Efficiently Pinpointing SPARQL Query Containments

Claus Stadler[1], Muhammad Saleem[1], Axel-Cyrille Ngonga Ngomo[2], and Jens Lehmann[3,4]

[1] Computer Science Institute, University of Leipzig, 04109 Leipzig, Germany
{cstadler|saleem}@informatik.uni-leipzig.de
[2] University of Paderborn, Warburger Str. 100, 33098 Paderborn, Germany
axel.ngonga@upb.de
[3] Smart Data Analytics Group, Computer Science Institute III, University of Bonn, 53117 Bonn, Germany
jens.lehmann@cs.uni-bonn.de
[4] Enterprise Information Systems Department, Fraunhofer IAIS, Germany
jens.lehmann@iais.fraunhofer.de

Abstract. Query containment is a fundamental problem in database research, which is relevant for many tasks such as query optimisation, view maintenance and query rewriting. For example, recent SPARQL engines built on Big Data frameworks that precompute solutions to frequently requested query patterns, are conceptually an application of query containment. We present an approach for solving the query containment problem for SPARQL queries – the W3C standard query language for RDF datasets. Solving the query containment problem can be reduced to the problem of deciding whether a sub graph isomorphism exists between the normalized algebra expressions of two queries.
Several state-of-the-art methods are limited to matching two queries only, as well as only giving a boolean answer to whether a containment relation holds. In contrast, our approach is fit for view selection use cases, and thus capable of efficiently enumerating all containment mappings among a set of queries. Furthermore, it provides the information about how two queries' algebra expression trees correspond under containment mappings. All of our source code and experimental results are openly available.

1 Introduction

Answering queries over views is a field within database research with several applications, such as in data integration, data warehousing and query optimization, question answering, and automatic composition of workflows [12]. In general, there are two main incentives: Performance improvements and data integration.

There are several scenarios where the evaluation of SPARQL queries over RDF datasets intrinsically suffers performance penalties without precomputation. For example, the S2RDF SPARQL engine [11], built on the Apache Spark

Big Data system, precomputes all joins between properties due to the way Spark processes joins.

As another example, consider a faceted browsing scenario, where a user dynamically filters for "Find all objects of type 'Bakery' that are located in 'Leipzig'" on a large dataset such as OpenStreetMap. Bakery is not selective, as there are several ten-thousands entities of that type world wide, and 'objects located in Leipzig' is not selective, as there are several ten-thousands other entities besides Bakery in the city. Yet, the size of the intersection between those two dimensions is comparatively low (around 250 entities). In data warehousing, this problem is known as the *sparse join*[5]. Join indexes are a type of materialized views used to address these problems.

We see the applications of this work as two fold: On the one hand, the system is aimed at providing a solid base for building advanced SPARQL caching solutions. On the other hand, the system opens up novel ways for studying SPARQL queries, such as those obtained from query logs. For example, how frequent are containments in practice? How many variants of semantically equivalent queries, such as retrieving all types, are typically used? Given a specific query workload, can one expect advantages from caching, and would a query-containment-based approach provide any advantage over a much simpler string based approach?

In this work, we make the following contributions:

- A system for SPARQL query containment that advances the state-of-the-art by supporting isomorphic containments.
- Thereby, we devise a hybrid graph index structure for fast in memory graph-based retrieval of isomorphic sub-graphs.
- We compare the performance and correctness of our system against four other systems in a benchmark.

The remainder is structured as follows: In Section 2, we introduce the preliminaries in regard to conjunctive queries, SPARQL, and query containment. In Section 3, we outline our approach to isomorphic query containment from a theoretical perspective. In Section 4, we present an efficient, generic and customizable implementation realized using a combination of state of the art components. Subsequently, relevant related work is discussed in Section 5. Our findings, obtained from running our system in a query containment benchmark, are presented in Section 6 Finally, Section 7 concludes this paper and presents directions for future work.

2 Preliminaries

In this section, we introduce fundamental concepts and terminology.

RDF Conceps: Let there be distinct sets of IRIs I, Blank Nodes B Literals L, and Variables V. The set $T = I \cup B \cup L$ is referred to as *RDF terms*. In accordance with

[5] `http://www.orafaq.com/tuningguide/sparse%20join.html`

the RDF standard, an RDF Graph G is defined as $G \subseteq (I \cup B) \times I \times (I \cup B \cup L)$, i.e. a set of triples with IRIs or blank nodes in the first position (also called subject), an IRI in the second position (also called predicate) and an RDF term in the third position (also called object). A generalized RDF Graph $G \subseteq T \times T \times T$ allows RDF terms as subject, predicate and object. For our work, we introduce the notions of an *Extended RDF Term* $E := T \cup V$ and *Extended RDF Graph* as $E \times E \times E$. We use the latter notion to model SPARQL queries as graphs thereby allowing constants and query variables to act directly as vertices (rather than as labels for nodes).

SPARQL Algebra and Semantics: SPARQL (SPARQL Protocol and RDF Query Language) is a standard language for querying and updating RDF data. SPARQL queries are formally executed against *RDF datasets*: An RDF dataset may contain zero or more *named graphs* and must contain one *default graph*.

A *solution binding* (short: binding) is a partial function which associates variables with RDF terms $\mu : V \to T$. The domain $dom(\mu)$ is the subset of V for which μ is defined. Two bindings μ_1, μ_2 are *compatible* if all common domain variables map to the same value, i.e. $\forall x \in dom(\mu_1 \cap \mu_2) : \mu_1(x) = \mu_2(x)$. In general, SPARQL result sets are multisets, which may be (partially) ordered.

In general, for the evalution of (SPARQL) algebra expressions, the Frege principle holds: The result of an evaluation of a whole is a function over the evaluation of its parts:

$$[\![O(a_1, \ldots, a_n)]\!] := \phi_O([\![a_1]\!], \ldots, [\![a_n]\!])$$

whereas ϕ_O is the semantic composition function corresponding to O.

Our work builds on SPARQL with multiset semantics (required for properly handling DISTINCT) as described in [1].

Definition 1 (Basic Query containment). *Query containment (CQ) is generally defined as:*

$$A \sqsubseteq B \text{ iff } [\![A]\!]_D \subseteq [\![B]\!]_D \text{ for every } D$$

Where A and B are queries and D stands for datasets.

Definition 2 (Containment Mapping and Homomorphic Query Containment).
A containment mapping *is a homomorphism $h : vars(v) \to vars(q) \cup consts(q)$ that maps the variables of a query (which we refer to as* view*) v to a query q, such that*

$$q \sqsubseteq_h v \text{ iff } q \sqsubseteq h(v)$$

We refer to basic query containment under containment mappings as homomorphic query containment*.*

Definition 3 (Transformational Query Containment and Equivalence). *This is a further generalization on query containment, which requires that there*

exists a sequence of unary operations that establishes the containment of a view in a query: Let $\Phi = \phi_1 \circ \ldots \circ \phi_n$ be a sequence of unary operations, with $\phi_i : Q \mapsto Q$, where Q is a set of (SPARQL) algebra expressions.

$$q \sqsubseteq_\Phi v \text{ iff } q \sqsubseteq \Phi(v)$$

Under this perspective, the application of a containment mapping is as special case of such operations, whereas examples of other operations that could be applied to establish a query containment include selection, projection, and distinct. For instance, according to Definition 1, if the result set of a query v had an additional column over that of q but was otherwise equivalent, there could not be a containment. Additional operations can also be used to establish query equivalence. An example is shown in Figure 1.

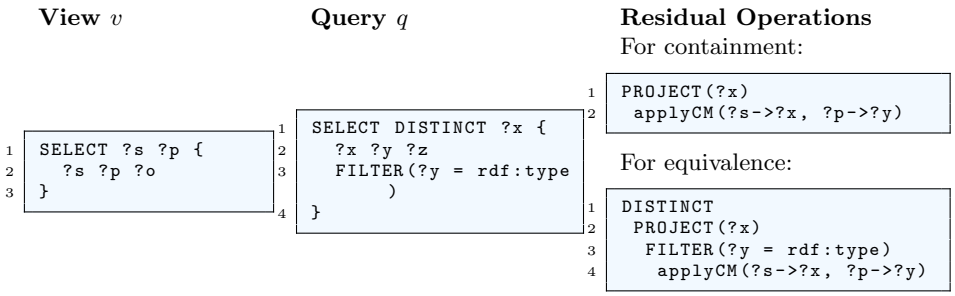

Fig. 1: In order to establish the containment $q \sqsubseteq_\Phi v$, the extra column $?p$ of the view must be projected away. In order to establish equivalence, additional filtering and distinct needs to be applied to view. Application of a containment mapping is denoted by *applyCM*.

Definition 4 (Algebra Expression Tree). *An algebra expression tree (AET) is a tree representation of a SPARQL algebra expression. For example, a join $A \times B$ can be represented as a tree with the three nodes A, B and the JOIN itself. The concrete representation we use is described in Section 3.2.*

Definition 5 (Conjunctive Queries). *Conjunctive queries, also known as* select-project-join queries*, are a sub-set of first-order logic queries that only allows conjunctions (i.e. logical and). Because of their simplicity they are of particular interest in studies.*

$$(x_1, \ldots, x_k).\exists x_{k+1} \ldots x_m.A_1, \ldots, A_r$$

The variables $x_1, \ldots, x_k$ are called distinguished variables *(i.e. projected variables), whereas $x_{k+1}, \ldots, x_m$ are referred to as* undistinguished. *A_i are* atomic *formulae over constants and variables, i.e. disjunction and negation are not permitted.*

View Selection: This is the problem of efficiently finding a set of candidate views among a set of views. The naive approach is to linearly scan all available views and test for whether a rewriting exists. However, as each containment check may be expensive due to the NP complete nature of the problem, significant performance improvements can be made by reducing the set of candidates. In this work, we devise an index structure which enables an efficient selection of candidate views on the basis of pruning the candidates for which no isomorphisms between the leaf nodes of the view to the query exist.

3 Approach

In this section, we describe our approach to homomorphic SPARQL query containment analysis, which can serve as the base for transformational QC. Given two SPARQL queries, one acting as a view v and the other as the request q, our basic approach is to first normalize their algebra expressions, such that if $q \sqsubseteq v$, v's AET would be a sub-tree of that of q in regard to a containment mapping. As leaf nodes of both AETs are conjunctive queries, the ones of v are converted to graphs and indexed using a subgraph isomorphism index (SII). For each of q's leaf AET graphs, an index lookup is performed in order to obtain a set of candidate leafs of v. On this basis, candidate matchings are enumerated, as shown in Figure 2. Note, that the index enables scaling the retrieval of candidate leaf graphs to a set of queries, which caters for candidate view selection use cases.

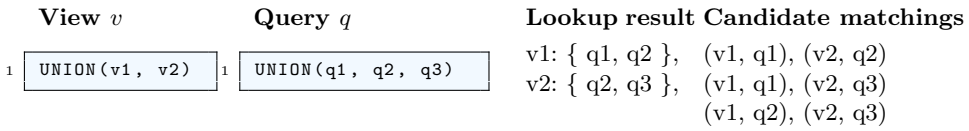

Fig. 2: Example of obtaining candidate matchings between the AET leaf graphs v_i and q_i of v and q, respectively. The candidate leafs are based on lookups in the SII. The set of candidate matchings is the corresponding *k-permutations of n* enumeration: If v1 mapped to q2, then we do not allow another v_i to map to it as well, hence {(v1, q2), (v2, q2)} is omitted.

Each candidate matching of the leaf graphs is based on a set of subgraph isomorphisms between CQs. The containment mapping for a single pair of CQs is obtained by simply deriving the mapping of their variables from that subgraph isomorphism. The containment mapping for the whole candidate matching is obtained by building the cartesian product over all individual containment mappings, and retaining those, whose union is *compatible*, i.e. a variable must not be mapped to different values. For every candidate matching and corresponding containment mapping, we employ bottom up scans in order to compute *node mappings* of the two involved queries' AETs. Hence, a major challenge is the

efficient computation of such candidate matchings under a set of views. Another challenge is to design the system in a way, such that additional normalizations and matching rules can be added or configured in a flexible way.

Figure 3 depicts our architecture to address these challenges.

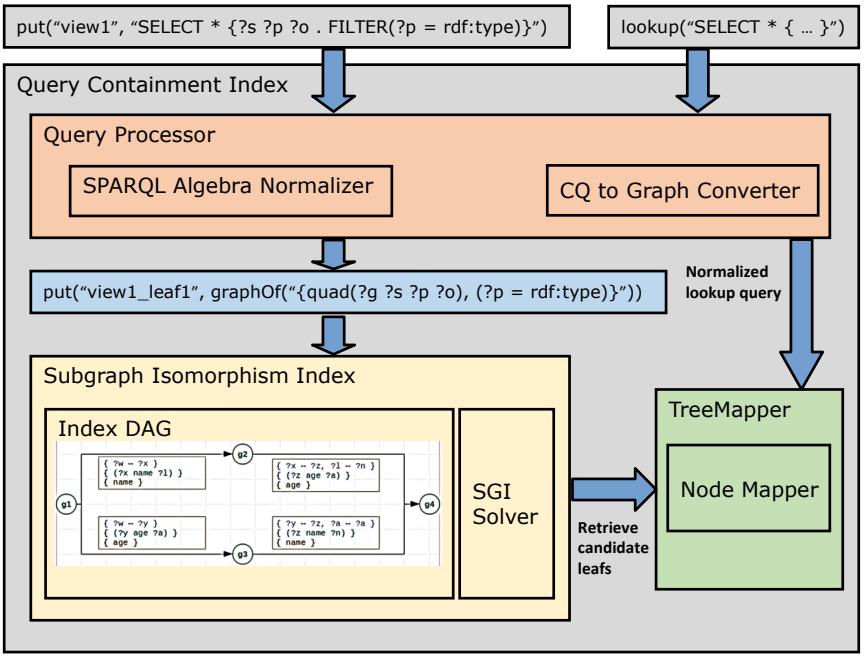

Fig. 3: Query Containment Engine Architecture

The most essential components are explained as follows:

- **Query Containment Index (QCI)**: This entity supports the *put(key, query)*, *remove(key)*, and *lookup(query)* operations which form the public API of the system. The result of *lookup(q)* is the for each matching *keys*, a set of *tree mapping* objects: The tree mapping comprises the detected containment mapping together with a map that associates each node in a query v's AET with one of the nodes of the lookup query q.
- **SPARQL Algebra Normalizer**: This component applies equivalence transformations to SPARQL queries, such that if a view was contained in a query, its AET would be a sub tree of that of the query. Most importantly, AETs are converted such that their leaf nodes become conjunctive queries.
- **CQ Graph Converter**: Obtains an extended RDF graph representation of the conjunctive queries. As it turns out, this approach exhibits some "natural" advantages: RDF terms (variables and constants) remain as such in this graph representation without the need for further conversion. Blank

nodes are used as a distinguished node type to represent syntactic elements, i.e. instances of triple/quad patterns and expressions. As a consequence, containment mappings can be directly obtained from detected subgraph isomorphisms by retaining only the mapped variables.
- **Subgraph Isomorphism Index (SII)**: A data structure for indexing the graphs of CQs. Supports finding all subgraph isomorphisms between the indexed graphs and a given query graph. When performing a lookup with a query q on the QCI, the QCI first obtains all CQ graphs of the leaf nodes of the normalized query q'. Afterwards, requests to the SII are made with the CQ graphs in order to establish a set of *candidate leaf matching* between q' and the queries in the QCI.
- **(Bottom-Up) Tree Mapper**: Component for performing a bottom-up scan of two trees under a candidate leaf matching.
- **Node Mapper**: This component is invoked by the bottom-up tree mapper for each pair of matching nodes of the view and the query's AET, in order to check for containment or equivalence by means of computing residual operations.
- **Subgraph Isomorphism Solver**: Our framework presently uses the VF2 algorithm, but it is designed to support any other algorithm for this purpose.

3.1 Query Normalization

Typical Query Normalization We apply several well known normalization techniques to the queries:

- Distribute joins over unions, i.e. $(A \cup B) \times C \rightarrow (A \times C) \cup (B \times C)$
- Filter placement: Any constraints are "pushed down" to the leaves of an algebra expression as close as possible. For example, $\sigma_e(A \cup B) \rightarrow \sigma_e(A) \cup \sigma_e(B)$. Ideally, filters become parents of quad blocks.
- Commutative binary operations, such as cross joins and unions, are converted to n-ary versions of the original ones. For example, $(A \times (B \times C)) \rightarrow \times_n(A, B, C)$. This makes testing semantic equivalence of two expressions easier, as it is avoids having to enumerate all possible equivalent expressions using the original binary operators.
- Merge consecutive filters into a single one, $\sigma_x(\sigma_y(\ldots)) \rightarrow \sigma_{x \wedge y}(\ldots)$
- Merge joins of quad blocks into a single quad block.
- Normalization of filter expressions to DNF for leaf nodes, and CNF for inner nodes in a query's AET.

Normalize leaves to Conjunctive Queries The leaves of SPARQL algebra expressions are either `VALUE` or quad nodes. Table 1 shows semantically equivalent queries, that differ by the constants appearing in the quad block and the filters. In order to treat these cases uniformly, we substitute every distinct constant c in a quad block with a corresponding fresh variable v (not appearing elsewhere in the query) and introduce an appropriate *FILTER(v = c)* expression. The filter placement optimization together with the merging of filters may add additional predicate expressions.

Next, we split the filter conditions into a conjunctive part and a remainder. For this purpose, we convert the whole expression to CNF. The conjunctive part is the set of clauses containing only a single expression, which means that there is no logical OR operation involved.

Now we have all information in place in order to obtain a normalized SPARQL algebra expression whose leaf nodes are conjunctive queries. The third column in Table 1 shows such an example.

As a consequence, in our case, containment mappings become mappings only between variables - instead of variables and constants.

```
SELECT * {
 ?s a ?o
 FILTER(?o = Bakery
   || ?o = Cafe)
}
```

```
SELECT ?s {
 ?s ?p ?o .
 FILTER(?p = rdf:type)
 FILTER(?o = Bakery
  || ?o = Cafe)
 }
```

```
FILTER(
 CQ(?s ?o,
  quadblock({?g ?s ?p ?o}),
  equals(?p, rdf:type)),
 ?o = Bakery || ?o = Cafe)
```

Table 1: Semantically equivalent SPARQL queries; third column uses a conjunctive query

3.2 Representation of Conjunctive Queries as Extended RDF Graphs

Here we describe how we represent conjunctive queries as graphs in order to be able to derive containment mappings from subgraph isomorphisms.

The basic idea is summarized as follows: Trivially, if two conjunctive queries are equivalent, so are their graphs. If two queries only differ by the naming of their variables, a graph isomorphism test will find the substitutions that would make them equal. And if a query q differs from a view v by having additional constraints, it means there exists a subgraph isomorphism from v to q.

The conversion of conjunctive queries to extended RDF graphs is quite direct: The projection is omitted. For every quad, a blank node is allocated which is described using *graph*, *subject*, *predicate*, *object* predicates.

The general transformation of expressions is as follows: Primitive expressions, namely variables and literals, are represented by themselves. For every compound (sub-)expression, a fresh blank node is allocated. Such a blank node carries a *:symbol* predicate to denote the operator or function name, whereas arguments are attached using : arg_i predicates, where i denotes the index of the argument. In the case of commutative operations, *:arg* (without index) is used.

Additional rules can be provided to cover more sophisticated cases: For instance, our system supports finding containments `contains(?v, ''ab'')` is covered by `contains(?v, ''a'')`. We accomplish this, by simply omitting the lit-

eral values that appear as the second argument of top-level[6] contains expressions in the graph representation.

This causes graph isomorphism algorithms to potentially map (the blank nodes of) two *contains* expressions with different arguments, and a subsequent semantic check tests for whether the omitted values are in a substring relation.

Note, that predicates are always constants (IRIs). Variables and literals of the query remain unchanged in the eRDF graph, and blank nodes correspond to entities (i.e. quads and expressions) that are subject for matching using subgraph isomorphism approaches. An example of a graph obtained from an expression is shown in Figure 4.

```
1  _:cq
2    :hasQuad _:q1 ;
3    :hasExpr _:e1 ;
4    :hasExpr _:e2 .
5
6  _:q1 :graph ?g ; :subject ?s ;
7    :predicate ?p ; :object ?o .
8
9  _:e1 :symbol "="; :arg ?p ;
10   :arg rdfs:label .
11
12 _:e2 :symbol "contains" ;
13   :arg1 ?o .
```

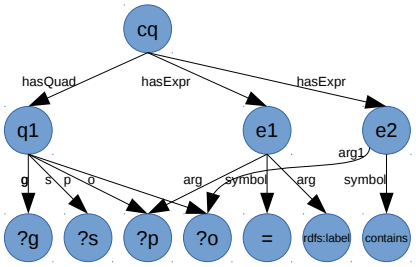

Fig. 4: Graph representation of the SPARQL query
*SELECT * { ?s ?p ?o . FILTER(?p = rdfs:label && contains(?o, 'Lu')) }*

4 Implementation

Our implementation is based on the Apache Jena Semantic Web Toolkit[7], the JGraphT[8] graph library, and our own Jena-based *Jena SPARQL API* toolkit[9].

Indexing with set tries An essential observation is, that all query graphs make use of sets of several constants, such as *rdf:type* and any other used URI and literal. These constants are invariant under isomorphism, i.e. if such an isomorphism exists, constants are mapped onto themselves. Hence, given query graph Q and a set of candidate graphs $\mathcal{C}$, then there can only be isomorphisms to a graph $C \in \mathcal{C}$ if Cs constants are a subset of that of G, i.e. $const(C) \subseteq const(Q)$. A datastructure for fast super and sub set queries, named *set trie*, is presented in [10].

[6] An expression is top-level if it is not a sub-expression
[7] `http://jena.apache.org/`
[8] `http://jgrapht.org/`
[9] `https://github.com/SmartDataAnalytics/jena-sparql-api`

Indexing with an isomorphism DAG Every vertex in the index *uniquely* corresponds to a (graph) key, whereas each edge represents a specific isomorphism between the corresponding graphs. Edges naturally represent sub-super graph relations between the graphs corresponding to an edge's source and target vertices. Conceptually, every edge holds three pieces of information: (1) The isomorphism mapping between A and B, referred to as iso, (2) the residual graph $B \setminus applyIsomorphism(A, iso)$ and (3) the residual graph tags. Note, that the residual tags is the set of tags by which the target graph B differs from the source graph A, i.e. $tags(B) \setminus tags(A)$.

We can use the set trie data structure for indexing a node's edges by its residual tags: If we were to lookup subgraphs at an index node n for a graph g, with tags T, then only graphs reachable by edges of n that provide a subset of T are applicable.

- put(I, K, G) Adds an entry that associates a key with a graph to the index
- getAllSubgraphsOf$(I, G) \rightarrow (K \rightarrow (V \rightarrow V))$ A function that finds all those entries in the index I that are subgraphs of G. For each matching graph, all isomorphisms are returned.

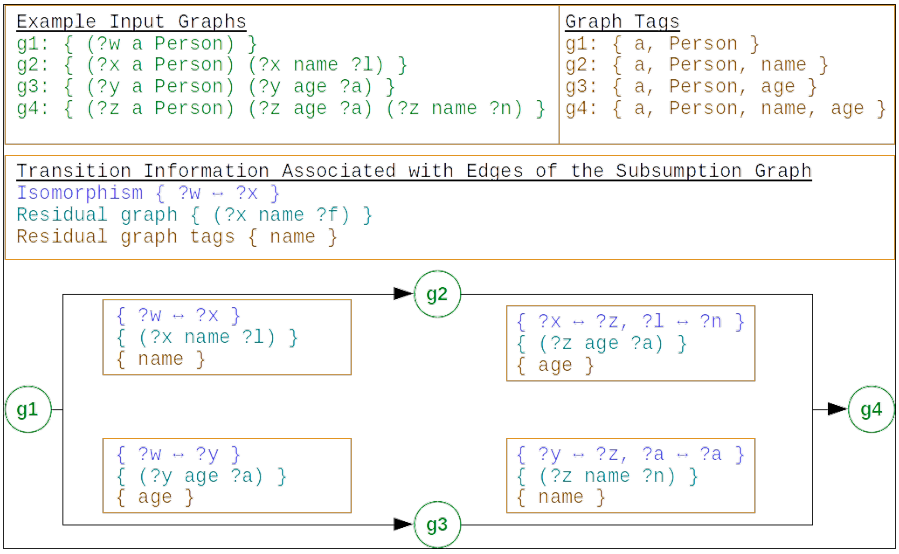

Fig. 5: Subgraph isomorphism subsumption graph created by the index

5 Related Work

Query optimization is concerned with the transformation of queries into equivalent ones that are less expensive to evaluate. In this context, conjunctive queries

are an extensively studied class of queries due to their practical relevancy and simplicity. Early work on minimizing conjunctive queries (which is somewhat similar to minimizing automatons) by eliminating redundant conjuncts was presented in [2]. However, theoretical studies on conjunctive queries at that time were mostly concerned with set semantics. Bag semantics are significantly more relevant in practice and are studied, e.g. in [3], or recently for SPARQL [1]. A survey of the closely related field of *query answering over views* given in [5], which classifies approaches in a taxonomy and provides an overview of further work handling extensions in view and query languages, such as unions and access pattern limitations.

The subgraph isomorphism problem is concerned with finding all embeddings of a graph in another. An extensive analysis of five representative (sub-) graph isomorphisms algorithms, namely VF2, QuickSI, GraphQL, GADDI, and SPath, has been performed in [6]. Although in the experiments there was no single winner, QuickSI had the best overall performance, only GraphQL was able to complete on all tests.

Early work on caching SPARQL queries was carried out in [7], where the authors describe a system for caching application domain objects and dependent SPARQL result sets. However, this work is unrelated to the rewriting queries using views field because cache entries for SPARQL queries are only determined based on hashes of their string representation. A SPARQL caching system that employs canonical graph labelling, such that isomorphic queries receive the same label, is described in [8]. A recent related system, although apparently not based on SPARQL, for fast super and subgraph queries by means of caching is presented [13]. Notably, this system integrates the aforementioned subgraph isomorphism implementations via Java bindings. Recently, SQCFramework [9] is proposed, a SPARQL query containment benchmark generation framework which generates customized SPARQL benchmarks from the real user queries log. The framework employs different clustering techniques to generate customized query containment benchmarks.

6 Evaluation

The main aspects we are interested in are: How does the performance of our system compare to existing query containment checkers, are there cases where we outperform the state-of-the-art, and is our system sufficiently fast for its application in query caching to be viable. We evaluate our system using the data from the *Inrialpes' SPARQL containment benchmark*[10][4] on a notebook with Intel I7-7700HQ CPU (2.80GHz), 16GB RAM, running Ubuntu 16.04. The original benchmark runner that is part of the benchmark suite required every single task to be run in a new VM, because some of the query containment checkers were not re-entrant. Hence, the original benchmark measured "cold" times, which include all the overhead of freshly launched Java VMs, such as

[10] `http://sparql-qc-bench.inrialpes.fr/`

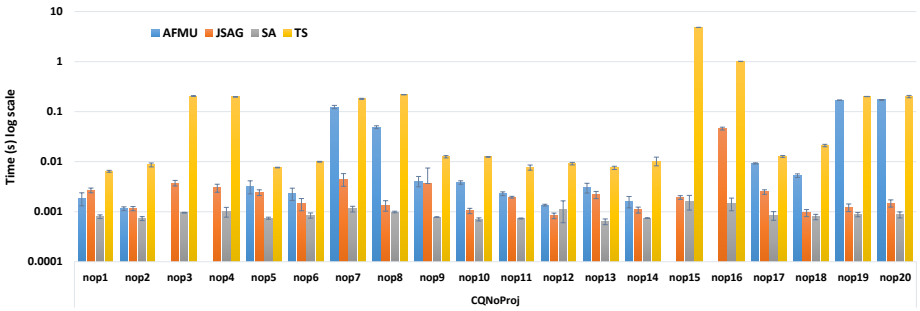

Fig. 6: CQNoProj performance chart across all tools - TS dominates.

just-in-time compilation. We managed to fix these issues[11], allowing to perform proper warm-up runs and thus obtain more realistic times.

This benchmark comprises three suites:

- **CQNoProj**: This suite defines a set of containment tests only based on conjunctive queries "without projection", i.e. `SELECT *` queries.
- **UCQProj**: A suite comprising union conjunctive queries with varying sets of distinguished and non-distinguished variables.
- **UCQrdfs**: Union conjunctive queries with projection with addition of RDFS entailments.

Our work does not consider reasoning, leaving us with 48 benchmark tasks from CQNoProj and UCQProj, with a total of 43 distinct queries. Our system yields the correct solution on 45 of these tasks, with the exception of UCQProj{#p24, #26, #27}. The reason is, that at present we require *all* UNION members of a view to have a corresponding member in the query. However, this can lead to false negatives: For example, a query *?s a Person* is contained in a view {*?s a Person*} *UNION* {*?s a Agent*}, although *?s a Agent* does not have a correspondence. Yet, such type of containments may be irrelevant to caching: if e.g. for any dataset `?s=Smith` was a solution of the view, it cannot be decided whether it is an answer to the query. Figure 6 and Figure 7 show the performance of the tools *AFMU*, *TreeSolver* (TS), *SparqlAlgebra* (SA) and our tool labeled *JSAG* (short for jena-sparql-api graph-isomorphism based query containment solver). Figure 6 is dominated by TS, followed by AFMU. Figure 8 is a comparison of our tool with SA. In most cases, JSAG performs only slightly worse than SA, which does not support containment mappings. Whereas SA is capable of solving all supported tasks with millisecond performance, there are only 3 instances where JSAG performed significantly worse but still within the one hundredth of a second range. JSAG's worst performance is at 0.046 seconds on CQNoProj#16, however more investigation is needed to determine the reason for the large performance difference to SA. The evaluation shows, that our

[11] This was achieved by resetting some private global variables using Java reflection.

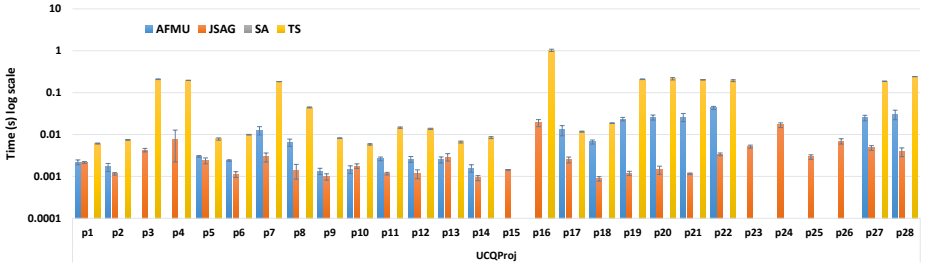

Fig. 7: UCQProj performance chart across all tools - TS dominates.

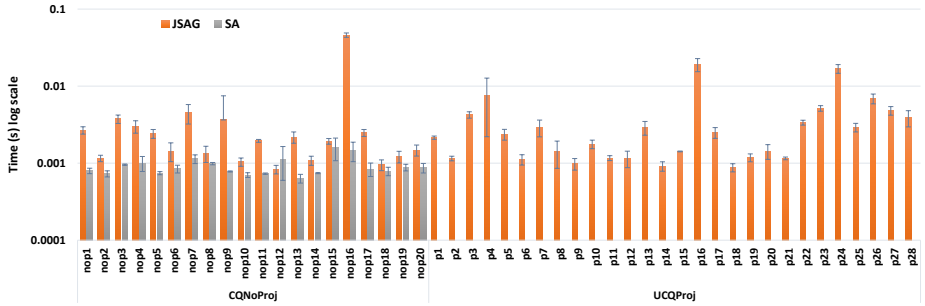

Fig. 8: Performance chart of JSAG and SA.

system not only outperforms existing solutions, but its performance is promising for the intended caching and analysis use cases.

7 Conclusions and Future Work

In this paper, we present a system for efficiently performing query containment checks. This is accomplished by means of normalizing algebra expressions, transforming the leaf nodes of AETs to conjunctive queries (CQs), converting CQs to (extended) RDF graphs and applying graph isomorphism testing to determine whether the leaf nodes of two AETs can be matched. Once a candidate matching between AET leaf nodes has been established, a bottom-up scan is performed to determine whether one query appears as a sub-(expression)-tree in the other, thus providing (incomplete) solutions to the query containment problem. The evaluation shows, that our system outperforms some state-of-the-art SPARQL query containment checkers, and it is only slightly slower than the fastest tool, which does not support containment mappings.

One direction of future work is to leverage this system as a building block for realizing advanced SPARQL precomputation and caching solutions. SPARQL-based faceted browsing and Big Data RDF processing may be among the areas

that could benefit the most from such systems. Furthermore, this work enables analysing novel aspects of SPARQL query workloads.

All of our components, the revised query containment benchmark and the raw benchmark result data (in RDF and CSV) are publicly available at [12].

Acknowledgements

This work was partly supported by the grant from the European Unions Horizon 2020 research Europe flag and innovation programme for the projects HOBBIT (GA no. 688227), QROWD (GA no. 732194) and WDAqua (GA no. 642795).

[12] `https://github.com/SmartDataAnalytics/jena-sparql-api/tree/master/jena-sparql-api-query-containment`

Bibliography

[1] R. Angles and C. Gutiérrez. The multiset semantics of SPARQL patterns. In *International Semantic Web Conference (1)*, volume 9981 of *Lecture Notes in Computer Science*, pages 20–36, 2016.

[2] 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.

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

[4] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. Evaluating and benchmarking SPARQL query containment solvers. In *International Semantic Web Conference*, pages 408–423. Springer, 2013.

[5] A. Halevy. Answering queries using views - a survey. *The VLDB Journal*, 10(4):270–294, 2001.

[6] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[7] M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with SPARQL query result caching. To appear in proceedings of 7th Extended Semantic Web Conference (ESWC2010), Heraklion, Crete, Greece,, May 30 - June 03 2010.

[8] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware, workload-adaptive SPARQL query caching. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *SIGMOD Conference*, pages 1777–1792. ACM, 2015.

[9] M. Saleem, C. Stadler, Q. Mehmood, J. Lehmann, and A.-C. N. Ngomo. Sqcframework: SPARQL query containment benchmark generation framework. In *Proceedings of the Knowledge Capture Conference*, K-CAP 2017, pages 28:1–28:8, New York, NY, USA, 2017. ACM.

[10] I. Savnik. Index data structure for fast subset and superset queries. In A. Cuzzocrea, C. Kittl, D. E. Simos, E. R. Weippl, and L. Xu, editors, *CD-ARES*, volume 8127 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2013.

[11] A. Schtzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF querying with SPARQL on spark. *CoRR*, abs/1512.07021, 2015.

[12] K. Singh, I. Lytra, M.-E. Vidal, D. Punjani, H. Thakkar, C. Lange, and S. Auer. Qaestro - semantic-based composition of question answering pipelines. In D. Benslimane, E. Damiani, W. I. Grosky, A. Hameurlain, A. P. Sheth, and R. R. Wagner, editors, *DEXA (1)*, volume 10438 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2017.

[13] J. Wang, N. Ntarmos, and P. Triantafillou. Graphcache: A caching system for graph queries. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K.-U. Sattler, and S. Bre, editors, *EDBT*, pages 13–24. OpenProceedings.org, 2017.