An Efficient Index for RDF Query Containment

Theofilos Mailis*
Athena Research Centre
University of Athens, Greece
tmailis@di.uoa.gr

Yannis Kotidis†
Athens University of Economics
and Business, Greece
kotidis@aueb.gr

Vaggelis Nikolopoulos
University of Athens
Greece
vgnikolop@di.uoa.gr

Evgeny Kharlamov‡
University of Oslo, Norway
Bosch Center for AI, Germany
evgeny.kharlamov@ifi.uio.no

Ian Horrocks
University of Oxford
United Kingdom
ian.horrocks@cs.ox.ac.uk

Yannis Ioannidis
Athena Research Centre
University of Athens, Greece
yannis@di.uoa.gr

ABSTRACT

Query containment is a fundamental operation used to expedite query processing in view materialisation and query caching techniques. Since query containment has been shown to be NP-complete for arbitrary conjunctive queries on RDF graphs, we introduce a simpler form of conjunctive queries that we name f-graph queries. We first show that containment checking for f-graph queries can be solved in polynomial time. Based on this observation, we propose a novel indexing structure, named mv-index, that allows for fast containment checking between a single f-graph query and an arbitrary number of stored queries. Search is performed in polynomial time in the combined size of the query and the index. We then show how our algorithms and structures can be extended for arbitrary conjunctive queries on RDF graphs by introducing f-graph witnesses, i.e., f-graph representatives of conjunctive queries. F-graph witnesses have the following interesting property, a conjunctive query for RDF graphs is contained in another query only if its corresponding f-graph witness is also contained in it. The latter allows to use our indexing structure for the general case of conjunctive query containment. This translates in practice to microseconds or less for the containment test against hundreds of thousands of queries that are indexed within our structure.

ACM Reference Format:
Theofilos Mailis, Yannis Kotidis, Vaggelis Nikolopoulos, Evgeny Kharlamov, Ian Horrocks, and Yannis Ioannidis. 2019. An Efficient Index for RDF Query Containment. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319864

1 INTRODUCTION AND MOTIVATION

The growing popularity of graph-structured data in many real-world applications such as life science databases, e.g. PUBCHEM, co-purchase networks, e.g. Amazon.com, and Web Search, e.g. Google Knowledge Graph, has led to a renaissance of research on graph data management. *RDF* [14] and *SPARQL* [61] are promising examples of a graph data model and the corresponding query language that have gained a lot of attraction. Indeed, *DBpedia*, an RDF version of Wikipedia, serves as the main hub for the Linked Open Data initiative and consists of more than 1 billion RDF triples. In order to handle the burst of RDF data that is available on the Web, much research has been devoted on scalable techniques for RDF processing. Various systems for RDF processing have been developed [1, 7, 15, 48, 54, 64, 72], using techniques such as indexing, caching, and view materialisation in order to accelerate the execution time of SPARQL queries.

Query caching and view materialisation are directly related to the problem of query containment [22, 45] as we will see in Section 2. The containment problem has been proved to be NP-complete for arbitrary conjunctive queries [17] and unions of conjunctive queries [62] over relational databases. The same results also apply for conjunctive queries on RDF graphs and their SPARQL counterparts [31, 60]. By examining the real-world query workload of DBPedia, we observe

*T. Mailis has received funding from EU Horizon2020, "DARE" project, Grant Agreement nr. 777413.
†Y. Kotidis was financed by the Research Centre of Athens University of Economics and Business, in the framework of the project entitled "Original Scientific Publications".
‡E. Kharlamov was partially financed by the Norwegian Research Council project under the number 237898.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319864

that only a small fragment of the queries have all the properties that make query containment so hard to solve. Based on this observation, we identify f-graph queries, a restricted form of conjunctive queries that allow to solve the containment checking problem in PTime. Based on f-graphs, we propose an efficient indexing structure, the mv-index, for checking the containment relation between a single f-graph query Q_f and a set of indexed queries $\mathcal{W}$ in polynomial time w.r.t. the combined size of the query and the index. By introducing witnesses, i.e., representatives of arbitrary queries, we can further extend mv-indices to evaluate containment for arbitrary conjunctive queries on RDF graphs. In real-world query workloads, this translates to microseconds or less for the containment test against hundreds of thousands of queries that are indexed within the structure. Because of this, mv-indices are the perfect candidate to be combined with existing and novel materialisation and caching techniques in order to efficiently accelerate the execution time of pragmatic query workloads. The major contributions of this paper are:

► **F-Graph Definition, Query Containment.** We define f-graph queries, restricted RDF conjunctive queries whose structure allows to check for query containment in PTime. We initially focus on query containments of the form $Q_f \sqsubseteq W$ between an f-graph and a query W that has only IRIs in the predicate position of a triple pattern. The algorithm for containment checking operates in PTime and is based on a serialised representation of queries that encodes each query starting from an anchor vertex and blending IRIs, literals, and parenthesis symbols to represent its nested subgraphs.

► **Mv-indices.** The structure of the serialised form of queries allows to introduce the materialised-view indices, mv-indices, novel indexing structures for checking query containment. Mv-indices are tree-like structures that are based on *Radix trees*. They represent queries as vertices within the Radix tree, while edges correspond to query patterns that appear in one or more queries. Mv-indices (i) allow to represent in a compact form thousands of queries by taking advantage of common patterns that appear in them; (ii) allow to evaluate query containment between an f-graph query Q_f and an arbitrary number of indexed queries within polynomial time in the combined size of the query and the index; (iii) permit updating of the mv-index structure with additional queries in linear time with respect to the newly-inserted-query size.

► **F-Graph Witnesses & Variables as Predicates.** F-graph witnesses allow to represent an arbitrary RDF conjunctive query in the left hand side of a query containment. F-graph witnesses have the following interesting property, an RDF conjunctive query is contained in another query only if its corresponding f-graph witness is contained in it. F-graph witnesses provide a partial answer to the query containment problem and a NP check has to be performed to check if the containment indeed applies. We additionally present the methodology for solving the containment problem for the unrestricted case where variables may appear as predicates. The latter allows to use the mv-index structure for arbitrary RDF conjunctive queries.

► **RDF Schema.** Finally, we have extended our algorithm for containment checking to take into consideration the implicit information that can be inferred based on the terminological knowledge that is expressed in the form of an RDF Schema (RDFS) [14]. This can be accomplished by introducing an additional step for containment checking that extends the examined query based on the RDF schema.

We have implemented our novel structures and algorithms and tested their efficiency in a combined query workload consisting of DBPedia, WatDiv, BSBM, LUBM, and LDBC queries. This workload is described in detail in our evaluation Section and consists of $1,536,378$ queries. We have evaluated insertion and containment performance with respect to different query and mv-index properties. The average time for query containment against an mv-index containing $397,507$ distinct queries from all 5 workloads was between 0.0093 msec and 0.041 msec.

In Section 2 we provide some preliminary definitions. In Section 3 we introduce f-graph queries and provide the polynomial algorithm for containment checking for f-graph queries. In Section 4 we present the mv-index and its usage for computing multiple containments. In Section 5 we introduce f-graph witnesses that allow to represent arbitrary conjunctive queries within the mv-index and check for containment. In Section 6 we extend our techniques to handle an available RDF Schema. In Section 7 we perform an experimental evaluation of our structures and indexes. Finally, Section 8 presents the current literature on RDF stores, query containment, and view materialisation, while Section 9 summarises the paper and mentions directions for future work.

2 PRELIMINARIES

Initially, we will present some preliminary definitions in order to formalise the problem of query containment. In the rest of the paper we assume that an RDF data graph is defined via set semantics. Bag semantics have also been suggested in the bibliography, but the theoretical complexity of query containment w.r.t. bag semantics remains an open problem [2].

RDF Graph [57]. Assume the pairwise disjoint infinite sets I, B, and L of IRIs, Blank nodes, and literals. *IRIs* are *Internationalised Resource Identifiers* that allow to uniquely identify resources within the Semantic web; *literals* are used for values such as strings, numbers, and dates; while *blank nodes* are used to represent resources for which an IRI or literal is

not given. A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple. In this tuple, s is the subject, p the predicate and o the object. An *RDF graph* is a set of RDF triples.

BGP [61]. In the rest of the paper we will denote with IL the union $I \cup L$. If we additionally assume the existence of an infinite set X of variables disjoint from the above sets, a *triple pattern* is an element of $(I \cup X) \times (I \cup X) \times (IL \cup X)$. A *basic graph pattern (BGP)* is a finite set of triple patterns: $\{t_1, \ldots, t_n\}$.

A *SPARQL query* is constituted of a graph pattern along with a solution modifier that specifies the answer variables. For a BGP P and a vector $\vec{x}$ of variables occurring in P, a SPARQL query may have the form `SELECT` $\vec{x}$ `WHERE` P. The `SELECT` clause identifies the variables to appear in the query results and the `WHERE` clause provides the BGP to match against the RDF graph. The variables in $\vec{x}$ are called *distinguished variables*. SPARQL queries constituting of a `SELECT` and a BGP `WHERE` clause are equivalent to conjunctive queries for RDF graphs. Therefore, we will say RDF conjunctive queries and BGP queries to denote SPARQL queries with the aforementioned form. For a BGP P, a SPARQL query may also have the form `ASK WHERE` P. This type of query corresponds to an RDF Boolean conjunctive query.

Query answering. A solution to a BGP query Q with a `SELECT` clause on an RDF graph G is a mapping $m : \text{vars}(Q) \rightarrow IL \cup B$ from the variables in Q to IRIs, Blank nodes, and literals in G such that the substitution of variables would yield a subgraph of G. For a BGP query, the substitutions of distinguished variables constitute the answers to the query. `ASK` queries have a yes (or no) answer if a corresponding mapping $m : \text{vars}(Q) \rightarrow IL \cup B$ exists (or not).

Query Containment. [17] A query Q is *contained* in a query W, denoted $Q \sqsubseteq W$, if the answer set of Q is contained in the answer set of W for every possible RDF graph.

Containment Mapping. A *containment mapping* or *homomorphism* $\sigma : W \rightarrow Q$ from a query W to a query Q is a mapping σ from the variables of W into the variables, IRIs, and literals of Q, such that every triple in the graph pattern of W is mapped to a triple in Q. Chandra and Merlin [17] have proved that for Boolean conjunctive queries, a containment mapping from W to Q implies that Q is contained in W [17].

Example 2.1. An RDF graph G contains information related to songs and albums represented as triples:

$(s1, name, \text{“Masquerade”}), (s1, fromAlbum, al1), (al1, name, \text{“The Phantom of the Opera”}), (al1, artist, ar3), (ar3, name, \text{“Andrew L. Webber”}), (ar3, type, MusicalArtist)$

In the corresponding RDF graph, we use quotation marks to distinguish literals from IRIs.

For our running example, we will ask for information related to a specific song. We ask for the name and the album name of a song that is contained within an album in which a musical artist participates. In the following query, elements with a question mark correspond to variables in X:

$$\begin{aligned} Q: \texttt{SELECT } (?sN, ?aN) \texttt{ WHERE } \{&(?sng, \text{name}, ?sN), \\ &(?sng, \text{fromAlbum}, ?alb), (?alb, \text{name}, ?aN), \\ &(?alb, \text{artist}, ?art), (?art, \text{type}, \text{MusicalArtist})\} \end{aligned} \tag{1}$$

The answer to the query if applied on the sample graph database will be the pair (“Masquerade", “The Phantom of the Opera").

Suppose we want to examine containment between the query Q and the view W:

$$\begin{aligned} W: \texttt{SELECT } (?y, ?w) \texttt{ WHERE } \{&(?x, \text{name}, ?y), \\ &(?x, \text{fromAlbum}, ?z), (?z, \text{name}, ?w)\} \end{aligned} \tag{2}$$

The containment mapping $\sigma : W \rightarrow Q$ such that $\sigma(?x) = ?sng$, $\sigma(?y) = ?sN$, $\sigma(?z) = ?alb$, $\sigma(?w) = ?aN$, indicates that $Q \sqsubseteq W$.

View Materialisation & Query Containment. A *view* is a stored query, while a *materialised view* is the result set of the stored query on a specific database instance. A query Q' is a *rewriting* of Q that uses the views $\mathcal{W} = \{W_1, \ldots, W_m\}$ if Q and Q' are equivalent, i.e., they have the same answer set and Q' contains one or more occurrences of materialised views in $\mathcal{W}$.

Levy et al. [45] prove that for the conjunctive queries Q and W, there is a rewriting of Q using W iff $\pi_\emptyset(Q) \sqsubseteq \pi_\emptyset(W)$, i.e., the projection of Q onto the empty set of columns is contained in the projection of W onto the empty set of columns (the projections $\pi_\emptyset(Q)$, $\pi_\emptyset(W)$ are actually Boolean conjunctive queries). Additionally, they provide the methodology for finding the rewritings of Q based on every containment mapping $\sigma : \pi_\emptyset(W) \rightarrow \pi_\emptyset(Q)$ with $W \in \mathcal{W}$. Given a query Q, a set of views $\mathcal{W}$, and their corresponding materialisations, a query optimiser that utilises the existing view materialisations has to: (i) identify the available rewritings of Q; (ii) determine the rewriting Q' that is less costly w.r.t. total execution time; (iii) decide whether it is beneficial to execute Q' instead of Q.

3 F-GRAPH QUERIES & QUERY CONTAINMENT

The objective of this paper is the construction of an indexing structure that will allow to: *(i)* efficiently store a set of BGP queries into an index $\mathcal{W}$; *(ii)* given a BGP query Q discover every query $W \in \mathcal{W}$ for which $Q \sqsubseteq W$ applies along with the corresponding containment mapping. Our indexing structure can be employed in view-materialisation scenarios, allowing the query optimiser to identify the rewritings of a query Q that employ the materialised views in $\mathcal{W}$.

Without loss of generality, we will focus on finding containment mappings for Boolean queries. For the non-Boolean queries Q and W, when checking if the containment $Q \sqsubseteq W$ applies, we just need an extra step that maps the SELECT clause of W to the SELECT clause of Q.

3.1 F-Graph Queries

The containment problem between two BGP queries is itself hard to solve, specifically it belongs to the NP-complete complexity class. In order to solve the containment problem and build the corresponding indexing structure, we initially focus on its variation $Q_f \sqsubseteq W$ where Q_f belongs to a special class of BGP queries that we name f-graph queries and W belongs to the class of BGP queries that have only IRIs as predicates. What motivates the choice of f-graph queries in the left-hand side of a query containment is that: *(i)* containment for f-graph queries can be solved in PTime; *(ii)* f-graph queries appear with a very high percentage within real-world as well as synthetic query workloads; *(iii)* f-graph queries can be employed as representatives of arbitrary queries and they provide us with invaluable information and a partial solution to the containment problem.

F-graph Query. An *f-graph query* Q_f is a BGP query for which: (i) For every pair of terms $o_1, o_2 \in IL \cup X$ such that $o_1 \neq o_2$, the triple patterns (s, p, o_1), (s, p, o_2) cannot both appear in Q_f; (ii) For every pair of terms $s_1, s_2 \in I \cup X$ such that $s_1 \neq s_2$, the triple patterns (s_1, p, o), (s_2, p, o) cannot both appear in Q_f. We name these queries f-graphs because of the functional and inverse functional characteristics of their predicates.

Example 3.1. When checking for the query containment $Q \sqsubseteq W$ in Example 2.1, the query W in Formula 2 does not have a variable in its predicate position, while the query Q is an f-graph query. Therefore as it will be later shown, the corresponding containment can be computed in PTime.

DBpedia-Query Workloads Analysis. To stress out the importance of f-graph queries, we examine a query workload[1] on the *DBpedia semantic knowledge graph* [8]. DBpedia allows users to semantically query relationships and properties of Wikipedia resources, including links to other related datasets. The DBpedia query workload is constituted of $1,291,489$ conjunctive queries. It should be noted that 99.707% of all the BGP queries appearing in DBPedia have only IRIs in the predicate position, while 73.158% of all the BGP queries hold the f-graph property. The previous analysis exemplifies that users tend to perform simple f-graph queries that can be computed in PTime. Thus, an indexing strategy that focuses on f-graph queries would effectively expedite query processing in a typical workload.

[1] https://github.com/AKSW/SPARQL2NL/tree/master/resources/dbpediaLog

Algorithm 1 The algorithm for writing a BGP to its serialised form.

```
1:  function SERIALISATION(Query W, Predicate/InversePredicate r, Vertex v)
2:      v.examined = TRUE
3:      if r = NULL then
4:          nForm:= [v]
5:      else
6:          nForm:= [⟨r, v⟩]
7:      E_v := {(p, o)|(v, p, o) ∈ W, o.examined = FALSE} ∪
        {(p^-1, s)|(s, p, v) ∈ W, s.examined = FALSE}
8:      if E_v = ∅ then
9:          return nForm
10:     else
11:         nForm.append([(])
12:         for all (r, u) in E_v do
13:             nFormSubGraph := SERIALISATION(W, r, u)
14:             nForm.append(nFormSubGraph)
15:         nForm.append([)])
16:     return nForm
```

We will now introduce the algorithm for query containment between an f-graph and a BGP query and prove its polynomial complexity. The algorithm uses a serialised form of BGP queries. As we will explain in Section 4, the corresponding serialised form also allows to represent a set of queries in a compact form by using Radix trees.

3.2 Serialised Form of BGP Queries

The serialisation rewrites each BGP query as a list of elements corresponding to IRIs, literals, and parenthesis symbols as delimiters that denote a subgraph structure. The serialisation operates by choosing a specific vertex of the BGP query as the starting point of the serialisation procedure. We will call this vertex the *anchor vertex* of the serialisation.

Algorithm 1. The serialisation procedure is presented in Algorithm 1. The SERIALISATION function takes as input the query W that needs to be serialised, a predicate or inverse predicate symbol r, and a vertex v of W. The call of the function SERIALISATION(W, NULL, v_a) will return the serialised form of the query W with v_a being the anchor of the serialisation. Intuitively, the serialisation algorithm rewrites the conjunctive query in the form of a list by performing a depth first traversal of its vertices. Opening and closing parenthesis $[(]$, $[)]$ are used to indicate the serialised form of a subgraph structure, while $[\langle p, t\rangle]$ and $[\langle p^{-1}, t\rangle]$ pairs are used to indicate outgoing and incoming edges to the anchor vertex. A parenthesis following a pair of elements $[\langle r, t\rangle]$ $[(]$ ... $[)]$ is

used to indicate the serialisation of the subgraph having t as its anchor vertex.

Example 3.2. With W being the query in Example 2.1 and $?x$ being a variable in W that we choose as an anchor vertex, the execution of Serialisation(W, Null, $?x$) will output the serialised form of the query:

$$\boxed{?x}\ \boxed{(}\ \boxed{\langle \text{fromAlbum}, ?z\rangle}\ \boxed{(}\ \boxed{\langle \text{name}, ?w\rangle}\ \boxed{)}\ \boxed{\langle \text{name}, ?y\rangle}\ \boxed{)}$$

Reading the serialised form of the query from left to right, we ask for a variable $?x$. The opening parenthesis and the first pair appearing in it indicates that $?x$ belongs to an album $?z$. The next opening parenthesis indicates that information about the variable $?z$ will be asked. Specifically, we ask for the name of the album $?z$ which is $?w$. The closing parenthesis indicates that we move back to examining the subgraph that has $?x$ in its anchor position and the next pair indicates that we are asking for the name of the song $?x$. The final parenthesis indicates that our query has been completed.

Execution Time. Based on the study of the depth-first traversal problem [21], the serialisation can be performed in $O(|W|)$ time with $|W|$ being the size of the query W. The output of the serialisation is also linear w.r.t. the size of W.

Note. The serialisation procedure and the rest of the algorithms in this paper focus on BGP queries whose undirected graph constitutes of a single connected component, i.e., we do not allow queries that express Cartesian product operations. Our algorithms and structures for handling queries that express Cartesian product operations can be straightforwardly extended to BGP queries of more than one connected components by separately examining each BGP's connected component.

3.3 Containment Checking

Algorithm 2 provides the function Containment that is used for checking containment between an f-graph query Q_f and the serialised form of a BGP query W named W_s. It takes as input: *(i)* the serialised form of the BGP query W_s; *(ii)* an f-graph query Q_f; *(iii)* a vertex v' in Q_f. If Containment(W_s, Q_f, v') returns True, then there exists a containment mapping from W to Q_f such that the anchor vertex of W is mapped to the term v'.

While examining if there exists a containment mapping from the query W to the f-graph query Q_f, each opening parenthesis in its serialised form W_s indicates that we currently focus on finding containment mappings for one of W's subgraphs. Each serialised subgraph that is enclosed between parentheses in W_s has a corresponding anchor vertex. Additionally, there exists a specific path of edges leading from the anchor vertex of W_s to the anchor vertex of the examined subgraph. Suppose that $\vec{p}_{\text{ath}}$ is the corresponding vector of

Algorithm 2 The algorithm for checking containment between a BGP and a graph.

```
1:  function Containment(SerialisedForm W_s, FGraph Q_f,
    Vertex v')
2:      σ(t) := t for every t ∈ IL
3:      Stack m⃗_path := ϵ
4:      for i := 1 to |W_s| do
5:          if W_s(i) = [t] with t ∈ IL ∪ X then
6:              σ(t) := v'
7:              Vertex v'_next := v'
8:          else if W_s(i) = [⟨p, o⟩] and (v', p, o') ∈ Q_f then
9:              σ(o) := o'
10:             Vertex v'_next := o'
11:         else if W_s(i) = [⟨p^-1, s⟩] and (s', p, v') ∈ Q_f then
12:             σ(s) := s'
13:             Vertex v'_next := s'
14:         else if W_s(i) = [(] then
15:             m⃗_path.push(v')
16:             v' := v'_next
17:         else if W_s(i) = [)] then
18:             v' := m⃗_path.pull()
19:         else
20:             return False
21:     return True
```

vertices in W that leads to the anchor of the subgraph that is currently being examined. $\vec{m}_{\text{path}}$ denotes the corresponding vector of vertices in Q_f such that each element in $\vec{p}_{\text{ath}}$ is mapped to its corresponding element in $\vec{m}_{\text{path}}$ by the examined containment mapping. Vector $\vec{m}_{\text{path}}$ is implemented as a stack that allows to focus attention to different parts within the f-graph by pushing and pulling vertices of Q_f in it. Pushing is performed when examining a nested subgraph after an opening parenthesis, while pulling is performed when a nested subgraph has just been examined and its closing parenthesis has appeared.

Algorithm 2. The algorithm Containment checks if there exists a containment mapping for all triples in the query W with each triple in W being represented in W_s by a corresponding pair of terms. If there exists a containment mapping from W to Q_f, the algorithm will return True and σ is the corresponding containment mapping, otherwise it will return False. Checking is performed in the following steps: The algorithm initially defines a mapping σ that maps all IRIs and literals to themselves and is undefined for all variables in X (line 2) and creates an empty stack $\vec{m}_{\text{path}}$ (line 3). Then the algorithm proceeds to examine all the elements in W_s (line 4) considering different cases: *(i)* The first case appears when the anchor vertex t of W_s is examined, in that case the mapping σ is extended to $\sigma(t) := v'$ (line 6). *(ii)* The next case appears when a pair $\langle p, o\rangle$ is read in W_s with $p \in I$,

$o \in IL \cup X$ and a corresponding triple (v', p, o') also appears in Q_f. Then the mapping σ will be extended to $\sigma(o) := o'$ (line 9). *(iii)* The next case appears when a pair $\langle p^{-1}, o\rangle$ is read in W_s with $p \in I$, $o \in IL \cup X$ and a corresponding triple (s', p, v') also appears in Q_f. Then the mapping σ will be extended to $\sigma(s) := s'$ (line 12). *(iv)* It should be noted that in the three previous cases (lines 6, 9, 12), the algorithm will return False when t, s or o cannot be mapped to the corresponding term because they have already been mapped to another term. *(v)* When an opening parenthesis is met, a subgraph mapping has to be examined (lines 14 to 16). The subgraph W_s' appears between an opening and a closing parenthesis. The first step is to push the term that is being currently examined into the $\vec{m}_{\text{path}}$ stack (line 15). The next step is to tell the algorithm that we are interested in finding a mapping between W_s' and Q_f that maps the anchor variable of W_s' to v'_{next}. Thus, v' takes the variable of v'_{next} (line 16) and our algorithm checks containment for W_s'. *(vi)* Pushing the term v' into $\vec{m}_{\text{path}}$ (line 15) allows to continue examining v' when a containment mapping for the subgraph of W_s' is found and a closing parenthesis has been met (line 17). Then v' takes its previous value from $\vec{m}_{\text{path}}$ (line 18). *(vii)* The algorithm will return False in the cases that: $\langle p, o\rangle$ appears in W_s but no corresponding triple (v', p, o') appears in Q_f; $\langle p^{-1}, s\rangle$ appears in W_s but no corresponding triple (s', p, v') appears in Q_f. Finally, when all the elements of W_s have been examined and matched, the algorithm returns True (line 21).

Proposition 3.3. *For (i) a BGP query W with only IRI predicates, (ii) its serialised form W_s, (iii) W's anchor vertex v_a, (iv) an f-graph query Q_f, and (v) a vertex v'_a in Q_f: there exists a containment mapping σ from the variables of W to the variables of Q_f for which $\sigma(v_a) = v'_a$ applies iff the function* Containment(W_s, Q_f, v'_a) *returns* True.

Example 3.4. With W_s being the serialised query in Example 3.2, Q being the f-graph query in Example 2.1, and ?*sng* being a vertex in Q, executing Containment$(W_s, Q, ?sng)$ will be performed in the following steps. Initially a partial mapping σ is created so that $\sigma(?x) :=? sng$. Then, because of the opening parenthesis, the subgraph of ?x is examined. The algorithm finds that the pair $\langle$fromAlbum, ?$z\rangle$ in W_s is matched and sets $\sigma(?z) :=? alb$. Then because of the second opening parenthesis, the algorithm examines the subgraph of ?z and focuses on the vertex ?*alb* of Q. It finds that the pair $\langle$name, ?$w\rangle$ is matched and sets $\sigma(?w) :=? aN$. Next, reading the closing parenthesis will put focus on the anchor vertex ?x in W_s and the corresponding vertex ?*sng* in Q. The pair $\langle$name, ?$y\rangle$ will be matched by setting $\sigma(?y) :=? sN$. The algorithm finally reads the last closing parenthesis of W_s and σ is a containment mapping from W to Q.

Execution Time. To study the execution time of the algorithm for containment checking, we observe that every execution cycle of the algorithm examines a different element of W_s. Steps 8 and 12 of Algorithm 2 examine if a predicate or inverse predicate appears in the related triple patterns. By definition of an f-graph query, when examining a certain vertex v' within the f-graph, there exists at most one such triple pattern about v'. This step can be performed in $O(\log|Q_f|)$ time by building the appropriate index for incoming and outgoing edges, e.g., a red-black tree [21]. Thus the algorithm terminates in $O(|W|\cdot\log|Q_f|)$ time. To check for containment $Q_f \sqsubseteq W$ between an f-graph query Q_f and a BGP query W we need to apply the Containment(Q_f, W_s, v') function for every term v' appearing in Q_f. Thus, the time for checking containment is $O(|W|\cdot|Q_f|\cdot\log|Q_f|)$ in the worst case.

We should point out that the proof of the polynomial algorithm for query containment can be performed without the serialisation step. Intuitively, because of the strong requirements of the f-graph structure, once a variable v in W has been mapped to a term v' in Q, there is a single deterministic choice for the remaining variables appearing in W. Nevertheless, the serialisation step is required for inserting queries and containment checking into the mv-index structure presented in Section 4.

4 MV-INDICES

In the previous section, we showed how to efficiently check for containment between two queries. In the case that we want to check for containment between a single f-graph query Q_f and a set of BGP queries $\mathcal{W}$, it would be inefficient to make each and every comparison. For that reason, we have introduced the *"Materialised-View Index"* structure, denoted with *mv-index*, that allows to store a set of queries $\mathcal{W}$ and use it to check for containment. Our structure is based on *Radix trees*, ordered tree data structures that are used in string matching [52].

4.1 Mv-index

An mv-index $\mathfrak{M}$ is a tree structure (V, E, L) where: *(i)* V is a set of vertices; *(ii)* $E \subseteq V^2$ is a finite set of edges; *(iii)* L is a labelling function that maps each edge to a non-empty ordered list of distinct elements (IRIs, literals, and parenthesis symbols) and each vertex to the serialised form of an f-graph query; *(iv)* L_Q is another vertex labelling function: it takes the value of True when a vertex corresponds to an actual query inserted into $\mathfrak{M}$ and False when that vertex does not correspond to such a query and was rather created during the insertion procedure.

The intuition for this form of representation is that queries are represented by their serialised form in the mv-index structure, either as intermediate or leaf vertices, using the

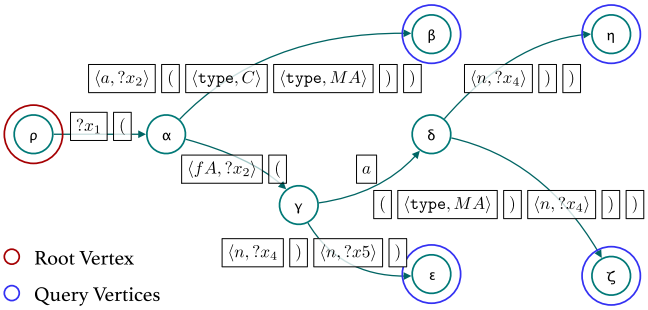


Figure 1: A simple mv-index.

labelling function L. For a vertex α in the mv-index structure, $L(\alpha)$ is its corresponding query in serialised form. The serialised form of the query represented by a vertex can be also obtained by following the path from the root of the mv-index to the specific vertex and concatenating the corresponding edge labels. Therefore, in our actual implementation we only store edge labels.

During the insertion phase, mv-indices are treated as regular Radix trees that instead of strings or numbers are used to represent queries in their serialised form. Therefore, instead of characters within a string, or digits within a number, mv-indices use IRIs, literals, variables and separators such as parenthesis symbols in order to represent serialised queries. More information on how insertion works in Radix trees can be found in the literature [52].

Example 4.1. Fig. 1 represents an mv-index. For space purposes we use the initial letters fA, n, a, MA, C to depict the IRIs *fromAlbum*, *name*, *artist*, *MusicalArtist*, and *Composer* respectively. The figure displays only edge labelings, vertex labelings can be inferred accordingly. The corresponding mv-index is used to represent 5 queries in total: Vertex ρ corresponds to an empty query and is the root of the mv-index. Vertex β corresponds to the serialised query

$$?x_1 \; (\; \langle \text{artist}, ?x_2 \rangle \; (\; \langle \text{type}, \text{Composer} \rangle \; \langle \text{type}, \text{MusicalArtist} \rangle \;) \;)$$

asking for some $?x_1$ (probably as song) that has an artist whom is both a composer and a musical artist. The labelling of vertex β is the concatenation of the labelings of the edges $L(\rho, \alpha), L(\alpha, \beta)$. Vertex ζ corresponds to a query that is similar to that presented in Example 3.2, but the name of the song is not asked. The labelling of vertex ζ is the concatenation of the labelings $L(\rho, \alpha), L(\alpha, \gamma), L(\gamma, \delta), L(\delta, \zeta)$. The other two vertices that are used to represent queries are η and ε.

4.2 Query Containment using Mv-indices

In order to check for query containment using mv-indices, we have devised an algorithm that takes advantage of the properties of an f-graph. In Algorithm 3, the ContQueries function takes as input *(i)* an mv-index $\mathfrak{M}$; *(ii)* a vertex α in the mv-index; *(iii)* an f-graph query Q_f; *(iv)* a term v' that appears in Q_f; *(v)* a term v'_{next} that appears in Q_f; *(vi)* the stack $\vec{m}_{\text{path}}$; *(vii)* and a partial mapping σ. The ContQueries function is used to acquire the set of vertices $V_{\text{cont}} \subseteq V$ such that: for every vertex $\zeta \in V_{\text{cont}}$ with $L(\zeta)$ representing the serialised form of a query W, it applies that $Q_f \sqsubseteq W$.

Algorithm 3 The algorithm for checking containment between queries within an mv-index and an f-graph query Q_f.

```
1:  function ContQueries(MVIndex 𝔐, Vertex α, FGraph-
    Query Q_f, Term v', Term v'_next, Stack m⃗_path, Mapping σ)
2:      for all β s.t. (α, β) ∈ E do
3:          λ⃗ := L(α, β)
4:          (toContinue, v', v'_next, m⃗'_path, σ)   :=
                    Containment(Q_f, λ⃗, v', v'_next, m⃗'_path, σ)
5:          if toContinue = True then
6:              V_cont := V_cont ∪
                ContQueries(𝔐, β, Q_f, v', v'_next, m⃗'_path, σ)
7:              if L_Query(β) = True then
8:                  V_cont := V_cont ∪ L(β)
9:      return V_cont
```

```
1:  function Containment(SerialisedForm W_s, FGraph Q_f,
    Term v', Term v'_next, Stack m⃗_path, Mapping σ)
2:      m⃗_path = CopyOf(m⃗_path)
3:      σ = CopyOf(σ)
        ▷ The rest of the function is identical to the Containment function in Algorithm 2
20:     return (True, v', v'_next, m⃗_path, σ)
```

Each path from the root ρ to a vertex γ of the mv-index such that $L_Q(\gamma) = \text{True}$ corresponds to the serialised form of some f-graph query W, with $L(\gamma)$ being the corresponding serialised form. When examining an mv-index path, on the transition from one vertex to another, the initial serialised form of the query is split up between the labelings of consecutive edges. Thus, when transitioning, we need to know what has happened so far. Therefore, we have made some minor changes in the Containment function of Algorithm 2 that are presented in Algorithm 3 (only the changed parts). The new version of the Containment function will return a quintuple of values. Along with the True value, it will return the terms v' and v'_{next} along with the $\vec{m}'_{\text{path}}$ stack and the σ partial mapping (line 20). The new version allows to transfer all the information that was conveyed in the previous steps to the next execution step of the algorithm.

Algorithm 3. Algorithm 3 presents the ContQueries procedure. For each vertex α of the mv-index, its corresponding serialised query is partially mapped for containment to Q_f.

The algorithm proceeds as follows: *(i)* If a vertex α in the mv-index has been partially matched for containment, then all of its child vertices should be examined for a containment mapping (lines 2 to 8). *(ii)* Initially, the vector $\vec{\lambda}$ takes the labelling of the corresponding edge (line 3). *(iii)* Subsequently, it is examined whether the corresponding edge violates the containment mapping or not (line 4). *(iv)* If there is a violation the ContainMent function will return (False, Null, Null, Null, Null) and the variable toContinue will take the value of False. Since the variable toContinue is False, we know that neither β nor any of its successors corresponds to a query containing Q_f and therefore they won't be further examined (line 5). *(v)* If, on the other hand, a containment mapping exists, the ContainMent function will return a quintuple of values that correspond to the current state of the mapping process. In such a case, the variable toContinue takes the value of True and the algorithm will also be applied for all the outgoing vertices of β (line 6). *(vi)* Additionally, if the vertex β corresponds to a query inserted into the mv-index (line 7), the corresponding vertex will be added to the vertices whose query contains Q_f (line 8). The algorithm terminates when there are no more vertices to be examined.

Theorem 4.2. *For the MVIndex $\mathfrak{M}$, its root vertex α, an f-graph query Q_f, a vertex v'_a appearing in the triple patterns of Q_f, an initially empty stack $\vec{m}_{path}$, and a partial mapping σ that maps each IRI and literal to itself, the execution of*

$$\textsc{ContQueries}(\mathfrak{M}, \alpha, Q_f, v'_a, \textsc{Null}, \vec{m}_{path}, \sigma)$$

will return every vertex ζ that appears in $\mathfrak{M}$ such that: $L(\zeta) = W_{s\zeta}$; $W_{s\zeta}$ being the serialised form of the query W_ζ; and a containment mapping $\sigma : W_\zeta \rightarrow Q_f$ exists such that $\sigma(v_a) = v'_a$ also applies, with v_a being the anchor vertex of $W_{s\zeta}$.

Based on the previous theorem, in order to find all the containment mappings, we need to call the ContQueries function for every vertex appearing in Q_f.

Optimisations. In order to reduce the size of the corresponding mv-index and the execution time of the ContQueries function, we have made the following optimisations: *(I)* When writing queries into their serialised form, we impose an ordering on $\langle r, o\rangle$ pairs where r is a predicate or an inverse predicate and $o \in IL \cup X$. The latter implies that if a vertex s has two outgoing edges (s, p_1, o_1) and (s, p_2, o_2) such that $p_1 \prec p_2$, when writing the serialised form of the subgraph of s, the pair $\langle p_1, o_1\rangle$ will appear before the pair $\langle p_2, o_2\rangle$. The corresponding total ordering $\prec$ may be based on the lexicographical order between IRIs and literals, or some other metric such as the frequency of their appearance within an RDF graph. *(II)* When inserting a serialised query W_s into the mv-index, our algorithm rewrites W_s's variables in such a way that the first variable appearing in W_s is mapped to $?x_1$, the second to $?x_2$, etc.. *(III)* We build a hash map from each mv-index vertex to its corresponding edges with IRIs, literals, and variables as key values.

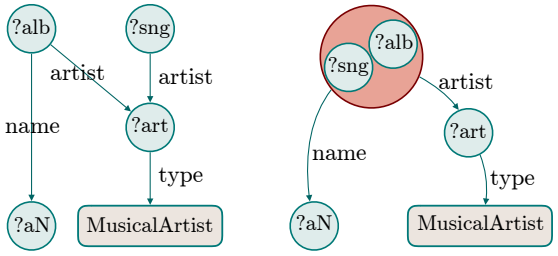
Optimisations (I) and (II) allow our algorithm to represent a set of queries in a compact form by revealing some of the patterns that are shared between multiple queries. The latter allows the same mv-index edge being used to encode triple patterns for multiple queries, making more efficient the search for containment mappings. E.g., in Figure 1, Example 4.1, the optimisations step ensures that the edges (ρ, α) and (α, γ) encode the triple $(?x_1, fromAlbum, ?x_2)$ that is shared in all three queries represented in the vertices η, ζ, and ε. The additional step of ordering elements in the serialised form can be performed in $O(n \log(n))$ on average and $O(n^2)$ on the worst case if we employ the Quicksort algorithm [21]. Optimisations (III) and (I) allow to quickly access specific edges of the mv-index, via hashing, that are meaningful for the part of the f-graph query currently being examined. E.g., if we have just examined a triple pattern (v', p, o') in Q_f, because of the ordering of elements within the serialised form of the queries inserted in the mv-index, we only need to examine the mv-index for triple patterns (v', p', o'') in Q_f such that $p \preceq p'$. Optimisation III additionally expedites insertion of serialised queries into the mv-index.

Execution Time. The worst case execution of our algorithm arises when all the queries in the mv-index can be mapped to the f-graph query Q_f. In each execution of the ContQueries we need to compare the f-graph Q_f against all the elements in $\mathfrak{M}$. The specific step will take $O(|\mathfrak{M}|\cdot \log|Q_f|)$ time to be executed, $|\mathfrak{M}|$ being the size of the mv-index. Since we have to repeat the process for all the terms that appear in Q_f, otherwise we may miss containments, the time needed to find all containments is $O(|\mathfrak{M}|\cdot|Q_f|\cdot \log|Q_f|)$.

The complexity of inserting a query into the mv-index is related to the complexity of inserting "*words*" into a Radix tree [52]. For a typical Radix Tree, that would be $O(|W_s|\cdot|\Sigma|)$ where W_s is the serialised form of the inserted query and Σ the alphabet for writing queries, i.e. the IRIs, literals, and variables appearing in the workload $\mathcal{W}$. Since our variation of the Radix Tree adopts a hash map for accessing mv-index edges, insertion time is performed in $O(|W_s|)$ on average and $O(|W_s|\cdot|\Sigma|)$ in the worst case.

5 INDEXING & CONTAINMENT FOR ARBITRARY BGP QUERIES

Sections 3 and 4 focus on solving the problem of query containment between an f-graph and a BGP query with only IRIs in the predicate position, i.e., $Q_f \sqsubseteq W$, with W either being a single query, or belonging to a set of queries $\mathcal{W}$. We will now discuss how to extend the existing structures to represent queries for which the two aforementioned restrictions do not apply. For each extension, we explain how



(a) Initial query **(b) F-graph witness**

Figure 2: A BGP query and its corresponding witnesses

the problem is solved for query containment between two queries and then describe how the mv-index structure needs to be extended in order to accommodate the aforementioned changes.

5.1 F-Graph Witnesses for BGP Queries

We first examine how to extend our algorithm for representing more expressive BGP queries in the left hand side of a query containment. In order to perform the specific task, we have introduced f-graph witnesses. The intuition is that each BGP query can be represented in the form of an f-graph when checking for query containment.

For a BGP query Q, its corresponding f-graph witness can be obtained by merging terms that violate conditions (i), (ii) in the definition of f-graph queries. To perform the aforementioned task, we initially define the equivalence relation $\sim$ on variables, IRIs, and literals in Q such that $o_1 \sim o_2$ when there exists a term s for which either the triple patterns (s, p, o_1) and (s, p, o_2) both appear in Q, or the triple patterns (o_1, p, s) and (o_2, p, s) both appear in Q. For a term s in Q, $[s]$ denotes its equivalence class on the $\sim$ relation that contains all the terms that are merged with s. Finding all equivalence classes is performed in polynomial time in the size of the graph (by reduction to the connected component problem). The f-graph witness Q_w of the query Q is obtained by replacing each triple pattern (s, p, o) in the body of Q with a triple pattern $([s], p, [o])$ where s, o are terms, $[s], [o]$ their corresponding equivalence classes, and p is a predicate. By construction, there is a unique witness for each query Q. It should be noted that for a containment mapping $\sigma : W \rightarrow Q_w$, Q_w being the f-graph witness of a query Q, each $a \in IL$ in W can be mapped only to its corresponding equivalence class $[a]$ in Q_w.

Proposition 5.1. *For a BGP query Q, its corresponding f-graph witness Q_w and a BGP query W, the following implication applies: $Q \sqsubseteq W \Rightarrow Q_w \sqsubseteq W$.*

Proposition 5.2. *For a BGP query Q, its corresponding f-graph witnesses Q_w and a BGP query W, for each containment mapping $\sigma : W \rightarrow Q$ for which $\sigma(s) = s'$ applies, there exists a containment mapping $\sigma_w : W \rightarrow Q_w$ such that $\sigma_w(s) = [s']$ applies.*

Containment Checking. What Proposition 5.1 conveys is that we need to check for containment $Q \sqsubseteq W$ only when the containment relation for the witness of Q is satisfied, i.e., $Q_w \sqsubseteq W$. The latter finding is of great importance for the following reason: checking $Q_w \sqsubseteq W$ can be performed in PTime as presented in Section 3.3, while checking $Q \sqsubseteq W$ is in the worst case a NP-complete problem. Therefore, we pay a PTime budget to solve specific instances of a NP-complete problem. The intuition is that we "postpone" non-deterministic checks that need to be performed in favour of a proof, computed in PTime, that $Q \sqsubseteq W$ does not apply. The non-deterministic check has to be performed only if there exists no such proof.

What Proposition 5.2 says is that every containment mapping $\sigma : Q \rightarrow W$ in NP can be inferred from a containment mapping $\sigma_w : Q_w \rightarrow W$ that is computed in PTime. It should be noted that each σ_w may result in more than one containment mappings σ. Suppose that $\mathcal{D}_{\sigma_w}$ is the domain of the mapping σ_w, a non-deterministic algorithm for finding every containment mapping σ from σ_w can be defined as follows. For each variable $?x \in \mathcal{D}_{\sigma_w}$, $\sigma_w(?x)$ is an equivalence class of variables, IRIs, and literals. A non-deterministic process chooses some arbitrary $s' \in \sigma_w(?x)$ and defines $\sigma(?x) := s'$. Each such mapping σ has additionally to be checked in PTime if it actually is a containment mapping. It should be noted, that the aforementioned procedure can be adjusted when W is an acyclic BGP so that containment is checked in PTime.

ND-Degree. With $|\cdot|$ denoting the elements within an equivalence class, we define the *non-determinism degree* (ND-degree) of a containment mapping from a query W to an f-graph witness Q_w as follows: $\prod_{?x \in \mathcal{D}_{\sigma_w}} |\sigma_w(?x)|$. The ND-degree is equal to the number of containment mappings that can result from σ_w. In a similar way, we may define the ND-degree of a query as the product of the sizes of all the equivalence classes that appear in its f-graph witness. Obviously f-graphs have a ND-degree that is equal to 1.

Example 5.3. Fig. 2b displays the f-graph witness corresponding to the query in Fig. 2a. The corresponding f-graph witness has a ND-degree that equals 2 since it contains exactly one equivalence class with two variables. When checking for a containment between the f-graph witness Q_w and the query W_s in serialised form:

$$\boxed{?x_1}\ \boxed{(}\ \boxed{\langle \text{artist}, ?x_2\rangle}\ \boxed{(}\ \boxed{\langle \texttt{type}, \text{MusicalArtist}\rangle}\ \boxed{)}\ \boxed{)}$$

Algorithm 2 will create in PTime the containment mapping σ_w: $\sigma_w(?x_1) = \{?alb, ?sng\}$, $\sigma_w(?x_2) = \{?art\}$. In order to acquire from σ_w the actual containment mapping(s) $\sigma : W \rightarrow Q$ we need to clarify the non-deterministic parts of the mapping σ_w (σ is the same for the other parts). For

the mapping of the variable $?x_1$ there are two alternatives based on σ_w, either $\sigma_1(?x_1) = ?alb$ or $\sigma_2(?x_1) = ?sng$. Since they both satisfy the triple pattern $(?x_1, \texttt{artist}, ?x_2)$ appearing in W_s, there exist two containment mappings σ_1, σ_2 such that: $\sigma_1(?x_1) = ?alb$, $\sigma_1(?x_2) = ?art$ and $\sigma_2(?x_1) = ?sng$, $\sigma_2(?x_2) = ?art$.

Containment Checking for mv-indices. While checking for containment for a query Q against an mv-index, in case it's not an f-graph, we first need to find its corresponding f-graph witness Q_w. Then, we find every W in the mv-index for which it applies that $Q_w \sqsubseteq W$. Finally, we compute whether $Q \sqsubseteq W$ actually applies based on Propositions 5.1, 5.2.

5.2 Unrestricted Predicates

This section focuses on how to extend the existing algorithms and structures to handle containments of the form $Q \sqsubseteq W$ for which a variable may appear in the predicate position of a triple pattern in W. The intuition is that we first solve the containment problem ignoring every triple pattern $(s, ?p, o)$ in W with a variable in its predicate position and then filter out solutions that do not satisfy the aforementioned triple patterns.

Containment Checking. The containment mapping is created in the following steps: *(i)* All triple patterns that have a variable in the predicate position are removed from the initial query W. The resulting query comprises of one or more subqueries $W_1, \ldots, W_n$ each corresponding to a different connected component in W. *(ii)* A connected component W_i is chosen and our algorithm finds every containment mapping $\sigma_i : W_i \rightarrow Q$. For every triple pattern $(s, ?p, o)$ with s, o both appearing in W_i the algorithm filters out solutions σ_i for which there exists a triple $(s, ?p, o)$ in W_i but no triple $(\sigma_i(s), p', \sigma_i(o))$ in Q. *(iii)* Then, the algorithm proceeds to examine a connected component W_j such that there exists a triple pattern $(s, ?p, o)$ in the initial W with s appearing in W_i and o appearing in W_j. For every containment mapping $\sigma_i : W_i \rightarrow Q$, we say that the mapping σ_i and the triple $(s, ?p, o)$ bound the mapping of o to one of the following values: $\{o' | (\sigma_i(s), p', o') \in Q\}$. A similar bounding occurs for every triple pattern $(s, ?p, o)$ in the initial W with o appearing in W_i and s appearing in W_j. *(iv)* Then, for every $\sigma_i : W_i \rightarrow Q$ we find every $\sigma_j : W_j \rightarrow Q$ that respects every bounding for σ_i. It is straightforward how to build a containment mapping combining σ_i and σ_j and our algorithm inductively builds a containment mapping that includes all the connected components in W.

Containment Checking for mv-indices. In order to embed the preceding algorithm into the mv-index structure, we need to encode the different connected components of a query W into the mv-index. This is easy to achieve since they can be represented as consecutive lists of elements within the mv-index structure. Additionally we encode the bounding information so that when an answer has been found for the i^{th} connected component of the query the variable mappings for the $(i+1)^{\text{th}}$ connected component are bounded accordingly.

6 MV-INDICES & RDFS REASONING

Our algorithm so far does not take into consideration the implicit information that can be inferred based on the terminological knowledge that is expressed in the form of an RDF Schema (RDFS) [14]. The problem of query containment becomes more complicated with the presence of class inclusions, property inclusions, domain and range restrictions.

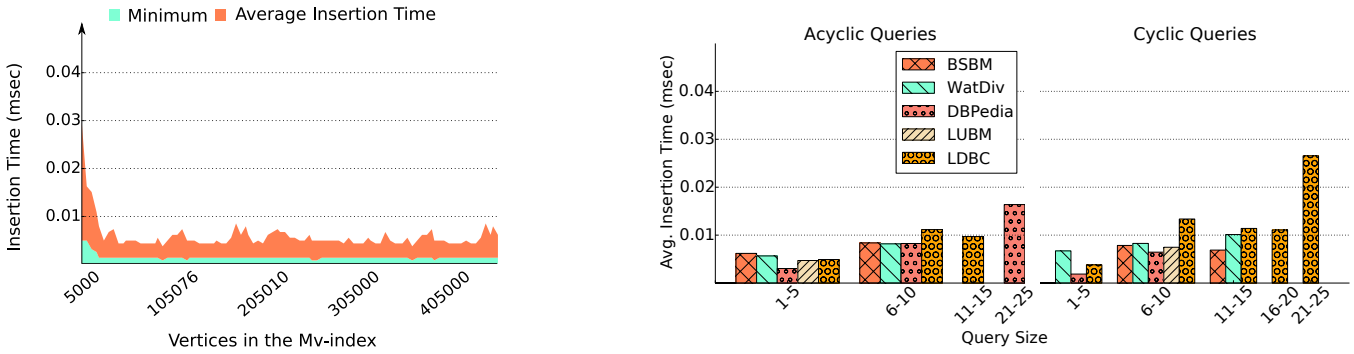
Our objective is to extend the corresponding algorithm for containment checking, without burdening the mv-index structure. This can be accomplished by introducing an additional step for containment checking. In order to check for containment $Q \sqsubseteq W$ between two queries, we first extend the query Q based on the semantic relationships that appear within an RDFS. This extension is performed by treating the variables in the query as if they were IRIs and reasoning is performed on the assertional knowledge that is extracted from the query. Then, we add the intensional knowledge that is acquired through the former reasoning step. Example A.1 in Appendix A shows how the previous algorithm works in practice. Additionally, Proposition 6.1 allows to utilise Algorithm 4.2 for finding containments into the mv-index structure by replacing Q with its extended form:

Proposition 6.1. *The query containment $Q \sqsubseteq_{\mathcal{R}} W$ applies w.r.t. to the RDF schema $\mathcal{R}$, iff there exists a containment mapping from W to the extended form of the query Q.*

7 EXPERIMENTAL EVALUATION

The aim of our evaluation section is to examine the performance of mv-indices during the insertion and query containment phases (Section 4). For the insertion scenario, the application takes as input a query workload $\mathcal{W}$ and produces the corresponding mv-index that encodes all queries in $\mathcal{W}$. For the containment testing scenario, the application takes as input an mv-index data structure that encodes a workload $\mathcal{W}$ and a query Q. The application will return every query $W \in \mathcal{W}$ such that $Q \sqsubseteq W$. In a practical application, $\mathcal{W}$ would comprise of all the views that have been materialised or all the queries whose results have been cached, while every $W \in \mathcal{W}$ such that $Q \sqsubseteq W$ would be a candidate for rewriting the query Q using W.

Hardware and memory. We deployed our implementation on a 2010 MacBook laptop, running on macOS High Sierra, having a single 2.66 GHz Intel Core i7 processor with 2 cores,



(a) Insertion time w.r.t. the mv-index size

(b) Insertion time w.r.t. query size (number of triple patterns)

Figure 3: Query Insertion Evaluation

and 8 GB of main memory. A single core was used during the experimental evaluation.

Implementation Setup. We have implemented our algorithm in Java 8 using the Apache Jena 3.6.0 open source Semantic Web framework [35] to parse SPARQL query workloads. In our experimental evaluation, when checking insertion and containment time, we have excluded the time Apache Jena needs to parse each BGP query.

Benchmarks. We used 5 different benchmarks for the evaluation of our implementation: (i) a real-world query workload[2] originating from the *DBpedia semantic knowledge graph* [8] containing 1,287,711 BGP queries; (ii) a synthetic query workload[3] originating from the *WatDiv SPARQL diversity test suite* [4] containing 148,800 generated BGP queries; (iii) a synthetic query workload[4] originating from the *Berlin SPARQL Benchmark (BSBM)* [9] containing 99,800 generated BGP queries; (iv) a synthetic query workload originating from the *Lehigh University Benchmark (LUBM)* [28] containing 14 BGP queries; (v) a query workload[5] originating from the *LDBC social network benchmark* [24] containing 53 BGP queries. It should be noted that the queries created by the BSBM query generator are based on a variation of 12 basic query patterns, while the queries produced by WatDiv are not based on specific patterns. The 5 datasets contain in total 1,536,708 queries of which 1,071,826 are f-graph and acyclic queries, 378,884 are acyclic queries (but not f-graph queries), 67,340 are f-graph queries (but not acyclic queries), and 18,658 are BGP queries that are neither acyclic, nor f-graphs.

[2]https://github.com/AKSW/SPARQL2NL/tree/master/resources/dbpediaLog
[3]http://dsg.uwaterloo.ca/watdiv/stress-workloads.tar.gz
[4]https://github.com/h31nr1ch/the-berlin-benchmark/blob/master/consults/rdf.sql
[5]https://github.com/ldbc/ldbc_snb_implementations

7.1 Insertion Cost

We first examine how inserting queries into the mv-index structure is affected by: the size of the mv-index structure and the size and the characteristics of the query. We performed insertions to the mv-index using all the queries from the 5 query workloads (1,536,378 queries in total). The time to insert all queries was 7.425 secs, while the insertion process resulted in an mv-index with a total of 466,576 intermediate vertices, containing a total of 397,507 distinct queries. This is attributed to the fact that recurring queries appear within the 5 workloads. Query insertion takes on average 0.0028 msec, 0.0098 msec, 0.0065 msec, 0.0070 msec, and 0.0072 msec for the DBPedia, LDBC, WatDiv, BSBM, and LUBM query workloads, respectively.

Mv-index Size. In Fig. 3a we examine the query insertion time w.r.t. the number of vertices in the mv-index structure. The x-axis measures the vertices in the mv-index structure, while the y-axis depicts the average and the minimum time needed to insert each query. Since our mv-index structure has almost half a million vertices, we measure the average and the minimum insertion time per 5,000 vertices. It should be noted that not all insertions change the size of the structure, since they may correspond to queries that are already represented in the mv-index. On analysing Fig. 3a we observe that there is not an apparent increase in insertion time w.r.t. the size of the mv-index structure. We also observe that insertion is slower during its initial phases. This is attributed to the fact that many changes occur during the initial phases that include the addition of vertices and edges into the mv-index and the corresponding changes in the internal structures (such as hash maps) of our implementation.

Query Size. In our second experiment we observe that there exists a more explicit relation between the query size, measured by the number of triple patterns within a query, and its insertion time into the mv-index structure. Fig. 3b displays

the average insertion time (y-axis) for different query sizes (x-axis), for the 5 different query workloads, and for acyclic or cyclic BGP queries. We observe that query insertions are really fast, insertion time scales almost linearly w.r.t. query size, as expected from the theoretical analysis presented in Section 4.2.

7.2 Containment Cost

We now examine the query containment time for different query parameters such as the size and the ND-degree of the query. In our experimental analysis we will consider the mv-index of the previous section that contains information from all the 5 query workloads that were described. The containment checking problem for an mv-index and a single query Q returns every query W that appears as a vertex in the mv-index such that $Q \sqsubseteq W$.

In our experimental evaluation, we consider containment time for different types of queries: f-graph & acyclic queries; f-graph & cyclic queries; non-f-graph & acyclic queries; non-f-graph & cyclic queries. It should be noted that for handling non-f-graph queries we employed the algorithm described in Section 5.1. We additionally compute the average time for containment w.r.t. the five different query workloads. The average time for query containment is 0.0092 msec for queries in the DBPedia workload, 0.0127 msec for queries in the WatDiv workload, 0.0166 msec for queries in the BSBM workload, 0.0409 msec for queries in the LDBC workload, and 0.0103 msec for queries in the LUBM workload.

Query Size. Fig. 4 displays the relation between the size of a query Q and the time to check for containment. The size of each query is measured as the number of triple patterns that appear in it and the average time is measured in milliseconds. The average containment time is computed for multiple queries of similar characteristics, therefore in the bar chart of Fig. 4 we display the 95%-confidence interval [55] along with each measurement. In the average case, we observe that the query containment time increases with the size of the query. We also observe that the average containment time for queries of similar sizes tends to increase for non-f-graph queries. This is attributed to the fact that containment checking is computed in PTime for f-graph queries. Additionally, acyclic queries need less time to be processed compared to cyclic queries with similar characteristics.

ND-degree. Figure 5 depicts how the query containment operation is affected by the ND-degree of a query. We have a figure for acyclic queries and one for cyclic queries. By definition of the ND-degree, we have that queries with a ND-degree of 1 are also f-graphs and can be answered in PTime, while queries with ND-degree greater than 1 are arbitrary queries that can be answered in NP with the algorithm presented in Section 5.1. It is evident from Figure 5 that the complexity of finding containments for a BGP query increases along with its ND-degree.

RDFS Reasoning. In the last part of our experimental evaluation, we examine how the RDFS knowledge differentiates the problem of query containment. We used the LUBM query workload, since LUBM is the only benchmark associated with RDFS knowledge. Since the original LUBM query workload is constituted of only 14 basic queries, we extended the initial workload to one containing 1,000 queries. The extension was performed as follows: *(i)* each triple of the form $(s, \texttt{type}, A)$ either remains unchanged, or is replaced with a triple $(s, \texttt{type}, A')$ with A' being a superclass or a subclass of A; *(ii)* each triple of the form (s, p, o) either remains unchanged, or is replaced with a triple (s, p', o) with p' being a superproperty or a subproperty of r; *(iii)* for each (s, p, o) triple within a query, the query generator may create additional triples based on domain and range restrictions within the RDFS. The extended workload ensures that in order to correctly answer to the containment problem, our algorithm needs to take into account the RDFS knowledge and extend the queries as described in Section 6.

Fig. 6a presents how the performance of the containment algorithm is affected by the existence of an RDF schema. The x-axis displays the size of the query, while the y-axis displays the average time needed to find all containments for the case that the query under examination remains unchanged (LUBM), or is extended according to the RDFS (LUBM extended). It should be noted, that in the case that Q is not extended, the containment algorithm will result to an incomplete solution, i.e. we miss some implicit containments. We observe that the containment time increases w.r.t the query size. This is attributed to the fact that, for complicated ontologies, the size of the extended query increases significantly. Moreover, the inference process, may result to some of the queries losing their f-graph properties, thus, making them harder to process. Finally, the number of answers to the containment problem is increased as well. The latter effect is evident in Figure 6b that displays the amortised cost over the number of queries W that appear into the mv-index and also contain the query under examination Q. It should be noted that for each initial query Q the algorithm ContQueries finds on average 2.553 queries W s.t. $Q \sqsubseteq W$, while for the extended form of the query, the algorithm ContQueries finds on average 29.513 such queries. Thus, we observe that the amortised cost actually decreases for LUBM's extended form. This is due to the mv-index's ability to check for multiple containments when checking a single mv-index's edge.

8 RELATED WORK

Our work is related to several fields of the Database and Semantic Web communities:

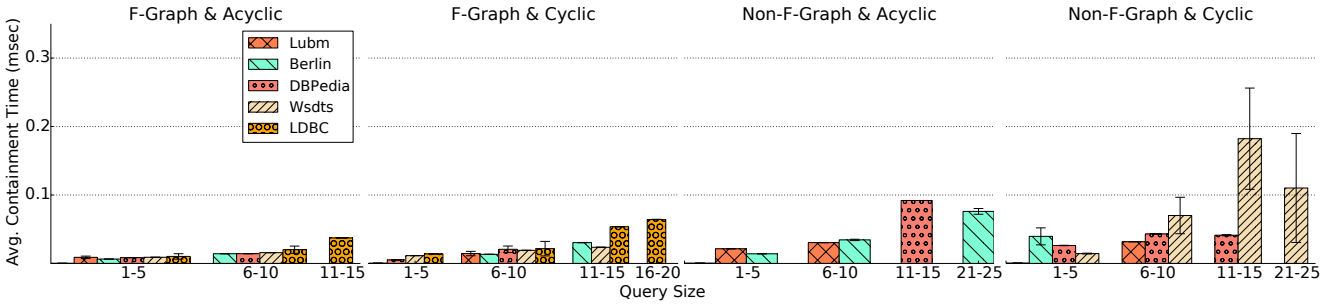


Figure 4: Containment Cost w.r.t. Query Size (number of triple patterns)

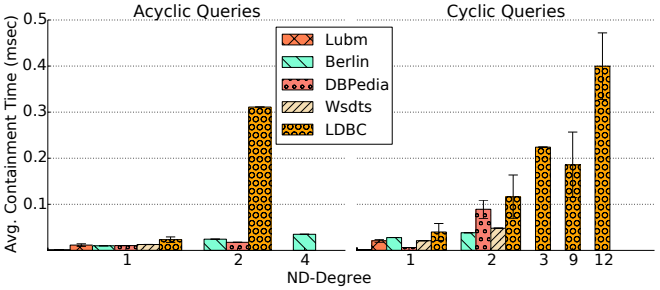
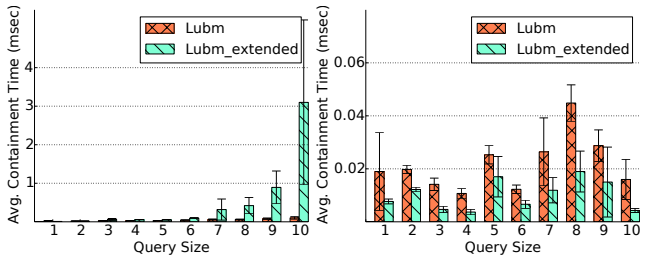


Figure 5: Containment Cost w.r.t. ND-Degree



(a) Overall Cost **(b) Amortised Cost**

Figure 6: Containment Cost for Extended Queries

RDF Stores. Much research effort has been invested in the development of scalable centralised or distributed RDF stores, techniques for indexing RDF data and for processing SPARQL queries. Among the centralised approaches, native RDF stores like Jena [48], Sesame [15], HexaStore [72], SW-Store [1], MonetDB-RDF [64], RDF-3X [54], and BitMat [7] have been carefully designed to keep up pace with the growing scale of RDF collections. Systems like TriAD [30], RDFox [53], H-RDF-3X [33], EAGRE [75] implement various optimisations for the distributed execution of joins. Mv-indices can be exploited to accelerate query processing by building optimisations based on view materialisations that benefit from mv-indices. Additionally they can be combined with structural indexes [49, 68] to accelerate query containment between an incoming query and the existing RDF patterns that reside within the indexes.

Query Containment. The mv-index structures are immediately related to the query containment problem that has been extensively studied by the Database community. Over the years, the problem of query containment under set [17, 20, 36, 42, 62] and bag semantics [2, 18, 34] has been investigated in depth by many researchers. Restricted forms of conjunctive queries that ensure the polynomial complexity of the query containment problem have also been studied w.r.t. set and bag semantics [2, 20, 36].

The results of the query containment problem can be transferred to SPARQL by proving its reducibility to relational algebra expressions. The equivalences of SPARQL to Relational Algebra and its relation with Datalog with negation as failure are studied in [5, 60]. In [63], query containment and equivalence are studied w.r.t. the RQL [37] query language. The problem of SPARQL query containment under the RDFS entailment regime is studied in [19]. The problem is reduced to the expressive logic of μ-calculus and a 2EXPTIME upper complexity bound is proved. With a similar methodology, the problem of SPARQL query containment under $\mathcal{SHI}$ ontologies is also proved to have a 2EXPTIME upper complexity bound [73]. A complexity analysis of containment and equivalence for several fragments of the SPARQL language, based on different SPARQL operators, is performed in [44, 59] and NP-complete to undecidable results are proved for different SPARQL fragments. Our work complements past work in this area by proposing an index that allows to simultaneously compute from a set of queries the subset that contains a specific query Q and can be used for the query's rewriting.

The f-graph queries studied here and the acyclic queries studied in [20, 27, 74] are two *different* classes of queries for which the problem of query containment is tractable based on *different* mechanisms. Firstly, f-graph queries have a restricted form of incoming and outgoing edges which is independent of acyclicity. Secondly, for the queries Q and W, the containment problem $Q \sqsubseteq W$ can be solved in polynomial time when *(i)* either Q is an f-graph query and W belongs to the class of BGP queries that have only IRIs as predicates; *(ii)* or W is an acyclic Boolean query — for this case, Gottlob et al. [27] have proved that the problem is LOGCFL complete. Therefore, the two classes of queries are complementary and allow to solve different instances of the query containment problem in PTime. The ND-degree (Section 5.1) and the

query width studied in [20] express the *deviation* from the ideal query type: *(i)* f-graph queries have a ND-degree of 1, while non-f-graph queries have a ND-degree greater than 1; *(ii)* acyclic queries have a query width of 1, while cyclic queries have a query width greater than 1. However, the ND-degree is a *different measure* than the query width and there exists no dependence between the two. I.e., there exist queries with a ND-degree of 1 and a query width greater than 1 and vice versa. Furthermore, *"it is open whether there is a polynomial-time algorithm for finding the query width"* [20], while we show that the ND-degree of a query can be computed in linear time. An interesting open problem is whether the algorithms in [20] for containment checking, exploiting the acyclicity of queries, can be extended into an indexing structure for computing multiple containment mappings.

F-graph queries for RDF data are related to the class of fan-out free queries for relational databases [36]. The main intuition for both types of queries is that the containment mapping from a query W to a query Q can be determined from the containment mapping between their corresponding parts. For fan-out queries this means that there can exist a single containment mapping that maps a conjunct in W to a conjunct in Q, while for f-graph queries there can exist at most one containment mapping that maps a variable in W to an term in Q. The main differences between the two formalisms are that: the fan-out free property refers to pairs of queries rather than being a property of a single query as for f-graph queries; fan-out queries focus on the problem of query equivalence (not query containment) and therefore they do not allow containment mappings from variables to constants (IRIs or literals in RDF terminology).

SPARQL Workload Analysis. In Section 3, we studied the DBPedia query workload w.r.t. the f-graph property. For a detailed analysis of SPARQL queries, the reader may refer to the existing bibliography. The SPARQL-query workload of DBPedia is studied in [58] and an analysis of the different SPARQL operators that appear within DBPedia queries is performed. For various workloads, the structural characteristics related to the graph and hypergraph representation of SPARQL queries are studied in [12], along with the evolution of SPARQL queries over time. Finally, a study of the Wikidata knowledge graph is presented in [47], while the different SPARQL operators appearing in Wikidata's workload are also analysed.

View Materialisation & Caching. View Materialisation and Caching techniques have been extensively studied by relational databases [3, 29, 45, 45, 50] and data warehouses [32, 43, 51]. They have recently gained attention by the Semantic Web community and graph data systems. In [23], an approach for the materialisation of shortcuts that reduces the execution cost of path queries is suggested. In [26], a different materialisation strategy where an initial query workload $\mathcal{W}$ is transformed to a set of simpler views $\mathcal{V}$ along with a set of rewritings is presented. In [56], a strategy that caches SPARQL-query results and uses them to rewrite queries is studied. Caching strategies for graph query processing have been studied in [69–71]. The caching algorithms in [56] and [69–71] are based on finding subgraph-isomorphisms between incoming and cached queries. The approach in [56] is based on a canonical labelling algorithm, while [69–71] adopt a *filter then verify* strategy where candidate graphs for isomorphism are filtered out based on certain features and then the actual test for isomorphism is performed, e.g. [25, 41]. It should be noted that subgraph-isomorphism cannot be used for solving the query containment problem since it would provide for an incomplete solution. For example, if we have the indexed BGP $\{(?x, r_1, ?y), (?y, r_2, ?z)\}$ and want to use it for answering the BGP $\{(?x', r_1, ?y'), (?y', r_2, ?x')\}$, our algorithm will find the containment mapping $\sigma(?x) = ?x'$, $\sigma(?y) = ?y'$, $\sigma(?z) = ?x'$, while it can be checked that there exists no corresponding subgraph isomorphism. The mv-index structure we propose is complementary to the existing caching systems and techniques and can be used to improve their performance.

9 CONCLUSIONS AND FUTURE WORK

Our study introduces f-graph queries and demonstrates that the containment problem $Q_f \sqsubseteq W$ between an f-graph query and a BGP query can be solved in PTime. We also present the mv-index structure, a novel indexing structure for BGP queries, that allows to check containment between an f-graph query and a set of queries in worst case linear time w.r.t. the size of the indexed queries. Finally we show how to apply our algorithm for much more expressive queries with the introduction of f-graph witnesses, i.e. graphs that can substitute an arbitrary query within the mv-index structure. In our experimental evaluation we showed that containment in practice runs much faster since most queries are not as complicated as the worst case analysis assumes.

In future work, we plan to examine how mv-indices can be exploited for view materialisation and query caching applications and to extend mv-indices for arbitrary queries on relational databases. Additionally, we intend to examine mv-indices and view materialisation techniques for providing semantic access to combined streaming and static information [38–40, 67], enhancing performance of end-user oriented query interfaces [6, 65], and study if our indexing structures can be extended for more expressive formalisms that introduce uncertainty to the problem of query answering and containment [13, 16, 46]. Exploiting the mv-index structure for graph indexing purposes [10, 11, 66] is another promising direction for future work.

REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.

[2] F. N. Afrati, M. Damigos, and M. Gergatsoulis. Query containment under bag and bag-set semantics. *Information Processing Letters*, 110(10):360–369, 2010.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[4] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *ISWC*, pages 197–212, 2014.

[5] R. Angles and C. Gutierrez. The expressive power of sparql. *ISWC*, pages 114–129, 2008.

[6] M. Arenas, B. C. Grau, E. Kharlamov, S. Marciuska, and D. Zheleznyakov. Faceted search over rdf-based knowledge graphs. *J. Web Sem.*, 37-38:55–74, 2016.

[7] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50. ACM, 2010.

[8] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. *ISWC/ASWC*, pages 722–735, 2007.

[9] C. Bizer and A. Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[10] D. Bleco and Y. Kotidis. Graph analytics on massive collections of small graphs. In *EDBT*, pages 523–534. Citeseer, 2014.

[11] D. Bleco and Y. Kotidis. Using entropy metrics for pruning very large graph cubes. *Information Systems*, 81:49–62, 2019.

[12] A. Bonifati, W. Martens, and T. Timm. An analytical study of large sparql query logs. *PVLDB*, 11(2):149–161, 2017.

[13] S. Borgwardt, T. Mailis, R. Peñaloza, and A.-Y. Turhan. Answering fuzzy conjunctive queries over finitely valued fuzzy ontologies. *Journal on Data Semantics*, 5(2):55–75, 2016.

[14] D. Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. *https://www.w3.org/TR/rdf-schema/*, 2004.

[15] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, pages 54–68, 2002.

[16] B. Cautis and E. Kharlamov. Answering queries using views over probabilistic XML: complexity and tractability. *PVLDB*, 5(11):1148–1159, 2012.

[17] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*. ACM, 1977.

[18] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*, pages 59–70. ACM, 1993.

[19] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. Sparql query containment under rdfs entailment regime. In *IJCAR*, 2012.

[20] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *ICDT*, pages 56–70, 1997.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[22] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan, et al. Semantic data caching and replacement. *VLDB*, 96:330–341, 1996.

[23] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Optimizing query shortcuts in rdf databases. *ESWC*, pages 77–92, 2011.

[24] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630. ACM, 2015.

[25] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS one*, 8(10):e76911, 2013.

[26] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(2):97–108, 2011.

[27] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[28] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Semant.*, 3(2-3):158–182, 2005.

[29] A. Gupta, I. S. Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[30] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, pages 289–300. ACM, 2014.

[31] C. Gutierrez, C. A. Hurtado, A. O. Mendelzon, and J. Pérez. Foundations of semantic web databases. *JCSS*, 77(3):520–541, 2011.

[32] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25:205–216, 1996.

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

[34] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *TODS*, 20(3):288–324, 1995.

[35] A. Jena. semantic web framework for java, 2007.

[36] D. S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM Journal on Computing*, 12(4):616–640, 1983.

[37] G. Karvounarakis, A. Magganaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. Querying the semantic web with rql. *Computer networks*, 42(5):617–640, 2003.

[38] E. Kharlamov, S. Brandt, E. Jiménez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. L. Özçep, C. Pinkel, C. Svingos, D. Zheleznyakov, I. Horrocks, Y. E. Ioannidis, and R. Möller. Ontology-based integration of streaming and static relational data with optique. In *SIGMOD*, pages 2109–2112, 2016.

[39] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, Y. E. Ioannidis, S. Lamparter, and R. Möller. Towards analytics aware ontology based access to static and streaming data. In *ISWC*, pages 344–362, 2016.

[40] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. Özçep, C. Svingos, D. Zheleznyakov, Y. Ioannnidis, S. Lamparter, R. Möller, and A. Waaler. An ontology-mediated analytics-aware approach to support monitoring and diagnostics of static and streaming data. *J. Web Semant.*, 2019.

[41] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *ICDE*, pages 1115–1126, 2011.

[42] A. Klug. On conjunctive queries containing inequalities. *JACM*, 35(1):146–160, 1988.

[43] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *TODS*, 26(4):388–423, 2001.

[44] A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *TODS*, 38(4):25, 2013.

[45] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views. In *PODS*, pages 95–104. ACM, 1995.

[46] T. Mailis, G. Stoilos, and G. Stamou. Expressive reasoning with horn rules and fuzzy description logics. *Knowledge and information systems*, 25(1):105–136, 2010.

[47] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In *ISWC*, pages 376–394, 2018.

[48] B. McBride. Jena: Implementing the rdf model and syntax specification. In *ISWC*, pages 23–28. CEUR-WS. org, 2001.

[49] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. Extended characteristic sets: graph indexing for sparql query optimization. In *ICDE*, pages 497–508, 2017.
[50] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, volume 30, pages 307–318. ACM, 2001.
[51] K. Morfonios, S. Konakas, Y. Ioannidis, and N. Kotsis. Rolap implementations of the data cube. *ACM Computing Surveys (CSUR)*, 39(4):12, 2007.
[52] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *JACM*, 15(4):514–534, 1968.
[53] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. Rdfox: A highly-scalable rdf store. In *ISWC*, pages 3–20, 2015.
[54] T. Neumann and G. Weikum. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1-2):256–263, 2010.
[55] J. Neyman. X—outline of a theory of statistical estimation based on the classical theory of probability. *Phil. Trans. R. Soc. Lond. A*, 236(767):333–380, 1937.
[56] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware, workload-adaptive sparql query caching. In *SIGMOD*, pages 1777–1792. ACM, 2015.
[57] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *ISWC*, pages 30–43, 2006.
[58] F. Picalausa and S. Vansummeren. What are real sparql queries like? In *SWIM*, page 7. ACM, 2011.
[59] R. Pichler and S. Skritek. Containment and equivalence of well-designed sparql. In *PODS*, pages 39–50. ACM, 2014.
[60] A. Polleres. From sparql to rules (and back). In *WWW*, pages 787–796. ACM, 2007.
[61] E. Prud, A. Seaborne, et al. Sparql query language for rdf. *https://www.w3.org/TR/rdf-sparql-query/*, 2006.
[62] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27(4):633–655, 1980.
[63] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *ISWC*, 2005.
[64] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
[65] A. Soylu, E. Kharlamov, D. Zheleznyakov, E. Jiménez-Ruiz, M. Giese, M. G. Skjæveland, D. Hovland, R. Schlatte, S. Brandt, H. Lie, and I. Horrocks. Optiquevqs: A visual query system over ontologies for industry. *Semantic Web*, 9(5):627–660, 2018.
[66] V. Spyropoulos and Y. K. Kotidis. Digree: Building a distributed graph processing engine out of single-node graph database installations. *ACM SIGMOD Record*, 46(4):22–27, 2018.
[67] C. Svingos, T. Mailis, H. Kllapi, L. Stamatogiannakis, Y. Kotidis, and Y. Ioannidis. Real time processing of streaming and static information. In *IEEE Big Data*, pages 410–415, 2016.
[68] T. Tran, G. Ladwig, and S. Rudolph. Managing structured and semistructured rdf data using structure indexes. *IEEE Transactions on Knowledge and Data Engineering*, 25(9):2076–2089, 2013.
[69] J. Wang, Z. Liu, S. Ma, N. Ntarmos, and P. Triantafillou. GC: A graph caching system for subgraph/supergraph queries. *PVLDB*, 11(12):2022–2025, 2018.
[70] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *EDBT*, pages 41–52, 2016.
[71] J. Wang, N. Ntarmos, and P. Triantafillou. Graphcache: A caching system for graph queries. In *EDBT*, pages 13–24, 2017.
[72] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
[73] M. Wudage, J. Euzenat, P. Genevès, and N. Layaıda. Sparql query containment under shi axioms. In *Proceedings 26th AAAI Conference on Artificial Intelligence*, pages 10–16, 2012.
[74] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.
[75] X. Zhang, L. Chen, Y. Tong, and M. Wang. Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In *ICDE*, 2013.

A PROOFS & EXAMPLES

Proof of Proposition 3.3. For convenience, we will use s, t, v to denote terms appearing in the serialised form of a BGP query W_s and its initial query W, while s', t', v' will denote terms appearing in the f-graph Q_f.

Soudness. Because of nested subgraphs, we will show the soundness of the proposition by induction in the structure of the serialised form of W_s. In the rest of the proof, σ is the mapping being created by the Containment function in Alg. 2.

> Claim ♠. We first need to prove that if *(i)* the function Containment is examining the i^{th} element of W_s; *(ii)* W_{nest} is the nested subgraph appearing from the i^{th} to the j^{th} position of W_s; *(iii)* v' is the vertex in Q_f currently being examined; *(iv)* v is the anchor vertex of W_{nest}; *(v)* $\sigma(v) = v'$ applies; *(vi)* and the Containment function does not return False during its execution from position i to j then σ is a containment mapping $\sigma : W_{\text{nest}} \rightarrow Q_f$.

We will prove the claim by induction in the structure of nested subgraphs.

Induction Basis. For the induction basis we need to show that the claim applies when there are no parenthesis symbols within the elements of W_{nest}. For the previous case, the "substring" from the i^{th} to the j^{th} position of W_s has the form:

$$\boxed{\langle r_i, t_i \rangle} \ \ldots \boxed{\langle r_j, t_j \rangle} \tag{3}$$

with r_l either being a predicate p_l or an inverse predicate p_l^{-1}. In the previous formula, all predicate assertions are related to the same anchor term v by construction of W_s. Therefore $\boxed{\langle r_l, t_l \rangle}$ corresponds to the triple (v, p_l, t_l) in W_{nest} if r_l is a predicate p_l or to the triple (t_l, p_l, v) in W_{nest} if r_l is p_l^{-1}. Since the Containment function does not return False, every triple (v, p_l, t_l) in W_{nest} is mapped by the constructed σ to a triple (v', p_l, t_l') in Q_f (line 9 of Alg. 2) and every triple (t_l, p_l, v) is mapped to a triple (t_l', p_l, v') in Q_f (line 12 of Alg. 2). Since every triple pattern in W_{nest} is mapped to a triple pattern in Q_f via σ, σ is a containment mapping from W_{nest} to Q_f.

Induction Step. We now extend the proof for the case that sequences of the form $\boxed{\langle p_l, t_l \rangle}\ \boxed{(}\ W_{\text{nest2}}\ \boxed{)}$ appear in W_{nest} of Equation 3 where p_l is a predicate IRI and W_{nest2} is a nested subgraph in W_{nest} with t_l its corresponding anchor vertex. By construction of W_s, $\boxed{\langle p_l, t_l \rangle}$ corresponds to a triple (v, p_l, t_l)

in W_{nest} and it is shown same way as before that the triple is mapped by σ to a triple in Q_f. When reading the opening parenthesis, the vertex v' of Q_f is pushed in $\vec{m}_{\text{path}}$ and v', the currently examined vertex of Q_f, will take the value of $\sigma(t_l)$ (lines 15 and 16). By construction of the serialised form of W_{nest2}, t_l is its corresponding anchor vertex while it is also mapped to the currently examined element of Q_f. Therefore the premises in Claim ♠ and the induction hypothesis ensure that σ is also a containment mapping $\sigma : W_{\text{nest2}} \rightarrow Q_f$. The closing parenthesis (line 18) ensures that when all the elements of W_{nest2} have been checked, v' will take its previous value and the rest of the W_{nest} will be correctly mapped as before. In a similar way we prove the induction step in the case that sequences with inverse predicates: $\boxed{\langle p_l^{-1}, t_l\rangle}$ $\boxed{(}$ W_{nest2} $\boxed{)}$ appear in W_{nest}.

Our algorithm is sound, because the first element in W_s is its anchor vertex v_a, and the rest of the graph information is encoded after an opening parenthesis. Since the algorithm does not return False when examining the first element of W_s and $\sigma(v_a) = v'_a$, because the premises of Claim ♠ also apply, the constructed σ will be a containment mapping and the algorithm is sound.

Completeness. For the opposite direction, suppose that there exists a containment mapping μ from the variables of W to the variables of Q_f for which $\mu(v_a) = v'_a$, we need to show that the Containment(W_s, Q_f, v'_a) will return True.

The soundness of the containment algorithm along with the f-graph properties ensure that during the i^{th} execution step of Alg. 2, the constructed mapping σ agrees with the containment mapping μ. This means that for every variable $?x$ such that $\sigma(?x)$ is defined, it applies $\sigma(?x) = \mu(?x)$. In order to complete our proof, we need to show that none of the lines that return False will be accessed.

▷ If line 20 was accessed for a pair $\langle p, o\rangle$ in W_s, there should exist a triple (v, p, o) in W with $\sigma(v)$ already been defined and no corresponding triple $(\sigma(v), p, o')$ appearing in Q_f. The latter cannot apply because of the existence of the containment mapping μ and the fact that $\sigma(v) = \mu(v)$. With a similar argumentation, line 20 cannot be accessed for pairs of the form $\langle p^{-1}, s\rangle$ in W_s.

▷ The Containment function cannot return False in line 6 since that would imply that $t \in IL$ and $t \neq v'$. The latter cannot occur because of the existence of the containment mapping μ.

▷ The Containment function cannot return False in line 8 since that would imply that the triple (v, p, o) appears in W; $\sigma(v) = v'$; the triple $(v', p, o') \in Q_f$; and $\sigma(o) \neq o'$. The latter cannot apply because σ agrees with μ, i.e. $\sigma(v) = \mu(v')$, $\sigma(o) = \mu(o)$, and the fact that μ is a containment mapping. With a similar argumentation, the Containment function cannot return False in line 12. □

Proof of Theorem 4.2. In order to prove the soundness and completeness of the theorem, we first need to show the following claim:

> *Claim* ♣. We assume that W_s is the serialised form of a query and $\vec{\lambda}_0, \vec{\lambda}_1 \ldots \vec{\lambda}_n$ are vectors of elements whose concatenation equals W_s. For an f-graph Q_f and one of its terms v'_a it applies that:
>
> $$\text{Containment}(W_s, Q_f, v'_a)$$
>
> returns True if and only if the consecutive execution of the function :
>
> $$\text{Containment}(\vec{\lambda}_i, Q_f, v'_i, v'_{\text{next}\,i}, \vec{m}_{\text{path}_i}, \sigma_i)$$
>
> returns $(\text{True}, s'_{i+1}, v'_{\text{next}\,i+1}, \vec{m}_{\text{path}_{i+1}}, \sigma_{i+1})$ for all $0 \leq i \leq n$ such that $v'_0 = v'_a$, $v'_{\text{next}0} = \text{Null}$, $\vec{m}_{\text{path}_0}$ is an empty stack of terms, and σ_0 only maps IRIs and literals to themselves. Each value of v_{i+1}, $v'_{\text{next}\,i+1}$, $\vec{m}_{\text{path}_{i+1}}$, σ_{i+1} is inferred from the i^{th} execution cycle of the function Containment.
>
> ▷ It should be noted that the different calls of the function Containment correspond to its two slightly different forms described in Algorithms 2 and 3 respectively. The previous claim applies because all the information is passed between consecutive executions of the Containment function. A detailed proof of the claim can be based on induction in the size of the $\vec{\lambda}_0, \ldots, \vec{\lambda}_n$ list.

We now proceed to prove the soundness and completeness of the theorem. For the proof we assume that there exists some vertex b_n and the corresponding path leading from the root vertex of the mv-index to b_n is $\alpha, b_1, \ldots, b_n$. We also assume that $L(b_n) = W_s$ which is the serialised form of some query W represented within the mv-index.

Soundness. Suppose that the execution of the function

$$\text{ContQueries}(\mathfrak{M}, \alpha, Q_f, v', \text{Null}, \vec{m}_{\text{path}}, \sigma)$$

returns V_{cont}. If b_n appears in V_{cont}, we have that $L(b_n)$ corresponds to a query inserted into the mv-index because $L_{Query}(b_n) = \text{True}$. We additionally need to show that b_n corresponds to the serialised form of a query that contains Q_f. If b_n appears in V_{cont}, based on line 4 in Algorithm 3, it applies that consecutive executions of the Containment function for the vectors $L(\alpha, b_1), L(b_1, b_2), \ldots, L(b_{n-1}, b_n)$ and the corresponding inputs $v'_0, \ldots v'_n$, $v'_{\text{next}0}, \ldots, v'_{\text{next}\,n}$, $\vec{m}_{\text{path}_0}, \ldots,$ $\vec{m}_{\text{path}_n}$, and $\sigma_0, \ldots, \sigma_n$ will all return True. The later along with Claim ♣ and Proposition 3.3 imply the soundness of the theorem.

Completeness. For the completeness proof let's assume that the query W corresponding to $L(b_n)$ contains Q_f. By Proposition 3.3 and Claim ♣ we have that the consecutive executions of the CONTAINMENT function for the vectors $L(\alpha, b_1)$, $L(b_1, b_2), \ldots, L(b_{n-1}, b_n)$ and the corresponding inputs v'_i, $v'_{\text{next}\,i}$, $\vec{m}_{\text{path}_i}$, and σ_i will all return TRUE. Line 2 in Algorithm 3 ensures that all vertices $\alpha, b_1, \ldots, b_n$ are examined therefore vertex b_n will be examined. Since $L_{Query}(b_n)$ was set to TRUE during the insertion phase of the query, line 8 of the algorithm will return vertex b_n to the corresponding answer set V_{cont}. □

PROOF OF PROPOSITIONS 5.1 & 5.2. We only need to prove Proposition 5.2. Proposition 5.1 directly follows from it. Suppose that $\sigma : W \rightarrow Q$ is a containment mapping from W to Q. We build the mapping σ_w from W to Q_w such that if s is a term in W, s' is a term in Q_w , and $\sigma(s) = s'$ then we set $\sigma_w(s) := [s']$. It should be noted that Q_w differentiates from traditional containment mappings since instead of mapping IRIs and literals to themselves, it maps them to a (possibly nominal) set that contains them.

We now need to show that σ_w is itself a containment mapping. This is an immediate consequence of the fact that, by construction of Q_w, for every triple pattern (s, p, o) appearing in Q, the triple pattern $([s], p, [o])$ appears in Q_w and no more additional triple patterns are added to Q_w. □

Example A.1. Suppose that we have the BGP Q and W

Q : SELECT ? x WHERE (? x, type, Car), (? x, type, Red)

W : SELECT ? x WHERE (? x, type, Vehicle), (? x, type, Red)

and the class inclusion that each car is a vehicle. In order to examine if $Q \sqsubseteq W$ applies, we first extend Q based on the corresponding terminological knowledge. This will add the triple pattern (? x, type, Vehicle) in the extended form of Q, i.e. Q_e. With the additional triple pattern it is obvious that our algorithm will return that there exists a containment mapping from W to Q_e and thus $Q \sqsubseteq W$ also applies.

PROOF OF PROPOSITION 6.1. In the rest of the proof we will denote with $\mathcal{R}$ the RDFS schema under consideration. We will say that a query Q is contained in query W under the RDFS $\mathcal{R}$, i.e. $Q \sqsubseteq_{\mathcal{R}} W$, if the answer set of Q is contained in the answer set of W for every graph G that also satisfies the RDFS $\mathcal{R}$.

If Direction. Suppose that there exists a containment mapping from W to the extended form of a query Q named Q_e, we need to show that $Q \sqsubseteq_{\mathcal{R}} W$ also applies. The existence of a containment mapping $\sigma : W \rightarrow Q_e$ implies that $Q_e \sqsubseteq W$ for every RDF graph G. For an RDF graph G that also satisfies $\mathcal{R}$, a solution to the query Q is a mapping m from the variables of Q to the elements in $IL \cup B$ such that every triple in Q is mapped to a corresponding triple in G. Since G satisfies $\mathcal{R}$, because of the restricted form of the RDFS language it is straightforward to show that m is also a solution for Q_e. Therefore, it applies that $Q \sqsubseteq_{\mathcal{R}} Q_e$. Since $Q \sqsubseteq_{\mathcal{R}} Q_e$ and $Q_e \sqsubseteq W$ apply, based on the transitivity of the $\sqsubseteq_{\mathcal{R}}$ relation, we have that $Q \sqsubseteq_{\mathcal{R}} W$ as we wanted to show.

Only If Direction. Suppose that $Q \sqsubseteq_{\mathcal{R}} W$, we want to show that there exists a containment mapping from Q_e to W. We build a graph G from Q_e as follows: (i) $f : Vars(Q_e) \rightarrow$ IRIs is an bijective function that maps each variable in Q_e to a fresh IRI (i.e. an IRI not already appearing in Q_e); (ii) for each triple pattern in Q_e a corresponding triple is added to G where each variable x has been replaced with $f(x)$. By construction of Q_e and G, it is obvious that the newly created RDF graph G satisfies $\mathcal{R}$. By construction of G, it is also obvious that there exists at least one solution to the query Q when applied to the graph G, the specific solution maps every variable x in Q to $f(x)$ in G, $m(x) := f(x)$. Since $Q \sqsubseteq_{\mathcal{R}} W$ also holds and G satisfies $\mathcal{R}$, we have that there also exists a solution for the query W on the graph G, $m' : W \rightarrow G$. The solution m' allows to infer a containment mapping $\sigma : W \rightarrow Q_e$ by setting for every variable x in W, $\sigma(x) := f^{-1}(m'(x))$ and our proof has finished. □