Capabilities-Based Query Rewriting in Mediator Systems*

Yannis Papakonstantinou	Ashish Gupta	Laura Haas
Univ. of California, San Diego	Junglee Corp.	IBM Almaden Research Center
Computer Science & Engineering	4149B El Camino Way	650 Harry Road
La Jolla, CA 92093-0114	Palo Alto, CA 94306	San Jose, CA 95120
yannis@cs.ucsd.edu	ashish@junglee.com	laura@almaden.ibm.com

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 report on the implementation of the CBR using the Starburst extensible optimizer. Particular attention is given to the combination of the CBR with a cost-based optimizer. The net effect is that we prune without losing completeness.[1]

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.

Figure (1.a) shows one commonly proposed integration architecture [1, 2, 3, 4]. 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 [1], 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

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

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

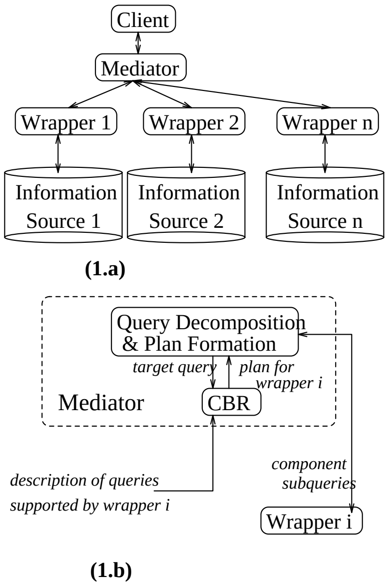

Figure 1: (a) A typical integration architecture. (b) CBR-mediator interaction.

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.[2] 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 [5] 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 [6, 7]. 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 [6, 7]. 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 [8], is the report on the implementation of the CBR using the extensible optimizer of Starburst. The use of Starburst's optimizer required changes in the description language and some compromises on the completeness of the CBR algorithm in order to simplify the

[2] In general, there is a one-to-one mapping and no optimization is involved in this translation. All optimization is done at the mediator.

implementation. On the other hand, it provided an excellent means for combining the CBR with a cost-based optimizer as we will describe.

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 summarizes the run-time performance of the CBR, while Section 8 compares the CBR with related work. Finally, Section 10 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 [9].

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, we believe that it is more expressive than context-free grammars (we are currently working on the proof).

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 [10] 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)
```

[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.

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)`

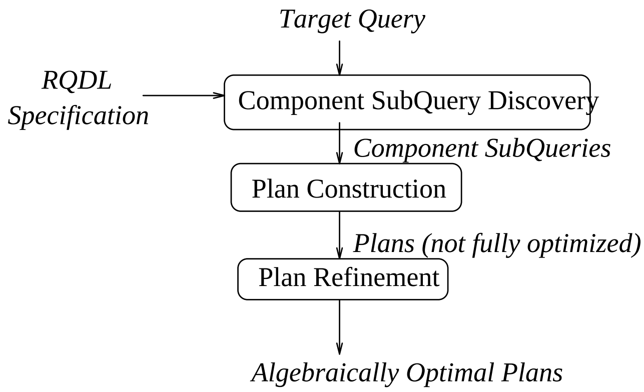

Figure 2: The CBR's components

```
_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.
- **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 3.1 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(FO,LO,O1):-emp(FO,LO,D,OO,'Y'),
       emp(F1,L1,D,O1,M1), substring(O1,'B')
```

The CSQ detection module identifies and outputs the following CSQs:

```
(Q14) answer_14(FO,LO,D,OO) :-
       emp(FO,LO,D,OO,'Y')
(Q15) answer_15(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 `FO`, `LO`, `D`, and `OO` (likewise for (Q15). The CSQs that export fewer variables are pruned.

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

```
(P16) answer(F0,L0,01) :-
        answer14(F0,L0,D,00),
        answer15(F1,L1,D,01,M1)
```

Finally, the plan refinement module detects that variables `00`, `F1`, `L1`, and `M1` in `answer`$_{14}$ and `answer`$_{15}$ are unnecessary. Consequently, it generates the more efficient plan (P19).

```
(Q17) answer17(F0,L0,D) :-
      emp(F0,L0,D,00,'Y')
(Q18) answer18(D,01) :-
      emp(F1,L1,D,01,M1), substring(01, 'B')
(P19) answer(F0,L0,01) :-
      answer17(F0,L0,D),  answer18(D,01)
```

□

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 3.1 (**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 [11]. However, bottom-up derivation often derives unnecessary facts, unlike top-down. We use a variant of *magic sets rewriting* [11] to "focus" the bottom-up derivation. To further reduce the set of derived CSQs we develop two CSQ pruning techniques as decsribed 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 4.1 Consider query (Q3) and description (D4) from page 4. 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 [11] 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 4.1) 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 rewriteen satisfy the condition of Theorem 4.1.

Theorem 4.1 *A program will always produce facts with bound vector variables if in all rules "***_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, specification (MS6) does not derive uninstantianted vectors because the nonterminal `_Cond`, that may derive uninstantianted variables, shares variables with `$Table(_A)`. [9] provides a formal criterion for deciding whether the bottom-up evaluation derives facts that have vector variables. This criterion is used by the following algorithm that derives CSQs given a target query and a description.

Algorithm 1
Input: Target query Q and Description D
Output: A set of CSQs $s_i, i = 1, \ldots, n$
Method:
Check if the program derives

Reorder each template R in D such that facts with vector variables (see [9])
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 described in [9]

Note, template R can always be reordered. The proof appears in [9].

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) :- ⟨*predicate and metapredicate list*⟩,
subset(V,_W)

and _V does not appear in the ⟨*predicate and metapredicate list*⟩ 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 4.2 *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.* □

The intuitive proof of this claim is 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. Then, given that the plan refinement component considers all plans represented by a representative plan, we can be sure that the CBR algorithm does not lose any plan. The complete proof appears in [9].

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 [11]:

Definition 4.1 (**Maximal CSQs**) A CSQ s_m is a *maximal CSQ* if there is no other CSQ s that is subsumed by s_m. □

[5] In general, the ⟨*list of predicates and metapredicates*⟩ may contain metapredicates of the form **_in**(⟨*position*⟩,⟨*variable$_i$*⟩, _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.

Evaluation: In general, the CSQ discovery module generates only *maximal* CSQs and prunes all others. This pruning technique is particularly effective when the 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 4.3 *Pruning non-maximal CSQs does not lose any algebraically optimal plan.* □

The reason is that 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.

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 5.1 (**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 5.1 *Each CSQ has a unique* maximal *set of consumed subgoals that is a superset of every other set of consumed subgoals.* □

The proof of the uniqueness of the maximal consumed set appears in [9]. 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 2
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 5.2 (**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 (see the extended version [9]) where the relevance criteria may erroneously prune out a CSQ that could be part of a plan. 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 incorectly 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 (Q23); then we collect the department names (*i.e.*, the `D` bindings) and for each binding d of `D`, we replace the `$D` in (Q24) with d and send the instantianted query to the wrapper. We use the notation `/$D` in the nested loops plan (P25) to denote that (Q24) receives values for the `$D` placeholder from `D` *bindings* of the other CSQs – (Q23) in this example.

(Q23) $\texttt{answer}_{23}$`(F0,L0,D,O0):-emp(F0,L0,D,O0,'Y')`
(Q24) $\texttt{answer}_{24}$`(F1,L1,O1,M1):-emp(F1,L1,$D,O1,M1)`
(P25) `answer(F0,L0,O1) :-` $\texttt{answer}_{23}$
`(F0,L0,D,O0),` $\texttt{answer}_{24}$`(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, $\mathcal{B}_{24} = \{\texttt{D}\}$ and $\mathcal{B}_{23} = \{\}$. 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 that appears in [9] 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 $\mathcal{B}_{24}$ requires that a CSQ that exports `D` is executed before (Q24). 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 5.2 *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.* □

The proof is based on the theory of containment mappings appropriately extended to take into consideration nested loops [9].

The plan construction algorithm in the extended version of the paper [9] is based on Theorem 5.2. 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 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 procedes by picking CSQs and testing the conditions of Theorem 5.2 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 5.1 *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 5.1 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 5.1 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.

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 6.1 (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 $\mathcal{S}$.

Due to space limitations the complete plan refinement algorithm and its evaluation appear in [9]. Its main complication is due to the fact that unecessary variables cannot always be projected out when the maximal consumed sets of the CSQs intersect.

7 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 5.2 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.)

8 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 [12, 13, 4, 14] 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.[6] 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 [12] 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 [15]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 [16] 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."

[6] 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.

TSIMMIS suggests an explicit description of the wrapper's query capabilities [5], 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 [5] 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 [17, 6, 7, 18]. 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 [7]), or infinitely large queries/views. Also, they do not consider generating plans that require a particular evaluation order due to binding requirements.

[6] shows that rewriting a conjunctive query is in general exponential in the total size of the query and views. [18] shows that if the query is acyclic we can rewrite it in time polynomial to the total size of the query and views. [6, 7] 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). [19] proposed Datalog for the description of supported queries. It also suggested an algorithm that essentially finds what we call maximal CSQs. Recently [20] 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.

9 Practical Issues in the Implementation of a Capabilities-Based Rewriter

A capabilities-based rewriter has been implemented for Garlic using the Starburst's extensible optimizer [21] as implemented for DB2 [22]. The implemented version has enhanced some aspects of capabilities-based rewriting (see the list below) and has simplified some algorithms whenever this did not cause a loss of functionality. A detailed description of the implementation can be found in [23]. In this section we focus on higher level issues as they pertain to capabilities-based rewriting described in the previous sections.

- **Capabilities Description Language** In the implemented version 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)* in the Starburst terminology – 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. In particular, describing supported queries via plans allowed the use of the Starburst extensible optimizer for both the CSQ discovery and the plan construction phase. However, the use of plans for capabilities description complicates the translation from the Garlic's query language into the native languages of the sources. We plan to work on using a variant of RQDL for the capabilities description.

- **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 existing implementation introduced a special kind of join that implements the merging of "fuzzy" result sets using an algorithm outlined in [24]. However, we will not describe 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.[7] For the rest of this paper we assume that we deal with SPJ queries 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

[7] 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 [25].

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 a CSQ in order to issue the right query to the right source.

- **Capabilities Based Rewriting and Cost Based Optimization** 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 a dynamic programming algorithm for discovering the most efficient supported plan while at the same time it prunes the space of supported 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. An abstraction of the plan construction algorithm, which actually does not compromise completeness, is described in Section 9.3.

- **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, etc. – and 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.

Section 9.3 describes a dynamic programming algorithm which combines capabilities based rewriting with cost based optimization and shows that this algorithm is complete in the sense that it always finds the most efficient supported plan despite the pruning of the plan space.

9.1 Capabilities Description Using STARs

We describe wrapper and mediator capabilities in a uniform way using STARs [21]. Intuitively, STARs play the role of nonterminals of a grammar that defines sets of plans. Some STARs are designated to be the roots of the "grammar". In this section we illustrate this idea with examples without going into the formalities of the grammar. Note that the Starburst extensible optimizer discriminates subgoals into tables, predicates, and method calls. For the sake of conceptual simplicity we will neglect this discrimination and we will keep talking about subgoals regardless of whether they are tables, predicates, or methods in Starburst terms.

Consider a relational wrapper that can retrieve any relation. We can describe the capabilities of such a wrapper with the following non-recursive STAR.

```
AccessRoot() :  Retrieve($T, ds($T))
```

The above rule declares that the wrapper supports plans consisting of the `Retrieve` operator applied on any relation `$T`. We assume that rules ending in `Root` are roots of our grammar. Let us then describe that joins are also supported by adding another two rules.

```
JoinR() :  AccessRoot()
JoinR() :  Join(JoinR(), JoinR())
JoinRoot() :  JoinR()
```

The above says that we can recursively synthesize plans by joining plans which correspond to joins or retrievals of a single relation. Keep in mind that the semantics is that any query equivalent to a plan described above is supported. In reality, the heads of the STARs have parameters that declare the relations that have been joined, the bindings needed by this plan, etc. These parameters are used for controlling the pruning and other functions of the optimizer. For simplicity we do not denote these parameters in the example (the most important ones are described below).

STARs are also used to describe the capabilities of the mediator – i.e., the plans that the mediator can construct. Indeed, it is the common use of STARs for declaring wrapper and mediator capabilities that allows us to use the Starburst optimizer for both purposes.

9.2 Architecture of the Implementation

The optimizer operates in three phases. During the first phase the rules describing the wrappers produce plans supported by the wrappers – this phase corresponds to CSQ discovery. During the second phase the rules associated with the mediator construct plans that compute the target query. 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. Cost based optimization is also performed during plan construction, as is explained in the next section. Finally there is a phase of (essentially minor) refinements and fixes. For example, the attributes are placed in the order requested by the query.

9.3 Cost-Based Optimization and Capabilities-Based Rewriting

The plan construction algorithm described in Section 5 constructs all algebraically optimal plans. Then a cost based optimizer can evaluate their cost and select the most efficient. The disadvantage of this strategy is that, in general, the number of algebraically optimal plans is huge and involves all possible orders and policies in which we can execute the joins of the CSQs. Optimizers, such as DB2, have effective ways to prune the space of plans. The actual DB2 based implementation of the plan construction phase of the CBR uses a dynamic programming algorithm for pruning the space of plans *without* compromising completeness. First we describe the intuition behind pruning and then we describe the dynamic programming plan construction algorithm.

The intuition behind pruning in the DB2 implementation is that we do not need to keep track of all possible (sub)plans to execute a join of a set of CSQs S. We may select the most efficient (sub)plan and use it whenever we want to join a CSQ with the CSQs of S. The following example illustrates how pruning is achieved.

EXAMPLE 9.1 Consider the following target query

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

Assume that the source can answer any query that refers to only one relation and, optionally, has one or more equality conditions on the relation attributes. Given the above query the plan construction derives twenty-one algebraically optimal plans. In particular,

- The first nine plans correspond to the parenthisization $(p \bowtie q) \bowtie r$. Notice that there are three possible join policies for joining p and q, namely (i) the mediator can retrieve the full p and q and join them, or (ii) it can retrieve p and pass bindings to q, or (iii) it can retrieve q and pass bindings to p. Similarly, the join between $p \bowtie q$ and r can be done in three ways.
- The second nine plans correspond to the parenthesization $(q \bowtie r) \bowtie p$.
- Another three plans correspond to the parenthesization $(p \bowtie r) \bowtie q$ because there is only one way to execute $(p \bowtie r)$ (it is actually a cartesian product) and three ways to execute the join with q.

The dynamic programming algorithm does not have to construct all twenty one plans to find the most efficient one. Instead it proceeds in the following way:

- It costs the three possible ways for computing $p \bowtie q$, "keeps" the most efficient, say passing bindings from p to q, and prunes the other two. Then it costs the three possible policies for joining $p \bowtie q$ with r assuming $p \bowtie q$ is done using passing bindings from p to q.
- Similarly the best plan for $(q \bowtie r) \bowtie p$ is discovered.
- The plans corresponding to $(p \bowtie r) \bowtie q$ are never considered because the optimizer prunes cartesian products.

□

A Dynamic Programming Plan Construction Algorithm We now describe the dynamic programming plan construction algorithm. For the sake of presentation simplicity we enhance the plan construction algorithm with dynamic programming pruning. Before we proceed we recall the following two important points of the implementation:

1. The dynamic programming functionality is an inherent feature of the DB2 optimizer.
2. The implemented algorithm represents CSQs via plans instead of queries. This difference, though important for employing the DB2 optimizer, is not essential for understanding the nature of combining cost based optimization with capabilities based rewriting.

The idea behind pruning is that if two plans P and P' consume the same set of subgoals and accept the same parameters then they are not both needed. We may retain only the cheaper, say P, without compromising completeness, since it is clear that any plan that uses P' can replace it with P. We first define formally the "consumed subgoals of a plan P" and the "set of bindings needed by a plan P". Then we provide the algorithm.

Definition 9.1 (**Consumed Subgoals of a plan** P) Every plan P is associated with a set of consumed subgoals as defined by the following rules:

- A plan P of the form `submit(s)` consumes the subgoals of s.

- A plan P of the form `join(`P_1`,`P_2`)` consumes the subgoals of P_1 and P_2 and it also consumes the predicate $X = Y$ that corresponds to the join. Notice that this definition can easily generalized to more complex join predicates.

- A Plan P of the form `bindjoin(`P_1`,`P_2`)` consumes the subgoals of P_1 and P_2. (Notice that the equality predicate that does the join is part of the consumed set of P_2.)

□

Definition 9.2 (**Set of bindings needed by a plan** P) Every plan P is associated with a set of needed bindings as defined by the following rules:

- A plan P of the form `submit(`s`)` needs the same bindings as s.
- A plan P of the form `join(`P_1`,`P_2`)` needs the bindings of both P_1 and P_2.
- A plan P of the form `bindjoin(`P_1`,`P_2`)` needs the binding of both P_1 and P_2, excluding the bindings of P_2 that are instantianted during the `bindjoin`.

□

Definition 9.3 (**Equivalent Plans**) Two plans P and P' are equivalent if they have identical sets of consumed subgoals and needed bindings. □

INPUT: (1) a target query q and a set $\mathcal{S}$ of CSQs of q
(2) a cost function f that given a plan consisting of CSQs combined with selections, projections, and joins assigns a cost
OUTPUT: The plan that computes q (if there is one) and has the least cost
METHOD:
For every CSQ s insert into the set of plans $\mathcal{A}$ a plan `submit(`s`)`
For $i = 2, \ldots, n$ where n is the number of subgoals in q
  Construct all plans $P =$`join(`P_1`,`P_2`)` or $P =$`passbindings(`P_1`,`P_2`)` where $P_1, P_2 \in \mathcal{A}$
  If there is no plan $P' \in \mathcal{A}$ equivalent to P such that $f(P') < f(P)$
    insert P in $\mathcal{A}$
    If there is a P' equivalent to P with $f(P') > f(P)$
      remove P' from $\mathcal{A}$
The plan P consuming all subgoals of q is the optimal supported plan

10 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 implementation of CBR for the Garlic system. The implementation is based on Starburst's extensible optimizer and combines capabilties-based rewriting with cost-based optimization and pruning.

Acknowledgements

We are grateful to Mike Carey, Hector Garcia-Molina, Anand Rajaraman, Anthony Tomasic, Jeff Ullman, Ed Wimmers, and Jennifer Widom for many fruitful discussions and comments.

References

[1] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. RIDE-DOM Workshop*, pages 124–31, 1995.

[2] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[3] J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.

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

[5] 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.

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

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

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

[9] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. Available via ftp at `db.stanford.edu` file `/pub/papakonstantinou/1995/cbr-extended.ps`.

[10] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems.* Computer Science Press, New York, NY, 1988.

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

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

[13] 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.

[14] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

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

[16] A. Levy, A. Rajaraman, and J. Ordille. Query processing in the information manifold. In *Proc. VLDB*, 1996.

[17] P.A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB Conf.*, pages 259–69, 1985.

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

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

[20] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. Available via `http://www-cse.ucsd.edu/ yannis/`.

[21] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD*, 1988.

[22] 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.

[23] L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. IBM Technical Report.

[24] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. PODS*, 1996.

[25] S. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27:633–55, 1980.

11 Algorithm for Plan Construction

Algorithm 4

Input: A set of CSQs $\{s_1, \ldots, s_m\}$
A target query Q

Output: A set of plans that satisfy Theorem 5.2 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 (the first CSQs of the plan's tail)
% 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

(0) $\langle$*description*$\rangle$	::=	$(\langle$*query template*$\rangle \vert \langle$*nonterminal template*$\rangle)*$
(1) $\langle$*query template*$\rangle$	::=	**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*$\rangle$	::=	ϵ *%subgoal list may be emtpy*
(5) $\langle$*subgoal*$\rangle$	::=	$\langle$*predicate*$\rangle$ ($\langle$*arguments*$\rangle$) *%predicate*
(6) $\langle$*subgoal*$\rangle$	::=	$\langle$*metapredicate name*$\rangle$ ($\langle$*arguments*$\rangle$) *%metapredicate*
(7) $\langle$*subgoal*$\rangle$	::=	$\langle$*nonterminal name*$\rangle$ ($\langle$*arguments*$\rangle$) *%nonterminal*
(8) $\langle$*arguments*$\rangle$	::=	$\langle$*vector*$\rangle \vert \langle$*variable*$\rangle$ (, $\langle$*variable*$\rangle)*$
(9) $\langle$*predicate name*$\rangle$	::=	$\langle$*identifier*$\rangle \vert \langle$*placeholder*$\rangle$
(10) $\langle$*metapredicate name*$\rangle$	::=	$\langle$*identifier*$\rangle$
(11) $\langle$*nonterminal name*$\rangle$	::=	$\langle$*identifier*$\rangle$

Figure 3: Normal-form RQDL syntax

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
$\mathcal{C}_s$ has at least one subgoal not in $sub(P)$ *% s consumes at least one more subgoal*
% 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 5.2
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*

12 Algorithm for Plan Refinement

Algorithm 5
Input: Plan P involving representative CSQ s.
Output: One or more 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 6.1.
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 3.1) than P''

discard P' and exit loop
else if P'' is algebraically worse than P'
discard P''

13 Bottom-up Algorithm for deriving CSQs

The algorithm below ensures that only CSQs with appropriate binding patterns are derived.

Algorithm 6
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 7
% Eliminate facts that use bindings not yet available
Eliminate facts that have in annotation (b) a frozen constant x where $x \notin A$
% Eliminate facts that do not use at least one new binding
Eliminate facts that do not have in annotation (b) a 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)

The algorithm below describes how to evaluate a single rule bottom up in a magic-sets rewritten program.

Algorithm 7
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
% constants unify with similarly named constants
% place holders unify with constants/frozen constants
% variables unify with constants, frozen constants, variables
% vector variables unify with vector variables, vectors of constants/variables
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 $\mathcal{B}_n$, same set of exported variables, than some existing fact n' do not add n to set of derived facts.

14 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 3. 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 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 14.1 (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

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

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

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

and a nonterminal template of the form

$$N(\langle \textit{definition arguments} \rangle) \ : \ \langle N \ \textit{definition body} \rangle$$

where

(a) the nonterminal template and the expansion e have no common variable,
(b) there is a collection of mappings θ such that $\theta(N(\langle \textit{arguments} \rangle))$ is identical to $\theta(N(\langle \textit{definition arguments} \rangle))$. We call θ a *unifier*. Definition 14.2 formally defines the application of a unifier on an RQDL expression.

□

Definition 14.2 (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 $\langle \textit{placeholder} \rangle \mapsto \langle \textit{constant} \rangle$, or $\langle \textit{variable} \rangle_1 \mapsto \langle \textit{variable} \rangle_2$, or $\langle \textit{vector} \rangle_1 \mapsto \langle \textit{vector} \rangle_2$ then replace all instances of $\langle \textit{placeholder} \rangle$, $\langle \textit{variable} \rangle_1$, and $\langle \textit{vector} \rangle_2$ with $\langle \textit{constant} \rangle$, $\langle \textit{variable} \rangle_2$, or $\langle \textit{vector} \rangle_2$ respectively.
2. If θ contains a mapping of the form $\langle \textit{vector} \rangle \mapsto [\langle \textit{variable list} \rangle]$ replace all instances of $\langle \textit{vector} \rangle$ that appear in metapredicates with $[\langle \textit{variable list} \rangle]$ and all the other instances with $\langle \textit{variable list} \rangle$.

□

Definition 14.3 (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 14.4 (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 rerive by sungoals permutations and variable renaming.