Paper Number 274

Capabilities-Based Query Rewriting in Mediator Systems (extended version)*

Yannis Papakonstantinou† Ashish Gupta ‡ Laura Haas§
IBM Almaden Research Center

Abstract

Users today are struggling to integrate a broad range of information sources that provide 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 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).* We provide a language to describe the query capability of data sources. The CBR takes as input a description of each data sources' capability. Given a query posed to the mediator, the CBR determines the component queries to be sent to the sources, commensurate with their abilities. It also computes a plan for combining the results of the component queries using joins, unions, selections, and projections. Our description language and plan generation algorithm are schema independent and handle SPJ queries.

1 Introduction

Organizations today require integrating 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 powerful query language, with location transparency, even though the underlying systems have such diverse capabilities.

Figure (1.a) shows one commonly proposed integration architecture [C+94, PGMW95, FK93, A+91]. Each data source has a *wrapper*, that 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 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

*Research partially supported by Wright Laboratories, Wright Patterson AFB, ARPA Contract F33615-93-C-1337.
†Current address: Computer Science Dept, Stanford University, CA 94305, email: yannis@DB.Stanford.EDU
‡Current address: Oracle Corporation, email: ashgupta@us.oracle.com
§email: laura@almaden.ibm.com

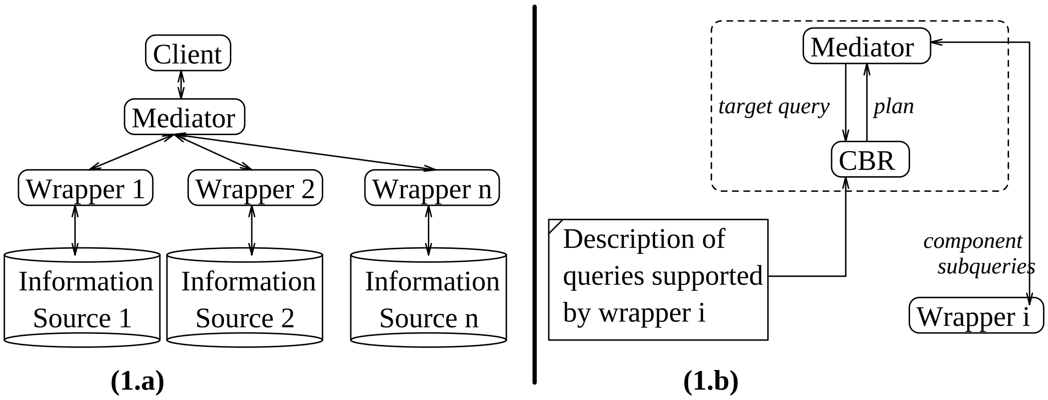

Figure 1: (a) A typical integration architecture. (b) CBR-mediator interaction.

wrappers, needed to support a powerful query interface against possibly quite limited data sources. Alternatively, it may lead to a "lowest common denominator" approach in which simple queries are sent to the sources, not exploiting the search capabilities of more sophisticated data sources, and hence forcing most of the work on the mediator 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 handle the differences and limitations in capabilities, and ensure that wrappers receive only queries that they can handle.

For Garlic [C+94], 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 class of queries, which we call the *supported queries* for that wrapper. When the mediator receives a query from a client, it decomposes it into a set of queries, each of which references data at a single wrapper. We call these individual queries *target queries* for the wrappers. A target query need not be a supported query; and it may sometimes be necessary to further decompose it into simpler supported *Component SubQueries (CSQs)*. A *plan* is formed to combine the results of the CSQs to produce the answer to the target query.

To obtain this functionality, we are exploring a *Capabilities-Based Rewriter (CBR)* module (Figure 1.b) that interacts with 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 and the CBR interact as follows. The mediator develops the target query for each wrapper w without considering whether the query is supported by w. It then passes this target query to the CBR for "inspection." The CBR compares the target query against the description of the queries supported by wrapper w, and tells the mediator one of three things. (i) The target query is directly supported by w. (ii) The target query is computable by the mediator through a plan that involves selection, projection and join of CSQs that are supported by the wrapper. (iii) The target query is not supported directly and the CBR cannot find a plan to compute the query.

The CBR allows a clean separation of wrapper capabilities and mediator internals. 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 for enabling the CBR approach. We first present a language for the description of wrappers' query capabilities. This language is the input the CBR needs to understand the set of queries supported by the wrapper. This paper focuses on Select-Project-Join queries. The descriptions look like context-free grammars – suitably modified to describe queries rather than arbitrary strings – that allow the designer to succinctly describe sets of supported queries. The descriptions may be recursive thus allowing the description of infinitely large supported queries. In addition, they may or may not be schema-specific. 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 [PGGMU] and we will not discuss it further in this paper.

The second contribution of this paper is a CBR architecture and an algorithm to build plans for answering a target query using the CSQs supported by the underlying sources. This problem is a generalization of the problem of determining if a query can be answered using a set of materialized queries/views [LMSS95, RSU95]. 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 [LMSS95, RSU95]. The CBR algorithm uses optimizations and heuristics to eliminate sources of exponentiality in the cases that arise in wrappers of the most common information sources such as bibliographic databases, lookup catalogs, relational systems, object repositories (represented relationally) and so on.

Paper outline In Section 2 we present the description language of a wrapper's query capabilities. In Section 3 we describe the basic architecture of the CBR, illustrating how it develops plans and discussing some techniques for eliminating exponentiality. Sections 4 and 5 detail the CBR modules for constructing plans and refining them, respectively. Section 6 presents the algorithm for discovering component queries. Section 7 analyzes the expressive ability and run-time performance of the CBR. Section 8 compares the CBR with related work. Finally, Section 9 concludes with some directions for future work in this area.

2 The Relational Query Description Language(RQDL)

In this section, we present a language to describe the supported queries for each wrapper. We introduce the basic language features in section 2.1, followed in sections 2.2 and 2.3 with 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 syntax and semantics of the language appears in Appendix A.

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.[1] 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 succesful 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 [Ull88] as our query language because it is well-suited to handling SPJ queries and facilitates the discussion of our algorithms.[2] 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) (QT2.1) answer(F,L,D,O,M) :- emp(F,L,D,O,M)
     (QT2.2) answer(F,L,D,O,M) :- emp(F,L,D,O,M), prefix(L, $LP), _isalpha($LP)
     (QT2.3) answer(F,L,D,O,M) :- emp(F,L,D,O,M), prefix(L, $LP), prefix(F,$FP),
                                  _isalpha($LP), _isalpha($FP)
```

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

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

2.2 Descriptions of Large and Infinite Sets of Supported Queries

In this section we extend RQDL to describe arbitrarily large sets of templates (and hence queries), by introducing nonterminals similarly to 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

[1] The "lookup" facility is very similar to a Stanford University facility.

[2] 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.

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

To see that description (D4) describes query (Q3), start by replacing `_Cond(F,L,D,O,M)` in (QT4.1) with the nonterminal template (NT4.4). We obtain the expansion (E5):

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

The variable `Office` of (NT4.4) is replaced by `O` because `Office` appears in the parameters of `_Cond` and hence it is *unified* with `O`. We replace again the `_Cond` in (E5) with (NT4.4), thus producing the expansion (E6).

```
(E6) answer(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,$OS),
                          substring(O,$OS1), _Cond(F,L,D,O,M)
```

In expansion (E6) the placeholder `$OS` that appears in (NT4.4) is renamed as `$OS1` to avoid confusion with the placeholder `$OS` that already is in (E5). Finally, we replace the `_Cond` in (E6) with the empty template (NT4.6) thus producing expansion (E7), that describes the query (Q3), *i.e.*, the placeholders and variables of (E7) can be mapped to the constants and variables of (Q3).

(E7) `answer(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,$OS), substring(O,$OS1)`

2.3 Schema Independent Descriptions of Supported Queries

Description (D4) assumes that a fixed schema is exported by the wrapper. 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 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 with 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 with *_R* appears in the list that matches with $\_A$. **_in**(`$Position, X,` $\_A$) succeeds if $\_A$ matches with a variable list, and there is a query variable that matches with `X` and appears at the position number that matches with `$Position`.

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. (D8) uses vectors and the built-in metapredicates to describe the queries supported by `file-wrap`. Note, for readability we will use *italics* for vectors and **bold** for metapredicates.

(D8) (QT8.1) `answer(`$\_R$`) :- $Table(`$\_A$`), _Cond(`$\_A$`),` **_subset**(`$\_R$`, `$\_A$`)`
(NT8.1) `_Cond(`$\_A$`) :` **_in**(`$Position, X, `$\_A$`), substring(X, $S), _Cond(`$\_A$`)`
(NT8.2) `_Cond(`$\_A$`) :`

In general, finding whether a query is described by a template containing vectors requires expanding nonterminals (as described above), mapping variables, placeholders, and vectors (see below), and finally, evaluating metapredicates. To illustrate this, let us follow the steps that prove that query (Q9) is described by (D8).

(Q9) `answer(L,D) :- emp(F,L,D,O,M), substring(O,'alma'), substring(O,'B')`

First, we expand (QT8.1) by replacing the nonterminal `_Cond` with (NT8.1). Then again we expand the result with (NT8.1), and finally with (NT8.2), thus obtaining expansion (E10).

(E10) `answer(`$\_R$`) :- $Table(`$\_A$`),` **_in**(`$Position,X,`$\_A$`), substring(X,$S),`
_in(`$Position1,X1,`$\_A$`), substring(X1,$S1),` **_subset**(`$\_R$`,`$\_A$`)`

Expansion (E10) describes query (Q9) because there is mapping of variables, vectors, and placeholders of (E10) that makes the metapredicates succeed and makes every predicate of the expansion identical to a predicate of the query. Namely, map vector $\_A$ 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 be repetitious, as in description (D4). In our running example, even if the attributes of the tables are known, we save effort by not having to explicitly mention the columns and the table names when saying that a substring condition can be placed on any column of any tuple.

2.4 Query and Description Normal Form

If we allow the 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 (Q11) that 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`.

(Q11) `answer(FL,D) :- emp(FL,FL,D,O,'Y')`

(Q11) should not be described by the description (D8). Nevertheless, we are still able to construct the expansion (E12) that erroneously matches with the query (Q11) if we map $\_A$ to `[FL,FL,D,O,'Y']` and $\_R$ to `[FL,D]`.

(E12) `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 then we 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. E.g., query (Q13) is the normal form of (Q11).

(Q13) `answer(FL1,D) :- employee(FL1,FL2,D,O,M), FL1=FL2, M='Y'`

Description (D8) does not describe (Q13) because (D8) does not support the equality conditions that appear in (Q13). Description (D14) supports equality conditions on any column and equalities between any two columns: (NT14.2) describes equalities with constants and (NT14.3) describes equalities between the columns of our table.

```
(D14) (QT14.1) answer(_R)  :- $Table(_A), _Cond(_A), _subset(_R, _A)
      (NT14.1) _Cond(_A) :  _in($Position,X,_A), substring(X, $S), _Cond(_A)
      (NT14.2) _Cond(_A) :  _in($Position1,X,_A), X=$C, _Cond(_A)
      (NT14.3) _Cond(_A) :  _in($Pos1,X,_A), _in($Pos2,Y,_A), X=Y, _Cond(_A)
      (NT14.4) _Cond(_A) :
```

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

3 The Capabilities-Based Rewriter

The Capabilities-Based Rewriter (CBR) upon receiving a target query q determines whether q is diretly supported by the appropriate wrapper, *i.e.*, whether it matches with 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 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. The CBR consists of three modules, which are invoked serially (see Figure 2):

- **CSQ discovery:** discovers Component SubQueries (CSQs), that are 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 the CSQ detection module. 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.

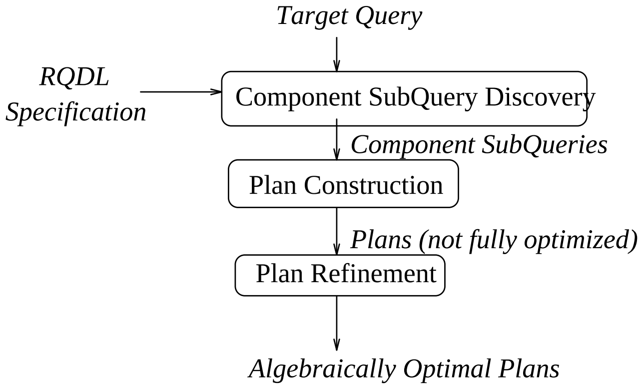

Figure 2: The CBR's components

- **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 (Q15), 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 (D14).

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

The CSQ detection module identifies and outputs the following CSQs:

(Q16) `answer`$_{16}$`(F0,L0,D,O0) :- emp(F0,L0,D,O0,'Y')`
(Q17) `answer`$_{17}$`(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 (Q16) but export a different subset of the variables `F0`, `L0`, `D`, and `O0` (likewise for (Q17)). The CSQs that export fewer variables are pruned.

The plan construction module detects that a join on `D` of `answer`$_{16}$ and `answer`$_{17}$ produces the required `answer` of (Q15). Consequently, it derives the plan (P18).

(P18) `answer(F0,L0,O1) :- answer`$_{16}$`(F0,L0,D,O0), answer`$_{17}$`(F1,L1,D,O1,M1)`

Finally, the plan refinement module detects that variables `O0`, `F1`, `L1`, and `M1` in `answer`$_{16}$ and `answer`$_{17}$ are unnecessary. Consequently, it generates the more efficient plan (P21).

(Q19) `answer`$_{19}$`(F0,L0,D) :- emp(F0,L0,D,O0,'Y')`
(Q20) `answer`$_{20}$`(D,O1) :- emp(F1,L1,D,O1,M1), substring(O1, 'B')`
(P21) `answer(F0,L0,O1) :- answer`$_{19}$`(F0,L0,D), answer`$_{20}$`(D,O1)`

□

We delay discussing the CSQ discovery algorithm until Section 6, as understanding how the CSQs are used is key to understanding the algorithm. However, before we discuss plan construction we describe in the remainder of this section two important techniques used by the CSQ discovery algorithm to prune the space of CSQs. These techniques are important because they restrict the size of the input to the plan construction module. First, we formally define an algebraically optimal plan.

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.) □

3.1 Pruning "Represented" CSQs

A large number of unneeded CSQs are generated by templates that use vectors and the **_subset** metapredicate. For example, template (QT14.1) 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 s that are derived from the same expansion e, of some template t, where e has the form

answer(*_V*) :- ⟨*list of predicates and metapredicates*⟩, **_subset**(*_V*,*_W*)

and *_V* does not appear in the ⟨*list of predicates and metapredicates*⟩ we generate only the *representative* CSQ that is derived by mapping *_V* to the same variable list as *_W*.[3] 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 (Q17) and the represented CSQ (Q20) both are derived from the expansion (E22) of template (QT14.1).

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

The CSQ discovery module generates only (Q17) and not (Q20) because (Q17) is derived 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.

Note that pruning represented 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 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 Section 5.

Evaluation For every representative CSQ with head arity a, pruning the represented queries eliminates $2^a - 1$ represented queries, 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 5). Thus, though postponing projection pushes part of the complexity to a later stage, it eliminates some complexity altogether. The eliminated complexity corresponds to those representated CSQs that finally do not participate in any plan because they retain too few variables.

---

[3] 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.

3.2 Pruning Non-Maximal CSQs

In general, the CSQ discovery module generates only *maximal* CSQs and prunes all others. 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 based on the notion of subsumption [Ull89]:

Definition 3.2 (**Maximal CSQs**) A CSQ s_m is a *maximal CSQ* if there is no other CSQ s that is subsumed by s_m. □

This pruning technique is particularly effective when the CSQs contain a large number of conditions. For example, assume that g conditions are applied on 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.

Finally, note that 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.

4 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 4.1 addresses (i) with the notion of "subgoal consumption." Section 4.2 checkes (ii), *i.e.*, checks join variables. Section 4.3 checks (iii) by ensuring bindings are available. Finally, Subsection 4.4 summarizes the conditions required for constructing a plan and provides an efficient plan construction algorithm.

4.1 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 provides all the information needed from that subgoal. A CSQ does not necessarily consume all its subgoals. For example, consider a wrapper, called `semijoin`, that supports queries that may include any set of predicates, with arbitrary join conditions between them, but that can export data from only one of the predicates. This is a typical situation in object-oriented databases, which export information of one class only, though the query may involve more than one class. Assuming that `semijoin` exports predicates `emp` and `substring` we can see that `semijoin` supports the CSQs (Q23) and (Q24), which allow us to compute (Q15) using plan (P25).[4] ((Q23) and (Q24) are representative CSQs.)

```
(Q23) answer23(F0,L0,D,O0) :- emp(F0,L0,D,O0,'Y'), emp(F1,L1,D,O1,M1),
                              substring(O1,'B')
(Q24) answer24(F1,L1,D,O1,M1) :- emp(F0,L0,D,O0,'Y'), emp(F1,L1,D,O1,M1),
                                 substring(O1,'B')
(P25) answer(F0,L0,O1) :-  answer23(F0,L0,D,O0), answer24(F1,L1,D,O1,M1)
```

[4] `semijoin` does not directly support the query (Q15) because (Q15) exports variables from both its `emp` subgoals.

In the above example (Q23) consumes only the subgoal `emp(F0,L0,D,O0,'Y')` while (Q24) consumes the subgoals `emp(F1,L1,D,O1,M1)` and `substring(O1,'B')`. Intuitively, (Q23) consumes the `emp` subgoal because it exports the distinguished variables `F0` and `L0` that appear in its consumed subgoal and also exports the join variable `D` that connects the consumed subgoal to the non-consumed subgoals. Thus, it provides all the information needed from the consumed subgoal. We formalize the above intuition by the following definition:

Definition 4.1 (**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 4.1 *Every subquery s has a unique* maximal *set $\mathcal{C}_s$ of consumed subgoals.*
Proof: s has at least one set of consumed subgoals (trivially, the empty set is a set of consumed subgoals.) and hence it has at least one maximal set of consumed subgoals. Let us assume that there are two maximal sets $\mathcal{C}_s^1$ and $\mathcal{C}_s^2$. Then, their union is also a consumed set since it satisfies both conditions of definition 4.1. Hence, $\mathcal{C}_s^1$ and $\mathcal{C}_s^2$ can not simoultaneously be maximal consumed sets. □

The maximal sonsumed 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.

Algorithm 1
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: *% remove subgoals that are not in* $\mathcal{C}_s$
      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 4.2 (**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 4.2) produces each plan in time polynomial in the number of CSQs produced by the CSQ discovery module. However, there are rare scenarios where the relevance criteria may erroneously prune out a CSQ that could be part of a plan. The following example provides such a scenario.

EXAMPLE 4.1 Consider a variation of `file-wrap` where the supported queries that accept `substring` conditions may output only the first and last name, e.g., (Q27). There is also a supported query – (Q28) – that returns the whole `emp` table. The target query (Q26), that requests the full information of employees in the 'database' department can be answered by the plan (P29).

(Q26) `answer(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,'database')`
(Q27) `answer`$_{27}$`(F,L) :- emp(F,L,D,O,M), substring(O,'database')`
(Q28) `answer`$_{28}$`(F,L,D,O,M) :- emp(F,L,D,O,M)`
(P29) `answer (F,L,D,O,M) :- answer`$_{27}$`(F,L), answer`$_{28}$`(F,L,D,O,M)`

CBR will not find the plan (P29) because the CSQ (Q28) does not consume any subgoal because it does not export the distinguished variables `D`, `O`, and `M`. Intuitively, (Q28) does not contribute subgoals but it contributes variables. □

We may avoid the loss of plans such as (P29) by not pruning irrelevant CSQs and thus sacrificing the polynomiality of the plan construction algorithm. In this paper we will not consider this option.

4.2 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, assume a version of `file-wrap`, called `strange`, that can export either first name, last name and department, or first name, last name and office address. Furthermore, `strange` allows at most one `substring` condition in every query, as shown in description (D30).

(D30) (QT30.1) `answer(F,L,D) :- emp(F,L,D,O,M), _Cond([F,L,D,O,M])`
(QT30.2) `answer(F,L,O) :- emp(F,L,D,O,M), _Cond([F,L,D,O,M])`
(NT30.3) `_Cond(_V) : _in($Position, Var, _V), substring(Var, $String)`
(NT30.4) `_Cond(_V) :`

Consider target query (Q31), CSQs (Q32) and (Q33), and plan (P34).

(Q31) `answer(F,L) :- emp(F,L,D,O,M), substring(D,'data'), substring(O, 'B')`
(Q32) `answer`$_{32}$`(F,L,O) :- emp(F,L,D,O,M), substring(D,'data')`
(Q33) `answer`$_{33}$`(F,L,D) :- emp(F,L,D,O,M), substring(O,'B')`
(P34) `answer(F,L) :- answer`$_{32}$`(F,L,O), answer`$_{33}$`(F,L,D)`

The union of the maximal consumed set of (Q32) {`emp(F,L,D,O,M), substring(D,'data')`} with the maximal consumed set of (Q33) {`emp(F,L,D,O,M), substring(O,'B')`} is equal to the target query's subgoals set. Nevertheless, plan (P34) is not correct[5]. Intuitively, the problem arises because (Q32) does not export the join variable `D` that is required by (Q33) and, vice versa. We can avoid this problem by checking that the combined CSQs export the join variables necessary for their successful combination. The theorem of Section 4.4 formally describes the conditions on join variables that guarantee the correct combination of CSQs.

[5] To see this, assume that the `emp` relation has two John Browns, one in the `database` department, with office C32, and the other in distributed systems, in `B15`.

4.3 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 (Q15) by the following steps: first we execute (Q35); then we collect the department names (*i.e.*, the `D` bindings) and for each binding d of `D`, we replace the `$D` in (Q36) with d and send the instantianted query to the wrapper. We use the notation `/$D` in the nested loops plan (P37) to denote that (Q36) receives values for the `$D` placeholder from `D` *bindings* of the other CSQs – (Q35) in this example.

(Q35) `answer`$_{35}$`(FO,LO,D,OO) :- emp(FO,LO,D,OO,'Y')`
(Q36) `answer`$_{36}$`(F1,L1,O1,M1) :- emp(F1,L1,$D,O1,M1)`
(P37) `answer(FO,LO,O1) :- answer`$_{35}$`(FO,LO,D,OO), answer`$_{36}$`(F1,L1,O1,M1)/$D`

The introduction of nested loops and *binding passing* requires increased functionality from the CSQ discovery and plan construction components:

- **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 `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, $\mathcal{B}_{36} = \{\mathtt{D}\}$ and $\mathcal{B}_{35} = \{\}$.
- **plan construction:** The bindings needed by each CSQ of a plan impose order constraints on the plan. For example, the existence of `D` in $\mathcal{B}_{36}$ requires that a CSQ that exports `D` is executed before (Q36). It is the responsibility of the plan construction module to ensure that the produced plans satisfy the order constraints.

4.4 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 4.2 *Given CSQs* $s_i, i = 1, \ldots, n$ *with corresponding heads* $\mathtt{answer}_i(A^i_1, \ldots, A^i_{a_i})$*, sets of maximal consumed subgoals* $\mathcal{C}_i$ *and sets of needed bindings* $\mathcal{B}_i$*, the plan*

$$\mathtt{answer}(H_1, \ldots, H_m) : -\mathtt{answer}_1(A^1_1, \ldots, A^1_{a_1}), \ldots, \mathtt{answer}_n(A^n_1, \ldots, A^n_{a_n})$$

is correct if

- **consumed sets condition:** *The union of maximal consumed sets* $\cup_{i=1,\ldots,n}\mathcal{C}_i$ *is equal to the target query's subgoal set.*
- **join variables condition:** *If the set of maximal consumed subgoals of CSQ* s_i *has a join variable* V *then every CSQ* s_j *that contains* V *in its set of maximal consumed subgoals* $\mathcal{C}_j$ *exports* V*.*
- **bindings passing condition:** *If* $V \in \mathcal{B}_i$ *then there must be a CSQ* $s_j, j < i$ *that exports* V*.*

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

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

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

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

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

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

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

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

Using 3 we can rewrite 2 as follows:

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

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

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

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

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

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

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

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

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

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

- otherwise, we map V_k^j to X_k^j.

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

Algorithm 2 for plan construction is based on Theorem 4.2. 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, Algorithm 2 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, the bindings passing condition imposes a partial order on how CSQs are executed in a plan. By ensuring that $\prec$ respects the partial order $\stackrel{b}{\prec}$, we guarantee that during plan generation CSQs are considered in an order that also allows pruning based on insufficient binding values being passed.

The first phase of Algorithm 2 sorts the input CSQs in a total order that respects the PO $\stackrel{b}{\prec}$. The second phase procedes by picking CSQs and testing the conditions of Theorem 4.2.

Algorithm 2

Input: A set of CSQs $\{s_1, \ldots, s_m\}$
  A target query Q
Output: A set of plans that satisfy Theorem 4.2 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* $\stackrel{b}{\prec}$
  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*
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 4.2
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*

As we mentioned above, Algorithm 2 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 4.1 *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 4.1 holds for typical wrappers of bibliographic information systems and lookup services (wrappers that have the structure of (D14)), 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 4.1 cover essentially any wrapper of practical importance. Thus allowing Algorithm 2 to generate a time polynomial in the number of CSQs.

5 Plan Refinement

The plan refinement module filters and refines constructed plans in two ways. First, it projects out unnecessary variables of representative CSQs. For this purpose it computes the set of *necessary* variables of each representative CSQ. Intuitively, this set contains the variables that allow the consumed set of the CSQ to "interface" with the consumed sets of other CSQs in the plan. We formalize the notion and the significance of necessary variables by the following definition (note, we do not restrict the definition to maximal consumed sets:)

Definition 5.1 (**Necessary Variables of a Set of Consumed Subgoals:**) A variable V is a necessary variable of a consumed subgoals set $\mathcal{S}_s$ of some CSQ s if by not exporting V $\mathcal{S}_s$ is no longer a consumed set. □

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

The second contribution of plan refinement is the elimination of all plans 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.

Projecting out unecessary variables calls for caution when the maximal consumed sets of the CSQs intersect. For example, consider a wrapper that exports predicates `emp` and `substring`. Every supported query has exactly one `emp` subgoal, at most one `substring` subgoal, and may export any subset of the `emp` variables. The target query (Q38) can be computed by plan (P41).

(Q38) `answer(F,L) :- emp(F,L,D,O,M), substring(D,'data'), substring(O,'B')`
(Q39) `answer`$_{39}$`(F,L,D,O,M) :- emp(F,L,D,O,M), substring(D,'data')`
(Q40) `answer`$_{40}$`(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,'B')`
(P41) `answer(F,L) :- answer`$_{39}$`(F,L,D,O,M), answer`$_{40}$`(F,L,D,O,M)`

Having both queries export all the variables is useless. An obvious optimization is to replace (Q40) with (Q42), which exports only the distinguished variables `F` and `L` and the join variable `D`.

(Q42) `answer`$_{42}$`(F,L,D) :- emp(F,L,D,O,M), substring(O,'B')`

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

However, reducing the exported variables of each representative query to the necessary variables of its maximal consumed set may result in an incorrect plan. For example, replacing CSQ (Q39) with CSQ (Q43) we construct the erroneous plan (P44). (P44) violates the join variables condition.

(Q43) `answer`$_{43}$`(F,L,O) :- emp(F,L,D,O,M), substring(D,'data')`
(P44) `answer(F,L) :- answer`$_{42}$`(F,L,D), answer`$_{43}$`(F,L,O)`

The problem arises because the maximal consumed sets of (Q39) and (Q40) intersect. It can be solved as follows: Since CSQ (Q42) consumes the subgoals `emp(F,L,D,O,M)` and `substring(O,'B')` we can modify the exported variables of the representative CSQ (Q39) so that it consumes only the subgoal `substring(D,'data')`. Thus, we can replace the representative CSQ (Q39) with the CSQ (Q45) that exports only the necessary variables of the set {`substring(D,'data')`}, *i.e.*, `D`. Consequently, we can construct the plan (P46).

(Q45) `answer`$_{45}$`(D) :- emp(F,L,D,O,M), substring(D,'data')`
(P46) `answer(F,L) :- answer`$_{42}$`(F,L,D), answer`$_{45}$`(D)`

Symmetrically, we may assume that (Q39) consumes `emp(F,L,D,O,M)` and `substring(D,'data')` in which case (Q40) consumes only `substring(O, 'B')` and hence we can produce the plan (P48).

(Q47) `answer`$_{47}$`(O) :- emp(F,L,D,O,M), substring(O,'B')`
(P48) `answer(F,L) :- answer`$_{43}$`(F,L,O), answer`$_{47}$`(O)`

Intuitively, the plans (P46) and (P48) correspond to two different partitions of the target query's subgoals among the sets of consumed subgoals the two representative CSQs. In general, given a representative plan, we may produce all plans that implement projections by partitioning the target query subgoals among the representative CSQs. Thus, subgoals that are in the consumed sets of more than one representative query are "assigned" to only one representative query. Then, we calculate the necessary variables for the "reduced" consumed sets of the representative queries.

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

Algorithm 3

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 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 5.1.
        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''
```

Algorithm 3 is exponential in the size of $\mathcal{C}_s$. However, it can be optimized by observing the following. If some subgoal in the maximal consumed set of s is not in the maximal consumed set of any other CSQ in plan P, then this subgoal necessarily has to be present in all non-discarded subsets M. Thus, options are generated only by subgoals consumed by multiple CSQs. Thus, the algorithm becomes exponential in the size of the largest intersection of the consumed sets of the representative CSQs.

Finally, we provide the following theorem that guarantees that the combination of plan refinement with pruning represented CSQs does not cause the loss of some algebraically optimal plan.

Theorem 5.1 *If there is an algebraically optimal plan p_s that involves a represented CSQ s then p_s will be discovered by CBR (provided it contains relevant CSQs only).*

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

6 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 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 and compare the sets of underlying facts of derived facts to prune out non-maximal CSQs.

As for rule production systems, we had to choose between top-down versus bottom-up derivation. The CSQ discovery module uses bottom-up because bottom-up evaluation is guaranteed to terminate even for recursive descriptions [Ull89] whereas top-down derivation may not always terminate unless we restrict the forms of recursion that we accept. Conversely, bottom-up often derives unnecessary facts, unlike top-down. We use a variant of *magic sets rewriting* [Ull89] to "focus" the bottom-up derivation.

We do not describe the magic-sets rewriting. Please refer to [Ull89] for details. Instead, we use the following example to illustrate the bottom-up derivation of CSQs, the pruning of non-maximal CSQs, and the gains that we realize from the use of the magic-sets rewriting. In the next subsection we show issues pertaining to the derivation of facts containing vectors.

EXAMPLE 6.1 Let us consider query (Q3) and description (D4) from page 5. 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 underlying facts of the derived fact is empty. Note, 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 again (NT4.6) 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)` 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')`}. All other derivations of `answer` do not "survive" because they have smaller sets of underlying facts. Hence we output the CSQ (Q3). (Incidentally, it is the target query.)

Note, 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 [Ull89] we obtain the following equivalent description. We show only the rewriting of templates (NT4.4) and (NT4.6). The others are rewritten similarily.

```
(D49) (QT49.1) answer(F,L,D,O,M) :- emp(F,L,D,O,M), _Cond(F,L,D,O,M)
      (NT49.4) _Cond(F,L,D,Office,M) : mg_Cond(F,L,D,Office,M), substring(Office, $OS),
                                       _Cond(F,L,D,Office,M)
      (NT49.6) _Cond(F,L,D,O,M) : mg_Cond(F,L,D,O,M)
      (MS49.1) _Cond(F,L,D,O,M) : 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 similar to a top-down strategy and derives relevant facts. □

6.1 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 let the variable as is in the derived fact. The CSQ discovery module can not handle this situtation. For most descriptions, the magic-sets rewriting solves the problem. We demonstrate how and we formally define the set of these non-problematic descriptions.

For example, let us fire template (NT8.1) of (D8) on the base facts produced by query (Q3). Assume also that (NT8.2) 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 (D8).

```
(D50) (QT50.1) answer(_R) :- $Table(_A), _Cond(_A), _subset(_R, _A)
      (NT50.1) _Cond(_A) :  mg_Cond(_A), _in($Position,X,_A), substring(X,$S), _Cond(_A)
      (NT50.2) _Cond(_A) :  mg_Cond(_A)
      (MS50.1) _Cond(_A) :  $Table(_A)
```

Rule (MS50.1) ensures that mg_Cond([f,l,d,o,m] is the only mg_Cond fact derived. When rules (NT50.1) and (NT50.2) 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]. Magic-sets rewriting does not always ensure that derived facts have bound vectors. Below we describe a class of descriptions where magic-sets rewriting does ensure this condition. Fortunately, this class contains all reasonable descriptions. First, we state a few definitions.

Definition 6.1 (**Target predicate**) A predicate that appears in some subgoal of the target query. □

Definition 6.2 (**Target subgoal**) A subgoal that uses a target predicate. □

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

Definition 6.3 (**Grounded Subgoal in a Rule** R) A target subgoal is grounded. A nonterminal subgoal is grounded as defined by Definition 6.5. A metapredicate subgoal s is grounded if s can be evaluated using the bindings of those arguments that appear in grounded subgoals of R. □

Definition 6.4 (**Grounded Rule**) A rule is grounded if every vector variable in the rule appears in some grounded subgoal. The rule is said to *depend* on the predicates of the grounded subgoals. □

Definition 6.5 (**Grounded nonterminal**) A nonterminal **_N** is grounded if each rule defining **_N** is grounded. For its grounding, **_N** depends on a nonterminal **_M** if some rule defining **_N** depends on **_M**. □

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

Theorem 6.1 *If each nonterminal in a description D is grounded then a bottom-up evaluation of magic-sets rewritten D produces no fact that has vector variables.* □

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

Theorem 6.2 *Nonterminals of a valid program can be completely ordered such that nonterminal* **_N** *in position i depends for its groundings only on nonterminal in positions* $1 \ldots i-1$. □

The following algorithm derives CSQs given a target query and description.

Algorithm 4
Input: Target query Q and Description D
Output: A set of CSQs $s_1, i = 1, \ldots, n$ annotated with $\mathcal{C}_i$, $\mathcal{B}_i$,
and a bit specifying whether the CSQ is representative
Method:
Check if D is valid *% that is, every nonterminal is grounded (see definition 6.5*
Reorder each template R in D such that
All predicate subgoals occur in the front of the rule
A nonterminal **_N** appears after **_M** if **_N** depends on **_M** for grounding.
Metapredicates appear at the end of the rule
Do Magic-sets rewriting on D
Do Bottom-up evaluation on rewritten D

Appendix B describes how to evaluate a single rule bottom up in a magic-sets rewritten program.

6.2 Avoiding the Computation of Relevant Queries with Inappropriate Binding Patterns

Note, $\mathcal{B}_s$ keeps track of the bindings needed by a CSQ s(refer Section 4.3). Algorithm 6 does not use bindings information while deriving facts. Namely, the algorithm derives useless CSQs that need bindings not exported by any other CSQ. Algorithm 6 can be modified to generate only CSQs whose required bindings are produced by at least on other CSQ.

The optimized derivation 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 newly 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. The third iteration derives facts that need at least one binding provided by a fact derived in the iteration two. The following algorithm formalizes this intuition.

Algorithm 5

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 6
% 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)

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.

7 Evaluation

The CBR algorithm employs many techniques to eliminate sources of exponentiality that would otherwise arise in many practical cases. Table 1 lists these techniques along with an informal description of the exponential factor they eliminate and the section or the example that illustrates how the exponential factor might be generated. 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 4.4.

Technique	Exponential factor eliminated
Pruning represented CSQs	2^a, where a is the arity of the representative CSQ (see **Evaluation** paragraph of Section 3.1)
Pruning non-maximal CSQs	2^g, where g is the number of "optional" subgoals in the maximal CSQ (see Section 3.2)
Pruning CSQs with inappropriate bindings	2^n, where n is the number of variables that may be or may not be bound (see **Evaluation** paragraph of Section 6.2)
Magic Sets	n^m, where n is the arity of a predicate p and there are another m predicates that involve variables of p (see Example 6.1)

Table 1: Sources of Exponentiality Eliminated by CBR

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 4.2 and the discussion of Section 4.1.) On the other hand, the techniques for eliminating exponentiality described in Table 1 preserve completeness, in that we do not miss any plan through applying one of these techniques (see justifications in Sections 3.1, 3.2, and 6.2).

8 Related Work

Significant systems and theoretical results have been developed for the resolution of semantic and schematic discrepancies while integrating heterogeneous information sources. Most of the systems [S+, HM93, A+91, Gup89] do not address the problem of different and limited query capabilities of the underlying sources because they assume that the underlying sources are full-fledged databases that can answer any query over their schema.[6] Thus, the need for describing repositories with limited capabilities was not considered in this context.

The recent interest in the integration of arbitrary information sources – not only databases, but also file systems, the Web, legacy systems – invalidates the assumption that all underlying sources can answer any query over the data they export and makes necessary that we resolve the mismatch between the query capabilities provided by the underlying source(s). In this section we first compare the CBR approach to the general approach of some heterogeneous systems. Then we compare with related work the particular technical challenge addressed in building a CBR.

Approaches to Addressing Different and Limited Query Capabilities HERMES [S+] proposes a rule language for the specification of mediators where some literals specify explicitly the parameterized calls that are made to the sources. At run-time the parameters are instantianted by specific values and corresponding calls are made. Thus, HERMES guarantees that the queries sent to the wrappers are always supported. Unfortunately, this reduces the interface between wrappers and mediators to a very simple form (as specified by the particular parameterized calls), consequently not fully utilizing the sources' query power.

[6]The work in query decomposition in distributed databases has also assumed that all underlying systems are relational and equally capable to perform any SQL query.

DISCO [TRV95] assumes that each source may be able to do only a few operations. E.g., some source may be able to do selections and projections whereas another one may do joins and selections, and so on. Unlike RQDL, DISCO is unable to specify restrictions on how the operations are used. E.g., it cannot say "apply only one selection," or "the first argument of relation *emp* must be bound." However, DISCO considers additional problems like query optimization and generation when not all sources are available.

The Information Manifold [LRO] 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 than that by being able to express schema-independent descriptions and also express descriptions like "exactly one condition is allowed."

TSIMMIS suggests an explicit description of the wrapper's query capabilities [PGGMU], using the context-free grammar approach of the current paper. (Indeed, 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 olny select or project the results of a single CSQ.

This paper enhances the query capability description language of [PGGMU] to describe queries over arbitrary schemas, namely relations with unspecified arities and names. E.g., capabilities like "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 as described by given descriptions. Also, we provide algorithms for generating plans for Q by combining the results of these CSQs using selections, projections, and joins.

Related Specific Problem The CBR problem is related to the problem of determining how to answer a query using a set of materialized views [LY85, LMSS95, RSU95, Qia92]. 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 [RSU95]), or infinitely large queries/views. Also, they do not consider generating plans that require respecting evaluation orders (due to binding requirements).

[LMSS95] show that the problem of finding a rewriting of a conjunctive query is in general exponential in the total size of the query and the views. [Qia92] shows that if the query is acyclic we can find a rewriting in time polynomial to the total size of the query and views. [LMSS95, RSU95] 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).

9 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 Constraint-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.

The output of the CBR algorithm, in terms of the number of derived plans, remains a major source of exponentiality. Though the CBR prunes the output plans by deriving a plan only if no other plan pushes more selections, projections or joins to the source, it may still be the case that the number of plans is exponential in the number of subgoals and/or join variables. For example, consider the case where our query involves a chain of n joins and each one of them can be accomplished either by a left-to-right nested loops join, or a right-to-left nested loops join, or a local join. In the general case, CBR has to output 3^n plans where each of the plans employs one of the three join methods. Then, the mediator's cost-based optimizer must estimate the cost of each one of the plans and choose the most efficient. We could generate all of these plans or only some of them, depending on the time spent on optimization, and select one.

Currently, we are looking at implementing a CBR for IBM's Garlic system [C+94]. We are also investigating tighter couplings between the mediator's cost-based optimizer and the CBR. We envision an architecture in which the cost-based optimizer will guide the CBR in pruning the derived plans. We are considering techniques that will allow the CBR to summarize many "similar" plans into one, thus avoiding some of the exponential explosion.

We are working to extend RQDL's descriptive power to include unions, aggregations and negations, eventually covering the whole SQL language. Finally, we are investigating more powerful rewriting techniques that may replace a target query's subgoals with combinations of semantically equivalent subgoals that are supported by the wrapper.

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

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

[C+94] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[FK93] J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.

[Gup89] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

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

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

[LRO] A. Levy, A. Rajaraman, and J. Ordille. Query processing in the information manifold.

[LY85] P.A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB Conf.*, pages 259–69, 1985.

[PGGMU] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for the rapid implementation of wrappers. To appear in DOOD95. Available via ftp at `db.stanford.edu` file `/pub/papakonstantinou/1995/querytran.ps`.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[Qia92] Xiaolei Qian. Query folding. Technical Report CSL-95-09, SRI, 1992.

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

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

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

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems.* Computer Science Press, New York, NY, 1988.

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

Appendix

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

Figure 3: Normal-form RQDL syntax

A 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 6 we restrict to descriptions where all nonterminals are grounded (see Definition 6.5).

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 A.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 A.2 formally defines the application of a unifier on an RQDL expression.

□

Definition A.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 A.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 A.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.

B Applying a Template

Algorithm 6

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
do not add n to set of derived facts.