Evaluating an Automata Approach to Query Containment

Michael Minock
KTH Royal Institute of Technology, Stockholm, Sweden
Umeå University, Umeå, Sweden.
minock@kth.se, mjm@cs.umu.se

Abstract

Given two queries Q_{super} and Q_{sub}, query containment is the problem of determining if $Q_{sub}(D) \subseteq Q_{super}(D)$ for all databases D. This problem has long been explored, but to our knowledge no one has empirically evaluated a straightforward application of finite state automata to the problem. We do so here, covering the case of conjunctive queries with limited set conditions. We evaluate an implementation of our approach against straightforward implementations of both the canonical database and theorem proving approaches. Our implementation outperforms theorem proving on a natural language interface corpus over a photo/video domain. It also outperforms the canonical database implementation on single relation queries with large set conditions.

1 Introduction

Given the queries Q_{super} and Q_{sub}, *query containment* is the problem of determining if $Q_{sub}(D) \subseteq Q_{super}(D)$ for all databases D. Not only interesting in itself, the problem is of practical importance in query optimization, data integration (Ullman, 2000) and, of note here, in natural language generation (Shieber, 1993), dialogue (Bos and Oka, 2002) and understanding (Minock, 2017).

Over time, increasingly complex cases of the problem have been solved: relational conjunctive queries (Chandra and Merlin, 1977); conjunctive queries with arithmetic comparisons over dense domains (Klug, 1988); negation of subgoals (Levy, 1999). More recent work[1] has looked at the problem for queries with aggregate operators (see the survey (Cohen, 2005)). Remarkably, the decidability of the problem remains open for queries under bag semantics (Afrati et al., 2010).

A typical approach to solving query containment is to generate a *canonical database* D' that represents Q_{sub} and then to evaluate $Q_{super}(D')$. If the answer to Q_{sub} within D' is within $Q_{super}(D')$, then Q_{super} contains Q_{sub} (see (Ullman, 2000)). Another approach to solve the problem is via *theorem proving*. If ϕ is the translation of Q_{super} to a first order formula free over the answer variables of Q_{super}, and likewise φ for Q_{sub}, then, assuming that the queries are compatible (i.e. $free(\phi) = free(\varphi)$), $\neg\texttt{SAT}(\exists free(\phi)(\neg\phi \wedge \varphi))$ if and only if Q_{super} contains Q_{sub}. A third approach, and what we look at here, is to use reduce query containment to determining the if a finite state automaton recognizes the empty language. Here we present, implement, and empirically evaluate such an approach. We compare performance against a canonical database and theorem proving implementation over a photo/video querying corpus. We also conduct several special scalability tests for queries with many predicates over the same relation and queries with large set conditions over a single relation.

2 Preliminaries

As is common, we present queries here in DATALOG, with which we assume the reader is familiar (see (Ullman, 1988)). As a quick refresher, under the database state and queries of Figure 1, the answer to Q1 is `{(h), (i)}` and the answer to Q2 is `{(h)}`. In fact no matter what the state of the database is, answers of Q2 are always contained in the answers of Q1. Likewise, Q3 contains Q1.

Before continuing, it is worth giving example runs of the canonical database and theorem prov-

[1] The problem has also been addressed in semi-structured query languages (Baumgartner et al., 2005; Björklund et al., 2011) and in description logics (Baader et al., 2009). Still the focus here is on query containment for relational databases.

Proceedings of the 13th International Conference on Finite State Methods and Natural Language Processing, pages 75–79,
Umeå, Sweden, 4–6 September 2017.
https://doi.org/10.18653/v1/W17-4010

```
R(A, B, C)    S(D, E)    T(F G)
  h  i  j       h  r
  i  k  l       i  b
  j  m  n       j  b
                k  r

Q1(X) :- R(X,Y,Z), S(Y,P)
Q2(X) :- R(X,Y,Z), S(Y,'b'), S(Z,'b')
Q3(X) :- R(X,Y,Z)
```

Figure 1: Example database state and queries.

ing approaches for deciding whether Q1 contains Q2. Under the canonical database approach, each variable in Q_{sub} (i.e. Q2) generates a fresh constant (we use the natural numbers). Predicates in the body of the conjunctive query Q_{sub} are frozen with these constants and generate tuples in the canonical database. Thus the canonical database state for Q2 is `{R(1,2,3),S(2,'b'),S(3,'b')}` with the frozen answer being `{(1)}`. It is easy to verify that Q1 evaluated over this database state, generates the frozen answer and thus Q1 contains Q2. In the theorem proving approach, the sentence sent to the SAT solver is $(\exists x)(\neg(\exists y, z, p)(R(x, y, z) \wedge S(y, p)) \wedge (\exists y, z, p1, p2)(R(x, y, z) \wedge S(y, p1) \wedge S(y, p2) \wedge p1 = \text{'b'} \wedge p2 = \text{'b'}))$. We now turn to our finite state automata approach.

3 An Automata-based Approach

The approach we develop here is quite straightforward. In short, for a query Q, we build the automaton M_Q which recognizes the language of database state encodings which generate non-empty answers to Q. Given the closure properties of regular languages, the query containment problem reduces to determining if $L(\overline{M}_{Q_{super}}) \cap L(M_{Q_{sub}})$ is empty.

Under simple runs of building M_Q, the resulting automaton is linear with a single final state. More complex runs require branching over variable settings, resulting in a tree shaped automaton with multiple final states. In general post processing is required to bypass parts of the automaton for which a witnessing tuple have already been consumed. Finally an assumption that tuples appear in a fixed lexicographic order compresses the automaton. The remainder of the section presents our approach in greater detail.

3.1 Encoding Database States

By way of example, the database state in Figure 1 is encoded as `hij_ikl_jmn_#hr_ib_jb_kr_##`. We are using the symbol _ to close off tuples and the symbol # to close off relation states. An *encoding scheme* orders a finite set of relations (and their attributes) and specifies which constants may appear under which attributes.

3.2 Fixing a Minimal Encoding Scheme

When deciding if Q_{super} contains Q_{sub}, we fix a minimal encoding scheme that covers both queries. First we collect the relations used in the bodies of both queries and, for efficiency, truncate these relations to include only the attributes used in join conditions, simple conditions or the head of a query. For example, in the containment problem determining if Q3 contains Q1, `R` is truncated down to two attributes. Given this we then determine an arbitrary ordering over the truncated relations. To add constants under variables, reminiscent of the canonical database approach, variables and constants of Q_{sub} are frozen and added as constants (e.g. `X` becomes `'x'`, and `'b'` remains `'b'`) under their corresponding attributes.

For example, for the problem if Q1 contains Q2, we collect the database relations `R` and `S`. Neither may be truncated. Arbitrarily we determine the relation ordering to be `[R,S]`. The constants under the attributes are `'x'` under `A`, `'y'` under `B`, `'z'` under `C`, `'y'` and `'z'` under `D` and `'b'` under `E`.

3.3 Building the Automaton M_Q

Given a query Q and an encoding scheme, we truncate and sort the predicates in the body of Q based on the encoding scheme. We then walk the new truncated/sorted query body from left to right constructing an automaton as we go. For each relation in the encoding, we construct what we term a *relation gobbler*. These relation gobblers consume associated relations in the input database state expression. They are chained together using a transition on the symbol # from the last state of one gobbler to the first state of the next.

In the normal case, in which the relation appears in a predicate in the truncated/sorted query body, the relation gobbler consists of what we term a *tuple gobbler*, followed by a *witness gobbler*, followed by a *tuple gobbler*. Tuple gobblers non-deterministically consume irrelevant tuples. The witness gobbler recognizes a tuple that matches the current query predicate. In the case in which the relation name does not appear in a predicate of the truncated/sorted query[2], the relation gobbler

[2]This occurs because the relation name *does* appear in a

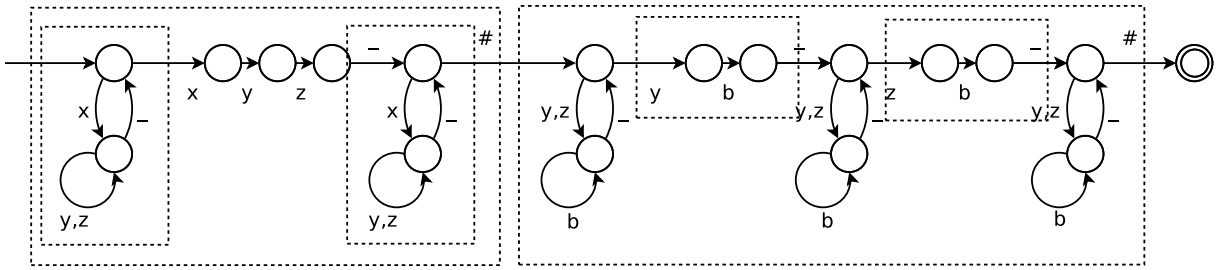

Figure 2: A NFA that recognizes a language that includes all encoded database states which generate answers to Q2 under the example encoding of 3.2, and no database state that does not generate answers.

consists of just a tuple gobbler. A special case occurs with predicates over the same relation name in a query. The relation gobbler for such cases is a chained sequence of tuple-witness-tuple gobblers.

The exact algorithm of this construction method is omitted here, but it is achieved via a fairly simple recursive function. Once the base case is reached, all the relations in the encoding have been treated and a final state after the last # is marked as accepting. The automaton in Figure 2 recognizes database states yielding answers to Q2 under the example encoding scheme in section 3.2. It consists of two relation gobblers, where the tuple gobblers are highlighted in the first and the witness gobbler in the second. Note that this automaton did not branch because there is only one constant under Y and only one constant under Z.

3.3.1 Branching on Variable Assignments

As we scan the predicates of the query body, a query variable may be bound to more than one constant. In such cases we branch the automaton to the alternative possibilities. For example, consider the encoding scheme of section 3.2, but with the relation ordering `[S,R]`. The automaton in Figure 3 shows the automaton (tuple gobblers are not expanded for reasons of space) that recognizes answers to Q2 under this alternative encoding scheme. This requires branching[3].

3.3.2 Witnesses over Multiple Predicates

When queries are self-joining (i.e. two or more predicates are over the same relation in the query body), the same tuple might need to serve as a witness in more than one witness gobbler. For example, consider the somewhat contrived `Q4(X) :- R(X,Y,Z),R(X,Y,Z).` Clearly this is equivalent to `Q3(X)`. This is handled by walking the tree shaped (or linear) automaton, keeping track of which witnesses have been used so far. Any predicate gobbler that is already witnessed is bypassed and removed.

3.3.3 Ordered Database State Assumption

The arbitrary order in which witnesses might appear requires quite deep trees many of which check the same cases. We may invoke an assumption that witnesses may only appear in some given ordering in database states and easily enforce this in the recursive automata shortening routine of section 3.3.2. As will be shown, this can lead to considerable performance improvement.

3.4 The Containment Test

Given that the automaton for both Q_{super} and Q_{sub} are constructed over the same encoding scheme, we may construct an automaton that recognizes $L(\overline{M}_{Q_{super}}) \cap L(M_{Q_{sub}})$ which is empty when Q_{super} contains Q_{sub}.

4 Evaluation

Our approach (**FSA**) is implemented in Python and for most of its automata routines it uses PADS, a library of Python Algorithms and Data Structures implemented by David Eppstein of the University of California, Irvine. Although not described above, we extended FSA to handle limited set conditions. The current set conditions supported are set conditions on non-joining attributes over relations that appear in only one predicate of a query. We have also implemented Python versions of the canonical database (**CDB**) and the theorem proving (**TP**) approaches. The database system used in CDB is SQLITE running in main memory; performance deteriorates by several orders of magnitude if the database must be written to disk. Our CDB implementation is extended to set con-

predicate of the other query in the containment problem.

[3] In this particular case only one of the branches ultimately succeeds due to possible constants under attributes, but in general such branching can lead to multiple final states.

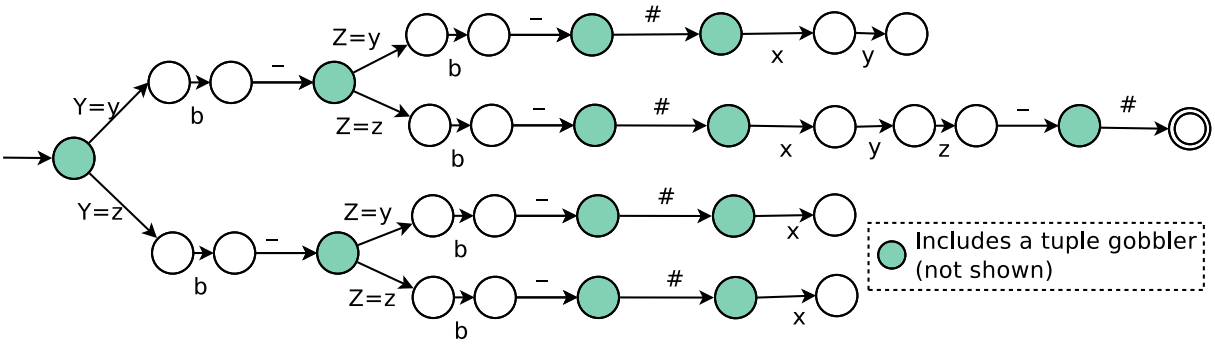

Figure 3: A NFA that recognizes Q2 under the relation ordering `[S,R]`.

Approach	Avg	Max	Min
CDB	0.3	1.5	0.1
TP	11.3	33.8	7.6
FSA	1.9	23.6	0.1

Table 1: Results over photo/video corpus (ms)

Approach	Case 1	Case 2	Case 3
CDB	1.0	0.6	1703
TP	62009	64.3	56235
FSA	0.9	294.9	222

Table 2: Results for special cases (ms)

ditions by converting queries using set conditions into unions of (non-set) conjunctive queries. Each simple non set query in the conjunction is frozen and inserted into the same database state rather than the less efficient technique of building separate canonical databases for each combination. For TP, we use PROVER9.

We evaluate all three implementations using a corpus of natural language queries over a photo/video domain (available at `sites.google.com/view/nli-corpora/`). The corpus consists of a schema and a set of 100 natural language questions (e.g. "give photos of Alice with Bob in London in 2016".) paired with corresponding SQL queries. The portion of the corpus that we run our evaluation over are 27 queries over the `Picture` table restricted to conjunctive queries with simple and limited set conditions. Obviously CDB, TP and FSA each return the same containment determinations. Table 1 gives performance results over the $27 \times 27 = 729$ problems.

We also run a series of scalability tests to determine performance over long queries with multiple predicates over the same relation and queries with large set conditions on a single relation. To motivate, consider applications where arbitrary knowledge, constraints or complex view definitions are added to the query containment problem. Also consider cases where common names denote large sets or contexts sets are represented. To get insight into such cases, we measure performance of long queries over a single edge relation (case 1,2) and over large set conditions (case 3). Specifically the first case is exactly example one in (Chandra and Merlin, 1977) which gives a contrived query which, through reasoning, can be radically simplified. The second case, also derived from the same article, is based on reducing graph colorability of graph rings of length 9 and 10 are 2-colorable (`False` for 9, `True` for 10). Finally our third case is over a single relation of 5 attributes, where queries have set conditions over these attributes with between 7 and 14 distinct values. Table 2 shows results for these scalability tests.

In summary FSA performs reasonably well on the photo/video corpus and, in the case of large sets it performs better than CDB. TP performs the worst on the photo/video corpus and is dangerously vulnerable to blow up on the scalability cases 1 and 3. We view scalability case 1 as a *haphazard case* in which many predicates may be satisfied by the same witness. Case 2 is a *purposeful case* in which an NP-Complete problem is being constructed and separate witnesses are matched to each predicate. FSA does the poorest under case 2 and even worse (561 ms) if we remove the optimization of section 3.3.3. That said, we posit that case 1 is much more likely to fit real world problems in which database constraints and view definitions are added to containment problems. Thus we are not alarmed by FSA's relative weakness on case 2.

5 Discussion

It has long been recognized that natural language questions over databases require quite advanced semantics, going well beyond the simple conjunctive query case (Copestake and Jones, 1990). While we cover queries that use simple set conditions, enabling us to represent and/or ambiguities (e.g. *"photos of Alice and Bob"*), we still do not yet cover questions expressing point inequalities (e.g. *''photos of Alice not taken on June 1,2017"*), sub-goal negation (e.g. *"photos of Alice without Bob"*), superlatives (e.g. *"latest photo of Alice with Bob"*, cardinality conditions (e.g. *"who appears in the most pictures?"*) or non-recursive DATALOG with negated sub-goals (e.g. *"is Bob in every picture in London in 2016?"*, etc. These and many other hard examples appear in our photo/video corpus. Finally we need to integrate key constraints (e.g. *"every video is taken in some location"*, *"no two distinct videos are stored in the same file"*) or knowledge into the containment determinations (e.g. *"Manhattan is in New York"*).

As we extend FSA, it may yield insight into these problems as well as performance advantages. For example, sub-goal negation, in general, requires the canonical database approach to consider a combinatorial number of databases. Our approach might only require special non-spoiler gobblers linked to spoiler gobblers linked to a dead-end states. Also we suspect that encoding arithmetic constraints can be managed via branching over alternative variable orderings in our recursive automaton construction method. Finally, we speculate that our approach might be brought to more powerful automata to capture containment over ever more expressive query classes. It will be interesting to see how far we can get.

6 Conclusions

This paper evaluated a finite state automata approach to determining relational query containment. For conjunctive queries with limited set conditions the approach showed itself to be competitive with canonical database and theorem proving approaches. The approach still needs to be formally proven correct, and, if there are counterexamples, we must either extend the approach or develop natural assumptions to limit it to correct cases. Either way, the approach has been empirically validated under a fairly realistic corpus as well as for several special scalability tests. Future work will focus on extending the approach to cover more and more of the examples in our photo/video corpus (available at `sites.google.com/view/nli-corpora/`).

Acknowledgments

Umeå University masters student David Hansson built the initial CDB and TP implementations.

References

Foto Afrati, Matthew Damigos, and Manolis Gergatsoulis. 2010. Query containment under bag and bag-set semantics. *Inf. Process. Lett.*, 110(10):360–369.

Franz Baader, Ian Horrocks, and Ulrike Sattler. 2009. Description logics. In *Handbook on Ontologies*, pages 21–43.

Robert Baumgartner, Oliver Frölich, Georg Gottlob, Marcus Herzog, and Peter Lehmann. 2005. Integrating semi-structured data into business applications. In *Professional Knowledge Management, WM Kaiserslautern, Germany*, pages 469–482.

Henrik Björklund, Wim Martens, and Thomas Schwentick. 2011. Conjunctive query containment over trees. *J. Comput. Syst. Sci.*, 77(3):450–472.

Johan Bos and Tetsushi Oka. 2002. An inference-based approach to dialogue system design. In *COLING Taipei, Taiwan, August 24 - September 1, 2002*.

Ashok Chandra and Philip Merlin. 1977. Optimal implementation of conjunctive queries in relational databases. In *Proc. of STOC*, pages 77–90.

Sara Cohen. 2005. Containment of aggregate queries. *SIGMOD Record*, 34(1):77–85.

Ann Copestake and Karen Sparck Jones. 1990. Natural language interfaces to databases. *Knowledge Eng. Review*, 5(4):225–249.

Anthony Klug. 1988. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160.

Alon Levy. 1999. Review - complexity of answering queries using materialized views. *ACM SIGMOD Digital Review*, 1.

Michael Minock. 2017. Cover: Covering the semantically tractable question. In *proceedings of EACL (Software Demonstrations)*, Valencia, April.

Stuart Shieber. 1993. The problem of logical-form equivalence. *Computational Linguistics*, 19(1):179–190.

Jeffrey Ullman. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.

Jeffrey Ullman. 2000. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210.