Efficient Indexing of RDF Queries

Theofilos Mailis
Kapodistrian University of Athens, Greece
tmailis@di.uoa.gr

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

Evgeny Kharlamov
University of Oxford
United Kingdom
evgeny.kharlamov@cs.ox.ac.uk

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

Yannis Kotidis
Athens University of Economics and Business
kotidis@gmail.com

Yannis Ioannidis
Kapodistrian 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 quadratic 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 quadratic 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 arbitrary 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.

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* [13] and *SPARQL* [49] are promising examples of a graph data model and the corresponding query language that have gained a lot of importance. Indeed, *DBpedia*, an RDF version of Wikipedia which serves as the main hub for the Linked Open Data initiative, alone currently 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 [38, 14, 55, 5, 53, 44, 10], 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 [19, 37]. Specifically, any query Q can be rewritten using a view W when a containment relation exists between the query and the view.

The containment problem has been proved to be NP-complete for arbitrary conjunctive queries [15] and unions of conjunctive queries [51] over relational databases. The same results also apply for conjunctive queries on RDF graphs and their SPARQL counterparts [48, 29]. By examining the real-world query workload of DBPedia, we observe that only a small fragment of the queries have all the properties that make query containment so hard to solve. Based on the previous observation, we put f-graph queries, a restricted form of SPARQL queries, under the spotlight. F-graph queries allow to solve the containment checking problem in polynomial time and are dominating real-world query workloads such that of DBPedia. Towards that direction, 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 worst case quadratic 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 represent and evaluate containment for the SPARQL analogues of relational conjunctive queries. 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, a restricted form of SPARQL queries whose structure allows to check for query containment in quadratic time. We initially focus on query containments of the form $Q_f \sqsubseteq W_t$ between an f-graph and a tree query W_t. The algorithm for containment checking operates in quadratic time and is based on a normalised form of tree queries that encodes each tree query starting from its root vertex and blending classes, predicates, and parenthesis symbols to represent its nested subtrees.

- **[Mv-indices]** The structure of tree queries and their corresponding normalised form allow to introduce the materialised-view indices (mv-indices), novel indexing structures for tree queries. 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 tree queries by taking advantage of common patterns that appear in them; (ii) evaluate query containment between an f-graph query Q_f and an arbitrary number of indexed queries within quadratic 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.
- **[Tree & F-Graph Witnesses]** Witnesses further allow to extend mv-indices for RDF conjunctive queries and their SPARQL counterparts.

  Tree witnesses allow to represent an RDF conjunctive query in the right hand side of a query containment as a combination of a tree query and a set of equality constraints on its terms. Triple patterns in witnesses may contain both predicates and inverse predicates (properties). We prove that containment between an f-graph query and an RDF conjunctive query has also a quadratic time complexity in the combined size of the two queries.

  F-graph witnesses allow to represent an 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 also contained in it. The latter allows to use the mv-index structure for arbitrary queries. F-graph witnesses also provide a partial answer to the query containment problem and a NPTime step has to be performed to check if the containment indeed applies.
- **[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) [13]. 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. The query workload consists of $1,536,378$ queries from the 5 different workloads. In our experimental section we evaluate 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 is between 0.0093 msec and 0.041 msec for these workloads.

Paper Structure. In Section 2 we provide some preliminaries on RDF graphs, SPARQL queries, and the query containment problem. We proceed in Section 3 introducing f-graph and tree queries and explaining the importance of the query containment problem between an f-graph and a tree query. In Section 4 we present the normalised form of tree queries and the quadratic algorithm for containment checking between an f-graph query and a tree query. The mv-index structure along with the algorithm for containment checking are presented in Section 5. In Section 6 we introduce tree and f-graph witnesses that allow to represent complicated conjunctive query structures within the mv-index and check for containment. In Section 8 we perform an experimental evaluation of our structures and indexes. Finally, Section 9 presents the current literature on RDF stores, query containment, and view materialisation, while Section 10 summarises the paper and mentions directions for future work.

2. PRELIMINARIES

In this Section we 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 [6].

RDF Graph [50, 46]. Assume the pairwise disjoint infinite sets I, B, and L of IRIs, Blank nodes, and literals. 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 [50]. 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.

Query answering. A solution to a BGP query Q on an RDF graph G is a mapping $m : \text{vars}(Q) \to I \cup L \cup B$ 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.

Query Containment. [15] 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* from a query W to a query Q is a mapping 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. As a convention, we will assume that the function m additionally maps every term $s \in IL$ to itself.

Chandra and Merlin [15] have proved that for boolean queries (queries with no distinguished variables), a containment mapping from W to Q implies that Q is contained in W [15]. The latter theorem implies that for two queries Q and W there is a rewriting of Q using W iff there is containment mapping from W to Q [37]. Thus, finding containment mappings is a fundamental operation used to expedite query processing in view materialisation and query caching techniques.

Example 1. An RDF graph G contains information related to songs and albums represented as triple patterns:

(song1, name, "The Music of the Night")
(song1, fromAlbum, album1)
(album1, name, "The Phantom of the Opera")
(album1, artist, artist3)
(artist3, name, "Andrew Lloyd Webber")
(artist3, type, MusicalArtist)

In the corresponding RDF graph, song1, album1, name, artist3, fromAlbum, artist, type, MusicalArtist are IRIs, while "The Music of the Night", "The Phantom of the Opera", and "Andrew Lloyd Webber" are literals. The RDF triple (artist3, type, MusicalArtist) is a class assertion denoting that artist3 belongs to the class of people that are musical artists.

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:

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

The answer to the query if applied on the latter graph database will be the pair ("The Music of the Night", "The Phantom of the Opera").

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

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

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

3. F-GRAPH & TREE QUERIES

The objective of this paper is the construction of an indexing structure that will allow to: *(i)* efficiently store a query workload $\mathcal{W}$; *(ii)* given a query Q discover every $W \in \mathcal{W}$ for which $Q \sqsubseteq W$ applies.

Motivation. The containment problem between two arbitrary 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 $Q_f \sqsubseteq W_t$

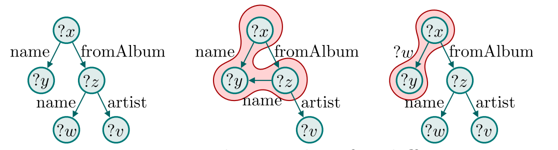


Figure 1: The graphs of 3 different queries

where Q_f belongs to a special class of BGP queries that we name f-graph queries and W_t belongs to the class of BGP queries having a tree structure. 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. What motivates the choice of tree queries in the right-hand side of a query containment is that they can be straightforwardly represented within Radix trees, the building blocks of our indexing structure. Additionally, tree queries can be combined with inverse predicates and equality constraints between terms in order to represent BGP queries.

After solving the containment problem and building the corresponding indexing structures for checking containments between f-graph and tree queries (Sections 4,5), we generalise our methodologies in Section 6. Specifically we introduce witnesses that allow to represent BGP queries within our indexing structure and check for containment.

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.

Tree query. A *tree* query has the form of a rooted tree where all of its edges either point away from the root, i.e. out-tree, or all its edges point towards the root, i.e. in-tree. A tree query with bound predicates is one that has an IRI in the predicate position, i.e. all triple patterns belong to the domain $(I \cup X) \times I \times (IL \cup X)$. For the rest of the paper we focus on out-tree queries with bound predicates. An f-graph query that is also a tree, will be denoted as an *f-tree* query.

Example 2. In Fig. 1 we see 3 different BGPs. From the three BGPs, the first and the third satisfy all the requirements of an f-graph, while the second fails requirement ii in the f-graph definition. Additionally, only the first query is an out-tree query with bound predicates: the second query has a variable node with two incoming edges violating the definition of an out-tree, while the third query has a variable in the predicate position violating the restriction on bound predicates.

4. QUERY CONTAINMENT BETWEEN F-GRAPH & TREE QUERIES

In this section we introduce the algorithm for query containment between an f-graph and a tree query and prove its polynomial complexity. The algorithm for containment is

Algorithm 1 The algorithm for writing a tree to its normal form.

```
1: function NORMALISE(FTreeQuery $W_t$, Term $s$ )
2:     nForm:= $\langle\rangle$
3:     $C_s := \{A | (s, \texttt{type}, A) \in W_t\}$
4:     $\vec{c}_s :=$ ORDER($C_s$)
5:     for i:=1 to $|\vec{c}_s|$ do
6:         nForm.append($\vec{c}_s(i)$)
7:     $E_s := \{(p, o) | (s, p, o) \in W_t\}$
8:     $\vec{e}_s :=$ ORDER($E_s$)
9:     for i:=1 to $|\vec{e}_s|$ do
10:        $(p, o) := \vec{e}_s(i)$
11:        nForm.append(p)
12:        nFormSubTree := NORMALISE($W_t, o$)
13:        if nFormSubTree$\neq \langle\rangle$ then
14:            nForm.append($\boxed{(}$)
15:            nForm.append(nFormSubTree)
16:            nForm.append($\boxed{)}$)
17:    return nForm
```

based on the normalised form of tree queries that we introduced. As we will explain in Section 5 the normalised form of tree queries additionally allows to represent a set of tree queries in a compact form by using Radix trees and use the structure to compute multiple containments.

4.1 Normal Form of Tree Queries

The normalisation rewrites each tree query in a simple form that takes advantage of the ordering between classes and predicates. The normalised form is constituted of a list of elements corresponding to: classes, predicates, and the parenthesis symbols $\boxed{(}$, $\boxed{)}$ as delimiters that are used to denote a subtree structure.

The normalised form of a tree query assumes a total ordering between its classes and predicates, with the symbol $\prec$ denoting the total order relation. The ordering may be based on the lexicographical order of classes and predicates, or some other metric such as the frequency of their appearences within an RDF graph.

Algorithm 1. The normalisation procedure is presented in Algorithm 1. If the NORMALISE(W_t, s) function is applied on the root term s of a tree query W_t, it will return the normal form of the tree. The normalisation is performed in the following steps: *(i)* Initially the normalised form of the query is an empty list of elements (line 2). *(ii)* Then, all the classes that characterise s are collected in a set C_s and ordered in a vector $\vec{c}_s$ from the least to the greatest element based on the total ordering relation $\prec$ (lines 3 and 4). *(iii)* Each class is appended to the normalised form of the tree query respecting the total order (lines 5 to 6). *(iv)* for every triple pattern (s, p, o) with s in the subject position, the pairs (p, o) corresponding to the predicate and the object of the triple are collected in the set E_s and ordered in the set $\vec{e}_s$ (lines 7 and 8). Ordering is performed based on the total ordering of predicates in each pair (p, o). *(v)* The pairs (p, o) are parsed based on the total order of their corresponding predicates (lines 9 to 10). *(vi)* Each predicate is written to the normal form of the query (line 11). *(vii)* If o is the root of a subtree of W_t, we write next to the predicate the normal form of the subtree enclosed in a left and right parenthesis symbol (lines 12 to 16). For the specific subtree we call $\boxed{(}$ its opening and $\boxed{)}$ its closing parenthesis, respectively.

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

```
1: function CONTAINMENT(NormalForm $W_{\text{norm}}$, FGraph $Q_f$, Term $s'$)
2:     Stack $\vec{m}_{\text{path}} := \langle\rangle$
3:     for $i := 1$ to $|Q_{\text{norm}}|$ do
4:         if $Q_{\text{norm}}(i)$ is a Class A then
5:             if $(s', \texttt{type}, A) \notin Q_f$ then
6:                 return FALSE
7:         else if $Q_{\text{norm}}(i)$ is a Predicate $p$ then
8:             if $(s', p, o') \in Q_f$ then
9:                 Term $s'_{\text{next}} := o'$
10:            else
11:                return FALSE
12:        else if $Q_{\text{norm}}(i) = \boxed{(}$ then
13:            $\vec{m}_{\text{path}}$.push($s'$)
14:            $s' := s'_{\text{next}}$
15:        else if $Q_{\text{norm}}(i) = \boxed{)}$ then
16:            $s' := \vec{m}_{\text{path}}$.pull()
17:    return TRUE
```

Example 3. When applying the NORMALISE($Q, ?sng$) function to the tree query Q presented in Example 1, with $?sng$ being the root variable of the query, the output will be the following list of elements:

$$\boxed{fromAlbum}\ \boxed{(}\ \boxed{artist}\ \boxed{(}\ \boxed{MusicalArtist}\ \boxed{)}\ \boxed{name}\ \boxed{)}\ \boxed{name}$$

In this example, we assume that the $\prec$ relation between classes (or predicates) follows their lexicographical order. Reading the normal form of the query from left to right, we ask for an entity (a song in our case) that belongs to an album. Next, the opening parenthesis indicates that information about the album is asked. Specifically we ask for an album that has an artist and because of the new nested subtree, we know that this artist is a musical artist. Before closing the subtree about the album, the name of the album is also asked. When the subtree concerning the album is closed, we ask for the name of the song.

We can see that the normalised form of the query holds all the initial information except from the corresponding terms. Thus, the normalisation process reflects the tree structure of the query but ignores IRIs and literals that may appear as subjects or objects within a triple pattern. In order to keep the corresponding information that a term corresponds to a specific IRI or literal, we introduce nominal classes that contain only the specific IRI or literal. A nominal class [20] is a singleton set that contains the element under consideration. In our implementation of the normal form we use separators to distinguish between nominal classes, classes, and predicates.

4.2 Containment Checking

Algorithm 2 provides the function CONTAINMENT that is used for checking containment between an f-graph query Q_f and the normal form of a tree query W_t named W_{norm}. It takes as input: *(i)* the normalised form of the tree query W_{norm}; *(ii)* an f-graph query Q_f; *(iii)* a term s' in Q_f. If CONTAINMENT(W_{norm}, Q_f, s') returns TRUE, then there exists a containment mapping from W_t to Q_f such that the root term of W_t is mapped to the term s'.

While examining if there exists a containment mapping from the query W_t to the f-graph query Q_f, each opening parenthesis in the normalised form of W_t indicates that we currently focus on finding containment mappings for one of W_t's subtrees. Each subtree has a corresponding root term that, though does not explicitly appear in the normalised form of W_t, exists and is unique. Additionally, there exists a specific path of edges leading from the root-term of the tree query W_t to the root-term of the examined subtree. Suppose that $\vec{p}_{\text{ath}}$ is the corresponding implicit vector of terms in W_t that leads to the root of the subtree that is currently being examined. $\vec{m}_{\text{path}}$ denotes the corresponding vector of terms 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. The $\vec{m}_{\text{path}}$ is implemented as a stack that allows to focus attention to different parts within the f-graph by pushing and pulling terms of Q_f in it. Pushing is performed when examining a nested subtree after an opening parenthesis, while pulling is performed when a nested subtree has just been examined and its closing parenthesis has appeared.

Algorithm 2. The algorithm checks if there exists a containment mapping for all triples in the tree query W_t (lines 3 to 16) with each triple in W_t being represented in W_{norm} by a corresponding class or predicate. Checking is performed in the following steps: *(i)* The algorithm will fail whenever there exists a class A characterising the root variable of W_t, but no corresponding triple $(s', \texttt{type}, A)$ exists about s' in Q_f (line 6). *(ii)* The same happens when the root term of W_t is connected to another term via a predicate p, but there exists no corresponding triple (s', p, o') about the term s' (line 11). *(iii)* If, on the other hand, a triple (s', p, o') appears in the body of Q_f, then the variable s'_{next} takes the value of the triple's object o' (line 9). *(iv)* When an opening parenthesis is met a subtree mapping has to be examined (lines 12 to 14). The subtree W'_{norm} appears between an opening and a closing parenthesis. *(v)* The algorithm now has to check if there exists a containment mapping between the root of W'_{norm} and the object of the last encountered triple pattern that has been stored in s'_{next}. The first step is to push the term that is being currently examined into the $\vec{m}_{\text{path}}$ stack (line 13). *(vi)* The next step is to tell the algorithm that we are currently interested in finding a mapping between W'_{norm} and Q_f that maps the root variable of W'_{norm} to s'_{next}. Thus s' takes the variable of s'_{next} (line 14) and our algorithm checks containment for W'_{norm}. *(vii)* Pushing the term s' into $\vec{m}_{\text{path}}$ (line 13), allows to continue examining s' when a containment mapping for the subtree of W'_{norm} is found (line 16). Then s' takes its previous value from $\vec{m}_{\text{path}}$. *(viii)* Finally, when all the elements of W_{norm} have been examined and matched, the algorithm returns True (line 17). If a nominal appears in line 4 in Algorithm 2, it is satisfied when the IRI or literal within the nominal, is equal to the term s' of the f-graph query that is currently under examination.

Proposition 1. *For (i) a tree query W_t with bounded predicates, (ii) its normalised form W_{norm}, (iii) W_t's root variable s', (iv) an* f-graph *query Q_f (v) and a term s' in Q_f: there exists a containment mapping m from the variables of Q_f to the variables of Q_f for which $m(s) = s'$ applies iff it is returned* True *by the function:*

$$\textsc{Containment}(W_{\text{norm}}, Q_f, s')$$

Execution Time. To study the execution time of Algorithm 2, we observe that every execution cycle of the algorithm examines a different element of W_{norm}. Steps 4 and 8 of Algorithm 2 examine if a class or a predicate appears in the related triple patterns. By definition of an f-graph query, when examining a certain term s' within the f-graph, there exists at most one such triple pattern about s'. If we assume that this step is performed in constant time via building a hash map based on a perfect hashing function [24, 21], the algorithm terminates in linear time w.r.t. the size of the tree query $\mathcal{O}(|W_t|)$. To check for containment $Q_f \sqsubseteq W_t$ between an f-graph query Q_f and a tree query W_t we need to apply the $\textsc{Containment}(Q_f, W_{\text{norm}}, s)$ function for every term s' appearing in Q_f. Thus the time for checking containment is quadratic in the size of the two queries $\mathcal{O}(|W_t| \cdot |Q_f|)$.

5. 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 tree queries $\mathcal{W}$, it would be non optimal 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 [42]. Mv-indices take advantage of the ordering between classes and predicates that was introduced in Section 4.1 in order to efficiently find the queries in $\mathcal{W}$ that contain Q_f.

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 (classes, predicates, and parenthesis symbols) and each vertex to the normalised form of an f-tree 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 is rather an intermediate query that was created during the insertion process.

The intuition for this form of representation is that queries are represented by their normalised form in the mv-index structure either as intermediate or leaf vertices using the labelling function L. For a vertex α in the mv-index structure, $L(\alpha)$ is its corresponding query in normalised form. The normalised 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.

Example 4. Fig. 2 represents an mv-index. For space purposes we use initial letters to depict classes and predicates. 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 α corresponds to an empty query and is the root of the mv-index. Vertex β corresponds to the normalised query $\boxed{\text{artist}}\ \boxed{(}\ \boxed{\text{Composer}}\ \boxed{\text{MusicalArtist}}\ \boxed{)}$. The labelling of vertex β is the same as the labelling of the edge (α, β). Vertex ζ corresponds to the query $\boxed{\text{fromAlbum}}\ \boxed{(}\ \boxed{\text{artist}}\ \boxed{(}\ \boxed{\text{MusicalArtist}}\ \boxed{)}\ \boxed{\text{name}}\ \boxed{)}$.

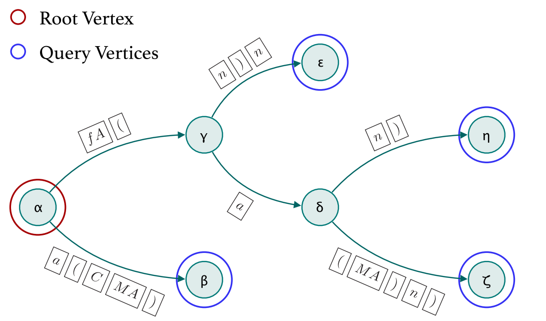


Figure 2: A simple mv-index. For space efficiency, we employ the shortcuts fA, n, a, MA, C to represent the predicates fromAlbum, name, artist and the classes MusicalArtist, Composer, respectively.

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

```
1: function ContainedQueries(MVIndex $\mathfrak{M}$, Vertex $\alpha$, FGraphQuery $Q_f$, Term $s'$, Term $s'_{\text{next}}$, Stack $\vec{m}_{\text{path}}$)
2:     for all $\beta$ s.t. $(\alpha, \beta) \in E$ do
3:         $\vec{\lambda} := L(\alpha, \beta)$
4:         $\vec{m}'_{\text{path}} = \text{CopyOf}(\vec{m}_{\text{path}})$
5:         $(\text{toContinue}, s', s'_{\text{next}}, \vec{m}'_{\text{path}}) := \text{Containment}(Q_f, \vec{\lambda}, s', s'_{\text{next}}, \vec{m}'_{\text{path}})$
6:         if toContinue = True then
7:             $V_{\text{cont}} := V_{\text{cont}} \cup \text{ContainedQueries}(\mathfrak{M}, \beta, Q_f, s', s'_{\text{next}}, \vec{m}'_{\text{path}})$
8:             if $L_{Query}(\beta) = \text{True}$ then
9:                 $V_{\text{cont}} := V_{\text{cont}} \cup L(\beta)$
10:    return $V_{\text{cont}}$
```

```
1: function Containment(NormalForm $W_{\text{norm}}$, FGraph $Q_f$, Term $s'$, Term $s'_{\text{next}}$, Stack $\vec{m}_{\text{path}}$)
2:     ~~Stack $\vec{m}_{\text{path}}$ := empty stack~~
......
6:                 return (False,Null,Null,Null)
......
11:                return (False,Null,Null,Null)
......
17:    return (True, $s'$, $s'_{\text{next}}$,$\vec{m}_{\text{path}}$)
```

The labelling of vertex ζ is the concatenation of the labelings $L(\alpha, \gamma)$, $L(\gamma, \delta)$, $L(\delta, \zeta)$. Vertex η corresponds to the query $\boxed{\text{fromAlbum}}\ \boxed{(}\ \boxed{\text{artist}}\ \boxed{\text{name}}\ \boxed{)}$. The labelling of vertex η is the concatenation of the labelings $L(\alpha, \gamma)$, $L(\gamma, \delta)$, $L(\delta, \eta)$. Vertex ε corresponds to the query $\boxed{\text{fromAlbum}}\ \boxed{(}\ \boxed{\text{name}}\ \boxed{)}\ \boxed{\text{name}}$. The labelling of vertex ε is the concatenation of the labelings $L(\alpha, \gamma)$, $L(\gamma, \varepsilon)$.

During the insertion phase, mv-indices are treated as regular Radix trees that instead of strings or numbers are used to represent tree queries in their normalised form. Therefore, instead of characters within a string, or digits within a number, mv-indices use classes, predicates and separators such as parenthesis symbols in order to represent normalised-tree queries. More information on how insertion works in Radix trees can be found in the literature [42].

5.1 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 element ordering within the normalised form of a tree query (Section 4.1) that is also reflected within the mv-index structure.

In Algorithm 3, the ContainedQueries 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 s' that appears in Q_f; *(v)* a term s'_{next} that appears in Q_f; *(vi)* and the stack $\vec{m}_{\text{path}}$. The ContainedQueries function is used to acquire the set of vertices $V_{\text{cont}} \subseteq V$ such that: for every vertex $\alpha \in V_{\text{cont}}$ with $L(a)$ representing the normal form of a tree query W_t, it applies that $Q_f \sqsubseteq W_t$.

Each path from the root ρ to a vertex γ of the mv-index such that $L_Q(\gamma) = \text{True}$ corresponds to the normalised form of some f-tree query W_t, with $L(\gamma)$ being the corresponding normalised form. When examining an mv-index path, on the transition from one vertex to another, the initial normal 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 quadruple of values. Along with the True value, it will return the terms s' and s'_{next} along with the $\vec{m}'_{\text{path}}$ stack (line 17). 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 ContainedQueries procedure. For each vertex α of the mv-index, its corresponding normalised 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 also examined for a containment mapping (lines 2 to 9). *(ii)* Initially, the vector $\vec{\lambda}$ takes the labelling of the corresponding edge (line 3). *(iii)* Subsequently it is examined if the corresponding edge violates the containment mapping or not (line 5). *(iv)* If there is a violation the Containment function will return (False, 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 6). *(v)* If on the other hand a containment mapping exists, the Containment function will return a quadruple 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 7). *(vi)* Additionally, if the vertex β corresponds to a query inserted into the mv-index (line 8), the corresponding vertex will be added to the vertices whose query contains Q_f (line 9). The algorithm terminates when there are no more vertices to be examined.

Theorem 1. For the MVIndex $\mathfrak{M}$, its root vertex α, an f-graph query Q_f, a term s' appearing in the triple patterns of Q_f, and an initially empty stack $\vec{m}_{\text{path}}$, the execution of

$$\text{ContainedQueries}(\mathfrak{M}, \alpha, Q_f, s', \text{Null}, \vec{m}_{\text{path}})$$

will return in V_{cont} every vertex α that appears in the mv-index $\mathfrak{M}$ for which a containment mapping $m : L(\beta) \rightarrow Q_f$

exists such that $m(s) = s'$ also applies, with s being the (implicit) root term of the $L(\beta)$ query.

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

Implementation. In our actual implementation, in order to avoid making unnecessary comparisons, we build a hash map from each mv-index vertex to its corresponding edges. The hash map allows to acquire specific edges of the mv-index that are meaningful for the part of the f-graph query Q_f that is being examined. For example if we have just examined a triple pattern (s', p, o') in Q_f, because of the ordering of elements within the normalised form of the queries inserted in the mv-index, we only need to examine the mv-index for triple patterns (s', p', o'') in Q_f such that $p \preceq p'$. Thus, we take advantage of the element ordering within mv-indices.

Execution Time. To analyse the cost of the algorithm execution, we need to analyse its worst case execution. The worst case appears when all the queries in the mv-index can be mapped to the f-graph query Q_f. In each execution of the ContainedQueries we need to compare the f-graph Q_f against all the elements in $\mathfrak{M}$. The specific step will take $\mathcal{O}(|\mathfrak{M}|)$ time to be executed where $|\mathfrak{M}|$ is the size of the mv-index measured as the number of classes, properties, and separators that appear in it. 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 $\mathcal{O}(|\mathfrak{M}| \cdot |Q_f|)$.

6. WITNESSING BGP QUERIES

Sections 4 and 5 focus on solving the problem of query containment between an f-graph and a tree query, i.e. $Q_f \sqsubseteq W_t$, with W_t either being a single tree 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 f-graph and tree restrictions do not apply. For each extension, we explain how the problem is solved for the simple case of query containment between two queries and then comment on how the mv-index structure needs to be extended in order to accommodate the aforementioned changes. To perform these tasks we introduce *tree* and *f-graph witnesses* that are used as substitutes for more complicated query structures. Witnesses permit us to use mv-indices with some minor modifications.

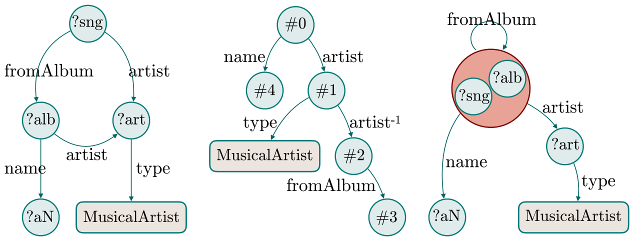
6.1 Tree Witnesses for BGP Queries

A *tree witness* of a BGP query, is a tree query that is used instead of the BGP in the right hand side of a containment-checking operation. The intuition for tree witnessing a BGP query is that: the direction of a triple pattern can be inversed with the adoption of inverse predicates; cycles within the BGP can be represented as equality constraints between its terms.

Algorithm 4. The methodology for acquiring the tree witness of an arbitrary BGP is presented in Algorithm 4. It takes as input a query W and an arbitrary term s in W and returns its corresponding witness W_w along with a set of equality constraints between its terms. *(i)* Initially, each class assertion about s in W will be added to the body of its tree witness W_w (line 4). *(ii)* For all triple patterns (s, p, o) we work as follows (line 5): *(iii)* If the object variable o has not been examined (line 6), we add the triple pattern to W_w (line 7). Additionally, we extract the tree witness of the subgraph starting from o and add all of its triples to W_w (line 8). *(iv)* In the case that the object variable o has already been examined (line 17), then we cannot add the triple patter (s, p, o) since W_w will lose its tree form. Instead, we choose a new variable name z that will be representing o in the triple (lines 10, 11) and the information that z and o initially corresponded to the same variable will be stored in the set of equality constraints $\mathfrak{R}$ (line 12). *(v)* Triple patterns about the term s of the form (o, p, s) are processed in a similar way (lines 13-20). The main difference is that, in order to maintain the directed tree property, the triple pattern (o, p, s) will be represented as (s, p^{-1}, o), with p^{-1} being the inverse predicate of p (lines 15, 19).

Algorithm 4 The algorithm for finding tree witnesses BGP queries.

```
 1: function Witness(BGP $W$, Term $s$, TreePattern $W_w$, Set $\mathfrak{R}$)
 2:     ExaminedTerms := ExaminedTerms $\cup \{s\}$
 3:     for all $(s, type, A) \in W$ do
 4:         $W_w := W_w \cup \{(s, type, A)\}$
 5:     for all $(s, p, o) \in W$ do
 6:         if $y \notin$ ExaminedTerms then
 7:             $W_w := W_w \cup \{(s, p, o)\}$
 8:             $W_w := W_w \cup$ Witness$(W, y, W_w, \mathfrak{R})$
 9:         else
10:             $z :=$ nextTermName( )
11:             $W_w := W_w \cup \{(s, p, z)\}$
12:             $\mathfrak{R} := \mathfrak{R} \cup \{z = o\}$
13:     for all $(o, p, s) \in W$ do
14:         if $y \notin$ ExaminedTerms then
15:             $W_w := W_w \cup \{(s, p^{-1}, o)\}$
16:             $W_w := W_w \cup$ Witness$(W, o, W_w, \mathfrak{R})$
17:         else
18:             $z :=$ nextTermName( )
19:             $W_w := W_w \cup \{(s, p^{-1}, z)\}$
20:             $\mathfrak{R} := \mathfrak{R} \cup \{z = o\}$
21:     return $W$
```



(a) Initial query (b) Tree witness (c) F-graph witness

Figure 3: A BGP query and its corresponding witnesses

Example 5. Fig.s 3a, 3b display a BGP query Q and its corresponding tree-witnesses W_w. Specifically, the tree witness $W_w :=$ Witness$(Q, ?alb, \langle\rangle, \emptyset)$ In Fig. 3b we have replaced the initial variables with the corresponding positions that they appear in the normal form of the query which is:

$$\boxed{artist}\ \boxed{(}\ \boxed{MusicalArtist}\ \boxed{artist^{-1}}\ \boxed{(}\ \boxed{fromAlbum}\ \boxed{)}\ \boxed{)}\ \boxed{name}$$

In order for the witness to accurately represent the initial graph structure, it should also satisfy the corresponding set of constraints $\mathfrak{R} = \{\#0 = \#3\}$.

The normalised form of a witness is similar to that of a tree with the exception that inverse predicates may appear in it. When calling $\textsc{Witness}(Q, s, \langle\rangle, \emptyset)$ for a BGP query W, the input term s will correspond to the root of the constructed tree-witness W_w. Since each variable in W can be chosen as the pseudo-root of W_w, there are more than one witnesses that can be constructed for a BGP query W. Proposition 2 implies that for containment checking we can choose an arbitrary witness as a representative of a BGP query.

For a containment mapping m from a tree query W_t to a query Q, a set of equality constraints $\mathfrak{R}$ is satisfied by m when for each pair of terms z, w appearing in W_t such that $(z = w) \in \mathfrak{R}$ it applies that $m(z) = m(w)$.

Proposition 2. *For a query W, its corresponding tree witness W_w, and a BGP query Q: there exists a containment mapping from W_w to Q that also satisfies the equality constraints $\mathfrak{R}$ iff there exists a containment mapping from W to Q.*

Containment Checking. Since the main difference between trees and tree witnesses is the existence of inverse predicates and equality constraints, we want Algorithm 2 to be extended to handle such predicates. In Algorithm 5, the extension of the Containment function in Algorithm 2 is presented in order to handle inverse predicates. The existing code will be positioned between lines 11 and 12 in Algorithm 2. It can be proved, similar to Proposition 5, that the algorithm is sound and complete with the presence of tree witnesses.

To encode equality constraints for the normalised form of tree queries, we replace variable names in $\mathfrak{R}$ with their corresponding positions in the normalised structure. The position #0 is registered for the root of the tree, while the position $\#i$ is the position of the $(i + 1)$th variable that is examined during the normalisation process in Section 4.1. Since the equality constraints between variables can be checked within linear time, our algorithm is executed in PTime for containment checking between an f-graph Q_f and a BGP query Q.

Containment Checking for mv-indices. The mv-index structure also needs to accommodate some changes in order to handle tree witnesses. First of all we need to extend the Containment function in Algorithm 3 the same way we did for the Containment function in Algorithm 2. Additionally, equality constraints between terms need to be encoded within the mv-index structure.

Since each tree query is represented as a vertex in the mv-index structure, we additionally need to encode the equality constraints related to a BGP and its corresponding tree witness. Because the same tree may witness more than one BGP queries, the same vertex in the mv-index may be used to encode more than one BGP queries. We use an additional labelling L_{eq} from each vertex to multiple constraint sets, each constraint set representing a different BGP query. Since the labelling function L_{eq} is used to encode multiple sets of equality constraints, in our implementation we have used an additional Radix tree to check for equality constraint satisfaction.

Algorithm 5 Extending the Containment function in Algorithm 2 to handle tree witnesses.

```
...
12:        else if $Q_{\text{norm}}(i)$ is an inverse Predicate then
13:            if $(o, p, s) \in Q$ then
14:                Term $nextTerm := y$
15:            else
16:                return False
...
```

With the new form of mv-indices, we need to extend the algorithm for containment to work in two steps. In the first step the ContainedQueries finds the vertices that match the tree part of the witness. In the next step, the ContainedQueries needs to check for equality constraint satisfaction. Because equality restrictions are checked in linear time, the execution time of the algorithm is the same as in Section 4.2.

Witnessing BGPs with Unbounded Predicates. The current mv-index implementation supports tree-witness for BGP queries with bounded predicates, i.e. all the triples within the BGP have an IRI in the predicate position. In the case that we want to extend our algorithm to allow the appearance of variables in the predicate position, we work as follows: *(i)* we store within the mv-index structure the part of the BGP query for which the bounded predicate restriction applies; *(ii)* we keep a pointer to the remaining part of the query for which the bounded predicate restriction does not apply; *(iii)* when checking for containment, our algorithm first finds the containment mapping for the indexed part of the query that contains only bounded predicates; *(iv)* an additional check has to be performed for the part of the initial BGP that has the unbounded predicates.

6.2 F-Graph Witnesses for BGP Queries

We now 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 idea, similar to the one presented in Section 6.1, is that each BGP query can be represented in the form of an f-graph.

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. It can be proved that finding all equivalence classes can be performed in linear 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. In a similar way a class assertion $(s, \texttt{type}, A)$ in Q will be replaced with $([s], \texttt{type}, A)$ in its corresponding witness Q_w. By construction there is a unique witness for each query Q.

Proposition 3. *For a BGP query Q, its corresponding* f-graph *witness Q_w and a BGP query W, it applies that:*

$$Q \sqsubseteq W \Rightarrow Q_w \sqsubseteq W.$$

Proposition 4. *For a BGP query Q, its corresponding* f-graph *witnesses Q_w and a BGP query W, for each containment mapping $m : W \rightarrow Q$ for which $m(s) = s'$ applies, there exists a containment mapping $m_w : W \rightarrow Q_w$ such that $m_w(s) = [s']$ applies.*

Containment Checking. What Proposition 3 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 polynomial time as presented in Section 4.2, while checking $Q \sqsubseteq W$ is in the worst case a NP-complete problem. Therefore we pay a polynomial time budget to solve specific instances of a NP-complete problem.

What Proposition 4 says is that every containment mapping $m : Q \rightarrow W$ that needs NPTime to be computed can be inferred from a containment mapping $m_w : Q_w \rightarrow W$ that is computed in PTime. It should be noted that each m_w may result in more than one containment mappings m. Suppose that $\mathcal{D}_{m_w}$ is the domain of the mapping m_w, a nondeterministic algorithm for finding all containment mapping m from m_w can be defined as follows. For each variable $x \in \mathcal{D}_{m_w}$, $m_w(x)$ is an equivalence class of variables, IRIs, and literals. An oracle chooses some arbitrary $s' \in m_w(x)$ and defines $m(x) := s'$. Each such mapping m has additionally to be checked in polynomial time if it actually is a containment mapping. It should be noted, that the aforementioned procedure can be adjusted for tree queries in order to be performed in PTime.

With $|\cdot|$ denoting the elements within an equivalence class, we define the *degree of non determinism* of a containment mapping from a query W to an f-graph witness Q_w as follows: $\prod_{x \in \mathcal{D}_{m_w}} |m_w(x)|$. The degree of non-determinism is equal to the number of containment mappings that can result from m_w. In a similar way, we may define the non-determinism 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 non-determinism degree that is equal to 1.

Example 6. Fig. 3c displays the f-graph witness corresponding to the query in Fig. 3a. The corresponding f-graph witness has a non-determinism 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 tree witness W_w in Fig. 3b, Algorithm 5 will create in polynomial time the containment mapping m_w: $m_w(\#0) = \{?alb, ?sng\}$, $m_w(\#1) = \{?art\}$, $m_w(\#2) = \{?alb, ?sng\}$, $m_w(\#3) = \{?alb, ?sng\}$, $m_w(\#4) = \{?aN\}$. In order to acquire from m_w the actual containment mapping $m : W_w \rightarrow Q$ we need to clarify the non-deterministic parts of the mapping m_w (m is the same for the other parts), i.e. the parts for positions $\#0, \#2, \#3$. For the mapping of the variable in position $\#0$ there are two alternatives based on m_w, either $m(\#0) = ?alb$ or $m(\#0) = ?sng$. From the two mappings, only the first one satisfies the triple $(\#0, \texttt{name}, \#4)$ appearing in W_w. In a similar way we find that $m(\#2) = ?sng$ and $m(\#3) = ?alb$. Finally we need to check if the constraints in the normal form of W_w are satisfied which is the case since $m(\#0) = m(\#3) = ?alb$.

Table 1: Semantic relationships expressible in an RDFS

Semantic relationship	Datalog notation
Class inclusion	$(x, \texttt{type}, A) \rightarrow (x, \texttt{type}, B)$
predicate inclusion	$(x, p_1, y) \rightarrow (x, p_2, y)$
Domain Restriction	$(x, p, y) \rightarrow (x, \texttt{type}, A)$
Range Restriction	$(x, p, y) \rightarrow (y, \texttt{type}, A)$

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 in NP-complete time if $Q \sqsubseteq W$ actually applies.

7. 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) [13]. The problem of query containment becomes more complicated with the presence of class inclusions, poroperty inclusions, domain and range restrictions. In Table 1 we see in Datalog notation the different forms of semantic relationships that are expressible in RDFS, with x, y being variable names, A, B class names, and p, p_1, p_2 predicate names.

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.

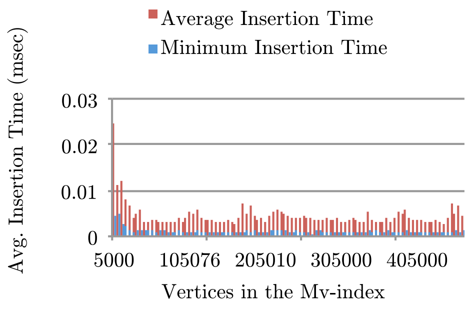
Proposition 5. *The query containment $Q \sqsubseteq W$ applies w.r.t. to an RDF schema, iff there exists a containment mapping from W to the extended form of the query Q.*

Example 7. Suppose that we have the BGP Q and W

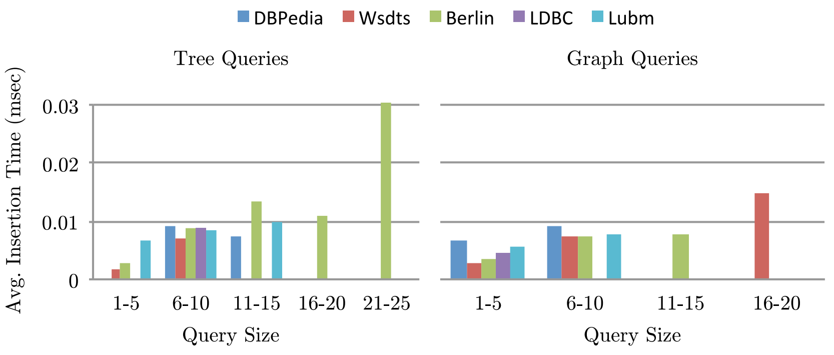
$$Q : \texttt{SELECT } ?x \texttt{ WHERE } (?x, \texttt{type}, \texttt{Car}), (?x, \texttt{type}, \texttt{Red})$$
$$W : \texttt{SELECT } ?x \texttt{ WHERE } (?x, \texttt{type}, \texttt{Vehicle}), (?x, \texttt{type}, \texttt{Red})$$

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 the triple pattern $(?x, \texttt{type}, \texttt{Vehicle})$ 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.

Based on the previous proposition, for a query Q and an RDFS, in order to find every query W within the mv-index such that $Q \sqsubseteq W$, we first need to rewrite Q to its extended form according to the RDFS. Then our algorithm is performed as before.



(a) Insertion time w.r.t. number of vertices



(b) Insertion time w.r.t. query size

Figure 4: Query Insertion Evaluation

8. EXPERIMENTAL EVALUATION

The aim of our evaluation section is to examine the performance of mv-indices during the insertion and query containment phases (Section 5). 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 $Q \sqsubseteq W$.

Hardware and memory. We deployed our implementation on a 2010 MacBook laptop having a single 2.66 GHz Intel Core i7 processor with 2 cores, and 8 GB of main memory. It should be noted that a single core was used during the experimental evaluation. The laptop was running macOS High Sierra.

Implementation Setup. We have implemented our algorithm in Java 8 using the Apache Jena 3.6.0 open source Semantic Web framework [33] to parse Sparql query workloads.

Benchmarks. We used 5 different benchmarks for the evaluation of our implementation: (i) a real-world query workload[1] originating from the *DBpedia semantic knowledge graph* [11] containing 1,287,711 BGP queries; (ii) a synthetic query workload[2] originating from the *WatDiv Sparql diversity test suite* [8] containing 148,800 generated BGP queries; (iii) a synthetic query workload[3] originating from the *Berlin SPARQL Benchmark (BSBM)* [12] containing 99,800 generated BGP queries; (iv) a synthetic query workload originating from the *Lehigh University Benchmark (LUBM)* [26] containing 14 BGP queries; (v) a query workload[4] originating from the *LDBC social network benchmark* [23] 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-tree queries, 378,884 are tree queries (but not f-graph queries), 67,340 are f-graph queries (but not tree queries), and 18,658 are BGP queries that are neither trees nor f-graphs.

8.1 Insertion Cost

Initially we examine how inserting queries into the mv-index structure is affected by: (i) the size of the mv-index structure, (ii) the size and the characteristics of the query. The mv-index is inserted with all the queries from the 5 query workloads, 1,536,378 queries in total. The insertion resulted in an mv-index with a total of 466,576 intermediate vertices, while the mv-index represents 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, 0.0072 msec for the DBPedia, LDBC, WatDiv, BSBM, and LUBM query workloads.

Mv-index Size. In Fig. 4a 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. 4a we observe that there is not a 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 phase. 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 of our implementation.

Query Size. In our second experiment we observe that there exists a more explicit relation between the query size and its insertion time into the mv-index structure. Fig. 4b displays the average insertion time (y-axis) for different query sizes (x-axis), for the 5 different query workloads, and for tree-structured or graph-structured queries, i.e. BGP queries that are not trees. We observe that the insertion cost augments almost linearly w.r.t. the query size. Additionally, there is a slight increase in the insertion time of a non-tree query that is attributed to the more complicated representation of a graph structure that involves introduction of inverse predicates and equalities between query terms. To conclude, we observe that query insertions are really fast, while insertion time scales almost linearly w.r.t. query size, as predicted from our analysis.

8.2 Containment Cost

We now examine the query containment time for different query parameters: (i) the size of the query; (ii) the height of the query; (iii) the non-determinism 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 graph predicates: f-tree queries are f-graph queries that also have a tree structure; non-f tree queries have a tree structure but are not f-graphs; f-graph queries are f-graph queries without a tree structure; non-f graph queries are queries that are not f-graphs and don't have a tree structure. We additionally compute the average time for containment w.r.t. the five different query workloads. The average time for query containment is 0.009288 msec for queries in the DBPedia workload, 0.012746 msec for queries in the WatDiv workload, 0.016620 msec for queries in the BSBM workload, 0.040968 msec for queries in the LDBC workload, and 0.010368 msec for queries in the LUBM workload.

Query Size. Fig. 5 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. In the average case, we observe that the query containment time increases along the size of the query. We also observe that the average containment time for queries of similar sizes tends to increase for non-f tree queries, non-f graph queries. This is attributed to the fact that containment checking is NP-complete for non-f graph queries. Additionally, tree queries need less time to be processed compared to non-tree queries with similar characteristics.

Query Height. Analogous behaviour is observed in Fig. 6 that examines the average containment time for queries of different heights. For queries that are not tree queries, we measure the height of their corresponding tree witness (since their height cannot be defined). Once again we observe that f-graph queries are executed in less time compared to non f-graph queries, while tree queries are usually answered faster than non-tree ones.

Non-Determinism Degree. Finally, Figure 7 displays how the query containment operation is affected by the non-determinism degree of a query. We have a figure for tree queries and one for graph queries without the tree property. By definition of the non-determinism degree, we have that queries with a non-determinism degree of 1 are also f-graphs. It is evident from Figure 7 that the complexity of answering a graph query increases along with its non-determinism 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 7.

Fig. 8 presents how the performance of the containment algorithm is affected by the existence of an RDF schema. The x-axis displays the query size, while the y-axis displays the average time needed to find all containments for the case that the query under examination remains unchanged, or is extended according to the RDFS. It should be noted, that in the case that Q remains unchanged, the containment algorithm will result to an incomplete solution, i.e. we may miss some implicit containments. We observe that the containment time is increasing almost exponentially w.r.t the query size. This is attributed to the fact that, for complicated ontologies, on the one hand the size of the original query increases, on the other hand the number of answers to the containment problem is increased as well. Finally, the inference process, may cause to some of the queries to lose their f-graph properties thus making them harder to process.

9. RELATED WORK

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

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 [38], Sesame [14], HexaStore [55], SW-Store [5], MonetDB-RDF [53], RDF-3X [44], and BitMat [10] have been carefully designed to keep up pace with the growing scale of RDF collections. Systems like TriAD [28], RDFox [43], H-RDF-3X [31], EAGRE [57] 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 [54, 39] 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 [15, 51, 34, 18] and bag semantics [16, 6, 32] 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 [18, 6]. The results of the query containment problem can be transferred to SPARQL by proving reducibility between SPARQL queries and relational algebra expressions [9, 48]. The problem of query containment has also been studied for RDF databases and SPARQL queries [47, 29], as well as SPARQL extensions with property paths [52], RDFS entailment [17], and $\mathcal{SHI}$ terminological knowledge [56] and `OPTIONAL` queries [36]. Our work complements past work on 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.

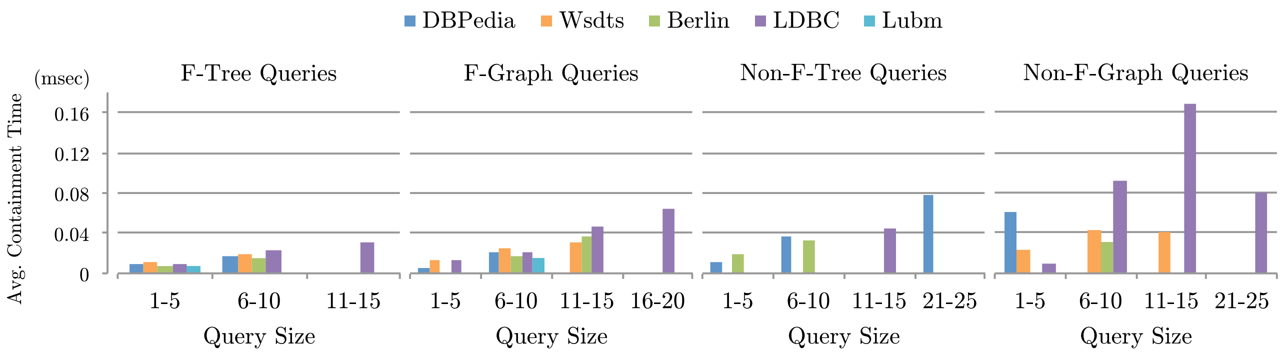


Figure 5: Containment Cost w.r.t. Query Size (number of triple patterns)

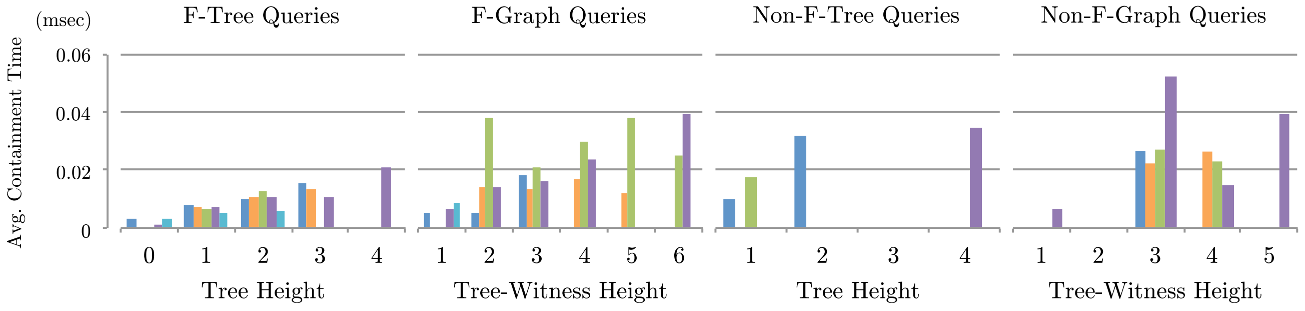


Figure 6: Containment Cost w.r.t. Query Height

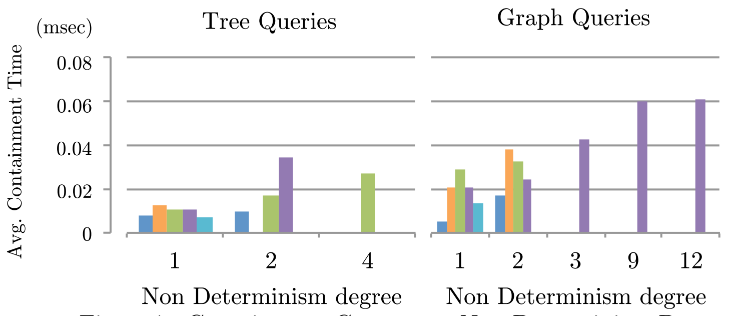


Figure 7: Containment Cost w.r.t. Non Determinism Degree

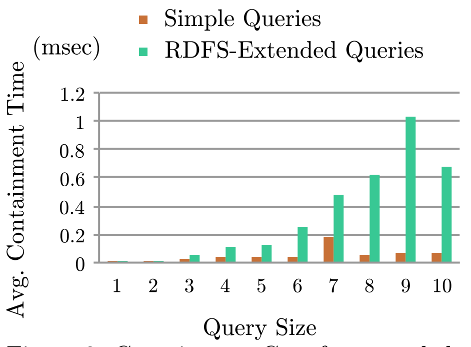


Figure 8: Containment Cost for extended queries

View Materialisation. Materialised view selection has been extensively studied by relational databases [40, 37, 27, 37, 7] and data warehouses [30, 35, 41]. The problem of view materialisation for SPARQL queries has recently gained attention by the Semantic Web community. Dritsou et al. [22] suggest the materialisation of shortcuts that reduce the execution cost of path queries. Goasdoue et al. [25] propose a view materialisation approach where an initial query workload $\mathcal{W}$ is transformed to a set of simpler views $\mathcal{V}$ along with a set of rewritings. Papailiou et al. [45] suggest a caching strategy that caches existing structures and rewrites queries accordingly if they match the specific cache. The mv-index structures we propose are complementary to the existing systems and techniques and can be used to boost their performance.

10. CONCLUSIONS AND FUTURE WORK

Our study introduces f-graph queries and demonstrates that the containment problem $Q_f \sqsubseteq W_t$ between an f-graph query and a tree query can be solved in polynomial time. We also present the mv-index structure, a novel indexing structure for tree queries, that allows to check containment between an f-graph query and a query workload in worst case linear time w.r.t. the size of the indexed workload. Finally we show how to apply our algorithm for much more expressive queries with the introduction of tree and f-graph witnesses, i.e. trees and 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 the future 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.

11. REFERENCES

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

[2] `http://dsg.uwaterloo.ca/watdiv/stress-workloads.tar.gz`.

[3] `https://github.com/h31nr1ch/the-berlin-benchmark/blob/master/consults/rdf.sql`.

[4] `https://github.com/ldbc/ldbc_snb_implementations`.

[5] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 18(2):385–406, 2009.

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

[7] 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.

[8] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of rdf data management systems. In *International Semantic Web Conference*, pages 197–212. Springer, 2014.

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

[10] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix bit loaded: a scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web*, pages 41–50. ACM, 2010.

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

[12] C. Bizer and A. Schultz. The berlin sparql benchmark, 2009.

[13] D. Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. 2004.

[14] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

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

[16] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 59–70. ACM, 1993.

[17] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. Sparql query containment under rdfs entailment regime. In *International Joint Conference on Automated Reasoning*, pages 134–148. Springer, 2012.

[18] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Database Theory—ICDT'97*, pages 56–70, 1997.

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

[20] M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Owl web ontology language reference. *W3C Recommendation February*, 10, 2004.

[21] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auF der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[22] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis. Optimizing query shortcuts in rdf databases. *The Semanic Web: Research and Applications*, pages 77–92, 2011.

[23] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[24] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[25] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *Proceedings of the VLDB Endowment*, 5(2):97–108, 2011.

[26] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

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

[28] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM, 2014.

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

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

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

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

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

[34] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, 35(1):146–160, 1988.

[35] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM Trans. Database Syst.*, 26(4):388–423, 2001.

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

[37] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views. In *Proceedings of the*

fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 95–104. ACM, 1995.

[38] B. McBride. Jena: Implementing the rdf model and syntax specification. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 23–28. CEUR-WS. org, 2001.

[39] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos. Extended characteristic sets: graph indexing for sparql query optimization. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 497–508. IEEE, 2017.

[40] 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.

[41] K. Morfonios, S. Konakas, Y. Ioannidis, and N. Kotsis. Rolap implementations of the data cube. *ACM Computing Surveys (CSUR)*, 39(4):12, 2007.

[42] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[43] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee. Rdfox: A highly-scalable rdf store. In *International Semantic Web Conference*, pages 3–20. Springer, 2015.

[44] T. Neumann and G. Weikum. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, 2010.

[45] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware, workload-adaptive sparql query caching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1777–1792. ACM, 2015.

[46] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[47] R. Pichler and S. Skritek. Containment and equivalence of well-designed sparql. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 39–50. ACM, 2014.

[48] A. Polleres. From sparql to rules (and back). In *Proceedings of the 16th international conference on World Wide Web*, pages 787–796. ACM, 2007.

[49] E. Prud, A. Seaborne, et al. Sparql query language for rdf. 2006.

[50] E. Prudhommeaux. Sparql query language for rdf. *http://www. w3. org/TR/rdf-sparql-query/*, 2008.

[51] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4):633–655, 1980.

[52] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *International Semantic Web Conference*, pages 607–623. Springer, 2005.

[53] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.

[54] 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.

[55] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[56] 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.

[57] X. Zhang, L. Chen, Y. Tong, and M. Wang. Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In *Data engineering (ICDE), 2013 ieee 29th international conference on*, pages 565–576. IEEE, 2013.

APPENDIX

A. PROOFS

Proof of Proposition 1. We will prove the soundness and completeness of the claim by induction on the structure of the tree query. For the proof we assume a slightly different form of writing classes and properties that keeps the corresponding variables in W_{norm} as subscripts next to classes and properties. I.e. for the triple patterns $(x, \texttt{type}, A)$ and (x, p, y) we assume that the elements $A_{\langle x \rangle}$ and $p_{\langle x,y \rangle}$ are written in the corresponding normal form instead of A and p. The subscripts ${}_{\langle x \rangle}$, ${}_{\langle x,y \rangle}$ do not interfere with the operations of Algorithm 2 and are used to denote the corresponding variables. For convenience, we will use x, y to denote variables appearing in the normal form of a tree query W_{norm}, while s', t' will denote terms appearing in the f-graph Q_f. It should be noted that in the normal form of a tree query only predicates and classes appear. As presented in Section 4.1, IRIs and literals in the subject and object position of a triple pattern are expressed via nominal assertions.

Our proof is based on the normal form of tree queries, it is straightforwards to show that if a containment mapping exists for the normal form, then it also exists for the triple patternive query form. Because of nested subtrees, we will show the correctness of the proposition by induction in the structure of the normalised form of W_{norm}. We first need to prove for a term s' in Q_f; a nested subtree W'_{norm} with x being its root variable; and W'_{norm} appearing from the $(i+1)^{\text{th}}$ to the j^{th} position of W_{norm} that:

♠ the Containment function won't return False during its execution from position $i+1$ to j iff there exists a containment mapping m from the variables of W'_{norm} to the variables of Q_f for which $m(x) = s'$ applies.

Only If Direction. Suppose that Containment doesn't return False, then we will show that there exists such a containment mapping from W'_{norm} to Q_f with $m(x) = s'$ by induction in the structure of nested subtrees.

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'_{norm}. For the previous case, the substring from the $(i+1)^{\text{th}}$ to the j^{th} position of W_{norm} has the form:

$$\underset{\substack{\uparrow \\ i+1}}{\boxed{A_{i+1\,\langle x \rangle}}} \cdots \boxed{A_{k\,\langle x \rangle}}\ \boxed{p_{k+1\,\langle x, y_{k+1} \rangle}} \cdots \underset{\substack{\uparrow \\ j}}{\boxed{p_{j\,\langle x, y_j \rangle}}}\ . \qquad (1)$$

In the previous equation, all class and predicate assertions are related to the same root term x by construction of W_{norm}. We construct a new mapping m from the variables of W'_{norm} to the variables of Q_f and we set $m(x) := s'$. Since the Containment function does not return False, then for every $i+1 \leq l \leq k$ the class assertion $(s', \texttt{type}, A_l)$ in eq. 1 appears in Q_f (because of line 4 in Algorithm 2). Analogously, for every $k+1 \leq \kappa \leq j$ and the predicate p_κ there exists some κ such that $(s', p_\kappa, o'_\kappa)$ appears in Q_f (because of line 9 in Algorithm 2) and we set $m(y_\kappa) := o'_\kappa$. Since every triple pattern in W'_{norm} is mapped to a triple pattern in Q_f via m, m is a containment mapping from W'_{norm} to Q_f.

Induction Step. We now extend the proof for the case that elements of the form $\boxed{p_{l\,\langle x, y_l \rangle}}\ \boxed{(}\ W''_{\text{norm}} \boxed{)}$ appear in eq. 1 where W''_{norm} is the normal form of a corresponding subtree. We create a mapping m as before, only this time the mapping needs to be extended for the nested subtree W''_{norm}. From line 8, since the execution of the algorithm does not return False, we have that for the p_l predicate there exists some triple (s', p_l, o'_l) and we will set $m(y_l) := o'_l$. When the $(l+1)^{\text{th}}$ element $\boxed{(}$ is being examined, the algorithm will push the term s' into the $\vec{m}_{\text{path}}$ structure and the currently examined term in Q_f will take the value of o'_l (lines 13 and 14). By the induction hypothesis and since Containment does not return False, there is a containment mapping m' that maps every variable in W''_{norm} to a variable in Q_f for which it applies that $m'(y_l) = o'_l$. Since W_{norm} is a tree query, all the variables in W''_{norm} except from o'_l do not appear in any other part of W_{norm}. Therefore it is safe to extend the mapping m based on m' since there is no conflict between the two mappings. Finally line 16 in Algorithm 2 ensures that s' takes its previous value when exiting a subtree and if the Containment function does not return False the corresponding containment mapping for the rest of W'_{norm} exists.

If Direction. For the opposite direction, we assume that there exists a containment mapping m from the variables of W'_{norm} to the variables of Q_f for which $m(x) = s'$ and we need to show that calling Containment from the starting to the ending position of the W'_{norm} subtree won't return False when the currently examined term of Q_f is s'.

Induction Basis. For a nested subtree W'_{norm} with the normal form of Formula 1 and since there exists a containment mapping $m : W'_{\text{norm}} \rightarrow Q_f$:

- For every element $A_{l_{\langle x \rangle}}$ in W'_{norm}, $(s', \texttt{type}, A_l)$ will appear in Q_f and therefore line 6 in Algorithm 2 will never be accessed.
- Similarly, for every element $p_{l\,\langle x, y_l \rangle}$ in W'_{norm} there exists a term $o'_l = m(y_l)$ such that (s', p, o'_l) appears in Q_f. Therefore line 11 in Algorithm 2 will never be accessed and the algorithm won't return False.

Induction Step. For the induction step we need to prove that if the sequence $\boxed{p_{l\,\langle x, y_l \rangle}}\ \boxed{(}\ W''_{\text{norm}}\ \boxed{)}$ appears in eq. 1 the algorithm won't return False. As before, because of the existence of a containment mapping for which $m(x) = s'$, $m(y_l) = o'_l$ applies, we have that the triple pattern (s', p_l, o'_l) appears in Q_f and line 11 won't be accessed. When examining the parenthesis element, the algorithm will push the value of s' into the $\vec{m}_{\text{path}}$ stack and assign to s' the term in o'_l (lines 13, 14 in Algorithm 2). Based on the induction hypothesis, the algorithm won't return False until meeting the closing parenthesis of the W''_{norm} subtree. Since line 16 ensures that s' will take its previous value from $\vec{m}_{\text{path}}$ and since there exists a containment mapping the algorithm will return true for the rest of the normalised form of W'_{norm}. □

Proof of Theorem 1. In order to prove the soundness and completeness of the theorem, we first need to show the following claim:

♣ We assume that W_{norm} is the normal form of a query containing classes properties and parenthesis symbols and $\lambda_0, \lambda_1 \ldots \lambda_n$ are vectors of elements whose concatenation equals W_{norm}. For an f-graph Q_f and one of its terms s' it applies that:

$$\textsc{Containment}(W_{\text{norm}}, Q_f, s')$$

returns $\textsc{True}$ if and only if the consecutive execution of the function :

$$\textsc{Containment}(\vec{\lambda}_i, Q_f, s'_i, s'_{\text{next}\,i}, \vec{m}'_{\text{next}_i})$$

returns $(\textsc{True}, s'_{i+1}, s'_{\text{next}\,i+1}, \vec{m}'_{\text{next}_{i+1}})$ for all $0 \leq i \leq n$ such that $s_0 = s'$, $s'_{\text{next}0} = \textsc{Null}$, and $\vec{m}'_{\text{next}_0}$ is an empty stack of terms. The subsequent values of s_i, $s'_{\text{next}\,i}$, and $\vec{m}'_{\text{next}_i}$ are inferred from the previous execution cycle of the $\textsc{Containment}$ function.

- It should be noted that the call of the $\textsc{Containment}$ function corresponds 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 $\textsc{Containment}$ function. A detailed proof of the claim can be based on induction in the size of the $\lambda_0, \ldots, \lambda_n$ 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 $L(b_n) = W_{\text{norm}}$ which is the normal form of some query W_t represented within the mv-index.

Soundness. Suppose that the execution of the function

$$\textsc{ContainedQueries}(\mathfrak{M}, \alpha, Q_f, s', \textsc{Null}, \vec{m}_{\text{path}})$$

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 since $L_{Query}(b_n) = \textsc{True}$. We additionally need to show that b_n corresponds to the normal form of a query that contains Q_f. If b_n appears in V_{cont}, based on line 5 in Algorithm 3, it applies that consecutive executions of the $\textsc{Containment}$ function for the vectors $L(\alpha, b_1), L(b_1, b_2), \ldots, L(b_{n-1}, b_n)$ and the corresponding inputs $s'_0, \ldots s'_n$, $s'_{\text{next}0}, \ldots, s'_{\text{next}n}$, and $\vec{m}'_{\text{next}_0}, \ldots, \vec{m}'_{\text{next}_n}$ will all return $\textsc{True}$. The later along with Claim $\clubsuit$ and Proposition 1 imply the soundness of the theorem.

Completeness. For the completeness proof let's assume that the query W_t corresponding to $L(b_n)$ contains Q_f. By Proposition 1 and Claim $\clubsuit$ we have that the consecutive executions of the $\textsc{Containment}$ function for the vectors $L(\alpha, b_1)$, $L(b_1, b_2), \ldots, L(b_{n-1}, b_n)$ and the corresponding inputs s'_i, $s'_{\text{next}\,i}$, and $\vec{m}'_{\text{next}_i}$ will all return $\textsc{True}$. Line 2 in Algorithm 3 ensures that all vertices $\alpha, b_1, \ldots, b_n$ are examined therefore vertex b_n will be examined and the process will continue to assess if $L_{Query}(b_n)$ applies. $L_{Query}(b_n)$ was set to $\textsc{True}$ during the insertion phase of the query. Therefore line 9 of the algorithm will return node b_n to the corresponding answer set V_{cont}. $\square$

Proof of Proposition 2. We prove the proposition for each direction of the equivalence:

If Direction. Suppose that there exists a containment mapping $m : W \rightarrow Q$ from the variables of W to the variables of Q. We first build the mapping $m' : W_w \rightarrow Q$ from the variables of W_w to the variables of Q. For every variable x that both appears in W_w and W we set $m'(x) := m(x)$. If a variable z only appears in W_w then it was created along with the restriction $z = y$ in line 10 of the algorithm. For such a variable z, we set $m'(z) := m(y)$ and so the corresponding equality restriction in $\mathfrak{R}$ is satisfied by the mapping m'. By construction of m', it is straightforward that it is a containment mapping from W_w to Q and that also satisfies the equality restrictions in $\mathfrak{R}$ as we wanted to show.

Only If Direction. Suppose that there exists a containment mapping $m : W_w \rightarrow Q$ that also satisfied the equality restrictions in $\mathfrak{R}$. Then we need to show that there exists a containment mapping from W to Q. We will show that the corresponding containment mapping is m itself. For every triple pattern (x, p, y) in W that also appears in W_w, it is obvious that the triple pattern $(m(x), p, m(y))$ will also appear in Q by definition of m. For a triple pattern (x, p, y) in W that appears as (x, p, z) in W_w along with the restriction $y = z$, since m is a containment mapping from W_w to Q we have that $(m(x), p, m(z))$ appears in W. Since the equality restrictions in $\mathfrak{R}$ are also satisfied, we also have that $m(z) = m(y)$ and therefore the triple pattern $(m(x), p, m(y))$ also appears in Q as we wanted to show. In a similar way we prove the cases for inverse properties and it is shown that m is the corresponding containment mapping. $\square$

Proof of Propositions 3 & 4. We only need to prove Proposition 4. Proposition 3 directly follows from it. Suppose that $m : V \rightarrow Q$ is a containment mapping from V to Q. We build the mapping m_w from V to Q_w such that if s is a term in V, s' is a term in Q_w , and $m(s) = s'$ then we set $m_w(x) := [s']$. 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 singleton) set that contains them.

We now need to show that m_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 for every triple pattern $(s, \texttt{type}, C)$ appearing in Q, the triple pattern $([s], \texttt{type}, C)$ appears in Q_w and no more additional triple patterns are added to Q_w. $\square$

Proof of Proposition 5. Because of the restricted form of RDFS semantic relations (Table 1), we have that a single triple assertion can result to a set of triple assertions via the reasoning process, but RDFS rules do not consider combinations of triple assertions in their body (only a single triple pattern is allowed in the body of a rule). In the rest of the proof we will denote with $\mathcal{R}$ the RDFS schema under consideration.

If Direction. Suppose that there exists a containment mapping from V to the extended form of a query Q named Q_e. The latter implies that $Q_e \sqsubseteq V$ 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 $\mathcal{R}$, it is straightforward to show that m is also a solution for Q_e. Therefore, for every graph G that satisfies the RDF schema, it applies that $Q \sqsubseteq Q_e$. Since $Q_e \sqsubseteq V$ also applies for arbitrary graphs, based on the transitivity of the $\sqsubseteq$ relation, we have that $Q \sqsubseteq V$ for every graph that satisfies $\mathcal{R}$ schema as we wanted to show.

Only If Direction. Suppose that $Q \sqsubseteq V$ for every graph G that satisfies $\mathcal{R}$, we want to show that there exists a containment mapping from Q_e to V. We build a graph G as follows: (i) $f : Vars(Q_e) \rightarrow$ IRIs is an injective 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 $\mathcal{R}$. Since, based on our hypothesis, $Q \sqsubseteq V$ holds, we have that for every solution $\mu : Q \rightarrow G$ there exists a solution $\mu_2 : V \rightarrow G$. Each corresponding solution allows to infer a containment mapping $m : V \rightarrow Q_e$ by setting for every x variable $m(x) := f^-1(\mu(x))$ and our proof has finished. $\square$