# A Query Translation Scheme for Rapid Implementation of Wrappers*

Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, Jeffrey Ullman

*Computer Science Department*
*Stanford University*
*Stanford, CA 94305-2140, USA*
*{yannis,agupta,hector,ullman}@cs.stanford.edu*

**Abstract**

Wrappers provide access to heterogeneous information sources by converting application queries into source specific queries or commands. In this paper we present a *wrapper implementation toolkit* that facilitates rapid development of wrappers. We focus on the query translation component of the toolkit, called the *converter*. The converter takes as input a *Query Description and Translation Language (QDTL)* description of the queries that can be processed by the underlying source. Based on this description the converter decides if an application query is (a) directly supported, i.e., it can be translated to a query of the underlying system following instructions in the QDTL description; (b) logically supported, i.e., logically equivalent to a directly supported query; (c) indirectly supported, i.e., it can be computed by applying a *filter*, automatically generated by the converter, to the result of a directly supported query.

## 1 Introduction

A *wrapper* or *translator* [C+94, PGMW95] is a software component that converts data and queries from one model to another. Typically, wrappers are used to provide access to heterogeneous information sources, as illustrated in Figure 1.a. In this case, an application (which could be a mediator [Wie92]), issues queries in a single, common query language like SQL. The wrapper for each source converts the query into one or more commands or queries understandable by the underlying source. The wrapper receives the results from the source, and converts them into a format understood by the application.

As part of the TSIMMIS project [PGMW95, GM+] we have developed hard-coded wrappers for a variety of sources, including legacy systems. We have observed, like everyone who has built a wrapper, that writing them involves a lot of effort [A+91, C+94, EH86, FK93, Gup89, LMR90, MY89, T+90]. However, we have also observed that only a relatively small part of the code deals with the specific access details of the source. A lot of code, on the other hand, is either common among wrappers (deals with buffering, communications to the application, and so on) or implements query and data transformations that could be expressed in a high level, declarative fashion.

Based on these observations we have developed a *wrapper implementation toolkit* for rapidly building wrappers. The toolkit contains a library of commonly used functions, such as for receiving

*This work was supported by ARPA Contract F33615-93-1-1339, by NSF IRI 92-23405, by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation.  The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government.

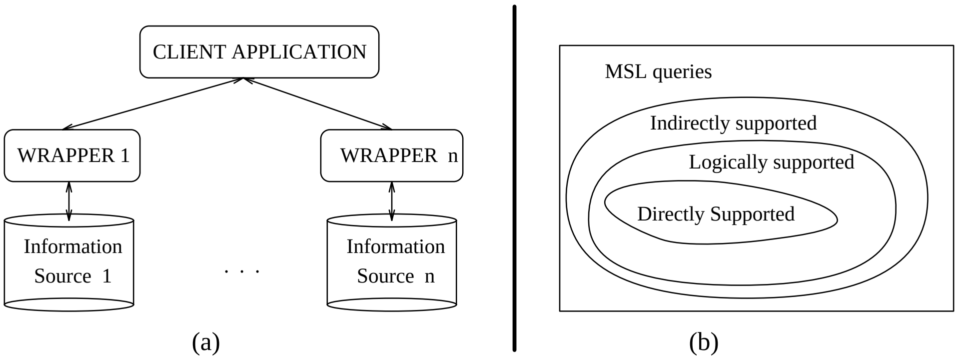


Figure 1: (a) Accessing information through wrappers (b) Supported queries.

queries from the application and packaging results. It also contains a facility for translating queries into source-specific commands and queries, and for translating results into a model useful to the application.

In this paper we focus on the query translation component of the toolkit, which we refer to as the *converter*. (In Section 6 we will describe the other toolkit components and how the converter is integrated with them.) The implementor gives the converter a set of templates that describe the queries accepted by the wrapper. If an application query matches a template, an implementor-provided action associated with the template is executed to produce the *native query* for the underlying source. Note, a native query is not necessarily a string of a well-structured query language (e.g. SQL). In general, the term "native query" may refer to any program used to access and retrieve information from the underlying source.

**EXAMPLE 1.1** To illustrate, consider an application that issues SQL queries. One of the sources it accesses has limited functionality, as is true for many sources encountered in a heterogeneous environment. For this illustrative example, assume that the source can only do selection on attribute `dept` of some table, followed by a projection. This ability may be specified as the following template.

```
select $X.$Y from $X where $X.dept=$Z
```

The symbols `$X`, `$Y`, and `$Z` represent *placeholders* that have to be bound to specific constants to produce a valid SQL query. Assume that the following query arrives at the wrapper and is given to the converter:

```
select emp.name from emp where emp.dept='toy'
```

This query matches the template with the bindings `$X = "emp"`, `$Y = "name"`, and `$Z = "'toy'"`. Given the match, the actions associated with the template would then generate the necessary native query to do the actual search on the source. For example, if the underlying source was a file system the actions could produce a "grep" command to search for the string `$Z` in say columns 10-20 of file `$X`. Out of the matching lines, it would return the characters between the string `$Y` and some termination character. □

Example 1.1 illustrates a very simple template matching facility that could be easily implemented using Yacc-like tools [LMB92]. However, since the matching facility is based entirely on string matching, it does not exploit the semantics of the common query language. The following examples show that if converters "understand" queries they are translating, then they can successfully handle many more queries.

**EXAMPLE 1.2** Consider the following query template:

```
select $X.$Y from $X where $X.sal=$Z1 and $X.dept=$Z2
```

Syntactically, only queries where the `$X.sal` and `$X.dept` appear in exactly the specified order match this template. The query

```
select emp.name from emp where emp.dept='toy' and emp.sal=100
```

would not match the template. If we wanted to process this type of query we would have to define a second template. In general, we would have to consider an exponential number of orderings of the terms in the `where` clause. It is not practical to have all these templates, especially since all of them would have almost identical actions associated with them. □

**EXAMPLE 1.3** Consider a data source that can only do selections on attribute `dept` and does not understand the notion of projecting out attributes. Such a source can be described with the following template:

```
select * from $X where $X.dept=$Z
```

The following query does not match this template because it includes a projection:

```
select emp.name from emp where emp.dept='toy'
```

However, the wrapper could process the above query by transforming it into one without a projection and then doing the projection on the returned answers. This approach would allow the wrapper to leverage its own capability to handle a much wider class of queries than those specified by the template.

As we will see, our wrapper toolkit can handle this type of query transformation. When the converter is given a query, it generates not only commands for the underlying source, but also a *filter* describing additional processing on the results, if any is required. In our example, the filter would specify a projection over the `name` attribute. □

In example 1.2 the converter must understand the notion of selection and conjunctive logical expressions. In example 1.3 the converter must understand projections and the fact that a projection over `emp.name` can be obtained a-posteriori from a projection over `*`. While this knowledge gives the converter the ability to handle more queries, it does mean that the converter must be targeted to a particular incoming query language. Being language specific does not pose a problem for converters because our goal is to develop many wrappers for a given common query language, so it is to our advantage to exploit the features of the common query language. Furthermore, most declarative query languages are based on common principles, so our converter should be easy to modify to other query languages.

Our converters are targeted for the MSL query language [PGMU]. (SQL was only used in our initial examples to motivate our ideas.) MSL is a logic-based language for a simple object-oriented data model, called OEM [PGMW95]. We believe that both OEM and MSL are well suited for integration of heterogeneous information sources. The converter is configured with templates written in the *Query Description and Translation Language* (QDTL). Each template is associated with an action that generates the commands for the underlying source.

Once configured, the converter takes as input an MSL query, and generates commands for the source and a *filter* to be applied to the results. (Actually, in our current design, the converter accepts only a subset of MSL; see Section 2.) The converter will process:

- *Directly supported queries.* These are queries that syntactically match a template.

- *Logically supported queries.* These are queries that produce the same results as a directly supported query. We use the notion of *logical equivalence* to detect queries that fall in this class.
- *Indirectly supported queries.* These are queries that can be executed in two steps: first a directly supported query is executed, and then a filter is applied to the results of the first step. We have appropriately extended the notion of *subsumption* in order to detect the queries that fall in this class.

Figure 1.b graphically shows the types of accepted queries. Although QDTL descriptions syntactically look like Yacc grammars – suitably modified for the description of queries, rather than arbitrary strings – our converter handles a much larger class of queries than the class of directly supported queries that is handled by Yacc. For example, our converter understands the commutativity of a logical conjunction, while Yacc would expect the terms to appear in a specific order. Furthermore, our converter introduces the following innovations:

- A designer can succinctly and clearly define the functionality of each source through a few QDTL templates. Note, a QDTL description is more than a list of "parameterized queries" since it allows the description and translation of infinite sets of queries (See Section 5.)
- The converter, in cooperation with the filter processor, automatically extends the query capabilities of sources that have limited functionality. Note that unlike relational and object oriented databases, where typically all possible queries over the schema are allowed, arbitrary information sources, e.g., legacy systems, permit only limited sets of queries. The automatic extension of query abilities allows us to bring to the same level of functionality different sources and then more easily integrate them.
- The converter, together with the other functions of the toolkit, make it possible to rapidly implement wrappers.

One important thing to notice is that the capabilities of wrappers can be "gracefully extended." That is, one can quickly design a simple wrapper with a few templates that cover some of the desired functionality, probably the one that is most urgently needed. Then templates can be added as more functionality is required.

We start our paper with a brief description of the OEM model and the MSL query language. Then in Section 3 we give a detailed example that shows how QDTL is used. Indirectly supported queries and the notion of query subsumption are further discussed in Section 4, while Section 5 introduces additional powerful QDTL features such as nonterminal templates and metapredicates. In Section 6 we discuss the architecture of wrappers and the wrapper toolkit; we also discuss how the converter is used by the wrapper toolkit to rapidly implement wrappers. Section 7 focuses on the query translation algorithm at the heart of the converter. This is the algorithm that maps input queries to templates and generates filters. The section gives an example-driven description of the algorithm, and the full details can be found in the Appendix A. Finally, Section 8 discusses related work, and Section 9 presents some conclusions and future work. A proof of the correctness of the algorithm can be found at [P$^+$].

## 2 The OEM Model and the MSL Language

When integrating heterogeneous information sources one often faces unstructured information whose form may change dynamically. Many applications that have to deal with such information use some type of *self-describing* data model where each data item has an associated descriptive label. Applications include tagged file systems [Wie87], Lotus Notes [Mar93], the Teknekron Information Bus [O+93], LOOM frames [MY89], electronic mail, RFC1532 bibliographic records, and many more. For this reason we have selected a self-describing model, in particular the Object Exchange Model (OEM) [PGMW95], as the common data model exported by our wrappers. OEM captures the essential features of models used in practice, generalizing them to allow arbitrary nesting and to include object identity. OEM does not directly support classes, methods, and inheritance; however, classes and methods can be emulated [PGMW95].

To illustrate OEM, consider the following objects (one object per line):

```
<ob1, person, {sub1,sub2,sub3,sub4,sub5}>
      <sub1, last_name, 'Smith'>
      <sub2, first_name, 'John'>
      <sub3, role, 'faculty'>
      <sub4, department, 'CS'>
      <sub5, telephone, 415-5141292>
```

Each OEM object consists of an *object-id* (e.g., `sub4`), a *label* that explains its meaning (e.g., `department`), and a *value* (e.g., `'CS'`). Object-id's can be of different types, but for this paper we may think of them as terms that are used to link objects to their subobjects. Labels are strings that are meaningful to the application or the end user. A value can be a scalar such as an integer or a string, or it can be a set of (sub)objects (e.g., the value of the "`person`" object).

At each source, some OEM objects are defined to be *top-level* or root objects. (Of course, the source itself probably does not store OEM objects; this is only the "illusion" created by the wrapper above that source.) Top-level objects provide "entry points" into the object structure from which subobjects can be requested, as explained below.

An application can request OEM objects from a wrapper using the MSL query language [PGMU]. In this paper we will use only a subset of MSL. In particular, we will consider only conjunctive queries that extract a single object together with all its descendants – i.e., direct or indirect subobjects. (In Section 9 we discuss why we make these restrictions.)

To illustrate, consider the following query that searches for top-level `person` objects (i.e., objects with `person` label) containing a `last_name` subobject with value `'Smith'`. The matching objects, together with their `last_name`, `first_name`, ... subobjects, are then retrieved.

```
(Q1) *P :- <P person {<L last_name 'Smith'>}>
```

The query consists of a single *head* and a single *tail* separated by the `:-` symbol. *Variables* are represented by identifiers starting with a capital letter, such as `P` and `L`. The tail describes the search pattern, while the head is the object-id of the objects that will be retrieved.[1] Intuitively, we match the tail pattern against the object structure exported by the wrapper, thereby binding the variables to object components of the wrapper's object structure. The result consists of all the objects (and their descendants) whose object-ids get bound to the variable that appears in the head.

Now we give more details about the matching process. Tails are based on patterns of the form <*object-id label value*>, where each field may be a constant or a variable. When a field (object-id, label, or value) contains a constant then the pattern binds successfully only with OEM objects that have the same constant in the corresponding field. On the other hand, when the field contains a variable the pattern can successfully bind with any OEM object (modulo the restrictions imposed

[1] The `*` in the head of the query indicates that subobjects are retrieved too. Without the asterisk, a single object is retrieved.

by the other fields in the pattern) and the variable binds to the contents of the corresponding field. If a variable `X` appears multiple times in a tail, all occurrences of `X` must bind to the same contents for the tail to successfully bind to an OEM object.

If a pattern $A$ contains a value that has curly braces and more patterns $B, C, ...$ inside, then pattern $A$ binds to OEM objects with a set value. The objects that bind to pattern $A$ have one or more subobjects, some of which bind to the patterns $B, C, ...$. For example, query Q1 requires that `person` objects have a `last_name` subobject with value `'Smith'`. Note that we allow the `person` objects to have subobjects other than `last_name` as well.

For notational convenience we remove object-id variables from object patterns when the object-id is not useful, i.e. when it appears exactly once in the query. For instance, in query Q1, variable `L` is not used in the head nor in other parts of the tail. Therefore we can replace the pattern `<L last_name 'Smith'>` in Q1 by `<last_name 'Smith'>` without affecting the query. Thus, notationally a pattern with two fields represents a three field pattern with a unique but unspecified variable in the first field.

## 3 A Detailed Example of Query Translation

We illustrate the use of our converter and QDTL using the following simple example. Say we wish to build a wrapper for a university "lookup" facility that contains information about employees and students. (This example is motivated by an actual service offered by our department at Stanford). The lookup facility is accessed from the command line of computers and offers limited query capabilities. In particular, it can return only the full records of persons, including all fields such as "last name", "first name", and "telephone." There is no way for the user to retrieve only one field, e.g., the telephone number, for a person. Furthermore, the only queries that are accepted by the lookup facility are:

1. Retrieve person records by specifying the last name, e.g.,

   (L2) `lookup -ln Smith`

2. Retrieve person records by specifying the first and the last name, e.g.,

   (L3) `lookup -ln Smith -fn John`

3. Retrieve all person records by issuing the command

   (L4) `lookup`

The queries accepted by the lookup facility can be easily described in our Query Description and Translation Language (QDTL). As discussed in Section 1, a QDTL description consists of a set of templates with associated actions. Below we state description D1 that consists of three *query templates* QT1.1, QT1.2, and QT1.3. For simplicity, we do not yet state the associated actions.

(D1) (QT1.1) `Query ::= *O :- <O person {<last_name $LN>}>`
(QT1.2) `Query ::= *O :- <O person {<last_name $LN> <first_name $FN>}>`
(QT1.3) `Query ::= *O :- <O person V>`

Each query template appears following the `::=` and is a "parameterized query." The identifiers preceded by `$`, such as `$LN` and `$FN`, are *constant placeholders* representing expected constants in the input query. Upper case identifiers, such as `O`, are *variable placeholders* denoting variables that are expected at that point in the input query. Note, the variable appearing in the query does not have to have the same name as the template variable.

Each template describes many more queries than those that match it syntactically. More specifically, each template describes the following classes of queries:

- *Directly supported queries.* A query $q$ is directly supported by a template $t$ if $q$ can be derived by substituting the constant placeholders of $t$ by constants and the variables of $t$ by variables. For example, query Q1 is directly supported by template QT1.1 by substituting `O` with `P` and `$LN` with `'Smith'`.

- *Logically supported queries.* A query $q$ is logically supported by template $t$ if $q$ is logically equivalent to some query $q'$ directly supported by $t$. Two queries $q$ and $q'$ are equivalent if they produce the same result regardless of the contents of the queried source. For example, the following queries are logically supported by template QT1.2 although they are not directly supported:

```
*O :- <O person {<first_name 'John'> <last_name 'Smith'>}>
*O :- <O person {<last_name 'Smith'>}> AND <O person {<first_name 'John>}>
*O :- <O person {<LO last_name 'Smith'>}>
      AND <O person {<LO L V> <first_name 'John'>}>
```

  All these queries are equivalent to the following query Q5, that is directly supported by the template QT1.2:

  (Q5) `*O :- <O person {<last_name 'Smith'> <first_name 'John'>}>`

- *Indirectly supported queries.* A query $q$ is indirectly supported by a template $t$ if $q$ can be "broken down" into a directly supported query $q'$ and a filter that is applied on the results of $q'$. We give a definition of indirect support in Section 4; for now we present an example. Consider the following query:

  (Q6) `*Q :- <Q person {<last_name 'Smith'> <role 'student'>}>`

  This query is not logically supported by any of the templates of description D1. However, our converter realizes that this query is *subsumed* by the directly supported query

  (Q7) `*Q :- <Q person {<last_name 'Smith'>}>`

  This means that the answer to Q7 contains all the information that is necessary for answering Q6. Thus, the converter matches Q6 to template QT1.1 as if it were Q7, binding `$LN` to `'Smith'` and `O` to `Q`. In addition, the converter generates the filter:

  `*O :- <O person {<role 'student'>}>`

  The filter is an MSL query that is applied to the result of query Q7 to produce the result of query Q6.

Note, we often say "the description $d$ supports directly, logically, or indirectly the query $q$" meaning that a template $t$ of $d$ supports directly, logically, or indirectly the query $q$.

### 3.1 Formulation of the Native Query

QDTL templates are accompanied by actions that formulate the native queries for the source. For our converter, the actions are written in C, although we could have selected any other language. Let us extend description D1 with actions that formulate native queries such as L2, L3, and L4.

```
(D2) (QT2.1) Query ::= *O:- <O person {<last_name $LN>}>
(AC2.1)                  { sprintf(lookup_query, 'lookup -ln %s', $LN) ;}
(QT2.2) Query ::= *O :- <O person {<last_name $LN> <first_name $FN>}>
(AC2.2)                  { sprintf(lookup_query, 'lookup -ln %s -fn %s', $LN, $FN) ; }
(QT2.3) Query ::= *O :- <O person V>
(AC2.3)                  { sprintf(lookup_query, 'lookup') ; }
```

To illustrate, consider again the input query Q5:

```
*O :- <O person {<last_name 'Smith'> <first_name 'John'>}>
```

This query matches template QT2.2. by binding placeholder `$LN` to `'Smith'` and `$FN` to `'John'`. Then, the action AC2.2 that consists of the C function

```
sprintf(lookup_query, 'lookup -ln %s -fn %s', $LN, $FN)
```

is executed. In this action, `$LN` and `$FN` behave as C variables that at execution time contain the values `'Smith'` and `'John'` respectively. The effect of this action is to write the string `'lookup -ln Smith -fn John'` in the variable `lookup_query`.

This completes the job of the converter on this query. Then, the implementor-provided part of the wrapper takes over, submits the string `lookup_query` to the source and waits for an answer.

## 4 Query Subsumption

In Section 3 we said that query Q6 was subsumed by Q7 because the former had an additional condition on the "role" subobject. Thus query Q6 selects a subset of the objects obtained by the subsuming query Q7.

A different type of subsumption, specific to object oriented data, occurs when the subsumed query extracts subobjects obtained by the subsuming query. For example, consider the following query Q8 that retrieves the `first_name` subobjects of `person` objects with last name `'Smith'`

```
(Q8) *F :- <O person {<F first_name X> <last_name 'Smith'>}>
```

Query Q8 is subsumed by the following query Q9, that retrieves the full `person` objects of persons with last name `'Smith'` and an unspecified first name.

```
(Q9) *O :- <O person {<F first_name X> <last_name 'Smith'>}>
```

Notice that Q8 and Q9 have exactly the same conditions. However, Q9 subsumes Q8 because the `person` objects retrieved by the latter *contain* the `first_name` objects required by the former. The following definitions formalize the notions we have illustrated.

**Definition 4.1** (**Object containment**) Object $O$ is contained in another object $O'$ if and only if

- Either $O$ and $O'$ are identical, i.e., they have identical object-id, label, and value; or
- $O$ is a subobject (direct or indirect) of $O'$.

$\square$

**Definition 4.2** (**Query subsumption**) A query $q$ is subsumed by another query $q'$ if each answer object for $q$ is contained in some answer object of $q'$.[2] $\square$

**Definition 4.3** (**Indirect support**) A query $q$ is indirectly supported by a query $q'$ if

1. $q'$ subsumes $q$, and
2. there is a filter query $f$ that when applied on the result of $q'$ produces the result of $q$.

A filter query is formally defined by Definition A.1 in Appendix A. We will say that a template $t$ indirectly supports a query $q$ if $t$ directly supports a query $q'$ that indirectly supports $q$. $\square$

Note, query subsumption does not necessarily imply indirect support. For example, consider the following query

(Q10) *F :- <person {<F first_name X>}>

that subsumes Q8, since it retrieves all first-name objects. However, Q10 does not indirectly support Q8, since given a first-name object in the result of Q10, we can not tell whether it is a subobject of a person with last_name 'Smith'.

## 4.1 Maximal Supporting Queries

Notice that given a query $q$ there may be more than one queries that support $q$, and these queries may not be logically equivalent. For example, query Q6 on page 7 is supported by query Q7 and also by the query

(Q11) *O :- <O person V>

that retrieves all person objects.

Note, query Q11 also subsumes query Q7. Thus, Q7 derives fewer unnecessary answers than Q11. From a performance point of view it is better for the wrapper to send Q7 to the source (after the necessary transformation to a native query) rather than Q11, because the former contains more conditions of the original query Q6. Indeed, for our example, query Q7 is the best query directly supported by description D1 that supports query Q6 because Q7 pushes to the source as many conditions as possible. We will say that Q7 is a *maximal supporting query* for Q6.

**Definition 4.4** (**Maximal supporting query**) A query $q_s$ is a maximal supporting query of query $q$ with respect to description $d$, if

- $q_s$ is directly supported by $d$,
- $q_s$ indirectly supports $q$, and
- there is no directly supported query $q'_s$ that indirectly supports $q$, is subsumed by $q_s$, and is not logically equivalent to $q_s$.

$\square$

[2] Note, more general forms of query subsumption may be defined.

Note, there may be more than one maximal supporting query for a given query. For example, assume that a source allows us to place a condition on exactly one subobject of the `person` objects. This source is specified by the QDTL description (actions not shown):

```
(D3) (QT3.1) Query ::= *O :- <O person {<$L $V>}>
```

For this source, consider input query Q5. This query has two maximal supporting queries:

```
(Q12) *O :- <O person {<last_name 'Smith'>}>
(Q13) *O :- <O person {<first_name 'John'>}>
```

Our converter actually considers all possible maximal supporting queries by considering different ways in which the input query can match the templates of a description. Choosing the optimal maximal subsuming query (when there is more than one) requires knowledge of the contents, semantics, and statistics of the database; our initial implementation does no optimization and simply selects one of the maximal supporting queries. Then, the converter executes the actions associated with that particular maximal query. We give additional details in Section 6.

## 5 Nonterminals and Other QDTL Features

QDTL allows the use of *nonterminals* to construct grammars that describe more complex sets of supported queries. To illustrate, say that our lookup facility lets us place selection conditions on zero or more of the fields of its records. That is, we can issue commands such as `'lookup -fn John'`, `'lookup -fn John -role faculty'`, `'lookup -role student'`, and so on. Explicitly listing all possible combinations of conditions in our templates would be impractical. (If there are 10 lookup fields, there would be $2^{10}$ templates.)

With nonterminals, this functionality can be described succinctly. For instance, assuming only three fields, `first_name`, `last_name`, and `role`, we can use the following description (without actions for now):

```
(D4) /* A description with nonterminals */
(QT4.1) Query ::= *OP :- <OP person { _OptLN _OptFN _OptRole}>  /*Query Template*/
(NT4.2) _OptLN ::= <last_name $LN>                            /*Nonterminal template*/
(NT4.3) _OptLn ::=                                     /* empty nonterminal template*/
(NT4.4) _OptFN ::= <first_name $FN>
(NT4.5) _OptFN ::=                                                        /* empty */
(NT4.6) _OptRole ::= <role $R>
(NT4.7) _OptRole ::=                                                      /* empty */
```

Nonterminals are represented by identifiers that start with an underscore (_). Every nonterminal has a *definition* that consists of a set of *nonterminal templates*. For example nonterminal `_OptRole` is defined by nonterminal templates NT4.6 and NT4.7.

A query $q$ is directly supported by a query template $t$ that contains nonterminals if $q$ is directly supported by one of the *expansions* of $t$. An expansion of $t$ is obtained by replacing each nonterminal $n$ of the query template $t$ with one of the nonterminal templates that define $n$. For example, the query

```
(Q14) *O :- <O person {<last_name 'Smith'> <role 'professor'>}>
```

is directly supported by template QT4.1 because Q14 matches with the expansion

```
(E15) *OP :- <OP person {<last_name $LN> <role $R>}>
```

This expansion is derived from query template QT4.1 by replacing the nonterminal `_OptLN` with the nonterminal template NT4.2, the nonterminal `_OptFN` with the nonterminal template NT4.5, and the nonterminal `_OptRole` with the nonterminal template NT4.6.

### 5.1 Actions and Attributes Associated with Nonterminals

Nonterminal templates have associated actions, just like query templates. When a query successfully matches with a template, the action for the nonterminal template used during the matching is executed. In addition, every nonterminal $n$ is associated with an *attribute* that is accessible from the templates that use $n$ and the templates that define $n$. These attributes are similar to the attributes that Yacc (in general context-free grammar parsers) associate with nonterminals, and are used to generate the native query of the underlying source.

Description D4 can be augmented with code to generate the required lookup native query as follows. Note that in the C code, a nonterminal attribute is represented by `$` followed by the name of the nonterminal.

```
(D5) (QT5.1) Query ::= *OP :- <OP person { _OptLN _OptFN _OptRole}>
(AC5.1)               { sprintf(lookup_query, 'lookup %s %s %s', $_OptLN,
                                   $_OptFN, $_OptRole)} ;
(NT5.2) _OptLN ::= <last_name $LN>
(AC5.2)              { sprintf($_OptLN, '-ln %s', $LN) ; }
(NT5.3) _OptLN ::=
(AC5.3)              { $_OptLN = '' ; }
(NT5.4) _OptFN ::= <first_name $FN>
(AC5.4)              { sprintf($_OptFN, '-fn %s', $FN) ; }
(NT5.5) _OptFN ::=
(AC5.5)              { $_OptFN = ''; }
(NT5.6) _OptRole ::= <role $R>
(AC5.6)               { sprintf($_OptRole, '-role %s',$R) ; }
(NT5.7) _OptRole ::=
(AC5.7)              { $_OptRole = '' ; }
```

As discussed earlier, query Q14 is directly supported by description D5. When nonterminal `_OptLN` matches the `<last_name 'Smith'>` clause in the query, its associated code is executed, storing the string `'-ln Smith'` in `$_OptLN`. Similarly, `'-role professor'` is stored in `$_OptRole`. When the query matches template QT5.1, variable `lookup_query` is assigned the string `'lookup -ln Smith -role professor'`, which is sent to the lookup facility.

### 5.2 Recursion

Nonterminal templates may recursively contain nonterminals. This flexibility allows us to describe infinite sets of expansions. The following description – that describes queries with an arbitrary number of conditions on the `person` subobjects – illustrates recursion

```
(D6) /* This query description involves recursion */
(QT6.1) Query ::= *OP :- <OP person { _Cond }>
(NT6.2) _Cond ::= <$Label $Value> _Cond
(NT6.3) _Cond ::=
```

The query template above directly supports query Q14. To see this we first expand `_Cond` with the nonterminal template NT6.2, yielding

```
(E7) Query ::= *OP :- <OP person { <$Label $Value> _Cond }>
```

Expanding `_Cond` again we obtain:

```
(E8) Query ::= *OP :- <OP person { <$Label $Value> <$Label1 $Value1> _Cond }>
```

Note that in the second expansion we replaced the placeholder names with new names `$Label1` and `$Value1`. This policy is essential to avoid confusion with names from other expansions. Finally, we expand `_Cond` with the nonterminal template NT6.3 (i.e., the "empty" template) to produce an expansion that directly matches query Q14.

In some cases we may want to force placeholder names obtained by expanding nonterminals to be the same as existing placeholder names in the query template. By using parameters as arguments of QDTL nonterminals we can force different templates to refer to the same variable or placeholder (refer to [P+] for details).

### 5.3 Metapredicates

Descriptions D4 and description D6 accept similar queries, with the exception that D6 accepts any subobject label. For example, D6 will accept the query

```
*P :- <P person {<M fuel 'gasoline'>}>
```

(and an action, not shown in description D6, may translate it into the string `'lookup -fuel gasoline'`) while D4 will not.

We can force D6 to check for particular labels (and effectively schemas) by using *metapredicates*. This capability gives us the same functionality as D4 with a more compact specification. To illustrate, consider the following modification of the description D6:

```
(QT9.1) Query ::= *OP :- <OP person { _Cond }>
(NT9.2) _Cond ::= <$Label $Value> _Cond personsub($Label)
(NT9.3) _Cond ::=
```

The metapredicate `personsub($Label)` checks whether the constant that matches `$Label` is a valid label for some subobject of `person`. The metapredicate `personsub()` is implemented by a C function of the same name. The wrapper implementor provides this function together with description D9.

The converter treats metapredicates simply as additional conditions that must hold for a query to match a template. In our example, after we expand query template QT9.1 with the nonterminal template NT9.2 and then with the nonterminal template NT9.3 we get:

```
*OP :- <OP person {<$Label $Value> personsub($Label)}>
```

Matching this expansion with query Q1 requires that we bind `$Label` to `'last_name'` and `$Value` to `'Smith'`. This binding implies that `personsub('last_name')` must hold. The C function `personsub` is thus invoked, and if it answers "yes" the expansion matches the query.

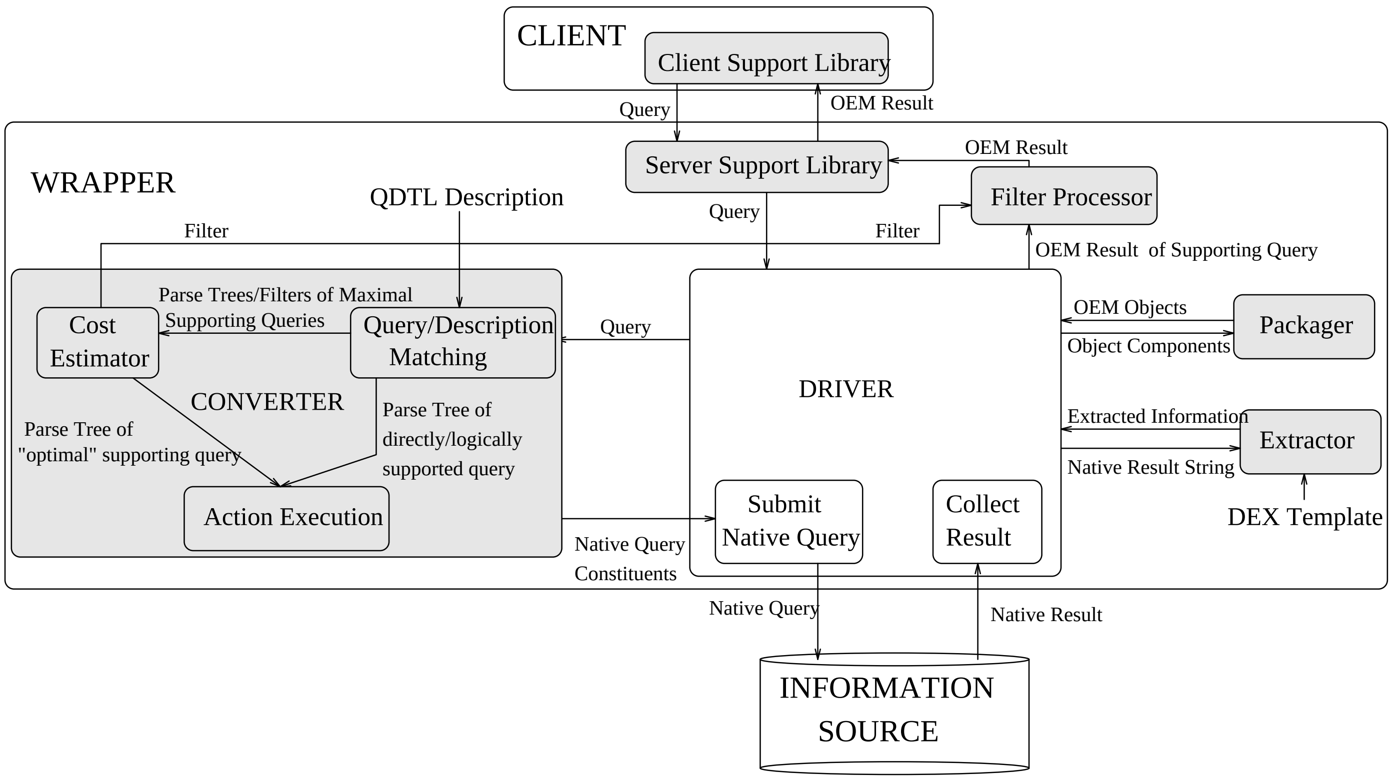


Figure 2: The Architecture of a Wrapper

# 6 Wrapper Architecture

Figure 2 shows the architecture of the wrappers generated with our toolkit. The shaded boxes represent components provided in the toolkit; the wrapper implementor provides the *driver* that has the primary control of query processing and invokes various services of the toolkit – as is shown in Figure 2. The implementor also provides the QDTL description for the converter, as well as the *Data EXtraction (DEX)* template for the *extractor component* of the toolkit.

Our wrappers behave as servers in a client-server architecture, where the clients are mediators or generic client application programs. Clients use the *client support library* to issue queries and receive OEM results (see Figure 2). The *server support library* component of the toolkit receives queries from the client and dispatches the driver for query processing. The driver invokes the converter, which finds a query that supports the input query and returns the *native query constituents*. The latter are values assigned to variables of the driver that are used to construct the native query. For example, variable `lookup_string` of description D2 contains the only native query constituent for the "lookup" wrapper.

The driver then submits the native query to the underlying information source and receives the result from the source. The driver uses the *extractor* to extract information from the received result and then uses the *packager* to pack the result components into OEM objects. Finally, if during the query/description matching a filter was produced, the driver passes the OEM result and the filter to the *filter processor*.

Subsection 6.1 discusses the converter architecture in more detail. Then, Subsection 6.2 discusses the extractor, while Subsection 6.3 discusses the filter processor.

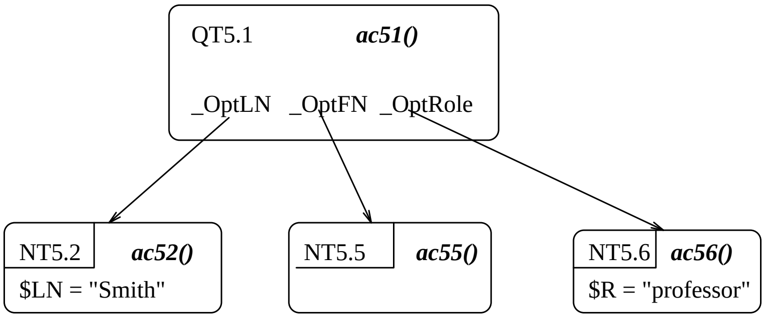


Figure 3: The parse tree

## 6.1 Converter Architecture

To illustrate, let us assume that the converter is given description D5 that directly supports query Q14 (see Section 5). The *query/description matching* component of the converter produces the parse tree of Figure 3 that contains all the information about the expansions and substitutions obtained while matching the query and the description. The parse tree is used by the *action execution* component of the converter to execute the actions that generate the native query constituents. Note, the converter – unlike the Yacc processor – performs the query/description matching and the action execution in two separate phases because there may be more than one maximal supporting queries, and consequently more than one parse trees. The converter executes actions only after it selects one of the parse trees.

The nodes of the parse tree correspond to the templates that were used for the matching. For readability, in Figure 3 we have named (top left corner) the nodes of the tree using the labels of the corresponding templates in description D5. Also, every node contains a pointer to a C function, such as `ac52()`, `ac55()`, etc, containing the code for the corresponding action. The root node of the parse tree corresponds to query template QT5.1 that matched with the query and points to nodes corresponding to the nonterminal templates – NT5.2, NT5.5, and NT5.6 – that were used. Every node contains a list of the constant placeholders that appear in the template, along with the matching constants.

If there are multiple maximal supporting queries, the query/description matching component passes all the corresponding parse trees to the cost estimator that chooses one of the parse trees either by an *arbitrary choice* or by *cost-based* selection. The later technique assumes that the wrapper has access to cost estimates of the functions provided by the underlying sources, catalog estimates, and so on. In our current implementation, our cost estimator does not perform cost optimization and selects the first parse tree. However, we believe it is important to have the cost optimizer framework in place initially so that optimization may be added later. Once a parse tree is selected, the *action executor* does a postorder traversal of the parse tree and invokes the corresponding action functions. The actions have access to the list of [constant placeholder, matching constant] pairs.

## 6.2 Information Extraction

Often, legacy systems return data as semi-structured strings. In these cases, the Data Extractor (DEX) can be used to parse the result and identify the required data. DEX is configured with a description of the source's output and information regarding which parts should be extracted. We use a brief example to illustrate how DEX works. Suppose that our sample "lookup" facility returns results as a sequence of text lines, of the form:

```
Record 1
  Last Name: Smith
  First Name: John
  Role: Student
Record 2
  Last Name: ...
```

The goal of the extractor is to extract the `last-name`, `first-name`, and `role` fields of the "lookup" result. This is achieved by giving the following DEX template to the extractor.

```
MATCH STRING (lookup_result)
{ records_number = 0 ;}
( Record # \n
  Last Name\:\ $$(lookup_array[records_number].last_name) \n
  First Name\:\ $$(lookup_array[records_number].first_name) \n
  Role\:\ $$(lookup_array[records_number].role) \n
  { records_number++ ; }
)*
```

Note, inside the `$$(...)` structures appear the names of C variables of the driver. For our running example we may assume that the following data structure has been declared in the driver:

```
struct lookup_type { char[40] last_name ;
                     char[20] first_name ;
                     char[30] role ; } lookuparray[200] ;
```

The above pattern specifies the expected syntax of the string `lookup_result` (that contains the result of lookup), specifies which parts of the output string will be extracted, and in which variables of the driver they will be placed. Our extractor can be viewed as a modification of the Yacc and Lex tools for the more specific problem of information extraction.

### 6.3 Result Creation and Filter Processing

After the extractor gathers the information in the appropriate data structures of the driver, and the packager constructs the OEM result objects, the *filter processor* applies the filter on the OEM result objects. The filter is produced by the converter while matching the input MSL query with the QDTL description. The filter is an MSL query and is applied to the output of the packager in a 2-step process by the filter processor: First the filter processor creates an algebraic description of the MSL query and then it executes the algebraic description. The algebraic operations can "find the subobjects of an object," "compare the object-id/label/value of an object to a constant," and so on.

## 7 The Query Translation Algorithm

Answering whether a MSL query $q$ is supported by a QDTL description $d$ is a hard problem. Often we need to reason with descriptions that support infinitely many queries (for instance, description D6). Fortunately, the problem can be reduced to a well-studied problem in deductive database systems. In this section, we discuss how to reduce the "support" problem for QDTL descriptions and MSL queries to a relational context, and we extend existing results from deductive database theory to solve the support problem.

## 7.1 Correspondence of OEM to Relational Models

In this subsection we discuss how to relationally represent OEM objects, MSL queries, and QDTL descriptions. Note that, the principles of our algorithm can be applied to other object-oriented models as well. For applying our algorithm, the MSL queries and QDTL descriptions are actually converted to relational terms. The objects in the underlying sources are not converted. We discuss how they might be represented relationally to better explain the algorithm.

OEM objects are represented relationally by flattening them into tuples. Each object is represented using tuples of three relations, namely `top`, `object`, and `member`. OEM objects can be converted mechanically to the relational representation using a few straightforward rules: For an object $o$ with object-id `oid`, label `l`, and an atomic value `v`, we introduce the tuple

$$\texttt{object}(\texttt{oid}, \texttt{l}, \texttt{v})$$

If $o$ is a set object with object-id `oid` and label `l`, then we introduce the tuple

$$\texttt{object}(\texttt{oid}, \texttt{l}, \texttt{set})$$

Assuming that $o$ has subobjects $o_i, 1 \leq i \leq n$, identified by $\texttt{oid}_i$ , $1 \leq i \leq n$ we introduce $n$ tuples

$$\texttt{member}(\texttt{oid}, \texttt{oid}_i)$$

where $1 \leq i \leq n$. Finally, if $o$ is a top-level object identified `oid`, we also introduce the tuple

$$\texttt{top}(\texttt{oid})$$

The relational representation of MSL queries is obtained similarly by querying the `top`, `object`, and `member` relations that represent the object structure referenced in the query.

**EXAMPLE 7.1** Consider the query

```
*O :- <O person {<LM last_name 'Smith'>}>
```

The above query selects top-level objects `O`, i.e., the subgoal `top(O)` must hold. Object `O` is a `person` set-object, i.e., the subgoal `object(O, person, set)` must hold. `O` must have a subobject identified by `LM`, i.e. `member(O, LM)` must hold. Finally, `LM` must be a `last_name` object with atomic value `'Smith'`, i.e., `object(LM, last_name, 'Smith')` must hold. We collect all the object-id's `O` that satisfy the stated conditions into a relation `answer`. Thus, the MSL query can be written as the following datalog query:

```
answer(O) :- top(O), object(O, person, set),
             member(O,LM), object(LM, last_name, 'Smith')
```

□

The general algorithm for converting an MSL query to a relational form is given in [P+]. A similar algorithm for translating a QDTL description to a relational description is described in [P+]. We illustrate the translation via an example.

**EXAMPLE 7.2** Consider description D6 from Subsection 5.2. The equivalent relational representation is:

```
(R10) Query ::= answer(OP) :- top(OP), object(OP, person, set), _Cond(OP)
      _Cond(OP) ::= member(OP, OS), object(OS, $Label, $Value), _Cond(OP)
      _Cond(OP) ::=
```

Note, the nonterminal `_Cond` has been replaced by the nonterminal `_Cond(OP)` that has one parameter. We need this parameter because we have to denote that object `OS` that appears in the nonterminal template associated with `_Cond` is a subobject of `OP`. We associate with every template of the relational representation, the action of the corresponding template of the original QDTL template. □

### 7.2 Algorithm

In this section we illustrate the algorithm that for a given MSL query written relationally, finds maximal supporting queries from a QDTL description also written relationally. If the query is indirectly supported, the algorithm derives the filter MSL query that needs to be applied to the OEM objects picked by the underlying source.

First we illustrate the process of finding a supporting query given the description $D$ and the query $Q$. Then we show how description $D$ can be expressed as a (possibly recursive) Datalog program $P(D)$. We show that the problem of determining if a description $D$ supports query $Q$, is the same as the problem of determining if program $P(D)$ contains[3] (subsumes) query $Q$ and a corresponding filter query exists. Thus, a supporting query is found in two steps: (a) find a subsuming query, and (b) find the corresponding filter. We extend an existing algorithm that checks containment (from Section 14.5 of [Ull89]), to answer step (a). We refer to the containment algorithm from [Ull89] as QinP. We extend the algorithm to handle step (b).

Algorithm QinP gives a yes/no answer to the containment question and thus to the subsumption question. Thus, we further extend the algorithm to find the actual maximal supporting queries, the corresponding filters, and also the native query constituents for the underlying source. We describe in detail the extended algorithm *X-QinP* in the Appendix. We continue with examples to illustrate the required extensions.

**EXAMPLE 7.3 (Finding Supporting Queries)** This example illustrates, in relational terms, how to find supporting queries for a MSL query from a QDTL description. We use this example in the rest of this subsection.

Consider the query Q16 that selects all `person` objects that have a subobject with label `last_name` and value 'Smith':

```
(Q16)  answer(O) :- top(O), object(O, person, set), member(O,N),
                    object(N, last_name, 'Smith')
```

Consider the description D11 that supports queries that select `person` objects that have at least one subobject that has a specified label and a specified value.

```
(D11) (QT11.1)   Query ::= answer(P) :- top(P), object(P, person, set), _Cond(P)
      (NT11.1)   _Cond(P) ::= member(P,X), object(X,$L,$V)
```

By expanding template QT11.1 using nonterminal expansion rule NT11.1 we obtain expansion (E17).

```
(E17):  answer(P) :-top(P), object(P, person, set), member(P,X), object(X,$L,$V)
```

(E17) is identical to query Q16 by substituting appropriately variables and placeholders. Thus, $D11$ directly supports $Q16$.

Alternatively, consider query Q18 that picks `person` objects with specified values of subobjects `last_name` and `ssn`.

```
(Q18)  answer(O) :- top(O), object(O, person, set), member(O,L),
                    object(L, last_name, 'Smith'), member(O,S),
                    object(S, ssn, '123')
```

[3] A query $Q$ is contained in a program $P$ if for all databases, $P$ derives a superset of the answers derived by $Q$.

Description D11 does not directly support query Q18 because the query imposes selection conditions on two subobjects whereas the description supports queries with only single subobject selections. However, E17 produces two queries that indirectly support Q18:

- E19 enforces the selection condition on subobject `last_name`.

  ($E19$): `answer(O) :- top(O), object(O, person, set), member(O,L), object(L, last_name, 'Smith')`

- E20 enforces the selection condition on subobject `ssn`.

  ($E20$): `answer(O) :- top(O), object(O, person, set), member(O,S), object(S, ssn, '123')`

□

As illustrated above, nonterminals in a query template are expanded to yield expansions of the query template that match the query of interest. If a nonterminal is defined using a recursive template, then the query template has an infinite number of expansions. To find a supporting query requires checking if query $Q$ matches one or more of the infinite number of expansions.

In the next section we show how to reduce the problem of finding a supporting query in a description to the problem of determining whether a conjunctive query is contained in a Datalog program. We extend a known solution to the latter problem to find all the supporting queries, the corresponding filter queries, and the corresponding native query constituents.

### 7.2.1 Expressing Descriptions as Recursive Datalog Programs

In description D11, if we replace the query template with the rule defining predicate `answer`, and replace `::=` with `:-` in the nonterminal template NT11.1, then we get a Datalog program that uses constant placeholders in addition to variables and constants.[4] The constant placeholders are similar to variables except that they match a subset of the constants which the variables match, and placeholders are used in the actions that produce the native query constituents. We use $P(D)$ to refer to the Datalog program corresponding to description $D$. The process of finding an expansion of a query template in a description $D$ that matches a target query $Q$, is the same as determining if the Datalog program $P(D)$ produces a rule $E$ that defines predicate `answer` and matches query $Q$. Rule $E$ matches query $Q$ if the head of $E$ maps to the head of $Q$, and each subgoal of $E$ maps to some subgoal of $Q$ (with appropriate restrictions on how to map variables, placeholders, and constants). Query $Q16$ and expansion (E17) in Example 7.3 illustrated this case.

Note, in our framework both $Q$ and $E$ are *conjunctive* queries [Ull89] extended with placeholders. From existing work on the containment of Datalog queries we know that the existence of a mapping from $E$ to $Q$ is a necessary and sufficient condition for the containment of $E$ in $Q$.[5] Thus, the problem of determining if a description $D$ supports a conjunctive query $Q$ is the same problem as determining if some rule produced by Datalog program $P(D)$ contains query $Q$ (modulo the existence of a filter query). Furthermore, for Datalog this question is the same as asking if the program $P(D)$ contains $Q$. Algorithm QinP from [Ull89] answers exactly this question.

### 7.2.2 Applicability and Extensions of Algorithm QinP

First, we illustrate how the containment algorithm QinP finds subsuming queries given a query and a description. Then we illustrate the extensions that need to be made to Algorithm QinP.

[4] Templates with empty expansions are handled as explained in the Appendix.

[5] The containment results used in this paper, hold in the presence of constant placeholders.

**EXAMPLE 7.4 (Applying Algorithm QinP)** Consider query Q18 from Example 7.3.

```
(Q18)  answer(O) :- top(O), object(O, person, set), member(O,L),
                    object(L, last_name, 'Smith'), member(O,M),
                    object(M, ssn, '123')
```

and the description D11

```
answer(P)  :-  top(P), object(P, person, set), Cond(P)
Cond(P)    :-  member(P,X), object(X,$L,$V)
```

To determine if program P(D11) contains query Q18 Algorithm QinP does the following: First the algorithm "freezes" Q18, i.e., it replaces each variable in each subgoal of Q18 by a corresponding "frozen" constant and puts the resulting frozen facts in a database DB(Q18). The frozen constant for a variable is represented by a constant of the same name in lower case and with a bar on it. The over-bars distinguish frozen constants from regular constants.

top($\bar{o}$), object($\bar{o}$, person, set), member($\bar{o}$,$\bar{l}$), object($\bar{l}$, last_name, 'Smith'),
member($\bar{o}$,$\bar{m}$), object($\bar{m}$, ssn, '123')

Then, the program P(D11) is evaluated on DB(Q18) to check if the program derives the frozen head of Q18, namely "answer($\bar{o}$)". If yes, then it is the case that the program contains the query.

While evaluating the program on the frozen database, constant placeholders in P(D11) are assigned only regular constants and not frozen constants, because frozen constants correspond to variables in the target query. Variables in P(D11) are assigned either frozen or regular constants. □

The above example illustrates that Algorithm QinP gives only a yes/no answer to the subsumption question. That is, if program $P(D)$ derives the frozen head of query $Q$ then we know that $D$ subsumes $Q$. However, the algorithm does not find the particular subsuming query (for instance, ($E19$) in Example 7.3). The algorithm does not find the selection conditions that are not enforced by each subsuming query (for instance, ($E19$) does not enforce `ssn = '123'`). Finally, algorithm QinP does not retain enough information to build the native query constituents. Algorithm X-QinP provides this functionality and finds all the maximal supporting queries (if there are multiple such queries). We illustrate these points via a set of examples.

**EXAMPLE 7.5 (Multiple Subsuming Queries)** Example 7.3 shows that query Q18 is indirectly supported by Description D11 (page 17) via two subsuming queries ($E19$) and ($E20$). We discuss in more detail how to obtain ($E19$).

```
(E19):   answer(O) :- top(O), object(O, person, set), member(O,L),
                      object(L, last_name, 'Smith')
```

($E19$) is obtained by algorithm X-QinP, because program P(D11) derives the frozen head of query Q18 using frozen base facts top($\bar{o}$), object($\bar{o}$,person,set), member($\bar{o}$,$\bar{l}$), and object($\bar{l}$,last_name,'Smith'). ($E20$) is obtained similarly. As guaranteed by extended algorithm X-QinP, ($E19$) and ($E20$) are *maximal.* □

Note, in Example 7.5 the subsuming queries ($E19$) and ($E20$) do not use all the frozen facts obtained by freezing the target query Q18. Facts not used to derive a subsuming query correspond to unenforced selection conditions and constitute the *residue* for that query. For instance, for subsuming query ($E19$) the frozen facts `member(ō,s̄)` and `object(s̄,ssn,'123')` constitute the residue. A non-empty residue implies that the subsuming query does not enforce all the selection conditions of the input query. Thus, we need to formulate a filter MSL query that when applied to the OEM objects picked by the subsuming query, gives the same result as the input query. A filter query may not always exist as illustrated by the following example.

**EXAMPLE 7.6 (Existence of a Filter query)** Consider a query $Q$ that for all persons with `last_name` 'Smith' picks the subobject corresponding to the `first_name`. Consider a query template $T$ that picks the `first_name` subobjects of all persons. Algorithm X-QinP infers that $T$ generates a query $Q_s$ that subsumes $Q$ along with the residue `member(P,LN), object(LN, last_name, 'Smith')`, i.e., the parent objects of the picked `first_name` subobjects have `last_name` value `'Smith'`. This unapplied selection condition cannot be enforced on the result of query $Q_s$ because there is no way to infer from the result what `first_name` is associated with which `last_name`. Thus, no filter query exists for query $Q_s$. Algorithm X-QinP discards subsuming queries for which no filter query may be formulated. For instance, we discard a subsuming query if its residue refers to an object that is not a subobject of the result of the subsuming query. We also discard queries based on other criteria described in the Appendix.

Algorithm X-QinP generates filter queries for subsuming queries that are retained and thus are supporting queries. A conservative filter query may consist of all the conditions in the input query that can be applied on the result of the supporting query. In this case, some conditions may be redundant. Our algorithm derives optimal filter queries, that is, removes all redundant conditions. Below we illustrate the filter MSL query produced by the algorithm for query ($E19$).

```
*O :- <O person {<S, ssn, '123'>}>
```

□

The last extension to algorithm QinP handles the actions that are executed by the converter to generate the native query constituents. The actions are associated with the nonterminal and query templates of a description $D$. When we reduce a query template or nonterminal template $T$ of a decription $D$ into a rule $R$ of the datalog program $P(D)$ we associate with $R$ the action that is associated with the template $T$. Then, the problem of executing the actions associated with the templates of $D$ reduces to the problem of executing the actions associated with the corresponding rules of $P(D)$. Algorithm X-QinP tracks the rules used to derive a supporting query and subsequently executes the actions associated with these rules to produce the native query constituents.

### 7.3 Performance of X-QinP

In the worst case, X-QinP is exponential in the number of conditions in the query plus the number of templates in the description. Nevertheless, in many practical cases X-QinP is polynomial. For example, if both the query and the templates have explicitly specified labels and there is no recursive template (e.g., description D4) X-QinP needs time proportional to the product of the query size and the number of templates. Furthermore, we expect the number of conditions and templates to be relatively small, so running time should be acceptable.

## 8 Related Work

Integration of heterogeneous information sources has attracted great interest from the database community[Wie92, LMR90, T+90, Gup89, A+91, C+94, FK93]. Significant work has been done on integrating and querying data that is in the same model as the integration system. However, underlying sources may have different data models, thus making necessary the existence of wrappers, and consequently, the facilitation of the wrapper construction. [EH86] points out that typically the construction of a wrapper requires "6 month work". Indeed, there are existing techniques for translating schemas and queries of a data model $A$ (say, relational) to schemas and queries of a data model $B$ (say, an object-oriented data model)[QR95, $A^+91$]. Our query translation methodology is different from the above cited work in two ways:

1. We provide a toolkit that can translate queries from our common data model to queries of *any* data model, i.e. we are not bound to a specific "target" data model. Note, the underlying information sources may even not have a well-defined data model.

2. We assume that the source may have limited query capabilities, i.e., not every query over the schema of the underlying source can be answered.

We contribute in two ways to the problem of limited query capabilities (that has been recently recognized [RSU, $C^+94$] as being very important in integration of arbitrary heterogeneous information sources): First, we provide a concise language for description of query capabilities. Second, we automatically increase the query capabilities of a source.

The problem of finding a supporting query is related to the problem of determining how to answer a query using a set of materialized views in place of some of the base relations used by the query [LY85, $L^+$, RSU]. This work uses a fixed set of prespecified views to answer a query. However, we use an infinite set of views that are specified via templates. The templates can specify views like "all relations obtained by applying a single selection predicate to any relation," thus not requiring that the relation name be known. Alternatively, arbitrary numbers of selection conditions can be specified, thereby allowing in the set of "available" views, views that have arbitrarily long specifications. Another difference from [LY85, $L^+$, RSU] is that our focus is on object oriented views and queries and not relational views even though we use some of the same tools, like containment.

# 9 Conclusions

In this paper we have presented a toolkit that facilitates the implementation of wrappers. The heart of the toolkit is a converter that maps incoming queries into native commands of the underlying source. The converter provides the translation flexibility of systems like Yacc, but giving substantially more power, i.e. translating a much wider class of queries.

The wrapper toolkit is currently under implementation, with the server support library, extractor, and packager already built and tested. These components, with hard-wired converters, have been used to build wrappers for Sybase, a collection of BiBTex files stored on a Unix file system, and a bibliographic legacy system. We are currently implementing the QDTL configurable converter described in this paper; it should be operational by the summer of 1995.

In the future, we plan to extend the power of QDTL descriptions and of the converter to handle a larger class of queries. The currently handled class of conjunctive MSL queries will be extended by using a containment checking algorithm more general than algorithm QinP. Also, we plan to extend the algorithm to detect when multiple queries of the underlying source together support the given application query.

## Acknowledgements

We are grateful to Jennifer Widom, Andreas Paepcke, Vasilis Vassalos, and the entire Stanford Database Group for numerous 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.

[EH86] A. K. Elmagarmid and A. A. Helal. Heterogeneous Database Systems. TR-86-004, Program of Computer Engineering, Pennsylvania State University, University Park, 1986.

[FK93] J.-C Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. In *Advanced database systems.* N.R Adam, B.K. Bhargava (editors), (ISBN 3-540-57507-3) Springer-Verlag, 1993, pages 313-36.

[GM+] H. Garcia-Molina et al. The TSIMMIS Approach to mediation: Data models and languages (extended abstract). To appear in 1995 NGITS workshop. Also available by ftp at `db.stanford.edu` as `pub/garcia/1995/tsimmis-models-languages.ps`.

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

[LMB92] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[L+] A.Y. Levy et al. Answering queries using views. To appear in *PODS*, 1995.

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

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

[Mar93] D. S. Marshak. Lotus Notes release 3. *Workgroup Computing Report*, 16:3–28, 1993.

[MY89] R. MacGregor and J. Yen. LOOM:integrating multiple AI programming paradigms. *Proc. Intl. Joint Conf. on Artificial Intelligence*, August 1989.

[O+93] B. Oki et al. The Information Bus—an architecture for extensible distributed systems. In *Proc. of the 14th ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, 1993.

[P+] Y. Papakonstantinou et al. A query translation scheme for rapid implementation of wrappers (extended version). Available by ftp at `db.stanford.edu` as the file `~/pub/papakonstantinou/1995/querytran-extended.ps`.

[PGMU] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. Available by ftp at `db.stanford.edu` as the file `~/pub/papakonstantinou/1995/medmaker.ps`.

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

[QR95] X. Qian and L. Raschid. Query interoperation among object-oriented and relational databases. In *Data Eng. Conf.*, pages 271–9, 1995.

[RSU] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. To appear in PODS 95. Also available by ftp at `db.stanford.edu` as `pub/rajaraman/1994/limited-opsets.ps`.

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

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, New York, 1989.

[Wie87] G. Wiederhold. *File Organization for Database Design.* McGraw Hill, New York, 1987.

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

# Appendix

## A Extended Algorithm QinP

### A.1 Extended Algorithm

Now we state a variant of algorithm QinP. Algorithm QinP gives a yes/no answer to the containment question and thus to the support question, modulo the existence of a filter query. We extend the algorithm to find the maximal supporting queries, to construct the corresponding filter queries, and and to construct the corresponding parse trees.

In particular, we extend and modify the algorithm QinP in the following ways:

1. we keep track of which specific expansion of the Datalog program actually contains the query and thus infer the conditions that constitute the residue for the expansions,
2. we keep track of the *implied equalities*. An implied equality arises when we map a variable to a constant. For example, consider the query

   (Q21) `answer(O) :- top(O), object(O,L,V)`

   that supports the query

   (Q22) `answer(O) :- top(O), object(O,person,V)`

   Note, we have to filter the result of Q21 to keep only the objects with label `person`. We will say that the corresponding filter has to check the implied equality `L = person`. Thus, we keep the subgoal `object` of Q22 in the residue, though it maps to the `object` subgoal of Q21.
3. we find "maximal" expansions that have as many conditions of the target query as is possible given the description,
4. we relax the condition that the head of the expansion is the same as the query head to allow the head of the expansion to represent a parent object of the query head,
5. we check that the residue conditions can be evaluated, and
6. we construct the filter that evaluates them.

The algorithm X-QinP follows four basic steps (there are comments in the algorithm that indicate the start of each step):

- **Step 1:** Find the queries with minimal residue with respect to the input query.
- **Step 2:** Select the maximal subsuming queries, i.e. the minimal residue queries that pick objects that contain the required objects.
- **Step 3:** Select the maximal supporting queries, i.e. check the existence of an appropriate filter query for every selected maximal subsuming query.
- **Step 4:** For every maximal supporting query construct an optimal filter query, in the sense that the constructed filter query has as few conditions as possible.

Note, in order to simplify the description of our algorithm we do not include metapredicates and we do not describe the execution of actions.

**Input**

Conjunctive Query $Q$ where head is of the form $answer(X)$
Description $P(D)$ (recursive Datalog program that
defines *answer* and uses EDB *member*, *object*, *top*)

**Output**

A set of maximal supporting queries, associated filters, and associated parse trees

**Method**

Minimize the query $Q$ (see [Ull89]), i.e. remove all rendundant subgoals
Freeze the query $Q$ – replace each variable $A$ with a constant $\bar{a}$

*% Start of Step 1 : Computation of minimal residue instances*
*% add the frozen facts to DB along with the set of underlying facts and implied equalities*
For each ground fact $f$ obtained from the frozen body of $Q$ add to $DB$
the five-tuple $< f, U, I, A, P >$ where
U = $\{f\}$ *% set of underlying facts for f*
I = $\{\}$ *% set of implied equalities used to derive f*
A = $\{\}$ *% set of residue facts resulting from I*
P = $f$ *% parse tree associated with the fact*
The five-tuple $< f, U, I, A, P >$ is called an "instance of fact $f$".
*% Apply the rules of P(D) to the facts in DB to generate all possible ground facts*
*% along with their underlying facts, implied equalities, and parse trees*
For all rules that have an empty body, "$h(\bar{H}) :-$"
Add the fact $< h(c), \{\}, \{\}, \{\}, nil >$ to $DB$ for all constants and frozen constants $c$ in $DB$.
Loop
For $1 \leq i \leq k$ where $k$ is the number of rules in description $P(D)$ do
Let rule $r_i$ be:
$h(\bar{H}) : -p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$
where $\bar{X}$ is the set of variables and placeholders in $\bar{H} \cup \bar{X}_1 \cup \ldots \cup \bar{X}_n$
For each assignment $\theta$ that:
1. maps variables $V$ in $\bar{X}$ to constants and frozen constants
2. maps every placeholder $V$ in $\bar{X}$ to a constant
3. there exists a vector $[t_1, \ldots, t_n]$
such that $t_j =< \theta(p_j(\bar{X}_j)), U_j, I_j, A_j, P_j >$ and $t_j$ is in $DB$

do *% derive "optimal" instances of $\theta(h(\bar{H}))$*
Initialize sets $I_{temp}$ and $A_{temp}$ to $\{\}$.
For each variable $V$ in $\bar{X}$ that $\theta$ maps to a constant
Add the mapping $V \rightarrow \theta(V)$ to $I_{temp}$ *% Add an implied equality*
Find a $\theta(p_j(\bar{X}_j))$, such that $V \in X_j$, insert $\theta(p_j(\bar{X}_j))$ in $A_{temp}$

*% For each valid instantiation of rule $r_i$, add an instance of a fact to DB*
(A) For every vector $[t_1, \ldots, t_n]$ where $t_j =< \theta(p_j(\bar{X}_j)), U_j, I_j, A_j, P_j >$ and $t_j$ is in $DB$
Let $t_{new} =< f_{new}, U_{new}, I_{new}, A_{new}, P_{new} > =$
$< \theta(h(\bar{H})), \bigcup_{j=1}^{n}(U_j), \bigcup_{j=1}^{n}(I_j) \cup I_{temp}, \bigcup_{j=1}^{n}(A_j) \cup A_{temp}, node(r_i, [P_1, \ldots, P_n]) >$
For all $t \in DB$ of the form $< f_{new}, U_t, I_t, A_t, P_t >$
*% Discard $t_{new}$ if it uses fewer subgoals and*

*% has more implied equalities than some $t \in DB$*
If $U_{new} \subseteq U_t$ and $I_{new} \supseteq I_t$
continue with next iteration of (A)
*% Discard $t$ if it uses fewer subgoals and has*
*% more implied equalities than $t_{new}$*
If $U_t \subset U_{new}$ and $I_t \supset I_{new}$
Remove $t$ from $DB$
*% Add "better" or incomparable new instances*
Add $t_{new}$ to $DB$
Until no new instances of facts are derived

*% Step 2: Find all maximal subsuming queries*
For each instance $t =< f, U_t, I_t, A_t, P_t >$ in $DB$ such that
assuming that $\texttt{answer}(\bar{x})$ is the frozen head of query $Q$, either
$f = \texttt{answer}(\bar{x})$, or
$f = \texttt{answer}(\bar{y})$ and there is a sequence of `member` facts
$\texttt{member}(\bar{x}, \bar{s_1}), \ldots, \texttt{member}(\bar{s_n}, \bar{y})$, i.e. $\bar{y}$ is reachable from $\bar{x}$
$residue(t)$ = ((subgoals in frozen $tail(Q)$) minus $U_f$) union $A_f$

*% Step 3: Check if an appropriate filter $f$ exists for the query $q$ represented by $t$*
*% if $f$ exists then $q$ is a maximal supporting query*
if $t =< \texttt{answer}(\bar{w}), U_t, I_t, A_t, P_t >$ satisfies the following conditions
1. for every subgoal $\texttt{object}(\bar{z}, L, V)$ or $\texttt{member}(\bar{z}, \bar{z'})$ there is a sequence of `member` facts $\texttt{member}(\bar{w}, \bar{s_1}), \ldots, \texttt{member}(\bar{s_n}, \bar{z})$, $n \geq 0$, i.e. $\bar{z}$ is reachable from $\bar{x}$
2. there is no frozen constant $\bar{v}$ that appears in more than two subgoals such that an instance of $\bar{v}$ appears in $residue(t)$ and another instance of $\bar{v}$ is not reachable from $\bar{w}$ via `member` facts

*% Step 4: Construct filter and maximal subsuming query*
For each instance $t =< f, U_f, I_f, A_f, P_f >$ do
Initialize `store` to be the empty set
For each subset $S$ of $body(Q)$ such that $S$ is a superset of $residue(t)$ do
if $Q$ is equivalent to "$head(Q) : -NV(U_f), S$" then
*% $NV$ replaces each frozen constant $\bar{x}$ with a unique variable $X'$ except*
*% the argument of $f$ that is replaced by the unfrozen variable $X$*
add $S$ to `store`
Eliminate all $S \in$ `store` if $\exists S' \in$ `store` such that $S' \subseteq S$
For each remaining $S \in$ `store`
output query $head(Q) : -f, S$ as the filter query.
else discard $t$

**Definition A.1 (Filter of a query $q_s$ with respect to input query $q$)** Assume that $q_s$ defines the predicate $\texttt{answer}_s$ and $q$ defines the predicate `answer`. A filter $q_f$ of $q_s$ wrt $q$ is any query of the form

$$\texttt{answer}_f(X) : -\texttt{answer}_s(Y), \langle cond(Y) \rangle$$

where $\langle cond(Y) \rangle$ is a set of subgoals `member` and `object` such that

- every subgoal $\texttt{member}(S_p, S_c)$ of $\langle cond(Y) \rangle$ is reachable from $Y$,

- every subgoal $\texttt{object}(O, \mathit{label}, \mathit{value})$ of $\langle \mathit{cond}(Y)\rangle$ is reachable from $Y$, and
- $\texttt{answer}_f(x)$ holds if and only if $\texttt{answer}(x)$ also holds

□

**Definition A.2 (Indirect support of a query $q$ by a query $q_s$)** A client query $q$ is indirectly supported by a query $q_s$ if there is a filter of $q_s$ with respect to $q$. □

**Definition A.3 (Minimal residue instance)** Any instance $t =< f_t, U_t, I_t, A_t, P_t >$ is called minimal residue instance if there is no $t' =< f_{t'}, U_{t'}, I_{t'}, A_{t'}, P_{t'} >$ such that $U_t \subset U_{t'}$ and $I_{t'} \subset I_t$. □