Query Containment with Negated IDB Predicates

Carles Farré, Ernest Teniente, and Toni Urpí

Universitat Politècnica de Catalunya
08034 Barcelona, Catalonia
{farre,teniente,urpi}@lsi.upc.es

Abstract. We present a method that checks Query Containment for queries with negated IDB predicates. Existing methods either deal only with restricted cases of negation or do not check actually containment but uniform containment, which is a sufficient but not necessary condition for containment. Additionally, our queries may also contain equality, inequality and order comparisons. The generality of our approach allows our method to deal straightforwardly with query containment under constraints. Our method is sound and complete both for success and for failure and we characterize the databases where these properties hold. We also state the class of queries that can be decided by our method.

1 Introduction

Query Containment (QC) is the problem concerned with checking whether the answers that a query obtains are a subset of the answers obtained by another query for every database. QC was first studied for the class of conjunctive queries [CM77]. QC of conjunctive queries with order comparisons was studied in [Klu88, Ull97]. Conjunctive QC with safe negated EDB atoms was investigated in [Ull97, WL03]. EDB stands for *extensional database*, that is, the database's stored relations whereas IDB means *intensional database*, that is, the relations constructed by deductive rules.

The methods that deal with negated IDB subgoals can be classified into two different approaches. The first one is to check QC for query classes where negation is used in a restrictive way [LS95]. The second approach is not to check "true" QC but another related property called *Uniform* QC [LS93], which is a sufficient but not necessary condition for QC [Sag88].

When considering integrity constraints, the containment relationship between two queries does not need to hold for any state of the database but only for those that satisfy the integrity constraints. This idea is captured by the notion of Query Containment under Constraints (QCuC). QCuC for datalog queries, without negation, and integrity constraints expressing tuple-generating dependencies was addressed in [Sag88] by taking the uniform containment approach. QCuC was also handled in the context of hybrid systems combining conjunctive queries and constraints expressed in a Description Logic language [LR96, CDL98].

In [FTU99] we sketched a method, named Constructive Query Containment method (CQC for short), to check "true" QC and QCuC in the presence of negation

L. Kalinichenko et al. (Eds.): ADBIS 2003, LNCS 2798, pp. 416–430, 2003.

on IDB subgoals. Intuitively, the aim of our CQC method was to construct a *counterexample* that proves that there is no QC (or QCuC). This method used different *Variable Instantiation Patterns (VIPs)*, according to the syntactic properties of the queries and the databases considered in each test. Such a customization only affects the way that the facts to be part of the counterexample are instantiated. The aim was to prune the search of counterexamples by generating only the relevant facts.

We extend here our previous work by:

- providing not just an intuitive idea but also the full formalization of the CQC method.
- proving two additional theorems that hold when there are no recursively defined IDB relations: *failure soundness*, which guarantees that containment holds if the method terminates without building any counterexample; and *failure completeness,* which ensures that if containment holds between two queries then our method fails finitely (and terminates).
- ensuring termination when checking containment for conjunctive queries with safe EDB negation and built-in literals.
- pointing out that the CQC method is not less efficient than other methods that deal with conjunctive queries with or without safe EDB negation. We propose an additional VIP, the simple VIP, to perform such a comparison.
- decomposing the General VIP in two: the discrete order VIP and the dense order VIP that allow us to deal with built-in literals assuming both discrete and dense order domains.

It follows from these new results that the method we describe here improves previously proposed algorithms since it provides an efficient decision procedure for known decidable cases and can also be applied for more general forms of queries that were not handled by previous algorithms. In these more general cases, our method is semidecidable because it cannot be guaranteed termination under the presence of infinite counterexamples. Nevertheless, if there is a finite counterexample our method finds it and terminates and if containment holds our method fails finitely and terminates, provided that there are no recursively defined IDB relations in both cases.

Section 2 sets the base concepts used through the paper. In Section 3, we introduce our method and Section 4 formalizes it. In Section 5, we present the main correctness results of our method. For a more detailed formalization and detailed proofs, we refer to [FTU02]. In Section 6, we discuss the decidability issues regarding our method. In Section 7, we compare our method with related work. The paper ends with the conclusions, Section 8, and references.

2 Base Concepts

A *deductive rule* has the form:

$$p(\bar{X}) \leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t$$

where p and r_1, …, r_m are *predicate* (also called *relation*) names. The atom $p(\bar{X})$ is called the *head* of the rule, and $r_1(\bar{X}_1)$, …, $r_n(\bar{X}_n)$, $\neg r_{n+1}(\bar{Y}_1)$, …, $\neg r_m(\bar{Y}_s)$ are positive and negative *ordinary literals* in the body of the rule. The tuples $\bar{X}$, $\bar{X}_1$, …, $\bar{X}_n$, $\bar{Y}_1$, …, $\bar{Y}_s$

contain *terms*, which are either variables or constants. Each C_i is a *built-in literal* in the form of $A_1\ \theta\ A_2$, where A_1 and A_2 are terms. Operator θ is $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$. We require that every rule be *safe*, that is, every variable occurring in $\bar{X}$, $\bar{Y}_1$, …, $\bar{Y}_s$, C_1 , … or C_t must also appear in some $\bar{X}_i$.

The predicate names in a deductive rule range over the *extensional database* (EDB) predicates, which are the relations stored in the database, and the *intensional database* (IDB) predicates (like p above), which are the relations defined by the deductive rules. EDB predicates must not appear in the head of a deductive rule.

A set of deductive rules P is *hierarchical* if there is a partition $P = P_1 \cup \ldots \cup P_n$ such that for any ordinary atom $r(\bar{X})$ occurring positively or negatively (as $\neg r(\bar{X})$) in the body of a clause in P_i, the definition of r is contained within P_j with $j < i$. Note that a hierarchical set of deductive rules contains no recursive IDB relations.

A *condition* has the denial form of:

$$\leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t$$

where $r_1(\bar{X}_1)$, …, $r_n(\bar{X}_n)$, $\neg r_{n+1}(\bar{Y}_1)$, …, $\neg r_m(\bar{Y}_s)$ are (positive and negative) ordinary literals; and C_1, …, C_t are built-in literals. We require also that every variable occurring in $\bar{Y}_1$, …, $\bar{Y}_s$, C_1 , … or C_t must also appear in some $\bar{X}_i$. Roughly, a condition in denial form expresses a prohibition: a conjunction of facts (literals in the body) that must no hold on the database all at once. Therefore, a condition is *violated* (*not satisfied*), whenever $\exists \bar{Z}\,(r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t)$ is true on the database, where $\bar{Z}$ contains the variables occurring in $\bar{X}_1$, …, $\bar{X}_s$, $\bar{Y}_1$, …, $\bar{Y}_s$, C_1 , … and C_t.

A *query* Q is a finite set of deductive rules that defines a dedicated n-ary *query predicate* q. Without loss of generality, other predicates than q appearing in Q are EDB or IDB predicates.

A query Q_1 is *contained* in a query Q_2, denoted by $Q_1 \subseteq Q_2$, if the set of answers of $Q_1(D)$ is a subset of those of $Q_2(D)$ for any database D. Moreover, Q_1 *is contained in* Q_2 *wrt IC*, denoted by $Q_1 \subseteq_{IC} Q_2$, if the set of answers of $Q_1(D)$ is a subset of those of $Q_2(D)$ for any database D satisfying a finite set IC of conditions (*integrity constraints*).

3 The Constructive Query Containment (CQC) Method

The containment relationship between two queries must hold for the whole set of possible databases in the general case. A suitable way of checking QC is to check the lack of containment, that is, to find just one database where the containment relationship that we want to check does not hold: Q_1 *is not contained in* Q_2, written $Q_1 \not\subseteq Q_2$, if there is at least one database D such that $Q_1(D) \not\subset Q_2(D)$.

Given Q_1 and Q_2 two queries, the CQC method is addressed to construct the extensional part of a database (EDB) where the containment relationship does not hold. It requires two main inputs: the *goal* to attain and the s*et of conditions to enforce*. Initially, the goal is defined $G_0 = \leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$, meaning that we want to construct a database where $(X_1, ..., X_n)$ could be instantiated in such a

way that $q_1(X_1, ..., X_n)$ is true and $q_2(X_1, ..., X_n)$ is false. The set of conditions to enforce is $F_0 = \emptyset$, since there is no initial integrity constraint to take care about.

When considering a set *IC* of integrity constraints, we say that Q_1 *is not contained in* Q_2 *wrt IC*, written $Q_1 \subset_{IC} Q_2$, if there is at least one database *D* satisfying *IC*, such that $Q_1(D) \not\subset Q_2(D)$. In this case, the EDB that the CQC method has to construct to refute the containment relationship must also satisfy the conditions in *IC*. This is guaranteed by making the initial set of conditions to enforce $F_0 = IC$ together with the goal $G_0 = \leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$.

3.1 Example: $Q_1 \subset Q_2$

The following example is adapted from the one in [FTU99] by introducing double negation on IDB predicates. It allows illustrating the main ideas of our method and to show its behavior under these complex cases. Let Q_1 and Q_2 be two queries:

$Q_1 = \{ sub_1(X) \leftarrow emp(X) \wedge \neg chief(X) \}$

$Q_2 = \{ sub_2(X) \leftarrow emp(X) \wedge \neg boss(X) \}$

where *emp* is an EDB predicate and *chief* and *boss* are IDB predicates defined by a set *DR* of deductive rules:

$DR = \{ boss(X) \leftarrow worksFor(Z, X)$

$chief(X) \leftarrow worksFor(Y, X) \wedge \neg boss(Y) \}$

where *worksFor* is another EDB predicate.

Intuitively, we can see that Q_1 is less restrictive than Q_2 because Q_2 does not retrieve those employees having anyone working for them, while Q_1 allows retrieving employees having some boss working for them. Hence, we can find a database containing EDB relations such as *emp*(joan), *worksFor*(mary, joan) and *worksFor*(ann, mary), where sub_1(joan) is true but sub_2(joan) is false (*chief*(joan) is false because *boss*(mary) is true whereas *boss*(joan) is true). Therefore, Q_1 is not contained in Q_2. Note that an even smaller EDB containing just *emp*(joan) and *worksFor*(joan, joan) would have lead us to the same conclusion.

A CQC-derivation that constructs an EDB that proves $Q_1 \subset Q_2$ are shown in Fig. 1. Each row on the figure corresponds to a CQC-node that contains the following information (columns):

1. The goal to attain: the literals that must be made true by the EDB under construction. When the goal is [] it means that no literal needs to be satisfied. Here, the initial CQC-node contains the goal $G_0 = \leftarrow sub_1(X) \wedge \neg sub_2(X)$. That is, we want the CQC method to construct a database where exists at least a constant k such that both sub_1(k) and $\neg sub_2$(k) are true
2. The conditions to be enforced: the set of conditions that the constructed EDB is required to satisfy. Recall that a condition is violated whenever all of its literals are evaluated as true. Here, the initial CQC-node contains the set of conditions to enforce $F_0 = \emptyset$.
3. The EDB under construction. Initial CQC-Nodes always have empty EDBs.
4. The conditions to be maintained: a set containing those conditions that are known to be satisfied in the current CQC-Node and that must remain satisfied

until the end of the CQC-derivation. Initial CQC-Nodes have always this set empty.

5. The account of constants introduced in the current and/or the ancestor CQC-nodes to instantiate the EDB facts in the EDB under construction. Initially, such a set contains always the constants appearing already in $DR \cup Q_1 \cup Q_2 \cup G_0 \cup F_0$.

	Goal to attain	*Conditions to enforce*	*EDB*	*Conditions to mantain*	*Used constants*
	$\leftarrow \underline{sub_1(X)} \wedge \neg sub_2(X)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
1:A1	$\leftarrow \underline{emp(X)} \wedge \neg chief(X) \wedge \neg sub_2(X)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
2:A2	$\leftarrow \underline{\neg chief(0)} \wedge \neg sub_2(0)$	$\varnothing$	$\{emp(0)\}$	$\varnothing$	$\{ 0 \}$
3:A3	$\leftarrow \underline{\neg sub_2(0)}$	$\{\leftarrow chief(0)\}$	$\{emp(0)\}$	$\varnothing$	$\{ 0 \}$
4:A3	[]	$\{\leftarrow \underline{chief(0)}, \leftarrow sub_2(0)\}$	$\{emp(0)\}$	$\varnothing$	$\{ 0 \}$
5:B1	[]	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y), \leftarrow sub_2(0)\}$	$\{emp(0)\}$	$\varnothing$	$\{ 0 \}$
6:B2	[]	$\{\leftarrow \underline{sub_2(0)}\}$	$\{emp(0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
7:B1	[]	$\{\leftarrow \underline{emp(0)} \wedge \neg boss(0)\}$	$\{emp(0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
8:B2	[]	$\{\leftarrow \underline{\neg boss(0)}\}$	$\{emp(0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
9:B3	$\leftarrow \underline{boss(0)}$	$\varnothing$	$\{emp(0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
10:A1	$\leftarrow \underline{worksFor(Z,0)}$	$\varnothing$	$\{emp(0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
11:A2	[]	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{emp(0), worksFor(0,0)\}$	$\varnothing$	$\{ 0 \}$
12:B2	[]	$\{\leftarrow \underline{\neg boss(0)}\}$	$\{emp(0), worksFor(0,0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
13:B3	$\leftarrow \underline{boss(0)}$	$\varnothing$	$\{emp(0), worksFor(0,0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
14:A1	$\leftarrow \underline{worksFor(Z,0)}$	$\varnothing$	$\{emp(0), worksFor(0,0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$
15:A2	[]	$\varnothing$	$\{emp(0), worksFor(0,0)\}$	$\{\leftarrow \underline{worksFor(Y,0)} \wedge \neg boss(Y)\}$	$\{ 0 \}$

Fig. 1.

The transition between two consecutive CQC-nodes, i.e. between an ancestor node and its successor, is a CQC-step that is performed by applying a CQC-expansion rule to a selected literal of the ancestor CQC-node. The selection of literals in the CQC-derivation of Fig. 1 is nearly arbitrary: the only necessary criterion is to avoid picking a non-ground negative-ordinary or built-in literal. In Fig. 1, the CQC-steps are labeled with the name of the CQC-expansion rule that is applied and the selected literal in

each step is underlined. We refer to Section 4.2 for a proper formalization of the CQC-expansion rules.

The first step unfolds the selected literal, the IDB atom $sub_1(X)$ from the goal part, by substituting it with the body of its defining rule. At the second step, the selected literal from the goal part is $emp(X)$, which is a positive EDB literal. To get a successful derivation, i.e. to obtain an EDB satisfying the initial goal, $emp(X)$ must be true on the constructed EDB. Hence, the method instantiates X with a constant and includes the new ground EDB fact in the EDB under construction. The procedure assigns an arbitrary constant to X, e.g. 0. So $emp(0)$ is the first fact included in the EDB under construction.

$\neg chief(0)$ is the selected literal in step 3. To get success for the derivation, $chief(0)$ must not be true on the EDB. This is guaranteed by adding $\leftarrow chief(0)$ as a new condition to be enforced. Step 4 is similar to step 3, yielding $\leftarrow sub_2(0)$ to be considered as another condition to be enforced. After performing this later step, we get a CQC-node with a goal like []. However, the work is not done yet, since we must ensure that the two conditions $\leftarrow sub_2(0)$ and $\leftarrow chief(0)$ are not violated by the current EDB. In other words, we must make both $chief(0)$ and $sub_2(0)$ false.

Step 5 unfolds the selected literal $chief(0)$ from one of the two conditions, getting $\leftarrow worksFor(Y, 0) \wedge \neg boss(Y)$ as a new condition that replaces $\leftarrow chief(0)$. At least one of the two literals of this condition must be false. In step 6, the selected literal is the positive EDB literal is $worksFor(Y, 0)$. Since it matches with no EDB atom in the EDB under construction, $worksFor(Y, 0)$ is false and, consequently, the whole condition $\leftarrow worksFor(Y, 0) \wedge \neg boss(Y)$ is not violated by the current EDB. For this reason, such a condition is moved from the set of conditions to enforce to the set of conditions to maintain.

Step 7 unfolds the selected IDB atom $sub_2(0)$ from the remaining condition to enforce. The EDB atom $emp(0)$ is the selected literal in step 8. Since $emp(0)$ is also present in the EDB under construction, it cannot be false. So this literal is dropped from the condition because it does not help to enforce the condition. In step 9 the selected literal is the negative literal $\neg boss(0)$. Since it is the only literal of the condition, it must be made false necessarily. So $boss(0)$ becomes a new (sub)goal to achieve and is transferred, thus, to the goal part.

Step 10 unfolds the selected literal $boss(0)$ from the goal part as in step 1. $worksFor(z, 0)$ is the selected literal in step 11. As in step 2, the method should instantiate Z with a constant. In this case, the chosen constant is 0 again, so $worksFor(0, 0)$ is added to the EDB under construction. Moreover, the condition $\leftarrow worksFor(Y, 0) \wedge \neg boss(Y)$ is moved back to the set of conditions to enforce to avoid that the new inclusion of $worksFor(0, 0)$ in the EDB violates it.

In step 12, the selected literal is the positive EDB literal is $worksFor(Y, 0)$ from the remaining condition to enforce. Now, it matches with the current contents of the EDB with $Y = 0$. As in step 8, such a literal is dropped from the condition. However, the whole condition $\leftarrow worksFor(Y, 0) \wedge \neg boss(Y)$ is moved again to the set of conditions to maintain in order to prevent further inclusions of new facts about *worksFor* in the EDB from violating it.

Steps 13 and 14 are identical to steps 9 and 10. In step 15, the constant 0 is selected again to instantiate $worksFor(Z, 0)$. Since $worksFor(0, 0)$ is already included in the

EDB, there is no need to transfer back any condition from the set of conditions to maintain to the set of conditions to enforce.

The CQC-derivation ends successfully since it reaches a CQC-node where the goal to attain is [] and the set of conditions to satisfy is empty. In other words, we can be sure that its EDB, {*emp*(0), *worksFor*(0, 0)}, contains a set of facts that makes the database satisfy the goals and conditions of all preceding CQC-nodes, including, naturally, the first CQC-node. Then we conclude $Q_1 \subset Q_2$.

3.2 Variable Instantiation Patterns

When a CQC-derivation terminates successfully, we obtain a proof, the constructed EDB, which shows that the containment relationship is not true. On the contrary, when a derivation ends unsuccessfully, that is, it terminates but it fails to construct a counterexample, we cannot conclude that containment holds based on a single result. Then the question is how many derivations must be considered before achieving a reliable conclusion. Indeed, rather than the account of all possible derivations, the real point is to know how many variable instantiation alternatives must be considered when adding new facts to the EDB under construction.

The aim of the CQC method is to test only the variable instantiations that are *relevant* without losing completeness. The "strategy" for instantiating the EDB facts to be included in the EDB under construction is connected to, indeed it is inspired by, the concept of *canonical databases* found in [Klu88, Ull97].

Since the canonical databases to be taken into account depend on the concrete subclass of queries that are considered, we distinguish three different *variable instantiation patterns*, VIPs for shorthand. Each of them defines how the CQC method has to instantiate the EDB facts to be added to the EDB under construction. The four VIPs that we define are: *Simple VIP*, *Negation VIP* (as considered in [FTU99]), *Dense Order VIP* and *Discrete Order VIP*.

The CQC method uses the *Simple VIP* when checking containment but not QC under constraints. Moreover, the deductive rules defining query predicates as well as IDB predicates must satisfy the following conditions: they must not have any negative or built-in literal in their rule bodies; they must not have constants in their heads; and they must not have any variable appearing twice or more times in their heads. According to this VIP, each distinct variable is bound to a distinct new constant.

The CQC method uses the *Negation VIP* when checking QCuC or when checking containment under the presence of negated IDB subgoals, negated EDB subgoals and/or (in)equality comparisons ($=, \neq$). In any case, order comparisons ($<, \leq, >, \geq$) are not allowed. EDBs generated and tested with this VIP correspond to the canonical EDBs considered in [Ull97] for the conjunctive query case with negated EDB subgoals. The intuition behind this VIP is clear: Each new variable appearing in a EDB fact to be grounded is instantiated with either some constant previously used or a constant never used before. This is the pattern used in t Fig. 1.

The *Dense Order VIP* and *Discrete Order VIP* are applied when there are order comparisons ($<, \leq, >, \geq$) in the deductive rules, with or without negation. In this case, each distinct variable must be bound to a constant according to either a former or a new location in the total linear order of constants introduced previously [Klu88,

Ull97]. The election between these two VIPS depends on whether the comparisons are interpreted on a dense (real numbers) or a discrete order (integer numbers).

4 Formalization of the CQC Method

Let Q_1 and Q_2 be two queries, *DR* the set of deductive rules defining the database IDB relations and *IC* a finite set of conditions expressing the database integrity constraints. If the CQC method performs a successful CQC-derivation from ($\leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$ $\varnothing$ $\varnothing$ $\varnothing$ K) to ([] $\varnothing$ T C K') then Q_1 c Q_2, where K is the set of constants appearing in $DR \cup Q_1 \cup Q_2$. Moreover, if the CQC method performs a successful CQC-derivation from ($\leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$ IC $\varnothing$ $\varnothing$ K'') to ([] $\varnothing$ T C K''') then Q_1 c_{IC} Q_2, where K'' is the set of constants appearing in $DR \cup Q_1 \cup Q_2 \cup IC$.

CQC-derivations start from a 5-tuple (G_0 F_0 T_0 C_0 K_0) consisting of the goal $G_0 = \leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$, the set of conditions to enforce $F_0 = \varnothing$ or *IC*, the initially-empty EDB $T_0 = \varnothing$, the empty set of conditions to maintain $C_0 = \varnothing$ and the set K_0 of constant values appearing in $DR \cup Q_1 \cup Q_2[\cup IC]$.

A successful CQC-derivation reaches a 5-tuple (G_n F_n T_n C_n K_n) = ([] $\varnothing$ T C K'), where the empty goal G_n = [] means that we have reached the goal G_0 we were looking for. The empty set $F_n = \varnothing$ means that no condition is waiting to be satisfied. $T_n = T$ is an EDB that satisfies G_0 as well as F_0. $C_n = C$ is a set of conditions recorded along the derivation and that T also satisfies. $K_n = K'$ is the set of constant values appearing in $DR \cup Q_1 \cup Q_2[\cup IC] \cup T$.

On the contrary, if every "fair" CQC-derivation starting from ($\leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$ $\varnothing$ [$\cup IC$] $\varnothing$ $\varnothing$ K) is finite but does not reach ([] $\varnothing$ T C K'), it will mean that no EDB satisfies the goal $G_0 = \leftarrow q_1(X_1, ..., X_n) \wedge \neg q_2(X_1, ..., X_n)$ together with the set of conditions $F_0 = \varnothing$ [$\cup IC$], concluding that Q_1 S Q_2 (Q_1 S_{IC} Q_2). Section 5 below provides the complete results and proofs regarding the soundness and completeness of the CQC method.

4.1 CQC-Nodes, CQC-Trees and CQC-Derivations

Let Q_1 and Q_2 be two queries, *DR* be the set of deductive rules defining the database IDB relations and *IC* be a finite set of conditions expressing the database integrity constraints. A *CQC-node* is a 5-tuple of the form (G_i F_i T_i C_i K_i), where G_i is a goal to attain; F_i is a set of conditions to enforce; T_i is a set of ground EDB atoms, an EDB under construction; C_i is a set of conditions that are currently satisfied in T_i and must be maintained; and K_i is the set of constants appearing in $R = DR \cup Q_1 \cup Q_2[\cup IC]$ and T_i.

A *CQC-tree* is inductively defined as follows:

1. The tree consisting of the single CQC-node (G_0 F_0 $\varnothing$ $\varnothing$ K) is a CQC-tree.
2. Let E be a CQC-tree, and (G_n F_n T_n C_n K_n) a leaf CQC-node of E such that $G_n \neq$ [] or $F_n \neq \varnothing$. Then the tree obtained from E by appending one or more descendant CQC-nodes according to a CQC-expansion rule applicable to (G_n F_n T_n C_n K_n) is again a CQC-tree.

It may happen that the application of a CQC-expansion rule on a leaf CQC-node $(G_n\ F_n\ T_n\ C_n\ K_n)$ does not obtain any new descendant CQC-node to be appended to the CQC-tree because some necessary constraint defined on the CQC-expansion rule is not satisfied. In such a case, we say that $(G_n\ F_n\ T_n\ C_n\ K_n)$ is a *failed* CQC-node.

Each branch in a CQC-tree is a *CQC-derivation* consisting of a (finite or infinite) sequence $(G_0\ F_0\ T_0\ C_0\ K_0)$, $(G_1\ F_1\ T_1\ C_1\ K_1)$, … of CQC-nodes.

A CQC-derivation is *finite* if it consists of a finite sequence of CQC-nodes; otherwise it is *infinite*. A CQC-derivation is *successful* if it is finite and its last (leaf) CQC-node has the form $([]\ \varnothing\ T_n\ C_n\ K_n)$. That is, both the goal to attain and the set of conditions to satisfy are empty. A CQC-derivation is *failed* if it is finite and its last (leaf) CQC-node is failed.

A CQC-tree is *successful* when at least one of its branches is a successful CQC-derivation. A CQC-tree is *finitely failed* when each one of its branches is a failed CQC-derivation.

4.2 The CQC-Expansion Rules

The nine *CQC-expansion rules* are listed in tables 4.1 and 4.2. For the sake of notation, if $G_i = \leftarrow L_1 \wedge \ldots \wedge L_{j-1} \wedge L_j \wedge L_{j+1} \wedge \ldots \wedge L_m$ then $G_i \backslash L_j = \leftarrow L_1 \wedge \ldots \wedge L_{j-1} \wedge L_{j+1} \wedge \ldots \wedge L_m$. If $G_i = \leftarrow L_1 \wedge \ldots \wedge L_m$ then $G_i \wedge p(\bar{X}) = \leftarrow L_1 \wedge \ldots \wedge L_m \wedge p(\bar{X})$.

Table 1. CQC-expansion rules: A#-rules.

A1) $P(G_i) = d(\bar{X})$ is a positive IDB atom:

$$\frac{(G_i\ F_i\ T_i\ C_i\ K_i)}{(G_{i+1,1}\ F_i\ T_i\ C_i\ K_i)\ |\ \ldots\ |\ (G_{i+1,m}\ F_i\ T_i\ C_i\ K_i)}$$

only if $m \geq 1$ and each $G_{i+1,j}$ is the resolvent for G_i and some deductive rule $d(\bar{Y}) \leftarrow M_1 \wedge \ldots \wedge M_q$ in R.

A2) $P(G_i) = b(\bar{X})$ is a positive EDB atom:

$$\frac{(G_i\ F_i\ T_i\ C_i\ K_i)}{((G_i \backslash b(\bar{X}))\sigma_1\ F_{i+1,1}\ T_{i+1,1}\ C_{i+1,1}\ K_{i+1,1})\ |\ \ldots\ |\ ((G_i \backslash b(\bar{X}))\sigma_m\ F_{i+1,m}\ T_{i+1,j}\ C_{i+1,m}\ K_{i+1,m})}$$

such that $F_{i+1,j} = F_i \cup C_i$, $T_{i+1,j} = T_i \cup \{b(\bar{X})\sigma_j\}$ and $C_{i+1,j} = \varnothing$ if $b(\bar{X})\sigma_j \notin T_i$; otherwise $F_{i+1,j} = F_i$, $T_{i+1,j} = T_i$ and $C_{i+1,j} = C_i$. Each σ_j is one out of m possible distinct ground substitutions, obtained via a variable instantiation procedure from $(vars(\bar{X}), \varnothing, K_i)$ to $(\varnothing, \sigma_j, K_{i+1,j})$ according to the appropriate variable instantiation pattern, that assigns a constant from $K_{i+1,j}$ to each variable in $vars(\bar{X})$. See more details in [FTU02].

A3) $P(G_i) = \neg p(\bar{X})$ is a ground negated atom:

$$\frac{(G_i\ F_i\ T_i\ C_i\ K_i)}{(G_i \backslash \neg p(\bar{X})\ F_i \cup \{\leftarrow p(\bar{X})\}\ T_i\ C_i\ K_i)}$$

A4) $P(G_i) = L$ is a ground built-in literal:

$$\frac{(G_i\ F_i\ T_i\ C_i\ K_i)}{(G_i \backslash L\ F_i\ T_i\ C_i\ K_i)}$$

only if L is evaluated true.

Table 2. CQC-expansion rules: B#-rules.

B1) $P(F_{i,j}) = d(\bar{X})$ is a positive IDB atom:

$$\frac{(G_i\ \{F_{i,j}\}\cup F_i\ T_i\ C_i\ K_i)}{(G_i\ S\cup F_i\ T_i\ C_i\ K_i)}$$

where S is the set of all resolvents S_u for clauses in R and $F_{i,j}$ on $d(\bar{X})$. S may be empty.

B2) $P(F_{i,j}) = b(\bar{X})$ is a positive EDB atom:

$$\frac{(G_i\ \{F_{i,j}\}\cup F_i\ T_i\ C_i\ K_i)}{(G_i\ S\cup F_i\ T_i\ C_{i+1}\ K_i)}$$

only if $[] \notin S$.
$C_{i+1} = C_i$ if $\bar{X}$ contains no variables and $b(\bar{X}) \in T_i$; otherwise, $C_{i+1} = C_i \cup \{F_{i,j}\}$
S is the set of all resolvents of clauses in T_i with $F_{i,j}$ on $b(\bar{X})$. S may be empty, meaning that $b(\bar{X})$ cannot be unified with any atom in T_i.

B3) $P(F_{i,j}) = \neg p(\bar{X})$ is a ground negative ordinary literal:

$$\frac{(G_i\ \{F_{i,j}\}\cup F_i\ T_i\ C_i\ K_i)}{(G_i\ \{\leftarrow p(\bar{X})\}\cup\{F_{i,j}\backslash\neg p(\bar{X})\}\cup F_i\ T_i\ C_i\ K_i)\ \textit{only if}\ F_{i,j}\backslash\neg p(\bar{X}) \neq [] \quad \mid \quad (G_i \wedge p(\bar{X})\ F_i\ T_i\ C_i\ K_i)}$$

B4) $P(F_{i,j}) = L$ is a ground built-in literal that is evaluated true:

$$\frac{(G_i\ \{F_{i,j}\}\cup F_i\ T_i\ C_i\ K_i)}{(G_i\ \{F_{i,j}\backslash L\}\cup F_i\ T_i\ C_i\ K_i)}$$

only if $F_i\backslash L \neq []$.

B5) $P(F_{i,j}) = L$ is a ground built-in literal that is evaluated false:

$$\frac{(G_i\ \{F_{i,j}\}\cup F_i\ T_i\ C_i\ K_i)}{(G_i\ F_i\ T_i\ C_i\ K_i)}$$

Once a literal is selected, only one of the CQC-expansion rules can be applied. We distinguish two classes of rules: A-rules and B-rules. A-rules are those where the selected literal belongs to the goal G_i. Instead, B-rules correspond to those where the selected literal belongs to any of the conditions $F_{i,j}$ in F_i. Inside each class of rules, they are differentiated with respect to the type of the selected literal.

In each CQC-expansion rule, the part above the horizontal line presents the CQC-node to which the rule is applied. Below the horizontal line is the description of the resulting descendant CQC-nodes. Vertical bars separate alternatives corresponding to different descendants. Some rules like A1, A5, B2 and B4 include also an "only if" condition that constraints the circumstances under which the expansion is possible. If such a condition is evaluated false, the CQC-node to which the rule is applied becomes a failed CQC-node.

Finally, note that three CQC-expansion rules, namely A1, B1 and B2, use the resolution principle as is defined in [Llo87].

The application of a CQC-expansion rule on a given CQC-node ($G_i\ F_i\ T_i\ C_i\ K_i$) may result in none, one or several alternative (branching) descendant CQC-nodes depending on the selected literal $P(J_i) = L$. Here, J_i is either the goal G_i or any of the conditions $F_{i,j}$ in F_i. L is selected according to a safe computation rule P [Llo87],

which selects negative and built-in literals only when they are fully grounded. To guarantee that such literals are sooner or later selected we require deductive rules and goals to be safe.

5 Correctness Results for the CQC Method

In this Section, we summarize and sketch the new proofs of correctness of the CQC method. We refer the reader to [FTU02] for the detailed proofs. We also state the class of queries that can be actually decided by the CQC method. Before proving these results, we need to make explicit the model-theoretic semantics to with respect those results are established.

Let R be a set of deductive rules. We define the *partial completion* of R, denoted by $pComp(R)$, as the collection of completed definitions [Cla77] of IDB predicates in R together with an equality theory. This later one includes a set of axioms stating explicitly the meaning of the predicate "=" introduced in the completed definitions.

Our partial completion is defined similarly to Clark's completion [Cla77], $Comp(R)$, but without including the axioms of the form $\forall x(\neg b_i(\bar{X}))$ for each predicate b_i which only occurs in the body of the clauses in R. We assume that these predicates are EDB predicates that, obviously, are not defined in R.

If Q_1 and Q_2 are two queries, DR is the set of deductive rules defining the database IDB relations and IC be a finite set of conditions expressing the database integrity constraints, we consider that problem of knowing whether $Q_1 \mathsf{S} Q_2$ *(*$Q_1 \mathsf{S}_{IC} Q_2$*)* is equivalent to the problem of proving that $pComp(R)[\cup \forall IC] \setminus \forall X_1 \ldots X_n \; q_1(X_1,\ldots,X_n) \rightarrow q_2(X_1,\ldots,X_n)$ is true, where $R = DR \cup Q_1 \cup Q_2$. If we define the initial goal $G_0 = \leftarrow q_1(X_1,\ldots,X_n) \wedge \neg q_2(X_1,\ldots,X_n)$ then testing $Q_1 \mathsf{S} Q_2$ *(*$Q_1 \mathsf{S}_{IC} Q_2$*)* is equivalent to proving $pComp(R)[\cup \forall IC] \setminus G_0$. This proof is tackled by the CQC method, which tries to refute $pComp(R)[\cup \forall IC] \setminus G_0$ by constructing an EDB T such that $R(T)$ is a model for $pComp(R)\ [\cup \forall IC] \cup \{\exists X_1 \ldots \exists X_n\ (q_1(X_1,\ldots,X_n) \wedge \neg q_2(X_1,\ldots,X_n))\}$.

In the following theorems, let $G_0 = \leftarrow q_1(X_1, \ldots, X_n) \wedge \neg q_2(X_1, \ldots, X_n)$ be the initial goal, $F_0 = \varnothing\ [\cup IC]$ be the initial set of conditions to enforce and K be the set of constants appearing in $DR \cup Q_1 \cup Q_2 \cup F_0$.

Before proving results related to failure of the CQC method, we review the results related to finite success already stated in [FTU99].

Theorem 5.1 (*Finite Success Soundness*)

If there exists a finite successful CQC-derivation starting from $(G_0\ F_0\ \varnothing\ \varnothing\ K)$ then $Q_1 \mathsf{c} Q_2$ $(Q_1 \mathsf{c}_{IC} Q_2)$ provided that $\{G_0\} \cup F_0 \cup DR \cup Q_1 \cup Q_2$ is safe and hierarchical.

Theorem 5.2 (*Finite Success Completeness*)

If $Q_1 \mathsf{c} Q_2$ *(*or $Q_1 \mathsf{c}_{IC} Q_2$*)* then there exists a successful CQC-derivation from $(G_0\ F_0\ \varnothing\ \varnothing\ K)$ to $([]\ \varnothing\ T\ C\ K')$ provided that $\{G_0\} \cup F_0 \cup DR \cup Q_1 \cup Q_2$ is safe and either hierarchical or strict-stratified [CL89].

These results ensure that, in the absence of recursive IDB predicates, if the method builds a finite counterexample, then containment does not hold (Theorem 5.1); and

that if there exists a finite counterexample, then our method finds it and terminates (Theorem 5.2). We extend these results by assessing the properties regarding failure of our method. In this sense, we prove *failure soundness* (Theorem 5.3), which guarantees that if the method terminates without building any counterexample then containment holds; and *failure completeness* (Theorem 5.5), which states that if containment holds between two queries then our method fails finitely.

Theorem 5.3 (*Failure Soundness*)

If there exists a finitely failed CQC-Tree rooted at (G_0 F_0 $\varnothing$ $\varnothing$ K) then Q_1 S Q_2 (Q_1 S_{IC} Q_2) provided that the deductive rules and conditions in $DR \cup Q_1 \cup Q_2[\cup IC]$ are safe.

The proof of Theorem 5.3 is made by using the principle of contradiction and may be intuitively explained as follows. Le us suppose that we have a finitely failed CQC-tree but Q_1 c Q_2 (Q_1 c_{IC} Q_2). If Q_1 c Q_2 (Q_1 c_{IC} Q_2) it means for us that $pComp(R)$ $[\cup \forall IC] \cup \{\exists X_1 ... \exists X_n (q_1(X_1,\ldots,X_n) \wedge \neg q_2(X_1,\ldots,X_n))\}$ has a model. However, if this is true, we prove that there is at least one CQC-derivation not finitely failed.

Lemma 5.4 is needed for proving Theorem 5.5. Before stating it, we need some new definitions.

A CQC-derivation is *open* when it is not failed. That is, when the derivation is either infinite or finite with its last (leaf) CQC-node having the form of ([] $\varnothing$ T_n C_n K_n). A CQC-derivation θ is *saturated for the CQC-expansion Rules* if for every CQC-node (G_i F_i T_i C_i K_i) in θ the following properties hold:

1. For each literal $L_{i,j} \in G_i$ there exists a node (G_n F_n T_n C_n K_n), $n \geq i$, such that $P(G_n) = L_{i,j}\sigma_{i+1}\ldots\sigma_n$ is the selected literal on that node to apply a CQC A-rule, where $\sigma_{i+1}\ldots\sigma_n$ is the composition of the substitutions used in the intermediate nodes.
2. For each condition $F_{i,j_i} \in F_i$ there exists a node (G_n F_n T_n C_n K_n), $n \geq i$, such that $F_{n,j_n} \in F_n$ is the selected condition on that node to apply a CQC B-rule and $F_{n,j_n} = F_{i,j_i}$.

A CQC-derivation is said to be *fair* when it is either failed or open and saturated for the CQC-expansion Rules. A CQC-tree is *fair* if each one of its CQC-derivations (branches) is fair. Note that a finitely failed CQC-tree is always fair, but the inverse is not necessarily true.

Lemma 5.4

Let R be a set of deductive rules, $G = \leftarrow L_1 \wedge \ldots \wedge L_k$ be a goal, F be a set of conditions and K be the set of constants in $\{G_0\} \cup F_0 \cup R$. If there exists a saturated open CQC-derivation starting from (G_0 $F_0 \varnothing$ $\varnothing$ K) then $pComp(R) \cup \{\exists(L_1 \wedge \ldots \wedge L_k)\} \cup \forall F_0$ has a model provided that $\{G_0\} \cup F_0 \cup R$ is safe and hierarchical.

Theorem 5.5 (*Failure Completeness*)

If Q_1 S Q_2 *(*Q_1 S_{IC} Q_2*)* then every fair CQC-Tree rooted at (G_0 F_0 $\varnothing$ $\varnothing$ K) is finitely failed provided that $\{G_0\} \cup F_0 \cup DR \cup Q_1 \cup Q_2$ is safe and hierarchical.

The proof is made by contradiction and may be intuitively explained as follows. If Q_1 S Q_2 (Q_1 S_{IC} Q_2) then $pComp(R)$ $[\cup \forall IC] \cup \{\exists X_1 \ldots X_n \quad q_1(X_1,\ldots,X_n) \wedge$

$\neg q_2(X_1,\ldots,X_n)\}$ cannot have a model. Assuming that it is true, let us suppose that we have a non-failed CQC-derivation starting from (G_0 F_0 $\varnothing$ $\varnothing$ K). However, lemma 5.4 shows that this derivation would indeed construct a model for $pComp(DR)$ $[\cup\forall IC] \cup \{\exists X_1 ... \exists X_n\ (q_1(X_1,\ldots,X_n) \wedge \neg q_2(X_1,\ldots,X_n))\}$.

6 Decidability Results

The QC problem is undecidable for the general case of queries and databases that the CQC Method covers [AHV95]. A possible source of undecidability is the presence of recursion, which could make the CQC Method build and test an infinite number of EDBs. In this sense, the CQC Method excludes explicitly the presence of any type of recursion as we have seen in the proofs of the failure completeness (Theorem 5.5) and the finite success soundness and completeness (Theorems 5.1 and 5.2).

Another reason for undecidability is the presence of "axioms of infinity" [BM86]. In this case, the initial goal to attain could only be satisfied on an EDB with an infinite number of facts because each new addition of a fact to the EDB under construction triggers a condition to be *repaired* with another insertion on the EDB.

For this reason, the CQC Method is semidecidable for the general case, in the sense that if either there exist one or more finite EDBs for which containment does not hold or there is no EDB (finite or infinite), the CQC Method terminates according to our completeness results (Theorems 5.2 and 5.5). Nevertheless, we cannot guarantee termination under the presence of infinite counterexamples.

One of the forms to assure always termination when using the CQC Method is to delimit a priori the type of schemas and queries for which it is guaranteed that infinite non-containment counterexamples never exist. It is well known that this is the case of all the different classes of conjunctive queries, including those allowing negated EDB atoms and built-in atoms. Then, we can guarantee that our method will always terminate in these cases.

7 Related Work

As we have seen, the CQC method deals with queries and database schemas that include negation on IDB predicates, integrity constraints and built-in order predicates. Previous methods that deal with negated IDB subgoals can be classified in two different approaches: either they check Uniform Query Containment or they consider just restricted cases of negation.

[LS93] checks *Uniform QC* for queries and databases with safe negated IDB atoms. The problem is that, as pointed out in [Sag88], Uniform QC (written $Q_1 \subseteq^u Q_2$) is a sufficient but not necessary condition for query containment. That is, if for a given pair of queries Q_1 and Q_2 we have that $Q_1 \subseteq^u Q_2$, then $Q_1 \subseteq Q_2$. On the other hand, if the result is that $Q_1 \not\subseteq^u Q_2$, then nothing can be said about whether or not $Q_1 \subseteq Q_2$ holds. In contrast, we have seen that the CQC method always checks "true" query containment.

The rest of the methods that handle negation restrict the classes of queries and database schemas they are able to deal with. Thus, we have that [Ull97, WL03] consider only conjunctive queries with negated EDB predicates, while [LS95] checks containment of a datalog query in a conjunctive query with negated EDB predicates. [CDL98] cannot express simple cases of negation on IDB predicates since it is not possible to define negation in the regular expression they consider.

In the long version of this paper [FTU03], we show the clear correspondence between the CQC Method and the algorithms defined in [Ull97] and [WL03] by means of examples. Our conclusion is that the CQC Method is not less efficient than those two outstanding methods for the cases that they handle.

When checking containment for conjunctive queries with negated EDB predicates, both the CQC Method with the Negation VIP and the algorithm of [Ull97] achieve the same results, but their strategies are different. The CQC builds and tests canonical EDBs dynamically since it finds one that fulfils the initial goal to attain or since no canonical EDB, with or without extension, satisfies the goal after having built all. Instead, the method of [Ull97] first builds all the canonical EDBs and then, it tests if each of them accomplishes the containment relationship.

The algorithm of [Ull97] can be easily extended to consider order predicates in the rule bodies of conjunctive queries over the two types of interpretations, dense or discrete. In this case, the canonical databases that would be built it should take into account every possible total ordering of variables to instantiate. Again, the CQC Method not only covers this class of queries but also constructs similar (canonical) EDBs with the Dense Order or the Discrete Order VIPs.

The algorithm proposed in [WL03] to check conjunctive query containment with safe negated EDB atoms improves the efficiency of [Ull97] since it does not generate necessarily the complete set of canonical EDBs that the method of [Ull97] needs to construct. If we adopt the theoretical results in which the algorithm of [WL03] is based, then it follows that the Simple VIP may replace the Negation VIP when using the CQC Method to check query containment for conjunctive queries with negated EDB subgoals, without any loss of completeness. In this way, the CQC Method with the Simple VIP and the algorithm of [WL03] become quite similar. Therefore, we do not need to generate all the canonical EDBs that the CQC Method with the Negation VIP and [Ull97] would consider and, accordingly, the CQC Method with the Simple VIP is as efficient as the algorithm in [WL03] for the cases covered by this latter.

8 Conclusions

In this paper we have presented the Constructive Query Containment (CQC) method for QC Checking which ckecks “true” QC and QCuC for queries over databases with safe negation in both IDB and EDB subgoals and with or without built-in predicates. As far as we know, ours is the first proposal that covers all these features in a single method and in a uniform and integrated way.

We have proved several properties regarding the correctness of the CQC method: finite success soundness for hierarchical queries and databases, failure soundness, finite success completeness for strict-stratified queries and databases and failure completeness for hierarchical queries and databases. From these results, and from

previous results that showed that infinite non-containment counterexamples never exist in the particular case of checking QC for conjunctive queries with safe EDB negation and built-in predicates, we can ensure termination, and thus decidability, of our method for those cases.

The main contributions of this paper are twofold. First, we have shown that the CQC method performs containment tests for more and broader cases of queries and database schemas than previous methods. Second, we have also shown that the CQC method is decidable and not less efficient than other methods to check query containment of conjunctive queries with or without safe negated EDB predicates.

As a further work, we plan to characterize other classes of queries and deductive rules for which our method always terminates.

References

[AHV95] S. Abiteboul, R. Hull, V. Vianu. Foundations of Databases. Addison Wesley, 1995.

[BM86] F. Bry, R. Manthey. Checking Consistency of Database Constraints: a Logical Basis. In Proceedings of VLDB'86, 13–20, 1986.

[CDL98] D. Calvanese, G. De Giacomo, M. Lenzerini. On the Decidability of Query Containment under Constraints. In Proceedings of the PODS'98, 149–158, 1998.

[CL89] L. Cavedon, J.W. Lloyd. A Completeness Theorem for SLDNF Resolution, Journal of Logic Programming, 7(3):177–191, 1989.

[Cla77] K.L. Clark. Negation as Failure. In Logic and Data Bases, 293–322, Plenum Press, 1977

[CM77] A.K. Chandra, P.M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In Proc. of the 9th ACM SIGACT Symposium on Theory of Computing, 77–90, 1977.

[FTU99] C. Farré, E. Teniente, T. Urpí. The Constructive Method for Query Containment Checking. In Proceedings of the DEXA'99, 583–593, 1999.

[FTU02] C. Farré, E. Teniente, T. Urpí. Formalization And Correctness Of The CQC Method. Technical Report LSI-02–68-R.

[FTU03] C. Farré, E. Teniente, T. Urpí. Query Containment With Negated IDB Predicates (Extended Version). Technical Report LSI-03–22-R.

[Klu88] A. Klug. On Conjunctive Queries Containing Inequalities. Journal of the ACM, 35(1):146–160, 1988.

[Llo87] J.W. Lloyd. Foundations of Logic Programming, Springer, 1987.

[LR96] A. Levy, M-C. Rousset. CARIN: A Representation Language Combining Horn Rules and Description Logics. In Proc. of the ECAI'96, 323–327, 1996.

[LS93] A. Levy, Y. Sagiv. Queries Independent of Updates. In Proceedings of the VLDB'93, 171–181, 1993.

[LS95] A. Levy, Y. Sagiv. Semantic Query Optimization in Datalog Programs. In Proceedings of PoDS'95, 163–173, 1995.

[Sag88] Y. Sagiv. Optimizing Datalog Programs. In Foundations of Deductive Databases and Logic Programming, 659–698, Morgan Kaufmann, 1988.

[Ull97] J. D. Ullman. Information Integration Using Logical Views. In Proc. of the ICDT'97, 19–40, 1997

[WL03] F. Wei, G. Lausen. Containment of Conjunctive Queries with Safe Negation. In Proceedings of ICDT'03: 346–360, 2003.