, , 1–42 ()

Capabilities-Based Query Rewriting in Mediator Systems * **

YANNIS PAPAKONSTANTINOU†, ASHISH GUPTA‡, LAURA HAAS§

Editor:

Abstract. Users today are struggling to integrate a broad range of information sources providing different levels of query capabilities. Currently, data sources with different and limited capabilities are accessed either by writing rich functional wrappers for the more primitive sources, or by dealing with all sources at a "lowest common denominator". This paper explores a third approach, in which a mediator ensures that sources receive queries they can handle, while still taking advantage of all of the query power of the source. We propose an architecture that enables this, and identify a key component of that architecture, the *Capabilities-Based Rewriter (CBR).* The CBR takes as input a description of the capabilities of a data source, and a query targeted for that data source. From these, the CBR determines component queries to be sent to the sources, commensurate with their abilities, and computes a plan for combining their results using joins, unions, selections, and projections. We provide a language to describe the query capability of data sources and a plan generation algorithm. Our description language and plan generation algorithm are schema independent and handle SPJ queries. We also extend CBR with a cost-based optimizer. The net effect is that we prune without losing completeness. Finally we compare the implementation of a CBR for the Garlic project with the algorithms proposed in this paper.

Keywords: heterogeneous sources, mediator systems, query rewriting, query containment, cost optimization

1. Introduction

Organizations today must integrate multiple heterogeneous information sources, many of which are not conventional SQL database management systems. Examples of such information sources include bibliographic databases, object repositories, chemical structure databases, WAIS servers, etc. Some of these systems provide powerful query capabilities, while others are much more limited. A new challenge for the database community is to allow users to query this data using a single powerful query language, with location transparency, despite the diverse capabilities of the underlying systems.

* Research partially supported by Wright Laboratories, Wright Patterson AFB, ARPA Contract F33615-93-C-1337.

** This paper is an extended version of a paper published in International Conference on Parallel and Distributed Information Systems, December 1996.

† UCSD, Computer Science & Engineering, La Jolla, CA 92093-0114, yannis@cs.ucsd.edu
‡ Junglee Corp., 4149B El Camino Way, Palo Alto, CA 94306, ashish@junglee.com
§ IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, laura@almaden.ibm.com

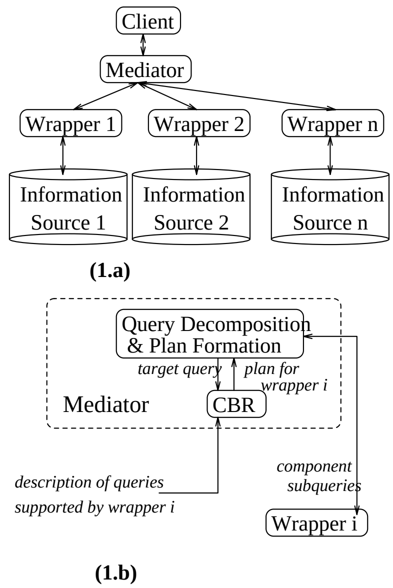

Figure 1. (a) A typical integration architecture. (b) CBR-mediator interaction.

Figure (1.a) shows one commonly proposed integration architecture [2, 16, 4, 1]. Each data source has a *wrapper*, which provides a view of the data in that source in a common data model. Each wrapper can translate queries expressed in the common language to the language of its underlying information source. The *mediator* provides an integrated view of the data exported by the wrappers. In particular, when the mediator receives a query from a client, it determines what data it needs from each underlying wrapper, sends the wrappers individual queries to collect the required data, and combines the responses to produce the query result.

This scenario works well when all wrappers can support any query over their data. However, in the types of systems we consider, this assumption is unrealistic. It leads to extremely complex wrappers, needed to support a powerful query interface against possibly quite limited data sources. For example, in many systems the relational data model is taken as the common data model, and all wrappers must provide a full SQL interface, even if the underlying data source is a file system, or a hierarchical DBMS. Alternatively, this assumption may lead to a "lowest common denominator" approach in which only simple queries are sent to the wrappers. In this case, the search capabilities of more sophisticated data sources are not exploited, and hence the mediator is forced to do most of the work, resulting in unnecessarily poor performance. We would like to have simple wrappers that accurately reflect the search capabilities of the underlying data source. To enable this, the mediator must recognize differences and limitations in capabilities, and ensure that wrappers receive only queries that they can handle.

For Garlic [2], an integrator of heterogeneous multimedia data being developed at IBM's Almaden Research Center, such an understanding is essential. Garlic needs to deal efficiently with the disparate data types and querying capabilities needed by applications as diverse as medical, advertising, pharmaceutical research, and computer-aided design. In our model, a wrapper is capable of handling some set of queries, known as the *supported queries* for that wrapper. When the mediator receives a query from a client, it decomposes it into a set of queries, each of which references data at a single wrapper. We call these individual queries *target queries* for the wrappers. A target query need not be a supported query; it may sometimes be necessary to further decompose it into simpler supported *Component SubQueries (CSQs)* in order to execute it. A *plan* combines the results of the CSQs to produce the answer to the target query.

To obtain this functionality, we explored a *Capabilities-Based Rewriter (CBR)* module (Figure 1.b) as part of the Garlic query engine (mediator). The CBR uses a description of each wrapper's ability, expressed in a special purpose *query capabilities description language*, to develop a plan for the wrapper's target query.

The mediator decomposes a user's query into target queries q for each wrapper w without considering whether q is supported by w. It then passes q to the CBR for "inspection." The CBR compares q against the description of the queries supported by wrapper w, and produces a plan p for q, if either (i) q is directly supported by w, or (ii) q is computable by the mediator through a plan that involves selection, projection and join of CSQs that are supported by w. The mediator then combines the individual plans p into a complete plan for the user's query.

The CBR allows a clean separation of wrapper capabilities from mediator internals. Wrappers are "thin" modules that translate queries in the common model into source-specific queries.[1] Hence, wrappers reflect the actual capabilities of the underlying data sources, while the mediator has a general mechanism for interpreting those capabilities and forming execution strategies for queries. This paper focuses on the technology needed to enable the CBR approach. We first present a language for describing wrappers' query capabilities. The descriptions look like context-free grammars, modified to describe queries rather than arbitrary strings. The descriptions may be recursive, thus allowing the description of infinitely large supported queries. In addition, they may be schema-independent. For example, we may describe the capabilities of a relational database wrapper without referring to the schema of a specific relational database. An additional benefit of the grammar-like description language is that it can be appropriately augmented with actions to translate a target query to a query of the underlying information system. This feature has been described in [14] and we will not discuss it further in this paper.

The second contribution of this paper is an architecture for the CBR and an algorithm to build plans for a target query using the CSQs supported by the relevant wrapper. This problem is a generalization of the problem of determining if a query can be answered using a set of materialized queries/views [9, 18]. However, the CBR uses a description of potentially infinite queries as opposed to a finite set of materialized views. The problem of identifying CSQs that compute the target query has many sources of exponentiality even for the restricted case discussed by [9, 18]. The CBR algorithm uses optimizations and heuristics to eliminate sources of exponentiality in many common cases.

The third contribution of this paper, which does not appear in [15], is the incorporation of cost-based optimization into the CBR. The described cost-based prunings do *not* compromise the completeness of the algorithm, i.e., if there is a supported plan the algorithm will find it.

Finally, we compare the algorithms of this paper with the algorithms that were eventually implemented for the Garlic system. The implementation uses the extensible optimizer of Starburst, hence having an excellent framework for combining capabilities-based rewriting with cost-based optimization. On the other hand, the use of Starburst's optimizer required changes in the description language and corresponding changes in the algorithms.

In the next section, we present the language used to describe a wrapper's query capabilities. In Section 3 we describe the basic architecture of the CBR, identifying three modules: Component SubQuery Discovery, Plan Construction, and Plan Refinement. These components are detailed in Sections 4, 5 and 6, respectively. Section 7 discusses the combination of CBR with cost-based optimization. Section 8 compares the proposed CBR algorithms against the CBR implementation of Garlic. Section 9 summarizes the run-time performance of the CBR. Section 10 discusses related work. Finally, Section 11 concludes with some directions for future work in this area.

2. The Relational Query Description Language(RQDL)

RQDL is the language we use to describe a wrapper's supported queries. We discuss only Select-Project-Join queries in this paper. In section 2.1 we introduce the basic language features , followed in sections 2.2 and 2.3 by the extensions needed to describe infinite query sets and to support schema-independent descriptions. Section 2.4 introduces a normal form for queries and descriptors that increases the precision of the language. The complete language specification appears in Appendix A.1.

The description language focuses on conjunctive queries. We have found that it is powerful enough to express the abilities of many wrappers and sources, such as lookup catalogs and object databases. Indeed, it is more expressive than context-free grammars.[2]

2.1. Language Basics

An RQDL specification contains a set of *query templates*, each of which is essentially a parameterized query. Where an actual query might have a constant, the query template has a *constant placeholder*, allowing it to represent many queries of the same form. In addition, we allow the values assumed by the constant placeholders to be restricted by specifier-provided *metapredicates*. A query is described by a template (loosely speaking) if (1) each predicate in the query matches one predicate in the template, and vice versa, and (2) any metapredicates on the placeholders of the template evaluate to `true` for the matching constants in the query. The order of the predicates in query and template need not be the same, and different variable names are of course possible.

For example, consider a "lookup" facility that provides information – such as name, department, office address, and so on – about the employees of a company. The "lookup" facility can either retrieve all employees, or retrieve employees whose last name has a specific prefix, or retrieve employees whose last name and first name have specific prefixes.[3] We integrate "lookup" into our heterogeneous system by creating a wrapper, called `lookup`, that exports a predicate `emp(First-Name, Last-Name, Department, Office, Manager)`. (The `Manager` field may be `'Y'` or `'N'`.) The wrapper also exports a predicate `prefix(Full, Prefix)` that is successful when its second argument is a prefix of its first argument. This second argument must be a string, consisting of letters only. We may write the following Datalog query to retrieve `emp` tuples for persons whose first name starts with `'Rak'` and whose last name starts with `'Aggr'`:

```
(Q1) answer(FN,LN,D,O,M) :- emp(FN,LN,D,O,M),
       prefix(FN,'Rak'), prefix(LN,'Aggr')
```

In this paper we use Datalog [23] as our query language because it is well-suited to handling SPJ queries and facilitates the discussion of our algorithms.[4] We use the following Datalog terms in this paper: *Distinguished variables* are the variables that appear in the target query head. A *join variable* is any variable that appears twice or more in the target query tail. In the query (Q1) the distinguished variables are `FN`, `LN`, `D`, `O` and `M` and the join variables are `FN` and `LN`.

Description (D2) is an RQDL specification of `lookup`'s query capabilities. The identifiers starting with `$` (`$FP` and `$LP`) are constant placeholders. `_isalpha()` is a metapredicate that returns `true` if its argument is a string that contains letters only. Metapredicates start with an underscore and a lowercase letter. Intuitively, template (QT2.3) describes query (Q1) because the predicates of the query match those of the template (despite differences in order and in variable names), and the metapredicates evaluate to `true` when `$FP` is mapped to `'Rak'` and `$LP` to `'Aggr'`.

```
(D2) answer(F,L,D,O,M) :-                          (QT2.1)
       emp(F,L,D,O,M)
     answer(F,L,D,O,M) :-                          (QT2.2)
       emp(F,L,D,O,M),
       prefix(L, $LP), _isalpha($LP)
     answer(F,L,D,O,M) :-                          (QT2.3)
       emp(F,L,D,O,M),
       prefix(L, $LP), prefix(F,$FP),
       _isalpha($LP), _isalpha($FP)
```

In general, a template describes any query that can be produced by the following steps:

1. *Map* each placeholder to a constant, e.g., map `$LP` to `'Aggr'`.
2. *Map* each template variable to a query variable, e.g., map `F` to `FN`.
3. *Evaluate* the metapredicates and discard any template that contains at least one metapredicate that evaluates to `false`.
4. *Permute* the template's subgoals.

2.2. Descriptions of Large and Infinite Sets of Supported Queries

RQDL can describe arbitrarily large sets of templates (and hence queries) when extended with nonterminals as in context-free grammars. Nonterminals are represented by identifiers that start with an underscore (_) and a capital letter. They have zero or more parameters and they are associated with *nonterminal templates*. A query template t containing nonterminals describes a query q if there is an *expansion* of t that describes q. An expansion of t is obtained by replacing each nonterminal N of t with one of the nonterminal templates that define N until there is no nonterminal in t.

For example, assume that `lookup` allows us to pose one or more substring conditions on one or more fields of `emp`. For example, we may pose query (Q3), which retrieves the data for employees whose office contains the strings `'alma'` and `'B'`.

```
(Q3) answer(F,L,D,O,M) :- emp(F,L,D,O,M),
       substring(O,'alma'), substring(O,'B')
```

(D4) uses the nonterminal `_Cond` to describe the supported queries. In this description the query template (QT4.1) is supported by nonterminal templates such as (NT4.1).

```
(D4)answer(F,L,D,O,M) :-                              (QT4.1)
     emp(F,L,D,O,M), _Cond(F,L,D,O,M)
    _Cond(F,L,D,O,M) :                                (NT4.1)
     substring(F, $FS), _Cond(F,L,D,O,M)
    _Cond(F,L,D,O,M) :                                (NT4.2)
     substring(L, $LS), _Cond(F,L,D,O,M)
    _Cond(F,L,D,O,M) :                                (NT4.3)
     substring(D, $DS), _Cond(F,L,D,O,M)
    _Cond(F,L,D,O,M) :                                (NT4.4)
     substring(O,$OS), _Cond(F,L,D,O,M)
    _Cond(F,L,D,O,M) :                                (NT4.5)
     substring(M, $MS), _Cond(F,L,D,O,M)
    _Cond(F,L,D,O,M) :                                (NT4.6)
```

To see that description (D4) describes query (Q3), we expand `_Cond(F,L,D,O,M)` in (QT4.1) with the nonterminal template (NT4.4) and then again expand `_Cond` with the same template. The `_Cond` subgoal in the resulting expansion is expanded by the empty template (NT4.6) to obtain expansion (E5).

(E5) `answer(F,L,D,O,M) :- emp(F,L,D,O,M),`
`substring(O,$OS), substring(O,$OS1)`

Before a template is used for expansion, all of its variables are renamed to be unique. Hence, the second occurrence of placeholder `$OS` of template (NT4.4) is renamed to `$OS1` in (E5). (E5) describes query (Q3), *i.e.*, the placeholders and variables of (E5) can be mapped to the constants and variables of (Q3).

2.3. Schema Independent Descriptions of Supported Queries

Description (D4) assumes that the wrapper exports a fixed schema. However, the query capabilities of many sources (and thus wrappers) are independent of the schemas of the data that reside in them. For example, a relational database allows SPJ queries on all of its relations. To support schema independent descriptions RQDL allows the use of placeholders in place of the relation name. Furthermore, to allow tables of arbitrary arity and column names, RQDL provides special variables called *vector variables*, or simply vectors, that match lists of variables that appear in a query. We represent vectors in our examples by identifiers starting with an underscore (_). In addition, we provide two built-in metapredicates to relate vectors and attributes: **_subset** and **_in**. **_subset**(*_R*, *_A*) succeeds if each variable in the list that matches *_R* appears in the list that matches *_A*. **_in**(`$Position, X,` *_A*) succeeds if *_A* matches a variable list, and there is a query variable that matches `X` and appears at the position number that matches `$Position`. (For readability we will use *italics* for vectors and **bold** for metapredicates).

For example, consider a wrapper called `file-wrap` that accesses tables residing in plain UNIX files. It may output any subset of any table's fields and may impose one or more substring conditions on any field. Such a wrapper may be easily implemented using the UNIX utility AWK. (D6) uses vectors and the built-in metapredicates to describe the queries supported by `file-wrap`.

```
(D6) (QT6.1) answer(_R) :- $Table(_A),
                 _Cond(_A), _subset(_R, _A)
     (NT6.1) _Cond(_A) :_in($Position,X,_A),
                 substring(X,$S), _Cond(_A)
     (NT6.2) _Cond(_A) :
```

In general, to decide whether a query is described by a template containing vectors we must expand the nonterminals, map the variables, placeholders, and vectors, and finally, evaluate any metapredicates. To illustrate this, we show how to verify that query (Q7) is described by (D6).

(Q7) `answer(L,D) :- emp(F,L,D,O,M),`
`substring(O,'alma'), substring(O,'B')`

First, we expand (QT6.1) by replacing the nonterminal `_Cond` with (NT6.1) twice, and then with (NT6.2), thus obtaining expansion (E8).

```
(E8) answer(_R) :- $Table(_A),
       _in($Position,X,_A),substring(X,$S),
       _in($Position1,X1,_A),substring(X1,$S1),
       _subset(_R,_A)
```

Expansion (E8) describes query (Q7) because there is a mapping of variables, vectors, and placeholders of (E8) that makes the metapredicates succeed and makes every predicate of the expansion identical to a predicate of the query. Namely, vector $\_A$ is mapped to `[F,L,D,O,M]`, vector $\_R$ to `[L,D]`, placeholders `$Position` and `$Position1` to 4, `$S` to `'alma'`, `$S1` to `'B'`, and the variables `X` and `X1` to `O`. We must be careful with vector mappings; if the vector $\_V$ that maps to $[X_1, \ldots, X_n]$ appears in a metapredicate, we replace $\_V$ with $[X_1, \ldots, X_n]$. However, if the vector $\_V$ appears in a predicate as $p(\_V)$ the mapping results in $p(X_1, \ldots, X_n)$. Finally, the metapredicate `_in(4, O, [F,L,D,O,M])` succeeds because `O` is the fourth variable of the list, and `_subset([L,D], [F,L,D,O,M])` succeeds because `[L,D]` is a "subset" of `[F,L,D,O,M]`.

Vectors are useful even when the schema is known as the specification may otherwise be repetitious, as in description (D4). In our running example, even though we know the attributes of `emp`, we save effort by not having to explicitly mention all of the column names to say that a substring condition can be placed on any column.

2.4. Query and Description Normal Form

If we allow templates' variables and vectors to map to arbitrary lists of constants and variables, descriptions may appear to support queries that the underlying wrapper does not support. This is because using the same variable name in different places in the query or description can cause an implicit join or selection that does not explicitly appear in the description. For example, consider query (Q9), which retrieves employees where the manager field is `'Y'` and the first and last names are equal, as denoted by the double appearance of `FL` in `emp`.

```
(Q9) answer(FL,D) :- emp(FL,FL,D,O,'Y')
```

(D6) should not describe query (Q9). Nevertheless, we can construct expansion (E10), which erroneously matches query (Q9) if we map $\_A$ to `[FL,FL,D,O,'Y']` and $\_R$ to `[FL,D]`:

```
(E10) answer(_R):-$Table(_A), _subset(_R,_A)
```

This section introduces a query and description *normal form* that avoids inadvertently describing joins and selections that were not intended. In the normal form both queries and descriptions have only explicit equalities. A query is normalized by replacing every constant c with a unique variable V and then by introducing the subgoal $V = c$. Furthermore, for every join variable V that appears $n > 1$ times in the query we replace its instances with the unique variables $V_1, \ldots, V_n$ and introduce the subgoals $V_i = V_j, i = 1, \ldots, n, j = 1 \ldots, i-1$. We replace any appearance of V in the head with V_1. For example, query (Q11) is the normal form of (Q9).

```
(Q11) answer(FL1,D) :- employee(FL1,FL2,D,O,M),
      FL1=FL2, M='Y'
```

Description (D6) does not describe (Q11) because (D6) does not support the equality conditions that appear in (Q11). Description (D12) supports equality conditions on any column and equalities between any two columns: (NT12.2) describes equalities with constants and (NT12.3) describes equalities between the columns of our table.

```
(D12) answer(_R) :-                                  (QT12.1)
       $Table(_A), _Cond(_A), _subset(_R, _A)
      _Cond(_A) :                                    (NT12.1)
       _in($Position,X,_A), substring(X, $S),
       _Cond(_A)
      _Cond(_A) :                                    (NT12.2)
       _in($Position1,X,_A), X=$C, _Cond(_A)
      _Cond(_A) :                                    (NT12.3)
       _in($Pos1,X,_A), _in($Pos2,Y,_A),
       X=Y, _Cond(_A)
      _Cond(_A) :                                    (NT12.4)
```

For presentation purposes we use the more compact unnormalized form of queries and descriptions when there is no danger of introducing inadvertent selections and joins. However, the algorithms rely on the normal form.

3. The Capabilities-Based Rewriter

The Capabilities-Based Rewriter (CBR) determines whether a target query q is directly supported by the appropriate wrapper, *i.e.*, whether it matches the description d of the wrapper's capabilities. If not, the CBR determines whether q can be computed by combining a set of supported queries (using selections, projections and joins). In this case, the CBR will produce a set of plans for evaluating the query. The CBR consists of three modules, which are invoked serially (see Figure 2):

- **Component SubQuery (CSQ) Discovery:** finds supported queries that involve one or more subgoals of q. The CSQs that are returned contain the largest possible number of selections and joins, and do no projection. All other CSQs are pruned. This prevents an exponential explosion in the number of CSQs.

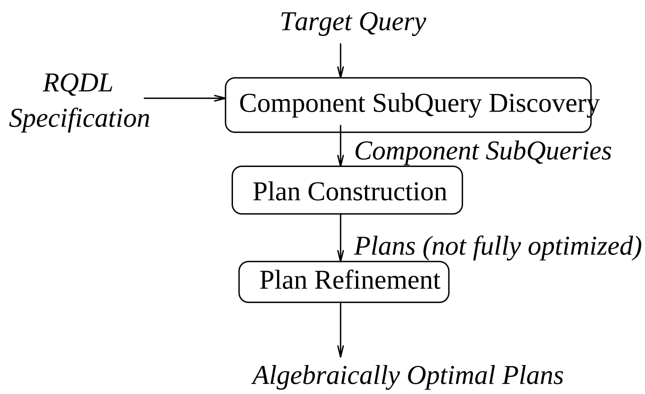

Figure 2. The CBR's components

- **Plan Construction:** produces one or more plans that compute q by combining the CSQs exported by CSQ Discovery. The plan construction algorithm is based on query subsumption and has been tuned to perform efficiently in the cases typically arising in capabilities-based rewriting.
- **Plan Refinement:** refines the plans constructed by the previous phase by pushing as many projections as possible to the wrapper.

Example 0: Consider query (Q13), which retrieves the names of all managers that manage departments that have employees with offices in the `'B'` wing, and the employees' office numbers. This query is not directly supported by the wrapper described in (D12).

```
(Q13) answer(F0,L0,O1):-emp(F0,L0,D,O0,'Y'),
       emp(F1,L1,D,O1,M1), substring(O1,'B')
```

The CSQ detection module identifies and outputs the following CSQs:

```
(Q14) answer14(F0,L0,D,O0) :-
      emp(F0,L0,D,O0,'Y')
(Q15) answer15(F1,L1,D,O1,M1) :-
      emp(F1,L1,D,O1,M1), substring(O1, 'B')
```

Note, the CSQ discovery module does not output the 2^4 CSQs that have the tail of (Q14) but export a different subset of the variables `F0`, `L0`, `D`, and `O0` (likewise for (Q15). The CSQs that export fewer variables are pruned.

The plan construction module detects that a join on `D` of $\texttt{answer}_{14}$ and $\texttt{answer}_{15}$ produces the required `answer` of (Q13). Consequently, it derives the plan (P16).

```
(P16) answer(F0,L0,O1) :-
        answer14(F0,L0,D,O0),
        answer15(F1,L1,D,O1,M1)
```

Finally, the plan refinement module detects that variables `O0`, `F1`, `L1`, and `M1` in $\texttt{answer}_{14}$ and $\texttt{answer}_{15}$ are unnecessary. Consequently, it generates the more efficient plan (P19).

```
(Q17) answer17(F0,L0,D) :-
        emp(F0,L0,D,O0,'Y')
(Q18) answer18(D,O1) :-
        emp(F1,L1,D,O1,M1), substring(O1, 'B')
(P19) answer(F0,L0,O1) :-
        answer17(F0,L0,D),  answer18(D,O1)
```

□

The CBR's goal is to produce all *algebraically optimal* plans for evaluating the query. An algebraically optimal plan is one in which any selection, projection or join that can be done in the wrapper is done there, and in which there are no unnecessary queries. More formally:

Definition. Algebraically Optimal Plan P A plan P is algebraically optimal if there is no other plan P' such that for every CSQ s of P there is a corresponding CSQ s' of P' such that the set of subgoals of s' is a superset of the set of subgoals of s (*i.e.*, s' has more selections and joins than s) and the set of exported variables of s is a superset of the set of exported variables of s' (*i.e.*, s' has more projections than s.)

In the next three sections we describe each of the modules of the CBR in turn.

4. CSQ Discovery

The CSQ discovery module takes as input a target query and a description. It operates as a rule production system where the templates of the description are the production rules and the subgoals of the target query are the base facts. The CSQ discovery module uses bottom-up evaluation because it is guaranteed to terminate even for recursive descriptions [24]. However, bottom-up derivation often derives unnecessary facts, unlike top-down. We use a variant of *magic sets rewriting* [24] to "focus" the bottom-up derivation. To further reduce the set of derived CSQs we develop two CSQ pruning techniques as described in Sections 4.2 and 4.3. Reducing the number of derived CSQs makes the CSQ discovery more efficient and also reduces the size of the input to the plan construction module.

The query templates derive `answer` facts that correspond to CSQs. In particular, a derived `answer` fact is the head of a produced CSQ whereas the *underlying* base facts, *i.e.*, the facts that were used for deriving `answer`, are the subgoals of the CSQ. Nonterminal templates derive intermediate facts that may be used by other query or nonterminal templates. We keep track of the sets of facts underlying derived facts for pruning CSQs. The following example illustrates the bottom-up derivation of CSQs and the gains that we realize from the use of the magic-sets rewriting. The next subsection discusses issues pertaining to the derivation of facts containing vectors.

Example 1: Consider query (Q3) and description (D4) from page 6. The subgoals `emp(F,L,D,O,M)`, `substring(O, 'alma')`, and `substring(O,'B')` are treated by the CSQ discovery module as base facts. To distinguish the variables in target query subgoals from the templates' variables we "freeze" the variables, e.g. `F,L,D,O`, into similarly named constants, e.g. `f,l,d,o`. Actual constants like `'B'` are in single quotes.

In the first round of derivations template (NT4.6) derives fact `_Cond(F,L,D,O,M)` without using any base fact (since the template has an empty body). Hence, the set of facts underlying the derived fact is empty. Variables are allowed in derived facts for nonterminals. The semantics is that the derived fact holds for any assignment of frozen constants to variables of the derived fact.

In the second round many templates can fire. For example, (NT4.4) derives the fact `_Cond(F,L,D,o,M)` using `_Cond(F,L,D,O,M)` and `substring(o,'alma')`, or using `_Cond(F,L,D,o,M)` and `substring(o,'B')`. Thus, we generate two facts that, though identical, they have different underlying sets and hence we must retain both since they may generate different CSQs. In the second round we may also fire (NT4.6) again and produce `_Cond(F,L,D,O,M)` but we do not retain it since its set of underlying facts is equal to the version of `_Cond(F,L,D,O,M)` that we have already produced.

Eventually, we generate `answer(f,l,d,o,m)` with set of underlying facts {`emp(f,l,d,o,m), substring(o, 'alma'), substring(o,'B')`}. Hence we output the CSQ (Q3), which, incidentally, is the target query.

The above process can produce an exponential number of facts. For example, we could have proved `_Cond(o,L,D,O,M)`, `_Cond(F,o,D,O,M)`, `_Cond(o,o,D,O,M)`, and so on. In general, assuming that `emp` has n columns and we apply m substrings on it we may derive n^m facts. Magic-sets can remove this source of exponentiality by "focusing" the nonterminals. Applying magic-sets rewriting and the simplifications described in Chapter 13.4 of [24] we obtain the following equivalent description. We show only the rewriting of templates (NT4.4) and (NT4.6). The others are rewritten similarly.

```
(D20) answer(F,L,D,O,M) :-                                  (QT20.1)
        emp(F,L,D,O,M), _Cond(F,L,D,O,M)
      _Cond(F,L,D,Office,M) :                               (NT20.4)
        mg_Cond(F,L,D,Office,M),
        substring(Office, $OS),
        _Cond(F,L,D,Office,M)
      _Cond(F,L,D,O,M) :                                    (NT20.6)
        mg_Cond(F,L,D,O,M)
      mg_Cond(F,L,D,O,M) :                                  (MS20.1)
        emp(F,L,D,O,M)
```

Now, only `_Cond(f,l,d,o,m)` facts (with different underlying sets) are produced. Note, the magic-sets rewritten program uses the available information in a way similar to a top-down strategy and thus derives only relevant facts. □

4.1. Derivations Involving Vectors

When the head of a nonterminal template contains a vector variable it may be possible that a derivation using this nonterminal may not be able either to bind the vector to a specific list of frozen variables or to allow the variable as is in the derived fact. The CSQ discovery module can not handle this situation. For most descriptions, magic-sets rewriting solves the problem. We demonstrate how and we formally define the set of non-problematic descriptions.

For example, let us fire template (NT6.1) of (D6) on the base facts produced by query (Q3). Assume also that (NT6.2) already derived `_Cond(_A)`. Then we derive that `_Cond(_A)` holds, with set of underlying facts {`substring(o, 'alma')`}, provided that the constraint "_A contains `o`" holds. The constraint should follow the fact until _A binds to some list of frozen variables. We avoid the mess of constraints using the following magic-sets rewriting of (D6).

```
(D21) answer(_R) :-                                  (QT21.1)
        $Table(_A), _Cond(_A),
        _subset(_R, _A)
      _Cond(_A) :                                    (NT21.1)
        mg_Cond(_A), _in($Position,X,_A),
        substring(X,$S), _Cond(_A)
      _Cond(_A) :  mg_Cond(_A)                       (NT21.2)
      mg_Cond(_A) :  $Table(_A)                      (MS21.1)
```

When rules (NT21.1) and (NT21.2) fire the first subgoal instantiates variable _A to `[f,l,d,o,m]` and they derive only `_Cond([f,l,d,o,m])`. Thus, magic-sets caused _A to be bound to the only vector of interest, namely `[f,l,d,o,m]`. Note a program that derives facts with unbound vectors may not be problematic because no metapredicate may use the unbound vector variable. However we take a conservative approach and consider only those programs that produce facts with only bound vector variables. Magic-sets rewriting does not always ensure that derived facts have bound vectors. In the rest of this section we describe sufficient conditions for guaranteeing the derivation of facts with bound vectors only. First we provide a condition (Theorem 1) that guarantees that a program (that may be the result of magic rewriting) does not derive facts with unbound vectors. Then we describe a class of programs that after being magic rewritten satisfy the condition of Theorem 1.

THEOREM 1 *A program will always produce facts with bound vector variables if in all rules "*$\mathbf{\_H}(\_V)$: − `tail`*"* `tail` *has a non-metapredicate subgoal that refers to*

_V, or in general _V can be assigned a binding if all non-metapredicate subgoals in `tail` *are bound.*

Intuitively, after we magic-rewrite a program it will keep deriving facts with unbound vectors only if a nonterminal of the initial program derives uninstantianted vectors and in the rules that is used it does not share variables with predicates or nonterminals s that bind their arguments (otherwise, the magic predicate will force the the rules that produce uninstantianted vectors to focus on bindings of s.) For example, description (D6) does not derive uninstantianted vectors because the nonterminal `_Cond`, that may derive uninstantianted variables, shares variables with `$Table(_A)`.

Nevertheless magic sets rewriting does not ensure that derived facts have bound vectors. Below we provide a formal criterion for deciding whether after we magic rewrite a program its bottom-up evaluation will derive facts that have bound vectors only. We believe that all reasonable descriptions satisfy the described criterion. First we state a few definitions needed for the formalization of our criterion.

Definition. Target predicate A predicate that appears in some subgoal of the target query.

Definition. Target subgoal A subgoal that uses a target predicate.

Target subgoals always instantiate their arguments using frozen constants. The following definitions capture how arguments of subgoal s are instantiated by other subgoals thereby allowing magic-sets to restrict the firing of rules defining s.

Definition. Grounded Subgoal in a Rule R A target subgoal is grounded. A nonterminal subgoal is grounded as defined by Definition 4.1. A metapredicate subgoal s is grounded if s can be evaluated using the bindings of those arguments that appear in grounded subgoals of R.

Definition. Grounded Rule A rule is grounded if every vector variable in the rule appears in some grounded subgoal. The rule is said to *depend* on the predicates of the grounded subgoals.

Definition. Grounded nonterminal A nonterminal **_N** is grounded if each rule defining **_N** is grounded. For its grounding, **_N** depends on a nonterminal **_M** if some rule defining **_N** depends on **_M**.

Grounded rules derive instantiated facts and only instantiated facts are derived for grounded nonterminals. We consider only those descriptions where all nonterminals are grounded. For such descriptions magic-sets rewriting always produces production rules that can be evaluated bottom-up without deriving facts with vector variables.

THEOREM 2 *If each nonterminal in a description D is grounded then a bottom-up evaluation of magic-sets rewritten D produces no fact that has vector variables.*

Descriptions that satisfy the above condition are considered *valid*.

THEOREM 3 *Nonterminals of a valid program can be completely ordered such that nonterminal* **_N** *in position i depends for its groundings only on nonterminal in positions* $1 \ldots i-1$.

The following algorithm derives CSQs given a target query and description. Notice that the following algorithm uses the Algorithm 2 of Figure 3, which may be skipped in a first reading. Understanding fully Algorithm 2 requires reading first the passing bindings join techniques discussed in Section 5.

Algorithm 1
Input: Target query Q and Description D
Output: A set of CSQs $s_i, i = 1, \ldots, n$
Method:
  Check if the description D is valid
  Reorder each template R in D such that
    All predicate subgoals occur in the front of the rule
    A nonterminal **_N** appears after **_M** if **_N** depends on **_M** for grounding.
    Metapredicates appear at the end of the rule
  Rewrite D using Magic-sets
  Evaluate bottom-up the rewritten description D as per Algorithm 2 of Figure 3

4.2. Retaining Only "Representative" CSQs

A large number of unneeded CSQs are generated by templates that use vectors and the **_subset** metapredicate. For example, template (QT12.1) describes for a particular **_A** all CSQs that have in their head any subset of variables in **_A**. It is not necessary to generate all possible CSQs. Instead, for all CSQs that are derived from the same expansion e, of some template t, where e has the form

`answer(`$\_V$`) :-` $\langle$*predicate and metapredicate list*$\rangle$`,` **_subset**`(`$\_V$`,`$\_W$`)`

and $\_V$ does not appear in the $\langle$*predicate and metapredicate list*$\rangle$ we generate only the *representative* CSQ that is derived by mapping $\_V$ to the same variable list as $\_W$.[5] All *represented* CSQs, *i.e.*, CSQs that are derived from e by mapping $\_V$ to a proper subset of $\_W$ are not generated. For example, the representative CSQ (Q15) and the represented CSQ (Q18) both are derived from the expansion (E22) of template (QT12.1).

(E22) `answer(`$\_R$`) :- $Table(`$\_A$`),`
  _in`($Position,X,`$\_A$`), substring(X,'B'),`
  _subset`(`$\_R$`,`$\_A$`)`

The CSQ discovery module generates only (Q15) and not (Q18) because (Q15) has fewer attributes than (Q18) and is derived by by mapping the vector $\_R$ to the same vector with $\_A$, *i.e.*, to `[F1,L1,D,O1,M1]`. Representative CSQs often retain unneeded attributes and consequently *Representative plans*, *i.e.*, plans containing representative CSQs, retrieve unneeded attributes. The unneeded attributes are projected out by the plan refinement module.

THEOREM 4 *Retaining only representative CSQs does not lose any plan,* i.e., *if there is an algebraically optimal plan p_s that involves a represented query s then p_s will be discovered by the CBR.*

Proof: The proof is based on the fact that for every plan p_s there is a corresponding representative plan p_r derived by replacing all CSQs of p_s with their representatives. For simplicity, let us assume that p_s involves only one represented CSQ (that is the CSQ s.) If r is the CSQ that represents s then the plan construction module will output a plan p_r, identical to p_s modulo that it uses r instead of s. The plan refinement module potentially removes some subgoals from the set of consumed subgoals of r. The remaining set of consumed subgoals is either identical to the consumed set of s or it is smaller. If it is identical, then by replacing the necessary variables of r with the necessary variables of the set we get the query with the smaller head that consumes the same set with s. Given that s is algebraically optimal, s is the query we found by reforming r. If the reduced set of consumed subgoals of r is smaller than the set of subgoals consumed by s, then r will have fewer exported variables than s and hence s is not algebraically optimal (it has same body with r but more variables.)

Evaluation: Retaining only a representative CSQ of head arity a eliminates $2^a - 1$ represented CSQs thus eliminating an exponential factor from the execution time and from the size of the output of the CSQ discovery module. Still, one might ask why the CSQ discovery phase does not remove the variables that can be projected out. The reason is that the "projection" step is better done after plans are formed because at that time information is available about the other CSQs in the plan and the way they interact (see Section 6). Thus, though postponing projection pushes part of the complexity to a later stage, it eliminates some complexity altogether. The eliminated complexity corresponds to those represented CSQs that in the end do not participate in any plan because they retain too few variables.

4.3. Pruning Non-Maximal CSQs

Further efficiency can be gained by eliminating any CSQ Q that has fewer subgoals than some other CSQ Q' because Q checks fewer conditions than Q'. A CSQ is maximal if there is no CSQ with more subgoals and the same set of exported variables, modulo variable renaming. We formalize maximality in terms of subsumption [24]:

Definition. Maximal CSQs A CSQ s_m is a *maximal CSQ* if there is no other CSQ s that is subsumed by s_m.

Evaluation: In general, the CSQ discovery module generates only *maximal* CSQs and prunes all others. This pruning technique is particularly effective when the

Algorithm 2

Input: A set of production rules of description D.
Set of frozen facts F corresponding to the target query Q.
Output: All facts derivable from applying D to F
Method:
Initialize to $\{\}$ the set (A) of frozen constants
available in `answer` derived facts.
Initialize to $\{\}$ the set (NA) of frozen constants newly
available in `answer` derived facts.
Repeat until no new facts are derived
For each rule r in the description
Apply rule r to base facts as per Algorithm 3 of Figure 4
% Eliminate facts that use bindings not yet available
Eliminate facts s where $\mathcal{B}_s$ has a frozen constant x where $x \notin A$
% Eliminate facts that do not use at least one new binding
Eliminate facts s where $\mathcal{B}_s$ has frozen constant x where $x \in NA$
% Update the sets of available and newly available frozen constants
Add the set of frozen constants in the heads of
the new derived facts to (NA)
Remove from (NA) those frozen constants also present in (A)
Add (NA) to (A)

Figure 3. Bottom-Up Evaluation of a Description

CSQs contain a large number of conditions. For example, assume that g conditions are applied to the variables of a predicate. Consequently, there are $2^g - 1$ CSQs where each one of them contains a different proper subset of the conditions. By keeping "maximal CSQs only" we eliminate an exponential factor of 2^g from the output size of the CSQ discovery module.

THEOREM 5 *Pruning non-maximal CSQs does not lose any algebraically optimal plan.*

Proof: For every plan p_s involving a non-maximal CSQ s there is also a plan p_m that involves the corresponding maximal CSQ s_m such that p_m pushes more selections and/or joins to the wrapper than p_s, since s_m by definition involves more selections and/or joins than s.

Algorithm 3

Input: Production rule R
A set of frozen base facts
A set of derived facts s associated with annotations:
$\mathcal{C}_s$, the set of frozen facts of the initial
database that have been used for deriving s
$\mathcal{B}_s$, the set of variables needed by the subgoals
that correspond to the facts of $\mathcal{C}_s$

Output: Derived facts + annotations obtained by firing R
using frozen and derived base facts.

Method:
% Each alternate unification may yield many facts.
Unify each subgoal in the body of R with a base fact deriving fact n
For each **_equal** subgoals s
if s equates a frozen variable x to itself then s can be ignored
if s equates two different frozen variables
then the whole unification fails
if s equates a frozen constant c and a place holder
then add c to annotation $\mathcal{B}_n$
For each **_subset** subgoal $s =$ **_subset**`(Sub,Super)`
if `Sub` and `Super` are different vector variables,
then unification fails
if `Sub` and `Super` are instantiated vectors
and `Sub` is not a subset of `Super` then fail.
if only `Super` is instantiated then equate `Sub` to the same vector.
In all other cases unification fails
For each **_in** subgoal $s =$ **_in**`(Pos,Ele,Vector)`
if `Pos,Ele,Vector` or `Ele,Vector` are instantiated
evaluate subgoal to true/false
if `Pos,Vector` are instantiated
then assign `Ele` the appropriate value
if `Vector` is instantiated
then assign `Pos, Ele` all possible values
In all other cases unification fails
For each non-meta subgoal s
Add $\mathcal{C}_s$ to $\mathcal{C}_n$; Add $\mathcal{B}_s$ to $\mathcal{B}_n$
% Eliminate non-maximal facts
If derived fact n has smaller annotation $\mathcal{C}_n$, larger $\mathcal{B}_n$,
and same set of exported variables with some existing fact n'
then do not add n to set of derived facts.

Figure 4. Algorithm for the Evaluation of a Single Rule

5. Plan Construction

In this section we present the plan construction module (see Figure 2.) In order to generate a (representative) plan we have to select a subset S of the CSQs that provides all the information needed by the target query, *i.e.*, (i) the CSQs in S check all the subgoals of the target query, (ii) the results in S can be joined correctly, and (iii) each CSQ in S receives the constants necessary for its evaluation. Section 5.1 addresses (i) with the notion of "subgoal consumption." Section 5.2 checks (ii), *i.e.*, checks join variables. Section 5.3 checks (iii) by ensuring bindings are available. Finally, Section 5.4 summarizes the conditions required for constructing a plan and provides an efficient plan construction algorithm.

5.1. Set of Consumed Subgoals

We associate with each CSQ a set of consumed subgoals that describes the CSQs contribution to a plan. Loosely speaking, a component query consumes a subgoal if it extracts all the required information from that subgoal. A CSQ does not necessarily consume all its subgoals. For example, consider a CSQ s_e that semijoins the `emp` relation with the `dept` relation to output each `emp` tuple that is in some department in relation `dept`. Even though this CSQ has a subgoal that refers to the `dept` relation it may not always consume the `dept` subgoal. In particular, consider a target query Q that requires the names of all employees and the location of their departments. CSQ s_e does not output the location attribute of table `dept` and thus does not consume the `dept` subgoal with respect to query Q. We formalize the above intuition by the following definition:

Definition. Set of Consumed Subgoals for a CSQ A set $\mathcal{S}_s$ of subgoals of a CSQ s constitutes a *set of consumed subgoals* of s if and only if

1. s exports every distinguished variable of the target query that appears in $\mathcal{S}_s$, and
2. s exports every join variable that appears in $\mathcal{S}_s$ and also appears in a subgoal of the target query that is not in $\mathcal{S}_s$.

THEOREM 6 *Each CSQ s has a unique* maximal *set $\mathcal{C}_s$ of consumed subgoals that is a superset of every other set of consumed subgoals.*

Proof: s has at least one set of consumed subgoals (trivially, the empty set is a set of consumed subgoals.) and hence it has at least one maximal set of consumed subgoals. Let us assume that there are two maximal sets $\mathcal{C}_s^1$ and $\mathcal{C}_s^2$. Then, their union is also a consumed set since it satisfies both conditions of definition 5.1. Hence, $\mathcal{C}_s^1$ and $\mathcal{C}_s^2$ can not simultaneously be maximal consumed sets.

Intuitively the maximal set describes the "largest" contribution that a CSQ may have in a plan. The following algorithm states how to compute the set of maximal consumed subgoals of a CSQ. We annotate every CSQ s with its set of maximal consumed subgoals, $\mathcal{C}_s$.

Algorithm 4

Input: CSQ s and target query Q
Output: CSQ s with computed annotation $\mathcal{C}_s$
Method:
  Insert in $\mathcal{C}_s$ all subgoals of s
  Remove from $\mathcal{C}_s$ subgoals that have a
  distinguished attribute of Q not exported by s
  Repeat until size of $\mathcal{C}_s$ is unchanged
    Remove from $\mathcal{C}_s$ subgoals that:
      Join on variable V with subgoal g
      of Q where g is not in $\mathcal{C}_s$, and
      Join variable V is not exported by s
  Discard CSQ s if $\mathcal{C}_s$ is empty.

This algorithm is polynomial in the number of the subgoals and variables of the CSQ. Also, the algorithm discards all CSQs that are not *relevant* to the target query:

Definition. Relevant CSQ A CSQ s is called *relevant* if $\mathcal{C}_s$ is non-empty.

Intuitively, irrelevant CSQs are pruned out because in most cases they do not contribute to a plan, since they do not consume any subgoal. Note, we decide the relevance of a CSQ "locally," *i.e.*, without considering other CSQs that it may have to join with. By pruning non-relevant CSQs we can build an efficient plan construction algorithm that in most cases (Section 5.2) produces each plan in time polynomial in the number of CSQs produced by the CSQ discovery module. However, there are scenarios where the relevance criteria may erroneously prune out a CSQ that could be part of a plan. The following example provides such a scenario.

Example 2: Consider a variation of the wrapper `file-wrap` (D6) where the supported queries that accept `substring` conditions may output only the first and last name, e.g., (Q24). There is also a supported query – (Q25) – that returns the whole `emp` table. The target query (Q23), that requests the full information of employees in the "database" department can be answered by the plan (P26).

(Q23) `answer(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,'database')`
(Q24) `answer`$_{24}$`(F,L) :- emp(F,L,D,O,M), substring(O,'database ')`
(Q25) `answer`$_{25}$`(F,L,D,O,M) :- emp(F,L,D,O,M)`
(P26) `answer (F,L,D,O,M) :- answer`$_{24}$`(F,L), answer`$_{25}$`(F,L,D,O,M)`

CBR will not find the plan (P26) because the CSQ (Q25) does not consume any subgoal because it does not export the distinguished variables `D`, `O`, and `M`. Intuitively, (Q25) does not contribute subgoals but it contributes variables. □

We may avoid the loss of such plans by not pruning irrelevant CSQs and thus sacrificing the polynomiality of the plan construction algorithm. In this paper we will not consider this option.

5.2. Join Variables Condition

It is not always the case that if the union of consumed subgoals of some CSQs is equal to the set of the target query's subgoals then the CSQs together form a plan. In particular, it is possible that the join of the CSQs may not constitute a plan. For example, consider an online employee database that can be queries for the names of all employees in a given division. The database can also be queried for the names of all employees in a given location. Further, the name of an employee is not uniquely determined by their location and division. The employee database cannot be used to find employees in a given division and in a given location by joining the results of two queries - one on division and the other on location. To see this, consider a query that looks for employees in "CS" in "New York". Joining the results of two independent queries on division and location will incorrectly return as answer a person named "John Smith" if there is a "John Smith" in "CS" in "San Jose" and a different "John Smith" in "Electrical" in "New York".

Intuitively, the problem arises because the two independent queries do not export the information necessary to correctly join their results. We can avoid this problem by checking that CSQs are combined only if they export the join variables necessary for their correct combination. The theorem of Section 5.4 formally describes the conditions on join variables that guarantee the correct combination of CSQs.

5.3. Passing Required Bindings via Nested Loops Joins

The CBR's plans may emulate joins that could not be pushed to the wrapper, with nested loops joins where one CSQ passes join variable bindings to the other. For example, we may compute (Q13) by the following steps: first we execute (Q27); then we collect the department names (*i.e.*, the `D` bindings) and for each binding d of `D`, we replace the `$D` in (Q28) with d and send the instantianted query to the wrapper. We use the notation `/$D` in the nested loops plan (P29) to denote that (Q28) receives values for the `$D` placeholder from `D` *bindings* of the other CSQs – (Q27) in this example.

(Q27) `answer`$_{27}$`(F0,L0,D,O0):-emp(F0,L0,D,O0,'Y')`
(Q28) `answer`$_{28}$`(F1,L1,O1,M1):-emp(F1,L1,$D,O1,M1)`
(P29) `answer(F0,L0,O1):-answer`$_{27}$`(F0,L0,D,O0),answer`$_{28}$`(F1,L1,O1,M1)/$D`

The introduction of nested loops and *binding passing* poses the following requirements on the CSQ discovery:

- **CSQ discovery:** A subgoal of a CSQ s may contain placeholders `/$`$\langle var \rangle$, such as `$D`, in place of corresponding join variables (`D` in our example.) Whenever this is the case, we introduce the structure /$\$\langle var \rangle$ next to the $\texttt{answer}_s$ that appears in the plan. All the variables of s that appear in such a structure are included in the set $\mathcal{B}_s$, called the *set of bindings needed by s*. For example, $\mathcal{B}_{28} = \{\texttt{D}\}$ and $\mathcal{B}_{27} = \{\}$. CSQ discovery previously did not use bindings information while deriving facts. Thus, the algorithm derives useless CSQs that need bindings not exported by any other CSQ.

The optimized derivation process uses two sets of attributes and proceeds iteratively. Each iteration derives only those facts that use bindings provided by existing facts. In addition, a fact is derived if it uses at least one binding that was made available only in the very last iteration. Thus, the first iteration derives facts that need no bindings, that is, for which $\mathcal{B}_s$ is empty. The next iteration derives facts that use at least one binding provided by facts derived in iteration one. Thus, the second iteration does not derive any subgoal derived in the first iteration, and so on. The complete algorithm of Figure 3 formalizes this intuition.

The bindings needed by each CSQ of a plan impose order constraints on the plan. For example, the existence of D in $\mathcal{B}_{28}$ requires that a CSQ that exports D is executed before (Q28). It is the responsibility of the plan construction module to ensure that the produced plans satisfy the order constraints.

Evaluation The pruning of CSQs with inappropriate bindings prunes an exponential number of CSQs in the following common scenario: Assume we can put an equality condition on any variable of a subgoal p. Consider a CSQ s that contains p and assume that n variables of p appear in subgoals of the target query that are not contained in s. Then we have to generate all 2^n versions of s that describe different binding patterns. Assuming that no CSQ may provide any of the n variables it is only one (out the 2^n) CSQs useful.

5.4. A Plan Construction Algorithm

In this section we summarize the conditions that are sufficient for construction of a plan. Then, we present an efficient algorithm that finds plans that satisfy the theorem's conditions. Finally, we evaluate the algorithm's performance.

THEOREM 7 *Given CSQs $s_i, i = 1, \ldots, n$ with corresponding heads $\texttt{answer}_i(V_1^i, \ldots, V_{v_i}^i)$, sets of maximal consumed subgoals $\mathcal{C}_i$ and sets of needed bindings $\mathcal{B}_i$, the plan*

$$\texttt{answer}(V_1, \ldots, V_m) \texttt{:-} \texttt{answer}_1(V_1^1, \ldots, V_{v_1}^1), \ldots, \texttt{answer}_n(V_1^n, \ldots, V_{v_n}^n)$$

is correct if

- **consumed sets condition:** *The union of maximal consumed sets $\cup_{i=1,\ldots,n} \mathcal{C}_i$ is equal to the target query's subgoal set.*

- **join variables condition:** *If the set of maximal consumed subgoals of CSQ s_i has a join variable V then every CSQ s_j that contains V in its set of maximal consumed subgoals $\mathcal{C}_j$ exports V.*
- **bindings passing condition:** *If $V \in \mathcal{B}_i$ then there must be a CSQ $s_j, j < i$ that exports V.*

Proof: We will show that the plan computes the same result with the target query when they are evaluated over the same database.

Let us first consider plans that do not contain nested loops joins. We will show the equivalence of the plan and the target query by showing that there is a mapping from the plan to the target query and vice versa. Note, we consider queries and descriptions in normal form. Let us assume that the target query has the form

$$\texttt{answer}(H_1, \ldots, H_h) :- g_1(V_1^1, \ldots, V_{v_1}^1), \ldots, g_m(V_1^m, \ldots, V_{v_m}^m) \quad (1)$$

every variable of the head appears in the tail. The plan has the form

$$\texttt{answer}(H_1, \ldots, H_h) :- \texttt{answer}_1(A_1^1, \ldots, A_{a_1}^1), \ldots, \texttt{answer}_l(A_1^n, \ldots, A_{a_n}^n) \quad (2)$$

where every variable of the head appears in the tail. The ith CSQ has the form

$$\begin{array}{l} \texttt{answer}_i(A_1^i, \ldots, A_{a_i}^i) :- \\ \quad g_{\theta(i,1)}(V_1^{\theta(i,1)}, \ldots, V_{v_{\theta(i,1)}}^{\theta(i,1)}), \ldots, g_{\theta(i,m_i)}(V_{m_i}^{\theta(i,m_i)}, \ldots, V_{v_{\theta(i,m_i)}}^{\theta(i,m_i)}) \end{array} \quad (3)$$

where the function θ maps subgoals of the CSQs to subgoals of the original target query 1.

Using 3 we can rewrite 2 as follows:

$$\begin{array}{l} \texttt{answer}(H_1, \ldots, H_n) :- \\ \quad g_{\theta(1,1)}(X_1^{\theta(1,1)}, \ldots, X_{v_{\theta(1,1)}}^{\theta(1,1)}), \ldots, g_{\theta(1,m_1)}(X_{m_1}^{\theta(1,m_1)}, \ldots, X_{v_{\theta(1,m_1)}}^{\theta(1,m_1)}), \\ \quad g_{\theta(n,1)}(X_1^{\theta(n,1)}, \ldots, X_{v_{\theta(n,1)}}^{\theta(n,1)}), \ldots, g_{\theta(n,m_n)}(X_{m_n}^{\theta(n,m_n)}, \ldots, X_{v_{\theta(n,m_n)}}^{\theta(n,m_n)}) \end{array} \quad (4)$$

where the variables that appear in the tail are identical to the ones that appear in the original CSQ if they appear in the CSQ's head. Otherwise, we rename the original variables of the CSQ so that they are not identical to the variables introduced by any other CSQ. Formally,

$$X_k^{\theta(i,j)} = \{ \begin{array}{l} V_k^{\theta(i,j)}, \textit{if } V_k^{\theta(i,j)} \in \{A_1^i, \ldots, A_{a_i}^i\} \\ N_k^{\theta(i,j)}, \textit{if } V_k^{\theta(i,j)} \notin \{A_1^i, \ldots, A_{a_i}^i\} \end{array} \quad (5)$$

If $V_k^{\theta(i,j)}$ is identical to $V_{k'}^{\theta(i,j')}$ then $N_k^{\theta(i,j)}$ is identical to $N_{k'}^{\theta(i,j')}$.

Now we will show that there is a mapping from the target query 1 to the plan 4 and vice versa. We map the plan 4 to the query 1 by mapping the variables $X_k^{\theta(i,j)}$ that are equivalent to some $N_k^{\theta(i,j)}$ to $V_k^{\theta(i,j)}$. The heads are identical so we do not have to do anything to establish a mapping.

Then, we map the query 1 to the plan 4. Essentially, we map the subgoals of the target query that correspond to the consumed set of a CSQ to the subgoals of the CSQ. It is not obvious that this can happen because the sets of consumed subgoals of the CSQs share variables.

First we map the subgoals of the target query that correspond to $\mathcal{C}_1$ to the the tail of the first CSQ, then we map the subgoals of the target query that correspond to $\mathcal{C}_2 - \mathcal{C}_1$ to to subgoals of the second CSQ and so on. Let us assume, without loss of the generality, that $\mathcal{C}_i - \mathcal{C}_{i-1} - \ldots - \mathcal{C}_1$ consists of

$$g_j(V_1^j, \ldots, V_{v_j}^j), j = 1, \ldots, c_i$$

For every $g_j(V_1^j, \ldots, V_{v_j}^j)$ there is a j' such that $\theta(i, j') = j$ (because of the consumed sets condition). We map $g_j(V_1^j, \ldots, V_{v_j}^j)$ to $g_j(X_1^j, \ldots, X_{v_j}^j)$. The mapping is possible because:

- if V_k^j has appeared either in the target query head or in $\mathcal{C}_{i'}$ where $i' < i$ then because of the definition of set of consumed subgoals and because of the join variables condition it is guaranteed that V_k^j appears in the head of CSQ i, i.e., $V_k^j \in \{A_1^i, \ldots, A_{a_i}^i\}$, and it also appears in the head of CSQ i', i.e., $V_k^j \in \{A_1^{i'}, \ldots, A_{a_{i'}}^{i'}\}$. Hence, X_k^j is identical to V_k^j and V_k^j has been earlier mapped to itself.
- otherwise, we map V_k^j to X_k^j.

Plans involving nested loops can (conceptually) be reduced to corresponding plans with local joins by moving the variables of the sets $\mathcal{B}$ in the head of the CSQ if they do not appear already in the head. Then the equivalence of the plan and the target query is obvious, provided that we can execute the plan. The bindings passing condition guarantees that we can execute the plan.

The plan construction algorithm of Figure 5 is based on Theorem 7. The algorithm takes as input a set of CSQs derived by the CSQ discovery process described later, and the target query Q. At each step the algorithm selects a CSQ s that consumes at least one subgoal that has not been consumed by any CSQ s' considered so far and for which all variables of $\mathcal{B}_s$ have been exported by at least one s'. Assuming that the algorithm is given m CSQs (by the CSQ discovery module) it can construct a set that satisfies the consumed sets and the bindings passing conditions in time polynomial in m. Nevertheless, if the join variables condition does not hold the algorithm takes time exponential in m because we may have to create exponentially many sets until we find one that satisfies the join variables condition. However, the join variables condition evaluates to true for most wrappers we find in practice (see following discussion) and thus we usually construct a plan in time polynomial in m.

For every plan p there may be plans p' that are identical to p modulo a permutation of the CSQs of p. In the worst case there are $n_p!$ permutations, where n_p is the number of CSQs in p. Since it is useless to generate permutations of the same

Algorithm 5

Input: A set of CSQs $\{s_1, \ldots, s_m\}$
A target query Q
Output: A set of plans that satisfy Theorem 7
and no two plans contain exactly the same CSQs

Method: Invoke procedure *sort*($\{s_1, \ldots, s_m\}$, L_0) % *sort input in* L_0 *using* $\overset{b}{\prec}$
Invoke procedure *plan*(L_0,$\{\}$)

Procedure *plan*(L, P)
% *P is list of CSQs that form part of a plan*
% *L is a sorted list of CSQs that are considered for generating P*
% *sub(P) refers to the union of the consumed sets*
$\mathcal{C}_i$ *of the CSQs* s_i *of the set P*
If $sub(P)$ is equal to the set of subgoals of the target query Q
output plan "$\langle Q\ head\rangle$:- $\langle s_1\ head\rangle \ldots \langle s_n\ head\rangle$"
where $P = [s_1, \ldots, s_n]$
Else
Scan L from the start to the end until we find a CSQ s such that
% *s consumes at least one more subgoal*
$\mathcal{C}_s$ has at least one subgoal not in $sub(P)$
% *Bindings needed by s are available*
All variables V of $\mathcal{B}_s$ are either exported by at least one CSQ in P
or there is a predicate **_equal**(V, W)
and `W` is exported by at least one CSQ in P
If no s is found return % *no plan can be derived*
Else
% *Define for s JV(s) the set of join variables*
%*corresponding to joins not pushed down*
For each variable V of each consumed subgoal of s
If **_equal**(V, W) occurs in Q and W is in a subgoal not consumed by s
Add V to JV(s)
% *check join variables condition of Theorem 7*
For each variable V in JV(s) such that **_equal**`(V,W)` occurs in Q
Ensure W is exported by each CSQ in P
that has a consumed subgoal using W.
For each CSQ $p \in P$
For each variable V in JV(p)
such that **_equal**`(V,W)` occurs in Q and W appears in s
Ensure W is exported by s
Invoke *plan*(L',P'),
where L' is the suffix of L that follows s and $P' = concatenate(P, [s])$
Invoke *plan*(L',P) % *find all plans that do not have s*

Figure 5. The Plan Construction Algorithm

plan, The algorithm creates a total order $\prec$ of the input CSQs and generates plans by considering CSQ s_1 before CSQ s_2 only if $s_1 \prec s_2$, *i.e.*, the CSQs are considered in order by $\prec$. Note, a query s_2 must always be considered after a query s_1 if s_1 provides bindings for s_2. Hence, $\prec$ must respect the partial order $\overset{\prec}{b}$ where $s_1 \overset{\prec}{b} s_2$ if s_1 provides bindings to s_2.

The plan construction algorithm first sorts the input CSQs in a total order that respects the PO $\overset{b}{\prec}$. Then it proceeds by picking CSQs and testing the conditions of Theorem 7 until it consumes all subgoals of the target query. The algorithm capitalizes on the assumption that in most practical cases every CSQ consumes at least one subgoal and the join variables condition holds. In this case, one plan is developed in time polynomial in the number of input CSQs. The following lemma describes an important case where the join variables condition always holds.

LEMMA 1 *The join variables condition holds for any set of CSQs such that*

1. *no two CSQs of the set have intersecting sets of maximal consumed subgoals, or*

2. *if two CSQs contain the subgoal $g(V_1, \ldots, V_m)$ in their sets of maximal consumed subgoals then they both export variables $V_1, \ldots, V_m$.*

Condition (1) of Lemma 1 holds for typical wrappers of bibliographic information systems and lookup services (wrappers that have the structure of (D12)), relational databases and object oriented databases – wrapped in a relational model. In such systems it is typical that if two CSQs have common subgoals then they can be combined to form a single CSQ. Thus, we end up with a set of maximal CSQs that have non-intersecting consumed sets. Condition (2) further relaxes the condition (1). Condition (2) holds for all wrappers that can export all variables that appear in a CSQ. The two conditions of Lemma 1 cover essentially any wrapper of practical importance.

6. Plan Refinement

The plan refinement module filters and refines constructed plans in two ways. First, it eliminates plans that are not algebraically optimal. The fact that CSQs of the representative plans have the maximum number of selections and joins and that plan refinement pushes the maximum number of projections down is not enough to guarantee that the plans produced are algebraically optimal. For example, assume that CSQs s_1 and s_2 are interchangeable in all plans, and the set of subgoals of s_1 is a superset of the set of subgoals of s_2 and s_1 exports a subset of the variables exported by s_2. The plans in which s_2 participates are algebraically worse than the corresponding plans with s_1. Nevertheless, they are produced by the plan construction module because s_1 and s_2 may both be maximal, and do not represent each other because they are produced by different template expansions. Plan refinement must therefore eliminate plans that include s_2.

Algorithm 6

Input: Plan P involving representative CSQ s.
Output: One ormore plans with s replaced by a CSQ
with fewer distinguished attributes
Method:
% Prune the set of maximal consumed subgoals of s
For each subset M of the set of maximal consumed subgoals of s
Replace annotation $\mathcal{C}_s$ by M
% Check that the resulting plan is legal
% sub(P) refers to the union of the maximal consumed sets of plan P
If $sub(P)$ contains all subgoals of Q then proceed else discard M
% consumes all subgoals
Compute set of necessary variables V of s as per Definition 6.
If V is not a subset of the set of variables exported by s
discard M
Else replace the set of exported variables of s by V
to construct a new plan P'
% Check if P' is an algebraically optimal plan and discard plans
% that are algebraically worse than P'
for every discovered plan P''
if P' is algebraically worse (see Definition) than P''
discard P' and exit loop
else if P'' is algebraically worse than P'
discard P''

Figure 6. The Plan Refinement Algorithm

Plan refinement must also project out unnecessary variables from representative CSQs. Intuitively, the *necessary* variables of a representative CSQ are those variables that allow the consumed set of the CSQ to "interface" with the consumed sets of other CSQs in the plan. We formalize this notion and its significance by the following definition (note, the definition is not restricted to maximal consumed sets):

Definition. Necessary Variables of a Set of Consumed Subgoals: A variable V is a necessary variable of the consumed subgoals set $\mathcal{S}_s$ of some CSQ s if, by not exporting V, $\mathcal{S}_s$ is no longer a consumed set.

The set of necessary variables is easily computed: Given a set of consumed subgoals $\mathcal{S}$, a variable V of $\mathcal{S}$ is a necessary variable if it is a distinguished variable, or if it is a join variable that appears in at least one subgoal that is not in $\mathcal{S}$.

The complete plan refinement algorithm appears in Figure 6. Its main complication is due to the fact that unnecessary variables cannot always be projected out when the maximal consumed sets of the CSQs intersect. For example, consider a wrapper that exports predicates `emp` and `substring`. Every supported query has exactly one `emp` subgoal, at most one `substring` subgoal, and may export any subset of the `emp` variables. The target query (Q30) can be computed by plan (P33).

(Q30) `answer(F,L):-emp(F,L,D,O,M),substring(D,'data'),substring(O,'B')`
(Q31) $\texttt{answer}_{31}$`(F,L,D,O,M) :- emp(F,L,D,O,M), substring(D,'data')`
(Q32) $\texttt{answer}_{32}$`(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,'B')`
(P33) `answer(F,L) :-` $\texttt{answer}_{31}$`(F,L,D,O,M),` $\texttt{answer}_{32}$`(F,L,D,O,M)`

Having both queries export all the variables is useless. An obvious optimization is to replace (Q32) with (Q34), which exports only the distinguished variables `F` and `L` and the join variable `D`.

(Q34) $\texttt{answer}_{34}$`(F,L,D) :- emp(F,L,D,O,M), substring(O,'B')`

Indeed, variables `F`, `L` and `D` are the only *necessary* variables of the maximal consumed subgoals set {`emp(F,L,D,O,M)`, `substring(O,'B')`}.

However, reducing the exported variables of each representative query to the necessary variables of its maximal consumed set may result in an incorrect plan. For example, replacing CSQ (Q31) with CSQ (Q35) we construct the erroneous plan (P36). (P36) violates the join variables condition.

(Q35) $\texttt{answer}_{35}$`(F,L,O) :- emp(F,L,D,O,M), substring(D,'data')`
(P36) `answer(F,L) :-` $\texttt{answer}_{34}$`(F,L,D),` $\texttt{answer}_{35}$`(F,L,O)`

The problem arises because the maximal consumed sets of (Q31) and (Q32) intersect. It can be solved as follows: Since CSQ (Q34) consumes the subgoals `emp(F,L,D,O,M)` and `substring(O,'B')` we can modify the exported variables of the representative CSQ (Q31) so that it consumes only the subgoal `substring(D,'data')`. Thus, we can replace the representative CSQ (Q31) with the CSQ (Q37) that exports only the necessary variables of the set {`substring(D,'data')`}, *i.e.*, `D`. Consequently, we can construct the plan (P38).

(Q37) $\texttt{answer}_{37}$`(D) :- emp(F,L,D,O,M), substring(D,'data')`
(P38) `answer(F,L) :-` $\texttt{answer}_{34}$`(F,L,D),` $\texttt{answer}_{37}$`(D)`

Symmetrically, we may assume that (Q31) consumes `emp(F,L,D,O,M)` and `substring(D,'data')` in which case (Q32) consumes only `substring(O, 'B')` and hence we can produce the plan (P40).

(Q39) $\texttt{answer}_{39}$`(O) :- emp(F,L,D,O,M), substring(O,'B')`
(P40) `answer(F,L) :-` $\texttt{answer}_{35}$`(F,L,O),` $\texttt{answer}_{39}$`(O)`

Intuitively, the plans (P38) and (P40) correspond to two different partitions of the target query's subgoals among the sets of consumed subgoals the two representative CSQs. In general, given a representative plan, we may produce all plans that implement projections by partitioning the target query subgoals among the representative CSQs. Thus, subgoals that are in the consumed sets of more than one representative query are "assigned" to only one representative query. Then, we calculate the necessary variables for the "reduced" consumed sets of the representative queries.

For ease of explanation we describe an algorithm (see Figure 6) to add projections to a plan with one representative CSQ. The algorithm works also for plans with multiple representative CSQs.

Evaluation The Plan Refinement Algorithm is exponential in the size of $\mathcal{C}_s$. However, it can be optimized by observing the following: If some subgoal in the maximal consumed set of s is not in the maximal consumed set of any other CSQ in plan P, then this subgoal necessarily has to be present in all non-discarded subsets M. Thus, options are generated only by subgoals consumed by multiple CSQs. Thus, the algorithm becomes exponential in the size of the largest intersection of the consumed sets of the representative CSQs.

7. Combining Cost-Based Optimization with Capabilities-Based Rewriting

The previous sections described a capabilities-based rewriter that produces all algebraically optimal plans. Then, a cost-based optimizer estimates the cost of each algebraically optimal plan and selects the absolutely optimal one. However, separating cost optimization and capabilities-based rewriting may result in a huge number (exponential in the number of query subgoals and join variables) of algebraically optimal plans. For sufficiently large queries it may be prohibitively expensive to generate and evaluate all algebraically optimal plans. In this section we solve this problem by incorporating cost optimization and *pruning* into the plan construction phase. The solution follows the same techniques with the well-known System R optimizer [20] and - in accordance with System R - does not compromise in practice the completeness of the optimization (i.e., the optimal plan is discovered) but the running time may still be exponential.

The intuition behind pruning is that we do not want to keep track of all possible (sub)plans to execute and join a subset S of subgoals. We may select the most efficient subplan and use it whenever we want to join a CSQ s' with the CSQs that compute S. We first provide an example that illustrates the performance problem arising when the CBR and the optimizer operate separately. Then we describe the enhanced CBR algorithm (which includes cost optimization) and we revisit the example.

Example 3: Consider the following target query

(Q41) `answer(X, Y, Z, W) :- p(X, Y), q(Y, Z), r(Z, W)`

(Q42) $\mathtt{answer}_{42}(\mathtt{X},\mathtt{Y}) :- \mathtt{p}(\mathtt{X},\mathtt{Y})$
(Q43) $\mathtt{answer}_{43}(\mathtt{X}) :- \mathtt{p}(\mathtt{X},\$\mathtt{Y})$
(Q44) $\mathtt{answer}_{44}(\mathtt{Y}) :- \mathtt{p}(\$\mathtt{X},\mathtt{Y})$
(Q45) $\mathtt{answer}_{45}() :- \mathtt{p}(\$\mathtt{X},\$\mathtt{Y})$
(Q46) $\mathtt{answer}_{46}(\mathtt{Y},\mathtt{Z}) :- \mathtt{q}(\mathtt{Y},\mathtt{Z})$
(Q47) $\mathtt{answer}_{47}(\mathtt{Y}) :- \mathtt{q}(\mathtt{Y},\$\mathtt{Z})$
(Q48) $\mathtt{answer}_{48}(\mathtt{Z}) :- \mathtt{q}(\$\mathtt{Y},\mathtt{Z})$
(Q49) $\mathtt{answer}_{49}() :- \mathtt{q}(\$\mathtt{Y},\$\mathtt{Z})$
(Q50) $\mathtt{answer}_{50}(\mathtt{Z},\mathtt{W}) :- \mathtt{r}(\mathtt{Z},\mathtt{W})$
(Q51) $\mathtt{answer}_{51}(\mathtt{Z}) :- \mathtt{r}(\mathtt{Z},\$\mathtt{W})$
(Q52) $\mathtt{answer}_{52}(\mathtt{W}) :- \mathtt{r}(\$\mathtt{Z},\mathtt{W})$
(Q53) $\mathtt{answer}_{53}() :- \mathtt{r}(\$\mathtt{Z},\$\mathtt{W})$

Figure 7. CSQs for query (Q41)

(Q54) $\mathtt{answer}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W}) :- \mathtt{answer}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{answer}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{answer}_{50}(\mathtt{Z},\mathtt{W})$
(Q55) $\mathtt{answer}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W}) :- \mathtt{answer}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{answer}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{answer}_{52}(\mathtt{W})/\$\mathtt{Z}$
(Q56) $\mathtt{answer}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W}) :- \mathtt{answer}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{answer}_{48}(\mathtt{Z})/\$\mathtt{Y}, \mathtt{answer}_{50}(\mathtt{Z},\mathtt{W})$
(Q57) $\mathtt{answer}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W}) :- \mathtt{answer}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{answer}_{48}(\mathtt{Z})/\$\mathtt{Y}, \mathtt{answer}_{50}(\mathtt{W})/\$\mathtt{Z}$

Figure 8. Plans corresponding to the order p, q, r

Assume that the source can answer any query that refers to only one relation and has zero or more equality conditions on the relation attributes. Figure 7 lists the CSQs that will be derived from the plan construction phase. The plan construction algorithm of Figure 5 derives 24 algebraically optimal plans. In particular, for every permutation of `p`, `q`, and `r` there are four possible plans because there are two CSQs that can consume the second subgoal and two CSQs that can consume the third subgoal. Figure 8 lists the plans corresponding to the order `p`, `q`, `r`.

□

If we generalize our scenario to a "chain join" query with n predicates of the form

(Q58) $\mathtt{answer}(\mathtt{X}_0,\mathtt{X}_1 \ldots,\mathtt{X}_n,\mathtt{X}_{n+1}) :- \mathtt{p}_0(\mathtt{X}_0,\mathtt{X}_1), \mathtt{p}_1(\mathtt{X}_1,\mathtt{X}_2) \ldots \mathtt{p}_n(\mathtt{X}_n,\mathtt{X}_{n+1})$

the number of CSQs is linear in n but the number of plans is exponential in n. Indeed, even for the single permutation $\mathtt{p}_0, \mathtt{p}_1, \ldots, \mathtt{p}_n$ there are 2^n algebraically optimal plans. The plan construction and optimization Algorithm 7 (Figure 9) employs two techniques to reduce the complexity:

1. For every subset S of subgoals it discovers once the optimum subplan to compute the join of the subgoals of S. It will consequently use the optimum plan whenever the CSQs corresponding to these subgoals will be joined with other CSQs.

2. Plans that correspond to cartesian products, i.e., plans where some CSQs do not have any common variables or do not exchange bindings with the rest of the CSQs are not considered.

Algorithm 7 discovers optimal plans for increasingly larger subsets of subgoals. At the end of round i it has discovered the optimal plans for consuming each set with less than $i+1$ subgoals. It may have also generated some plans that join $i+1$ or more subgoals because upon trying to consume a set of i subgoals, say, $\{s_1, \ldots, s_i\}$ it may have to use CSQs that consume some additional subgoals such as s_{i+1}.

The following definitions clarify the notion of (sub)plan and "set of subgoals consumed by a plan".

Definition. Plan A plan p of a target query q is a sequence $\langle s_1, \ldots, s_n \rangle$ of CSQs of q such that the bindings passing condition holds, i.e., if $V \in \mathcal{B}_{s_i}$ then there is a CSQ $s_j, j < i$ that exports V.

The "sequence" definition of plan indicates only (1) which CSQs will be used in the plan and (2) what sets of bindings will be received by the CSQs that require bindings. The latter info is implied by the order in which the CSQs appear. We are not concerned with the join order and join policies for the joins that will be done by the mediator. This simplifying assumption is often justified from the predominance of network and source costs.

Definition. Consumed Set of Subgoals A plan $p = \langle s_1, \ldots, s_n \rangle$ *consumes* the set of subgoals $\mathcal{C}_p = \cup_{s_i = 1 \ldots n} \mathcal{C}_{s_i}$, i.e. $\mathcal{C}_p$ is the set of subgoals consumed by the CSQs of p.

Notice that we are not concerned with estimating the cost of our plans - though it is an important and difficult problem. Instead, we assume the existence of an appropriate cost estimation function $f(p)$.
Notation: $\langle s_1, \ldots, s_n \rangle \circ s_{n+1} = \langle s_1, \ldots, s_n, s_{n+1} \rangle$.

Example 4: Let us demonstrate Algorithm 7 in the case of the Example 4. In step $i = 1$ it generates plans consisting of one CSQ (see Figure 10). By the end of step $i = 1$ all plans for singular sets of subgoals have been constructed. However, if there were CSQs consuming more than one subgoal we would also have some sets that consume more than one subgoals.

In step $i = 2$ we pick the best plan for each of the three sets of size one and we "extend" it every time with another CSQ. We use the notation $\langle s_1, \ldots, s_n \rangle \circ s = \langle s_1, \ldots, s_n, s \rangle$ to extend a plan with one more CSQ. Notice that we avoid cartesian

Algorithm 7

INPUT: (1) a target query q and a set $\mathcal{S}$ of CSQs of q
(2) a cost function f that estimates the cost of plans
OUTPUT: A plan that computes q (if there is one) and has the least cost
METHOD:
For every CSQ s where $\mathcal{B}_s$ is empty
  insert into $\mathcal{P}$ the plan $\langle s \rangle$
For $i = 2, \ldots, n$ where n is the number of subgoals in q
  For every plan $p = \langle s_1, \ldots, s_m \rangle$ where $\mathcal{C}_p$ has less than i subgoals
    For every CSQ s
      If s consumes at least one subgoal that is not in $\mathcal{C}_p$, and
        for all $j = 1, \ldots, m : s_j \overset{\prec}{b} s$, and
        s exportsat least one variable V
            that is also exported by a CSQ of p, or
            $\mathcal{B}_s$ has a variable V that is exported by a CSQ of p
        Then create a plan $p' = p \circ s = \langle s_1, \ldots, s_m, s \rangle$
      If there is no plan p'' with $\mathcal{C}_{p''} = \mathcal{C}_{p'}$
        insert p' in $\mathcal{P}$
      Else if there is a plan p'' with $\mathcal{C}_{p''} = \mathcal{C}_{p'}$ and $f(p') < f(p'')$
        delete p'' from $\mathcal{P}$
        insert p' into $\mathcal{P}$
Output the unique plan p (if there is one) where $\mathcal{C}_p$ includes all subgoals of q

Figure 9. Plan Construction Algorithm enhanced with Cost Optimization and Pruning

products and hence we do not have any plan for the set $\{\mathtt{p}, \mathtt{r}\}$. Technically, we avoid cartesian products by requiring that the CSQ that will extend a plan shares at least one variable with the plan or takes at least one set of bindings from the plan. Furthermore, we do not generate plans which are mere permutations of each other. For example, we do not extend the optimal plan for consuming q with the CSQ (Q42). Technically, we avoid permutations by producing only sequences that conform to the partial order $\overset{\prec}{b}$ (see Section 5.4).

In step $i = 3$ we pick the optimum plan for each of the two subsets and appropriately extend it. For the sake of the example, let us assume that

$$\langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle$$

is the optimum plan for $\{p, q\}$ and

$$\langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}), \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z}\rangle$$

is the optimal plan for $\{q, r\}$. The optimal plan for $\{p, q\}$ can be extended in two ways; either with $\mathtt{ans}_{50}(\mathtt{Z},\mathtt{W})$ or with $\mathtt{ans}_{52}(\mathtt{W})/\$\mathtt{Z}$. The optimal plan for for $\{q, r\}$ can be extended with $\mathtt{ans}_{43}(\mathtt{X})/\$\mathtt{Y}$ or with $\mathtt{ans}_{42}(\mathtt{X},\mathtt{Y})$.

□

8. Practical Issues in the Implementation of a Capabilities-Based Rewriter

A capabilities-based rewriter has been implemented for Garlic using Starburst's extensible optimizer [10] as implemented for DB2 [5]. The implementation has enhanced some aspects of capabilities-based rewriting (see the list below) and has simplified algorithms whenever the corresponding functionality loss does not impede the inclusion of target sources. A detailed description of the implementation can be found in [7]. In this section we summarize the most important aspects of the implementation and we compare with the algorithms described in previous section.

- **Capabilities Description Language** In Garlic's implementation of the CBR the capabilities of a wrapper are described via a description of the set of *plans* that can be executed by the wrapper. At a sufficient level of abstraction the plans are trees where the leaves are the source relations and the inner nodes are operators, called *Plan OPerators (POPs)*, such as selection, join, projection, and so on. The role of nonterminals is assumed by the optimizer's *STARs (STrategy Alternative Rules)* which are essentially the production rules of a grammar that generates a possibly infinite number of plans. Describing capabilities using plans – as opposed to queries – facilitated the use of the Starburst extensible optimizer. However, the use of plans for describing capabilities forces a wrapper writer to understand the meaning and use of plans by the optimizer, complicating the task of writing a description. We plan to work on using a variant of RQDL for the capabilities description.

$i = 1$
$\{p\}$
$\langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y})\rangle$
$\{q\}$
$\langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle$
$\{r\}$
$\langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W})\rangle$

$i = 2$
$\{p, q\}$
$\langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y})\rangle \circ \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z}) = \langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle$ $\langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y})\rangle \circ \mathtt{ans}_{48}(\mathtt{Z})/\$\mathtt{Y} = \langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{48}(\mathtt{Z})/\$\mathtt{Y}\rangle$ $\langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle \circ \mathtt{ans}_{43}(\mathtt{X})/\$\mathtt{Y} = \langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{ans}_{43}(\mathtt{X})/\$\mathtt{Y}\rangle$
$\{q, r\}$
$\langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle \circ \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}) = \langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W})\rangle$ $\langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle \circ \mathtt{ans}_{52}(\mathtt{W})/\$\mathtt{Z} = \langle \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{ans}_{52}(\mathtt{W})/\$\mathtt{Z}\rangle$ $\langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W})\rangle \circ \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z} = \langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}), \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z}\rangle$

$i = 3$
$\{p, q, r\}$
$\langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle \circ \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}) = \langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W})\rangle$ $\langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z})\rangle \circ \mathtt{ans}_{52}(\mathtt{W})/\$\mathtt{Z} = \langle \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}), \mathtt{ans}_{46}(\mathtt{Y},\mathtt{Z}), \mathtt{ans}_{52}(\mathtt{W})/\$\mathtt{Z}\rangle$ $\langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}), \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z}\rangle \circ \mathtt{ans}_{43}(\mathtt{X})/\$\mathtt{Y} = \langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}), \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z}, \mathtt{ans}_{43}(\mathtt{X})/\$\mathtt{Y}\rangle$ $\langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}), \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z}\rangle \circ \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y}) = \langle \mathtt{ans}_{50}(\mathtt{Z},\mathtt{W}), \mathtt{ans}_{47}(\mathtt{Y})/\$\mathtt{Z}, \mathtt{ans}_{42}(\mathtt{X},\mathtt{Y})\rangle$

Figure 10. Plans constructed and evaluated for the query (Q41)

- **Mediator Capabilities Description** A unique feature of the implementation is that the capabilities of the mediator are also described – as opposed to the algorithms of the previous sections where the mediator can only do selections, projections, and joins. Having an open set of mediator capabilities is important for a system that targets not only conjunctive queries but also aggregates, similarity queries on multimedia data, etc. For example, the implementation introduced a special kind of join that implements the merging of "fuzzy" result sets using an algorithm outlined in [3]. However, we will not further discuss the non-SPJ abilities of the implemented version because the fundamental issues pertaining to capabilities-based rewriting of non-SPJ queries are not yet fully understood. For example, we do not know yet how to characterize the completeness of an algorithm for rewriting aggregates or negations.[6] For the comparison between the CBR algorithms of this paper and the Garlic implementation we assume that we deal with SPJ queries only and that the mediator capabilities include arbitrarily complex plans consisting of selections, projections, and joins.

- **Capabilities-Based Rewriter Operates with Descriptions of Multiple Sources** The target query refers to multiple sources and the description provides all the supported queries of all the sources. It is straightforward to see that the algorithms described in the previous sections are not affected by this feature modulo that the CBR has to keep track of the source that supports a given CSQ in order to issue the right query to the right source.

- **Implementation Architecture** The optimizer operates in three phases. Together, phase 1 and phase 2 do the work of CSQ discovery and plan construction, looking at first single subgoals, and then increasingly larger subsets of subgoals, using rules provided by both mediator and wrappers at each step. We assume that the mediator can retrieve all the variables associated with the tables participating in a plan supported by the wrappers; this assumption reduces plan construction to a search for plans that use all tables appearing in the target query and furthermore it eliminates the need for inserting projections during plan refinement. Finally there is a phase of (essentially minor) refinements and fixes. For example, the attributes are placed in the order requested by the query.

  The implementation combines capabilities-based rewriting with cost-based optimization in order to avoid generating all possible algebraically optimal plans (recall, there can be an exponential number of algebraically optimal plans.) In particular, the plan construction phase employs the dynamic programming algorithm of Starburst's optimizer – similar to Algorithm 7 of Figure 9 – for discovering the most efficient supported plan while at the same time it prunes the space of plans. A side effect of the cost-based optimization enhancement is that the prunings for algebraic optimality are not that crucial anymore because they become a special case of the cost-based prunings.

- **Join Variables Condition** The implementation assumes that all variables of the body of a query can be obtained. This assumption is most often valid; it posed no problem to the integration of over 10 different sources, including a relational database, two Web sources, Lotus Notes databases, a chemical structure search engine, text and image content search systems. It greatly simplifies the implementation in two ways. The first one, that we have already discussed, is that it validates the join variables condition (recall, the join variables condition requires that join variables are returned.) Second, the plan refinement step is integrated into plan construction because there is no need to construct plans with too many variables as is done by the plan construction and CSQ discovery of the previous sections.

9. Evaluation

The CBR algorithm employs many techniques to eliminate sources of exponentiality that would otherwise arise in many practical cases. The **evaluation** paragraphs of many sections in this paper describe the benefit we derive from using these techniques. Remember that our assumption that every CSQ consumes at least one subgoal led to a plan construction module that develops a plan in time polynomial to the number of CSQs produced by the CSQ detection module, provided that the join variables condition holds. This is an important result because the join variables condition holds for most wrappers in practice, as argued in Subsection 5.4.

The CBR deals only with Select-Project-Join queries and their corresponding descriptions. It produces algebraically optimal plans involving CSQs, *i.e.*, plans that push the maximum number of selections, projections and joins to the source. However, the CBR is not complete because it misses plans that contain irrelevant CSQs (see Definition 5.1 and the discussion of Section 5.1.) On the other hand, the techniques for eliminating exponentiality preserve completeness, in that we do not miss any plan through applying one of these techniques (see justifications in Sections 4.2, 4.3.)

10. Related Work

Significant results have been developed for the resolution of semantic and schematic discrepancies while integrating heterogeneous information sources. However, most of these systems [19, 8, 1, 6] do not address the problem of different and limited query capabilities in the underlying sources because they assume that those sources are full-fledged databases that can answer any query over their schema.[7] The recent interest in the integration of arbitrary information sources, including databases, file systems, the Web, and many legacy systems, invalidates the assumption that all underlying sources can answer any query over the data they export and forces us to resolve the mismatch between the query capabilities provided by these sources. Only a few systems have addressed this problem.

HERMES [19] proposes a rule language for the specification of mediators in which an explicit set of parameterized calls can be made to the sources. At run-time the parameters are instantiated by specific values and the corresponding calls are made. Thus, HERMES guarantees that all queries sent to the wrappers are supported. Unfortunately, this solution reduces the interface between wrappers and mediators to a very simple form (the particular parameterized calls), and does not fully utilize the sources' query power.

DISCO [22]describes the set of supported queries using context-free grammars. This technique reduces the efficiency of capabilities-based rewriting because it treats queries as "strings."

The Information Manifold [11] develops a query capabilities description that is attached to the schema exported by the wrapper. The description states which and how many conditions may be applied on each attribute. RQDL provides greater expressive power by being able to express schema-independent descriptions and descriptions such as "exactly one condition is allowed."

TSIMMIS suggests an explicit description of the wrapper's query capabilities [14], using the context-free grammar approach of the current paper. (The description is also used for query translation from the common query language to the language of the underlying source.) However, TSIMMIS considers a restricted form of the problem wherein descriptions consider relations of prespecified arities and the mediator can only select or project the results of a single CSQ.

This paper enhances the query capability description language of [14] to describe queries over arbitrary schemas, namely, relations with unspecified arities and names, as well as capabilities such as "selections on the first attribute of any relation." The language also allows specification of required bindings, *e.g.*, a bibliography database that returns "titles of books given author names." We provide algorithms for identifying for a target query Q the algebraically optimal CSQs from the given descriptions. Also, we provide algorithms for generating plans for Q by combining the results of these CSQs using selections, projections, and joins.

The CBR problem is related to the problem of determining how to answer a query using a set of materialized views [13, 9, 18, 17]. However, there are significant differences. These papers consider a specification language that uses SPJ expressions over given relations specifying a finite number of views. They cannot express arbitrary relations, arbitrary arities, binding requirements (with the exception of [18]), or infinitely large queries/views. Also, they do not consider generating plans that require a particular evaluation order due to binding requirements.

[9] shows that rewriting a conjunctive query is in general exponential in the total size of the query and views. [17] shows that if the query is acyclic we can rewrite it in time polynomial to the total size of the query and views. [9, 18] generate necessary and sufficient conditions for when a query can be answered by the available views. By contrast, our algorithms check only sufficient conditions and might miss a plan because of the heuristics used. Our algorithm can be viewed as a generalization of algorithms that decide the subsumption of a datalog query by a datalog program (*i.e.*, the description). [12] proposed Datalog for the description

of supported queries. It also suggested an algorithm that essentially finds what we call maximal CSQs. Recently [25] discussed the expressive power of Datalog and the expressive power of an RQDL extension. The most important result was that Datalog cannot express the capabilities of powerful sources. In particular, it is proven that there is no Datalog program that can express the set of all conjunctive queries over a given schema. It is also proven that RQDL can do so. Furthermore, the extended RQDL is reduced into Datalog with functions.

11. Conclusions and Future Work

In this paper, we presented the Relational Query Description Language, RQDL, which provides powerful features for the description of wrappers' query capabilities. RQDL allows the description of infinite sets of arbitrarily large queries over arbitrary schemas. We also introduced the Capabilities-Based Rewriter, CBR, and presented an algorithm that discovers plans for computing a wrapper's target query using only queries supported by the wrapper. Despite the inherent exponentiality of the problem, the CBR uses optimizations and heuristics to produce plans in reasonable time in most practical situations.

We also described the enhancement of CBR with a cost-based optimizer and we discussed practical issues in the implementation of a capabilities-based rewriting algorithm for Garlic.

Acknowledgments

We are grateful to Mike Carey, Hector Garcia-Molina, Anthony Tomasic, Vasilis Vassalos and Ed Wimmers for many fruitful discussions and comments.

Appendix

A.1. Syntax and Semantics of RQDL

In this section we formally present the syntax and semantics of RQDL. We focus on normal-form RQDL. (We may reduce non-normal form descriptions to normal form applying the transformations described in Section 2.4.)

The syntax appears in Figure A.1. Furthermore, we restrict to descriptions where there is a nonterminal template, with matching arity, for every nonterminal that appears in a template. Additionally, for the implementation reasons described in Section 4 we restrict to descriptions where all nonterminals are grounded.

The following definitions formally define the set of queries that is described by a description. First we define the set of expansions of a query template. Then we use the set of *terminal expansions*, *i.e.*, the set of expansions that do not contain any

$$
\begin{array}{lll}
(0)\ \langle description \rangle & ::= & (\langle query\ template \rangle | \langle nonterminal\ template \rangle)* \\
(1)\ \langle query\ template \rangle & ::= & \textbf{answer}(\ \langle predicate\ arguments \rangle\ ) \\
 & & :-\ \langle subgoal\ list \rangle \\
(2)\ \langle nonterminal\ template \rangle & ::= & \langle nonterminal\ name \rangle\ (\ \langle arguments \rangle\ ) \\
 & & \langle subgoal\ list \rangle \\
(3)\ \langle subgoal\ list \rangle & ::= & \langle subgoal \rangle\ (,\ \langle subgoal \rangle)* \\
(4)\ \langle subgoal\ list \rangle & ::= & \epsilon \quad \%subgoal\ list\ may\ be\ emtpy \\
(5)\ \langle subgoal \rangle & ::= & \langle predicate \rangle\ (\ \langle arguments \rangle\ ) \quad \%predicate \\
(6)\ \langle subgoal \rangle & ::= & \langle metapredicate\ name \rangle\ (\ \langle arguments \rangle\ ) \\
(7)\ \langle subgoal \rangle & ::= & \langle nonterminal\ name \rangle\ (\ \langle arguments \rangle\ ) \\
(8)\ \langle arguments \rangle & ::= & \langle vector \rangle | \langle variable \rangle\ (,\ \langle variable \rangle)* \\
(9)\ \langle predicate\ name \rangle & ::= & \langle identifier \rangle | \langle placeholder \rangle \\
(10)\ \langle metapredicate\ name \rangle & ::= & \langle identifier \rangle \\
(11)\ \langle nonterminal\ name \rangle & ::= & \langle identifier \rangle
\end{array}
$$

Figure A.1. Normal-form RQDL syntax

nonterminal, for defining the set of queries described by terminal expansions and hence described from the description. Note, from a syntactical viewpoint expansions are equivalent to templates.

Definition. Set of expansions $\mathcal{E}_t$ of query template t The set of expansions $\mathcal{E}_t$ contains

1. the template t
2. every expansion e derived by permuting the subgoals of an expansion $g \in \mathcal{E}_t$
3. every expansion e derived by renaming the variables, vectors, and placeholders of an expansion $g \in \mathcal{E}_t$
4. every expansion e of the form

$$\langle answer\ predicate \rangle\ :-\ \langle N\ definition\ body \rangle,\ \langle other\ subgoals \rangle$$

such that there is an expansion $g \in \mathcal{E}_t$ that has the form

$$\langle answer\ predicate \rangle\ :-\ N(\langle arguments \rangle),\ \langle other\ subgoals \rangle$$

and a nonterminal template of the form

$$N(\langle definition\ arguments \rangle)\ :\ \langle N\ definition\ body \rangle$$

where

(A) the nonterminal template and the expansion e have no common variable,

(B) there is a collection of mappings θ such that $\theta(N(\langle arguments \rangle))$ is identical to
$\theta(N(\langle definition\ arguments \rangle))$. We call θ a *unifier*. Definition A.1 formally defines the application of a unifier on an RQDL expression.

Definition. Application of unifier on RQDL expression Given the RQDL expression e, where e may be subgoal, subgoal list, or nonterminal template head, and the unifier θ, $\theta(e)$ is computed by the following steps

1. If θ contains a mapping of the form $\langle placeholder \rangle \mapsto \langle constant \rangle$, or $\langle variable \rangle_1 \mapsto \langle variable \rangle_2$, or $\langle vector \rangle_1 \mapsto \langle vector \rangle_2$ then replace all instances of $\langle placeholder \rangle$, $\langle variable \rangle_1$, and $\langle vector \rangle_2$ with $\langle constant \rangle$, $\langle variable \rangle_2$, or $\langle vector \rangle_2$ respectively.

2. If θ contains a mapping of the form $\langle vector \rangle \mapsto [\langle variable\ list \rangle]$ replace all instances of $\langle vector \rangle$ that appear in metapredicates with $[\langle variable\ list \rangle]$ and all the other instances with $\langle variable\ list \rangle$.

Definition. Set of terminal expansions $\mathcal{T}_t$ of query template t The set of terminal expansions $\mathcal{T}_t$ of a template t consists of all expansions of $\mathcal{E}_t$ that do not contain a nonterminal.

Definition. Set of queries described by query template t The set of queries described by query template t consists of all queries that are obtained by applying the following transformations to an expansion $g \in \mathcal{T}_t$

1. replace every vector with a variable list,
2. replace every placeholder with a constant,
3. remove all metapredicates that evaluate to `true`

If there is at least one metapredicate left then the transformed expansion is *not* a query.

We do not have to include all permutations of subgoals and renamings of variables in the above because $\mathcal{T}_t$ contains all expansions we can derive by subgoals permutations and variable renaming.

Notes

1. In general, there is a one-to-one mapping and no optimization is involved in this translation. All optimization is done at the mediator.
2. We see next that RQDL has nonterminals with parameters. The nonterminals of context-free grammars are a special case with 0 parameters.
3. The "lookup" facility is very similar to a Stanford University facility.
4. We could have used SPJ SQL queries instead of Datalog. Then, we would use a description language that looks like SQL and not Datalog. The same notions, *i.e.*, placeholders, nonterminals, and so on, hold. The CBR algorithm is also the same.

5. In general, the $\langle$*list of predicates and metapredicates*$\rangle$ may contain metapredicates of the form **_in**($\langle position\rangle$, $\langle variable_i\rangle$, _$V$), $i = 1, \ldots, m$. In this case, the template describes all CSQs that output a subset of _W and a superset of $\mathcal{S} = \{\langle variable\rangle_1, \ldots, \langle variable\rangle_m\}$. The CSQ discovery module outputs, as usual, the representative CSQ and annotates it with the set $\mathcal{S}$ that provides the "minimum" set of variables that represented CSQs must export. In this paper we will not describe any further the extensions needed for the handling of this case.
6. Indeed, there is not a complete algorithm for handling arbitrary SQL queries with negation. This is a consequence of the undecidability, in the general case, of the equivalence of two SQL queries with negation [21].
7. The work in query decomposition in distributed databases has also assumed that all underlying systems are relational and equally able to perform any SQL query.

References

1. R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.
2. M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. RIDE-DOM Workshop*, pages 124–31, 1995.
3. R. Fagin. Combining fuzzy information from multiple systems. In *Proc. PODS*, 1996.
4. J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.
5. P. Gassner, G. Lohman, B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. *IEEE Data Engineering Bulletin*, 16:4–18, September 1993.
6. A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.
7. L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, 1997.
8. 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.
9. A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, pages 95–104, 1995.
10. G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD*, 1988.
11. A. Levy, A. Rajaraman, and J. Ordille. Query processing in the information manifold. In *Proc. VLDB*, 1996.
12. A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external processors. In *Proc. PODS*, pages 227–37, 1996.
13. P.A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB Conf.*, pages 259–69, 1985.
14. 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.
15. Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS*, 1996.
16. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.
17. Xiaolei Qian. Query folding. In *Proc. ICDE*, pages 48–55, 1996.
18. A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS Conf.*, pages 105–112, 1995.
19. V.S. Subrahmanian et al. HERMES: A heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes/overview/paper.
20. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD*, 1979.

21. S. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27:633–55, 1980.
22. A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. Technical report, INRIA, 1995.
23. J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems.* Computer Science Press, New York, NY, 1988.
24. J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies.* Computer Science Press, New York, NY, 1989.
25. V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. Available via `http://www-cse.ucsd.edu/ yannis/`.

Received Date
Accepted Date
Final Manuscript Date