Linearly Bounded Reformulations of Unary Databases

Rada Chirkova and Michael R. Genesereth,
Stanford University, Stanford CA 94305, USA

Stanford University, Stanford CA 94305, USA
{rada,genesereth}@cs.stanford.edu

Abstract. *Database reformulation* is the process of rewriting the data and rules of a deductive database in a functionally equivalent manner. We focus on the problem of automatically reformulating a database in a way that reduces query processing time while satisfying strong storage space constraints.
In this paper we consider one class of deductive databases — those where all stored relations are unary. For this class of so-called *unary databases*, we show that the database reformulation problem is decidable if all rules can be expressed in nonrecursive datalog with negation; moreover, we show that for such databases there always exists an "optimal" reformulation. We also suggest how this solution for unary databases might be extended to the general case, i.e., to that of reformulating databases with stored relations of arbitrary arity.

1 Introduction

Abstraction and reformulation techniques have been used successfully in a number of domains to reduce the complexity of the problems to solve. We present an application of abstraction and reformulation in the database domain, to the problem of reducing query processing time. While this problem is formulated in the database context, it is easy to generalize, since broad classes of problems can be viewed and solved as database problems.

A database system undergoes a number of transformations during its lifetime. Database schema and/or rule transformations are central to database design, data model translation, schema (de)composition, view materialization, and multidatabase integration. Interestingly, nearly all these tasks can be regarded as aspects of the same problem in a theoretical framework that we proceed to describe.

Consider an abstract database transformation problem. Suppose the input to the problem comprises the schema and rules of a deductive database and a set of elementary queries which, together with some algebra, form a query language on the database. Suppose the objective of database transformation is to build an "optimal" structure of the database with respect to the requirements and constraints that are also provided in the input.

Generally, the transformations of the database schema and rules need to be performed in such a way that the resulting database satisfies three conditions. First, it should be possible to extract from the transformed database, by means of the input query language, exactly the same information as from the original database. Second, the result should satisfy the input requirements, such as minimizing query processing costs. Finally, the result should satisfy the input constraints; one common constraint is a guarantee of a (low) upper bound on the disk space for storing the transformed database. Notice that all three conditions must hold for all instances of the input database.

We call this problem *database reformulation* and consider logic-based approaches to its solution. Database reformulation is the process of rewriting the data and rules of a deductive database in a functionally equivalent manner. By specifying various input requirements and constraints, the database reformulation problem translates into any of the database schema/query transformation problems mentioned above.

We focus on database reformulations whose input requirement is to minimize the computational costs of processing the given queries, under strong storage space constraints that guarantee no more than linear increase in database size. In this formulation, the database reformulation framework is most suitable for dealing with the problems of view materialization and multidatabase integration.

In this paper we give a definition and a formal specification of the database reformulation problem. We then present the main contribution of this paper, a complete solution of the database reformulation problem

for one class of databases. In this class of so-called *unary databases*, all stored relations are unary, i.e., have one attribute each; in addition, all rules can be expressed without recursion or built-in predicates.

There are a number of important applications where unary databases occur naturally. Unary databases come to mind whenever there is a need to single out and process features of objects. One example is indexing in libraries: books and articles are routinely classified by subject, and it is common for one item to belong to more than one class. Possible classes can be represented as unary relations with relevant books represented by tuples in the relations. For example, an article on statistical profile estimation in database systems can belong to classes "physical design", "languages", and "systems" at the same time.

Unary databases are also useful for taxonomic search in e-commerce; there, some of the more frequent queries are unions and intersections of classes in several taxonomies. For example, one might want to find all products which satisfy at least one of the stipulated properties (union of classes), or those products each of which satisfies all of the stipulated properties (intersection of classes).

After describing our solution to the database reformulation problem for unary deductive databases, we suggest how this solution might be extended to the general case, i.e., to the problem of reformulating databases with stored relations of arbitrary arity.

In this paper, proofs of the results presented in the text can be found in the appendix.

2 Preliminaries and Terminology

Our representation of the domain includes a set of *relations*; the set of attribute names for a relation is called a *relation schema*. A relation is called *unary* if it has exactly one attribute.

A relation is referred to as *stored* if it is physically recorded, as a table (a set of tuples, each tuple having a value for each attribute of the relation), on some storage media; a collection of stored relations is called a (regular) *database*. A *database schema*, for a given database D, is a collection of relation schemas for all stored relations in D. See [28] for more details.

A *nonrecursive datalog*$^{\neg}$ (*nr-datalog*$^{\neg}$ [2]) rule is an expression of the form

$$p(\bar{X}) \; :- \; l_1(\bar{Y}), \; \dots \; , \; l_n(\bar{Z}), \tag{1}$$

where p is a relation name, $\bar{X}$, $\bar{Y}$, ... , $\bar{Z}$ are tuples of variables and constants, and each l_i is a literal, i.e., an expression of the form p_i or $\neg p_i$ (by $\neg$ we denote negation), where p_i is a relation name. $p(\bar{X})$ is called the *head* of the rule, and its *body* is a conjunction of *subgoals* $l_1(\bar{Y})$, ... , $l_n(\bar{Z})$. A rule is called *safe* if each variable in the rule occurs in a non-negated subgoal in the rule's body.

A *query (view)* is a set of rules (in *nr-datalog*$^{\neg}$, for our purposes) with one distinguished relation name in the head of some rule(s). A *query relation* is the distinguished relation of the query, computed from the query using bottom-up logic evaluation, formalized, for example, in Algorithm 3.6 in [28]; a *view relation* is defined analogously. A query (view) is *materialized* if the query (view) relation is precomputed and stored in the database.

Two queries (views) are called *equivalent* if their relations are the same in any database. Given a query q, a query q' is called a *rewriting* of q in terms of a set $\mathcal{V}$ of relations if q and q' are equivalent and q' contains only literals of $\mathcal{V}$.

A *deductive database* (see, for example, [22]) is a (regular) database as defined above, together with a set of queries and views defined on (the stored relations of) the database. A deductive database is called *unary* if all its stored relations are unary. In this paper we consider unary deductive databases where all queries and views are defined in safe *nr-datalog*$^{\neg}$. Since, as shown in [19], any recursive program with safe negation and unary stored relations is nonrecursive, all our results also apply to this more general case.

3 Example of a Unary Database

Let us consider an abstract example that involves a unary database. Suppose that an application queries a database with three unary stored relations, r, s, and t; see Table 1 for a concrete example. Suppose there are three important queries in that application, defined as follows:

$$q_1(X) :- r(X),\ s(X),\ \neg t(X); \quad (2)$$
$$q_2(X) :- s(X),\ \neg t(X); \quad (3)$$
$$q_3(X) :- t(X),\ \neg r(X); \quad (4)$$

see Table 2 for the resulting relations.

Table 1. Stored relations r, s, and t.

r
a
b
c
d

s
a
b
c
e
f

t
c
d
f
g

Also suppose that in this application, all queries of interest can be expressed in terms of the three queries above. For example, one might pose to the database the following query q_4:

$$q_4(X,\ Y) :- r(X),\ s(X),\ \neg t(X),\ t(Y),\ \neg r(Y). \quad (5)$$

Notice that q_4 is simply a cross-product of queries q_1 and q_3, i.e., a set of combinations of each answer to query q_1 with each answer to q_3.

Table 2. Query relations q_1, q_2, and q_3.

q_1
a
b

q_2
a
b
e

q_3
f
g

A straightforward solution to the database reformulation problem in this case would be to materialize queries q_1 through q_3. This solution would certainly reduce the query processing times for these queries, and consequently for all queries in the application. However, it would also materialize in the database duplicate copies of the same objects — those that belong to both r and s but not to t (objects a and b in our example), since answers to both q_1 and q_2 include such objects. If the number of such duplicate objects in the database is considerable, the resulting storage space overhead is a cause of concern. Our solution to the database reformulation problem for unary applications like this one guarantees good query execution time while avoiding the overhead suggested in the example.

4 Database Reformulation

We study a class of database applications where all queries of interest can be expressed in terms of some predefined set of elementary queries; this elementary set can be viewed as an alphabet which defines a query language. We would like to make "good" decisions on which views to materialize, in order to minimize query processing costs for this elementary set of queries (and, consequently, for all expected queries) and to satisfy some (for example, storage space) constraints on the resulting database.

Database reformulation is the process of rewriting the data and rules of a deductive database in a functionally equivalent manner. Our cost model for query execution is the classical bottom-up logic evaluation model; see Algorithm 3.6 in [28].

We describe the input and the output of the database reformulation process. Consider a set $\mathcal{P}$ of relation names. Let $\mathcal{S}$ be a database schema that consists of relation schemas for some relation names in $\mathcal{P}$; $\mathcal{S}$ is the set of schemas for all *stored* relations in the input. Let $\mathcal{R}_{\mathcal{S}}$ be a set of definitions, in terms of $\mathcal{S}$, for some relations whose names are in $\mathcal{P}$; $\mathcal{R}_{\mathcal{S}}$ is the set of *views* in the input. Let $\mathcal{Q}$ be a set of names of all elementary *query* relations of interest, such that $\mathcal{Q} \subseteq \mathcal{P}$ and that $\mathcal{R}_{\mathcal{S}}$ contains definitions of all relations in $\mathcal{Q}$.

Now let $\mathcal{V}$ be a database schema which consists of schemas for some relation names in $\mathcal{P}$; $\mathcal{V}$ describes new stored relations which are materialized in the process of database reformulation. Finally, let $\mathcal{R}_{\mathcal{V}}$ be a set of views defined in terms of $\mathcal{V}$.

Definition 1. *For a given triple $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$, a triple $(\mathcal{V}, \mathcal{R}_{\mathcal{V}}, \mathcal{Q})$ is a reformulation of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ if for each query relation in $\mathcal{Q}$ with a definition $q_{\mathcal{S}}$ in $\mathcal{R}_{\mathcal{S}}$, $\mathcal{R}_{\mathcal{V}}$ contains a rewriting of $q_{\mathcal{S}}$.*

As has already been mentioned, we focus on the problem of database reformulation under strong storage space constraints. Other constraints may be included as well; all constraints relevant to the application in question are considered part of the reformulation input. Let us describe the storage space constraints we focus on in this paper. Suppose D is an arbitrary database with the schema $\mathcal{S}$; let D' be a database that consists of the tables for all and only those (materialized, starting from D) view relations in $\mathcal{V}$ that are used in defining the query relations in $\mathcal{Q}$. For a fixed database schema $\mathcal{S}$ and a fixed set of views that define relations in $\mathcal{V}$ in terms of $\mathcal{S}$, consider all possible databases D and all corresponding databases D', with sizes (in bytes) $|D|$ and $|D'|$ respectively.

Definition 2. *A reformulation $(\mathcal{V}, \mathcal{R}_{\mathcal{V}}, \mathcal{Q})$ of an input $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ satisfies the no-growth storage space constraint if for all pairs $(D,\ D')$, the storage space $|D'|$ taken up by D' does not exceed $|D|$:*

$$|D'| \ \leq \ |D|. \tag{6}$$

A reformulation $(\mathcal{V}, \mathcal{R}_{\mathcal{V}}, \mathcal{Q})$ of a given input $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ is called a *candidate reformulation* if it satisfies the constraints specified in its input. A reformulation output is called *worthwhile* if, in that reformulation, at least one elementary query in $\mathcal{Q}$ is executed faster than in the input formulation, for all database instances. In this paper we focus on candidate worthwhile reformulations of unary databases under the no-growth storage space constraint.

5 The Orthogonal Basis of a Unary Database Schema

Our ultimate objective in solving the database reformulation problem is to automate the reformulation process in as general a setting as possible; in other words, we would like to come up with some *reformulation algorithm*. We try to answer the question of whether the potentially infinite, for each input, search space of reformulations can be transformed in such a way that it would become finite but would still contain valuable reformulations.

One way of making the search space of reformulations more tractable is to restrict the number of view relations that are used to rewrite the input queries. Suppose we could show that, for unary databases, the set of view relations that can define any "good" reformulation, is finite, and that all and only these view relations can be defined in a particular format. Then the problem of finding "good" reformulations of arbitrary unary databases would be reduced to the clearly feasible problem of enumerating and combining all views defined in this particular format, thereby giving us a nice *enumeration algorithm*.

In this section we substantiate this hypothesis by showing that for an arbitrary unary input there exists a "good" reformulation with certain desirable properties and such that its materialized views are defined in a particular format.

Let us analyze the definition of query q_1 given in equation 2 in Section 3. The body of the definition is a conjunction of subgoals with the same variable; notice that each of the stored relations r, s, t yields exactly one subgoal in the definition. Let us build a pattern based on this observation. For a unary database with n stored relations $s_1,\ s_2,\ \ldots\ ,\ s_n$, the pattern looks as follows:

$$l_1(X),\ l_2(X),\ \ldots\ ,\ l_n(X); \tag{7}$$

here, $l_i(X)$ is either $s_i(X)$ or $\neg s_i(X)$.

In our example, the body of query q_1 is an instance of the pattern. We will show below that arbitrary unary queries, when defined on unary databases, can be rewritten as unions of such patterns. For instance, q_3 in our running example (equation 4 in Section 3) can be rewritten as a union of two patterns:

$$q_3(X)\ :-\ t(X),\ \neg r(X),\ \neg s(X) \bigcup t(X),\ \neg r(X),\ s(X). \tag{8}$$

For an arbitrary unary database schema one can define a set of relations as (nearly) all possible instances of the pattern described in equation 7. The only exception is the instance where all subgoals are negated, since we only consider safe rules.

It is easy to show that a set $\mathcal{B}$ of relations defined in such a manner on a unary database schema $\mathcal{S}$ always exists and is unique, up to reorderings of subgoals in rules and to variable renamings. Notice that, if $\mathcal{S}$ has n elements, then there are $2^n - 1$ relations in the set $\mathcal{B}$ for $\mathcal{S}$. Another property of the set $\mathcal{B}$ is that, for any instance D of a database with schema $\mathcal{S}$, each object in the universe of discourse of D belongs to exactly one relation in $\mathcal{B}$; for this reason, we call the set $\mathcal{B}$ the *orthogonal basis* of the unary database schema $\mathcal{S}$.

Definition 3. *The orthogonal basis of a unary database schema* $\mathcal{S} = \{ s_1, s_2, \ldots, s_n \}$ *is the set* $\mathcal{B}$ *of (nearly) all possible relations defined as*

$$b_i(X) \ :- \ l_1(X), \ l_2(X), \ \ldots, \ l_n(X), \tag{9}$$

where each $l_j(X)$ *is either* $s_j(X)$ *or* $\neg s_j(X)$*; the only such combination which is not in* $\mathcal{B}$ *is that where all subgoals are negated.*

Notice that this definition effectively provides an algorithm to construct the orthogonal basis of a unary database schema.

We observe the following property of unary relations.

Theorem 1. *Any unary relation that can be defined in nr-datalog*$^\neg$ *on a unary schema* $\mathcal{S}$ *can be rewritten as a union of relations in the orthogonal basis* $\mathcal{B}$ *of the schema* $\mathcal{S}$*.*

An important result is an immediate corollary of Theorem 1. Let r be a rule in *nr-datalog*$^\neg$ which defines an arbitrary (not necessarily unary) query relation on a unary database schema $\mathcal{S}$. Then:

Corollary 1. *There exists a unique, up to reordering of subgoals and variable renamings, rewriting of* r *in terms of the orthogonal basis* $\mathcal{B}$ *of* $\mathcal{S}$*.*

Let us build the orthogonal basis and rewrite all the queries in our running example from Section 3.

Example 1. The unary database schema is $\mathcal{S} = \{ r, \ s, \ t \}$. The three query relations q_1 through q_3 constitute the set $\mathcal{Q}$; their definitions in equations 2 - 4 constitute the set $\mathcal{R}_\mathcal{S}$.

The orthogonal basis $\mathcal{B}$ of the schema $\mathcal{S}$ consists of seven $(2^3 - 1)$ relations with the following definitions:

$$b_1(X) \ :- \ \neg r(X), \ \neg s(X), \ t(X); \tag{10}$$

$$b_2(X) \ :- \ \neg r(X), \ s(X), \ \neg t(X); \tag{11}$$

$$\ldots$$

$$b_7(X) \ :- \ r(X), \ s(X), \ t(X); \tag{12}$$

and queries q_1 through q_3 can be rewritten in terms of the elements of $\mathcal{B}$ as:

$$q_1(X) \ :- \ b_6(X); \tag{13}$$

$$q_2(X) \ :- \ b_2(X) \bigcup b_6(X); \tag{14}$$

$$q_3(X) \ :- \ b_1(X) \bigcup b_3(X). \tag{15}$$

Now the query q_4, which is a cross-product of queries q_1 and q_3, can be rewritten as the following disjunction of two rules:

$$q_4(X, \ Y) \ :- \ b_6(X), \ b_1(Y); \tag{16}$$

$$q_4(X, \ Y) \ :- \ b_6(X), \ b_3(Y). \tag{17}$$

□

Let $\mathcal{B}$ be the orthogonal basis of a unary database schema $\mathcal{S}$, and let $\mathcal{R}_\mathcal{B}$ be the set of rewritings of all rules in $\mathcal{R}_\mathcal{S}$ in terms of the elements of the set $\mathcal{B}$.

Definition 4. *The triple* $(\mathcal{B}, \mathcal{R}_\mathcal{B}, \mathcal{Q})$ *is called the orthogonal basis reformulation of the triple* $(\mathcal{S}, \mathcal{R}_\mathcal{S}, \mathcal{Q})$*.*

Notice that Definition 3 and the proofs of Theorem 1 and of Corollary 1 effectively provide an algorithm for constructing the orthogonal basis reformulation of an arbitrary unary input.

It is easy to show that for any unary database schema, its orthogonal basis reformulation exists and is unique. To formulate another property of the orthogonal basis reformulation, we will need this definition.

Definition 5. *A database satisfies the minimal-space constraint if each object in the universe of discourse (UOD) of the database is only stored once.*

In other words, the minimal-space constraint requires a database to "fit into" the minimal space needed to store all the information about the database. Notice that if a database satisifes the minimal-space constraint then it also satisfies the no-growth storage space constraint.

Theorem 2 (Properties of the Orthogonal Basis). *For the orthogonal basis reformulation* $(\mathcal{B}, \mathcal{R}_\mathcal{B}, \mathcal{Q})$ *of a triple* $(\mathcal{S}, \mathcal{R}_\mathcal{S}, \mathcal{Q})$*, where* $\mathcal{S}$ *is unary, the following properties hold:*

1. *The only operations in all rules in* $\mathcal{R}_\mathcal{B}$ *are union and cross-product: there are no intersections or negations.*
2. $(\mathcal{B}, \mathcal{R}_\mathcal{B}, \mathcal{Q})$ *satisfies the minimal-space constraint.*
3. *Maintenance costs in the reformulated database, provided certain simple index structures are in place, are linear in the size of the schema* $\mathcal{S}$*, i.e., in the number of the original stored relations.*

Notice the low cost of updates in the reformulated database.

Not surprisingly, these nice properties come at a price: since the number of relations in the orthogonal basis is exponential in the size of the original database schema $\mathcal{S}$, according to our cost model the time to answer the queries in $\mathcal{Q}$ will probably increase in the orthogonal basis reformulation, relative to that in database instances with the schema $\mathcal{S}$. However, the increase is not too high because, even though the number of stored relations in the reformulated database is exponential in the number of the original stored relations, the size of the actual data (stored tuples) does not change after the reformulation. Thus, queries and updates on the reformulated database can be made faster by using certain simple index structures.

6 Enumerating Candidate Relations

From the previous section we know how to obtain one interesting reformulation of the given input. Is it possible, in the unary case, to generate all interesting reformulations, i.e., those that have the same nice properties as the orthogonal basis reformulation? It turns out that the answer is yes: in this section, we show how to finitely enumerate all worthwhile candidate (see definitions in the last paragraph of Section 4) reformulations of an arbitrary unary reformulation input.

Consider a unary database schema $\mathcal{S}$. Let r be an arbitrary relation defined in *nr-datalog*$^\neg$ on $\mathcal{S}$, and let D be an arbitrary database instance with schema $\mathcal{S}$. Consider the space $|D|$ required to store D and the space $|r|$ required to store r when it is materialized; both $|D|$ and $|r|$ are in bytes.

Theorem 3. *In all databases* D *with schema* $\mathcal{S}$*,* $|r|$ *does not exceed* $|D|$*:*

$$\forall\, D: \quad |r| \;\leq\; |D|, \tag{18}$$

if and only if r *is a unary relation.*

This result has one important consequence: it means that if we want to obtain candidate reformulations, i.e., those that satisfy a strong storage space constraint (see Definition 5 in Section 5), the only relations we can choose as stored (materialized) in reformulated databases are unary relations.

Using Theorem 1, we have designed a *unary enumeration algorithm* whose input is a unary database schema $\mathcal{S}$ and whose output is a set $\mathcal{W}$ of relations defined on $\mathcal{S}$.

Algorithm 1 (Unary Enumeration). *First build the orthogonal basis $\mathcal{B}$ of $\mathcal{S}$, then output all unions of the elements of $\mathcal{B}$.*

By Theorems 1 and 3, this algorithm generates the definitions of all and only those relations that can be defined in terms of the schema $\mathcal{S}$, and, at the same time, can fit in the storage space of the original database for all database instances with schema $\mathcal{S}$. Thus, the following holds.

Theorem 4. *For a given reformulation input $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ where $\mathcal{S}$ is unary, the unary enumeration algorithm 1 generates all views that could possibly be used to rewrite the definitions in $\mathcal{R}_{\mathcal{S}}$ and, at the same time, fit in the storage space of the original database for all databases with schema $\mathcal{S}$.*

In what follows, we will consider as candidate reformulations only those reformulations that satisfy the minimal-space constraint. Notice that under this requirement, the orthogonal basis reformulation is a candidate reformulation.

Using this notion of candidacy, we propose the following algorithm for reformulating unary deductive databases. Let $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$, where $\mathcal{S}$ is unary, be an input to the database reformulation problem. Let $\mathcal{W}$ be the set of relations output by Algorithm 1. Algorithm 2 described below outputs reformulations of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$.

Algorithm 2 (Enumeration of Candidate Reformulations). *Output all triples $(\mathcal{V}, \mathcal{R}_{\mathcal{V}}, \mathcal{Q})$ where $\mathcal{V}$ is a subset of $\mathcal{W}$ and $\mathcal{R}_{\mathcal{V}}$ is a set of rewritings of the rules in $\mathcal{R}_{\mathcal{S}}$ in terms of $\mathcal{V}$, provided such rewritings exist for all relations defined in $\mathcal{R}_{\mathcal{S}}$.*

The following result is an easy observation on Algorithm 2:

Theorem 5. *For an arbitrary reformulation input $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ where $\mathcal{S}$ is unary, Algorithm 2 generates all its possible candidate reformulations.*

7 The Minimal Non-Forking Reformulation

In the previous section we have described an algorithm that generates all candidate reformulations of a given unary input; the problem with the algorithm is that it may generate many non-candidate reformulations as well and, in general, the search space for finding candidate reformulations is too large. Fortunately, it turns out that one does not even need to generate and compare all "potentially good" reformulations of the given input by using this algorithm. Instead of applying the storage space criterion to each output of Algorithm 2, one can reduce the search space in advance by using the same storage space constraint.

For the ease of exposition, we will need the following notion: for a query r defined on a unary database, a *unary subquery* of r for some variable X (for some constant c) is the conjunction, in some rule for r, of all subgoals of r with that variable X (constant c). Notice that each unary subquery of an arbitrary query is a definition of a unary relation.

For a unary reformulation input $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$, consider a bipartite graph $\mathcal{G} = (\mathcal{U}, \mathcal{B}, E)$ where $\mathcal{U}$ and $\mathcal{B}$ are two sets of vertices and E is the set of edges, $E \subseteq \mathcal{U} \times \mathcal{B}$. The graph is constructed as follows: $\mathcal{U}$ is the set of relation names for all unary subqueries of all input queries in $\mathcal{Q}$; $\mathcal{B}$ is the set of names of all relations in the orthogonal basis of $\mathcal{S}$; E contains an edge (u, b) iff the definition of the unary query denoted by u, as a union of basis relations, includes the relation denoted by b. We call this graph the *reformulation graph* of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$.

Example 2. Consider our running example from Section 3; Example 1 in Section 5 shows the orthogonal basis reformulation for that example.

Let us build the reformulation graph $\mathcal{G}$ of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ from Section 3.

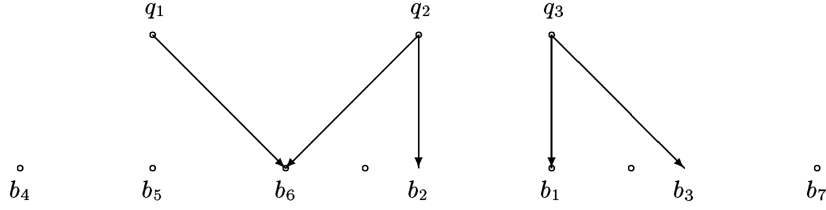

Fig. 1. The reformulation graph $\mathcal{G}$ in Example 2.

1. The set $\mathcal{U}$ of the graph consists of three vertices, one for each elementary unary query (q_1 through q_3).
2. The set $\mathcal{B}$ represents all relations in the orthogonal basis $\mathcal{B}$ of the set $\mathcal{S}$.
3. The set E contains edges $(q_1,\ b_6)$, $(q_2,\ b_6)$, $(q_2,\ b_2)$, $(q_3,\ b_1)$, and $(q_3,\ b_3)$.

The resulting graph is shown in Figure 1; here we see a depiction of the three unary subqueries of queries q_1 through q_3, redefined as unions of basis relations; for example, the only unary subquery of q_2 is a union of two basis relations b_2 and b_6, and so on. □

Reformulation graphs, built as illustrated in Example 2, suggest a method for building "good" reformulations of unary databases: the idea is to materialize all maximal unions of basis relations whose elements are used to define no more than one unary subquery. For instance, in Example 2 we would materialize three relations: b_2, b_6, and the union of b_1 and b_3. Materializing such relations would optimize query processing costs by minimizing the time required to compute the unary subqueries, under the constraint that none of the objects in the UOD of the database is stored twice. This idea is embodied in Algorithm 3, which takes as input a triple $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$, where $\mathcal{S}$ is unary, and outputs a reformulation $(\mathcal{M}, \mathcal{R}_{\mathcal{M}}, \mathcal{Q})$ of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$.

Algorithm 3 (Minimal Non-Forking Reformulation).

1. *Construct the bipartite graph $\mathcal{G}$ of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$; $\mathcal{G} = (\mathcal{U},\ \mathcal{B},\ E)$.*
2. *Classification of the vertices in $\mathcal{B}$: for each vertex $b \subseteq \mathcal{B}$, place b into the set N (nonforking) if exactly one edge in $\mathcal{G}$ is incident on b, and place b into the set F (forking) if more than one edge in $\mathcal{G}$ is incident on b.*
3. *Transform $\mathcal{G}$ into $\mathcal{G}'$ by removing from $\mathcal{B}$ all vertices which are neither in N nor in F, i.e., those that are not incident on any edge in $\mathcal{G}$.*
4. *View materialization I: materialize separately each relation b in F.*
5. *View materialization II: transform the graph $\mathcal{G}'$ into $\mathcal{G}''$ by removing all vertices in F and all edges incident on these vertices, then materialize all unions of relations b such that the corresponding vertices in $\mathcal{B}$ belong to a connected subgraph of $\mathcal{G}''$.*
6. *Construct a set of rules $\mathcal{R}_{\mathcal{M}}$ by rewritings all queries in $\mathcal{R}_{\mathcal{S}}$ in terms of the relations $\mathcal{M}$ materialized in steps 4 and 5.*

In Example 2, $N = \{\ b_1,\ b_2,\ b_3\ \}$, $F = \{\ b_6\ \}$, the vertices discarded in step 3 are $b_4,\ b_5,\ b_7$; view materialization I materializes b_6, and view materialization II materializes relations b_2 and $b_1 \bigcup b_3$. Notice that since the stored relations in $(\mathcal{M}, \mathcal{R}_{\mathcal{M}}, \mathcal{Q})$ are parts of unary subqueries of relations in $\mathcal{Q}$, step 6 of the algorithm, i.e., rewriting the query relations in terms of $\mathcal{M}$, is straightforward.

Definition 6. *The output $(\mathcal{M}, \mathcal{R}_{\mathcal{M}}, \mathcal{Q})$ of Algorithm 3 is called a minimal non-forking reformulation of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$.*

The name *non-forking* comes from the method of building the materialized relations: in the bipartite graph $\mathcal{G}$ for our running example, in Figure 1 we can see a *fork* (more than one edge) at the basis relation b_6, which means that b_6 is used in the definition of more than one unary subquery and, for this reason, needs to be materialized as a separate relation.

It is easy to show that for any unary reformulation input, the minimal non-forking reformulation exists and is unique; moreover, by construction it is always a candidate reformulation of the input.

Now let us recall that the objective of database reformulation is to minimize query processing costs by materializing views. The most important result of this paper is that any input query is answered in the minimal non-forking reformulation at least as fast as in any candidate reformulation:

Theorem 6. *In the minimal non-forking reformulation* $(\mathcal{M}, \mathcal{R}_{\mathcal{M}}, \mathcal{Q})$ *of a reformulation input* $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ *where* $\mathcal{S}$ *is unary, any query is answered at least as fast (for all database instances) as in any candidate reformulation of* $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$.

Notice that, depending on whether the input database itself satisfies the minimal-space constraint, the minimal non-forking reformulation may or may not process the queries faster than the input database. In any case, Theorem 6 reduces the search space of reformulations to just two formulations: the input formulation $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ and the minimal non-forking formulation $(\mathcal{M}, \mathcal{R}_{\mathcal{M}}, \mathcal{Q})$.

8 Going Beyond the Unary Case

Now that we have the complete solution to the unary database reformulation problem, we would like to extend the obtained results to the general case of reformulating databases with stored relations of arbitrary arity. We don't have a solution yet, but the results we have obtained for the unary case give us insight into the directions to move in the general (n-ary) case. The example below shows one possible scenario.

Example 3. Suppose we have a database with five binary stored relations $s_1,\ s_2,\ s_3,\ s_4$, and s_5. Suppose we have only three elementary queries of interest, p, q, and r, with the following definitions:

$$p(X,\ Y)\ :-\ s_1(X,\ Z),\ s_2(Y,\ Z),\ \neg s_3(X,\ Y),\ s_4(X,\ W); \tag{19}$$

$$q(X,\ T)\ :-\ s_1(X,\ Z),\ s_2(Y,\ Z),\ s_3(X,\ Y),\ s_5(X,\ T); \tag{20}$$

$$r(X,\ W)\ :-\ s_1(X,\ Z),\ s_2(Y,\ Z),\ s_4(X,\ W). \tag{21}$$

We could notice a common subexpression $s_1(X,\ Z),\ s_2(Y,\ Z)$ in these three definitions, and could materialize a new relation t defined as:

$$t(X, Y)\ :-\ s_1(X,\ Z),\ s_2(Y,\ Z); \tag{22}$$

this materialization might be done in traditional query optimization.

However, we can do better than that. Consider relations

$$b_1(X,\ Y)\ :-\ s_1(X,\ Z),\ s_2(Y,\ Z),\ \neg s_3(X,\ Y); \tag{23}$$

$$b_2(X,\ Y)\ :-\ s_1(X,\ Z),\ s_2(Y,\ Z),\ s_3(X,\ Y); \tag{24}$$

they are reminescent of the orthogonal basis relations in the unary case.

Notice that the union of b_1 and b_2 gives us exactly the relation t. Now, if we dematerialize $s_1,\ s_2,\ s_3$ and materialize b_1 and b_2, we can rewrite our queries as

$$p(X,\ Y)\ :-\ b_1(X,\ Y),\ s_4(X,\ W); \tag{25}$$

$$q(X,\ T)\ :-\ b_2(X,\ Y),\ s_5(X,\ T); \tag{26}$$

$$r(X,\ W)\ :-\ b_1(X,\ Y),\ s_4(X,\ W); \tag{27}$$

$$r(X,\ W)\ :-\ b_2(X,\ Y),\ s_4(X,\ W). \tag{28}$$

□

The resulting database still consists of binary relations only, so the required storage space cannot increase dramatically (assuming the absence of any functional dependencies in the original stored relations), but now the query definitions look much simpler and can be computed faster.

9 Related Work

Database schema evolution is an integral part of database design, data model translation, schema (de)composition, and multidatabase integration; fundamental to these problems is the notion of equivalence between database schemata.

Database schema equivalence was first studied in [4, 7, 24]. Later, relative information capacity was introduced in [16] as a fundamental theoretical concept which encompasses schema equivalence and dominance. Tutorial [15] surveys a number of frameworks, including relative information capacity, for dealing with the issue of semantic heterogeneity arising in database integration.

In practical database systems, database design frequently uses normalization, first introduced in [8] and described in detail in [28]. [6, 17] survey methods and issues in multidatabase integration.

Query transformation is another aspect of database transformation tasks. Query rewriting is important for query optimization (see [5, 27, 29]), especially in deductive databases [22] where queries can be complex and the amount of data accessed can be overwhelming. [23] is a survey on implementation techniques and implemented projects in deductive databases.

There is an extensive body of work on theoretical aspects of query rewriting. The paper [1] discusses the complexity of answering queries using materialized views and contains references to major results in the areas of query containment and view materialization. [13, 14, 18, 25] describe various approaches to view materialization. [3, 9, 10, 21] treat the problem of using available materialized views for query evaluation.

Transformations of database schemas and queries can be considered together as reformulations of logical theories. [26] provides a theoretical foundation for theory reformulations, and [12, 20] contain work on general transformations of logical theories.

Descriptions of basic methods used in this paper can be found, e.g., in [11].

10 Conclusions and Future Work

We have defined and formally specified database reformulation, as the process of rewriting the data and rules of a deductive database in a functionally equivalent manner. We focus on the problem of automatically reformulating a database in a way that reduces the processing time for a prespecified set of queries while satisfying strong storage space constraints.

In this paper, we have described a complete solution of the database reformulation problem for one class of deductive databases, those where all stored relations are unary and all queries and views are expressed in nonrecursive datalog with negation. We have shown that the reformulation problem for these unary databases is decidable. Furthermore, we have shown that for any such unary database, there is a special reformulation which satisfies strong storage space constraints and where query processing costs for all input queries are as low or lower than in any reformulation that satisfies the same constraints. We have described how to build such a reformulation.

We have also suggested a possible extension of our solution for unary databases to the general case of deductive databases with stored relations of arbitrary arity, under strong storage space constraints.

This paper describes just the first step in the formidable task of taming database reformulation. Our long-term research objective is to explore how database reformulation can be automated for databases of arbitrary arity, with rules expressed in successively more complex standard query languages, i.e., various extensions of datalog. (We have already solved the problem for databases whose rules can be expressed as conjunctive queries.) We also plan to study reformulation of databases with various forms of integrity constraints.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Serge Abiteboul and Oliver Duschka. Complexity of answering queries using materialized views. In *PODS-98*, pages 254–263.
2. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, Reading, Mass., 1995.
3. F.N. Afrati, M. Gergatsoulis, and T.G. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT-99*, pages 435–452.
4. P. Atzeni, G. Ausiello, C. Batini, and M. Moscarini. Inclusion and equivalence between relational database schemata. *Theoretical Computer Science*, 19:267–285, 1982.
5. E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *VLDB-97*, pages 156–165.
6. C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.
7. C. Beeri, A.O. Mendelzon, Y. Sagiv, and J.D. Ullman. Equivalence of relational database schemes. *SIAM J. Comput.*, 10(2):352–370, 1981.
8. E.F. Codd. A relational model of data for large shared data banks. *Comm. ACM*, 13(6):377–387, June 1970.
9. Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *PODS-97*, pages 109–116.
10. Oliver M. Duschka and Michael R. Genesereth. Query planning with disjunctive sources. In *AAAI-98 Workshop on AI and Information Integration.*
11. Herbert B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.
12. Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.
13. Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT-97*, pages 98–112.
14. Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT-99*, pages 453–470.
15. Richard Hull. Managing semantic heterogeneity in databases: a theoretical perspective. In *PODS-97*, pages 51–61.
16. Richard Hull. Relative information capacity of simple relational database schemata. *SIAM J. Comput.*, 15(3):856–886, August 1986.
17. Won Kim, editor. *Modern Database Systems.* ACM Press, New York, New York, 1995.
18. Yannis Kotidis and Nick Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *SIGMOD-99.*
19. Alon Y. Levy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Equivalence, query-reachability and satisfiability in datalog extensions. In *PODS-93*, pages 109–122.
20. Alon Y. Levy and P. Pandurang Nayak. A semantic theory of abstractions. In *IJCAI-95*, pages 196–203.
21. A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS-95*, pages 95–104.
22. Jack Minker. Logic and databases: a 20 year retrospective. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases*, pages 3–57. Springer, 1996. (Proceedings of the LID'96 international workshop).
23. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *J. Logic Progr.*, 23(2):125–149, May 1995.
24. J. Rissanen. On equivalences of database schemes. In *PODS-82*, pages 23–26.
25. K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. In *SIGMOD-96*, pages 447–458.
26. Devika Subramanian. *A theory of justified reformulations.* PhD thesis, Stanford University, 1989.
27. D. Theodoratos and T. Sellis. Data warehouse configuration. In *VLDB-97*, pages 126–135.
28. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, New York, 1988.
29. J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB-97*, pages 136–145.

A Theorem Proofs and Additional Examples

A.1 Proofs for Section 5

We start this section with a simple observation which we will be using in the proofs below.

Observation A.1 *Any query in nr-datalog$^\neg$ on a database schema $\mathcal{S}$ has an (equivalent) safe rewriting where the set of relation schemas for all the subgoals is a subset of $\mathcal{S}$.*

We will call the rewriting of a query q where all predicates in rule bodies correspond to stored relations in $\mathcal{S}$, the *schema rewriting* of q.

Proof (Theorem 1). Let $\mathcal{S} = \{ s_1, s_2, \ldots, s_n \}$. Consider a fixed pair $(\mathcal{S}, q)$, where q is a unary query defined on $\mathcal{S}$; let $\mathcal{B}$ be the orthogonal basis of $\mathcal{S}$.

It is easy to show that the schema rewriting $\tilde{q}$ (see Observation A.1) of q on $\mathcal{S}$ is a set of rules where the body of each rule is a unary subquery.

Let us show that the body of each rule in $\tilde{q}$ can be converted into a union of relations in the orthogonal basis $\mathcal{B}$ of $\mathcal{S}$. Consider an arbitrary rule r in $\tilde{q}$; let the only variable in r be X. The body of r is a unary subquery; let us call it $\mathcal{C}(X)$.

By definition of the schema rewriting $\tilde{q}$, each subgoal in r corresponds to a relation name in $\mathcal{S}$, and thus $\mathcal{C}(X)$ consists of literals which are (possibly negated) relation names in $\mathcal{S}$; notice that because all rules are safe, at least one conjunct in $\mathcal{C}(X)$ is not negated. We can assume without loss of generality that each relation name in $\mathcal{S}$ occurs in $\mathcal{C}(X)$ no more than once. Then $\mathcal{C}(X)$ looks as follows:

$$l_{i_1}(X),\ l_{i_2}(X),\ \ldots,\ l_{i_r}(X); \tag{29}$$

here, $l_j(X)$ is either $s_j(X)$ or $\neg s_j(X)$, where j is between 1 and n; since each relation name in $\mathcal{S}$ occurs in $\mathcal{C}(X)$ at most once, the total number m of conjuncts in $\mathcal{C}(X)$ does not exceed the size n of $\mathcal{S}$: $m \leq n$.

Now let us show, by induction on the difference k between n and m, that $\mathcal{C}(X)$ has an equivalent rewriting as a union of relations in the orthogonal basis $\mathcal{B}$ of $\mathcal{S}$.

1. Basis: $k = n - m = 0$. Here each relation $s_j \in \mathcal{S}$ is represented in $\mathcal{C}(X)$ exactly once, and at least one of the subgoals of $\mathcal{C}(X)$ is not negated. Thus, $\mathcal{C}(X)$ is the body of the definition of one of the orthogonal basis relations $b_i \in \mathcal{B}$, and we can rewrite $\mathcal{C}(X)$ as b_i.
2. Induction: $k = n - m > 0$. Consider $\mathcal{C}(X)$ with m literals. Since $m < n$, there is at least one relation s_i in $\mathcal{S}$ which is not represented in $\mathcal{C}(X)$. Then $\mathcal{C}(X)$ can obviously be rewritten as a disjunction:

$$\mathcal{C}(X) \equiv (\ \mathcal{C}(X),\ s_i(X)\ ) \bigcup (\ \mathcal{C}(X),\ \neg s_i(X)\ ). \tag{30}$$

   Now each disjunct in the RHS of the equation has $m + 1$ literals and thus, by the inductive hypothesis, can be represented as a union of basis relations in $\mathcal{B}$.
3. By repeatedly rewriting $\mathcal{C}(X)$ as an increasingly long union of components, as shown in 1 and 2 above, we obtain a disjunction of relations in $\mathcal{B}$ which is an equivalent rewriting of $\mathcal{C}(X)$. The process terminates when the number of conjuncts in each disjunct reaches n.

The case when X is a not a variable but a constant is treated analogously to the case with variables.

Now we replace each such $\mathcal{C}(X)$, for each variable or constant, in each rule in $\tilde{q}$ by its rewriting as a union of orthogonal basis relations in $\mathcal{B}$. The resulting set of rules $q_{\mathcal{B}}$ is equivalent to $\tilde{q}$. Finally, by transitivity of equivalence via $\tilde{q}$, we can conclude that $q_{\mathcal{B}}$ is a rewriting of q. □

Proof (Corollary 1). Let $\mathcal{S} = \{ s_1, s_2, \ldots, s_n \}$. Consider a fixed pair $(\mathcal{S}, q)$, where q is an arbitrary query defined in *nr-datalog*$^\neg$ on $\mathcal{S}$; let $\mathcal{B}$ be the orthogonal basis of $\mathcal{S}$.

(1) *Existence of a rewriting:* since any rule in q is a cross-product of unary subqueries, any such rule can be (equivalently) rewritten completely as a cross-product of unions of relations in the orthogonal basis of the schema $\mathcal{S}$; see Theorem 1. To turn the resulting query into the *nr-datalog*$^\neg$ format, one may need to convert cross-products of unions, in bodies of rules, into a set of conjunctions, using a standard procedure.

(2) *Uniqueness of the rewriting*: Suppose there are two rewritings of q in terms of the set $\mathcal{B}$, $q_{\mathcal{B}}^{(1)}$ and $q_{\mathcal{B}}^{(2)}$. It is easy to show that any rule in these rewritings must be in the following format:

$$r_i^{(j)}(X_1,\ X_2,\ \dots\ ,\ X_m)\ :-\ b_{k_1}(X_1),\ b_{k_2}(X_2),\ \dots\ ,\ b_{k_m}(X_m); \tag{31}$$

where j is either 1 or 2, all m variable names in the head of the rule are different, and each b_{k_l} in the rule's body is in $\mathcal{B}$. Notice that since all variable names are different, there are no intersections of subgoals in the bodies of the rules; also, since all rules are safe, there can be no negated subgoals in the rules.

Now, since $q_{\mathcal{B}}^{(1)}$ and $q_{\mathcal{B}}^{(2)}$ are equivalent, by the containment mapping theorem for positive datalog with disjunctions, the relation for each rule in $q_{\mathcal{B}}^{(1)}$ is contained in the relation for some single rule in $q_{\mathcal{B}}^{(2)}$, and vice versa. Consider an arbitrary rule $r^{(1)}$ in $q_{\mathcal{B}}^{(1)}$, and consider the rule $r^{(2)}$ in $q_{\mathcal{B}}^{(2)}$ such that $r^{(1)}$ is contained in $r^{(2)}$. It is not possible that the containment is proper in any database instance with schema $\mathcal{B}$, since the sets of objects in the tables for basis relations are pairwise disjoint. Thus the definitions of $r^{(1)}$ and $r^{(2)}$ are the same, up to variable renamings.

From this observation it is clear that there is a one-to-one correspondence between the rules in $q_{\mathcal{B}}^{(1)}$ and $q_{\mathcal{B}}^{(2)}$. Thus, the rewriting of q in terms of the orthogonal basis $\mathcal{B}$ of $\mathcal{S}$ is unique up to reorderings of subgoals. □

Proof (Theorem 2).

1. Follows from the proof to Corollary 1.

2. Follows from the property that for any database instance D with schema $\mathcal{S}$, each object in the universe of discourse (UOD) of D belongs to exactly one relation in $\mathcal{B}$.

3. We consider three elementary types of database updates: (A) insertion, (B) deletion, and (C) proper update which we model as a deletion followed by an insertion. Let us consider a fixed database instance D with schema $\mathcal{S}$; let D' be the database instance with the schema $\mathcal{B}$, obtained from D by the orthogonal basis reformulation. In what follows, we assume the presence of certain indexes and metadata that will be descibed as needed.

Now let us consider, in turn, the three elementary update operations in D that we have isolated, and study the complexity of the corresponding operations in D'.

(A) For an insertion of an object α into the table for a relation s_i in D, there are two cases:

- if α is not already in the UOD of D then, in D', it needs to be placed into a relation b_j which contains objects belonging to s_i only and not to any other relation; this relation b_j can be mapped to s_i once before D' is populated; therefore, the time required to insert α into D' is constant;
- if, however, α is already in the UOD of D, then the first action in D' will be to access, from α, the table to which it belongs (this operation takes constant time with the use of an index), and then to examine, in the metadata for D', the definition of the basis relation for that table (takes time which is linear in the length of the definition of the relation, i.e., in the number of elements in $\mathcal{S}$); if the subgoal for s_i is not negated in this definition then the object is already in the correct table, and no further action is required; if, on the other hand, the subgoal for s_i is negated in the definition, then, after deleting α from that table, the next and final action is to find the basis relation which has exactly the same definition except that s_i is not negated there (takes constant time with an index), and to place α into the corresponding table; in both cases the total complexity of the insertion operation in D' depends on simple index accesses described above and is thus linear in the number of elements of $\mathcal{S}$.

(B) For a deletion of an object α from the table for a relation s_i in D, there are also two cases, and the analysis is similar to that for the insertion case.

(C) A proper update is a deletion followed by an insertion; therefore, its complexity is the maximum of the complexities of its components, i.e., is also linear in the size of the schema $\mathcal{S}$. □

A.2 Proofs for Section 6

Proof (Theorem 3). In this proof, we consider a relation r, defined in *nr-datalog*$^\neg$ on a unary database schema $\mathcal{S}$, and a database instance D with schema $\mathcal{S}$.

(1) The "if" part: let r be a unary relation. Consider an arbitrary database D with schema $\mathcal{S}$; the set of answers to r in D is effectively a set of some objects that are already stored in D. In the worst case, the set of answers to r includes all the objects stored in D; even in this case, the space required to store the set of answers to r cannot exceed the space required to store D. We conclude the proof by noting that this result does not depend on the choice of the database instance D.

(2) The "only if" part: suppose some relation r is such that for any database instance D with schema $\mathcal{S}$, the set of answers to r in that database does not require more storage space than D itself.

Assume r is not unary; suppose r is a binary relation. We will show that in this case, there exists a database D with schema $\mathcal{S}$, such that the set of answers to r on that database cannot "fit into" the storage space required to store D.

Consider a schema rewriting of the rules for r (see Observation A.1). For r to be binary, there must be at least one rule in the schema rewriting with two different variables in the head, since relations like $r(X,\ X)$ are essentially unary; let us call these variables X and Y. For this rule to be safe, the body of the rule must have at least two nonnegated subgoals, one with argument X and the other with argument Y; let these subgoals be $s_i(X)$ and $s_j(Y)$, $X \neq Y$, $s_i \in \mathcal{S}$ and $s_j \in \mathcal{S}$. Notice that for the set of answers to the rule not to be empty in all databases with schema D, no negated subgoal with argument X in the body of the rule can have relation name s_i; similarly for Y and s_j. Let $\mathcal{S}'$ be the set of all relation names in $\mathcal{S}$ such that this rule for r has a nonnegated subgoal with that relation name (notice that subgoals with variables other than X or Y are redundant in the body of the rule); let k be the number of relations in $\mathcal{S}'$.

Now consider a database instance D with schema $\mathcal{S}$, such that the only nonempty tables in D are those for the relation names in $\mathcal{S}'$. Let the size of the UOD of D be any $m > k/2$; let each of the k nonempty tables in D contain all the m objects in the UOD of D. Then the number of objects stored in D is $k * m$.

Now, when we compute this particular rule for r, we see that the set of answers to this rule is the set of two-element tuples, where there is a tuple for each combination of two objects in the UOD of D. Thus, the number of answers to this particular rule in D is m^2, and the number of objects that need to be stored for these answers is $2 * m^2$ (we count as a unit the space needed to store an argument value). Since $m > k/2$, we have $2 * m^2 > k * m$. Since the set of answers to r includes all answers to the rule, the space needed to store the set of answers to r is at least the space needed for this rule. Therefore, the set of answers to r in this database D requires more storage space that D itself.

We have shown that our premise does not hold when r is binary; thus we have proved the claim by contradiction for all binary relations that can be defined on a unary database schema $\mathcal{S}$. A similar counterexample can be built for a relation r of arbitrary arity greater than 2. We can conclude that to "fit into" the storage space of an arbitrary database with schema $\mathcal{S}$, r needs to be unary. □

Proof (Theorem 4). After we notice that a union of orthogonal basis relations, when materialized, satisifes the minimal-space constraint, the claim of the theorem follows immediately from Theorems 1 and 3. □

Proof (Theorem 5). Consider an arbitrary candidate reformulation $(\mathcal{V}, \mathcal{R}_{\mathcal{V}}, \mathcal{Q})$ of a triple $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$ where $\mathcal{S}$ is unary. By definition, for any database instance D with schema $\mathcal{S}$ and its reformulated counterpart D' with schema $\mathcal{V}$, none of the stored (materialized) relations in D' take up more storage space than D. Thus in all candidate reformulations of $(\mathcal{S}, \mathcal{R}_{\mathcal{S}}, \mathcal{Q})$, all stored relations are unary relations. Observing that Algorithm 6 outputs all reformulations whose all stored (materialized) relations are unary, concludes the proof. □

A.3 Proofs for Section 7

Proof (Theorem 6). Observe that in the minimal non-forking reformulation, the only operations are unions and cross-products (since any candidate reformulation has the same properties as the orthogonal basis reformulation, and from Theorem 2). We assume the standard bottom-up query evaluation cost model; in this model, all unary subqueries of each rule are computed before any Cartesian product is processed. The stored (materialized) relations in the minimal non-forking reformulation are maximal unions of basis relations, such that these unions belong to the same unary subgoal. Assuming that it is at least as fast to scan a union once and then to perform a Cartesian product, than it is to retrieve the elements of the union one by one, combined with the Cartesian product each time, and then to take the union of all the results, we obtain the result of the theorem. □