v at the root of the trie. Consider a data vertex v_i , with a set of n neighbourhood vertices $N(v_i)$. Now, for every pair of incoming edge $(v_i, N^j(v_i))$, where $j \in \{1, \dots, n\}$, there exists a multi-edge $\{t_i, \dots, t_j\}$, which is inserted into the OTIL structure \mathcal{N}^+ . Similarly for every pair of outgoing edge $(N^j(v_i), v_i)$, there exists a multi-edge $\{t_m, \dots, t_n\}$, which is inserted into the OTIL structure \mathcal{N}^- maintaining two OTIL structures that constitute \mathcal{N} . Each multi-edge is ordered (w.r.t. increasing edge type indexes), before inserting into the respective OTIL structure, and the order is universally maintained for all data vertices. Further, for every edge type t_i , we maintain a *list* that contains all the neighbourhood vertices $N^+(v_i)/N^-(v_i)$, that have the edge type t_i incident on them.

To understand the utility of \mathcal{N} , let us consider an illustrative example. Considering the query multigraph Q in Figure 2c, let us assume that we want to find the matches for the query vertices u_1 and u_0 in order. Thus, for the initial vertex u_1 , let us say, we have found the set of candidate solutions which is $\{v_2\}$. Now, to find the candidate solutions for the next query vertex u_0 , it is important to maintain the structure spanned by the query vertices, and this is where the indexing structure \mathcal{N} is accessed. Thus to retain the structure of the query multigraph (in this case, the structure between u_1 and u_0), we have to find the data vertices that are in the neighbourhood of already matched vertex v_2 (a match for vertex u_1), that has the same structure (edge types) between u_1 and u_0 in the query graph. Thus to fetch all the data vertices that have the edge type t_5 , which is directed towards v_2 and hence '+', we access the neighbourhood index trie \mathcal{N}^+ for vertex v_2 , as shown in Figure 3. This gives us a set of candidate solutions $C_{u_0}^{\mathcal{N}} = \{v_1, v_7\}$. It is easy to observe that, by maintaining two separate indexing structures \mathcal{N}^+ and \mathcal{N}^- , for both incoming and outgoing edges, we can reduce the time to fetch the candidate solutions.

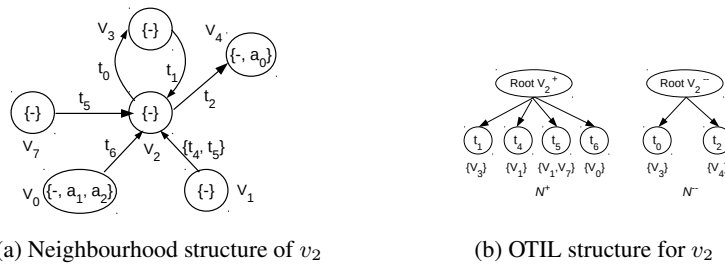
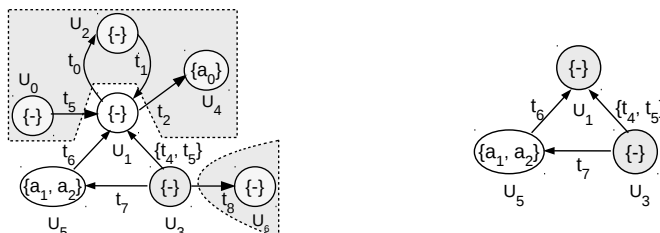


Figure 3 – Building Neighbourhood Index for data vertex v_2



(a) Query graph with satellite vertices (shaded) (b) Query graph spanned by core vertices

Figure 4 – Decomposing the query multigraph into *core* and *satellite* vertices

Thus, in a generic scenario, given an already matched data vertex v , the edge direction ‘+’ or ‘-’, and the set of edge types $T' \subseteq T$, the index \mathcal{N} will find a set of neighbourhood data vertices $\{v' | (v', v) \in E \wedge T' \subseteq L_E(v', v)\}$ if the edge direction is ‘+’ (incoming), while \mathcal{N} returns $\{v' | (v, v') \in E \wedge T' \subseteq L_E(v, v')\}$ if the edge direction is ‘-’ (outgoing).

0.6. Query Matching Procedure

In order to follow the working of the proposed query matching procedure, we formalize the notion of *core* and *satellite* vertices. Given a query graph Q , we decompose the set of query vertices U into a set of *core* vertices U_c and a set of *satellite* vertices U_s . Formally, when the degree of the query graph $\Delta(Q) > 1$, $U_c = \{u | u \in U \wedge deg(u) > 1\}$; however, when $\Delta(Q) = 1$, i.e., when the query graph is either a vertex or a multiedge, we choose one query vertex at random as a *core* vertex, and hence $|U_c| = 1$. The remaining vertices are classified as satellite vertices, whose degree is always 1. Formally, $U_s = \{U \setminus U_c\}$, where for every $u \in U_s$, $deg(u) = 1$. The decomposition for the query multigraph Q is depicted in Figure 4, where the satellite vertices are separated (vertices under the shaded region in Figure 4a), in order to obtain the query graph that is spanned only by the core vertices (Figure 4b). Thus, during query decomposition, satellite vertices are separated from the query graph, leaving a core graph and a set of satellite vertices; the original query structure is preserved by storing the edge information that links each satellite vertex and the corresponding core vertex in the query graph spanned by core vertices.

The proposed AMBER-Algo (Algorithm 3) performs recursive sub-multigraph matching procedure only on the query structure spanned by U_c as seen in Figure 4b. Since the entire set of satellite vertices U_s is connected to the query structure spanned by the core vertices, AMBER-Algo processes the satellite vertices while performing sub-multigraph matching on the set of core vertices. Thus during the recursion, if the current *core* vertex has *satellite* vertices connected to it, the algorithm retrieves

directly a list of possible matching for such *satellite* vertices and it includes them in the current partial solution. Each time the algorithm executes a recursion branch with a solution, the solution not only contains a data vertex match v_c for each query vertex belonging to U_c , but also a set of matched data vertices V_s for each query vertex belonging to U_s . Each time a solution is found, we can generate not only one, but a set of embeddings through the Cartesian product of the matched elements in the solution.

Since finding SPARQL solutions is equivalent to finding homomorphic embeddings of the query multigraph, the homomorphic matching allows different query vertices to be matched with the same data vertices. Recall that there is no injectivity constraint in sub-multigraph homomorphism as opposed to sub-multigraph isomorphism [KIM 15]. Thus during the recursive matching procedure, we do not have to check if the potential data vertex has already been matched with previously matched query vertices. This is an advantage when we are processing satellite vertices: we can find matches for each satellite vertex independently without the necessity to check for a repeated data vertex.

Before getting into the details of the AMBER-Algo, we first explain how a set of candidate solutions is obtained when there is information associated only with the vertices. Then we explain how a set of candidate solutions is obtained when we encounter the satellite vertices.

0.6.1. Vertex Level Processing

To understand the generic query processing, it is necessary to understand the matching process at vertex level. Whenever a query vertex $u \in U$ is being processed, we need to check if u has a set of attributes A associated with it or any *IRI*s are connected to it (recall Section 0.3.2).

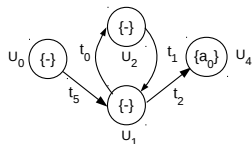
Algorithm 1: PROCESSVERTEX($u, Q, \mathcal{A}, \mathcal{N}$)

```

1 if  $u.A \neq \emptyset$  then
2    $C_u^A = \text{QUERYATTINDEX}(\mathcal{A}, u.A)$ 
3 if  $u.R \neq \emptyset$  then
4    $C_u^I = \bigcap_{u_i^{iri} \in u.R} (\text{QUERYNEIGHINDEX}(\mathcal{N}, L_E^Q(u, u_i^{iri}), u_i^{iri}))$ 
5  $CandAtt_u = C_u^A \cap C_u^I$  /* Find common candidates */
6 return  $CandAtt_u$ 

```

To process an arbitrary query vertex, we propose a procedure PROCESSVERTEX, depicted in Algorithm 1. This algorithm is invoked only when a vertex u has at least, either a set of vertex attributes or any *IRI* associated with it. The PROCESSVERTEX procedure returns a set of data vertices $CandAtt_u$, which are matchable with u ; in case $CandAtt_u$ is empty, then the query vertex u has no matches in V . As seen


 Figure 5 – A star structure in the query multigraph Q

in Lines 1-2, when a query vertex u has a set of vertex attributes i.e., $u.A \neq \emptyset$, we obtain the candidate solutions C_u^A by invoking QUERYATTINDEX procedure, that accesses the index \mathcal{A} as explained in Section 0.5.1. For example, the query vertex u_5 with vertex attributes $\{a_1, a_2\}$, can only be matched with the data vertex v_0 ; thus $C_{u_5}^A = \{v_0\}$.

When a query vertex u has *IRIs* associated with it, i.e., $u.R \neq \emptyset$ (Lines 3-4), we find the candidate solutions C_u^I by invoking the QUERYNEIGHINDEX procedure. As we recall from Section 0.3.2, a vertex u is connected to an *IRI* vertex u_i^{iri} through a multi-edge $L_E^Q(u, u_i^{iri})$. An *IRI* vertex u_i^{iri} always has only one data vertex v , that can match. Thus, the candidate solutions C_u^I are obtained by invoking the QUERYNEIGHINDEX procedure, that fetches all the neighbourhood vertices of v that respect the multi-edge $L_E^Q(u, u_i^{iri})$. The procedure is invoked until all the *IRI* vertices $u.R$ are processed (Line 4). Considering the example in Figure 2c, u_3 is connected to an *IRI*-vertex u_0^{iri} , which has a unique data vertex match v_5 , through the multi-edge $\{-t_3\}$. Using the neighbourhood index \mathcal{N} , we look for the neighbourhood vertices of v_5 , that have the multi-edge $\{-t_3\}$, which gives us the candidate solutions $C_{u_3}^I = \{v_1\}$. Finally in Line 5, the merge operator $\bar{\cap}$ returns a set of common candidates $CandAtt_u$, only if $u.A \neq \emptyset$ and $u.R \neq \emptyset$. Otherwise, C_u^A or C_u^I are returned as $CandAtt_u$.

0.6.2. Processing Satellite Vertices

In this section, we provide insights on processing a set of satellite vertices $U_{sat} \subseteq U_s$ that are connected to a core vertex $u_c \in U_c$. This scenario results in a structure that appears frequently in SPARQL queries called star structure [GUB 14, HUA 11]. A typical star structure depicted in Figure 5, has a core vertex $u_c = u_1$, and a set of satellite vertices $U_{sat} = \{u_0, u_2, u_4\}$ connected to the core vertex. For each candidate solution of the core vertex u_1 , we process u_0, u_2, u_4 independently of each other, since there is no structural connectivity (edges) among them, although they are only structurally connected to the core vertex u_1 .

Lemma 2. *For a given star structure in a query graph, each satellite vertex can be independently processed if a candidate solution is provided for the core vertex u_c .*

Proof. Consider a core vertex u_c that is connected to a set of satellite vertices $U_{sat} = \{u_0, \dots, u_s\}$, through a set of edge-types $T' = \{t_0, \dots, t_s\}$. Let us assume v_c is a candidate solution for the core vertex u_c , and we want to find candidate solutions for $u_i \in U_{sat}$ and $u_j \in U_{sat}$, where $i \neq j$. Now, the candidate solutions for u_i and u_j can

be obtained by fetching the neighbourhoods of already matched vertex v_c that respect the edge-type $t_i \in T'$ and $t_j \in T'$ respectively. Since two satellite vertices u_i and u_j are never connected to each other, the candidate solutions of u_i are independent of that of u_j . This analogy applies to all the satellite vertices. ■

Algorithm 2: MATCHSATVERTICES($\mathcal{A}, \mathcal{N}, Q, U_{sat}, v_c$)

```

1 SET:  $M_{sat} = \emptyset$ , where  $M_{sat} = \{\{u_s, V_s\}\}_{s=1}^{|U_{sat}|}$ 
2 for all  $u_s \in U_{sat}$  do
3    $Cand_{u_s} = \text{QUERYNEIGHINDEX}(\mathcal{N}, L_E^Q(u_c, u_s), v_c)$ 
4    $Cand_{u_s} = Cand_{u_s} \cap \text{PROCESSVERTEX}(u_s, Q, \mathcal{A}, \mathcal{N})$ 
5   if  $Cand_{u_s} \neq \emptyset$  then
6      $M_{sat} = M_{sat} \cup (u_s, Cand_{u_s})$                                 /* Satellite solutions */
7   else
8     return  $M_{sat} := 0$                                               /* No solutions possible */
9 return  $M_{sat}$                                                          /* Matches for satellite vertices */
```

Given a core vertex u_c , we initially find a set of candidate solutions $Cand_{u_c}$, by using the index \mathcal{S} . Then, for each candidate solution $v_c \in Cand_{u_c}$, the set of solutions for all the satellite vertices U_{sat} that are connected to u_c are returned by the MATCHSATVERTICES procedure, described in Algorithm 2. The set of solution tuple M_{sat} defined in Line 1, stores the candidate solutions for the entire set of satellite vertices U_{sat} . Formally, $M_{sat} = \{\{u_s, V_s\}\}_{s=1}^{|U_{sat}|}$, where $u_s \in U_{sat}$ and V_s is a set of candidate solutions for u_s . In order to obtain candidate solutions for u_s , we query the neighbourhood index \mathcal{N} (Line 3); the QUERYNEIGHINDEX function returns all the neighbourhood vertices of already matched v_c by considering the multi-edge in the query multigraph $L_E^Q(u_c, u_s)$. As every query vertex $u_s \in U_{sat}$ is processed, the solution set M_{sat} that contains candidate solutions grows until all the satellite vertices have been processed (Lines 2-8).

In Line 4, the set of candidate solutions $Cand_{u_s}$ are refined by invoking Algorithm 1 (VERTEXPROCESSING). After the refinement, if there are finite candidate solutions, we update the solution M_{sat} ; else, we terminate the procedure as there can be no matches for a given matched vertex v_c . The MATCHSATVERTICES procedure performs two tasks: firstly, it checks if the candidate vertex $v_c \in Cand_{u_s}$ is a valid matchable vertex and secondly, it obtains the solutions for all the satellite vertices.

0.6.3. Arbitrary Query Processing

Algorithm 3 shows the generic procedure we develop to process arbitrary queries. Recall that for an arbitrary query Q , we define two different types of vertexes: a set of core vertices U_c and a set of satellite vertices U_s . The QUERYDECOMPOSE procedure in Line 1 of Algorithm 3, performs this decomposition by splitting the query vertices

U into U_c and U_s , as observed in Figure 4. To process arbitrary query multigraphs, we perform recursive sub-multigraph matching procedure on the set of core vertices $U_c \subseteq U$; during the recursion, satellite vertexes connected to a specific core vertex are processed too. Since the recursion is performed on the set of core vertices, we propose a few heuristics for ordering the query vertices.

Ordering of the query vertices forms one of the vital steps for subgraph matching algorithms [KIM 15]. In any subgraph matching algorithm, the embeddings of a query subgraph are obtained by exploring the solution space spanned by the data graph. But since the solution space itself can grow exponentially in size, we are compelled to use intelligent strategies to traverse the solution space. In order to achieve this, we propose a heuristic procedure VERTEXORDERING (Line 2, Algorithm 3) that employs two ranking functions.

The first ranking function r_1 relies on the number of satellite vertices connected to the core vertex, and the query vertices are ordered with the decreasing rank value. Formally, $r_1(u) = |U_{sat}|$, where $U_{sat} = \{u_s | u_s \in U_s \wedge (u, u_s) \in E(Q)\}$. A vertex with more satellite vertices connected to it, is rich in structure and hence it would probably yield fewer candidate solutions to be processed under recursion. Thus, in Figure 4, u_1 is chosen as an initial vertex. The second ranking function r_2 relies on the number of incident edges on a query vertex. Formally, $r_2(u) = \sum_{j=1}^m |\sigma(u)^j|$, where u has m multiedges and $|\sigma(u)^j|$ captures the number of edge types in the j^{th} multiedge. Again, U_c^{ord} contains the ordered vertices with the decreasing rank value r_2 . Further, when there are no satellite vertices in the query Q , this ranking function gets the priority. Despite the usage of any ranking function, the query vertices in U_c^{ord} , when accessed in sequence, should be structurally connected to the previous set of vertices. If two vertices tie up with the same rank, the rank with lesser priority determines which vertex wins. Thus, for the example in Figure 4, the set of ordered core vertices is $U_c^{ord} = \{u_1, u_3, u_5\}$.

The first vertex in the set U_c^{ord} is chosen as the initial vertex u_{init} (Line 3), and subsequent query vertices are chosen in sequence. The candidate solutions for the initial query vertex $CandInit$ are returned by QUERYINDEX procedure (Line 4), that are constrained by the structural properties (neighbourhood structure) of u_{init} . By querying the index \mathcal{S} for initial query vertex u_{init} , we obtain the candidate solutions $CandInit \in V$ that match the structure (multiedge types) associated with u_{init} . Although some candidates in $CandInit$ may be invalid, all valid candidates are present in $CandInit$, as deduced in Lemma 1. Further, PROCESSVERTEX procedure is invoked to obtain the candidate solutions according to vertex attributes and IRI information, and then only the common candidates are retained.

Before getting into the algorithmic details, we explain how the solutions are handled and how we process each query vertex. We define M as a set of tuples, whose i^{th} tuple is represented as $M_i = [m_c, M_s]$, where m_c is a solution pair for a core vertex,

Algorithm 3: AMBER-Algo (\mathcal{I}, Q)

```

1 QUERYDECOMPOSE: Split  $U$  into  $U_c$  and  $U_s$ 
2  $U_c^{ord} = \text{VERTEXORDERING}(Q, U_c)$ 
3  $u_{init} = u | u \in U_c^{ord}$ 
4  $CandInit = \text{QUERYSYNINDEX}(u_{init}, \mathcal{S})$ 
5  $CandInit = CandInit \cap \text{PROCESSVERTEX}(u_{init}, Q, \mathcal{A}, \mathcal{N})$ 
6 FETCH:  $U_{init}^{sat} = \{u | u \in U_s \wedge (u_{init}, u) \in E(Q)\}$ 
7 SET:  $Emb = \emptyset$ 
8 for  $v_{init} \in CandInit$  do
9   SET:  $M = \emptyset, M_s = \emptyset, M_c = \emptyset$ 
10  if  $U_{init}^{sat} \neq \emptyset$  then
11     $M_{sat} = \text{MATCHSATVERTICES}(\mathcal{A}, \mathcal{N}, Q, U_{init}^{sat}, v_{init})$ 
12    if  $M_{sat} \neq \emptyset$  then
13      for  $[u_s, V_s] \in M_{sat}$  do
14        UPDATE:  $M_s = M_s \cup [u_s, V_s]$ 
15        UPDATE:  $M_c = M_c \cup [u_{init}, v_{init}]$ 
16         $Emb = Emb \cup \text{HOMOMORPHICMATCH}(M, \mathcal{I}, Q, U_c^{ord})$ 
17  else
18    UPDATE:  $M_c = M_c \cup (u_{init}, v_{init})$ 
19     $Emb = Emb \cup \text{HOMOMORPHICMATCH}(M, \mathcal{I}, Q, U_c^{ord})$ 
20 return  $Emb$ 

```

/* Homomorphic embeddings of query multigraph */

and M_s is a set of solution pairs for the set of satellite vertices that are connected to the core vertex. Formally $m_c = (u_c, v_c)$, where u_c is the core vertex and v_c is the corresponding matched vertex; M_s is a set of solution pairs, whose j^{th} element is a solution pair (u_s, V_s) , where u_s is a satellite vertex and V_s is a set of matched vertices. In addition, we maintain a set M_c whose elements are the solution pairs for all the core vertices. Thus during each recursion branch, the size of M grows until it reaches the query size $|U|$; once $|M| = |U|$, homomorphic matches are obtained.

For all the candidate solutions of initial vertex $CandInit$, we perform recursion to obtain homomorphic embeddings (lines 8-19). Before getting into recursion, for each initial match $v_{init} \in CandInit$, if it has satellite vertices connected to it, we invoke the `MATCHSATVERTICES` procedure (Lines 10-11). This step not only finds solution matches for satellite vertices, if there are, but also checks if v_{init} is a valid candidate vertex. If the returned solution set M_{sat} is empty, then v_{init} is not a valid candidate and hence we continue with the next $v_{init} \in CandInit$; else, we update the set of solution pairs M_s for satellite vertices and the solution pair M_c for the core vertex (Lines 12-15) and invoke `HOMOMORPHICMATCH` procedure (Lines 17). On the other hand, if there are no satellite vertices connected to u_{init} , we update the core vertex solution set M_c and invoke `HOMOMORPHICMATCH` procedure (Lines 18-19).

In the `HOMOMORPHICMATCH` procedure (Algorithm 4), we fetch the next query vertex from the set of ordered core vertices U_c^{ord} (Line 4). Then we collect the neighbourhood vertices of already matched core query vertices and the corresponding matched data vertices (Lines 5-6). As we recall, the set M_c maintains the solution

Algorithm 4: HOMOMORPHICMATCH($M, \mathcal{I}, Q, U_c^{ord}$)

```

1 if  $|M| = |U|$  then
2   return GENEMB( $M$ )
3  $Emb = \emptyset$ 
4 FETCH:  $u_{nxt} = u | u \in U_c^{ord}$ 
5  $N_q = \{u_c | u_c \in M_c\} \cap adj(u_{nxt})$ 
6  $N_g = \{v_c | v_c \in M_c \wedge (u_c, v_c) \in M_c\}$ , where  $u_c \in N_q$ 
7  $Cand_{u_{nxt}} = \bigcap_{n=1}^{|N_q|} (QueryNeighIndex(\mathcal{N}, L_E^Q(u_n, u_{nxt}), v_n))$ 
8  $Cand_{u_{nxt}} = Cand_{u_{nxt}} \cap PROCESSVERTEX(u_{nxt}, Q, \mathcal{A}, \mathcal{N})$ 
9 for each  $v_{nxt} \in Cand_{u_{nxt}}$  do
10  FETCH:  $U_{nxt}^{sat} = \{u | u \in V_s \wedge (u_{nxt}, u) \in E(Q)\}$ 
11  if  $U_{nxt}^{sat} \neq \emptyset$  then
12     $M_{sat} = MATCHSATVERTICES(\mathcal{A}, \mathcal{N}, Q, U_{nxt}^{sat}, v_{nxt})$ 
13    if  $M_{sat} \neq \emptyset$  then
14      for every  $[u^s, V^s] \in M_{sat}$  do
15        UPDATE:  $M_s = M_s \cup [u^s, V^s]$ 
16        UPDATE:  $M_c = M_c \cup (u_{nxt}, v_{nxt})$ 
17         $Emb = Emb \cup HOMOMORPHICMATCH(M, \mathcal{I}, Q, U_c^{ord})$ 
18  else
19    UPDATE:  $M_c = M_c \cup (u_{nxt}, v_{nxt})$ 
20     $Emb = Emb \cup HOMOMORPHICMATCH(M, \mathcal{I}, Q, U_c^{ord})$ 
21 return  $Emb$ 

```

pair $m_c = (u_c, v_c)$ of each matched core query vertex. The set N_q collects the already matched core vertices $u_c \in M_c$ that are also in the neighbourhood of u_{nxt} , whose matches have to be found. Further, N_g contains the corresponding matched query vertices $v_c \in M_c$. As the recursion proceeds, we find those matchable data vertices of u_{nxt} that are in the neighbourhood of all the matched vertices $v \in N_g$, so that the query structure is maintained. In Line 7, for each $u_n \in N_q$ and the corresponding $v_n \in N_g$, we query the neighbourhood index \mathcal{N} , to obtain the candidate solutions $Cand_{u_{nxt}}$, that are in the neighbourhood of already matched data vertex v_n and have the multiedge $L_E^Q(u_n, u_{nxt})$, obtained from the query multigraph Q . Finally (line 7), the set of candidate solutions that are common for every $u_n \in N_q$ are retained in $Cand_{u_{nxt}}$.

Further, the candidate solutions are refined with the help of PROCESSVERTEX procedure (Line 8). Now, for each of the valid candidate solution $v_{nxt} \in Cand_{u_{nxt}}$, we recursively call the HOMOMORPHICMATCH procedure. When the next query vertex u_{nxt} has no satellite vertices attached to it, we update the core vertex solution set M_c and call the recursion procedure (Lines 19-20). But when u_{nxt} has satellite vertices attached to it, we obtain the candidate matches for all the satellite vertices by invoking the MATCHSATVERTICES procedure (Lines 11-12); if there are matches, we update both the satellite vertex solution M_s and the core vertex solution M_c , and invoke the recursion procedure (Line 17).

Once all the query vertices have been matched for the current recursion step, the solution set M contains the solutions for both core and satellite vertices. Thus when all the query vertices have been matched, we invoke the GENEMB function (Line 2) which returns the set of embeddings, that are updated in Emb . The GENEMB function treats the solution vertex v_c of each core vertex as a singleton and performs Cartesian product among all the core vertex singletons and satellite vertex sets. Formally, $Emb_{part} = \{v_c^1\} \times \dots \times \{v_c^{|U_c|}\} \times V_s^1 \times \dots \times V_c^{|U_s|}$. Thus, the partial set of embeddings Emb_{part} is added to the final result Emb .

0.7. Experimental Analysis

In this section, we report on our extensive experiments on two RDF datasets. We evaluate the time performance and the robustness of AMBER w.r.t. the state-of-the-art competitors by varying the size, and the structure of the SPARQL queries. Experiments are carried out on a 64-bit Intel Core i7-4900MQ @ 2.80GHz, with 32GB memory, running Linux OS - Ubuntu 14.04 LTS. AMBER is implemented in C++.

0.7.1. Experimental Setup

We compare AMBER with the four standard RDF engines: *Virtuoso-7.1* [ERL 12], *x-RDF-3X* [NEU 10], *Apache Jena* [CAR 04] and *gStore* [ZOU 14b]. For all the competitors we employ the source code available on the web site or obtained by the authors. Another recent work *Turbo_{HOM}++* [KIM 15] has been excluded since it is not publicly available.

For the experimental analysis we use two RDF datasets - *DBPEDIA* and *YAGO*. *DBPEDIA* constitutes the most important knowledge base for the Semantic Web community. Most of the data available in this dataset comes from the Wikipedia Infobox. *YAGO* is a real world dataset built from factual information coming from *Wikipedia* and *WordNet* semantic network. The time required to build the multigraph database as well as to construct the indexes are reported in Table 3b. We note that the database building time and the corresponding size are proportional to the number of triples. Regarding the indexing structures, we underline that both building time and size are proportional to the number of edges. For instance, *DBPEDIA* has the highest number of edges ($\sim 15M$) and, consequently, AMBER employs more time and space to build and store its data structure.

0.7.2. Workload Generation

In order to test the scalability and robustness of the different RDF engines, we generate the query workloads considering a similar setting as in [GUB 14, ALU 14a,

| Dataset | # Triples | # Vertices | # Edges | # Edge types | Dataset | Database | | Index \mathcal{I} | |
|----------------|------------|------------|------------|--------------|----------------|---------------|------|---------------------|------|
| | | | | | | Building Time | Size | Building Time | Size |
| <i>DBPEDIA</i> | 33 071 359 | 4 983 349 | 14 992 982 | 676 | <i>DBPEDIA</i> | 307 | 1300 | 45.18 | 1573 |
| <i>YAGO</i> | 35 543 536 | 3 160 832 | 10 683 425 | 44 | <i>YAGO</i> | 379 | 2400 | 29.1 | 1322 |

(a) Statistics of Datasets

(b) Database and Index Construction time (seconds) and memory usage (Mbytes)

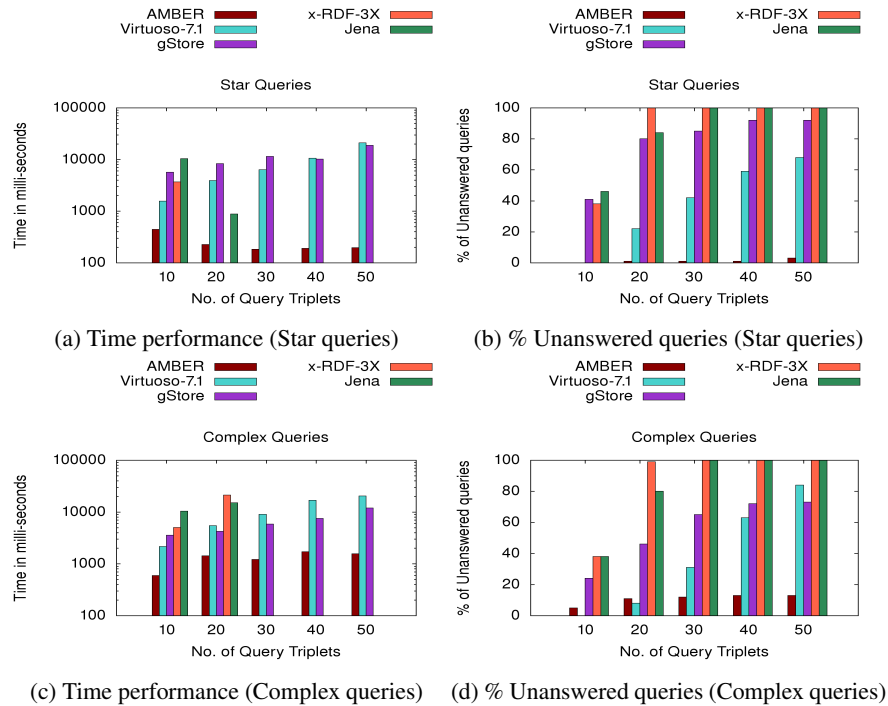
HAN 13]. We generate the query workload from the respective RDF datasets, which are available as RDF triplesets. In specific, we generate two types of query sets: a star-shaped and a complex-shaped query set; further, both query sets are generated for varying sizes (say k) ranging from 10 to 50 triples, in steps of 10.

To generate star-shaped or complex-shaped queries of size k , we pick an initial-entity at random from the RDF data. Now to generate star queries, we check if the initial-entity is present in at least k triples in the entire benchmark, to verify if the initial-entity has k neighbours. If so, we choose those k triples at random; thus the initial entity forms the central vertex of the star structure and the rest of the entities form the remaining star structure, connected by the respective predicates. To generate complex-shaped queries of size k , we navigate in the neighbourhood of the initial-entity through the predicate links until we reach size k . In both query types, we inject some object literals as well as constant *IRIs*; rest of the *IRIs* (subjects or objects) are treated as variables. However, this strategy could choose some very unselective queries [GUB 14]. In order to address this issue, we set a maximum time constraint of 60 seconds for each query. If the query is not answered in time, it is not considered for the final average (similar procedure is usually employed for graph query matching [HAN 13] and RDF workload evaluation [ALU 14a]). We report the average query time and, also, the percentage of unanswered queries (considering the given time constraint) to study the robustness of the approaches.

0.7.3. Comparison with RDF Engines

In this section we report and discuss the results obtained by the different RDF engines. For each combination of query type and benchmark we report two plots by varying the query size: the average time and the corresponding percentage of unanswered queries for the given time constraint. We remind that the average time per approach is computed only on the set of queries that were answered.

The experimental results for *DBPEDIA* are depicted in Figure 6. The time performance (averaged over 200 queries) for *Star-Shaped* queries (Fig. 6a), affirm that *AMBER* clearly outperforms all the competitors. Further the robustness of each approach, evaluated in terms of percentage of unanswered queries within the stipulated time, is shown in Figure 6b. For the given time constraint, *x-RDF-3X* and *Jena* are

Figure 6 – Evaluation of time performance and robustness for *DBPEDIA*.

unable to output results for size 20 and 30 onwards respectively. Although *Virtuoso* and *gStore* output results until query size 50, their time performance is still poor. However, as the query size increases, the percentage of unanswered queries for both *Virtuoso* and *gStore* keeps on increasing from $\sim 0\%$ to 65% and $\sim 45\%$ to 95% respectively. On the other hand AMBER answers $>98\%$ of the queries, even for queries of size 50, establishing its robustness.

Analyzing the results for *Complex-Shaped* queries, we observe that in Figure 6c, *x-RDF-3X* and *Jena* are the slowest engines; *Virtuoso* and *gStore* perform better than them but nowhere close to AMBER. We further observe that *x-RDF-3X* and *Jena* are the least robust as they don't output results for size 30 onwards (Fig. 6d); on the other hand AMBER is the most robust engine as it answers $>85\%$ of the queries even for size 50. The percentage of unanswered queries for *Virtuoso* and *gStore* increase from 0% to $\sim 80\%$ and 25% to $\sim 70\%$ respectively, as we increase the size from 10 to 50.

The results for *YAGO* are reported in Figure 7. For *Star-Shaped* queries, the time performance of AMBER is 1-2 order of magnitude better than its nearest competitor

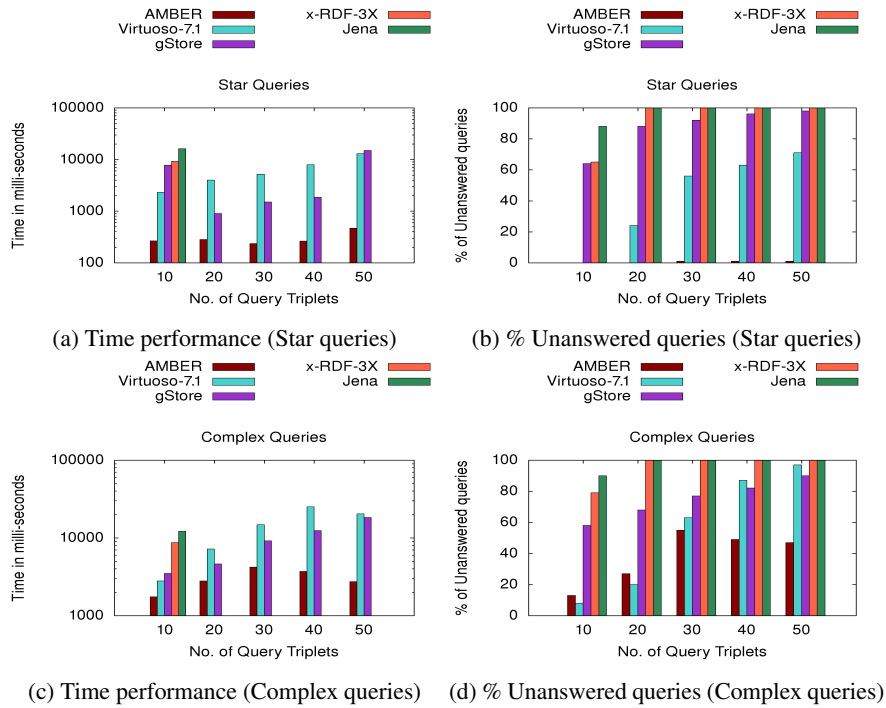


Figure 7 – Evaluation of time performance and robustness for *YAGO*.

Virtuoso as observed in (Fig. 7a), and the performance remains stable even with increasing query size (Fig. 7b). *x-RDF-3X*, *Jena* are not able to output results for size 20 onwards. As observed for *DBPEDIA*, *Virtuoso* seems to become less robust with the increasing query size. For size 20-40, time performance of *gStore* seems better than *Virtuoso*; the reason seems to be the fewer queries that are being considered. Conversely, *AMBER* is able to supply answers most of the time (>98%). Coming to the results for *Complex-Shaped* queries, we observe that *AMBER* is still the best in time performance as seen in Figure 7c; *Virtuoso* and *gStore* are the closest competitors. Only for size 10 and 20, *Virtuoso* seems more robust than *AMBER*. *Jena*, *x-RDF-3X* do not answer queries for size 20 onwards, as seen in Figure 7d.

To summarise, we observe that *Virtuoso* is robust for *Complex-Shaped* smaller queries (10-20), but fails for bigger (>20) queries. *x-RDF-3X* fails for queries with size bigger than 10. *Jena* has reasonable behavior until size 20, but fails to deliver from size 30 onwards. *gStore* has a reasonable behavior for size 10, but its robustness deteriorates from size 20 onwards. *AMBER* clearly outperforms the state-of-the-art approaches in terms of time performance and robustness.

0.8. Conclusion

In this chapter, a multigraph based engine AMBER has been proposed in order to answer SPARQL queries. The multigraph representation has bestowed us with two advantages: on one hand, it enables us to construct lightweight indexing structures, that ameliorate the time performance of AMBER; on the other hand, the graph representation itself motivates us to exploit recent graph management techniques. The proposed engine AMBER has been tested over large RDF triplestores. We have observed that AMBER outperforms its state-of-the-art competitors on two main aspects: its robustness with respect to the query size; it stands out for its time performances.

0.9. Acknowledgment:

This work has been funded by LabEx NUMEV integrated into the I-SITE MUSE (ANR- 10-LABX-20).

Bibliography

- [ALU 14a] ALUÇ G., HARTIG O., ÖZSU M. T., DAUDJEE K., “Diversified Stress Testing of RDF Data Management Systems”, *ISWC*, p. 197–212, 2014.
- [ALU 14b] ALUÇ G., ÖZSU M. T., DAUDJEE K., “Workload Matters: Why RDF Databases Need a New Design”, *PVLDB*, vol. 7, num. 10, p. 837–840, 2014.
- [BRO 02] BROEKSTRA J., KAMPMAN A., VAN HARMELEN F., “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema”, *ISWC*, p. 54–68, 2002.
- [CAB 12] CABRIO E., COJAN J., APROSIO A. P., MAGNINI B., LAVELLI A., GANDON F., “QAKiS: an Open Domain QA System based on Relational Patterns”, *ISWC*, 2012.
- [CAR 04] CARROLL J. J., DICKINSON I., DOLLIN C., REYNOLDS D., SEABORNE A., WILKINSON K., “Jena: implementing the semantic web recommendations”, *WWW*, p. 74–83, 2004.
- [DAS 14] DAS S., SRINIVASAN J., PERRY M., CHONG E. I., BANERJEE J., “A Tale of Two Graphs: Property Graphs as RDF in Oracle.”, *EDBT*, p. 762–773, 2014.
- [ERL 12] ERLING O., “Virtuoso, a Hybrid RDBMS/Graph Column Store”, *IEEE Data Eng. Bull.*, vol. 35, num. 1, p. 3–8, 2012.
- [GUB 14] GUBICHEV A., NEUMANN T., “Exploiting the query structure for efficient join ordering in SPARQL queries.”, *EDBT*, p. 439–450, 2014.
- [HAN 13] HAN W., LEE J., LEE J., “Turbo_{iso}: towards ultrafast and robust subgraph isomorphism search in large graph databases”, *SIGMOD*, p. 337–348, 2013.
- [HUA 11] HUANG J., ABADI D. J., REN K., “Scalable SPARQL querying of large RDF graphs”, *PVLDB*, vol. 4, num. 11, p. 1123–1134, 2011.
- [ING 16] INGALALLI V., IENCO D., PONCELET P., VILLATA S., “Querying RDF Data Using A Multigraph-based Approach”, *EDBT: Extending Database Technology*, Page 12, 2016.
- [KIM 15] KIM J., SHIN H., HAN W., HONG S., CHAFI H., “Taming Subgraph Isomorphism for RDF Query Processing”, *PVLDB*, vol. 8, num. 11, p. 1238–1249, 2015.
- [MOR 11] MORSEY M., LEHMANN J., AUER S., NGOMO A., “DBpedia SPARQL benchmark performance assessment with real queries on real data.”, *ISWC*, p. 454–469, 2011.

- [NEU 10] NEUMANN T., WEIKUM G., “x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases”, *PVLDB*, vol. 3, num. 1, p. 256–263, 2010.
- [TER 06] TERROVITIS M., PASSAS S., VASSILIADIS P., SELLIS T., “A combination of trie-trees and inverted files for the indexing of set-valued attributes”, *CIKM*, ACM, p. 728–737, 2006.
- [ZOU 14a] ZOU L., HUANG R., WANG H., YU J. X., HE W., ZHAO D., “Natural language question answering over RDF: a graph data driven approach”, *SIGMOD Conference*, p. 313–324, 2014.
- [ZOU 14b] ZOU L., ÖZSU M. T., CHEN L., SHEN X., HUANG R., ZHAO D., “gStore: a graph-based SPARQL query engine”, *VLDB J.*, vol. 23, num. 4, p. 565–590, 2014.