This is the accepted version of a paper presented at 13th International Conference on Finite State Methods and Natural Language Processing (FSMNLP), 4-6 September, Umeå.

Citation for the original published paper:


N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-144608
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 $\phi$ is the translation of $Q_{super}$ to a first order formula free over the answer variables of $Q_{super}$, and likewise $\varphi$ for $Q_{sub}$, then, assuming that the queries are compatible (i.e. $\text{free}(\phi) = \text{free}(\varphi)$), $\neg\text{SAT}(\exists\text{free}(\phi)(\neg\phi \land \varphi))$ 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 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 DATA-LOG, 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 $Q_1$ is $\{(h), (i)\}$ and the answer to $Q_2$ is $\{(h)\}$. In fact no matter what the state of the database is, answers of $Q_2$ are always contained in the answers of $Q_1$. Likewise, $Q_3$ contains $Q_1$.

Before continuing, it is worth giving example runs of the canonical database and theorem prov-
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 gobber, followed by a witness gobber, followed by a tuple gobber. Tuple gobblers nondeterministically consume irrelevant tuples. The witness gobber 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

\[ R(A, B, C) \quad S(D, E) \quad T(F, G) \]

\[ h \quad i \quad j \quad h \quad r \]

\[ i \quad k \quad l \quad i \quad b \]

\[ j \quad m \quad n \quad j \quad b \]

\[ k \quad r \]

\[ Q1(X) := R(X, Y, Z), \ S(Y, F) \]

\[ Q2(X) := R(X, Y, Z), \ S(Y, 'b'), \ S(Z, 'b') \]

\[ Q3(X) := R(X, Y, Z) \]

---

\(^{2}\)This occurs because the relation name does appear in a
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 \(L(M_{Q_{super}}) \cap L(M_{Q_{sub}})\) 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-

---

\(^3\)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.
conditions 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 hazardous 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


