Query Containment in Entity SQL (Extended Abstract)

Guillem Rull[1]
Univ. Politècnica de Catalunya
grull@essi.upc.edu

Philip A. Bernstein
Microsoft Corporation
philbe@microsoft.com

Ivo Garcia dos Santos
Microsoft Corporation
ivosan@microsoft.com

Yannis Katsis[1]
UC San Diego
ikatsis@cs.ucsd.edu

Sergey Melnik[1]
Google, Inc.
melnik@google.com

Ernest Teniente[2]
Univ. Politècnica de Catalunya
teniente@essi.upc.edu

ABSTRACT

We describe a software architecture we have developed for a constructive containment checker of Entity SQL queries defined over extended ER schemas expressed in Microsoft's Entity Data Model. Our application of interest is compilation of object-to-relational mappings for Microsoft's ADO.NET Entity Framework, which has been shipping since 2007. The supported language includes several features which have been individually addressed in the past but, to the best of our knowledge, they have not been addressed all at once before. Moreover, when embarking on an implementation, we found no guidance in the literature on how to modularize the software or apply published algorithms to a commercially-supported language. This paper reports on our experience in addressing these real-world challenges.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures; H.2.4 [**Database Management**]: Query Processing

General Terms

Algorithms, Design

Keywords

Query, query containment, Entity Data Model, canonical instance

1. INTRODUCTION

Query containment is a well-known problem, with many proposed algorithms. However, when embarking on an implementation, we found no guidance in the literature on how to modularize the software or apply published algorithms to a commercially-supported language. This paper reports on our experience in addressing these real-world challenges.

Our application of interest is compilation of object-to-relational mappings [10] for Microsoft's ADO.NET Entity Framework (EF) [1], which has been shipping since 2007. It uses containment tests to check that mappings round-trip, and hence do not lose information. Right now, the product uses a custom containment algorithm that is specific to the currently-supported mappings. We needed a more general algorithm because some extensions to EF require query containment tests that are not supported by the custom containment algorithm. Some extensions under consideration are incremental mapping compilation (currently only full compilation is supported), optimizations of mapping compilation, and support for richer mapping expressions.

The implementation of our query containment checker is based on the Constructive Query Containment (CQC) method [8]. The original definition of this method is quite theoretical, and it is not clear how to implement it in a modular and efficient way. Moreover, the original CQC does not handle all the features of Entity SQL, e.g., type inheritance. In this paper, we have (1) decomposed CQC into implementable steps and (2) adapted it to support Entity SQL queries and schemas expressed in Microsoft's extended ER model, EDM. The result is a modular implementation, which is easily extensible if new features need to be added to the query language. The language supported by the checker includes several features, such as type inheritance [3, 2, 9], key and foreign key constraints [13], null values [5], etc., that have been individually addressed by previous works, but have not, to the best of our knowledge, been addressed all at once before. Checking query containment in such a language is non-trivial. It requires dealing with multiple language features that may interact with each other and affect the result of the containment test. In addition to the aforementioned, other specific features that are particularly interesting and that are also supported by our checker are outer joins, arbitrary boolean conditions, and case statements.

Constructive query containment is aimed at building a counterexample that shows that containment does not hold for the given pair of queries. That is, the containment test of Q_1 in Q_2—denoted $Q_1 \sqsubseteq Q_2$—will fail, if there is a consistent instance in which the answer to Q_1 has a tuple that is not in the answer to Q_2. The idea is that the checker has to explore the space of possible instances and look for such a counterexample. However, since the solution space is infinite, it reduces the exploration to a finite set of canonical instances, each representing a fraction of the solution space.

The construction of the canonical instances is driven by the queries. Essentially, the checker has to consider the different ways in which Q_1 can produce a tuple that Q_2 does not return. In doing so, it has to infer tuples that must be of some type within the corresponding hierarchy, and must have a value for each of their attributes. Instead of trying all possible types and values, it analyzes the queries and schema to determine finite sets of relevant types and values that cover all possible cases; i.e., if no counterexample can be built using these, then no counterexample exists.

Another approach to containment testing is to encode queries as boolean formulas and use a SAT solver to check for satisfiability. However, for the class of queries and constraints in our language,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright © ACM 978-1-4503-2037-5/13/06...$15.00.

[1] The work was done while working at Microsoft.

[2] Partially supported by Ministerio de Ciencia y Tecnología under TIN 2011-24747.

the best encoding we know of produces an exponential blowup in the size of the boolean formulas. Therefore, such an exponential cost in the reduction would defeat the purpose of using a SAT solver, since the bottleneck would be the reduction itself.

The paper is structured as follows. Section 2 details the language supported by the checker. Section 3 explains its modular architecture. Section 4 reports on some experimental results. Section 5 discusses related work, and Section 6 concludes the paper.

2. SUPPORTED LANGUAGE

A schema in the Entity Data Model (EDM) is defined as a collection of entity and association sets. Each entity set stores a hierarchy of entity types, while each association set stores an association type. We will refer to the concepts of entity type and association type as entity and association, respectively, and we will refer to the instances of entity and association types as tuples of the corresponding entity and association. An example EDM schema is shown in Figure 1. It has two entity sets: Pset, which stores the hierarchy rooted at entity P; and Dset, which only stores entity D. The schema also has one association set Aset that stores association A.

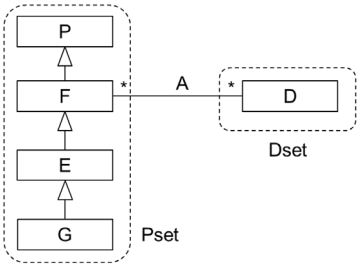

Figure 1: Example EDM schema

Data is stored in the entities in the form of primitive-typed attributes (a.k.a. properties); our algorithm deals with strings, numbers (integers and reals), dates and booleans. Each entity must have a key defined by one or more attributes. A foreign key can be modeled as a special kind of association, one with a referential constraint attached to it.

The supported class of Entity SQL queries is that of unions of select-project-join queries, where a join can be an inner, left outer or full outer join. Case expressions are also supported (i.e., conditional expressions in the form of case when ... then ... else ...). Boolean conditions in the queries may contain conjunctions, disjunctions and negations of atoms. In particular, the supported atomic conditions are: equality of an attribute with a constant, is null, is of (type), and is of (only type)[3]. A pair of Entity SQL queries is shown in Figure 2.

```
Q1: select p.Id                          ⊑   Q2: select p.Id
    from Aset as a inner join Pset as p          from Pset as p
    on key(a.F).Id = p.Id                        where p is of (only F)
    where p.attr = 5
```

Figure 2: Example of a query containment test

Supported queries also conform to the following restrictions:

- The same entity or association set does not appear more than once in the from clause.
- Joins are between a key and a foreign key, or between two keys.
- Any condition on a key or foreign key member is either is null or its negation.
- Key and foreign key members are of a primitive, non-boolean type.
- A query returns a tuple projected from the underlying entity/association sets.

This class of queries is general enough to cover the requirements of our application (incremental EF mapping compilation), with acceptable performance of the query containment checks.

[3] Condition is of (type) is true when the tested entity variable is of either the given type or a subtype. Condition is of (only type) is true when the tested entity variable is exactly of the given type.

3. ARCHITECTURE OF THE QUERY CONTAINMENT CHECKER

As shown in Figure 3, the input to the checker is a pair of queries to be tested, and the output is a boolean that states whether containment holds or not.

In order to simplify the implementation of the query-guided construction of the canonical instances, we reduce the problem from containment of Q_1 in Q_2 to satisfiability of query $Q = Q_1 – Q_2$. Using the constructive approach, we explore the space of canonical instances searching for an example that shows Q is satisfiable. The original containment test is true if and only if Q is not satisfiable.

The three main components of the checker's architecture are:

1. Computation of relevant types and values.
2. Construction of the canonical instances using the relevant types and values.
3. Evaluation of Q on the canonical instances.

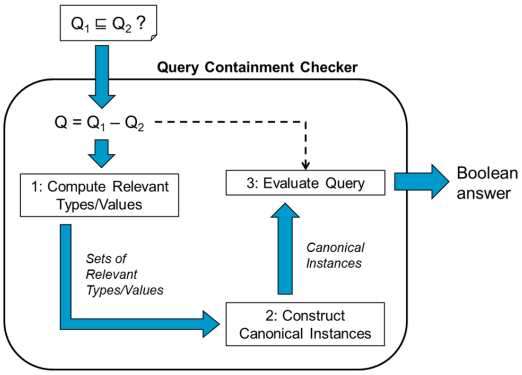

Figure 3: Architecture of the query containment checker

The checker's answer will be `false` if Q has a non-empty answer on some canonical instance, and `true` otherwise.

We describe the components in the next subsections.

3.1 Computation of Relevant Types & Values

To compute the relevant types and values, we need to analyze the query Q to see which entities and associations it refers to, and which types and values are compared with which entity sets and attributes, respectively. Our modular architecture assigns this task to a component that constructs a tuple with the signature of Q, fills it with variables, and then pushes it down the query tree of Q, annotating the tuples and variables with the types and values they are compared to. As an example, pushing the tuple (id) through the projection at the root of Q_2's tree, produces the tuple (id, attr, ...), where attr,... are variables bound to the other attributes of P. Then, pushing this latter tuple down the selection annotates it with type F, i.e., the output is (id, attr, ...):{F}. If the selection had also compared an attribute with a constant, e.g., `p.attr = 3`, the corresponding variable would have been annotated, i.e., the output would be (id, attr:{3}, ...):{F}. At the end of the process, the

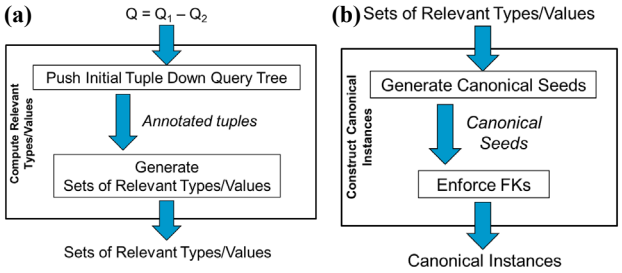

Figure 4: (a) Computing relevant types and values. (b) Constructing canonical instances

component has produced a set of annotated tuples that is passed to a second component that extracts from them all the necessary information and computes the relevant types and values. The workflow is shown in Figure 4(a).

Let us focus now on the second component. A set of relevant types is computed for each entity set in the schema. The idea is to partition each entity set into types that have the same behavior w.r.t. the satisfiability check. The partitioning depends on both query Q and the schema. Query Q induces a partition on each type hierarchy, which distinguishes the types that are mentioned in the query's definition from the types that are not mentioned by the query. Consider for example the schema in Figure 1 and the containment test in Figure 2. Only the entity type F is mentioned in the queries Q_1 and Q_2. Therefore, Q induces the partition {{F}, {P,E,G}} on the type hierarchy stored in Pset. Similarly, each association in the schema induces a partition on the type hierarchies at the association's ends, which distinguishes the type(s) of that end from the rest. In the example, association A induces the partition {{F,E,G}, {P}} in the hierarchy of P, where {F,E,G} are the possible end's types. An agreement has to be reached between all the partitions of the same type hierarchy. We refer to the *agreement* of partitions $P_1,...,P_n$ as the partition P such that two types belong to the same set in P if and only if the two types belong to the same set in all the partitions $P_1,...,P_n$. In the example, there are two partitions for the hierarchy of P: (i) {{F}, {P,E,G}} and (ii) {{F,E,G}, {P}}. The agreement of these two partitions is {{F}, {P}, {E,G}}. After obtaining an agreement, we must choose one arbitrary type from each set in the partition as a relevant type for the entity set in which the hierarchy is stored. Continuing with the example, the set of relevant types of Pset includes F and P and one type from {E, G}, e.g., E. It is obvious that the set of relevant types of Dset includes just D.

A set of relevant values is computed for each attribute of the entities and associations that are accessed by Q. The general rule is that the set of relevant values of an attribute includes the constants which the attribute is compared to in Q plus an additional fresh value. The intuition is that if we have attribute X which is compared to 1 (i.e., X = 1), then, from the point of view of query satisfiability, to cover all cases it suffices to consider the value 1 and some new value, e.g., 2. In our example, query Q_1 compares attribute P.attr with value 5, so the relevant values for P.attr will be {5, $fresh_1$}, where $fresh_1$ denotes an arbitrary new value.

Computing relevant values gets trickier when Q has joins. In that case, we want to cover the case in which the attributes being compared in the join condition have the same value. To do this, we need to extend the general rule and, for each pair of attributes being compared in the join condition, add a single additional fresh value to the set of relevant values of both attributes. Continuing with the example, attributes P.Id and A.F (denoting the F end of association A, which references the key of entity F) must share the same value. Applying the general rule, each of the two attributes has only one relevant value in the form of $fresh_i$ (since they are not compared to any constant in Q). With the new extended rule for join attributes, we create an additional fresh value and add it to the set of relevant values of both attributes. Thus, the relevant values of P.Id will be {$fresh_2$, $fresh_4$} and the ones of A.F will be {$fresh_3$, $fresh_4$} (note that $fresh_4$ is shared by both sets).

3.2 Construction of Canonical Instances

Once we have computed the relevant types and values, we can proceed with the construction of the canonical instances. The checker's module responsible for this task works in two stages, depicted in the workflow in Figure 4(b).

In the first stage, the checker tries the different ways of building a counterexample for the containment test without taking into account the integrity constraints of the schema. The idea is that to satisfy a query Q in which all entity/association sets are distinct, we need at most one tuple in each entity/association set (recall that we consider only queries in which all entity/association sets are distinct). Note that, since our queries belong to the class of unions of select-project-join queries, if tuple t is an answer to Q, then t is a projection of a single tuple that results from the join of some underlying entity or association sets. Therefore, to build an instance in which Q is non-empty, we need to insert at most one tuple in each entity or association set. The type of each tuple is chosen from the corresponding set of relevant types, and the value of each of the tuple's attributes is chosen from the corresponding set of relevant values. Choosing different types and values results in different instances being constructed. We refer to these instances as *canonical seeds*.

In our example, we need to insert one tuple in Pset and another in Aset. Recall that: relevant types of Pset are {F, P, E}, relevant values of P.Id are {$fresh_2$, $fresh_4$}, relevant values of P.attr are {5, $fresh_1$}, relevant values of A.F are {$fresh_3$, $fresh_4$}, and relevant values of A.D are {$fresh_5$}. Then, there is one canonical seed for each combination of these relevant types and values: {F($fresh_2$, 5), A($fresh_3$, $fresh_5$)}, ..., {E($fresh_4$, $fresh_1$), A($fresh_4$, $fresh_5$)}.

In the second stage, the checker evaluates the integrity constraints on each canonical seed. Actually, since canonical seeds cannot violate the key constraints, what this second stage does is to enforce the foreign keys. That is, if a foreign key is violated, the checker tries to repair it by adding a new tuple of the referenced type. If values need to be invented for some of the attributes of the new tuple, distinct fresh values are used. If the new tuple causes the violation of some key constraint, this means that the canonical seed does not represent any consistent instance of the schema and, thus, it is discarded. Otherwise, the checker continues to enforce the foreign keys until the instance becomes consistent. The result of successfully enforcing the foreign keys on a particular canonical seed is a *canonical instance*.

Continuing with the example, consider the canonical seed {E($fresh_4$, 5), A($fresh_4$, $fresh_5$)}. It violates the implicit foreign key from the D end of association A to entity D. Therefore, the checker inserts a new D tuple: D($fresh_5$) (if D had additional attributes they would be filled with distinct fresh values, e.g., D($fresh_5$, $fresh_6$, $fresh_7$, ...)). After this insertion, the instance becomes consistent; therefore, {E($fresh_4$, 5), A($fresh_4$, $fresh_5$), D($fresh_5$)} is a canonical instance.

3.3 Evaluate Query on Canonical Instances

Each canonical instance is a potential counterexample for the containment test. To determine whether a canonical instance *ci* is actually a counterexample or not, the checker evaluates the query Q on it. If Q returns the empty answer, then *ci* is not a counterexample. Otherwise, *ci* shows that Q_1 produces a tuple that Q_2 does not, which is precisely the tuple being returned by Q

(recall $Q = Q_1 – Q_2$). This shows that containment does not hold in this case. Thus containment holds only if the empty answer is obtained as a result of evaluating Q on all canonical instances.

In our example, canonical instance {E($fresh_4$, 5), A($fresh_4$, $fresh_5$), D($fresh_5$)} makes Q return tuple Q($fresh_4$) and shows that containment does not hold.

4. EXPERIMENTAL EVALUATION

To study the performance of our query containment algorithm, we used a scenario that we designed to test those parameters that affect most significantly the algorithm's running time. In this scenario, the schema is a grid of entities, where the entities in each row are connected by foreign keys, and the entities in each column form a hierarchy. In particular, each entity has a foreign key to the next entity in the row, and is a subtype of the entity above it in the column. Figure 5 shows a generic instantiation of this schema. Parameters N and M control the number and depth of type hierarchies, respectively.

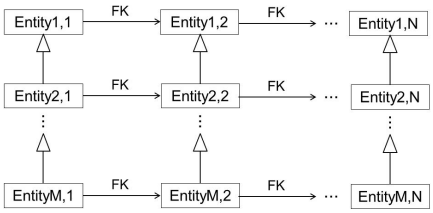

Figure 5: Schema used on the experiments

The queries to be checked for containment are generated over this schema as unions of join queries: $Q = Q_1 \cup ... \cup Q_M$, where $Q_i = Entity_{i,1} \bowtie ... \bowtie Entity_{i,N}$, $1 \le i \le M$. Parameters N and M control the number of joins and unions, respectively. To generate a pair of queries for the containment test, we start with a first query Q, with the above structure, and then generate a second query Q′ by applying a series of transformations on Q. When the transformations are containment-preserving, we know that Q will be contained in Q′. If the transformations do not preserve containment, then we know that Q will not be contained in Q′.

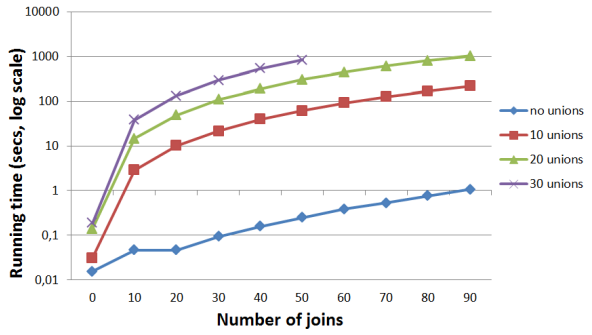

Figure 6: Running times when varying the number of joins and unions in the queries and containment holds.

Figure 6 shows the running times of the checker for an increasing number of joins and unions in the queries. The reported experiments focus on the case in which containment holds, with a variety of optimizations enabled[4]. Our implementation of the checker was done in C#. All the experiments were run on an Intel Core 2 CPU 2.40 GHz, with 4 GB of RAM, and Windows 7 Enterprise SP1 x64 (build 7601). The .NET Framework version used was 4.0.30319 SP1Rel, and the Entity Framework version was "June 2011 CTP".

[4] Details of the optimizations will be presented in the talk.

5. RELATED WORK

Containment has been studied for a wide variety of query languages. The most well-known is the language of conjunctive queries (CQ) [4], which has multiple extensions: unions of CQ [11], CQ with arithmetic comparisons [7], CQ with negation [12], CQ under integrity constraints, etc. For the problem of checking containment under integrity constraints, many different kinds of constraints have been considered, including functional and inclusion dependencies [6], and implication and referential constraints [13], among others. Containment of CQ under class inheritance constraints has also been considered [2, 3, 9]. However, published containment algorithms do not efficiently handle all the details of many practical languages, nor do they explain how to decompose the algorithm into software modules.

6. CONCLUSION

We have proposed a software architecture we have developed for a query containment checker used in the compilation of object-to-relational mappings for Microsoft's ADO.NET Entity Framework. The implementation of our checker is based on the Constructive Query Containment (CQC) method, which we have decomposed into implementable steps and adapted to support Entity SQL queries and EDM schemas. The result is a modular implementation, which is easily extensible if new features need to be added to the query language.

7. REFERENCES

[1] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. SIGMOD 2007, pp. 877–888.

[2] A. Calì. Containment of Conjunctive Queries over Conceptual Schemata. DASFAA 2006, pp. 628-643.

[3] E. P. F. Chan. Containment and Minimization of Positive Conjunctive Queries in OODB's. PODS 1992, pp. 202-211.

[4] A.K. Chandra, and P.M. Merlin. Optimal implementation of conjunctive queries in relational data bases. 9th ACM Symposium on Theory of Computing, 1977, pp. 77–90.

[5] C. Farré, W. Nutt, E. Teniente, and T. Urpí. Containment of Conjunctive Queries over Databases with Null Values. ICDT 2007, pp. 389-403.

[6] D. S. Johnson, and A. C. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. J. Comput. Syst. Sci. 28(1) (1984), pp. 167-189.

[7] A. C. Klug. On conjunctive queries containing inequalities. J. ACM 35(1) (1988), pp. 146-160.

[8] C. Farré, E. Teniente, and T. Urpí. Checking query containment with the CQC method. Data Knowl. Eng. 53(2) (2005), pp. 163-223.

[9] M. Meier, M. Schmidt, F. Wei, G. Lausen. Semantic query optimization in the presence of types. PODS 2010: 111-122.

[10] S. Melnik, A. Adya, and P. A. Bernstein. Compiling mappings to bridge applications and databases. ACM Trans. Database Syst. 33(4) (2008).

[11] Y. Sagiv, and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. J. ACM 27 (4) (1980), pp. 633–655.

[12] F. Wei, and G. Lausen. Containment of Conjunctive Queries with Safe Negation. ICDT 2003, pp. 343-357.

[13] X. Zhang, and Z. M. Özsoyoglu. Implication and Referential Constraints: A New Formal Reasoning. IEEE Trans. Knowl. Data Eng. 9(6) (1997), pp. 894-910.