Scalable Containment for Unions of Conjunctive Queries under Constraints

George Konstantinidis
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292
konstant@usc.edu

Jose Luis Ambite
Information Sciences Institute
University of Southern California
Marina Del Rey, CA 90292
ambite@isi.edu

ABSTRACT

We consider the problem of query containment under ontological constraints, such as those of RDFS. Query containment, i.e., deciding whether the answers of a given query are always contained in the answers of another query, is an important problem to areas such as database theory and knowledge representation, with applications to data integration, query optimization and minimization. We consider unions of conjunctive queries, which constitute the core of structured query languages, such as SPARQL and SQL. We also consider ontological constraints or axioms, expressed in the language of Tuple-Generating Dependencies. TGDs capture RDF/S and fragments of Description Logics. We consider classes of TGDs for which the chase is known to terminate. Query containment under chase-terminating axioms can be decided by first running the chase on one of the two queries and then rely on classic relational containment. When considering unions of conjunctive queries, classic algorithms for both the chase and containment phases suffer from a large degree of redundancy. We leverage a graph-based modeling of rules, that represents multiple queries in a compact form, by exploiting shared patterns amongst them. As a result we couple the phases of both for chase and regular containment and end up with a faster and more scalable algorithm. Our experiments show a speedup of close to two orders of magnitude.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query Languages*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

The classic version of the problem we study is: given a database schema R and two queries Q_1, Q_2 on R, decide whether for all database instances I of R, it is the case that $Q_1(I) \subseteq Q_2(I)$, with $Q(I)$ being the set of answer tuples of a query Q on database I.

SWIM'13, June 23, 2013, New York, USA.

We focus on unions of conjunctive queries (UCQs). UCQs is an important class of structured queries; it corresponds to the select-project-join-union fragment of structured languages like SPARQL and SQL. Deciding containment for two conjunctive queries is an NP-complete problem [15] , which can be solved by finding a homomorphism that maps one query into the other. A UCQ Q_1 is contained in another UCQ Q_2, if every conjunctive query in Q_1 is contained in at least one conjunctive query of Q_2 [30]. Testing query containment (and query equivalence) is fundamental to database and knowledge representation systems. It is central to query optimization and minimization [15] and to data integration problems such as view-based query answering [25].

Given the great practical significance of the problem considerable effort has been devoted in order to find tractable classes of queries. Most efforts focus in identifying syntactic restrictions of queries for which polynomial-time algorithms for containment, equivalence and minimization exist [6, 21, 9]. However, less attention has been paid to developing optimized algorithms for conjunctive query containment per se, which is the focus of our work.

Recently, there has been significant interest in approaches to incorporate intensional knowledge on a database schema, in the form of ontologies, constraints or dependencies. Relevant research has focused on specific types of constraints that provide a good trade-off between expressiveness and complexity [13, 11, 12, 19, 18, 17].

We deal with query containment under constraints that can be written as tuple-generating dependencies (TGDs) [8, 2]. TGDs are a generalization of the language of inclusion dependencies. Query containment and answering under general TGDs is undecidable [7], so efforts have been made to devise syntactic restrictions of TGDs so that these problems are decidable and/or tractable. These devised fragments are usually studied along with relevant reasoning algorithms that allow us to solve the aforementioned problems. The reasoning algorithm that we employ here is the well-known chase algorithm [5, 26]. The chase is a tool that allows query answering over incomplete databases (w.r.t a set of decidable TGDs), by "completing" the data missing. The chase is also useful for checking containment; for all conjunctive queries q_1, q_2 and for all sets Σ of TGD constraints, it holds that q_1 is contained in q_2 under the constraints, iff the chase of q_1 with Σ is contained in q_2. Hence, once having an algorithm for containment of conjunctive queries, we can reuse it for queries under dependencies by firstly using the chase on one of the two queries [21, 29, 10].

The chase algorithm does not always terminate (even for decidable TGD languages). For example under DL-Lite/OWL-2 QL [13] axioms (which is a decidable language mostly expressible in TGDs) the chase does not terminate (i.e., it is infinite[1]). For such languages there are other reasoning algorithms (e.g., *perfect reformu-*

[1]The term *chase* refers both to the chase algorithm and its output.

lation [13]) for query answering and containment. In other languages (such as sticky-TGDs [12] and variants of Datalog+/- [11]) only a finite part of the infinite chase is needed. While our work has applications to these languages as well, we mainly focus on cases of TGDs for which the chase terminates (i.e., is finite), such as weakly-acyclic constraints [19] or full TGDs [2] (which capture RDF/S). In this paper in particular, we focus on weakly acyclic constraints with a single antecedent predicate. Our contributions are the following:

- We leverage a previously introduced graph-modeling of rules [23], which can represent multiple queries into compact graph structures, exploiting overlapping parts of these queries (Sect. 3).
- This modeling allows us to create an efficient index for the predicates in a UCQ. Through this index we can map (or fail to map) a certain predicate into a set of predicates/queries at once. Moreover for a certain predicate pattern we keep pointers to other joined predicates in the UCQ.
- This allows us to compactly chase a UCQ, by triggering constraints for a set of queries in batch, and by adding multiple consequents across queries in a single step. This compact chase algorithm runs much faster than the classic chase algorithm, and in addition it results in a compact representation of the chased queries.
- This compact graph representation of UCQs allows us to compute homomorphisms among multiple queries in batches, saving the redundant cost of checking each predicate of each individual query. These designs result in a faster, more scalable containment under constraints. We experimentally show (Sect. 5) that we can check containment among UCQs of several hundreds of queries, under hundreds of constraints, being two orders of magnitude faster than the classic solutions.

2. PRELIMINARIES

Queries and Containment. We use the well-known notions of variables, predicates, terms, and atoms of first-order logic. We use safe conjunctive queries (CQs); these are rules of the form $q(\vec{x}) \leftarrow P_1(\vec{y_1}), ..., P_n(\vec{y_n})$ where q,P_1, ..., P_n are predicates of some finite arity and $\vec{x}$,$\vec{y_1}$,...,$\vec{y_n}$ are tuples of variables. In the scope of the current paper we only consider variables in our queries. We define the body of the query to be $body(q) = \{P_1(\vec{y_1}), ..., P_n(\vec{y_n})\}$, while $q(\vec{x})$ is the *head* of the query. Predicates appearing in the body of a query stand for relations of a database schema R, while the head represents the answer relation of the query over R. The query being safe means that $\vec{x} \subseteq \bigcup_{i=1}^{n} \vec{y_i}$. All variables in the head are called *head*, *distinguished*, or *returning* variables, while the variables appearing only in the body (i.e., those in $\bigcup_{i=1}^{n} \vec{y_i} \setminus \vec{x}$) are called *existential* variables. The same variable name used in two predicates denotes equality of the corresponding arguments of the predicates, within one formula (variables across multiple rules are considered different). For all sets of atoms S, $vars(S)$ is the set of variables appearing in all the atoms in S (similarly, $vars(Q)$ is the set of all query variables). $q(I)$ refers to the result of evaluating the query q over the database instance I. A union of conjunctive queries (UCQ) is a set of queries, all having the same head. When considering UCQs, we will formally refer to the UCQ as the query while an individual conjunctive query in a query will be explicitly called so. We denote conjunctive queries with lower-case letters (e.g., q), while UCQs use upper-case letters (e.g., Q). The result of evaluating a UCQ Q over a database I is $Q(I) = \bigcup_{q \in Q} q(I)$.

The following query asks for doctors treating patients with chronic diseases:

q_1(doc, dis) ← TreatsPatient(doc,pat), HasChronicDisease(pat,dis)

In our example, q_1 asks only for references to the doctors and the diseases they treat, but not to the patients that have these diseases; this information conceptually exists in the body but is not required in the answer. In effect, 'pat' is an existential variable in q_1.

DEF. 1. ***Query Containment:*** *For all schemas R, for all queries Q_1, Q_2 on R, query Q_2 is contained in query Q_1, denoted by $Q_2 \subseteq Q_1$, iff for all database instances I of R, $Q_2(I) \subseteq Q_1(I)$.*

DEF. 2. ***Query Equivalence:*** *Q_1 is equivalent to Q_2, denoted $Q_1 \cong Q_2$ iff $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$.*

To ground the above definitions, consider the following query which asks for doctors treating chronic diseases such that the doctor is also a surgeon:

q_2(d, ds) ← TreatsPatient(d,p), HasChronicDisease(p,ds), Surgeon(d)

It is easy to notice that $q_2 \subseteq q_1$, since q_2 asks for the same information as q_1 plus an additional join with $Surgeon$ which can only cut down on the answers. Also, the queries are not equivalent since q_1 returns doctors who might not be surgeons. These intuitions can be formalized with the use of containment mappings (which use homomorphisms) [15, 2].

DEF. 3. ***Homomorphism:*** *Given two sets of atoms S_1, S_2, a homomorphism from S_1 to S_2 is a function h:$vars(S_1) \rightarrow vars(S_2)$, such that for all atoms $A(\vec{x}) \in S_1$, it holds that $A(h(\vec{x})) \in S_2$.*

DEF. 4. ***Containment Mappings:*** *Given two conjunctive queries q_1, q_2, a containment mapping from q_1 to q_2, is a homomorphism h:$body(q_1) \rightarrow body(q_2)$ s.t. $h(head(q_1)) = head(q_2)$ (a homomorphism h is extended over atoms, sets of atoms, and queries in the obvious manner). For same schema conjunctive queries q_1, q_2, $q_2 \subseteq q_1$ iff there is a containment mapping from q_1 to q_2.*

For the aforementioned query, the containment mapping that proves that $q_2 \subseteq q_1$, is $h : \{doc \rightarrow d, dis \rightarrow ds, pat \rightarrow p\}$ which "maps" q_1 to q_2. It is important to emphasize that containment mappings map distinguished to distinguished variables. For example, q_3 below is not contained in q_1 nor q_2 as the disease is not being returned.

q_3(d) ← TreatsPatient(d,p), HasChronicDisease(p,ds), Surgeon(d)

For deciding query containment among unions of conjunctive queries, the following holds [30].

PROPOSITION 1. *For all UCQs Q_1 and Q_2, $Q_2 \subseteq Q_1$ iff for all conjunctive queries $q_j \in Q_2$, there exists a conjunctive query $q_i \in Q_1$ and a containment mapping from q_i to q_j.*

As mentioned, deciding containment is an NP-complete problem and containment mappings is an expensive procedure (exponential time is expected, unless P=NP). Moreover for UCQs this procedure has to be repeated (worst-case) for all pairs of conjunctive queries in the two unions. As we will explain later, our algorithms try to exploit overlapping parts of the conjunctive queries in a union so as to prune the number of candidate containment mappings.

Constraints on the relational schema. Various forms of constraints have been studied in the literature. Our focus, Tuple Generating Dependencies, are a generalization of inclusion dependencies. TGDs are formulas of the form: $\forall \vec{x}, \vec{z}\, \phi(\vec{x}, \vec{z}) \rightarrow \exists \vec{y}\, \psi(\vec{x}, \vec{y})$, with ϕ and ψ conjunctive formulas over a schema R and $\vec{x}, \vec{z}, \vec{y}$ tuples of variables.

The semantics of query answering in the presence of constraints can be formalized using the notion of certain answers [1, 20]. Intuitively, a tuple t is a certain answer of a query Q over a schema R

with respect to a set of constraints if t is an answer in any database I of R that is consistent with the constraints. In general, query answering and containment under TGDs is undecidable [8, 14]. Nevertheless several syntactic restrictions have been studied that provide expressive and useful fragments of TGDs, which are decidable in the problems above, and some times even computationally tractable. Note that apart from TGDs there is another important class of dependencies in the literature, the equality generating dependencies (generalizations of functional dependencies) [14]. Although we focus on TGDs we conjecture that our results are applicable to certain classes of EGDs as well.

The chase [5, 26, 19]. Let B be a conjunction of atoms of some finite arity (B can be a database instance or the body of a query). Let σ be a TGD constraint/rule of the form $\forall \vec{x}, \vec{z}\, \phi(\vec{x}, \vec{z}) \rightarrow \exists \vec{y}\, \psi(\vec{x}, \vec{y})$. We say that the rule σ is *applicable* to B iff there is a homomorphism h from the antecedent, $\phi(\vec{x}, \vec{z})$ of the rule, to B (intuitively this means that the premise of the rule holds in B), s.t. h cannot be extended from $\phi(\vec{x}, \vec{z}) \wedge \psi(\vec{x}, \vec{y})$ to B (i.e., the rule is not already satisfied). The *application* of the rule happens by adding the consequent of the rule in B. As the *chase step* we define the addition to B of $\psi(h(\vec{x}), f(\vec{y}))$, where $h(\vec{x})$ is the tuple of the images of the homomorphism h for the tuple $\vec{x}$ and $f(\vec{y})$ creates *fresh* (that is, new and unforseen) variables for all the (existential) variables in $\vec{y}$.

The *standard chase* is an exhaustive series of chase steps (rule applications) which could be terminating (i.e., be finite) or be infinite depending on the constraints at hand. By $chase_{\Sigma}(B)$ (and occasionally just $chase(B)$ when Σ is clear from context) we denote the result of chasing B with all constraints in a set Σ. Similarly for a UCQ Q, $chase_{\Sigma}(Q)$ refers to the union of $chase_{\Sigma}(q_i)$, for all conjunctive queries $q_i \in Q$. Next, we present a fairly general class of TGDs called *weakly acyclic* TGDs [19], for which the chase is guaranteed to terminate in polynomial time.

DEF. 5. ***Weakly Acyclic TGDs [19]*** *Let Σ be a set of TGDs over schema $\Re = \{R_1, R_2, ..., R_n\}$. Construct a directed graph, called the dependency graph, as follows:*
(1) there is a node for every pair (R_i, A) with A an attribute of R_i, call such pair (R_i, A) a position;
(2) add edges as follows: for every TGD $\forall \vec{x}, \vec{z}\, \phi(\vec{x}, \vec{z}) \rightarrow \exists \vec{y}\, \psi(\vec{x}, \vec{y})$ in Σ, for every x in $\vec{x}$ that occurs in ψ and for every occurrence of x in ϕ in position (R_i, A_i):

- *for every occurrence of x in ψ in position (R_j, B_k), add an edge $(R_i, A_i) \rightarrow (R_j, B_k)$ (if it does not already exist)*
- *in addition, for every existentially quantified variable y and for every occurrence of y in ψ in position (R_t, C_m), add a special edge $(R_i, A_i) \rightsquigarrow (R_t, C_m)$ (if it does not already exist).*

Then Σ is weakly acyclic (wa) if the dependency graph has no cycle going through a special edge.

In this paper we will use weakly acyclic sets of constraints which only have a single atom in the antecedent. These are known as weakly-acyclic LAV constraints [4] and are a superset of the useful class of weakly-acyclic inclusion dependencies, as well as the class of LAV TGDs with no existential variables (LAV full TGDs), including useful web ontology languages like RDF/S[2]. Weakly-acyclic LAV constraints are known to have good properties in data integration and exchange [3] and in inconsistent databases [4].

Containment under Constraints. A query Q_2 is contained in a query Q_1 under TGD constraints Σ, denoted by $Q_2 \subseteq_{\Sigma} Q_1$, iff for all databases I that satisfy Σ, $Q_2(I) \subseteq Q_1(I)$.

[2]http://www.w3.org/RDF/

Under chase-terminating sets of constraints query containment (or answering) can be solved by first chasing the candidate "containee" query (or data, resp.). The next theorem holds (even for non chase-terminating constraints).

THEOREM 1. ***CQ Containment using the chase [21, 29, 10]:*** *For all conjunctive queries q_1, q_2, for all sets of TGD constraints Σ, $q_2 \subseteq_{\Sigma} q_1$ iff there is a homomorphism that maps the $body(q_1)$ onto the chase of the $body(q_2)$, and the head of q_1 onto the head of q_2, that is $q_2 \subseteq_{\Sigma} q_1$ iff $chase_{\Sigma}(q_2) \subseteq q_1$.*

The above theorem is straightforwardly extended to check containment under constraints for UCQs. In effect, it holds that $Q_2 \subseteq_{\Sigma} Q_1$ iff $chase_{\Sigma}(Q_2) \subseteq Q_1$. In order to decide containment we can chase all conjunctive queries in Q_2 then check for regular containment, through containment mappings, according to Prop. 1.

Consider the following rules which capture "domain" and "range" properties in RDF/S, as well as "subclass" relations. Constraint c_1 states that the domain and the range of the TreatsPatient relation are Doctors and Patients respectively. Constraint c_2 states that Doctors are ClinicEmployees.

c_1: $\forall x, y$ TreatsPatient(x,y) $\rightarrow$ Doctor(x), Patient(y)

c_2: $\forall x$ Doctor(x) $\rightarrow$ ClinicEmployee(x)

Consider the following UCQ Q_4:

q_4(d) $\leftarrow$ TreatsPatient(d,p), Surgeon(d)

q_4(d) $\leftarrow$ TreatsPatient(d,p), HasCronicDisease(p,dis)

q_4(d) $\leftarrow$ TreatsPatient(d,p), Doctor(d)

Without considering the constraints, no query in Q_4 is contained in query q_5 below:

q_5(doc) $\leftarrow$ ClinicEmployee(doc), Doctor(doc)

However, under the constraints c_1 and c_2, the entire UCQ Q_4 is contained in q_5. This can seen by chasing Q_4 and noticing that there exist containment mappings from q_5 to each one of the queries in the $chase(Q_4)$:

$chase(q_4)(d) \leftarrow$ TreatsPatient(d,p), Surgeon(d), Doctor(d), Patient(p), ClinicEmployee(d)

$chase(q_4)$(d) $\leftarrow$ TreatsPatient(d,p), HasCronicDisease(p,dis), Doctor(d), Patient(p), ClinicEmployee(d)

$chase(q_4)$(d) $\leftarrow$ TreatsPatient(d,p), Doctor(d), Patient(p), ClinicEmployee(d)

The standard chase algorithm repeatedly finds homomorphisms from the rule's antecedents into the queries. We firstly notice that there is a redundancy here. In effect, the chase algorithm will separately consider the three occurrences of the predicate TreatsPatient in Q_4 and add the consequents of c_1. Similarly for every different occurrence of the Doctor predicate across all conjunctive queries, rule c_2 will be applied. In this paper we adopt a graph-based modeling of queries, introduced in [23], which can compactly represent different occurrences of the same predicate across multiple rules (in [23] this is done for views). This ends up in an optimized chase algorithm that detects homomorphisms and chases parts of multiple queries in a single rule application. Moreover this algorithm adds consequents in the same graph-based form, resulting in a compact representation of the chased queries. The compact output of our chase algorithm is tailored towards, and proves particularly useful for, optimizing the relational containment algorithm as well. Classic containment of chase(Q_4) in q_5 suffers for the same redundancies as the chase, since the algorithm has to iterate over all different predicates of chase(Q_4) in order to find mappings for the predicates of q_5, considering the same patterns multiple times. This redundancy would symmetrically be worse in the case that q_5 was a UCQ. All candidate "containing" queries would be checked (no matter how overlapping they are), until we find one that maps to a particular query in chase(Q_4). As our initial results show, our compact representation speeds up containment under constraints significantly. Next we present our graph-based modeling of queries.

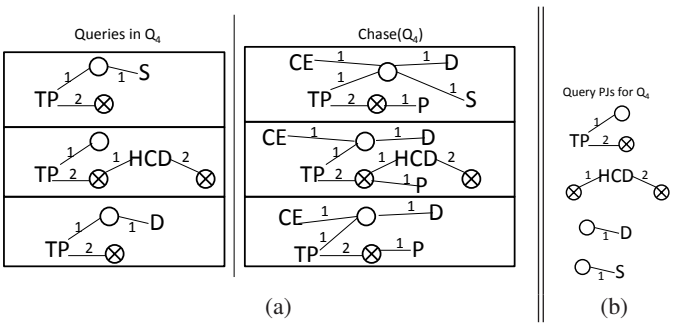

Figure 1: (a) The conjunctive queries in Q_4, and in $chase(Q_4)$ as graphs. (b) Predicate Join Patterns (PJs) for all the queries in Q_4.

3. GRAPH-BASED MODELING

Our graph representation, translates predicates and their arguments to graph nodes. Predicate nodes are labeled with the name of the predicate, and they are connected through edges to their arguments. Shared variables between atoms result in shared variable nodes, directly connected to predicate nodes. We equip our edges with integer labels that stand for the variables' positions within the atom's parentheses, and we discard variables' names; the only knowledge we require for deciding on a mapping is the types of the variables involved. Distinguished variable nodes are depicted with a circle, while for existential ones we use the symbol $\otimes$. Using these constructs the queries for Q_4 of the previous section (with abbreviated predicate names for brevity, e.g., $q_4(doc) \leftarrow TP(doc)$, $S(doc)$) correspond to the graphs seen on the left part of Fig. 1(a). The right part of Fig. 1(a) shows the graph representation (again with abbreviated predicate names) of the conjunctive queries in chase(Q_4). Our algorithms consist of mapping subgraphs of the constraints or queries to subgraphs of queries, and to this end the smallest subgraphs we consider represent one atom's "pattern": they consist of one central predicate node and its (existential or distinguished) variable nodes. We call these primitive graphs *predicate join patterns* (or PJs) for the predicate they contain. Fig. 1(b) shows all predicate join patterns that query Q_4 contains.

A critical feature that boosts our algorithms' performance is that the patterns of predicates as graphs repeat themselves in multiple queries. Therefore we choose to compactly represent each such occurrence of the same predicate across different queries with the same PJ. This has a tremendous advantage; mappings from a query PJ to another one are computed just once instead of every time this predicate (or set of predicates) is met in the queries. For a constraint predicate that has 2 variables there are 4 distinct patterns as query PJs that could potentially trigger the constraint (all combinations of distinguished/existential variables). These 4 patterns represent all different of occurrences of the constraint predicate in the queries, and unless our queries contain one of these four patterns the UCQ fails (right away) to trigger the constraint.

The "join conditions" for a particular PJ within each query are different and some "bookkeeping" is needed to capture these joins. To retain this information we use a conceptual data structure called *information box* (or infobox). Each infobox is attached to a variable. Fig. 2 shows PJs with their infoboxes. A variable's infobox contains a list of queries that this PJ appears in and for each such query the variable's *join descriptions*. This way we record which other PJs this variable (directly) joins to within any of the queries this pattern appears in. Fig. 2(a) shows for all predicates of Q_4, all the different PJs (with their infoboxes) that appear in its conjunctive queries (we assume the queries are named q_{4_1},q_{4_2}, and q_{4_3}).

In the face of multiple occurrences of the same predicate in a conjunctive query, it is suitable to imagine all query PJs discussed so far as classes of PJs: we instantiate the same query PJ that covers a specific predicate pattern as many times as this patterns appears within the same conjunctive query. Each time we instantiate the same PJ we "prime" its name creating a different PJ for it but knowing that we are using the same conjunctive pattern but a second, different time. This modeling is a natural extension of our approach for repeated predicates in the views (described in [23]). We omit further details due to space limitations.

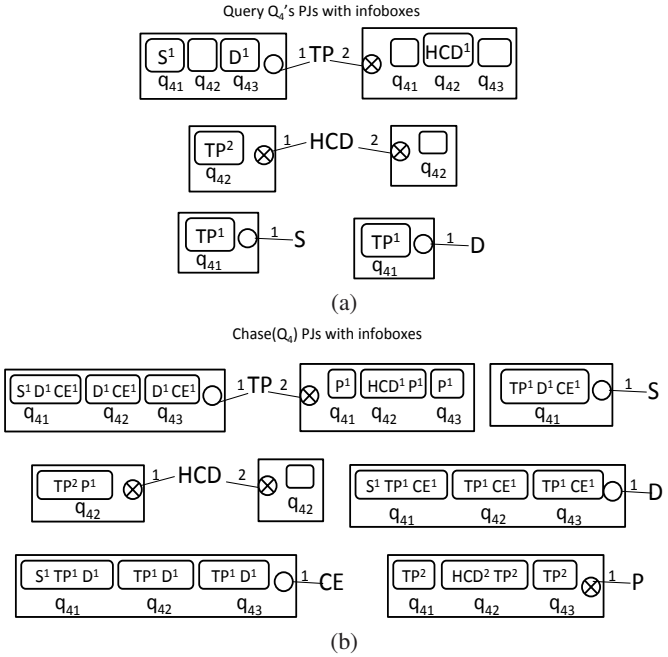

Figure 2: (a) All the PJs of Q_4 with their infoboxes. (b) All the PJs of chase(Q_4) (the result of the compact chase algorithm on the PJs of Q_4). In the two figures we see the infoboxes for all variable nodes. For example, we can find the PJ for P (bottom right corner of figure b) in three queries: q_{4_1}, q_{4_2}, and q_{4_3}. The two join descriptions related to q_{4_2} in the PJ for P tell us that its variable, in query q_{4_2}, joins with the second argument of HCD and the second argument of TP. In figure (b) all the pre-existing infoboxes have been updated by our algorithm to reflect the new chased set of queries.

4. UCQ CONTAINMENT UNDER CONSTRAINTS

Given the set of queries and considering them as graphs, we break them down to atomic PJs, by splitting each graph on the shared variable nodes. On top of the PJ generation we construct infoboxes for every variable node in those PJs. Fig. 2(a) shows all the query PJs as constructed by this phase. The details of the PJ construction algorithm are rather obvious and omitted, but it can be easily verified that this phase has a polynomial complexity to the number and length of the queries. On top of our PJ generation we create a simple index on the queries, by creating a hashtable on every different pattern (PJ) so we can retrieve it efficiently. As noted for a specific pattern we might retrieve more than one PJs if we have repeated predicates within the conjunctive queries themselves.

4.1 Our Algorithm for chasing

Algorithm 1 is used for chasing a union of queries using a set of weakly acyclic LAV dependencies. In line 1 the algorithm iterates over all constraints, and in line 2 finds all the different antecedents

in the queries. SetA in line 2 contains all different patterns (PJs) for the antecedent predicate. As mentioned SetA could contain the same exact pattern twice, in order to distinguish a predicate with same pattern repeating itself within one query. Line 3 checks that indeed there have been occurrences of the antecedent found in the queries (otherwise we can go to the next constraint).

Algorithm 1 Compact Chase

Input: A UCQ query Q, a set of LAV-WA constraints Σ
Output: A set of PJs representing the $chase_{\Sigma}(Q)$

1: **for all** $\sigma \in \Sigma : P(\vec{x},\vec{z}) \rightarrow C_1(\vec{x},\vec{y}), C_2(\vec{x},\vec{y}), ..., C_n(\vec{x},\vec{y})$ **do**
2: $SetA \leftarrow RetrievePJwithPredicate(P)$ //occurrences of $P(\vec{x},\vec{z})$ in the query
3: **if** $SetA$ is empty **then**
4: **continue** to the next constraint (line 1)
5: **else**
6: **if** σ has been triggered for all PJs in $SetA$ **then**
7: **continue** to the next constraint (line 1)
8: **for all** PJs $PJ_a \in SetA$ **do**
9: **for all** i, take consequent C_i of σ **do**
10: $SetC_i \leftarrow RetrievePJSet(C_i)$ //occurrences of $C_i(\vec{x},\vec{z})$ in the queries
11: **for all** PJs $PJ_{C_i} \in SetC_i$ **do**
12: **for all** common conjunctive queries q in PJ_a and PJ_{C_i} **do**
13: **if** checkQueryPJCanBeUsed(q,PJ_{C_i},C_i,PJ_a) **then**
14: mark C_i already satisfied in q
15: **if** there is a consequent C_i with a unsatisfied query q **then**
16: mark all other consequents as unsatisfied for query q
17: **for all** i, take consequent C_i of σ **do**
18: construct a new query PJ holding information for all conjunctive queries q not satisfied in C_i
19: add the new query PJ in our index and update the query-boxes of the other PJs

For every distinguished PJ, we trigger every applicable constraint only once; if the consequents have been added for this specific PJ, we will not apply the rule again. This is checked in line 6. In line 8 we iterate over all candidate antecedent PJs, PJ_a, in our UCQ and in the following lines we will try to see whether the consequents of the rule are already implied for some of the conjunctive queries in the infoboxes of PJ_a (those are the queries that the pattern PJ_a appears in). For some of those we might find PJs that imply them and for some others we might have to create new PJs to stand for the consequents predicates. For each consequent C_i in the rule, we retrieve all PJs, PJ_{C_i}, that match this consequent and we get all the common queries between PJ_a and PJ_{C_i} (lines 11,12). Depending on the different join descriptions of these queries inside PJ_{C_i}, the already existing PJ PJ_{C_i} might be useful to "stand as" C_i for some of those queries and for some might not. This is what the call to checkQueryPJCanBeUsed (which is shown in Alg. 2) decides.

Alg. 2 essentially checks that for a specific query the joins of C_i are described in the query's boxes inside PJ_{C_i}. This guarantees that for the specific query, C_i homomorphically maps to PJ_{C_i}. Notice that later, some other PJ_{C_j} might fail to cover another consequent, C_2, of the same constraint for the same query. This renders the entire containment infeasible. The intuition is that when we mapped PJ_{C_i} to C_i (by checking the inclusion of their joins), we believed that there is a homomorphism from all the consequents to the PJs of the specific conjunctive query. However C_j spoils that. Our algorithm remembers that fact, backtracks and "cancels" all previous associations. For ease of presentation we included this in lines 15-16 of Alg. 1; even though in our implementation this check is done at the same time as the mappings, using some pointers and data structures in order to remember "what" to cancel. Lastly in line 17, we construct new PJs to hold information for all queries and consequents left unsatisfied, essentially compactly adding consequents in the original queries.

Algorithm 2 checkQueryPJCanBeUsed(q,PJ_{C_i},C_i,PJ_a)

Input: A conjunctive query q, a prexisting query PJ PJ_{C_i}, a constraint PJ C_i, the query pattern PJ_a which unified with constraint antecedent
Output: true if PJ_{C_i} already implies C_i

1: **for all** edges k of PJ_{C_i} get node N_k **do**
2: $box_q \leftarrow$ the query box for q from N_k's infobox
3: $M_k \leftarrow$ node in edge k of C_i
4: $box_c \leftarrow$ the constraint box in M_k's infobox
5: **if** M_k is a distinguished in the constraint **then**
6: **if** joins with antecedent in $box_c \nsubseteq$ joins in box_q **then**
7: **return** false
8: **else**
9: **if** Joins in $box_c \nsubseteq$ joins in box_q **then**
10: **return** false
11: **return** true

We would like to point out line 6 in Alg. 2. This basically relaxes the demand that all joins of a variable in the constraint need to be in the query, if that variable is a distinguished variable in the constraint. If a variable is a distinguished variable in the consequent this means it belongs to the constraint antecedent as well. When we unify the constraint antecedent with a predicate p in a query all the consequents of the constraint mentioning this variable will automatically map this to the same variable. As an example, consider chasing q_6 with c_3 below.

c_3: $\forall x, y$ TreatsPatient(x,y) $\rightarrow$ Doctor(x), Surgeon(x), Patient(y)

q_6(d) $\leftarrow$ TreatsPatient(d,p), Doctor(d)

There is no homomorphism from the consequents of c_3 to q_6, nevertheless we don't have to add the $Doctor$ predicate again in q_6; the fact that x is in the antecedent of the rule means that there is only one value it can take when we add it in q_6, (and that is d since TreatsPatient(x,y) unified with TreatsPatient(d,p)). Hence if the Doctor in the query already joins with TreatsPatient on d, we don't have to look to satisfy other joins on the constraint.

Running our compact chase algorithm on the query PJs of Fig. 2(a) with constraints c_1 and c_2, results in the query PJs of Fig. 2(b).

4.2 Our Algorithm for containment

After running our compact chase algorithm we are left with a set of PJs representing our chased UCQ. In order to check containment among of this UCQ in another one, we transform the second one into PJs as well and run Alg. 3.

In line 1, our algorithm keeps list $queriesLeft$ with all conjunctive queries q_1 in the "containee[3]" UCQ, Q_1 (the one coming out from the chase algorithm). As soon as we find a query q_2 in the other UCQ, Q_2 (the "containing" one), such that $q_1 \subseteq q_2$, we remove q_1 from the list $queriesLeft$. Alg. 3 starts (line 2) by iterating among all PJs for Q_1. For each such PJ, PJ_{Q_1} we retrieve the same exact patterns that appear in Q_2. If we fail to retrieve any PJ in Q_2 having the same pattern as PJ_{Q_1}, the algorithm instantly fails (line 4), since this means that the queries of PJ_{Q_1} cannot be covered (i.e., proven contained). If we do retrieve some PJs we will try to use them to prove containments into all the queries in the infoboxes of PJ_{Q_1}, that have not already been satisfied (line 6). For all such conjunctive queries q_1 in PJ_{Q_1}, we iterate over the retrieved PJs of Q_2 and the queries they contain (lines 7-8).

[3]We are using the terms "'containee" and "containing" to ease the presentation even if containment might fail for some queries.

Algorithm 3 checkContainment({PJs in Q_1},{PJs in Q_2})

Input: Two sets of PJs representing the PJs of two UCQs Q_1 and Q_2 resp.
Output: true if $Q1 \subseteq Q_2$

```
1: queriesLeft ← all conjunctive queries in Q1
2: for all PJ_{Q_1} ∈ {PJs in Q_1} do
3:    PJsInQ2ForQ1 ← RetrievePJs(PJ_{Q_1},{PJs in Q_2})
4:    if PJsInQ2ForQ1 is empty then
5:       return false
6:    for all conjunctive queries q_1 in the intersection of
      queriesLeft and PJ_{Q_1} do
7:       for all PJ_{Q_2} ∈ PJsInQ2ForQ1 do
8:          for all conjunctive queries q_2 in PJ_{Q_2} do
9:             for all edges k of PJ_{Q_2} get node N_k do
10:               box_{q_2} ← the query box for q_2 from N_k's infobox
11:               M_k ← node in edge k of PJ_{Q_1}
12:               box_{q_1} ← the query box for q_1 from M_k's
13:               if Joins in box_{q_2} ⊈ joins in box_{q_1} then
14:                  continue queries in PJ_{Q_2} //goto line 8
15:               else
16:                  for all joined/neighbour PJs NPJ_{Q_2} in {PJs in
                     Q_2} as described in box_{q_2} do
17:                     if there is no joined/neighbour PJ NPJ_{Q_1}
                        in {PJs in Q_1} as described in box_{q_1}, s.t.
                        checkPJQ2mapsOnPJQ1(NPJ_{Q_2},NPJ_{Q_1},q_1,q_2)
                        then
18:                        continue queries in PJ_{Q_2} //goto line 8
19:            queriesLeft = queriesLeft \ q_1
               // reaching here means q_2 maps to q_1 so q_1 is con-
               tained
20:            if queriesLeft is empty then
21:               return true
22:            else
23:               continue queries in PJ_{Q_1} //goto line 6
24: return false
```

Inside the for loop in line 8 we consider whether a specific PJ_{Q_2} for one of the queries it contains, say q_2, can map onto query q_1 in PJ_{Q_1}. This is done by looking in all variables of PJ_{Q_2} and getting the joins described for q_2 (line 10). If those joins are not contained in q_1 in PJ_{Q_1}, q_2 cannot map to q_1, so we should try the next candidate containing query in the infobox of PJ_{Q_2} (lines 13-14). Else, if the joins are contained in the information related to q_1 in the infobox of PJ_{Q_1}, this is an indication that q_2 might map on q_1. However we are not done yet since we need to make sure that the other (joined) predicates of q_2 in PJ_{Q_2} can themselves map to q_1. Lines 16-18 describe this; we *follow the joins* described in the corresponding infoboxes of PJ_{Q_2} and PJ_{Q_1} and make sure the "neighboring" PJs also map to each other for q_2 and q_1. In fact since we are looking to map q_2 in q_1, only one (line 17) of the neighbors of PJ_{Q_1} is sufficient to cover a neighbor of PJ_{Q_2} (essentially this says that q_1 can have more joins). The aforementioned check involves a call to Alg. 4 which is almost identical to Alg. 2 and always checks that the neighbors indeed map on all variables for queries q_1, q_2 (variable types don't matter here as in Alg. 2). Line 19 is in the for loop of line 8 and if we reach there it means q_1 was satisfied (otherwise previous lines would jump back to the beginning of the loop and exhaust it). Hence in lines 19-23 we remove q_1 from our list and goto line 6 to continue with the next containee conjunctive query.

5. EXPERIMENTAL EVALUATION

We evaluated our approach by comparing against our implementation of the brute force algorithms for chase and UCQ containment using containment mappings. To the best of our knowledge there is no other optimized algorithm available for the definitions we introduced in section 2. The classic chase algorithm is straightforward: we get all containee query predicates and for all those for which a chase rule is applicable (i.e., there is a homomorphism from the antecedent to the query predicate, that cannot be extended over the consequents) we add the (image of the) consequents in the query.

Algorithm 4 checkPJQ2mapsOnPJQ1(PJ_{Q_2},PJ_{Q_1},q_1,q_2)

Input: Conjunctive "containee" query q_1, and conjunctive "containing" query q_2, a "containee" query PJ PJ_{Q_1}, and a "containing" query PJ_{Q_2}
Output: true if q_2 in PJ_{Q_2} maps to q_1 in PJ_{Q_1}

```
1: for all edges k of PJ_{Q_2} get node N_k do
2:    box_{q_2} ← the query box for q_2 from N_k's infobox
3:    M_k ← node in edge k of PJ_{Q_1}
4:    box_{q_1} ← the query box for q_1 from M_k's infobox
5:    if joins in box_{q_2} ⊈ joins in box_{q_1} then
6:       return false
7: return true
```

The classic containment that we implemented takes all chased containee query predicates and then looks in the first predicates of every containing query until it finds one that can map onto the containee predicate. For a query to be containing, all its predicates must map to the containee, so we choose the first one as a "seed" for the homomorphism. For that first containing predicate and a given containee query, if we don't find a mapping we check the next containee predicate, of the same query. If we do, then we check that the rest of the containing query has an extended containment mapping to the containee query; if not, we go again to the next containee predicate, for the same query. Alg.5 describes this procedure.

Algorithm 5 checkBruteForceContainment(Q_1,Q_2)

Input: A "containee" UCQ query Q_1, and a "containing" UCQ query Q_2
Output: true if $Q_1 \subseteq Q_2$

```
1: for all conjunctive queries, q_1 ∈ Q_1 do
2:    for all conjunctive queries, q_2 ∈ Q_2 do
3:       firstAtom ← first atom of q_2
4:       for all atoms p_1 ∈ q_1 do
5:          if exists a containment mapping from firstAtom to p_1
            then
6:
7:             if exists an extension to the containment mapping
               from the rest of q_2 to q_1 then
8:                continue with the next containee query
                  //goto line 1
9:             else
10:               continue with the next containee predicate
                  //goto line 4
11:         else
12:            continue with the next containee predicate
               //goto line 4
13:   return false
      //if we reach here there is some q_1 which could not be
      mapped by any q_2
14: return true
```

We used the random-data generator from [23] to produce 1000 chain queries (queries where each predicate joins with the next one). We created a space with 20 predicate names out of which each conjunctive query chooses randomly 8 to populate its body and it can choose the same one up to 5 times (for instantiating repeated predicates). Each atom has 4 randomly generated variables. We generated the first 80 queries with 10 head variables, and the rest with just 3. Having less distinguished variables among the queries in general, makes the containment problem harder as containment mappings need to map distinguished to distinguished

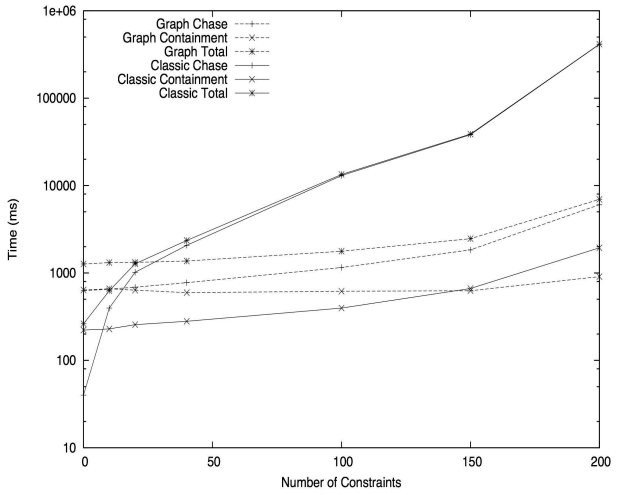

Figure 3: Checking containment for two UCQs of 700 queries each, under various numbers of constraints. The containment check fails for all cases. We run each experiment 5 times and took the average times.

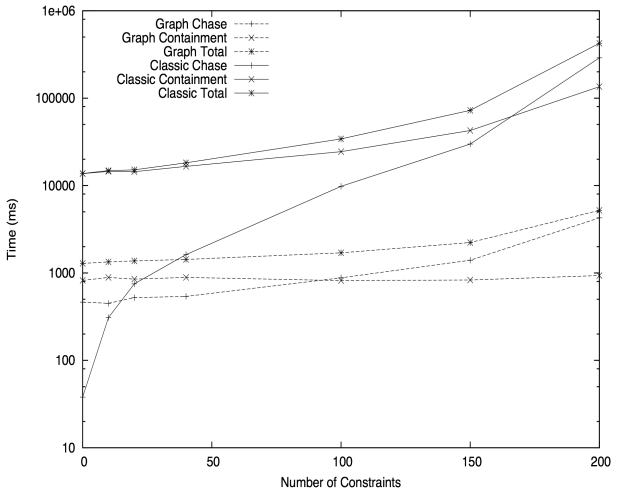

Figure 4: Checking containment for two UCQs with 500 and 1000 queries resp. under various numbers of constraints. The containment check always succeeds. We run each experiment 5 times and took the average times.

variables. For generating our constraints we wrote a weakly-acyclic constraint generator, and we generated 200 constraints. Each constraint had a single antecedent predicate with 4 head variables, and 4 consequent predicates (with 4 variables each) joined in a chain. In order to generate the predicates and variables of constraints we, again, chose randomly from the same predicate and variable space as in the queries. Each constraint could have up to 3 repeated predicates. We run our experiments on a mac book with a 2.3GHz processor. We implemented our code in Java and gave 2GB of RAM to the running environment. We run two sets of experiments.

For the first run we randomly chose two sets of 700 queries out of the same 1000 above. We ended with two UCQs with no containment between them. This fact does not change even when we chase the containee UCQ and add more predicates to it (when $q_2 \not\subseteq q_1$, then chase(q_2) $\not\subseteq q_1$ as well). We run containment checks for these two sets of 700 queries under several numbers of constraints.

Fig. 3 shows our results. As the number of constraint grows, our total time becomes about two orders of magnitude better than the classic algorithms. Our total time is divided into the graph chase time and the graph containment time. The latter is the time it takes for Algorithm 3 to check containment after the queries have been chased; it hence gives a feeling of how our algorithm behaves when we have no constraints, and rather we have "longer" (chased) containee queries. From the graph of Fig. 3 we see that when: 1) there is no containment among the UCQs, 2) there are no constraints (interpreting each point of the 'x' axis as a point with no constraints but with longer "chased" containee queries), and 3) the queries in the containee UCQ are not much "longer" than the containing queries ($x \leq 150$), the classic algorithm for containment seems to perform better. This is because it is sufficient for one query to be proved non-contained and the classic algorithm stops. The classic containment algorithm pays the full cost when there are actually containments for every query; it then has to check all of them. Nevertheless we see that as the queries become larger (e.g. after chasing them with 150 constraints in this setting), our graph containment starts performing better (still with no constraints and no containment). Our graph-based approach for just the containment induces a cost of generating graphs which does not pay off when the algorithms fail fast to prove containment, since the containee queries are "short". Nevertheless our compact format of the containee queries pays off as they get longer ($x \geq 150$).

For our second experiment we would like to see how the algorithms perform when there are containment mappings (Fig. 4). Hence for our containee UCQ we randomly chose 500 queries from our original set of 1000 queries, and we checked for containment against the latter (containment always exists). Here our graph-based approach outperforms the classic times in all phases of the problem, again by about two orders of magnitude. Interestingly, it seems that once we have the chased containee queries as graphs our containment time seems to remain constant, which means that the algorithm efficiently navigates through our compact graphs and finds the same containments at the same time even though the length of the queries grows. This is still true when ignoring the constraints and assuming relatively short ($x > 30$) chased queries as our input.

The dominating time in the algorithms in both figures is the time to chase the queries. This is because while in the containment case it is sufficient for one query to fail, in the chase case one needs to consider all predicates of all containee queries. Our compact chase algorithm does this much faster and hence is a clear win in all interesting cases in both figures. Moreover, when we consider constraints as part of the problem our algorithms combined run much faster than the classics, in almost all cases (with more than 30 constraints). An additional advantage of our approach in the presence of constraints is that our chase algorithm outputs our graphs right away (so in a sense we get those for free for the containment phase).

6. RELATED WORK

The problem of query containment has been thoroughly studied, starting with the seminal work of Chandra and Merlin [15], who proved that conjunctive query containment, without constraints, is an NP-complete problem. Containment of conjunctive queries with comparison predicates is π_2^P [22], and containment of datalog programs is undecidable [16]. Containment of conjunctive queries under unrestricted functional and inclusion dependencies is undecid-

able [28]. Starting with the work of Johnson and Klug [21] a number of decidable combinations have been explored, using the chase [27] as the core reasoning tool. Ensuring termination of the chase by imposing syntactic restrictions on the form of the constraints has been particularly fruitful [10, 11, 12, 31]. We follow on this work by presenting an algorithm for query containment under LAV weakly-acyclic dependencies.

7. CONCLUSIONS & FUTURE WORK

We have presented a radically improved (by two orders of magnitude) solution to the problem of UCQ query containment under weakly acyclic LAV dependencies, for which the chase terminates, and which can represent practically important languages such as RDFS. Inspired by our previous work on query rewriting [23], we achieve significant scalability by exploiting common patterns in the constraints and queries. Thus, we provide a practical algorithm to reason and optimize query evaluation in the semantic web.

We are working towards extending the languages of supported constraints. Extending to TGDs with more than one predicates in the antecedent should be relatively straightforward since we already have a graph-based method for computing homomorphisms among conjunctive formulas. Extending to non chase-terminating cases of constraints would need a preprocessing of the containing query as well, with algorithms that look more like perfect reformulation [13] rather than the chase. In line with our previous work [23, 24] we would like to extend the algorithms presented here for supporting query answering using views under constraints.

8. REFERENCES

[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *SIGMOD*, pages 254–263, Seattle, Washington, 1998.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Citeseer, 1995.

[3] F. Afrati and N. Kiourtis. Computing certain answers in the presence of dependencies. *Information Systems*, 35(2):149–169, 2010.

[4] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *Proceedings of the 12th ICDT*, pages 31–41. ACM, 2009.

[5] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database Systems (TODS)*, 4(3):297–314, 1979.

[6] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems (TODS)*, 4(4):435–454, 1979.

[7] C. Beeri and M. Vardi. The implication problem for data dependencies. *Automata, Languages and Programming*, pages 73–85, 1981.

[8] C. Beeri and M. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[9] J. Biskup, P. Dublish, and Y. Sagiv. Optimization of a subclass of conjunctive queries. *Acta informatica*, 32(1):1–26, 1995.

[10] A. Calı, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *Proc. of KR*, pages 70–80, 2008.

[11] A. Cali, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *Logic in Computer Science (LICS), 2010 25th Annual IEEE Symposium on*, pages 228–242. IEEE, 2010.

[12] A. Calì, G. Gottlob, T. Lukasiewicz, and A. Pieris. A logical toolbox for ontological reasoning. *SIGMOD Record*, 40(3):5, 2011.

[13] D. Calvanese, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of automated reasoning*, 39(3):385–429, 2007.

[14] A. K. Chandra, H. R. Lewis, and J. A. Makowsky. Embedded implicational dependencies and their inference problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 342–354. ACM, 1981.

[15] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th ACM Symposium on Theory of Computing (STOC)*, pages 77–90, Boulder, Colorado, 1977.

[16] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 55–66, San Diego, CA, 1992.

[17] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *ACM SIGMOD Record*, 35(1):65–73, 2006.

[18] R. Fagin, P. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Transactions on Database Systems (TODS)*, 30(1):174–210, 2005.

[19] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89 – 124, 2005.

[20] S. Greco, C. Molinaro, and F. Spezzano. Incomplete data and data dependencies in relational databases. *Synthesis Lectures on Data Management*, 4(5):1–123, 2012.

[21] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *PODS*, 1982.

[22] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1), Jan. 1988.

[23] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: A graph-based approach. In *ACM SIGMOD Conference*, Athens, Greece, June 2011.

[24] G. Konstantinidis and J. L. Ambite. Optimizing query rewriting for multiple queries. In *Proceedings of the 9th Workshop on Information Integration on the Web (IIWeb 2012)*, Scottsdale, Arizona, 2012.

[25] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the 14th ACM Symposium on Principles of Database Systems*, pages 95–104, San Jose, California, 1995.

[26] D. Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983.

[27] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, Dec. 1979.

[28] J. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56:112–138, 1983.

[29] A. Nash, A. Deutsch, and J. Remmel. *Data Exchange, Data Integration, and Chase*. TR, CS2006-0859, UCSD, 2006.

[30] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4):633–655, 1980.

[31] F. Spezzano and S. Greco. Chase termination: A constraints rewriting approach. *PVLDB*, 3(1):93–104, 2010.