Expressive Capabilities Description Languages and Query Rewriting Algorithms*

Vasilis Vassalos† Yannis Papakonstantinou‡

Abstract

Information integration systems have to cope with a wide variety of different information sources, which support query interfaces with very varied capabilities. To deal with this problem, the integration systems need descriptions of the query capabilities of each source, *i.e.*, the set of queries supported by each source. Moreover, the integration systems need algorithms for deciding how a query can be answered given the capabilities of the sources. Finally, they need to translate a query into the format that the source understands. We present two languages suitable for descriptions of query capabilities of sources and compare their expressive power. We also use one of the languages to automatically derive the capabilities description of the integration system itself, in terms of the capabilities of the sources it integrates. We describe algorithms for deciding whether a query "matches" the description and show their application to the problem of translating user queries into source-specific queries and commands. We propose new, improved algorithms for the problem of answering queries using these descriptions. Finally, we identify an interesting class of source capability descriptions, for which our algorithms are much more efficient.

1 Introduction

Users and applications today require integrated access to multiple heterogeneous information systems, many of which are not conventional SQL database management systems. Examples of such systems are Web sources with forms interfaces, object repositories, bibliographic databases, etc. Some of these systems provide powerful query capabilities, while others provide limited query interfaces. Systems that integrate information from multiple sources have to cope with the different and limited capabilities of the sources. In particular, integrating systems must allow users to query the data using a single powerful query language, without having to know about the diverse capabilities of each source.

Figure 1 illustrates a typical high level architecture of an integration system. The *mediator* decomposes incoming client queries, which are expressed in some common query language, into new common-language queries which are sent to the *wrappers*. Then the wrappers translate the incoming queries into queries and commands which are expressed in the native language of the source and are supported by it. Indeed, the queries received by the wrappers should be supported by the sources, in the sense that they directly correspond to supported source queries. (It is counterproductive to build wrappers that accept queries which are not directly supported by the corresponding source [PGH96, HKWY97].) Apparently, both the wrappers and the mediators require *descriptions* of

*Research partially supported by NSF grant IRI–96–31952, ARO grant DAAH04–95–1–0192, Air Force contract F33615–93–1–1339 and the Lilian Voudouri Foundation.

†Computer Science Dept., Stanford University, Stanford, CA 94305. email: `vassalos@db.stanford.edu`

‡Computer Science and Engineering Dept., UCSD, San Diego, CA 92093. email: `yannis@cs.ucsd.edu`

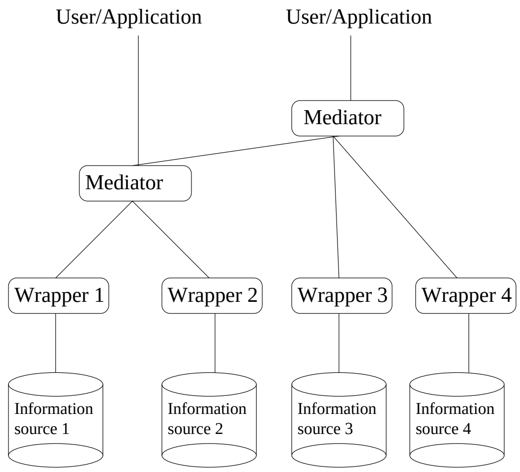

Figure 1: A common architecture for integration

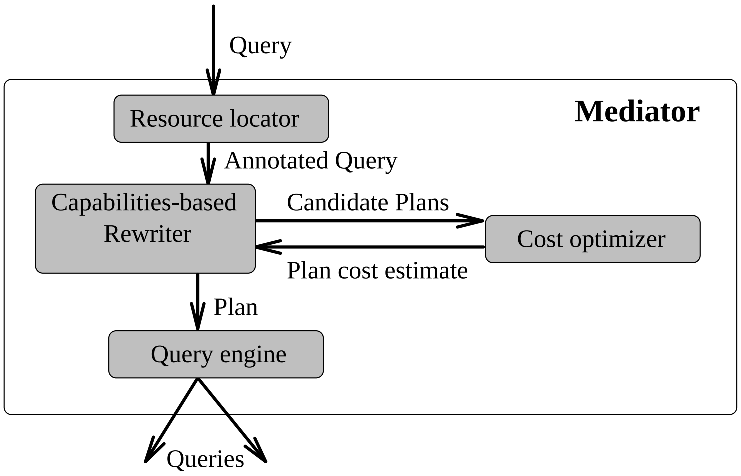

Figure 2: Mediator architecture

the query capabilities of the participating sources in order to correctly reduce the client query into queries supported by the wrappers and, then, translate it into a supported native query.

In particular, a special module of the mediator, called the *Capabilities-Based Rewriter (CBR)*, uses the description to adapt to the query capabilities of the sources. Let us use an example to illustrate the query processing steps followed by the mediator modules (see Figure 2). Consider a source that exports a "lookup" catalog $lookup(Employee, Manager, Specialty)$ for the employees of a company. The description indicates that this source supports only selection queries. Let us now assume that the *client query*, or simply "query", requests the managers who have at least one employee specialized in *Java* and at least one employee specialized in *Databases*. Notice that this query is answered with a self join of the *lookup* table on *Manager*. The first module of the mediator, called *resource locator*, knows (from metadata or views or source data descriptions) that all the data needed for answering this query reside on "lookup". Consequently it formulates an *annotated query* where each relation is annotated with its origin. Notice that finding *where* are the needed data is a problem orthogonal to *how* they can be obtained. The problem treated in this paper is the latter one.

Then the CBR takes as input the descriptions and the annotated query and it infers *plans* for retrieving the required data. In our running example the plan, which is executed by the mediator's *engine*, could first retrieve the set of managers of *Java* employees, then the set of managers of *Database* employees, and finally it would intersect the two sets. Alternatively, the CBR may form a *sideways information passing* plan: First it retrieves the set of managers of *Java* employees and then, for each manager m, it issues a query to check if m has a *Database* employer. Indeed, the CBR typically produces more than one *candidate plans* for the query. We assume that a *cost optimizer* will provide *cost estimates*. Notice that our approach is based on a loose coupling of the CBR with the optimizer. Systems and algorithms where a CBR module and the optimizer are tightly coupled are described in [HKWY97] and [PGH]. At any rate, we are not concerned in this paper with estimating the cost of our plans. Relevant work can be found in [ACPS96, DKS92].

The wrappers also need descriptions of the source capabilities in order to translate the supported common-language queries into queries and commands understood by the source interface. In particular, each description is associated with *actions* that perform the translation, in the same style with Yacc [ASU87]. Using this approach, in the TSIMMIS project at Stanford we have wrapped a number of real life bibliographic sources [PGGMU95, JBHM$^+$97, H$^+$97].

It is clear that languages for describing the set of supported queries are needed. The introduction of new languages for describing query capabilities brings up two questions studied in this paper: (i) are these languages expressive enough? (ii) Given a description of the wrappers' capabilities, how can we answer a client query using only queries answerable (*i.e.*, supported) by the wrappers? We refer to this problem as the Capabilities-Based Rewriting (CBR) problem [PGH96, HKWY97] since the corresponding algorithm is the one run by the CBR module; it is also clearly related to the Answering Queries Using Views problem [LMSS95, RSU95, LRU96] (see Section 4). In this paper, we focus on sources that support conjunctive queries, *i.e.*, their capabilities are a subset of CQ [AHV95].

This paper extends the results of [VP97]. In particular the topics and novel contributions are as follows:

- We introduce the description language p-Datalog, we formally define the set of queries described by p-Datalog programs, and present complete and efficient procedures that (i) decide whether a query is described by a p-Datalog description. This is the algorithm run by the wrapper and note that it also finds out what translating actions must be executed. (ii) decide whether a query can be answered by combining supported queries (the CBR problem). This algorithm is run by the mediator. Our algorithm runs in time non-deterministic exponential in the size of the query and the description, a substantial improvement over the algorithm described in [LRU96], which was non-deterministic doubly exponential.

- We study the expressive power of p-Datalog. We reach the important result that p-Datalog can *not* describe the query capabilities of certain powerful sources. In particular, we show that there is no p-Datalog program that can describe all conjunctive queries over a given schema. Indeed, there is no program that describes all boolean conjunctive queries over the schema. This paper presents expressiveness results that have not been reported in [VP97] and it also provides formal proofs.

- We describe and extend RQDL [PGH96, PGH], a provably more powerful language than p-Datalog, which also keeps the salient features of p-Datalog.

- We provide a reduction of RQDL descriptions into p-Datalog augmented with *function symbols*. The reduction has important practical and theoretical value. From a practical point of view, it reduces the CBR problem for RQDL to the CBR problem for p-Datalog, thus giving a complete algorithm that is applicable to all RQDL descriptions[1]. From a theoretical point of view, it clarifies the difference in expressive power between RQDL and p-Datalog. The current paper presents the reduction, as well as the algorithms for the complete RQDL language.

Besides presenting the complete CBR algorithms, expressiveness results, and proofs the current paper also makes the following contributions, not present in [VP97]:

- We identify an important class of descriptions, covering sources such as document retrieval systems, lookup catalogs, and object repositories, and we show that the complexity of the CBR problem for the specific class is significantly lower than the complexity for the general case.
- We provide an algorithm that takes as input descriptions of the queries supported by the wrappers and outputs a description of all queries supported by a mediator that accesses these wrappers. This algorithm is important when we have mediators accessing other mediators, as in Figure 1 — hence requiring knowledge of the query capabilities of the accessed mediators.
- We investigate the expressive power relationship between the proposed description languages and Datalog queries annotated with binding patterns. Furthermore, we provide the completeness proofs and complexity arguments for a p-Datalog CBR algorithm which also produces plans using sideways information passing.

The next section introduces the p-Datalog description language. Section 3 describes the algorithm run by the wrappers. Section 4 describes a CBR algorithm run by the mediators. Section 5 studies a useful large class of descriptions, for which the CBR problem has lower computational complexity. Section 6 discusses expressive power issues. Section 7 introduces RQDL. Section 8 discusses the RQDL description of mediator capabilities. Section 9 describes the reduction of RQDL to p-Datalog with function symbols and Section 10 describes the wrapper and mediator algorithms for RQDL. Section 11 discusses the related work. Section 12 gives conclusions.

2 The p-Datalog Source Description Language

It is well known that the most popular real-life query languages, like SPJ queries [AHV95] and Web-based query forms are equivalent to conjunctive queries. A Datalog program is a natural encoding of many sets of conjunctive queries: the set is described by the expansions of the Datalog program. First, we describe informally a Datalog-based source description language and illustrate it with examples. A formal definition follows in the next subsection.

In the simple case, when we deal with a weak information source, the source can be described using a set of parameterized queries. Parameters, called *tokens* in this paper, specify that some constant is expected in some fixed position in the query [PGGMU95, PGH96, LRU96, LRO96]. Without loss of generality, we assume the existence of a designated predicate ans that is the head of all the parametrized queries of the description.

Example 2.1 Consider a bibliographic information source, that provides information about books. This source exports a predicate $books(isbn, author, title, publisher, year, pages)$. The source also exports "indexes," $author\_index(author\_name, isbn)$, $publisher\_index(publisher, isbn)$ and

[1] The algorithm presented in [PGH96, PGH] only works for RQDL descriptions without the important *union* metapredicate.

$title\_index(title\_word, isbn)$. Conceptually, the tuple (X, Y) is in $author\_index$ if the string X resembles the actual name of an author and Y is the ISBN of a book by that author. Similarly, (X, Y) is in $title\_index$ if X is a word of the actual title and Y is the ISBN of a book with word X in the title. The following parameterized queries describe the wrapper that answers queries specifying an author, a title or a publisher.

$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), author\_index(\$c, Id)$$
$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), title\_index(\$c, Id)$$
$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), publisher\_index(\$c, Id)$$

where \$c denotes a *token*. The query

$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), author\_index(\text{`}\mathtt{Smith}\text{'}, Id)$$

can be answered by that source, because it is derived by the first parameterized query by replacing $\$c$ by the constant `'Smith'`. □

In the previous example, the source is described by parameterized conjunctive queries. Note that if, for instance, the source accepts queries where values for any combination of the three indexes are specified, we would have to write $2^3 = 8$ parameterized conjunctive queries. The next example uses IDB predicates (*i.e.*, predicates that are defined using source predicates and other IDB predicates) to describe the abilities of such a source more succinctly. Finally, example 2.3 uses recursive rules to describe a source that accepts an infinite set of query patterns.

Example 2.2 Consider the bibliographical source of the previous example. Assume that the source can answer queries that specify any combination of the three indexes. The p-Datalog program that describes this source is the following:

$$\begin{array}{lll}
ans(Id, Aut, Titl, Pub, Yr, Pg) & \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), & \\
 & \quad ind_1(Id), ind_2(Id), ind_3(Id) & (1) \\
ind_1(Id) & \leftarrow title_index(\$c, Id) & \\
ind_1(Id) & \leftarrow \epsilon & (2) \\
ind_2(Id) & \leftarrow author_index(\$c, Id) & (3) \\
ind_2(Id) & \leftarrow \epsilon & \\
ind_3(Id) & \leftarrow publisher_index(\$c, Id) & \\
ind_3(Id) & \leftarrow \epsilon & (4)
\end{array}$$

ϵ denotes an empty body, *i.e.*, an ϵ–rule has an empty expansion. Notice that ϵ–rules are unsafe [Ull89]. In general, p-Datalog rules can be unsafe but that is not a problem under our semantics. Note also that the number of rules is only polynomial in the number of the available indexes, whereas the number of possible expansions is exponential.

The query

$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), author\_index(\mathtt{Smith}, Id)$$

can be answered by that source, because it is derived by expanding rule 1 using rules 2, 3 and 4, and by replacing $\$c$ by the constant `Smith`. We can easily modify the description to require that *at least* one index is used. □

In general, a p-Datalog program describes all the queries that are expansions of an *ans*-rule of the program. In particular, p-Datalog rules that have the *ans* predicate in the head can be expanded into a possibly infinite set of conjunctive queries. Among the expansions generated, some will only refer to source predicates[2]. We call these expansions *terminal expansions*. A p-Datalog program can have unsafe terminal expansions. We say that the p-Datalog program *describes* the set of conjunctive queries that are its safe terminal expansions. (see formal definitions in the next subsection).

Example 2.3 Consider again the bibliographical source of Example 2.1. Assume that there is an abstract index $abstract\_index(abstract\_word, Id)$ that indexes books based on words contained in their abstracts. Consider a source that accepts queries on books given one or more words from their abstracts. The following p-Datalog program can be used to describe this source.

$$\begin{array}{ll} ans(Id, Aut, Titl, Pub, Yr, Pg) & \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), ind(Id) \\ ind(Id) & \leftarrow abstract\_index(\$c, Id) \\ ind(Id) & \leftarrow ind(Id), abstract\_index(\$c, Id) \end{array}$$

□
As another example of a recursive source description, we can think of a transportation company, such as FedEx, that has an information source capable of answering queries about flights. Assume that the source can answer whether there exists a flight between cities A and B that makes n stops. We can model such a source with a p-Datalog program.

2.1 Formal description of p-Datalog.

We assume familiarity with Datalog, e.g. [Ull89, AHV95]. Besides the constant and variable sorts, we use a third disjoint set of symbols, the set of *tokens*.

Definition: p-Datalog Program Syntax A *parametrized Datalog rule* or *p-Datalog* rule is an expression of the form

$$p(u) \leftarrow p_1(u_1), \ldots, p_n(u_n)$$

where $p, p_1, p_2, \ldots, p_n$ are relation names, and $u, u_1, u_2, \ldots, u_n$ are tuples of constants, variables and tokens of appropriate arities. A *p-Datalog program* is a finite set of p-Datalog rules. □

Tokens are variables that have to be instantiated to form a query. We now formalize the semantics of p-Datalog as a source description language.

Definition: Set of Queries Described/Expressible by a p-Datalog Program Let P be a p-Datalog program with a particular IDB predicate ans. The set of *expansions* $\mathcal{E}_P$ of P is the smallest set of rules such that:

- each rule of P that has ans as the head predicate is in $\mathcal{E}_P$;
- if r_1: $p \leftarrow q_1, \ldots, q_n$ is in $\mathcal{E}_P$, r_2: $r \leftarrow s_1, \ldots, s_m$ is in P (assume their variables and tokens are renamed, so that they don't have variables or tokens in common) and a substitution θ is the most general unifier of some q_i and r then the resolvent

  $$\theta p \leftarrow \theta q_1, \ldots \theta q_{i-1}, \theta s_1, \ldots, \theta s_m, \theta q_{i+1}, \ldots, q_n$$

  of r_1 with r_2 using θ is in $\mathcal{E}_P$.

The set of *terminal expansions* $\mathcal{T}_P$ of P is the subset of all expansions $e \in \mathcal{E}_P$ containing only EDB predicates in the body. The set of queries *described by* P is the set of all rules $\rho(r)$, where $r \in \mathcal{T}_P$ and ρ assigns arbitrary constants to all tokens in r. The set of queries *expressible* by P is the set of all queries that are equivalent to some query described by P. □

[2] We stated that source predicates are the EDB predicates of our descriptions.

Unification extends to tokens in a straightforward manner: a token can be unified with another token, yielding a token. When unified to a variable, it also yields a token. When unified to a constant, it yields the constant. The above definitions can easily be extended to accommodate more than one "designated" predicates (like ans).

In the context of the above description semantics, we will use the terms p-Datalog *program* and *description* interchangeably.

Informally, we observe that expansions are generated in a grammar-like fashion, by using Datalog rules as productions for their head predicates and treating IDB predicates as "nonterminals" [ASU87]. Resolution is a generalization of non-terminal expansion; rules of context-free grammars can simply be thought of as Datalog rules with 0 arguments.

Rectification: For deciding expressibility as well as for solving the CBR problem the following rectified form of p-Datalog rules simplifies the algorithms. We assume the following conditions are satisfied:

- No variable appears twice in subgoals of the query body. Instead, multiple occurrences of the same variable are handled by using distinct variables and making equalities explicit with the use of the equality predicate $equal$.
- No variable appears twice in the head of the query. Again, equalities are made explicit with use of the predicate $equal$.
- No constants or tokens appear among the ordinary[3] subgoals. Instead, every constant c or token $\$c$ is replaced by a unique variable C, and an equality subgoal $equal(C, c)$ or $equal(C, \$c)$ is added to equate the variable to the constant.
- No variables appear only in an $equal$ subgoal of a query.

Example 2.4 Consider the query

$$ans(X, X, Z) \leftarrow r(X, Y, Z), p(a, Y) \tag{1}$$

which contains a join between the second columns of r and p, a selection on the first column of p, and the same variable in two columns of ans. Its rectified equivalent is

$$ans(X_1, X, Z) \leftarrow r(X, Y, Z), p(A, Y_1), , equal(X, X_1), equal(Y, Y_1), equal(A, a) \tag{2}$$

□

Notice that we treat the $equal$ subgoal not as a built-in predicate, but as a source predicate. We call rules that obey these conditions *rectified* rules and the process that transforms any rule to a rectified rule *rectification*. We call the inverse procedure (that would give us rule 1 from rule 2) *de-rectification*.

In sections 3 and 4 we provide algorithms for deciding whether a query is expressible by a description and for solving the CBR problem.

[3] We refer to the EDB and IDB relations and their facts as *ordinary*, to distinguish them from facts of the *equal* relation.

3 Deciding query expressibility with p-Datalog descriptions

In this section we present an algorithm for query expressibility of p-Datalog descriptions. In doing that, we develop the techniques that will allow us in the next section to give an elegant and improved solution to the problem of answering queries using an infinite set of views described by a p-Datalog program[LRO96].

Our algorithm, the **Q**uery **E**xpressibility **D**ecision algorithm, is an extension of the classic algorithm for deciding query containment in a Datalog program that appears in [RSUV89] (also see [Ull89]). The algorithm tries to identify one expansion of the p-Datalog program that is equivalent to our query. We next illustrate the workings of the algorithm with an example.

Example 3.1 Let us revisit the bibliographic source of previous examples. Assume that the source contains a table $books(isbn, author, publisher)$, a word index on titles, $title\_index(title\_word, isbn)$ and an author index $au\_index(au\_name, isbn)$. Also assume that the query capabilities of the source are described by the following p-Datalog program:

$$\begin{array}{ll} ans(A,P) & \leftarrow books(Id,A,P), ind_1(Id_1), ind_2(Id_2), equal(Id,Id_1), equal(Id,Id_2) \\ ind_1(Id) & \leftarrow title\_index(V,Id), equal(V,\$c) \\ ind_1(Id) & \leftarrow \epsilon \\ ind_2(Id) & \leftarrow au\_index(V,Id), equal(V,\$c) \\ ind_1(Id) & \leftarrow \epsilon \end{array}$$

Let us consider the query Q

$$ans(X,Y) \leftarrow books(Id,X,Y), title\_index(\text{`}\texttt{Zen}\text{'}, Id), au\_index(\text{`}\texttt{Smith}\text{'}, Id)$$

First we produce its rectified equivalent

$$\begin{array}{ll} Q' : ans(X,Y) \leftarrow & books(Id,X,Y), title\_index(V_1,Id_1), au\_index(V_2,Id_2), equal(V_1, \text{`}\texttt{Zen}\text{'}), \\ & equal(V_2, \text{`}\texttt{Smith}\text{'}), equal(Id,Id_1), equal(Id,Id_2) \end{array}$$

Apparently the above query is expressible by the description. Intuitively, our algorithm discovers expressibility by "matching" the Datalog program rules with the subgoals. In particular, the "matching" is done as follows: first we create a DB containing a "frozen fact" for every subgoal of the query. Frozen facts are derived by turning the variables into unique constants which will be denoted with a bar.

Moreover, we want to capture all the information carried by *equal* subgoals into the DB. If, for example, subgoals $equal(X,Y), equal(X,Z)$ exist in the query, we will generate "frozen" facts for all implicit equalities as well, *i.e.*, $equal(Y,X), equal(Y,Z)$ etc. In the interests of space and clarity, we will write $equal(X,Y,Z)$ to mean that all the previously mentioned facts are in the DB. The DB for our running example is then

$$\begin{array}{l} books(\bar{id}, \bar{x}, \bar{y}), title\_index(\bar{v}_1, \bar{id}_1), au\_index(\bar{v}_2, \bar{id}_2), equal(\bar{id}, \bar{id}_1, \bar{id}_2), \\ equal(\bar{v}_1, \text{`}\texttt{Zen}\text{'}), equal(\bar{v}_2, \text{`}\texttt{Smith}\text{'}) \end{array}$$

We then evaluate the Datalog program on the DB, deriving more facts for the IDB's. In addition, we keep track of the set of frozen facts, called *supporting set*, that we used for deriving each fact. Here is the set of facts and supporting sets derived by a particular evaluation of the Datalog program.

$< ind_1(Id), \quad \{\} >$
$< ind_2(Id), \quad \{\} >$
$(1) ::< ans(\bar{x}, \bar{y}), \quad \{books(\bar{id}, \bar{x}, \bar{y}), equal(\bar{id}, \bar{id})\} >$
$< ind_1(\bar{id}_1), \quad \{title\_index(\bar{v}_1, \bar{id}_1), equal(\bar{v}_1, \text{`Zen'})\} >$
$< ind_2(\bar{id}_2), \quad \{au\_index(\bar{v}_2, \bar{id}_2), equal(\bar{v}_2, \text{`Smith'})\} >$
$(2) ::< ans(\bar{x}, \bar{y}), \quad \{books(\bar{id}, \bar{x}, \bar{y}), title\_index(\bar{v}_1, \bar{id}_1), equal(\bar{v}_1, \text{`Zen'}),$
$au\_index(\bar{v}_2, \bar{id}_2), equal(\bar{v}_2, \text{`Smith'}), equal(\bar{id}, \bar{id}_1, \bar{id}_2)\} >$

Every *ans* fact that is identical to the frozen head of the client query "corresponds" to a query that contains the client query. Furthermore, we can derive the containing query from the <fact, supporting set> pair by translating "frozen" facts back into subgoals. In our running example, the two containing queries[4] correspond to (1) and (2). If the supporting set is identical to the DB that we started with (modulo redundant equality subgoals) then the "corresponding" query is equivalent to the client query. Indeed, the "corresponding" query to (2) is

$$ans(X, Y) \leftarrow books(Id, X, Y), title\_index(Id, \text{`Zen'}), au\_index(Id, \text{`Smith'})$$

which is equivalent (actually identical) to our given query. □

Algorithm QED starts by mapping the subgoals of the given query into "frozen" facts, such that every variable maps to a unique constant, thus creating the *canonical database* [RSUV89, Ull89] of the query, and then evaluates the p-Datalog program on it, trying to produce the "frozen" head of the query. Moreover, it keeps track of the different ways to produce the same fact; that is achieved by "annotating" each produced fact f with its *supporting* facts, *i.e.*, the facts of the canonical DB that were used in that derivation of f.

We next formalize the notion of the canonical database. A formal definition of supporting facts follows.

Definition: Canonical DB of Query Q Let $Q : H \leftarrow G_1, \ldots, G_k, \ldots, E_1, \ldots, E_m$ be a rectified conjunctive query, where $G_1, \ldots, G_k$ are the ordinary subgoals and $E_1, \ldots, E_m$ are the equality subgoals. Select a mapping τ that assigns to every variable X of Q a unique "frozen" constant $\tau(X) = \bar{x}$ and is the identity mapping on constants and predicate names. This way we construct k "frozen" ordinary facts: $\tau(G_1), \ldots, \tau(G_k)$. We also construct m "frozen" facts of the EDB predicate *equal*: $\tau(E_1), \ldots, \tau(E_k)$. These m facts constitute an instance of the *equal* relation. We create additional *equal* facts so that we get the smallest set of *equal* facts that includes this instance and is an equivalence relation. All the constructed facts constitute the canonical DB of query Q. □

Notice that this DB contains two "kinds" of constants: "regular" constants and frozen constants.

Example 3.2 Consider the rectified query:

$$ans(Y) \leftarrow p(X, X_1), q(X_2, Y, Z), equal(X, X_1), equal(X_1, X_2), equal(X, X_3), equal(Z, c)$$

The canonical DB produced by this query is

$p(\bar{x}, \bar{x}_1), q(\bar{x}_2, \bar{y}, \bar{z}), equal(\bar{x}, \bar{x}_1), equal(\bar{x}_1, \bar{x}_2), equal(\bar{z}, c), equal(\bar{x}, \bar{x}), equal(\bar{x}_1, \bar{x}_1),$
$equal(\bar{x}_2, \bar{x}_2), equal(\bar{x}, \bar{x}_2), equal(\bar{x}_2, \bar{x}), equal(\bar{x}_1, \bar{x}), equal(\bar{x}_2, \bar{x}_1), equal(c, \bar{z}),$
$equal(\bar{z}, \bar{z}), equal(c, c)$

□ [4] Algorithm QED uses pruning to eliminate (1) from the output.

Shorthand notation: Before we proceed, let us formalize the shorthand notation introduced in Example 3.1. It is obvious that if the *equal* facts form an equivalence relation, the constants and frozen constants appearing in *equal* facts are divided in equivalence classes.

Let us look at the canonical DB of some query Q. If variables $X_1, \ldots, X_k$ appearing in the canonical DB belong to the same equivalence class, we replace all *equal* facts involving $X_1, \ldots, X_k$ by $equal(X_1, \ldots, X_k)$. For example, $equal(X_1, X_2, X_3)$ "stands for" all $equal(X_i, X_j), 1 \leq i, j \leq 3$.

The canonical DB produced by the query of Example 3.2 above can be written as

$$p(\bar{x}, \bar{x}_1), q(\bar{x}_2, \bar{y}, \bar{z}), equal(\bar{z}, c), equal(\bar{x}, \bar{x}_1, \bar{x}_2)$$

It is easy to see that

$$equal(Y_1, \ldots, Y_l) \text{ is a subset of } equal(X_1, \ldots, X_m) \text{ iff } \forall i \leq l, Y_i \in \{X_1, \ldots, X_k\}$$

Definition: Supporting Set of Fact Let h be an ordinary fact produced by an application of the p-Datalog rule

$$r : H \leftarrow G_1, \ldots, G_k, E_1, \ldots, E_m$$

of a p-Datalog description P on a database DB that consists of a canonical database CDB and other facts, and let μ be a mapping from the rule into the database such that $\mu(G_i), \mu(E_j) \in DB$ and $h = \mu(H)$. The set $\mathcal{S}_h$ of supporting facts of h, or *supporting set* of h, with respect to P, is the smallest set such that

- if $\mu(G_i) \in CDB$, then $\mu(G_i) \in \mathcal{S}_h$,
- if $\mu(G_i) \notin CDB$ and $\mathcal{S}'$ is the set of supporting facts of $\mu(G_i)$, then $\mathcal{S}' \subseteq \mathcal{S}_h$,
- if E is the set of all $\mu(E_i) \in \mathcal{S}_h$, then the smallest set of equality facts that includes E and is an equivalence relation is included in $\mathcal{S}_h$.

□

Let us notice that $\mathcal{S}_h$ is the set of leaves of a proof tree [Ull89] for h. We can further annotate the produced fact with the "id" of the rule used in its production, thus generating the whole proof tree for this fact.

Example 3.3 We can apply the rule

$$ans(X_1, Z_1) \leftarrow author(X_1, Z_1), publisher(Z_2, W), equal(Z_1, Z_2), equal(W, \$w)$$

on the following canonical DB

$$author(\bar{a}, \bar{b}), author(\bar{a}, \bar{a}), publisher(\bar{d}, \bar{f}), publisher(\bar{g}, \bar{h}), equal(\bar{b}, \bar{d}), equal(\bar{a}, \bar{g}),$$
$$equal(\bar{f}, \text{`}\texttt{PrenticeHall}\text{'})$$

to produce fact $ans(\bar{a}, \bar{b})$. The supporting set $\mathcal{S}$ is

$$\{author(\bar{a}, \bar{b}), publisher(\bar{d}, \bar{f}), equal(\bar{b}, \bar{d}), equal(\bar{f}, \text{`}\texttt{PrenticeHall}\text{'})\}$$

□

We next define the notions of *extended facts* and *extended canonical DB*:

Definition: Extended Facts and Extended Canonical DB An *extended fact* is a pair of the form $< h, \mathcal{S}_h >$, where h is a fact and $\mathcal{S}_h$ is the supporting set for h, with respect to some description P. Let Q be a rectified conjunctive query. The *extended canonical DB* of Q is a database of extended facts $< f, \{f\} >$, such that every f belongs in the canonical DB of Q. $\Box$

Referring to Example 3.3, the extended fact "associated" with our production of $ans(\bar{a}, \bar{b})$ is

$$< ans(\bar{a}, \bar{b}), \{author(\bar{a}, \bar{b}), publisher(\bar{d}, \bar{f}), equal(\bar{b}, \bar{d}), equal(\bar{f}, \text{`}\mathtt{PrenticeHall}')\} >$$

We now introduce the notion of the *corresponding query* for a fact, that makes our intuition about the supporting set explicit.

Definition: Corresponding Query Let $< h, \mathcal{S}_h >$ be an extended fact of the DB. Then, for every fact $g_i \in \mathcal{S}_h$, we can define a mapping ρ that is the identity on constants and predicate names and maps every frozen constant to the variable which it came from. It is easy to see that this mapping is well-formed. Moreover, it maps $\mathcal{S}_h$ into a query body and the fact h into a query head. The query Q:$\rho(h) \leftarrow \rho(g_1), \ldots, \rho(g_k)$ is called the *corresponding query* for extended fact $< h, \mathcal{S}_h >$. $\Box$

Intuitively, the corresponding query is an instantiated expansion of the rules of the description that can prove h and uses only source and equality predicates.

Algorithm QED produces a set of *candidate* queries: these are the corresponding queries to the produced extended facts. Candidate queries are described by the p-Datalog description; they are the only "interesting" expansions, in that they could be equivalent to the given query. As we will show later, each candidate query has an important property: its projection over the empty list of attributes contains the projection over the empty list of attributes of the given query Q. Said otherwise, the body of a candidate query contains the body of the given query. That means that if there exists a candidate query whose head is identical to the head of Q, then obviously this a containing query for Q with respect to P. Moreover, Q is expressible by P *iff* one of the candidate queries in the set is equivalent to Q.

The algorithm is presented in detail in Figure 3. Notice that the algorithm only generates *maximal* supporting sets for each produced fact. Therefore, the produced candidate queries are in a sense "minimal". We will formalize that notion later in this section.

We proceed to give results on the correctness and running time of the algorithm. Before that, let us just demonstrate with an example why rectification is necessary.

Example 3.5 To illustrate why rectification is necessary in identifying the candidate queries, let us consider the query $ans(X) \leftarrow p(X, c)$ and the p-Datalog description[5] $ans(A) \leftarrow p(A, B)$. Evaluating the description on the canonical DB $\{p(\bar{x}, c)\}$ (without rectification), would produce the extended fact $< ans(\bar{x}), \{p(\bar{x}, c)\} >$.

The corresponding query is

$$ans(X) \leftarrow p(X, c)$$

which is *not* a correct candidate query, because it is not expressible (by Definition 2.1) by the given description. If on the other hand we use rectification, we get the canonical DB $\{p(\bar{x}, \bar{y}), equal(\bar{y}, c)\}$. Evaluating the description on it, we get the candidate query

$$ans(X) \leftarrow p(X, Y)$$

which is a containing query for our given query (but not equivalent). $\Box$

[5]This is obviously the description of a source with a very simple query interface

Algorithm 3.4

Input

Minimized [Ull89] (non-rectified) conjunctive query Q of the form $H \leftarrow G_1, G_2, \ldots, G_k$, where the head subgoal H is of the form $ans(X_1, \ldots, X_n)$.
(non-rectified) p-Datalog description P.

Output

A set of candidate queries.

Method

Rectify P and Q
Construct the extended canonical DB of Q
Apply the rules of P to the facts in DB to generate *all possible extended facts* using bottom up evaluation [Ull89] modified in the following ways:

%items 1 and 2 guarantee the generation of extended facts
%with maximal supporting sets

1. populate IDB relations with extended facts, *i.e.*, if fact h is produced by the rule, compute $\mathcal{S}_h$ and then enter $< h, \mathcal{S}_h >$ in the database iff
   - $< h, \mathcal{S}_h >$ is not already in the database and
   - No $< h, \mathcal{S}'_h >$ where $\mathcal{S}_h \subseteq \mathcal{S}'_h$ is present in the DB.
2. when a new fact $< h, \mathcal{S}_h >$ is added to the DB, delete from the DB all facts of the form $< h, \mathcal{S}'_h >$, where $\mathcal{S}'_h \subset \mathcal{S}_h$.
3. if a rule is unsafe, *i.e.*, some distinguished variables do not appear in the rule body, simply leave those variables in the produced fact.

In the end:

4. if $< h, \mathcal{S}_h >$ is an extended fact, h is an ans fact and h contains variables, delete the extended fact.
5. de-rectify the resulting extended facts, and the query Q.
6. Create the corresponding queries of the extended facts.

□

The treatment of unsafe rules is the same as in generalized magic sets [Ull89].

Figure 3: *Algorithm QED*

Now we are ready to state some formal results about algorithm QED. We ultimately formally state and prove its correctness criterion (*i.e.*, solving the expressibility problem) and state and prove its computational complexity.

Lemma 3.6 Algorithm QED produces extended facts with maximal supporting sets.

By maximal, we mean that if $< h, \mathcal{S}_h >, < h, \mathcal{S}'_h >$ are two extended facts for the same fact h, it cannot be that $\mathcal{S}_h \subseteq \mathcal{S}'_h$ or that $\mathcal{S}'_h \subseteq \mathcal{S}_h$. Thus lemma 3.6 directly follows from Algorithm 3.4

Theorem 3.7 Soundness and Completeness of Set of Candidate Queries Let Q be a query, P be a p-Datalog description and $\{Q_i\}$ be the set of candidate queries that is the result of algorithm QED on Q and P. Then the following are true:

1. For all i, $\pi_\emptyset Q \subseteq \pi_\emptyset Q_i$.

2. For all i the identity mapping can map the body of Q_i to the body of Q.

3. If R is a query described by P and is not in $\{Q_i\}$ then

- $\pi_\emptyset R$ does not contain $\pi_\emptyset Q$ *or*
- there exists an i such that the heads of R and Q_i are *identical* and $Q_i \subseteq R$. Moreover, the identity mapping μ is a containment mapping from R to Q_i.

4. If R is a query described by P and is not in $\{Q_i\}$, $R \equiv Q$ only if there exists i such that $Q_i \equiv Q$.

Proof: (Sketch) **(2)** is derived directly from the Algorithm and **(1)** is a direct consequence of the existence of the mapping. For **(3)**: Algorithm QED is exhaustive, *i.e.*, it generates all "relevant" (in the sense of (1)) candidate queries, with the exception of those that are pruned due to Lemma 3.6. So let $R : Head_R \leftarrow Body_R$ be "relevant" and not in the candidate set. Then, for the extended fact[6] $< Head_R, Body_R >$, $Body_R$ is not a maximal supporting set. That means that there exists an extended fact $\mathcal{F} :< Head_R, \mathcal{S} >$ such that $Body_R \subseteq \mathcal{S}$. It is then clear from the definition of a corresponding query that the corresponding query Q_F to $\mathcal{F}$ is contained in R, and that the mapping from R to Q_F is the identity.
(4) is a direct consequence of (1) and (3). □

Theorem 3.7 says that any described query R that is not in the candidate set either is not equivalent to Q, or there already exists a "smaller" query Q_i in the candidate set that still "contains" Q. In the above sense, the candidate set contains "minimal" queries. Moreover, it says that queries in the candidate set are not "interesting": even if $R \equiv Q$, there is always a query Q_i in the candidate set that is also equivalent to Q.

Algorithm QED produces output that allows us to correctly decide query expressibility. To that effect, we prove the following:

Lemma 3.8 Expressibility Criterion Q is expressible by P iff the set of supporting facts for some extended fact $< h, \mathcal{S}_h >$ of the frozen head h of Q is identical[7] to the canonical DB for Q.

Proof: (Sketch)

IF: It is obvious from the way the "corresponding" query is defined, that if DB $\equiv \mathcal{S}_h$, then the corresponding query is equivalent to Q.

ONLY IF: The output of algorithm QED contains candidate queries for which Theorem 3.7 holds, *i.e.*, there is no expansion that is a "tighter fit" to the given query than the queries in the output. If for every $\mathcal{S}_h$, there exists some fact in the canonical DB that is not in that $\mathcal{S}_h$ set, then the corresponding query cannot be equivalent to Q. The reason for that is that Q is minimized, and minimization is unique up to isomorphism, so all subgoals (*i.e.*, all facts in the canonical DB) are necessary. □

The number of extended facts that can be generated per "real" fact is equal to the number of different maxinal supporting sets for the fact, *i.e.*, it is exponential in the size of the canonical DB. The number of facts is exponential in the size of the description, so we have the following:

Theorem 3.9 Algorithm QED produces an answer in time exponential to the size of the description and the size of the query.

Finally, let us notice that the problem of query containment in Datalog is reducible to the problem of query expressibility described here. Query containment in Datalog is EXPTIME-complete [RSUV89]. Hence we have the following:

Theorem 3.10 Query expressibility is EXPTIME-complete.
Therefore, Algorithm 3.4 meets the theoretical lower bound.

[6] Where $Head, Body$ are "frozen".
[7] After de-rectification of both.

3.1 Expressibility and translation

Let us consider the case of a wrapper that receives a query. It is easy to see that we could extend Algorithm 3.4 so that it annotates each fact not only with its supporting set, but also with its proof tree. The wrapper then can use the parse tree to perform the actual translation of the user query in source-specific queries and commands, by applying the translating *actions* that are associated with each rule of the description [PGGMU95, JBHM+97].

4 Answering Queries Using p-Datalog Descriptions

Mediators are faced with a tougher problem than wrappers: Given the descriptions for one or more wrappers, the mediator has to answer the user query by sending to the wrappers only queries expressible by the wrapper descriptions and consequently combine the answers to produce the answer to the given query. This is the Capabilities-Based Rewriting (CBR) problem [PGH96, HKWY96]. Notice that the mediator can combine queries using only selections, projections, and joins. Formally, it considers *rewritings* of the user query that are conjunctive rules, as described below.

Definition: Rewriting of Query Given a conjunctive query Q and a set of queries $\{Q_1, \ldots, Q_n\}$, of the form

$$ans_i \leftarrow body_i \ , i = 1, \ldots, n$$

a *rewriting* of Q using $\{Q_i\}$ is a rule Q' of the form

$$ans \leftarrow ans_1, \ldots, ans_n, optional\ \ equalities$$

such that $Q' \equiv Q$ □

As we have said in previous sections, a source description defines the (possibly infinite) set of conjunctive queries answerable by the source. So, the CBR problem is equivalent to the problem of answering the user query using an infinite set of views described by a Datalog program [LRU96].

Our algorithm proceeds in two steps. The first step finds a finite set of expansions. The second step uses an algorithm for answering queries using views [LMSS95, Qia96] to combine some of these expansions to answer the query. The first step uses the Algorithm 3.4 to generate a finite set of expansions (see Figure 3). We prove that if we can answer the query using any combination of expressible queries, then we can answer it using a combination of expansions in our finite set. In [LRU96], a solution is presented for the problem whose complexity is non-deterministic doubly exponential in the size of the query and the description. The solution is based on "signatures" for the expansions of the description, that divide the queries that are expressible by the description into equivalence classes. We will show that our solution is non-deterministic exponential in the size of the query and the description. Moreover, the proof of our solution is more intuitive and simpler.

Given a user query Q and a wrapper description P in p-Datalog, Algorithm QED produces all[8] the candidate queries of Q with respect to P. We can show that there is at most an exponential number of those:

Lemma 4.1 The output of Algorithm 3.4 contains at worst an exponential number of queries, whose length is at most linear to the size of the given user query.

Moreover, we can prove that these are the only queries expressible[9] by P that are "relevant" in answering Q.

[8] modulo variable renaming

[9] The corresponding queries Q_i, that are the output of Algorithm 3.4, actually are *described by* P.

Theorem 4.2 (**CBR**) Assume we have a query Q and a p-Datalog description P without tokens, and let $\{Q_i\}$ be the result of applying Algorithm 3.4 on Q and P. There exists a rewriting Q' of Q, such that $Q' \equiv Q$, using any $\{Q_j | Q_j$ is expressible by $P\}$ *if and only if* there exists a rewriting Q'' , such that $Q'' \equiv Q$, using only $\{Q_i\}$.

Proof: (Sketch) The *if* direction is trivial. For the *only if*: It must be that $\pi_\emptyset(Q) \subseteq \pi_\emptyset(Q_j)$ [LMSS95]. Since Q_j is expressible by P, Q_j could be a candidate query. But $\{Q_i\}$ contains all the "interesting" candidate queries of Q with respect to P by Theorem 3.7. This means that for any Q_j, either $Q_j \in \{Q_i\}$ or there exists some "corresponding" Q_i such that $Q_i \subseteq Q_j$, and the containment mapping from Q_j to Q_i is the identity mapping. Let Q':$Q_{j_1}, \ldots, Q_{j_k}, \ldots, Q_{j_m}$ be the rewritten query. If we replace each Q_{j_k} with its "corresponding" Q_{i_k} identified above, then Q'':$Q_{i_1}, \ldots, Q_{i_m}$ is also equivalent to Q. In proof:

- there exists a containment mapping from Q'' to Q. In particular, the identity mapping is a containment mapping from Q'' to Q
- there exists a containment mapping from Q to Q' and from Q' to Q'', and therefore also from Q to Q''.

Therefore, by the containment mapping theorem [CM77], Q'' and Q are equivalent. That completes the proof of the theorem. $\Box$

Now that we are sure that all we need to solve the rewriting problem is to compute the candidate queries (using Algorithm 3.4) we need an algorithm to combine some of the candidate queries into a rewriting of the given query. The problem of finding an equivalent rewriting of a query using a finite number of views is known to be NP-complete in the size of the query and the view set [LMSS95] and there are known algorithms for solving it in the absence of tokens [LMSS95, Qia96]. Hence, the total computational complexity of our CBR scheme in the worst case is

- First stage (QED): Exponential in the size of the query and the description.
- Second stage (answering queries using views): NP in the size of its input. The size of the input is the cardinality of the candidate set *times* the size of the largest candidate.

Since the QED algorithm has output of exponential size, the second stage dominates and the total complexity of the algorithm in the worst case is nondeterministic exponential. In particular, the cardinality of the candidate set is exponential in the arity of the head of the candidate queries *and*, more importantly, in the size of the canonical database. (See also subsection 5.2.)

4.1 CBR with binding requirements

The discussion in the previous section ignores the presence of tokens. To handle tokens in the p-Datalog description, we need to modify both steps of our CBR scheme. Let us discuss what changes are necessary.

To correctly solve the CBR problem in the presence of binding requirements, we first of all need to modify the QED algorithm. Let us consider an example that will show that algorithm QED, if used unchanged, is inadequate for the solution of the CBR problem with binding patterns.

Example 4.3 Let the "target" query be

$$Q : ans(X) \leftarrow p(c, Y), p(Y, X)$$

and let the description be

$$V : v(X) \leftarrow p(\$c, X)$$

The rectified query is

$$Q : ans(X) \leftarrow p(A, Y), p(Y_1, X), equal(A, c), equal(Y, Y_1)$$

The rectified p-Datalog description of the source is

$$V : ans(W) \leftarrow p(B, W), equal(B, \$c)$$

Algorithm QED produces the following candidate query (after de-rectification):

$$C : ans(Y) \leftarrow p(c, Y)$$

There is no rewriting of Q using only C that is equivalent to Q. But there is a way to answer Q using our p-Datalog description. To see that, let us rewrite the query and the view to make the binding patterns explicit:

$$\begin{array}{lll} Q: & ans^{fb}(X, A) & \leftarrow p(A, Y), p(Y, X) \\ V: & v^{fb}(X, A) & \leftarrow p(A, X) \end{array}$$

Then we can rewrite Q as follows:

$$Q : ans^{fb}(X, A) \leftarrow v^{fb}(Y, A), v^{fb}(X, Y)$$

This rewriting respects the binding requirements of the views, is processed by passing Y bindings, and is equivalent to the target query. □

Therefore, we need to modify algorithm QED. The necessary change over QED consists basically of a pre-processing step: replace tokens in the p-Datalog description with variables, but maintain as an *extra annotation* the information that these variables need to be bound. In particular, that information can be attached to each extended fact as an extra annotation. The modified algorithm *QED-T* is presented in detail in Figure 4.

Applying that modification to the previous example, V becomes

$$V : ans(W) \leftarrow p(B', W)$$

where B' needs to be bound. Algorithm QED-T on this input produces two candidate queries:

$$C' : ans(X) \leftarrow p(Y_1, X)$$

where Y_1 needs to be bound, and

$$C'' : ans(Y) \leftarrow p(A, Y)$$

where A needs to be bound. Finally, QED-T uses the binding information to turn the candidate queries into queries *with binding patterns*. So, C', C'' turn into

$$C' : ans^{fb}(X, Y_1) \leftarrow p(Y_1, X)$$

and

$$C'' : ans^{fb}(Y, A) \leftarrow p(A, Y)$$

Algorithm 4.4

Input

Minimized [Ull89] (non-rectified) conjunctive query Q of the form $H \leftarrow G_1, G_2, \ldots, G_k$, where the head subgoal H is of the form $ans(X_1, \ldots, X_n)$.
(non-rectified) p-Datalog description P.

Output

A set of candidate queries *with binding patterns.*

Method

Rectify P and Q
Construct the extended canonical DB of Q
Replace tokens in P with variables. Annotate rules with binding information.
Apply the rules of P to the facts in DB to generate *all possible extended facts* using bottom up evaluation [Ull89] modified in the following ways:

1. populate IDB relations with extended facts, *i.e.*, if fact h is produced by the rule, compute $\mathcal{S}_h$ and then enter $< h, \mathcal{S}_h >$ in the database iff
   - $< h, \mathcal{S}_h >$ is not already in the database and
   - No $< h, \mathcal{S}'_h >$ where $\mathcal{S}_h \subseteq \mathcal{S}'_h$ is present in the DB.
2. when a new fact $< h, \mathcal{S}_h >$ is added to the DB, delete from the DB all facts of the form $< h, \mathcal{S}'_h >$, where $\mathcal{S}'_h \subset \mathcal{S}_h$.
3. if a rule is unsafe, *i.e.*, some distinguished variables do not appear in the rule body, simply leave those variables in the produced fact.
4. *Update bound variables annotation for the extended fact:* *A variable gets an annotation when it binds to an already annotated variable.*

In the end:

5. if $< h, \mathcal{S}_h >$ is an extended fact, h is an ans fact and h contains variables, delete the extended fact.
6. de-rectify the resulting extended facts, and the query Q.
7. Create the corresponding queries of the extended facts. *Use the binding information to construct their binding patterns.*

$\square$

The treatment of unsafe rules is the same as in generalized magic sets [Ull89].

Figure 4: *Algorithm QED-T*

C and C' together with Q are the input to the second stage of our CBR scheme, which per Section 4 is an algorithm for answering queries using views. The algorithms [LMSS95, Qia96] proposed in the previous section do *not* deal properly with tokens. As we have mentioned in Section 2, tokens describe binding requirements. Therefore, we need to take into account the binding requirements of candidate queries. [RSU95] studies the problem of answering queries using views with binding requirements. The authors use binding patterns to describe binding requirements. They show that the problem is NP-complete and they also describe an algorithm for it. The algorithm takes as input a finite set of conjunctive views with binding patterns and a "target" query with a binding pattern and rewrites the query using the views in a way that respects the view binding patterns. Example 4.3 is an example of query rewriting using views with binding patterns.

We use this algorithm, henceforth referred to as the *AnsBind* algorithm, for the second part of our CBR scheme, that is, to find a rewriting of the user query using the candidate queries. Using Q, C', C'' as input to *AnsBind*, we obtain the correct and efficient rewriting of Q that is shown in

Example 4.3.

Theorem 4.5 (**CBR-tokens**) Assume we have a query Q and a p-Datalog description P *with tokens*, and let $\{Q_i\}$ be the result of applying Algorithm 4.4 on Q and P. There exists a rewriting Q' of Q, such that $Q' \equiv Q$, using any $\{Q_j | Q_j$ is expressible by $P\}$ *if and only if* there exists a rewriting Q'' , such that $Q'' \equiv Q$, using only $\{Q_i\}$.

Proof: (Sketch) The only issue is that QED-T is "missing" some candidate queries by "ignoring" tokens. But it is easy to see that any candidate query we are thus "missing" is identical to one of the queries in the candidate set of QED-T, *modulo equality subgoals*. Moreover, if there is a rewriting of a query using some candidate Q_i *with* some binding pattern, then there is also a rewriting of the query using Q_i without a binding pattern. The theorem then follows. □

The solution for the CBR problem with binding requirements is also non-deterministic exponential.

5 An interesting and more efficient class of p-Datalog descriptions

We identify an interesting class of p-Datalog descriptions with a simple syntactic characterization, for which the CBR algorithm of Section 4 is much more efficient. In particular, for this class of descriptions the output of the QED algorithm is only exponential in the arity of the candidate query head, and does *not* depend on the size of the canonical database. Hence, the second stage of the CBR scheme is more efficient, since it receives smaller input. Overall, the CBR scheme for this class is non-deterministic exponential in the arity of the head predicate.

Definition: A p-Datalog description P belongs in $\mathcal{P}_{loop}$ if and only if

- P contains only one IDB predicate
- If p is the IDB predicate and

  $$R : p(X_1, \ldots, X_n) \leftarrow pred_1(A_{11}, \ldots, A_{1m_1}) \ldots, p(Y_1, \ldots, Y_n), \ldots, pred_k(A_{k1}, \ldots, A_{km_k})$$

  is any rule where p appears, Y_i is actually X_i for all i.

□

Descriptions in $\mathcal{P}_{loop}$ therefore consist of simple loops and exit rules.

Example 5.1 Let us repeat the description of the source of Example 2.3. The source accepts queries on books given one or more words from their abstracts, assuming there exists an abstract index $abstract\_index(abstract\_word, Id)$ The following p-Datalog program is used to describe this source.

$$\begin{array}{ll} ans(Id, Aut, Titl, Pub, Yr, Pg) & \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), ind(Id) \\ ind(Id) & \leftarrow abstract\_index(\$c, Id) \\ ind(Id) & \leftarrow ind(Id), abstract\_index(\$c, Id) \end{array}$$

The above description clearly belongs in $\mathcal{P}_{loop}$.[10] □

We use *lattices* to help explain why the output of QED on descriptions in $\mathcal{P}_{loop}$ does not depend on the size of the canonical database but it solely depends on the arity of the *ans* facts. The next subsection is a short reminder about lattices.

[10]The description also happens to be monadic[AHV95]. Descriptions in $\mathcal{P}_{loop}$ in general don't have to be.

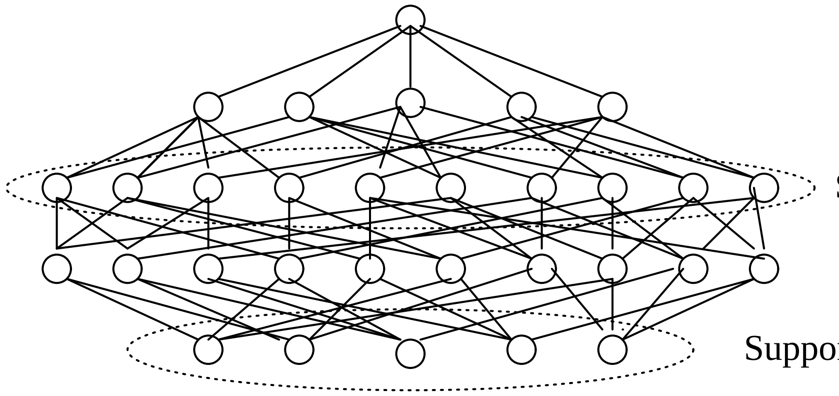

Figure 5: Supporting set lattice for fact f for a database of size 5

5.1 Lattice Framework

Let us consider the subset relation $\subseteq$ between sets.

We denote a lattice with set of elements (supporting sets in this section) L and the subset relation $\subseteq$ by $\langle L, \subseteq \rangle$. For elements a and b of a lattice $\langle L, \subseteq \rangle$, $a \subset b$ means that $a \subseteq b$ and $a \neq b$.

The ancestors and descendants of an element of a lattice $\langle L, \subseteq \rangle$, are defined as follows:

$$ancestor(a) = \{b \mid a \subseteq b\}$$

$$descendant(a) = \{b \mid b \subseteq a\}$$

Note that every element of the lattice is its own descendant and its own ancestor. The immediate proper ancestors of a given element a in the lattice belong to a set we shall call $next(a)$. Formally,

$$next(a) = \{b \mid a \subset b, \nexists c, a \subset c, c \subset b\}$$

It is common to represent a lattice by a *lattice diagram*, a graph in which the lattice elements are nodes and there is an edge from a below to b above if and only if b is in $next(a)$. Thus, for any two lattice elements x and y, the lattice diagram has a path downward from y to x if and only if $x \subseteq y$.

Figure 5 shows the lattice diagram for the *possible* supporting sets of a fact f for a database of size 5. The next subsection discusses the size of the output of the QED algorithm for the $\mathcal{P}_{loop}$ class of p-Datalog descriptions.

5.2 QED and $\mathcal{P}_{loop}$

The cardinality of the candidate set produced by QED can in general be exponential in the size of the canonical database. Figure 5 gives a graphical explanation for the potential exponentiality of supporting sets of even fixed size for a fact f. Therefore, the number of candidate queries can also be exponential in the size of the canonical database.

For descriptions in $\mathcal{P}_{loop}$, let us make the following crucial observation: Let $\mathcal{S}_i$ and $\mathcal{S}_j$ be two supporting sets for fact f that are produced by algorithm QED with a description P that is in $\mathcal{P}_{loop}$. Let $\mathcal{S}$ be their least common ancestor, as in Figure 6. Then, $\mathcal{S}$ is also produced by QED for f. Since QED only keeps extended facts with *maximal* supporting sets, the extended fact $< f, \mathcal{S} >$ will be kept for f, and it will replace the extended facts $< f, \mathcal{S}_i >$ and $< f, \mathcal{S}_j >$.

Thus, it is easy to see that only one extended fact per fact f will be generated, and therefore just one candidate query. Therefore, the output of the QED algorithm for $\mathcal{P}_{loop}$, and thus the

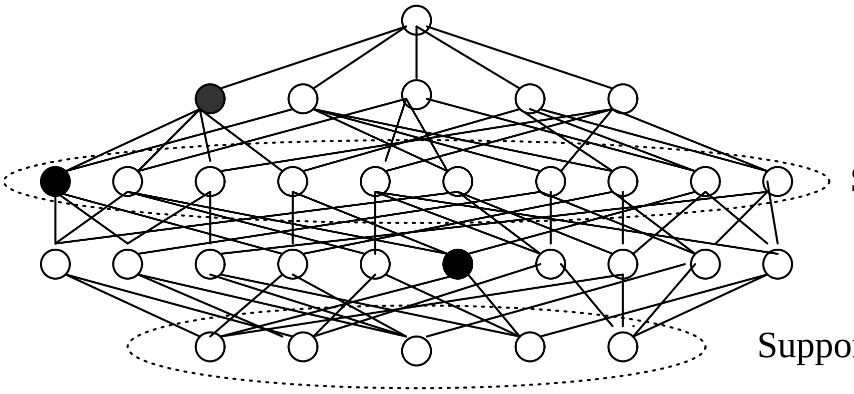

Figure 6: Supporting sets and least common ancestor

complexity of the second stage of the CBR scheme, is only exponential in the arity of the head of the candidate queries, and not in the size of the canonical database.

The importance of the class lies in the fact that we have observed that it is expressive enough to describe a large number of common sources, such as document retrieval systems and Web-based sources.

6 Expressive Power of p-Datalog

We have illustrated the use of p-Datalog programs as a source description language. In this section, we explore some limits of its description capabilities. It should be noted that although we focus here on the description of conjunctive queries, similar results hold when negation and disjunction are introduced.

Clearly, there are sets of conjunctive queries that cannot be described by any p-Datalog description. Moreover:

Lemma 6.1 There exist *recursive* sets of conjunctive queries that are not expressible by any p-Datalog description.

Proof: As we have seen in the previous section, the decision procedure for the description semantics of p-Datalog is exponential. Therefore, any recursive set of conjunctive queries with a membership function that is super-exponential is not expressible by any p-Datalog description. □

However, the practical question is whether there are recursive sets of conjunctive queries, that correspond to "real" sources, and cannot be expressed by p-Datalog programs. We show next that some common sources (intuitively the "powerful" ones) exhibit this behavior. Before we prove this result, we demonstrate the expressive abilities and limitations of p-Datalog.

Let us start with an observation: For every p-Datalog description program P, the arity of the result is exactly the arity of the *ans* predicate. This restriction is somewhat artificial, since we can define descriptions with more than one "answer" predicate. However, even in that case, a given program would still bound the arities of answers. Furthermore, a more serious bound is the number of variables that occur in any one of the rules of the program. We will see that this bound is imposing severe restrictions on the queries that can be expressed.

But first, if we bound the number of variables, we can show the following:

Theorem 6.2 Let k be some integer. Let $p_1, \ldots, p_m$ be the EDB predicates of a database. There exists a p-Datalog program P that describes all conjunctive queries with at most k variables[11] on this database.

Proof: (Sketch) We show the construction for $k = 3$ and for the case where $p_1, \ldots, p_m$ are each predicates of arity two. The program P that can describe all conjunctive queries is the following:

$$\begin{array}{rcll}
ans_3(X_i, X_j, X_l) & \leftarrow & temp(X_1, \ldots, X_k), \forall i, j, l \leq k & (3) \\
ans_2(X_i, X_j) & \leftarrow & ans_3(X_i, X_j, X_j), \forall i, j \leq k & (4) \\
ans_1(X_i) & \leftarrow & ans_2(X_i, X_i), \forall i \leq k & (5) \\
ans_0() & \leftarrow & ans_1(X) & (6) \\
temp(X_1, \ldots, X_k) & \leftarrow & p_l(X_i, X_j), temp(X_1, \ldots, X_k) \;\; \forall l \leq m, \; \forall i, j \leq k & (7) \\
temp(X_1, \ldots, X_k) & \leftarrow & p_l(X_i, \$c), temp(X_1, \ldots, X_k) \;\; \forall l \leq m, \; \forall i \leq k & (8) \\
temp(X_1, \ldots, X_k) & \leftarrow & p_l(\$c, X_j), temp(X_1, \ldots, X_k) \;\; \forall l \leq m, \; \forall j \leq k & (9) \\
temp(X_1, \ldots, X_k) & \leftarrow & p_l(\$c_1, \$c_2), temp(X_1, \ldots, X_k) \;\; \forall l \leq m & (10) \\
temp(X_1, \ldots, X_k) & \leftarrow & \epsilon & (11)
\end{array}$$

where $X_1, \ldots, X_k$ are distinct variables. It is easy to see that a similar construction can provide the program that describes all conjunctive queries for $k > 3$ and larger arities. □

As mentioned above, a fixed p-Datalog program bounds the arity of the results, but this bound is not the *only* cause of limitation. Even if we focus on arity-0 results, *i.e.*, queries that answer yes or no and do not provide data, p-Datalog is limited. The limitation is related to the number of variables. Let FO^k be the set of sentences of first order logic [AHV95] with at most k variables. Note that the same variable can be "reused" as much as wanted using quantification. The following relates the queries described by a p-Datalog program to formulas expressible in first-order logic with a bounded number of variables. It states that although one such query may use an arbitrary number of variables, with appropriate "reuse" only a bounded number of variables suffice.

Lemma 6.3 Let P be a Datalog program and k the maximum number of variables occurring in a rule of P. Then for each Q expressible by P, Q is equivalent to a query in FO^k (using only $\wedge$ and $\exists$).

Proof: (Sketch) Let $x_1, \ldots, x_k$ be the variables appearing in the rules of description P. Also, let

$$Q' : ans(u_1) \leftarrow p_1(u_2), p_2(u_3), \ldots, p_n(u_n)$$

be in $descr(P)$ such that $Q \equiv Q'$. We will show that Q' is equivalent to a first order sentence with only k variables.

The proof is by induction on the number of resolution steps used to construct a rule. If Q' is a rule of P, then the claim is true. Otherwise, when doing a step of the resolution, let q_i be the literal that is unified with some rule head. Then, the variables not used in q_i can be reused existentially quantified for the extra variables in the rule. □

The limitation on the number of variables of the program prohibits the description of the set of all conjunctive queries over a schema — a set that is supported by common powerful sources.

[11]We disregard repeated variables in the head of the conjunctive queries, so we assume that the result predicate has arity at most k.

Theorem 6.4 Let the database schema $\mathbf{S}$ have a relation of arity at least two. For every p-Datalog description P over $\mathbf{S}$, there exists a boolean query Q over $\mathbf{S}$, such that Q is not expressible by P. (So, in particular, there is no p-Datalog description that could describe a source that can answer all conjunctive queries, even if we fix the arity of the answer.)

In order to prove this, we first need to prove the following lemma:

Lemma 6.5 Let a database consist of a binary relation G that contains no self loops. The question "is there a k-clique in G" can be expressed by a conjunctive query (with k variables) but is not in FO^{k-1}.

Proof: (Sketch) The question is clearly expressed by the following query:

$$\begin{array}{rcl} ans() & \leftarrow & G(x_1,x_2),\ldots,G(x_1,x_k),\ldots, \\ & & G(x_i,x_1),\ldots,G(x_i,x_{i-1}),G(x_i,x_{i+1}),\ldots,G(x_i,x_k) \\ & & G(x_k,x_1),\ldots,G(x_k,x_{k-1}) \end{array}$$

This query cannot be expressed [AHV95] by an FO^{k-1} formula, as can be shown by playing an Ehrenfeucht-Fraissé game (see [AHV95]), on the following two structures: G_1, a k-clique without self-loop and G_2, a k-1 clique without self-loop. $\Box$

Now we are ready to prove Theorem 6.4.

Proof: Let $\mathbf{S}$ (without loss of generality) contain the binary predicate G. Suppose such a description P exists. Let k be the maximum number of variables in a rule of P. Then each conjunctive query expressible with P is in FO^k by Lemma 6.3. But then the $k+1$ clique without self-loop is not in P. $\Box$

The theorem 6.4 points out a rather serious limitation of p-Datalog descriptions.

7 The RQDL description Language

Given the limitations of p-Datalog for the description of powerful information sources, we are proposing the use of a more powerful query description language. RQDL (Relational Query Description Language) is a Datalog-based rule language used for the description of query capabilities. It was first proposed in [PGH96] and used for describing query capabilities of information sources. [PGH96] shows its advantages over Datalog when it is used for descriptions that are not schema specific, *i.e.*, the description does not refer to specific relations or arities in the schema of the specific source. In this way the descriptions are more concise and they gracefully handle schema evolution.

In this paper we present a formal specification of *extended-RQDL*, which provably allows us to describe large sets of queries. For example, we can prove that the extended-RQDL (from now on, we will by default refer to the extended-RQDL as RQDL), unlike p-Datalog, can describe the set of all conjunctive queries. Furthermore, we reduce RQDL descriptions to terminating p-Datalog programs with function symbols. Consequently, the decision on whether a given conjunctive query is expressed by an RQDL description is reduced to deciding expressibility of the query by the resulting p-Datalog program.

Note, the reduction of RQDL to Datalog with function symbols is important because

- It reduces the comparison between the expressive power of p-Datalog and RQDL to a comparison between Datalog and Datalog with function symbols.
- It reduces the decision procedure for expressibility to Algorithm 3.4. That allows us to give a complete solution to the CBR problem for RQDL.

Subsections 7.1 and 7.2 demonstrate the use of RQDL for the description of source capabilities and define the syntax and semantics of RQDL. Section 9 describes the reduction of RQDL descriptions to p-Datalog programs with function symbols and Section 10 proceeds to give algorithms for query expressibility by RQDL description and for the CBR problem for RQDL descriptions.

7.1 Using RQDL for query description

To support schema independent descriptions, RQDL allows the use of *predicate tokens*[12] in place of the relation names. Furthermore, to allow tables of arbitrary arity and column names, RQDL provides special variables called *vector variables*, or simply vectors, that match with sets of relation attributes that appear in a query. Vectors can "stand for" arbitrarily large sets of attributes. It is this property that eventually allows the description of large, interesting sets of conjunctive queries (like the set of all conjunctive queries).

Example 7.1 illustrates RQDL's ability to describe source capabilities without referring to a specific schema. Example 7.2 demonstrates an RQDL program that describes all conjunctive queries over any schema. Subsection 7.2 describes the formal syntax and semantics of RQDL. Before we go ahead with the examples, let us introduce some notation.

Named Attributes in Conjunctive Queries: For notational convenience, we slightly modify the query syntax so that we can refer to the components of tuples by attribute names instead of column numbers. For example, consider the relation *book* with schema *book*(*title*, *isbn*). We will write *book* subgoals by explicitly mentioning the attribute names; instead of writing

$$ans() \leftarrow book(X, Z), equal(X, \texttt{DataMarts})$$

we will write

$$ans() \leftarrow book(title : X, isbn : Z), equal(X, \texttt{DataMarts})$$

We will be using named attributes in the rest of this paper. Every predicate will then have a *set* of named attributes (and not a list of attributes). The connection of this scheme to SQL syntax is evident.

Example 7.1 Consider a source that accepts queries that refer to exactly one relation and pose exactly one selection condition over the source schema.

$$ans() \leftarrow \$r(\vec{V}), item(\vec{V}, \$a, X'), equal(X', \$c)$$

The above RQDL description[13] describes, among others, the query

$$ans() \leftarrow books(title : X, isbn : Z), equal(X, \texttt{DataMarts})$$

because, intuitively, we can map $\$r$ to relation *books*, $\vec{V}$ to the set of attribute-variable pairs $\{title : X, isbn : Z\}$, X' to X, and $\$c$ to *DataMarts*. The *metapredicate* $item(\vec{V}, \$a, X')$ declares that the variable X' maps to one of the variables in the set of attribute-variable pairs that $\vec{V}$ is mapped to, *i.e.*, X' maps to one of the variables of the subgoal $\$r$. The token $\$a$ maps to the attribute name of the variable X' in $\vec{V}$. $\$a$ can map to any of the attribute names and hence X' can map to either X or Z.

[12]Predicate tokens belong to the same sort as tokens.

[13]Notice that both the RQDL descriptions and the queries are rectified.

RQDL descriptions do not have to be completely schema independent. For example, let us assume that we can put a selection condition only on the title attribute of the relation. Then we modify the above RQDL description as follows:

$$ans() \leftarrow \$r(\vec{V}), item(\vec{V}, name, X'), equal(X', \$c) \tag{12}$$

The replacement of $\$a$ by *name* forces the selection condition to refer to the *name* attribute only. □

Next we present the RQDL description P_{CQ} that describes all conjunctive queries over any schema.

Example 7.2

$$\begin{array}{lll}
(i) & ans(\vec{V}_1) & \leftarrow cond(\vec{V}), \vec{V}_1 \subseteq \vec{V} \\
(ii) & cond(\vec{V}) & \leftarrow \$p(\vec{V}_1), cond(\vec{V}_2), \vec{V} = \vec{V}_1 \cup \vec{V}_2 \\
(iii) & cond(\vec{V}) & \leftarrow item(\vec{V}, \$a, X), equal(X, \$c), cond(\vec{V}) \\
(iv) & cond(\vec{V}) & \leftarrow item(\vec{V}, \$a_1, X_1), item(\vec{V}, \$a_2, X_2), equal(X_1, X_2), cond(\vec{V}) \\
(v) & cond(\vec{V}) & \leftarrow \$p(\vec{V})
\end{array}$$

Given any rectified conjunctive query (without arithmetic), the description above describes it. Each rule deals with a particular capability: The first rule describes arbitrary projection capabilities over any "condition". The third rule describes a selection on an attribute of a condition. The fourth rule describes a join over one variable. The second rule "augments" conditions as necessary with new literals, to create conditions that are conjunctions of predicates of arbitrary length. The **union** *metapredicate* "carries" of the attribute list of the augmented condition. □

7.2 Formal Syntax and Semantics of RQDL

The full syntax of RQDL appears in Appendix A, Fig. 8. An RQDL description is a finite set of RQDL rules. The description semantics of RQDL are a generalization of the description semantics of p-Datalog, to account for the existence of vectors and metapredicates. We start by defining what is an *expansion* of an RQDL description.

Definition: Let P be an RQDL description with a particular IDB predicate *ans*. The set of *expansions* $\mathcal{E}_P$ of P is the smallest set of rules such that:

- each rule of P that has *ans* as the head predicate is in $\mathcal{E}_P$;
- if r_1: $p \leftarrow q_1, \ldots, q_n$ is in $\mathcal{E}_P$, r_2: $r \leftarrow s_1, \ldots, s_m$ is in P, and a substitution θ is the most general unifier of some q_i and r then the resolvent

  $$\theta p \leftarrow \theta q_1, \ldots \theta q_{i-1}, \theta s_1, \ldots, \theta s_m, \theta q_{i+1}, \ldots, q_n$$

  of R_1 with R_2 using θ is in $\mathcal{E}_P$.

□

Unification: Unification extends to vectors in the following way:

1. a vector can unify with another vector, yielding a vector;
2. a vector can unify with a set consisting of attribute-variable pairs, yielding that set; for example $p(\vec{V})$ can unify with $p(attr_1 : X, attr_2 : Y)$ yielding

   $$p(attr_1 : X, attr_2 : Y)$$

Metapredicates: There are three *metapredicates*, and their argument list has to be of a specific type: We define

$$union(\vec{V}, \vec{V_1}, \vec{V_2}) \text{ to mean } \vec{V} = \vec{V_1} \cup \vec{V_2} \tag{13}$$

where $\vec{V}$ is a vector and $\vec{V_1}, \vec{V_2}$ can be vectors, or sets of attribute-variable pairs. We also define

$$item(\vec{V}, \$a, X) \text{ to mean } \vec{V}[\$a] \equiv X, \tag{14}$$

and

$$item(\vec{V}, a, X) \text{ to mean } \vec{V}[a] \equiv X \tag{15}$$

which means that the variable X belongs to the set of attribute-variable pairs that $\vec{V}$ maps to, with attribute name $\$a$ (or a). a is a constant. $\$a$ is a token. X is a variable. $\vec{V}$ can be a vector or a set of attribute-variable pairs. Finally, we define

$$subset(\vec{V}, \vec{V_1}) \text{ to mean } \vec{V} \subseteq \vec{V_1} \tag{16}$$

where $\vec{V}$ and $\vec{V_1}$ can be vectors or sets of attribute-variable pairs $\vec{V}$ can only appear in the head of the rule (in addition to the *subset* subgoal). The intuition behind *subset* is that it allows us to do arbitrary projections.

We call a metapredicate that does not contain any vectors *ground*.

Safety: Metapredicates must observe some binding pattern constraints. In particular, all vectors that appear in metapredicates must be *safe* as defined below:

- If a vector appears in an EDB or IDB subgoal then it is safe.
- If a vector $\vec{V}$ appears in a subgoal $union(\vec{V}, \vec{V_1}, \vec{V_2})$ and $\vec{V_1}$ and $\vec{V_2}$ are safe, then $\vec{V}$ is also safe.
- If a vector $\vec{V}$ appears in a subgoal $subset(\vec{V}, \vec{V_1})$ and $\vec{V_1}$ is safe, then $\vec{V}$ is also safe.

Following the definition of description semantics of Section 2, we now define the description semantics of RQDL.

Definition: Set of Queries Described by an RQDL Program The set of *terminal expansions* $\mathcal{T}_P$ of P is the subset of all expansions $e \in \mathcal{E}_P$ containing only EDB predicates or predicate tokens in the body. A *valid* terminal expansion is a terminal expansion where all ground metapredicates evaluate to `true`

The set of *instantiated terminal expansions* $\mathcal{I}_\mathcal{P}$ of RQDL description P is the set of all (rectified) conjunctive queries $\tau(r)$, where r belongs to the set of terminal expansions of P and τ is a mapping of the RQDL rule r to a conjunctive query, that:

1. maps every token $\$c$ to a constant c. (Note, we consider relation names to be of constant type.)
2. maps every vector $\vec{V}$ to a set of attribute-variable pairs $\{(a_1 : X_1), \ldots, (a_n : X_n)\}$ such that
   (a) after we replace every predicate subgoal $p(\vec{V})$ with $p(a_1 : X_1, \ldots, a_n : X_n)$ no variable appears in more than one predicate subgoals,

(b) for every subgoal of the form $union(\vec{V}, \vec{V_1}, \vec{V_2})$, $\tau(\vec{V}) = \tau(\vec{V_1}) \cup \tau(\vec{V_2})$,

(c) for every subgoal of the form $item(\vec{V}, a, X)$, $\tau(\vec{V})$ includes a pair $(a : X)$,

(d) for every subgoal of the form $item(\vec{V}, \$a, X)$, $\tau(\vec{V})$ includes a pair (a, X), for some a,

(e) for every subgoal of the form $subset(\vec{V}, \vec{V_1})$, τ maps $\vec{V}$ to a subset of $\tau(\vec{V_1})$.

3. and drops all metapredicate subgoals.

The set of *described* queries of an RQDL description P with "designated" predicate ans (when ans is understood), is the set of safe instantiated terminal expansions of P. □

Example 7.3 Let us refer to the RQDL description P_{CQ} of Example 7.2. The RQDL rule

$$\begin{aligned} R : ans(\vec{V'}) \leftarrow \quad & \$p_1(\vec{V}_1), \$p_2(\vec{V}_2), union(\vec{V}, \vec{V}_1, \vec{V}_2), item(\vec{V}, \$a_1, X_1), item(\vec{V}, \$a_2, X_2), \\ & equal(X_1, X_2), subset(\vec{V'}, \vec{V}) \end{aligned}$$

is a terminal expansion of that RQDL description. In particular, this rule is derived from the RQDL description P_{CQ} by using rules (i), (iv), (ii) and (v) in that order. The conjunctive query

$$R_i : ans(a_1 : X, a_2 : Y) \leftarrow p(a_1 : X, b : Z), q(a_2 : Y, c : Z'), equal(Z, Z')$$

is an instantiated terminal expansion of the RQDL description, since it is an instantiation of rule R. In particular,

- $\$p_1, \$p_2$ map to predicate names p, q respectively.
- $\$a_1, \$a_2$ map to attribute names b, c respectively.
- $\vec{V}_1$ maps to $(a_1 : X, b : Z)$, $\vec{V}_2$ maps to $(a_2 : Y, c : Z')$ and $\vec{V}$ maps necessarily to their union, namely to $(a_1 : X, a_2 : Y, b : Z, c : Z')$.
- X_1, X_2 map to Z, Z' respectively.
- $\vec{V'}$ maps to $(a_1 : X, a_2 : Y)$.

All metapredicate subgoals are dropped. □

If Q is a conjunctive query with head predicate ans and P is an RQDL description, we say that Q *is expressible by* P, if there exists Q' described by P, such that $Q \equiv Q'$.

Referring to Example 7.3, query

$$Q : ans(a_1 : A, a_2 : B) \leftarrow p(a_1 : A, b : Z), q(a_2 : B, c : Z'), q(a_2 : W, c : U), equal(Z, Z')$$

is expressible by the description P_{CQ}, since it is equivalent to R_i.

Note here that RQDL can be easily extended (e.g., allowing not only tokens but also variables in place of predicate names) to describe the capabilities of information sources that understand and can process higher order logics, for example sources that understand HiLog [CKW93] or F-Logic [KL89]. We do not pursue this issue further in this paper.

The next section explains how to use RQDL to describe the capabilities of networks of mediators.

8 RQDL and mediator capabilities

Let us revisit the mediation architecture of Figure 1. In a dynamic environment such as the Internet, or the intranet of a big organisation, when integrating information we would like to be able to leverage existing integration "machinery" [Wie92]. Specifically, if a mediator exists that offers an integrated view of some information we want to access, we would like to be able to use it, instead of accessing each one of the sources it integrates. This is why networks of mediators, as in Figure 1, are possible and necessary. Using a mediator as a "source" to another mediator means of course that we must be able to describe mediator capabilities. As explained in the introduction, we assume that mediators have query processing capabilities that allow them to "handle" every conjunctive query over the data that they integrate. Given the expressiveness results of Section 6, p-Datalog cannot describe the capabilities of such a mediator. RQDL is powerful enough for that task. Let us consider a mediator M that integrates sources $S_1, \ldots, S_n$ and let the descriptions of these sources be $D_1, \ldots, D_n$. Also, assume that each wrapper understands one *answer* predicate, and let these be $ans_1, \ldots, ans_n$. Then, the RQDL program D_M that describes the capabilities of the mediator is the following:

$$
\begin{array}{ll}
ans(\vec{V}_1) & \leftarrow cond(\vec{V}), \vec{V}_1 \subseteq \vec{V} \\
cond(\vec{V}) & \leftarrow choose(\vec{V_1}), cond(\vec{V_2}), \vec{V} = \vec{V_1} \cup \vec{V_2} \\
cond(\vec{V}) & \leftarrow item(\vec{V}, \$a, X), equal(X, \$c), cond(\vec{V}) \\
cond(\vec{V}) & \leftarrow item(\vec{V}, \$a_1, X_1), item(\vec{V}, \$a_2, X_2), equal(X_1, X_2), cond(\vec{V}) \\
cond(\vec{V}) & \leftarrow choose(\vec{V}) \\
choose(\vec{V}) & \leftarrow ans_1(\vec{V}) \\
\vdots & \\
choose(\vec{V}) & \leftarrow ans_n(\vec{V}) \\
D_1 & \\
\vdots & \\
D_n &
\end{array}
$$

The similarity of this description to P_{CQ} of Example 7.2 is evident. D_M describes all conjunctive queries that the mediator can answer: that is any conjunctive query that combines results from queries that are accepted by the sources the mediator integrates; thus the concatenation of $D_1, \ldots, D_n$ in D_M. Given $D_1, \ldots, D_n$, the description D_M can obviously be automatically generated.

Next we will discuss an efficient algorithm for deciding whether a query is expressible by an RQDL description. The algorithm is based on a reduction of both the query and the description into a simple standard schema which facilitates reasoning about relations and attribute names.

9 Reducing RQDL to p-Datalog with function symbols

Deciding whether a query is expressible by an RQDL description requires "matching" the RQDL description with the query. This is a challenging problem because vectors have to match with non-atomic entities, *i.e.*, sets of variables, hence making matching much harder.

In [PGH96], where that problem is also identified, a brute force approach is used, that in effect tries to generate instantiated terminal expansions *bottom up*, so vectors actually match with sets during the derivation. Unfortunately, this approach soon leads to complicated problems which forced [PGH96] to restrict the applicability of matching algorithms to a subset of RQDL descriptions. A particularly tough problem is the existence of unsafe rules that have vectors in the head. A brute force approach may then derive extended facts where a vector is "half-specified", *i.e.*, we know some of the attr-variable pairs that it should contain but not all of them. Note that [PGH96] is not applicable to RQDL descriptions that may exhibit this behavior.

In this section we present an algorithm that avoids these problems by reducing the problem of query expressibility by RQDL descriptions to the problem of query expressibility by p-Datalog *with function symbols*, *i.e.*, we reduce the RQDL description into a corresponding description in p-Datalog with function symbols. The reduction is based on the idea that every database DB can be reduced into an equivalent database DB' such that the attribute names and relation names of DB appear in the data (and not the schema) of DB'. We call DB' a *standard schema database*. We then rewrite the query so that it refers to the schema of DB' (*i.e.*, the standard schema) and we also rewrite the description into a p-Datalog description *with function symbols* which refers to the standard schema as well.

Subsection 9.1 presents the conceptual reduction of a database into a standard schema database. Subection 9.2 presents the rewriting of queries and subsection 9.3 presents the rewriting of RQDL descriptions. Each of the subsections starts with one or two examples and continues with a formal definition of the reduction which can be skipped at the first reading.

9.1 Reduction of any database to standard schema database

In order to reason with the relation names and attribute names of the queries, we conceptually reduce the original database into a standard schema database where the relation names and the attribute names appear as data and hence can be manipulated without the need of higher order syntax. First we present a reduction example and then we formally define the reduction of a database into its standard schema counterpart.

Example 9.1 Consider the following database DB with schema $b(au, isbn)$ and $f(subj, isbn)$.

b	
au	*isbn*
Smith	123
Jones	345

f	
subj	*isbn*
Logic	123
Theology	345

The corresponding standard schema database DB' consists of two relations $tuple(table\ name, tuple\ id)$ and $attr(tuple\ id, attr\ name, value)$ which are common to all standard schema databases. In the running example DB' is

tuple	
table name	*tuple id*
b	b(au,Smith,isbn,123)
b	b(au,Jones,isbn,345)
f	f(subj,Logic,isbn,123)
f	f(subj,Theology,isbn,345)

attr		
tuple id	*attr name*	*value*
b(au,Smith,isbn,123)	au	Smith
b(au,Smith,isbn,123)	isbn	123
b(au,Jones,isbn,345)	au	Jones
b(au,Jones,isbn,345)	isbn	345
f(subj,Logic,isbn,123)	subj	Logic
f(subj,Logic,isbn,123)	isbn	123
f(subj,Theology,isbn,345)	subj	Theology
f(subj,Theology,isbn,345)	isbn	345

Notice above how we invented one tuple id for each tuple of the original database. □

Definition: Given a database DB, we say that the standard schema database corresponding to DB is the smallest database DB' such that

1. its schema is $tuple(table\ name, tuple\ id)$ and $attr(tuple\ id, attr\ name, value)$, and

2. for every tuple $t(a_1 : v_1, \ldots, a_n : v_n)$ in DB, there is a tuple $tuple(t, t(a_1, v_1, \ldots, a_n, v_n))$ in DB' and for every attribute $a_i, i = 1, \ldots, n$ there is a tuple $attr(t(a_1, v_1, \ldots, a_n, v_n), a_i, v_i)$ in DB'.

□

9.2 Reduction of queries to standard schema queries

The RQDL expressibility algorithm first reduces a given conjunctive query Q over some database DB into a corresponding query Q' over the standard schema database DB'. The reduction is correct in the following sense: the result of asking query Q' on DB' is equivalent, modulo tuple-id naming, to the reduction into standard schema of the result of Q on DB.

To illustrate the query reduction, let us consider a couple of examples. We first consider a boolean query Q over the schema of Example 9.1.

$$ans() \leftarrow b(au : X, isbn : S_1), f(subj : A, isbn : S_2), equal(S_1, S_2), equal(A, \texttt{Theology})$$

Query Q is reduced into the following query Q':

$$\begin{aligned} tuple(ans, ans()) \quad \leftarrow \quad & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), equal(S_1, S_2), \\ & attr(F, subj, A), equal(A, \texttt{Theology}) \end{aligned}$$

Notice that for every ordinary subgoal we introduce a *tuple* subgoal and invent a tuple id. For every attribute we introduce an *attr* subgoal. The tuple id for the result relation *ans* is simply *ans()* because the result relation has no attributes. When the query head has attributes, a single conjunctive query is reduced to a non-recursive Datalog program. For example, consider the following query that returns the authors and ISBNs of books if their subject is `Theology`.

$$\begin{aligned} ans(au : X, isbn : S_1) \quad \leftarrow \quad & b(au : X, isbn : S_1), f(subj : A, isbn : S_2), equal(S_1, S_2), \\ & equal(A, \texttt{Theology}) \end{aligned}$$

This query is reduced to the following program Q' where the first rule defines the *tuple* part of the standard schema answer and the last two rules describe the *attr* part.

$$
\begin{array}{lcl}
tuple(ans, ans(au, X, isbn, S_1)) & \leftarrow & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), \\
 & & equal(S_1, S_2), attr(B, au, X), attr(F, subj, A), \\
 & & equal(A, \texttt{Theology}) \\
attr(ans(au, X, isbn, S_1), au, X) & \leftarrow & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), \\
 & & equal(S_1, S_2), attr(B, au, X), attr(F, subj, A), \\
 & & equal(A, \texttt{Theology}) \\
attr(ans(au, X, isbn, S_1), isbn, S_1) & \leftarrow & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), \\
 & & equal(S_1, S_2), attr(B, au, X), attr(F, subj, A), \\
 & & equal(A, \texttt{Theology})
\end{array}
$$

In general, the reduction is accomplished by the following procedure:

Procedure 9.2 (Reduction) If Q's head is $ans(a_1 : V_1, \ldots, a_n : V_n)$, generate a program with $n+1$ rules such that

1. one rule has head $tuple(ans, ans(a_1, V_1, \ldots, a_n, V_n))$,
2. for every attribute $a_i, i = 1, \ldots, n$ there is a rule with head $attr(ans(a_1, V_1, \ldots, a_n, V_n), a_i, V_i)$, and
3. all rules have the same body which is constructed by the following steps:
   (a) For every subgoal of Q of the form $r(a_1 : X_1, \ldots, a_m : X_m)$, invent and associate to it a unique variable T. The variables such as T bind to tuple id's of the standard schema database and hence we call them *tuple id variables.*,
   (b) include in the standard schema query body the subgoal $tuple(r, T)$,
   (c) and for every attribute $a_i, i = 1, \ldots, m$ include in the standard schema query the subgoal $attr(T, a_i, X_i)$.
   (d) Add to that body all equality subgoals of the original query.

□

X_i can be a variable, a token or a constant. It is easy to see that under a few obvious constraints there exists the inverse reduction.

Next we show how we reduce RQDL descriptions into p-Datalog descriptions over standard schema databases.

9.3 Reduction of RQDL programs to Datalog programs operating on standard schema

In the previous sections we showed how schema information, *i.e.*, relation and attribute names, becomes data in standard schema databases. Based on this idea, we will reduce RQDL descriptions into p-Datalog descriptions that do not use higher order features such as metapredicates and vectors. In particular, we "reduce" vectors to tuple identifiers. Intuitively, if a vector matches with the arguments of a subgoal, then the tuple identifier associated with this subgoal is enough for finding all the attr-variable pairs that the vector will match to. Otherwise, if a vector $\vec{V}$ is the result of a union of two other vectors $\vec{V_1}$ and $\vec{V_2}$, then we associate with it a new *constructed* tuple id, the function $u(T_1, T_2)$ where T_1 and T_2 are the tuple id's that correspond to $\vec{V_1}$ and $\vec{V_2}$. As we will see later, the reduction carefully produces a program which terminates despite the use of the u function.

Example 9.3 Let us first consider a simple but interesting one-rule description:

$$ans(\vec{V}) \leftarrow \$p(\vec{V}), item(\vec{V}, name, X)$$

This RQDL rule describes all conjunctive queries that refer to any schema over one relation, with the constraint that the schema of the relation contains an attribute "name". This description reduces to the following p-Datalog description:

$$\begin{array}{ll} tuple(ans, ans(T)) & \leftarrow tuple(\$p, T), attr(T_1, name, X), equal(T, T_1) \\ attr(ans(T), \$a, X) & \leftarrow tuple(ans, ans(T)), attr(T_1, \$a, X), equal(T, T_1) \end{array}$$

The vector variable $\vec{V}$ is reduced to the variable T which matches with a tuple id. The metapredicate $item(\vec{V}, name, X)$, is reduced to the predicate $attr(T, name, X)$.
□

Example 9.4 The description of Example 7.2 describes all boolean conjunctive queries. It reduces into the following p-Datalog description (with function symbols):

$$\begin{array}{ll} tuple(ans, ans(T)) \leftarrow cond(T) & (i) \\ cond(T) \leftarrow tuple(\$p, T_1), cond(T_2), valid(T, T_1, T_2) & \\ cond(T) \leftarrow attr(T', \$a, X), equal(X, \$c), cond(T), equal(T', T) & \\ cond(T) \leftarrow attr(T_1, \$a_1, X_1), attr(T_2, \$a_2, X_2), equal(X_1, X_2), cond(T), equal(T, T_1), equal(T, T_2) & \\ cond(T) \leftarrow \epsilon & \end{array}$$

and $subset\_flag_{(i,T)} = 1$.
The reduction of each rule is independent from the reduction of other rules. In the second rule, notice that we reduced $\vec{V}$ to T, which is "produced" by the predicate *valid*, given T_1 and T_2. *valid* is a predicate defined by the rules of Fig. 9 (see Appendix B), which have to be included in all reduced p-Datalog descriptions. *valid* constructs a new tuple id of a restricted form, that has "associated" with it all the attributes associated with T_1 or T_2. The role of *valid* is to "simulate" the union that it replaces, by not allowing generation of arbitrary u terms[14] but only those that follow the order mentioned below.

The intuition behind *valid* terms is the following: tuple id variables bind either to tuple ids or to constructed tuple ids, *i.e.*, u terms "built" from tuple ids. Assuming that there is a total order for the tuple ids of the standard schema database, $valid(T, T_1, T_2)$ creates a u term in which all tuple ids appear in sorted order, and none are repeated. For example, $valid(T, u(t_2, u(t_3, t_4)), u(t_3, t_5))$ will bind T to $u(t_2, u(t_3, u(t_4, t_5)))$.

Finally, the description has to include the "default" rules[15] of Fig. 7, that make sure that all attributes of tuple with ids T_1 and T_2 are also attributes of tuples with id T, constructed from T_1, T_2. □

Formally, an RQDL description P is reduced to a p-Datalog description P' by the following steps:

1. Include in P' the rules of Figures 7 and 9.
2. Reduce each rule r of the description to p-Datalog with functions as follows:
   (a) Reduce predicates that do not involve vectors as described in subsection 9.2.

[14]The analogy is that union includes attr-var pairs only once.
[15]Notice that, because of its simplicity, we did not need to include these rules in Example 9.3.

(b) For each subgoal of the form $r(\vec{V})$ include in the reduced rule a subgoal $tuple(r, T)$. T is the *reduction* of $\vec{V}$.

(c) For each subgoal of the form $item(\vec{V}, a, X)$, where a is a token or a constant, include in the reduced rule the subgoal $attr(T, a, X)$, where T is the reduction of $\vec{V}$.

(d) For each subgoal of the form $union(\vec{V}, \vec{V_1}, \vec{V_2})$, replace in the reduced rule all instances of $\vec{V}$ with T and include the subgoal $valid(T, T_1, T_2)$, where T_1 and T_2 are the reductions of $\vec{V_1}$ and $\vec{V_2}$.

(e) For each subgoal of the form $subset(\vec{V}_1, \vec{V})$, let T_1 be the reduction of $\vec{V}_1$ and T be the reduction of $\vec{V}$. Replace T_1 by T in the rule where *subset* appears, set the *subset flag* for the variable T and the rule to 1 (see below) and drop the *subset* subgoal.

(f) If the head is of the form $ans(\vec{V})$ then reduce it to $tuple(ans, T)$.

(g) If the head is of the form *ans*(attr-var set) then follow Procedure 9.2 to generate all the p-Datalog rules that r reduces to.

The intuition behind the *subset_flag* of a rule and a variable is as follows: Assume the existence of a subgoal $subset(\vec{V}_1, \vec{V})$ in rule r. As we have said earlier, $\vec{V}_1$ must appear in the rule head, so let the head of r be $p(\vec{V}_1)$, and $\vec{V}$ must appear in an ordinary subgoal, say $q(\vec{V})$. The subset subgoal means that the RQDL rule r describes all conjunctive queries whose head attribute set is any projection of the attribute set of relation q. In the reduction, we replace T_1 (the reduction of $\vec{V}_1$) by T (the reduction of $\vec{V}$), saying effectively that the attribute set of p must be the same as the attribute set of q. That's why we set a flag on the rule for that variable, the *subset flag*, to make sure we also consider described those conjunctive queries that include projections on q.[16]

Theorem 9.5 Let P be an RQDL description and P' its reduction in p-Datalog with functions. Let also DB be a canonical standard schema database of a query Q. Then P' applied on DB terminates.

Crux: It suffices to see that the generation of u terms cannot fall into an infinite loop, since no tuple id present in the database can appear twice in any constructed tuple id. □

In the remaining sections, we will denote p-Datalog with functions with *p-Datalog*^{f}. The next section explains the semantics of p-Datalog with functions, and shows how to solve the CBR problem for RQDL using the algorithms developed for p-Datalog in Sections 3 and 4.

[16] Another way to handle the subset metapredicate is by defining an ordering among constructed tuple ids, *i.e.*by defining what $T_i < T_j$ means if T_i, T_j are not atomic values. Then $subset(\vec{V}_1, \vec{V})$ would just be reduced to $T_1 < T$, where T_1, T are correspondingly the reductions of $\vec{V}_1$ and $\vec{V}$.

$$attr(T, \$a, X) \leftarrow attr(T_1, \$a, X), valid(T, T_1, T_2)$$
$$attr(T, \$a, X) \leftarrow attr(T_2, \$a, X), valid(T, T_1, T_2)$$

Figure 7: Default rules for generation of *attr* tuples

10 QED and CBR for RQDL descriptions

The reduction presented in the previous section allows us to formulate a solution to the expressibility problem for RQDL descriptions. In particular, we show that we can use QED with small changes for p-Datalog^f; we prove that the modified QED is sound and complete over the fragment of p-Datalog^f that is generated by the RQDL reduction. We then proceed to discuss the CBR scheme for RQDL; that also uses the RQDL reduction to reduce the CBR problem for RQDL to the CBR problem for p-Datalog^f.

We first illustrate QED for RQDL with an example. Notice that there are now two "designated" predicates, the predicates $tuple$ and $attr$.

Example 10.1 Consider the query Q: $ans(a:X) \leftarrow books(au:X, titl:Y)$ and the description

$$\begin{array}{lcl} ans(a:X) & \leftarrow & \$r(au:X, titl:Y) \\ ans(b:Y) & \leftarrow & \$r(au:X, titl:Y) \end{array}$$

The reduction of the query is

$$\begin{array}{lcl} tuple(ans, ans(a,X)) & \leftarrow & tuple(p,T_0), attr(T_1, au, X), attr(T_2, titl, Y), equal(T_0,T_1), equal(T_0,T_2) \\ attr(ans(a,X), a, X) & \leftarrow & tuple(p,T), attr(T, au, X), attr(T, titl, Y), equal(T_0,T_1), equal(T_0,T_2) \end{array}$$

The canonical DB is

$$tuple(books, t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0, t_1, t_2)$$

The reduction of the description (after rectification) is

$$\begin{array}{lcl} tuple(ans, ans(X)) & \leftarrow & tuple(\$r,T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T,T_1), equal(T,T_2) \\ attr(ans(X), a, X) & \leftarrow & tuple(\$r,T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T,T_1), equal(T,T_2) \\ tuple(ans, ans(Y)) & \leftarrow & tuple(\$r,T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T,T_1), equal(T,T_2) \\ attr(ans(Y), b, Y) & \leftarrow & tuple(\$r,T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T,T_1), equal(T,T_2) \end{array}$$

Notice that we didn't include the rules of Figures 7 or 9 ($valid$ rules) in the reduced description, since the original description didn't contain any metapredicates.

If we run the Algorithm 3.4 on the canonical DB, the following extended facts are produced:

$$\begin{array}{ll} (1) < tuple(ans, ans(x)), & \{tuple(books,t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0,t_1,t_2)\} > \\ (2) < attr(ans(x), a, x), & \{tuple(books,t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0,t_1,t_2)\} > \\ < tuple(ans, ans(y)), & \{tuple(books,t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0,t_1,t_2)\} > \\ < attr(ans(y), b, y), & \{tuple(books,t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0,t_1,t_2)\} > \end{array}$$

The output of the algorithm includes extended facts with the same tuple id. We "group" together the extended facts with the same tuple id. We notice that group consisting of the extended facts (1) and (2) corresponds to the exact two conjunctive queries that are the reduction of Q. We therefore say that Q is expressible by our description. $\Box$

Before presenting the theorem that states the condition for RQDL expressibility, let us make the following important observations:

- Lemmata 3.6 and 3.8 and Theorem 3.7 still hold for p-Datalog^f.

- Let Q be a conjunctive query and let $\{Q_i, i \leq n\}$ be the set of standard schema queries it reduces to. Let H_i be the heads of those queries. As we pointed out in Section 9.2, all Q_i have the same body. Moreover, for Q_1, H_1 is of the form $tuple(ans, T)$, where T is a term that denotes a tuple id, and for $Q_i, i \neq 1$, H_i are of the form $attr(T, c_i, X_i)$ *for the same* T. We call T the *query id*. In reference to the previous example, the query id is $ans(a, X)$.

Theorem 10.2 A query Q is expressible by an RQDL description P without the *subset* metapredicate *if and only if* there exists a maximal set $\{Q_i', i \leq n\}$ [17] of queries described by the reduced description P', where all Q_i' have the same id, such that $Q_i' \equiv Q_i$, $\forall i \leq n$. Maximal means that $\{Q_i'\}$ includes *all* described queries with that same query id.

Referring again to Example 10.1, the maximal set $\{Q_i'\}$ is the set of the corresponding queries to extended facts (1) and (2).

Let us observe that the exact "value" of tuple ids is not important: their use is to identify components (*i.e.*, attributes) of the same relation. Therefore, we say that a reduced query Q in p-Datalog^f is expressible by a reduced p-Datalog^f description P if and only if there exists Q' equivalent to Q *up to tuple-id naming* that is described by P.

Proof: (Sketch) The above theorem is easy to see in the case where the RQDL description contains no vectors. When the RQDL description contains vectors, the intuition is as follows: Let Q be a conjunctive query without projection[18], and let $\{Q_i, i \leq n\}$ be the set of standard schema queries it reduces to. Also let P be the RQDL description and P_{red} be the reduced p-Datalog^f description.

For the *IF* direction: The reduction directly maps the RQDL rules to rules "producing" *tuple* subgoals, so it ensures that if Q is expressible by P, then Q_1 is expressible by P_{red}. Because of this and by use of the "default" rules of Figure 7, all $\{Q_i\}$ are also expressible.

The *ONLY IF* direction is straightforward in the absence of a *subset* subgoal. In the presence of *subset*, the crux is that $\{Q_i\}$ is the *maximal* set of described queries with the same query id. The result follows from this together with Theorem 3.7. □

Because of Theorems 9.5 and 10.2, we can use Algorithm QED (see Section 3 to answer the expressibility question in RQDL. QED generates all possible extended facts for *tuple* and *attr*. We then check whether (i) all and only the necessary "frozen" *tuple* and *attr* facts are produced and have the same id, and (ii) their corresponding queries are equivalent to the Q_i's. For the algorithm to work properly, a change needs to be made to the definition[19] of the supporting set of a fact: due to the reduction introduced in Sections 9.2 and 9.3, there is an implicit "connection" between a fact $tuple(const_1, T)$ and facts $attr(T, const_2, X)$, *i.e.*, between the *tuple* fact and the *attribute* facts that are created by the reduction. We make that connection explicit by modifying the definition of supporting set as follows:

Definition: Supporting Set - Modified Let h be an ordinary fact produced by an application of the p-Datalog^f rule

$$r : H \leftarrow G_1, \ldots, G_k, E_1, \ldots, E_m$$

of a (reduced) p-Datalog^f description P on a database DB that consists of a canonical database CDB and other facts, and let μ be a mapping from the rule into DB such that $\mu(G_i), \mu(E_j) \in DB$ and $h = \mu(H)$. The set $\mathcal{S}_h$ of supporting facts of h, or *supporting set* of h, with respect to P, is the smallest set such that

[17] If *subset* exists, then it could be $\{Q_i', i \leq m\}$ with $m \geq n$.

[18] Projection is taken care of with the *subset* metapredicate, that directly maps to the *subset_flag*.

[19] We could have the same effect by correspondingly changing the RQDL to p-Datalog^f reduction procedure.

- if $\mu(G_i) \in CDB$, then $\mu(G_i) \in \mathcal{S}_h$,
- if $\mu(G_i) \notin CDB$ and $\mathcal{S}'$ is the set of supporting facts of $\mu(G_i)$, then $\mathcal{S}' \subseteq \mathcal{S}_h$,
- if $tuple(c,t) \in \mathcal{S}_h$ for some[20] c and t, then for all c', x, if $attr(t, c', x)$ is in the canonical DB, then $attr(t, c', x) \in \mathcal{S}_h$,
- if E is the set of all $\mu(E_i) \in \mathcal{S}_h$, then the smallest set of equality facts that includes E and is an equivalence relation is included in $\mathcal{S}_h$.

□

Modifications in the presence of *subset* subgoals: We have already explained that a $subset(\vec{V}, \vec{V}')$ subgoal is reduced into a statement setting a *subset_flag* "attached" to the rule for the variable T that $\vec{V}$ reduces to. During execution of the QED algorithm, whenever a *tuple* fact is generated from this rule, we set a *subset_flag* annotation on its tuple id. That annotation is used after the execution is complete together with Theorem 10.2 to determine expressibility.

Let us now consider the following example.

Example 10.3 If our RQDL description is

$$ans(\vec{V}) \leftarrow p(\vec{V}), item(\vec{V}, au, X)$$

as in Example 9.3 then the query $Q : ans(au : X) \leftarrow p(au : X, subj : Y)$ is *not* expressible by our description. The reduction of the description is

$$\begin{array}{lcl} tuple(ans, T) & \leftarrow & tuple(p, T), attr(T_1, au, X), equal(T, T_1) \\ attr(T, \$a, X) & \leftarrow & attr(T_1, \$a, X), valid(T, T_2, T_3), equal(T_1, T_2) \\ attr(T, \$a, X) & \leftarrow & attr(T_1, \$a, X), valid(T, T_2, T_3), equal(T_1, T_3) \end{array}$$

plus the rules defining the *valid* predicate (see Appendix B)[21]. and the reduction of the query (*i.e.*, the set $\{Q_i\}$) is

$$\begin{array}{lcl} tuple(ans, ans(au, X)) & \leftarrow & tuple(p, T), attr(T, au, X), attr(T, subj, Y) \\ attr(ans(au, X), au, X) & \leftarrow & tuple(p, T), attr(T, au, X), attr(T, subj, Y) \end{array}$$

The canonical DB is then

$$tuple(p, t_0), attr(t_1, au, \bar{x}), attr(t_2, subj, \bar{y}), equal(t_0, t_1, t_2)$$

The extended facts produced by Algorithm 3.4, taking into account the modification of the definition of supporting sets introduced above, are

$$\begin{array}{lll} (1) & < tuple(ans, t_0), & \{tuple(p, t_0), attr(t_1, au, \bar{x}), attr(t_2, subj, \bar{y}), equal(t_0, t_1, t_2)\} > \\ (2) & < valid(t_0, t_0, t_0) & \{tuple(p, t_0), attr(t_1, au, \bar{x}), attr(t_2, subj, \bar{y}), equal(t_0, t_1, t_2)\} > \\ (3) & < attr(t_0, au, \bar{x}) & \{tuple(p, t_0), attr(t_1, au, \bar{x}), attr(t_2, subj, \bar{y}), equal(t_0, t_1, t_2)\} > \\ (4) & < attr(t_0, subj, \bar{y}) & \{tuple(p, t_0), attr(t_1, au, \bar{x}), attr(t_2, subj, \bar{y}), equal(t_0, t_1, t_2)\} > \end{array}$$

Let us look in more detail into how extended fact (1) was produced. Application of the first rule of the p-Datalog^f program generates $< tuple(ans, t_0), \{tuple(p, t_0), attr(t_1, au, \bar{x}), equal(t_0, t_1)\} >$.

[20] c and t can be frozen or regular constants.

[21] The reduction presented in Example 9.3 is simplified

Then, a valid rule fires and generates $< valid(t_0, t_0, t_0)\{tuple(p, t_0), attr(t_1, au, \bar{x}), equal(t_0, t_1)\} >$. The second rule of the program consequently fires and gives

$$< attr(t_0, au, \bar{x})\{tuple(p, t_0), attr(t_1, au, \bar{x}), equal(t_0, t_1)\} >$$

and

$$< attr(t_0, subj, \bar{y})\{tuple(p, t_0), attr(t_2, subj, \bar{y}), equal(t_0, t_2)\} >$$

Then, according to the modified definition of supporting set, we need to augment the supporting set of $tuple(ans, t_0)$, to include $attr(t_2, subj, \bar{y})$, thus getting extended fact (1). Performing the augmentation step cannot take more than exponential amount of time. Finally, a valid rule fires again to generate (2), and then the second rule of the program fires, to generate (3) and (4).

Even though both standard schema queries of the reduction are expressible by our reduced description, the original query as pointed out is not expressible by the RQDL description. That is because the only maximal set of described queries produced (consisting of the corresponding queries for (1),(3) and (4)) is larger than the set of reduced queries.

On the other hand, if the description were

$$ans(\vec{V}) \leftarrow p(\vec{V}_1), item(\vec{V}, au, X), subset(\vec{V}, \vec{V}_1)$$

then Q is described by the modified description. The reduction of the description would be exactly the same, but we would set the $subset\_flag$ for $\vec{V}$ on the rule. Then, using the modification described previously and following Theorem 10.2, the algorithm would decide correctly that Q is described by the modified description. $\Box$

Let us consider a more complicated example.

Example 10.4 The following source can accept queries that perform a join between relation q with any other relation over any set of attributes. The description of this source is a simplification of description P_{CQ}, of Example 7.2.

$$\begin{array}{lcl}
ans(\vec{V}) & \leftarrow & cond(\vec{V}) \\
cond(\vec{V}) & \leftarrow & q(\vec{V}_1), union(\vec{V}, \vec{V}_1, \vec{V}_2), cond(\vec{V}_2) \\
cond(\vec{V}) & \leftarrow & item(\vec{V}, \$a_1, X_1), item(\vec{V}, \$a_2, X_2), equal(X_1, X_2), cond(\vec{V}) \\
cond(\vec{V}) & \leftarrow & \$r(\vec{V})
\end{array}$$

The reduction of the description, after rectification, is

$$\begin{array}{lcl}
tuple(ans, T) & \leftarrow & cond(T) \\
cond(T) & \leftarrow & tuple(q, T_1), cond(T_2), valid(T, T_3, T_4), equal(T_1, T_3), equal(T_2, T_4) \\
cond(T) & \leftarrow & attr(T, \$a_1, X_1), attr(T_1, \$a_2, X_2), equal(X_1, X_2), cond(T_2), \\
 & & equal(T, T_1), equal(T, T_2) \\
cond(T) & \leftarrow & tuple(\$r, T) \\
attr(T, \$a, X) & \leftarrow & attr(T_1, \$a, X), valid(T, T_2, T_3), equal(T_1, T_2) \\
attr(T, \$a, X) & \leftarrow & attr(T_1, \$a, X), valid(T, T_2, T_3), equal(T_1, T_3)
\end{array}$$

plus the rules in Fig. 9 (see Appendix B).
The user query submitted to the source is the following:

$$ans(au : X, ln : X, subj : Z) \leftarrow q(au : X, subj : Z), s(ln : X)$$

(where ln stands for last name) which produces the extended canonical DB

$$tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)$$

The standard schema reduction of the user query is

$$\begin{array}{lll}
tuple(ans, ans(au,X,ln,X,subj,Z)) & \leftarrow & tuple(q,Q), tuple(s,S), attr(Q_1,au,X), \\
& & attr(Q_2,subj,Z), attr(S_1,ln,X_1), equal(S,S_1), \\
& & equal(X,X_1), equal(Q,Q_1,Q_2) \\
attr(ans(au,X,ln,X,subj,Z),au,X) & \leftarrow & tuple(q,Q), tuple(s,S), attr(Q_1,au,X), \\
& & attr(Q_2,subj,Z), attr(S_1,ln,X_1), equal(S,S_1), \\
& & equal(X,X_1), equal(Q,Q_1,Q_2) \\
attr(ans(au,X,ln,X,subj,Z),ln,X) & \leftarrow & tuple(q,Q), tuple(s,S), attr(Q_1,au,X), \\
& & attr(Q_2,subj,Z), attr(S_1,ln,X_1), equal(S,S_1), \\
& & equal(X,X_1), equal(Q,Q_1,Q_2) \\
attr(ans(au,X,ln,X,subj,Z),subj,Z) & \leftarrow & tuple(q,Q), tuple(s,S), attr(Q_1,au,X), \\
& & attr(Q_2,subj,Z), attr(S_1,ln,X_1), equal(S,S_1), \\
& & equal(X,X_1), equal(Q,Q_1,Q_2)
\end{array}$$

Running Algorithm 3.4 on the canonical DB produces the following extended facts[22]:

$$\begin{array}{ll}
< valid(u(t_0,t_4),t_0,t_4), & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4)\} > \\
< cond(t_4), & \{tuple(s,t_3), attr(t_4,ln,\bar{x}_1), equal(t_3,t_4)\} > \\
< cond(u(t_0,t_4)), & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4)\} > \\
< cond(u(t_0,t_4)), & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} > \\
(1) < tuple(ans,u(t_0,t_4)), & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} > \\
< valid(u(t_0,t_4),u(t_0,t_4),u(t_0,t_4)), & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} > \\
(2) < attr(u(t_0,t_4),au,\bar{x}) & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} > \\
(3) < attr(u(t_0,t_4),ln,\bar{x}_1) & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} > \\
(4) < attr(u(t_0,t_4),subj,\bar{z}) & \{tuple(q,t_0), attr(t_1,au,\bar{x}), attr(t_2,subj,\bar{z}), tuple(s,t_3), \\
& attr(t_4,ln,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >
\end{array}$$

The maximal set of described queries with query id $u(t_0,t_4)$ (corresponding to (1),(2),(3) and(4)) is equal to the set of the standard schema queries that are the reduction of the user query. Therefore, the user query is expressible by our RQDL description, by Theorem 10.2. □

10.1 The CBR problem for RQDL

We solve the CBR problem for a given query and a given (reduced) RQDL description in two steps:

[22]We are only showing some of the extended facts that could be produced, for the sake of brevity.

- We generate the set of relevant described queries from the output of the Algorithm 3.4, by "glueing" together the *tuple* and *attr* subgoals that have *the same supporting set*. In other words, we create the corresponding standard schema queries for the extended facts and then do the inverse reduction on the sets of those that have the same id and body (thus ending up with some queries on the original schema). These are the relevant queries of the description with respect to the given query.
- When we have the given query (over some schema) and a number of relevant queries (or views) over the same schema, we can apply an answering queries using views algorithm [Qia96, LMSS95] or [RSU95] on that problem.

Let us notice that, in the presence of *subset* subgoals in the RQDL description, the QED algorithm produces candidate queries that can have set the *subset_flag* annotation. In principle, these annotations can be ignored for the solution of the CBR problem, since we assume that the mediator has the capability to do projections locally (*i.e.*, projections can always be handled by the final rewriting at the mediator).

The complexity of this procedure is non-deterministic exponential in the input size.

It is obvious that the discussion of subsection 4.1 about binding requirements holds for RQDL as well.

Example 10.5 We consider a source that expects a selection condition on attribute au or on attribute $subj$, but not both. The RQDL description for this source is

$$\begin{aligned} ans(\vec{V}) &\leftarrow \$r(\vec{V}), item(\vec{V}, au, \$c) \\ ans(\vec{V}) &\leftarrow \$r(\vec{V}), item(\vec{V}, subj, \$c) \end{aligned}$$

The description reduces to

$$\begin{aligned} tuple(ans, T) &\leftarrow tuple(\$r, T), attr(T_1, au, X), equal(T, T_1) \\ tuple(ans, T) &\leftarrow tuple(\$r, T), attr(T_1, subj, X), equal(T, T_1) \\ attr(T, \$a, X) &\leftarrow attr(T_1, \$a, X), valid(T, T_2, T_3), equal(T_1, T_2) \\ attr(T, \$a, X) &\leftarrow attr(T_1, \$a, X), valid(T, T_2, T_3), equal(T_1, T_3) \end{aligned}$$

plus the rules in Fig. 9 (see Appendix B).

Let the user query be

$$Q : ans(subj : X, au : Y, isbn : Z) \leftarrow books(subj : X, au : Y, isbn : Z), equal(X, \texttt{Logic}), equal(Y, \texttt{Smith})$$

It is obvious that Q can be answered with a combination of queries expressible by the description: First send the selection condition on au, then on $subj$ and finally intersect the two results. Q reduces to

$$\begin{aligned} tuple(ans, ans(subj, X, au, Y, isbn, Z)) &\leftarrow tuple(books, T), attr(T, au, X), attr(T, subj, Y), \\ &\quad\ attr(T, isbn, Z), equal(X, \texttt{Logic}), equal(Y, \texttt{Smith}) \\ attr(ans(subj, X, au, Y, isbn, Z), subj, X) &\leftarrow tuple(books, T), attr(T, au, X), attr(T, subj, Y), \\ &\quad\ attr(T, isbn, Z), equal(X, \texttt{Logic}), equal(Y, \texttt{Smith}) \\ attr(ans(subj, X, au, Y, isbn, Z), au, Y) &\leftarrow tuple(books, T), attr(T, au, X), attr(T, subj, Y), \\ &\quad\ attr(T, isbn, Z), equal(X, \texttt{Logic}), equal(Y, \texttt{Smith}) \\ attr(ans(subj, X, au, Y, isbn, Z), isbn, Z) &\leftarrow tuple(books, T), attr(T, au, X), attr(T, subj, Y), \\ &\quad\ attr(T, isbn, Z), equal(X, \texttt{Logic}), equal(Y, \texttt{Smith}) \end{aligned}$$

The canonical DB is then[23]

$$tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}), equal(\bar{x}, \texttt{Logic}), equal(\bar{y}, \texttt{Smith})$$

The following extended facts are generated[24] by algorithm QED-T:

$$
\begin{array}{ll}
< tuple(ans, t), & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{x}, \texttt{Logic})\} > \\
< tuple(ans, t), & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(Y, \texttt{Smith})\} > \\
< attr(t, subj, \bar{x}) & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{x}, \texttt{Logic})\} > \\
< attr(t, au, \bar{y}) & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{x}, \texttt{Logic})\} > \\
< attr(t, isbn, \bar{z}) & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{x}, \texttt{Logic})\} > \\
< attr(t, subj, \bar{x}) & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{y}, \texttt{Smith})\} > \\
< attr(t, au, \bar{y}) & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{y}, \texttt{Smith})\} > \\
< attr(t, isbn, \bar{z}) & \{tuple(books, t), attr(t, subj, \bar{x}), attr(t, au, \bar{y}), attr(t, isbn, \bar{z}, equal(\bar{y}, \texttt{Smith})\} >
\end{array}
$$

The result (after the inverse reduction) is two candidate conjunctive queries, with binding information:

$$C_1 : ans^{bff}(subj : X, au : Y, isbn : Z) \leftarrow books(subj : X, au : Y, isbn : Z)$$

and

$$C_2 : ans^{fbf}(subj : X, au : Y, isbn : Z) \leftarrow books(subj : X, au : Y, isbn : Z)$$

Using Q and C_1, C_2 as input to algorithm $AnsBind$, we get the expected answer. $\Box$

11 Related Work

Many projects have dealt with data integration of structured sources (e.g., [LMR90, A^+91, HM93, K^+93, T^+90].) These projects ignored the problem of the different and limited query capabilities of information sources, which is important for integration systems that deal with heterogeneous sources. In what follows, we discuss the approaches taken by a newer generation of projects and we also discuss some theoretical work in this area.

HERMES [S^+] decribes the capabilities of sources by using some literals to explicitly specify the parameterized calls that are sent to the sources. Unfortunately, this reduces the interface between the integration system and the sources to a limited set of explicitly listed parameterized calls.

[PGGMU95] suggested a grammar-like approach for describing query capabilities and [LRU96] used a Datalog with tokens for the same purpose. These works are focused on showing how we can compute a query Q given a capabilities description P. The algorithm presented in [PGGMU95] only applies to specific classes of descriptions. We already mentioned that we improved upon the result of [LRU96] for the problem of answering a query using an infinite number of views. Moreover, our paper also studies RQDL, which is more powerful than p-Datalog, and also gives expressiveness results.

RQDL was proposed by [PGH96] to allow capabilities descriptions that are not schema specific. Furthermore, [PGH96, PGH] proposed the mediator architecture which includes a CBR. In this paper we show that RQDL is more expressive than p-Datalog. Furthermore, we present CBR algorithms which also include arbitrary use of the "union" and "subset" metapredicates and we provide proofs and complexity results.

[23] For brevity we are not doing full rectification.

[24] We are only showing the extended facts of interest.

The Information Manifold [LRO96] focuses on the capabilities description of sources found on the Web; hence it does not consider recursion. The expressive power of its capabilities-describing mechanism is strictly less than p-Datalog.

The DISCO system [TRV95] describes the capabilities of the sources using context-free grammars appropriately augmented with actions. DISCO enumerates plans initially ignoring limited wrapper capabilities. It then checks the queries that appear in the plans against the wrapper grammars and rejects the plans containing unsupported queries. DISCO's strategy can be much more expensive than doing capabilities-based rewriting, which ensures that the queries emitted to the wrappers are indeed answerable by the source.

The Garlic system [HKWY97] combines capabilities-based rewriting with cost-based optimization. The assumption is made that all the variables mentioned in a query are always available by the wrapper. This compromises the expressiveness of the description language but greatly simplifies the proposed algorithm. It is also interesting that capabilities descriptions are given in terms of plans supported by the wrappers. Additional assumptions are made at this point regarding the class of plans that can be described.

RQDL's handling of constructed tuple ids is based on a use of Skolem functions that is close to the ideas in [Mai86, KL89].

The following subsection discusses the use of tokens for the description of binding requirements and compares that approach to the use of binding patterns [RSU95, Ull89].

11.1 Describing binding requirements in p-Datalog

As we have already noticed, sources can often only answer queries that have specific *binding requirements*. As mentioned in section 2, we are using tokens to specify that some constant is expected in some fixed position in the query, *i.e.*, to implicitly define the *binding requirements* of described queries. In contrast, [RSU95] uses explicit enumeration of accepted binding patterns [Ull89] for each described query to achieve the same goal.

Example 11.1 Let us consider the following p-Datalog rule:

$$ans(X, Y) \leftarrow p(X, Z, \$c_1), q(Y, Z, \$c_2, W) \tag{17}$$

Rule 17 describes a join query that requires two bindings, one for the third argument of relation p and one for the third argument of relation q. Using the notation of [RSU95], also used in [Ull89], we could write rule 17 above as follows:

$$ans^{ffbb}(X, Y, A, B) \leftarrow p(X, Z, A), q(Y, Z, B, W) \tag{18}$$

This rule describes the same binding requirements as rule 17. □

Explicitly specifying accepted binding patterns as in rule 18 presents a number of problems. In particular, it obscures the distinction between variable and constant in the rule. This complicates answering the query expressibility question. Moreover, and more importantly, explicit specification of binding patterns does not generalize in the presence of recursion. When query capabilities are described with a p-Datalog program, it is not even possible to enumerate all posssible binding patterns: the description encodes a possibly infinite number of described queries that have different bound variables.

On the other hand, using tokens allowed us to naturally extend the description of binding requirements to the case of p-Datalog programs. The difference is made clearer by the following example.

Example 11.2 Let us revisit Example 2.3, that describes a particular bibliographic source. The p-Datalog description for that source is the following[25]

$$
\begin{array}{ll}
ans(I,A,T,P,Y,Pg) & \leftarrow books(I,A,T,P,Y,Pg), ind(I) \\
ind(I) & \leftarrow abstract\_index(\$c, I) \\
ind(I) & \leftarrow ind(I), abstract\_index(\$c, I)
\end{array}
$$

The source describes the following infinite family of conjunctive queries:

$$
\begin{array}{l}
ans(I,A,T,P,Y,Pg) \leftarrow books(I,A,T,P,Y,Pg) \\
ans(I,A,T,P,Y,Pg) \leftarrow books(I,A,T,P,Y,Pg), abstract\_index(c_1, I) \\
ans(I,A,T,P,Y,Pg) \leftarrow books(I,A,T,P,Y,Pg), abstract\_index(c_1, I), abstract\_index(c_2, I) \\
\text{etc.}
\end{array}
$$

The queries in this family have an increasing number of bound variables, so their binding patterns would look like this:

$ffffff$,
$ffffffb$,
$ffffffbb$,
etc.

The use of tokens allows us to describe the binding requirements succinctly. □

12 Conclusions and Future Work

We discussed the problems of (i) describing the query capabilities of sources and (ii) using the descriptions for source wrapping and mediation. We first considered a Datalog variant, called p-Datalog, for describing the set of queries accepted by a wrapper. We also provide algorithms for solving (i) the expressibility and (ii) the CBR problems. The first algorithm decides whether a given query is equivalent to one of the queries described by a p-Datalog program. This algorithm is used by the wrapper. The second algorithm is run by the mediators and it finds out if a given query can be computed using queries which are expressible by a p-Datalog program.

We then study the expressive power of p-Datalog. We show that it is more powerful than using binding patterns but we also reach the important negative result that p-Datalog can *not* describe the query capabilities of certain powerful sources. In particular, we show that there is no p-Datalog program that can describe all conjunctive queries over a given schema. Indeed, there is no program that describes all boolean conjunctive queries over the schema. A direct consequence of our result is that p-Datalog can not model a fully-fledged relational DBMS.

We subsequently describe and extend RQDL, which is a provably more expressive language than p-Datalog. The extra power is mainly a result of *vector variables* which can match to sets of attributes of arbitrary length. One consequence of the extra power is the ability to automatically derive a description of the capabilities of the mediator, given the descriptions of the wrapper capabilities. However, the existence of vector variables makes very hard a brute force implementation of mediator and wrapper algorithms using RQDL. We get around this problem by providing a reduction of RQDL descriptions into p-Datalog augmented with function symbols. Using this reduction we discuss complete algorithms for solving the expressibility and the CBR problem.

We have focused exclusively on conjunctive queries. We plan to extend our work to non-conjunctive queries, *i.e.*, queries involving aggregates and negation.

[25] Variable names are changed

Acknowledgements

The authors wish to thank Serge Abiteboul for his help in formalizing the presentation of p-Datalog and the proof of Theorem 6.4, and to Jeff Ullman for many fruitful discussions.

References

[A+91] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

[ACPS96] S. Adali, S. C. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. SIGMOD*, pages 137–48, 1996.

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

[ASU87] A. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1987.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. Hilog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, February 1993.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Proc. VLDB Vonference*, pages 277–91, Vancouver, Canada, August 1992.

[H+97] J. Hammer et al. Extracting semistructured information from the Web. In *Workshop on Management of Semistructured Data, ACM SIGMOD Conf.*, 1997.

[HKWY96] Laura Haas, Donald Kossman, Edward Wimmers, and Jun Yang. An optimizer for heterogeneous systems with non-standard data and search capabilities. *Special Issue on Query Processing for Non-Standard Data, IEEE Data Engineering Bulletin*, 19:37–43, December 1996.

[HKWY97] L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, 1997.

[HM93] J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *Intl Journal of Intelligent and Cooperative information Systems*, 2:51–83, 1993.

[JBHM+97] J.Hammer, M. Breunig, H.Garcia-Molina, S.Nestorov, V.Vassalos, and R. Yerneni. Template-based wrappers in the tsimmis system. In *Proc. ACM SIGMOD*, pages 532–535, 1997.

[K+93] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[KL89] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf.*, pages 134–46, Portland, OR, June 1989.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, pages 95–104, 1995.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, pages 251–262, 1996.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external processors. In *Proc. PODS*, pages 227–37, 1996.

[Mai86] D. Maier. A logic for objects. In J. Minker, editor, *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, Washington, DC, USA, August 1986.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for the rapid implementation of wrappers. In *Proc. DOOD Conf.*, pages 161–86, 1995.

[PGH] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. To appear in DAPD.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS*, 1996.

[Qia96] Xiaolei Qian. Query folding. In *Proc. ICDE*, pages 48–55, 1996.

[RSU95] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS Conf.*, pages 105–112, 1995.

[RSUV89] R. Ramakrishnan, Y. Sagiv, J.D. Ullman, and M.Y. Vardi. Proof tree transformations and their applications. In *Proc. PODS Conf*, pages 172–182, 1989.

[S^+] V.S. Subrahmanian et al. HERMES: A heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes/overview/paper.

[T^+90] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[TRV95] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. Technical report, INRIA, 1995.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*. Computer Science Press, New York, NY, 1989.

[VP97] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proc. VLDB Conf.*, pages 256–266, 1997.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

A RQDL Grammar

The following table contains the complete syntax of RQDL.

B Definition of the *valid* predicate

The next figure contains the rules that define the predicate *valid* (see subsection 9.3).

$$
\begin{array}{lll}
(0)\ \langle \textit{description} \rangle & ::= & (\langle \textit{ans\_query template} \rangle | \langle \textit{rule template} \rangle)* \\
(1)\ \langle \textit{ans\_query template} \rangle & ::= & \mathbf{ans}()\ \leftarrow\ \langle \textit{subgoal list} \rangle \\
(2)\ \langle \textit{rule template} \rangle & ::= & \langle \textit{predicate name} \rangle\ (\ \langle \textit{arguments} \rangle\ )\ \leftarrow\ \langle \textit{subgoal list} \rangle \\
(3)\ \langle \textit{subgoal list} \rangle & ::= & \langle \textit{subgoal} \rangle\ (,\ \langle \textit{subgoal} \rangle)* \\
(4)\ \langle \textit{subgoal list} \rangle & ::= & \langle \epsilon \rangle \% \textit{subgoal list may be emtpy} \\
(5)\ \langle \textit{subgoal} \rangle & ::= & \langle \textit{predicate} \rangle\ (\ \langle \textit{arguments} \rangle\ )\qquad \% \textit{predicate} \\
(6)\ \langle \textit{subgoal} \rangle & ::= & \langle \textit{metapredicate name} \rangle\ (\ \langle \textit{arguments} \rangle\ )\qquad \% \textit{metapredicate} \\
(7)\ \langle \textit{subgoal} \rangle & ::= & \langle \textit{token} \rangle\ (\ \langle \textit{arguments} \rangle\ )\qquad \% \textit{token as predicate} \\
(8)\ \langle \textit{arguments} \rangle & ::= & \langle \textit{argument} \rangle (, \langle \textit{argument} \rangle)* \\
(9)\ \langle \textit{argument} \rangle & ::= & \langle \textit{vector} \rangle | \langle \textit{variable} \rangle | \langle \textit{identifier} \rangle | \langle \textit{token} \rangle \\
(10)\ \langle \textit{predicate name} \rangle & ::= & \langle \textit{identifier} \rangle | \langle \textit{token} \rangle \\
(11)\ \langle \textit{metapredicate name} \rangle & ::= & \mathbf{union} | \mathbf{set\_item} | \mathbf{subset} \\
(12)\ \langle \textit{nonterminal name} \rangle & ::= & \langle \textit{identifier} \rangle \\
(13)\ \langle \textit{token} \rangle & ::= & \$ \langle \textit{identifier} \rangle
\end{array}
$$

Figure 8: RQDL syntax

$$
\begin{array}{l}
valid(u(T_1,T_2),T_1,T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2 \\
valid(u(T_2,T_1),T_1,T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1 \\
valid(T,T,T) \leftarrow tuple(N_1,T) \\
valid(u(T_1,T),u(T_1,T_1'),T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2, valid(T,T_1',T_2) \\
valid(u(T_2,u(T_1,T_1')),u(T_1,T_1'),T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1 \\
valid(u(T,T_1),u(T,T_1),T) \leftarrow tuple(N,T), valid(u(T,T_1),T,T_1) \\
valid(u(T_1,u(T_2,T_2')),T_1,u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2 \\
valid(u(T_2,T),T_1,u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1, valid(T,T_1,T_2') \\
valid(u(T,T_1),T,u(T,T_1)) \leftarrow tuple(N,T), valid(u(T,T_1),T,T_1) \\
valid(u(T_1,u(T_2,T)),u(T_1,T_1'),u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2, valid(T,T_1',T_2') \\
valid(u(T_2,u(T_1,T)),u(T_1,T_1'),u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1, valid(T,T_1',T_2') \\
valid(u(T,T'),u(T,T_1),u(T,T_2)) \leftarrow tuple(N,T), valid(T',T_1,T_2)
\end{array}
$$

Figure 9: Default rules for the generation of valid $u - terms$.