Decidability of First-Order Logic Queries over Views

To appear in ICDT 99

James Bailey[1] and Guozhu Dong[2]

[1] Department of Computer Science, King's College London
Strand, London WC2R 2LS, United Kingdom, james@dcs.kcl.ac.uk
[2] Department of Computer Science, The University of Melbourne
Parkville 3052, Australia, dong@cs.mu.oz.au

Abstract. We study the problem of deciding satisfiability of first order logic queries over views, our aim being to delimit the boundary between the decidable and the undecidable fragments of this language. Views currently occupy a central place in database research, due to their role in applications such as information integration and data warehousing. Our principal result is the identification of an important decidable class of queries over unary conjunctive views. This extends the decidability of the classical class of first order sentences over unary relations (the Löwenheim class). We then demonstrate how extending this class leads to undecidability. In addition to new areas, our work also has relevance to extensions of results for related problems such as query containment, trigger termination, implication of dependencies and reasoning in description logics.

Key words: satisfiability, decidability, first order logic, database queries, database views, conjunctive queries, unary views, inequality, the Löwenheim class.

1 Introduction

The study of views in relational databases has recently attracted considerable attention, with the advent of applications such as data integration and data warehousing. In such domains, views can be used as "mediators" for source information that is not directly accessible to users. This is especially helpful in modelling the integration of data from diverse sources, such as legacy systems and/or the world wide web.

Much of this recent research has been on fundamental problems such as containment and rewriting/optimisation of queries using views [12]. In this paper, we examine the use of views in a somewhat different context, such as for mediating and monitoring constraints on data sources where only views are available.

An important requirement is that the resulting query/constraint language be *decidable*. We give results on the decision problem in this paper. As a by-product, our results also help us to analyse the specific problems mentioned above.

1.1 Statement of the Problem

Consider a database with a schema over a set of base relations $R_1, \ldots, R_p$ and a set of views $V_1, \ldots, V_n$ defined over them. A *first order view query* is a first order query *solely* in terms of the given views. For example, given the views

$$V_1(X,Y) \leftarrow R(X,Y), S(Y,Y,Z), T(Z,A,X), \neg Q(A,X)$$
$$V_2(Z) \leftarrow R(Z,Z)$$

then $\exists X,Y((V_1(X,Y) \vee V_1(Y,X)) \wedge \neg V_2(X)) \wedge \forall Z(V_2(Z) \Rightarrow V_1(Z,Z))$ is an example first order view query, but $\exists X,Y(V_1(X,Y) \vee R(Y,X))$ is not.

First order view queries are realistic for applications where the source data is unavailable, but summary data (in the form of views) is. Since most database languages are based on first order logic (or extensions thereof), we choose this as the language for manipulating the views.

Our purpose in this paper is to determine, for what types of view definitions, satisfiability (over finite models) is decidable for the language. If views can be binary, then this language is as powerful as first order logic over binary base relations, and hence undecidable (even with only identity views). The situation becomes much more interesting, when we restrict the form views may take - in particular, when their arity must be unary. Such a restriction has the effect of constraining which parts of the database can be "seen" by a query and the way different parts of the database may interact.

1.2 Contributions

Our main contribution is the definition of a language called the *first order unary conjunctive view language* and a proof of its decidability. As the name suggests, it uses unary arity views defined by conjunctive queries. It is a maximal decidable class in the sense that increasing the expressiveness of the view definition results in undecidability. Some interesting aspects of this result are:

- It is well known that first order logic over solely monadic relations is decidable, but the extension to dyadic relations is undecidable [3]. The unary conjunctive view language can be seen as an interesting intermediate case between the two, since although only monadic predicates (views) appear in the query, they are intimately tied to database relations of higher arity.
- The language is able to express interesting properties such as generalisations of unary inclusion dependencies [6] and containment of monadic datalog queries under constraints. This is discussed in Section 4.

1.3 Paper Outline

The paper is structured as follows: Section 2 defines the unary conjunctive view language and shows its decidability. Section 3 shows that extensions to the language, such as negation, inequality or recursion in views, result in undecidability. Section 4 looks at applications of the results. Section 5 gives some comparisons to other work and section 6 summarises and then looks at future work.

2 Unary Conjunctive Views

The first language we consider is first order queries (without inequality) over unary conjunctive views. Each view has arity one and it is defined by a union of some arbitrary conjunctive queries. Some example views are

$$\begin{aligned}
&V_1(X) \leftarrow R(X,Y), R(Y,Z), R(Z,X)\\
&V_2(X) \leftarrow S(X,X,Y), T(Z,X), R(Y,Y)\\
&V_2(X) \leftarrow S(X,X,X)\\
&V_3(X) \leftarrow T(X,Y), R(Y,X), S(X,Y,Z)
\end{aligned}$$

and an example first order query over them is

$$\exists X (V_1(X) \wedge \neg V_3(X) \wedge \forall Y (V_2(X) \Rightarrow (\neg V_1(Y) \vee V_3(Y))))$$

Note that this sentence can of course be rewritten to eliminate the views, since each view is just a positive existential query (e.g. $V_1(X)$ can be rewritten as $\exists Y, Z(R(X,Y) \wedge R(Y,Z) \wedge R(Z,X))$). Observe that there is no restriction on the number of variables that can be used by a view. Since this language can express the property that a graph has a clique of size $k+1$, it is not subsumed by any finite variable logic L_k (for finite k) [7]. Of course, having unary arity does have a price. In particular, some restrictions inherent in this language are

- *Limited negation:* Since negation can only be applied to the unary views, properties such as transitivity $(\forall X, Y(edge(X,Y) \wedge edge(Y,Z) \Rightarrow edge(X,Z)))$ cannot be expressed.
- *No Inequality*: It isn't possible to test inequality of variables and in particular it is impossible to write a sentence specifying that a single attribute X of a relation functionally determines the others (namely $\neg\exists X, Y, Z(edge(X,Y) \wedge edge(X,Z) \wedge Y \neq Z)$.

Our results in this paper verify that this language indeed has these two limitations, since the language itself is decidable, whereas adding $\neq$ or $\neg$ results in undecidability. Given these limitations, a proof of decidability might seem to be straightforward. However, this turns out not to be the case. Our plan is to show that the first order view queries have a bounded model property, for arbitrary fixed view definitions. To accomplish this, we need to construct a small model using a bounded number of constants from a large model. Difficulties arise, however, from the fact that sentences such as $\neg\exists X(V_1(X) \wedge V_2(X) \wedge \neg V_3(X) \wedge \neg V_4(X))$

can be expressed. Such sentences act like constraints on the space of possible models. When constructing the small model, we need to map different constants in the large model to identical ones, and we must do so without violating such constraints. We achieve this by using a procedure similar to chasing, plus an interesting duplicating and wrap-around technique. The rest of this section is devoted to describing this procedure and technique.

Theorem 1. *Satisfiability for the first order unary conjunctive view query language is decidable. In fact it is in $SPACE(2^{(m \times p)^m})$, but $NTIME(2^{p/\log p})$ hard, for queries using views of maximum length m over p relations.*

Proof. Let ψ be an arbitrary first order view query in the language. The first step consists of transforming ψ to ψ' in order to eliminate the union of views, that is, ψ' is equivalent to ψ and ψ' only uses views without union. ψ' is easily constructed by replacing each $V(X)$ in ψ using the disjunct $V_1(X) \vee V_2(X) \ldots \vee V_w(X)$, where (i) V's definition uses union, (ii) $V_1, \cdots, V_w$ is an enumeration of the conjunctive views in the definition of V, and (iii) no union is used in defining each V_i. Let there be n views in ψ' over the p relations. Let m be the maximum number of variables (including duplicates) used in defining any view body. For example, $m = 7$ for the views listed at the beginning of section 2. In addition to these views, we define all possible other views using at most m variables (including duplicates). Let N denote the total number of non isomorphic views obtained. Then it can be easily verified that $N \leq (m \times p)^m \times m$. We then "and" the formula $(V_x \vee \neg V_x)$ with ψ', for each view V_x that didn't originally appear in $V_1, \ldots V_n$. The purpose of defining these "redundant" views is to ensure maximum coverage of equivalence classes (whose definition will be given shortly).

It is possible to interpret ψ' in two possible ways. The standard way is by treating it as a first order formula over base-relation schema $\{R_1, ..., R_p\}$ (and indeed this corresponds to our intended meaning). It is also possible to interpret it in a non standard way, as a first order formula over the view schema $\{V_1, ..., V_N\}$, where each V_i is treated as a *materialised view.* Let us define a function $f : I_{base} \rightarrow I_{view}$, which relates base-relation models to view models. f is clearly a total function, since any base-relation model clearly has a view-model counterpart. f is not an onto function though, since some view-model interpretations could not be realised by the underlying base relations.

We now define equivalence classes and equivalence class states for the query ψ'. An *equivalence class* is simply a formula over the N views

$$C(X) = \{< X > \mid {}^{(\neg)}V_1(X) \wedge^{(\neg)} V_2(X) \wedge \cdots \wedge^{(\neg)} V_N(X)\},$$

where each $V_j(X)$ can appear either positively or negatively. Examples of such a class include $\{< X > \mid V_1(X) \wedge V_2(X) \wedge \cdots \wedge V_N(X)\}$ and $\{< X > \mid {}^{(\neg)}V_1(X) \wedge V_2(X) \wedge \cdots \wedge V_N(X)\}$. These equivalence classes can be enumerated as $C_1, C_2, \ldots, C_{2^N}$.

Using standard techniques, which we explain below, any instance which satisfies ψ' can be completely described by the behaviour of these equivalence classes. An equivalence class C_i appears positively in a model for ψ' if the formula

$\exists X C_i(X)$ is true and it appears negatively if the formula $\neg\exists C_i(X)$ is true. An *equivalence class state* is a partition of all the equivalence classes into two groups - those appearing positively and those appearing negatively. We number the positive classes in such a state by $C_1, \ldots, C_{n_1}$ and the negative classes $C'_1, \ldots, C'_{n_2}$. Clearly $n_1 + n_2 = 2^N$. Thus an equivalence class state is described by the formula

$$\begin{aligned}&\exists X C_1(X) \wedge \exists X C_2(X) \wedge \ldots \wedge \exists X C_{n_1}(X)\\ &\wedge \neg\exists X C'_1(X) \wedge \neg\exists X C'_2(X) \wedge \ldots \wedge \neg\exists X C'_{n_2}(X).\end{aligned}$$

Let $\phi_1, \cdots, \phi_{2^{2^N}}$ be an enumeration of the equivalence class states. Since the equivalence class states are mutually exclusive, it follows that ψ' is satisfiable iff one of $(\psi' \wedge \phi_i)$ is satisfiable. Hence to test satisfiability of ψ', it suffices to be able to test satisfiability of a single $\psi' \wedge \phi_i$ expression.

For the situation where we consider views in $\psi' \wedge \phi_i$ as arbitrary relations, we observe that $\phi_i \Rightarrow \psi'$ is valid if $\phi_i \wedge \psi'$ is satisfiable (see lemma 2 in appendix).

Therefore, given that ϕ_i is satisfiable by some realisable view instance I_v, if $\phi_i \wedge \psi'$ is satisfiable by some arbitrary view instance I'_v (not necessarily realisable), then $\phi_i \wedge \psi'$ is satisfiable by I_v. The converse is obviously true. Thus the satisfiability of $\phi_i \wedge \psi'$ can be tested by considering its truth over all possible small models of ϕ_i. Lemma 1 shows that we can test satisfiability of a ϕ_i by considering all small instances using at most $k = 2^{2\times(m\times p)^m \times m} \times ((m \times p)^m \times m \times m)^{2m+3}$ constants, which is $\mathcal{O}(2^{(m\times p)^m})$.

Overall, to test satisfiability of ψ', we need to search for satisfiable ϕ_i's by testing all possible models of bounded size. This size bound then provides us with the upper bound of $SPACE(2^{(m\times p)^m})$ for the decision procedure.

Hardness: In [3], it is shown that satisfiability is $NTIME(2^{c\times l/\log l})$ hard for first order queries over unary relations, a special case of our language. Here c is some constant and l is the length of the formula. Since the number of relations is $\leq l$, our problem is $NTIME(2^{c\times p/\log p})$ hard. ■

We next sketch a proof for the lemma we have just needed. The key insight is that for a query ψ, there is a bound m on the number of variables a view may have. This allows a complex chasing over "justifications" of constants that only "wraps around" at "distances" $\geq m$. (Given a set of tuples, we view a tuple as one generalised edge, and define distance as the length of shortest paths.)

Lemma 1. *Let ϕ be an equivalence class state formula over N views of maximum definition length m using p relations. If ϕ is satisfiable, then it has a model using at most $k = 2^{2\times(m\times p)^m \times m} \times ((m \times p)^m \times m \times m)^{2m+3}$ constants.*

Proof. (sketch) The proof is quite complex and so we begin with an overview. We first show how any model for ϕ can be described by a structure H called a *justification hierarchy*, which links tuples in the model together according to their distribution in equivalence classes. We then show how to build a new justification hierarchy H', which is a maximal (enlarged) version of H, has some "nicer" properties and is also a model for ϕ. After this, we build another hierarchy H'' which consists of a number of distinct copies of H', each of which is pruned to a certain depth. The constants at the lowest level are justified by the constants at the roots of the copies. H'' can be associated with a new model I'' such that it uses no more than k constants, and I'' is also a model of ϕ. Some of the steps are further explained in the appendix.

Observe that, if the only positive class in ϕ is $\exists X(\neg V_1(X) \wedge ... \wedge \neg V_N(X))$, then ϕ is unsatisfiable since it says (i) that there are no constants in the database and (ii) there must exist a constant. So we can assume that ϕ contains some other positive class.

Justification Hierarchies: Recall that the positive equivalence classes are numbered $C_1, \ldots, C_{n_1}$ and observe that any constant in a model for ϕ appears in exactly one equivalence class. Given an arbitrary constant a_1 in class C_i, there must exist at least one associated *justification set* for the truth of $C_i(a_1)$. A justification set is simply a set of tuples, in the given model, which make $C_i(a_1)$ true. A minimal justification set is a justification set containing no redundant tuples (such a set is not in general unique). We will henceforth regard all justifications as being minimal.

A justification hierarchy for an equivalence class C_i is a hierarchy of justification sets. At the top is a justification set for some constant a in equivalence class C_i. The second level consists of justification sets for the equivalence class of each constant appearing in the justification set for $C_i(a)$. The third level consists of justification sets for the equivalence class of each new constant appearing at level two etc. The hierarchy extends until every constant in the hierarchy has been justified. Figure 1 shows a justification hierarchy for an equivalence class state ψ in the instance $I = \{(1,2),(2,3),(3,4),(4,5),(5,1)\}$. ψ is defined by

$$\begin{aligned}&\exists X(V_1(X) \wedge V_2(X)) \wedge \neg\exists X(V_1(X) \wedge \neg V_2(X))\\ &\wedge\neg\exists X(\neg V_1(X) \wedge V_2(X)) \wedge \neg\exists X(\neg V_1(X) \wedge \neg V_2(X))\end{aligned}$$

where $V_1(X) \leftarrow R(X,Y)$ and $V_2(X) \leftarrow R(Y,X)$. (ψ contains one positive class and three negative ones.)

In general, to describe a model for ϕ, we will need a set of justification hierarchies, one for each positive equivalence class.

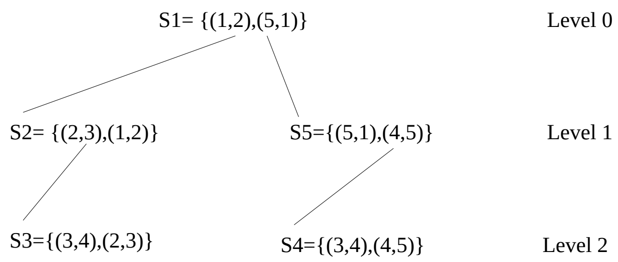

Figure 1: Example justification hierarchy for query ψ

Maximal Justification Hierarchies: Suppose we have a model I, and we have constructed exactly one justification hierarchy for each positive class. Let H

consist of these justification hierarchies. By copying justification sets if necessary, we can assume that the justification hierarchies have a balanced structure. Here "balanced" refers to the notion that a hierarchy can be viewed as tree, whose edges link justification sets.

The next step consists of renaming the constants in the hierarchy to remove 'unnecessary' equalities among the constants in justifications. We build a new justification hierarchy set of the same height. We do this by iterating over all justifications: for each constant a, we replace all constants used in the justification of a using new constants. Call this new hierarchy the *maximised justification hierarchy set* (abbreviated MJHS).

Copying and Pruning Justification Hierarchies: Let I_{max} be a maximised hierarchy set for a model of ϕ. Let n_l be the number of constants at height m. Then
$n_l \leq$ (no. of hierarchies in MJHS) $\times$ ((no. of views) $\times$ (max. length of a view))^m
$\leq 2^N \times (N \times m)^m$.

In this step, we make n_l copies, denoted by $I_1, I_2, \ldots I_{n_l}$, of I_{max}, using a different set of constants for each copy. Let I_{union} be the union of $I_1 \cup I_2 \cup \ldots \cup I_{n_l}$. Then it can be verified that ϕ will still be true for this new instance. Next, all levels below m in the hierarchy are removed and constants at level m are "wrapped-around" by re-justifying them with the root of one of the copies for the same class. Again it can be verified that ϕ will still be true for this new instance.

The number of constants in I_{union} is bounded by $\leq$ num_copies $\times$ (max num. of constants in a justification hierarchy of height $m + 3$). Thus this number is

$\leq n_l \times (2^N \times (N \times m)^{m+3})$
$\leq (2^N \times (N \times m)^m) \times (2^N \times (N \times m)^{m+3})$
$\leq 2^{2N} \times (N \times m)^{2m+3}$
$\leq 2^{2 \times (m \times p)^m \times m} \times ((m \times p)^m \times m \times m)^{2m+3}$, which is $\mathcal{O}(2^{(m \times p)^m})$.

■

Theorem 1 also holds for infinite models, since even if the initial justification hierarchies are infinite, the method used in lemma 1 can still be used to exhibit a finite model. We thus obtain finite controllability (every satisfiable formula is finitely satisfiable).

Proposition 1. *The first order unary conjunctive view language is finitely controllable.* ■

3 Extending the View Definitions

The previous section showed that the first order language using unary conjunctive view definitions is decidable. A natural way to increase the power of the language is to make view bodies more expressive (but retain unary arity for the views). Unfortunately, as we will show, this results in satisfiability becoming undecidable.

The first extension we consider is allowing inequality in the views, e.g.,

$$V(X) \leftarrow R(X,Y), S(X,X), X \neq Y$$

Call the first order language over such views the *first order unary conjunctive*$^{\neq}$ *view language.* This language allows us to check whether a two counter machine computation is valid and terminates. This leads to the following result:

Theorem 2. *Satisfiability is undecidable for the first order unary conjunctive*$^{\neq}$ *view query language.*

Proof. The proof is by a reduction from the halting problem of two counter machines (2CM's) starting with zero in the counters. Given any description of a 2CM and its computation, we can show how to a) encode this description in database relations and b) define queries to check this description. We construct a query which is satisfiable iff the 2CM halts. The basic idea of the simulation is similar to one in [8], but with the major difference that *cycles are allowed* in the successor relation, though there must be at least one good chain. See appendix for further details. ■

The second extension we consider is to allow "safe" negation in the conjunctive views, e.g.

$$V(X) \leftarrow R(X,Y), \neg S(X,X)$$

Call the first order language over such views the *first order unary conjunctive*$^{\neg}$ *view language.* It is also undecidable, by a result in [2].

Theorem 3. *[2] Satisfiability is undecidable for the first order unary conjunctive*$^{\neg}$ *view query language.* ■

A third possibility for increasing the expressiveness of views would be to keep the body as a pure conjunctive query, but allow views to have *binary* arity, *e.g.*

$$V(X,Y) \leftarrow R(X,Y)$$

This doesn't yield a decidable language either, since this language has the same expressiveness as first order logic over binary relations, which is known to be undecidable.

Proposition 2. *Satisfiability is undecidable for the first order binary conjunctive view language.* ■

A fourth possibility is to use unary conjunctive views, but allow recursion in the view bodies, e.g.

$$\begin{aligned} V(X) &\leftarrow edge(X,Y) \\ V(X) &\leftarrow V(X) \wedge edge(Y,X) \end{aligned}$$

Call this the first order unary conjunctive^{rec} language. This causes undecidability also.

Theorem 4. *Satisfiability is undecidable for the first order unary conjunctive^{rec} view language.*

Proof. (sketch): The proof of theorem 2 can be adapted by removing inequality and instead using recursion to ensure there is a connected chain in *succ*. It then becomes more complicated, but the main property needed is that *zero* is connected to *last* via the constants in *succ*. This can be done by

$$\begin{array}{l} conn\_zero(X) \leftarrow zero(X) \\ conn\_zero(X) \leftarrow conn\_zero(Y), succ(Y, X) \\ \exists X(last(X) \wedge conn\_zero(X)) \end{array}$$

■

4 Applications

4.1 Containment

We now briefly examine the application of our results to conjunctive query containment. Theorem 1 implies we can test whether $Q_1(X) \subseteq Q_2(X)$ under the constraints $C_1 \wedge C_2 \ldots \wedge C_n$ where $Q_1, Q_2, C_1, \ldots, C_n$ are all first order unary conjunctive view queries. This just amounts to testing whether the sentence $\exists X(Q_1(X) \wedge \neg Q_2(X)) \wedge C_1 \wedge \ldots \wedge C_n$ is unsatisfiable.

We can similarly test containment and equivalence of monadic datalog programs under first order unary conjunctive view constraints. Monadic datalog is datalog with intensional relations having unary arity. To illustrate the idea, the *conn_zero* view defined above could be written

$$\forall X(conn\_zero(X) \Leftrightarrow (zero(X) \vee tmp(X))$$

where *tmp* is the view

$$tmp(X) \leftarrow conn\_zero(X) \wedge succ(Y, X)$$

Containment of monadic datalog programs was studied in [5], where it was shown to be EXPTIME hard and solvable in 2-EXPTIME using automata theoretic techniques. These complexity results are in line with theorem 1, since the decision problem we are addressing is more general.

Of course, we can also show that testing the containment $Q_1 \subseteq Q_2$ is undecidable if Q_1 and Q_2 are first order unary conjunctive view$^{\neq}$ queries, first order unary conjunctive view$^{\neg}$ queries and first order unary conjunctive^{rec} view queries. containment problem $Q \subseteq \emptyset$.

Containment of queries with negation was first considered in [10]. There it was essentially shown that the problem is decidable for queries which do not apply projection to subexpressions with difference. Such a language is disjoint from ours, since it cannot express a sentence such as $\exists Y V_4(Y) \wedge \neg \exists X(V_1(X) \wedge \neg V_2(X))$ where V_1 and V_2 are views which may contain projections.

4.2 Dependencies

Unary inclusion dependencies were identified as useful in [6]. They take the form $R[X] \subseteq S[Y]$. If we allow R and S above to be unary conjunctive views, we get *unary conjunctive view containment dependencies.* Observe that the unary views are actually unary projections of the join of one or more relations.

We can also define a special type of dependency called a *proper* unary conjunctive inclusion dependency, having the form $Q_1(X) \subset Q_2(X)$. The need for such a dependency could arise in security applications where it is necessary to restrict redundancy. If $\{d_1, \ldots, d_k\}$ is a set of such dependencies, then it is straightforward to test whether they imply another dependency d_x by testing the satisfiability of an appropriate first order unary conjunctive view query.

Theorem 5. *Implication is decidable for unary conjunctive view containment dependencies with subset and proper subset operators.* ■

Similarly, we can consider unary conjunctive$^{\neq}$ containment dependencies. The tests in the proof of theorem 2 for the 2CM can be written in the form $Q_1(X) \subseteq Q_2(X)$, with the exception of the non-emptiness constraints, which must use the proper subset operator.

Theorem 6. *Implication is undecidable for unary conjunctive$^{\neq}$ (or conjunctive$^{\neg}$) view containment dependencies with the subset and the proper subset operators.*
■

4.3 Termination

The languages in this paper have their origins in [2], where active database rule languages based on views were studied. The decidability result for first order unary conjunctive views can be used to positively answer an open question raised in [2], which essentially asked whether termination is decidable for active database rules using unary conjunctive views.

5 Related Work

Satisfiability of first order logic has been thoroughly investigated in the context of the classical decision problem [3]. The main thrust there has been determining for which quantifier prefixes first order languages are decidable. We are not aware of any result of this type which could be used to demonstrate decidability of the unary conjunctive view language. Instead, our result is best classified as a new decidable class generalising the traditional decidable unary first-order language (the Löwenheim class). Use of the Löwenheim class itself for reasoning about schemas is described in [11], where applications towards checking intersection and disjointness of object oriented classes are given.

As observed earlier, a main direction of work concerning views has been in areas such as containment. An area of emerging importance is description logics and coupling them with with horn rules [9, 4]. In [4], the query containment problem is studied in the context of the description logic $\mathcal{DLR}_{reg}$. There are certain similarities between this and the first order (unary) view languages we have studied in this paper. The key difference appears to be that although $\mathcal{DLR}_{reg}$ can be used to define view constraints, these constraints cannot express unary conjunctive views (since assertions do not allow arbitrary projection). Furthermore, $\mathcal{DLR}_{reg}$ can express functional dependencies on a single attribute, a feature which would make our language undecidable (see proof of theorem 2). There is a result in [4], however, showing undecidability for a fragment of $\mathcal{DLR}_{reg}$ with inequality, which could be adapted to give an alternative proof of theorem 2 (though there inequality is used in a slightly more powerful way).

A recent work that deals with complexity of views is [1]. There, results are given for the *view consistency* problem. This involves determining whether there exists an underlying database instance that realises a *specific* (bounded) view instance . The problem we have looked at in this paper is slightly more complicated; testing satisfiability of a first order view query asks the question whether there exists an (*unbounded*) view instance that makes the query true. This explains how satisfiability can be undecidable for first order unary conjunctive$^{\neq}$ queries, but view consistency for non recursive datalog$^{\neq}$ views is in NP.

6 Summary and Further work

Table 1 provides a summary of our decidability results. We can see that they are tight in the sense that the unary conjunctive view language cannot be obviously extended without undecidability resulting. In one sense the picture is negative - we cannot expect to use view bodies with negation, inequality or recursion or allow views to have binary arity if we are aiming for decidability. On the other hand, we feel that the decidable case we have identified, is sufficiently natural and interesting to be of practical use.

Table 1:
Summary of Decidability Results for First Order View Languages

Unary Conjunctive View	Decidable
Unary Conjunctive$^{\neq}$ View	Undecidable
Unary Conjunctive^{rec} View	Undecidable
Unary Conjunctive$^{\neg}$ View	Undecidable [2]
Binary Conjunctive View	Undecidable

As part of future work we intend to investigate relationships to description logics and also look at further ways of introducing negation into the language. One possibility is to allow views of arity zero to specify description logic like constraints, such as $R_1(X,Y) \subseteq R_2(X,Y)$. We would also like to narrow the complexity gap which appears in theorem 1 and gain a deeper understanding of the expressiveness of the view languages we have studied.

Acknowledgements: We wish to thank several people for discussions about the novelty of the problem at the initial stage of this work, including Michael Benedikt, Eric Graedel, and Phokion Kolaitis. Guozhu Dong benefited from discussions with Jianwen Su, and part of his work was done while visiting Simon Fraser University and George Mason University during his study leave.

References

1. S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 254–263, Seattle, Washington, 1998.
2. J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 264–273, Seattle, Washington, 1998.
3. E. Boerger, E. Graedel, and Y. Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1996.
4. D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 149–158, Seattle, Washington, 1998.
5. S. Cosmadakis, H. Gaifman, P. Kanellakis, and M. Vardi. Decidable optimization problems for database logic programs. *Proceedings of the ACM Symposium on the Theory of Computing*, pages 477–490, 1988.
6. S. Cosmadakis, P. Kanellakis, and M. Vardi. Polynomial time implication problems for unary inclusion dependencies. *Journal of the ACM*, 37(1):15–46, 1990.
7. N. Immerman. Upper and lower bounds for first-order expressibility. *Journal of Computer and System Sciences*, 25:76–98, 1982.
8. A. Levy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query reachability, and satisfiability in datalog extensions. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 109–122, Washington D.C., 1993.
9. A. Levy and M. Rousset. The limits on combining recursive horn rules with description logics. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, 1996.
10. Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.
11. D. Theodoratos. Deductive object oriented schemas. In *Proceedings of ER'96, 15th International Conference on Conceptual Modeling*, pages 58–72, 1996.
12. J. D. Ullman. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory, LNCS 1186*, pages 19–40, Delphi, Greece, 1997.

Appendix

Lemma 2. *Suppose we treat the unary view symbols as arbitrary base relations. If $\psi' \wedge \phi_i$ is satisfiable, then $\phi_i \Rightarrow \psi'$ is valid.*

Proof. (Sketch) Suppose there exists an instance I for which $\psi' \wedge \phi_i$ is true. By deleting "extra" constants, we can transform I into a *minimal* model I_{min} such that each positive class contains exactly one constant. Then $\psi' \wedge \phi_i$ is also true in I_{min}. (One can verify this by translating $\psi' \wedge \phi_i$ into relational algebra. Then constants in the same equivalence class always travel together when we evaluate the subexpressions.) Let I' be any instance for which ϕ_i is true. Then I_{min} is isomorphic to a submodel of I', and we can view I' as the result of adding constants to the positive classes of I_{min}. Similar to deleting constants, adding constants doesn't alter the truth value of ψ'. Hence the formula $\phi_i \wedge \neg\psi'$ is unsatisfiable, or equivalently, the formula $\phi_i \Rightarrow \psi'$ is valid. ■

Some Further Descriptions of Procedures Used in Lemma 1

Building a justification hierarchy Each level of the hierarchy contains justification sets for constants in the model for ψ (call this model I). Lower levels justify constants which appear in higher levels.

The base case: Choose a constant (say a_1) in $C_i(I)$. Choose a justification set for a_1 in I (say S_{a_1}). Insert S_{a_1} at level zero.

The recursive case: For each tuple t in the hierarchy at level i containing a constant a_k such that S_{a_k} doesn't appear at any level $\leq i$, create a justification set S_{a_k} at level $i + 1$.

Maximisation Procedure It is possible to define conjunctive query paths through a justification hierarchy between the root node and the leaf nodes. The distance between two constants in the hierarchy is a measure of the number of joins required to link these constants together. To ensure correctness, we would like the distance between a constant at the root node and a constant at level m to be m. Since constants within a justification hierarchy may appear in several justification sets, this property may not be true. The following procedure replaces duplicate constants in the hierarchy with new ones, without compromising the truth of ψ. The distance requirement is then satisfied.

Suppose that each of the hierarchies has height of at least $m + 3$.

/* *Rename* */
For $i = 1$ to $(m + 2)$ do
    Let x be a constant in a justification set S_q at level i such that $x \neq q$.
    If x appears at a level $< i$ then
        choose an entirely new constant k
        For each S_t at level i containing x and $t \neq x$ do
            $S_t = S_t[x \backslash k]$;

S_x at level $i+1 = S_x[x \backslash k]$;
end For
end For

Rename is a function which isomorphically substitutes new constants for all those currently in the hierarchy. Note that if x gets renamed to k, then S_x gets renamed to S_k.

Copying and Pruning Algorithm

Make a number of distinct copies of the hierarchies, as explained in the proof
Call the result of the above step H'
Suppose the constants at height $m+1$ in H' by $a_0, a_1, \ldots, a_k$
For $j = 1$ to k
suppose (i) C_s is the equivalence class of a_j and (ii) b is the constant being justified at root of hierarchy for C_s in $I_{(j+1)\ mod\ k}$
delete S_{a_j} at level $m+1$
S_{a_j} at level $m+1 = S_b[b \backslash a_j]$
End For
remove all levels of hierarchy at height $> m+1$

One can now envisage the model for ψ as a collection of interlinked justification hierarchies. Constants at level m of each hierarchy are justified by the root node of another hierarchy.

Correctness proof for this construction is quite lengthy, and so will be deferred to the full version of the paper.

Proof Details of Theorem 2 A two-counter machine is a deterministic finite state machine with two non-negative counters. The machine can test whether a particular counter is empty or non-empty. The transition function has the form

$$\delta : S \times \{=,>\} \times \{=,>\} \rightarrow S \times \{pop, push\} \times \{pop, push\}$$

For example, the statement $\delta(4,=,>) = (2, push, pop)$ means that if we are in state 4 with counter 1 equal to 0 and counter 2 greater than 0, then go to state 2 and add one to counter 1 and subtract one from counter 2.

The computation of the machine is stored in the relation $config(T, S, C_1, C_2)$, where T is the time, S is the state and C_1 and C_2 are values of the counters. The states of the machine can be described by integers $0, 1 \ldots, h$ where 0 is the initial state and h the halting (accepting) state. The first configuration of the machine is $config(0,0,0,0)$ and thereafter, for each move, the time is increased by one and the state and counter values changed in correspondence with the transition function.

We will use some relations to encode the computation of 2CMs starting with zero in the counters. These are:

- $S_0, \ldots, S_h$: each contains a constant which represents that particular state.

- $succ$: the successor relation. We will make sure it contains one chain starting from $zero$ and ending at $last$ (but it may in addition contain unrelated cycles).
- $config$: contains computation of the 2CM.
- $zero$: contains the first constant in the chain in $succ$. This constant is also used as the number zero.
- $last$: contains the last constant in the chain in $succ$.

Note that we sometimes blur the distinction between unary relations and unary views, since a view V can simulate a unary relation U if it is defined by $V(X) \leftarrow U(X)$.

The unary and nullary views (the latter can be eliminated using quantified unary views) are:

- $halt$: true if the machine halts.
- bad: true if the database doesn't correctly describe the computation of the 2CM.
- $dsucc$: contains all constants in $succ$.
- dT: contains all time stamps in $config$.
- dP: contains all constants in $succ$ with predecessors.
- $dCol_1, dCol_2$: are projections of the first and second columns of $succ$.

When defining the views, we also state some formulas (such as $hasPred$) over the views which will be used to form our first order sentence over the views.

- The "domain" views (those starting with the letter d) are easy to define, e.g.

$$\begin{aligned} dP(X) &\leftarrow succ(Z, X) \\ dCol_1(X) &\leftarrow succ(X, Y) \\ dCol_2(X) &\leftarrow succ(Y, X) \end{aligned}$$

- $hasPred$ says "each nonzero constant in $succ$ has a predecessor:"

$$hasPred : \forall X(dsucc(X) \Rightarrow (zero(X) \vee \ dP(X)))$$

- $sameDom$ says "the constants used in $succ$ and the timestamps in $config$ are the same set":

$$sameDom : \forall X(dsucc(X) \Rightarrow dT(X)) \wedge \forall Y(dT(Y) \Rightarrow dsucc(Y)))$$

- $goodzero$ says "the zero occurs in $succ$":

$$goodzero : \forall x(zero(x) \Rightarrow dsucc(x))$$

- $nempty$: each of the domains and unary base relations is not empty

$$nempty : \exists X(dsucc(X))$$

- Check that each constant in $succ$ has at most one successor and at most one predecessor and that it has no cycles of length 1.

$$bad \leftarrow succ(X,Y), succ(X,Z), Y \neq Z$$
$$bad \leftarrow succ(Y,X), succ(Z,X), Y \neq Z$$
$$bad \leftarrow succ(X,X)$$

Note that the first two of these rules could be enforced by the functional dependencies $X \rightarrow Y$ and $Y \rightarrow X$ on *succ*.

- Check that every constant in the chain in succ which isn't the last one must have a successor

$$hassuccnext: \forall Y(dCol_2(Y) \Rightarrow (last(Y) \vee dCol_1(Y))$$

- Check that the last constant has no successor and zero (the first constant) has no predecessor.

$$bad \leftarrow last(X), succ(X,A)$$
$$bad \leftarrow zero(X), succ(A,X)$$

- Check that every constant eligible to be in last and zero must be so.

$$eligiblezero: \forall Y(dCol_1(Y) \Rightarrow (dCol_2(Y) \vee zero(Y))$$
$$eligiblelast: \forall Y(dCol_2(Y) \Rightarrow (dCol_1(Y) \vee last(Y)))$$

- Each S_i and $zero$ and $last$ contain ≤ 1 element.

$$bad \leftarrow S_i(X), S_i(Y), X \neq Y$$
$$bad \leftarrow zero(X), zero(Y), X \neq Y$$
$$bad \leftarrow last(X), last(Y), X \neq Y$$

- Check that $S_i, S_j, last, zero$ are disjoint $(0 \leq i < j \leq h)$:

$$bad \leftarrow zero(X), last(X)$$
$$bad \leftarrow S_i(X), S_j(X)$$
$$bad \leftarrow zero(X), S_i(X)$$
$$bad \leftarrow last(X), S_i(X)$$

- Check that the timestamp is the key for $config$. There are three rules, one for the state and two for the two counters; the one for the state is:

$$bad \leftarrow config(T,S,C_1,C_2), config(T,S',C_1',C_2'), S \neq S'$$

- Check the configuration of the 2CM at time zero. $config$ must have a tuple at $(0,0,0,0)$ and there must not be any tuples in config with a zero state and non zero times or counters.

$$V_{z_s}(S) \leftarrow zero(T), config(T,S,\_,\_)$$
$$V_{z_{c_1}}(C) \leftarrow zero(T), config(T,\_,C,\_)$$
$$V_{z_{c_2}}(C) \leftarrow zero(T), config(T,\_,\_,C)$$
$$V_{y_s}(T) \leftarrow zero(S), config(T,S,\_,\_)$$
$$V_{y_{c_1}}(C_1) \leftarrow zero(S), config(T,S,C_1,\_)$$
$$V_{y_{c_2}}(C_2) \leftarrow zero(S), config(T,S,\_,C_2)$$
$$goodconfigzero: \forall X(V_{z_s}(X) \Rightarrow S_0(X) \wedge$$
$$(V_{z_{c_1}}(X) \vee V_{z_{c_2}}(X) \vee V_{y_s}(X) \vee V_{y_{c_1}}(X) \vee V_{y_{c_2}}(X)) \Rightarrow zero(X))$$

- For each tuple in $config$ at time T which isn't the halt state, there must also be a tuple at time $T+1$ in $config$.

$$\begin{array}{l} V_1(T) \leftarrow config(T,S,C_1,C_2), S_h(S) \\ V_2(T) \leftarrow succ(T,T2), config(T2,S',C_1',C_2') \\ hasconfignext : \forall T((dt(T) \wedge \neg V_1(T)) \Rightarrow V_2(T)) \end{array}$$

- Check that the transitions of the 2CM are followed. For each transition $\delta(j,>,=) = (k, pop, push)$, we include three rules, one for checking the state, one for checking the first counter and one for checking the second counter. For the transition in question we have for the state

$$\begin{array}{l} V_\delta(T') \leftarrow config(T,S,C_1,C_2), succ(T,T'), S_j(S), succ(X,C_1), zero(C_2) \\ V_{\delta_s}(S) \leftarrow V_\delta(T), config(T,S,C_1,C_2) \\ goodstate_\delta : \forall S(V_{\delta_s}(S) \Leftrightarrow S_k(S)) \end{array}$$

and for the first counter, we (i) find all the times where the transition is definitely correct for the first counter

$$\begin{array}{ll} Q_{1_\delta}(T') \leftarrow & config(T,S,C_1,C_2), \\ & succ(T,T'), S_j(S), succ(X,C_1), \\ & zero(C_2), succ(C_1'',C_1), config(T',S',C_1'',C_2') \end{array}$$

(ii) find all the times where the transition may or may not be correct for the first counter

$$Q_{2_\delta}(T') \leftarrow config(T,S,C_1,C_2), succ(T,T'), S_j(S), succ(X,C_1), zero(C_2)$$

and make sure Q_{1_δ} and Q_{2_δ} are the same

$$goodtrans_{\delta_{c_1}} : \forall T(Q_{1_\delta}(T) \Leftrightarrow Q_{2_\delta}(T))$$

Rules for second counter are similar.
For transitions $\delta_1, \delta_2, \ldots, \delta_k$, the combination can be expressed thus:

$$\begin{array}{l} goodstate : goodstate_{\delta_1} \wedge goodstate_{\delta_2} \wedge \ldots \wedge goodstate_{\delta_k} \\ goodtrans_{c_1} : goodtrans_{\delta_{1_{c_1}}} \wedge goodtrans_{\delta_{2_{c_1}}} \wedge \ldots \wedge goodtrans_{\delta_{k_{c_1}}} \\ goodtrans_{c_2} : goodtrans_{\delta_{1_{c_2}}} \wedge goodtrans_{\delta_{2_{c_2}}} \wedge \ldots \wedge goodtrans_{\delta_{k_{c_2}}} \end{array}$$

- Check that halting state is in $config$.

$$\begin{array}{l} hlt(T) \leftarrow config(T,S,C_1,C_2), S_h(S) \\ halt : \exists X\, hlt(X) \end{array}$$

Given these views, we claim that satisfiability is undecidable for the query $\psi = \neg bad \wedge hasPred \wedge sameDom \wedge halt \wedge goodzero \wedge goodconfigzero \wedge \wedge nempty \wedge hassuccnext \wedge eligiblezero \wedge eligiblelast \wedge goodstate \wedge goodtrans_{c_1} \wedge goodtrans_{c_2} \wedge hasconfignext$ ■