Describing and Using Query Capabilities of Heterogeneous Sources

Vasilis Vassalos*
Computer Science Dept.
Stanford University
vassalos@db.stanford.edu

Yannis Papakonstantinou
Computer Science and Engineering Dept.
University of California, San Diego
yannis@cs.ucsd.edu

Abstract

Information integration systems have to cope with the different and limited query interfaces of the underlying information sources. First, the integration systems need descriptions of the query capabilities of each source, i.e., the set of queries supported by each source. Second, the integration systems need algorithms for deciding how a query can be answered given the capabilities of the sources. Third, 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 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. Finally, we propose new improved algorithms for the problem of answering queries using these descriptions.

1 Introduction

Users and applications today must integrate multiple heterogeneous information systems, many of which are not conventional SQL database management systems. Examples of such systems are Web sources with forms

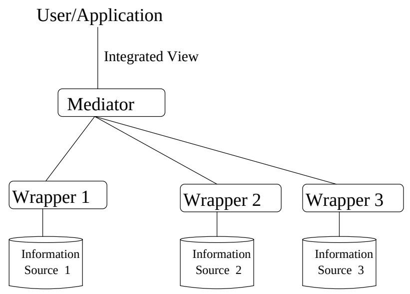

Figure 1: A common architecture for integration

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. Such systems need descriptions of the query capabilities of the participating sources, *i.e.*, descriptions of the set of queries that each source supports. They also need algorithms for adapting to the diverse capabilities of the sources as specified by the descriptions.

To further motivate the need for source descriptions, let us consider the typical integration architecture of Figure 1. *Mediators* decompose incoming client queries, which are expressed in some common query language, into new common-language queries which are supported by the *wrappers*. Then the wrappers translate the incoming queries into source-specific queries and commands. Both mediators and wrappers need the descriptions:

- The mediators use the description to adapt to the query capabilities of the sources. For example, 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 requests the managers

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

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

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 mediator knows that all the data needed for answering this query reside on "lookup" [1] but finding out *how* to retrieve this data is nontrivial. The query processing algorithm must infer from the description that only selection queries can be submitted and then can come up with the following plan for retrieving the required data: First the mediator retrieves the set of managers of *Java* employees, then the set of managers of *Database* employees, and finally it intersects the two sets.

- The wrappers 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* [1, 2] that perform the translation. Using this approach, in the TSIMMIS project at Stanford [3] we have wrapped a number of real life bibliographic sources.

What is an appropriate language for describing the set of supported queries and its translation to source-specific queries? Since we need descriptions of supported queries along with translating actions, Yacc programs look like a valid candidate. However, Yacc programs do not capture the logical properties of queries — they perceive queries as mere strings. This behavior imposes limitations on both the mediators and the wrappers. For example, the description may specify that an acceptable `WHERE` clause is "`lname='Smith' AND fname='John'`". The wrapper then does not know how to translate a query that asks for "`fname='John' AND lname='Smith'`" because it ignores the commutativity of `AND`. The mediator faces even more severe problems, as we discuss in Section 8.

Since we want to preserve the salient connection between description and translation in Yacc, we propose the use of Datalog variants as a more powerful description language. In particular, context-free grammar rules can be thought of as Datalog rules with 0-arity predicates. 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 [4, 5]; it is also clearly related to the Answering Queries Using Views problem [6, 7, 8] (see Section 3). In this paper, we focus on sources that support conjunctive queries, *i.e.*, their capabilities are a subset of *CQ* [9]. The contributions of this paper are as follows:

[1] In many cases the data reside at multiple sources and the mediator may have to locate them first. However, finding *where* are the important data is a problem orthogonal to *how* they can be obtained. In this paper we only deal with the latter problem.

- 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 [8], 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.
- We describe and extend RQDL, 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. (The algorithm presented in [4] only works for certain classes of RQDL descriptions.) From a theoretical point of view, it clarifies the difference in expressive power between RQDL and p-Datalog.

The next section introduces the p-Datalog description language. Section 3 describes the algorithm run by the wrappers and a CBR algorithm run by the mediators. Section 4 discusses expressive power issues. Section 5 introduces RQDL. Section 6 describes the reduction of RQDL to p-Datalog with function symbols and Section 7 describes the wrapper and mediator algorithms for RQDL. Section 8 discusses the related work. Section 9 gives conclusions and future work. For a more formal and complete presentation of these topics (and for all proofs of results in this paper) please see [10].

2 The p-Datalog Source Description Language

It is well known that the most popular real-life query languages, like SPJ queries [9] and Web-based query forms are equivalent to conjunctive queries. A Datalog program is a natural encoding of conjunctive queries: it "represents" all its expansions. First, we describe informally a Datalog-based source description language and demonstrate it with examples. A formal definition follows in the next subsection.

In the simple case of weak information sources, 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 [2, 8, 11]. For example, query forms found in Web sites expect constants in some of their fields [11]. Without loss of generality, we assume the existence of a designated predicate *ans* that is the head of all the parameterized queries of the description.

Example 2.1 Consider a bibliographic information source that provides information about books. This source exports a "book" predicate $b(isbn, author, title, publisher, year, pages)$.

The source also exports "indexes," on authors $au\_i(au\_name, isbn)$, publishers $pub\_i(pub, isbn)$ and titles $titl\_i(t\_word, isbn)$. Conceptually, the tuple (X, Y) is in au_i 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 $titl\_i$ 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.

$$ans(I, A, T, P, Y, Pg) \leftarrow b(I, A, T, P, Y, Pg), au\_i(\$c, I)$$
$$ans(I, A, T, P, Y, Pg) \leftarrow b(I, A, T, P, Y, Pg), titl\_i(\$c, I)$$
$$ans(I, A, T, P, Y, Pg) \leftarrow b(I, A, T, P, Y, Pg), pub\_i(\$c, I)$$

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

$$ans(I, A, T, P, Y, Pg) \leftarrow b(I, A, T, P, Y, Pg), au\_i(\texttt{'Doe'}, I)$$

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

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:

$$\begin{array}{lll} ans(I, A, T, P, Y, Pg) & \leftarrow b(I, A, T, P, Y, Pg), & \\ & \quad ind_1(I), ind_2(I), ind_3(I) & (1) \\ ind_1(I) & \leftarrow titl\_i(\$c, I) & \\ ind_1(I) & \leftarrow \epsilon & (2) \\ ind_2(I) & \leftarrow au\_i(\$c, I) & (3) \\ ind_2(I) & \leftarrow \epsilon & \\ ind_3(I) & \leftarrow pub\_i(\$c, I) & \\ ind_3(I) & \leftarrow \epsilon & (4) \end{array}$$

ϵ denotes an empty body, *i.e.*, ϵ–rules have an empty expansion. Notice that ϵ–rules are unsafe [12]. 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 linear in the number of the available indexes, whereas the number of possible expansions is exponential. The query

$$\begin{array}{l} ans(I, A, T, P, Y, Pg) \leftarrow b(I, A, T, P, Y, Pg), \\ \qquad\qquad\qquad au\_i(\texttt{'Doe'}, I) \end{array}$$

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 `'Doe'`. 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 (formal definitions follow the example).

Example 2.3 Consider again the bibliographical source of Example 2.1. Assume that there is an abstract index $ab\_i(ab\_word, I)$ 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.

$$\begin{array}{ll} ans(I, A, T, P, Y, Pg) & \leftarrow b(I, A, T, P, Y, Pg), ind(I) \\ ind(I) & \leftarrow ab\_i(\$c, I) \\ ind(I) & \leftarrow ind(I), ab\_i(\$c, I) \end{array}$$

□

We now give the p-Datalog semantics. We assume familiarity with Datalog, e.g. [12, 9]. Besides the constant and variable sorts, we use a third disjoint set of symbols, the set of token variables or *tokens*.

Definition: A *parameterized 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 $u, u_1, u_2, \ldots, u_n$ 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: 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:

[2] Source predicates are the EDB predicates of our descriptions.

- 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. □

The above definitions can easily be extended to accommodate more than one "designated" predicates (like ans). 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.

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" [1]. 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 or in the head of the query. Equalities between variables are made explicit through the use of an *equality* predicate $e(X, Y)$.
- No constants or tokens appear among the ordinary[3] subgoals. Every constant or token is replaced by a unique variable that is equated to the constant or token through an e subgoal.
- No variables appear only in an e subgoal of a query.

We treat the e subgoal not as a built-in predicate, but as a source predicate. We call rules that obey the above conditions *rectified* rules and the process that transforms any rule to a rectified rule *rectification*. We call the inverse procedure *de-rectification*.

In the next two sections 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 e relation.

3 Query expressibility and CBR with p-Datalog descriptions

In this section we present an algorithm for query expressibility of p-Datalog descriptions. We also give an elegant and improved solution to the problem of answering queries using an infinite set of views [11].

Our QED (Query Expressibility Decision) algorithm is an extension of the classic algorithm for deciding query containment in a Datalog program that appears in [13] (also see [12]). Our 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 on books $b(isbn, author, publisher)$, a word index on titles, $titl\_i(t\_word, isbn)$ and an author index $au\_i(au\_name, isbn)$. Also assume that the query capabilities of the source are described by the following p-Datalog program:

$$\begin{array}{ll} ans(A,P) & \leftarrow b(I,A,P), ind_1(I_1), ind_2(I_2), \\ & \quad e(I,I_1), e(I,I_2) \\ ind_1(I) & \leftarrow title\_i(V,I), e(V,\$c) \\ ind_1(I) & \leftarrow \epsilon \\ ind_2(I) & \leftarrow au\_i(V,I), e(V,\$c) \\ ind_2(I) & \leftarrow \epsilon \end{array}$$

Let us consider the query Q'

$$ans(X,Y) \leftarrow b(I,X,Y), title\_i(\texttt{'Zen'}, I), au\_i(\texttt{'Doe'}, I)$$

First we produce its rectified equivalent Q':

$$\begin{array}{l} ans(X,Y) \leftarrow b(I,X,Y), title\_i(V_1,I_1), au\_i(V_2,I_2), \\ \quad e(V_1,\texttt{'Zen'}), e(V_2,\texttt{'Doe'}), e(I,I_1), e(I,I_2) \end{array}$$

Apparently the above query is expressible by the description. Intuitively, the QED 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 e subgoals into the DB. If, for example, subgoals $e(X,Y), e(X,Z)$ exist in the query, we will generate "frozen" facts for all implicit equalities as well, *i.e.*, $e(Y,X), e(Y,Z)$ etc. In the interests of space and clarity, we will write $e(X,Y,Z)$ to mean that all the previously mentioned facts are in the DB [4]. We will use this shorthand notation in the rest of this paper. The DB for our running example is then

$$b(\bar{\imath},\bar{x},\bar{y}), title\_i(\bar{v}_1,\bar{\imath}_1), au\_i(\bar{v}_2,\bar{\imath}_2), e(\bar{\imath},\bar{\imath}_1,\bar{\imath}_2), \\ e(\bar{v}_1,\texttt{'Zen'}), e(\bar{v}_2,\texttt{'Doe'})$$

[4]It is easy to see that $e(Y_1,\ldots,Y_l)$ is a subset of $e(X_1,\ldots,X_m)$ iff $\forall i \leq l, Y_i \in \{X_1,\ldots,X_m\}$.

The QED algorithm then evaluates the Datalog program on the DB, deriving more facts for the IDB's. In addition, it keeps track of the set of frozen facts, called *supporting set*, that are used for deriving each fact. Here is the set of facts and supporting sets derived by a particular evaluation of the Datalog program.

$$\begin{array}{ll}
< ind_1(I), & \{\} > \\
< ind_2(I), & \{\} > \\
(5) :< ans(\bar{x}, \bar{y}), & \{b(\bar{\imath}, \bar{x}, \bar{y}), e(\bar{\imath}, \bar{\imath})\} > \\
< ind_1(\bar{\imath}_1), & \{title_i(\bar{v}_1, \bar{\imath}_1), e(\bar{v}_1, '\mathtt{Zen}')\} > \\
< ind_2(\bar{\imath}_2), & \{au_i(\bar{v}_2, \bar{\imath}_2), e(\bar{v}_2, '\mathtt{Doe}')\} > \\
(6) :< ans(\bar{x}, \bar{y}), & \{b(\bar{\imath}, \bar{x}, \bar{y}), title_i(\bar{v}_1, \bar{\imath}_1), e(\bar{v}_1, '\mathtt{Zen}'), \\
 & au_i(\bar{v}_2, \bar{\imath}_2), e(\bar{v}_2, '\mathtt{Doe}'), e(\bar{\imath}, \bar{\imath}_1, \bar{\imath}_2)\} >
\end{array}$$

We call <fact,supporting set> pairs *extended facts.* 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 *corresponding* containing query from the extended fact by translating "frozen" facts back into subgoals. In our running example, the two containing queries[5] correspond to (5) and (6). 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 (6) is

$$ans(X, Y) \leftarrow b(I, X, Y), title\_i('\mathtt{Zen}', I), au\_i('\mathtt{Doe}', I)$$

which is equivalent 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* [13, 12] 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. The set of supporting facts for f is the set of leaves of a proof tree [12] for f. 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.

The algorithm keeps for each produced fact only the *maximal* supporting sets. As a result, it produces the set of expansions of the description program that most tightly contain the given query. We call these expansions the *minimal containing queries.* Notice that there may be more than one minimal containing queries for a given query and a given description. Moreover, notice that a minimal containing query for Q contains the maximal number of (non-redundant) subgoals of Q among containing queries of Q.

These are the only expansions that could be equivalent to the given query. If that set is nonempty, obviously there exists a containing query for Q with respect to P. Moreover, Q is expressible by P *iff* one of the minimal containing queries in the set is equivalent to Q. Algorithm QED is presented in detail in [10].

Theorem 3.2 (Correctness) Algorithm QED terminates and produces the set of minimal containing queries of input query Q.

Using algorithm QED we can decide whether Q is expressible by P:

Lemma 3.3 Q is expressible by P iff the set of supporting facts for the frozen head of Q is identical[6] to the canonical DB for Q.

Theorem 3.4 (Complexity) Algorithm QED terminates in time exponential to the size of the description and the size of the query.

Translation: Let us consider the case of a wrapper that receives a query. It is easy to see that we could extend Algorithm QED 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.

Mediators are faced with a different problem than wrappers: Given the descriptions for one or more wrappers, the mediator has to answer the user query by "issuing" only queries expressible by the wrapper descriptions. That is the Capabilities-Based Rewriting (CBR) problem [4, 5]. 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 [8].

Our CBR algorithm proceeds in two steps. The first step uses Algorithm QED to generate the finite set of expansions. The second step uses an algorithm for answering queries using views [6, 14] to combine some of these expansions to answer the query. 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. The time complexity of our CBR algorithm is nondeterministic exponential in the size of the query and the description, which is a considerable improvement over the previously known solution [8].

Theorem 3.5 (CBR) Assume we have a query Q and a p-Datalog description P, and let $\{Q_i\}$ be the result of applying Algorithm QED on Q and P. There exists a rewriting Q' of Q, such that $Q' \equiv Q$, using any $\{Q_j | Q_j \text{ is expressible by } P\}$ *if and only if* there exists a rewriting Q'' , such that $Q'' \equiv Q$, using only $\{Q_i\}$.

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

[5] The algorithm actually uses pruning to eliminate (5) from the output.

[6] After de-rectification of both.

[6] and there are known algorithms for solving it [6, 14]. Using the set $\{Q_i\}$ of minimal containing queries as input to one of these algorithms, we obtain a solution to the CBR problem for p-Datalog that is non-deterministic exponential, since $|\{Q_i\}|$ is exponential in the size of the p-Datalog description and the user query.

4 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 4.1 There exist *recursive* sets of conjunctive queries that are not expressible by any p-Datalog description.

However, the practical question is whether there exist 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.

Theorem 4.2 Let k be some integer, and let $\mathbf{S}$ be a database schema. There exists a p-Datalog program that describes all conjunctive queries over $\mathbf{S}$ with at most k variables.

Let us now discuss the limitations of p-Datalog. It is obvious that 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 that one "answer" predicate. However, even in that case, a given program would still bound the arities of answers. A more serious restriction is due to the fixed number of variables that occur in any one of the rules of the program. In particular, even if we focus on arity-0 results, *i.e.*, boolean queries, p-Datalog is limited.

Theorem 4.3 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.)

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

5 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 [4] and used for describing query capabilities of information sources. [4] shows its advantages over Datalog when it is used for descriptions that are not schema-specific.

In this paper we extend RQDL and prove that it allows us to describe large sets of queries. For example, we prove that 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. That allows us to give a complete solution to the CBR problem for RQDL.

To support schema independent descriptions, RQDL allows the use of *predicate tokens* 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 "carry" 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).

In the rest of this paper we will be using named attributes in our conjunctive queries. For example, consider the relation *book* with schema *book*(*title*, *id*). Instead of writing

$$ans() \leftarrow book(X, Z), e(X, '\mathtt{Data}')$$

we will write

$$ans() \leftarrow book(title : X, id : Z), e(X, '\mathtt{Data}')$$

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 5.1 illustrates RQDL's ability to describe source capabilities without referring to a specific schema.

Example 5.1 Consider a source that accepts queries that refer to exactly one relation and pose exactly one selection condition over the source schema.

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

The above RQDL description[7] describes, among others, the query

$$ans() \leftarrow b(f1 : X, f2 : Z), e(X, '\mathtt{Data}')$$

because, intuitively, we can map $\$r$ to relation b, $\vec{V}$ to the set of attribute-variable pairs $\{f1 : X, f2 : Z\}$, X' to X, and $\$c$ to `'Data'`. The *metapredicate* $item(\vec{V}, \_A, X')$ declares that the variable X' maps to

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

any 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 $\$r$. Moreover, the variable $\_A$ maps to the attribute name of the variable that X' maps to in $\vec{V}$. No condition is placed on $\_A$ and hence X' can be either X or Z. We call the process described above, that maps an RQDL rule into a conjunctive query, an *instantiation* of the RQDL rule.

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:

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

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

Example 5.2 The following RQDL program describes all boolean conjunctive queries over any schema.

$$\begin{aligned}
ans() &\leftarrow cond(\vec{V})\\
cond(\vec{V}) &\leftarrow \$p(\vec{V_1}), cond(\vec{V_2}), union(\vec{V}, \vec{V_1}, \vec{V_2})\\
cond(\vec{V}) &\leftarrow item(\vec{V}, _P, X), e(X, \$c), cond(\vec{V})\\
cond(\vec{V}) &\leftarrow item(\vec{V}, _P_1, X_1), item(\vec{V}, _P_2, X_2),\\
&\qquad e(X_1, X_2), cond(\vec{V})\\
cond(\vec{V}) &\leftarrow \$p(\vec{V})
\end{aligned}$$

The *metapredicate* $union(\vec{V}, \vec{V_1}, \vec{V_2})$ declares that $\vec{V}$ is mapped to the union of the sets of attribute-variable pairs that $\vec{V_1}$ and $\vec{V_2}$ are mapped to.

Given any rectified conjunctive query, the description above describes it, *i.e.*, for any conjunctive query Q, there exists an expansion of the rules that can be *instantiated* to Q. □

The semantics of RQDL are an extension of the semantics of p-Datalog described in Section 2. Informally, we say that a conjunctive query Q is *described* by an RQDL description P if Q is an instantiated expansion of P. We say that Q is *expressible by* P, if there exists Q' described by P, such that $Q \equiv Q'$.

The next section describes the reduction of RQDL descriptions to p-Datalog programs with function symbols. Section 7 proceeds to give algorithms for query expressibility by RQDL description and for the CBR problem for RQDL descriptions.

6 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. A brute force approach, such as the one proposed by [4], where vectors actually match with sets during the derivation, quickly leads to serious problems. In particular, the brute force approach breaks down in the presence of unsafe rules that have vectors in the head.

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.

Reduction to standard schema: We conceptually reduce the original database into a standard schema database where the relation names and the attribute names appear as data. We illustrate the reduction through an example.

Example 6.1 Consider the following database DB with schema $b(au, id)$ and $f(subj, id)$.

b	
au	*id*
'Doe'	1
'Sax'	2

f	
subj	*id*
'Law'	1
'Art'	2

The standard schema consists of two relations, a *tuple* relation $t(rel, tuple\ id)$ and an *attribute* relation $a(tuple\ id, attr, value)$. For this example, the corresponding standard schema database DB' is

t	
rel	*tuple id*
b	b('Doe',1)
b	b('Sax',2)
f	f('Law',1)
f	f('Art',2)

a		
tuple id	*attr*	*value*
b('Doe',1)	au	'Doe'
b('Doe',1)	id	1
b('Sax',2)	au	'Sax'
b('Sax',2)	id	2
f('Law',1)	subj	'Law'
f('Law',1)	id	1
f('Art',2)	subj	'Art'
f('Art',2)	id	2

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

Reduction of CQ 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 sense that 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 6.1.

$$\begin{aligned} ans() \leftarrow\ & b(\mathtt{au} : X, \mathtt{id} : S_1), f(\mathtt{subj} : A, \mathtt{id} : S_2), \\ & e(S_1, S_2), e(A, '\mathtt{Art}') \end{aligned}$$

Query Q is reduced into the following query Q':

$$\begin{aligned} t(ans, ans()) \leftarrow\ & t(b, B), t(f, F), a(B, \mathtt{id}, S_1), e(S_1, S_2), \\ & a(F, \mathtt{id}, S_2), a(F, \mathtt{subj}, A), e(A, '\mathtt{Art}') \end{aligned}$$

Notice that for every ordinary subgoal we introduce a t subgoal and invent a tuple id. For every attribute we introduce an a 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 IDs of books if their subject is `'Art'`.

$$\begin{aligned} ans(\mathtt{au} : X, \mathtt{id} : S_1) \leftarrow\ & b(\mathtt{au} : X, \mathtt{id} : S_1), e(S_1, S_2), \\ & f(\mathtt{subj} : A, \mathtt{id} : S_2), e(A, '\mathtt{Art}') \end{aligned}$$

This query is reduced to the following program Q' where the first rule defines the t part of the standard schema answer and the last two rules describe the a part.

$$\begin{aligned} t(ans, ans(X, S_1)) \leftarrow\ & t(b, B), t(f, F), a(B, \mathtt{id}, S_1), \\ & a(F, \mathtt{id}, S_2), a(B, \mathtt{au}, X), \\ & e(S_1, S_2), a(F, \mathtt{subj}, A), \\ & e(A, '\mathtt{Art}') \\ a(ans(X, S_1), \mathtt{au}, X) \leftarrow\ & t(b, B), t(f, F), a(B, \mathtt{id}, S_1), \\ & a(F, \mathtt{id}, S_2), a(B, \mathtt{au}, X), \\ & e(S_1, S_2), a(F, \mathtt{subj}, A), \\ & e(A, '\mathtt{Art}') \\ a(ans(X, S_1), \mathtt{id}, S_1) \leftarrow\ & t(b, B), t(f, F), a(B, \mathtt{id}, S_1), \\ & a(F, \mathtt{id}, S_2), a(B, \mathtt{au}, X), \\ & e(S_1, S_2), a(F, \mathtt{subj}, A), \\ & e(A, '\mathtt{Art}') \end{aligned}$$

It is easy to see that under obvious constraints we can go from the standard schema query to the "original" query.

Reduction of RQDL programs to standard schema Datalog programs[8]: Based on the ideas presented in the previous paragraphs, 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 attribute-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 $\vec{V_2}$. The reduction carefully produces a program which terminates despite the use of the u function. We will illustrate the reduction through an example.

[8] With function symbols.

Example 6.2 The description of Example 5.2 describes all boolean conjunctive queries. It reduces into the following p-Datalog description (with function symbols):

$$\begin{aligned} & t(ans, ans()) \leftarrow cond(T) \\ & cond(T) \leftarrow t(\$p, T_1), cond(T_2), v(T, T_1, T_2) \\ & cond(T) \leftarrow a(T', \_P, X), e(X, \$c), cond(T), e(T', T), \\ & cond(T) \leftarrow a(T_1, \_P_1, X_1), a(T_2, \_P_2, X_2), e(X_1, X_2), \\ & \qquad cond(T), e(T, T_1), e(T, T_2) \\ & cond(T) \leftarrow t(\$p, T) \end{aligned}$$

The reduction of each rule is independent from the reduction of other rules. In the reduction of the first rule, notice that the vector variable has been "replaced" by the variable T which matches with a tuple id. In the second rule, notice that we reduced $\vec{V}$ to T, which is "produced" by the predicate v, given T_1 and T_2. v constructs a new "valid" tuple id of a restricted form, that has associated with it all the attributes associated with T_1 or T_2. The role of v is to simulate the union that it replaces, by not allowing generation of arbitrary u terms. Assuming that there is a total order for the tuple ids of the standard schema database, $v(T, T_1, T_2)$ creates a u term in which all tuple ids appear in sorted order, and none are repeated. In particular, $v(T, u(t_2, u(t_3, t_4)), u(t_3, t_5))$ will bind T to $u(t_2, u(t_3, u(t_4, t_5)))$. Each description has to include the rules that define v. These rules are given in [10].

Finally, the description has to include the "standard" rules of Fig. 2, that make sure that all attributes of tuples with ids T_1 and T_2 are also attributes of tuples with id T, constructed from T_1, T_2.

$$\begin{aligned} a(T, A, X) \leftarrow a(T_1, A, X), v(T, T_1, T_2) \\ a(T, A, X) \leftarrow a(T_2, A, X), v(T, T_1, T_2) \end{aligned}$$

Figure 2: Default rules for generation of *attr* tuples

In the reduction of the third rule of the description, notice that the metapredicate $item(V, \_P, X)$ is reduced to the predicate $a(T, \_P, X)$. □

Theorem 6.3 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.

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 Section 3.

7 Expressibility and CBR with RQDL descriptions

Let us start by clarifying the semantics of p-Datalog with functions. We will denote p-Datalog with functions with *p-Datalog*^{f}. The set of expressible queries of a p-Datalog^f program is defined the same way as for a p-Datalog program (see Section 2), with one difference: there are two "designated" predicates, the predicates *tuple* and *attr*.

Let us note that the transformation of a conjunctive query to refer to the standard schema results in a set of standard schema queries with the same body. The following theorem states that the reduction described in the previous section is correct:

Theorem 7.1 Let us consider a query Q over some schema and an RQDL description P. Also let $Q' = \{Q_i\}$ be the set of standard schema queries that is the reduction of Q and P' be the standard schema reduction of P. Then Q is expressible by P *if and only if* each Q_i is expressible by P', *i.e.*, iff set Q' is expressible by P'.

Because of Theorems 6.3 and 7.1, we can use Algorithm QED to answer the expressibility question in RQDL. The idea is to generate all possible extended facts for *tuple* and *attr* and then, as in Section 3, check whether (i) the necessary "frozen" *tuple* and *attr* facts are produced and (ii) their corresponding queries are equivalent to the Q_i's.

Example 7.2 Consider the query

$$Q : ans(a : X) \leftarrow books(au : X, titl : Y)$$

and the description

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

The standard schema canonical DB is
$t(books, t_0), a(t_1, au, \bar{x}), a(t_2, titl, \bar{y}), e(t_0, t_1, t_2)$
The reduction of the description (after rectification) is

$$\begin{array}{rl} t(ans, ans(X)) & \leftarrow t(\$r, T), a(T_1, au, X), a(T_2, titl, Y), \\ & \quad e(T, T_1), e(T, T_2) \\ a(ans(X), a, X) & \leftarrow t(\$r, T), a(T_1, au, X), a(T_2, titl, Y), \\ & \quad e(T, T_1), e(T, T_2) \\ t(ans, ans(Y)) & \leftarrow t(\$r, T), a(T_1, au, X), a(T_2, titl, Y), \\ & \quad e(T, T_1), e(T, T_2) \\ a(ans(Y), b, Y) & \leftarrow t(\$r, T), a(T_1, au, X), a(T_2, titl, Y), \\ & \quad e(T, T_1), e(T, T_2) \end{array}$$

Notice that we didn't include the rules of Figure 2 or the rules for predicate v in the reduced description, since the original description didn't contain any metapredicates.

After we run Algorithm QED on the canonical DB, the following extended facts are produced:

$$\begin{array}{ll} (7) < t(ans, ans(x)), & \{t(books, t_0), a(t_1, au, x), \\ & \quad a(t_2, titl, y), e(t_0, t_1, t_2)\} > \\ (8) < a(ans(x), a, x), & \{t(books, t_0), a(t_1, au, x), \\ & \quad a(t_2, titl, y), e(t_0, t_1, t_2)\} > \\ < t(ans, ans(y)), & \{t(books, t_0), a(t_1, au, x), \\ & \quad a(t_2, titl, y), e(t_0, t_1, t_2)\} > \\ < a(ans(y), b, y), & \{t(books, t_0), a(t_1, au, x), \\ & \quad a(t_2, titl, y), e(t_0, t_1, t_2)\} > \end{array}$$

The output of the algorithm includes the exact two conjunctive queries (the corresponding queries to the extended facts (7) and (8)) that are the reduction of Q. Q is therefore expressible by our description, by Theorem 7.1. □

The CBR problem for RQDL: Before attempting to solve the Capabilities-Based Rewriting problem for CBR, let us make the following observations: Algorithm QED produces *tuple* and *attr* extended facts with maximal supporting sets. If there exists an extended fact $< attr(t, attr\_name, x), S_t >$ in the result, then there also exists an extended fact $< tuple(r, t), S_t >$ for some table r.

We solve the CBR problem for a given query in two steps:

- We generate the set of relevant described queries from the output of the Algorithm QED, 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 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.
- Given a query (over some schema) and a number of relevant described queries over the same schema, we can apply an answering queries using views algorithm [14, 6], where the views are the relevant described queries.

8 Related Work

The different and limited query capabilities of information sources are an important problem for integration systems. In this section we discuss the approaches taken by various systems and we also discuss some theoretical work in this area.

[2] suggested a language for describing query capabilities. The expressive power of the language is equivalent to Datalog. [8] proposed 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. We already mentioned that we improved upon the result of [8] for the problem of answering a query using an infinite number of views.

RQDL was proposed by [4] to allow capabilities descriptions that are not schema specific. The Information Manifold [11] 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 less than p-Datalog.

The DISCO system [15] 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. There is an ongoing DISCO effort to develop more sophisticated algorithms.

The Garlic system [5, 16] combines capabilities-based rewriting with cost-based optimization. The assumption is made that all the variables mentioned in a query are always made 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.

Finally, RQDL's handling of constructed tuple ids is based on a use of Skolem functions that is close to the ideas in [17, 18]

9 Conclusions and Future Work

We discuss the problems of (i) describing the query capabilities of sources and (ii) using the descriptions for source wrapping and mediation. We first consider 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.

We then 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. A direct consequence of our result is that p-Datalog cannot model a fully-fledged relational DBMS.

We subsequently describe and extend RQDL, which is a provably more expressive language than p-Datalog. We provide 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 problems.

We have focused exclusively on conjunctive queries. We plan to extend our work to non-conjunctive queries, *i.e.*, queries involving aggregates and negation. Combining our CBR algorithm with cost-based query optimization also presents interesting challenges.

Acknowledgements: We are grateful to Serge Abiteboul for his help in formalizing the presentation of p-Datalog and the proof of Theorem 4.3. We also thank Jeff Ullman for many fruitful discussions and Luis Gravano for comments on a previous draft of this paper.

References

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

[2] 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.

[3] 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 Conf.*, pages 532–535, 1997.

[4] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS Conf.*, pages 170–181, 1996.

[5] L. Haas, D. Kossman, E.L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. To appear in VLDB '97.

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

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

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

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

[10] V. Vassalos and Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources (extended version). Available online from `http://www-db.stanford.edu/pub/papers/`, file `query-cap-ext.ps`.

[11] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, 1996.

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

[13] 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.

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

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

[16] M. Tork Roth and P.M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. To appear in VLDB '97.

[17] 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.

[18] 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.