Abstract

The thesis presents a system that provides integrated access to heterogeneous information sources that may contain unstructured or semistructured data that are not described by a regular schema (e.g., the World-Wide-Web). The sources may have different and limited query capabilities and complete knowledge of their contents and structure may not exist.

First an abstraction is proposed for the representation of semistructured sources. Then a query translation scheme is proposed for the rapid development of wrappers, i.e., agents that transform queries expressed in the common data model to queries in the native language of the underlying information source. The implementor provides a description of the (potentially limited) set of queries supported by the wrapper along with actions that do the translation.

Finally, an object-oriented logic is proposed for the declarative specification of mediators, i.e., agents that create integrated views of the data exported by the wrappers. The mediators can fuse data in an environment of semistructured sources and/or sources with changing schemas (indeed, the implementor does not need complete knowledge of the sources schemas.) The thesis presents and evaluates key query decomposition and optimization techniques that significantly reduce the cost associated with information fusion in the described environment. In addition, it presents an algorithm, run by the mediator, that given the descriptions of the (potentially limited set of) queries supported by the underlying wrappers it develops plans that retrieve the needed data using supported queries only. The descriptions may or may not be schema specific and they can describe very large or even infinite sets of "query patterns".

Most of the proposed system is implemented, as part of the TSIMMIS project at Stanford University, and integrates information from relational databases, semistructured files, and legacy systems. Part of the work has been done for the Garlic project at IBM Almaden.

Chapter 1

Introduction

A significant challenge facing the database field in recent years has been the integration of heterogeneous databases. Enterprises tend to represent their data using a variety of conflicting data models and schemas, while users want to access all data in an integrated and consistent fashion. There has been substantial progress on database integration techniques [A^+91, Gup89, LMR90, T^+90]; in addition, emerging standards such as SQL3 are aimed at eliminating many of the problems.

At the same time, however, the problem of integration has become much more challenging because users want integrated access to *information*—data stored not just in standardized SQL databases, but also in, e.g., object repositories, knowledge bases, file systems, and document retrieval systems. In addition, users want to integrate this information with "legacy" data, and even with data that is not stored but rather arrives on-line, e.g. over a news wire. As an example, consider a stock broker tracking a company, say IBM. The broker's information sources may include IBM product announcements, the stock market ticker tape, IBM profit/loss statements, news articles, structured databases containing historical information (dividends per year), personnel information (the 100 top-paid executives), general information (the Fortune 500), and so on. Queries may range from simple ones over a single source (e.g., *What were IBM sales in 1990?*), to ones involving multiple sources (e.g., *Get all recent news items where an IBM executive is mentioned*), to complex analyses (e.g., *Is IBM stock a good buy today?*).

The goal of this thesis is the development of technologies that allow *integrated query processing* over heterogeneous information sources. Work done on the integration of conventional databases has already demonstrated the complexity of the integration activity. Moreover, the integration of multiple arbitrary sources is even harder because of the following special characteristics of the environment:

- The integration system can not be based on the assumption that there are well-structured schemas describing each source - as is the case with relational or object-oriented databases. There are two reasons for this requirement:

  1. Many of the sources contain data that is unstructured or semistructured, having no regular schema to describe the data. For example, a source may consist of free-form text; even if the text does have some structure, the "fields" (e.g., author, title, etc.) may vary in unpredictable ways.
  2. The environment is dynamic. The number of sources, their schemas, and the meaning of their schemas may change frequently. For example, the stock broker's company may add or drop an information source depending on its cost and usefulness; attributes may be dropped from some tables and other attributes may be introduced. The sources may not give any notification to the integration system about these changes.

  In light of these challenges we first developed a schema-less information exchange model. Then we developed an integration system where the integration process is described declaratively and the system can gracefully handle semistructured data and/or limited knowledge of the source structure.

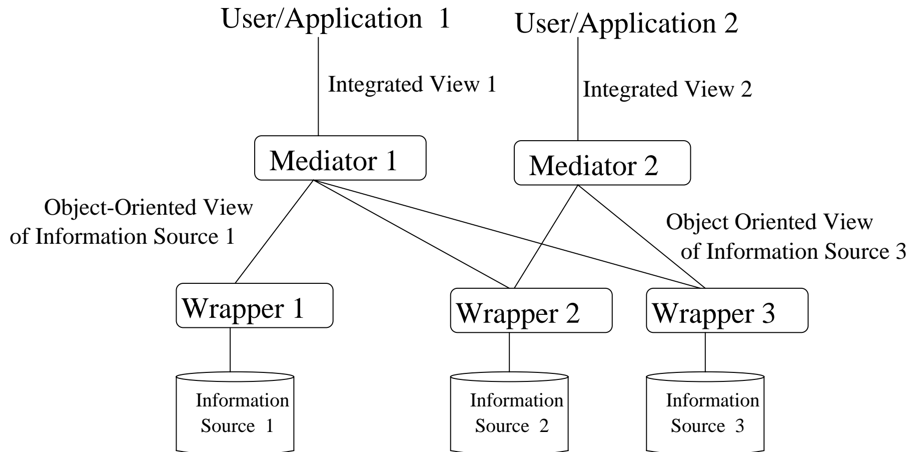

Figure 1.1: The TSIMMIS architecture for integration

- The sources have different and limited query capabilities. For example, a powerful relational database can answer arbitrarily complex SQL queries, such as "find the stocks that increased more than the average." On the other hand, a primitive source may only be able to export a file with recent stock market prices. In this case, the complex query mentioned above can not be directly answered. Nevertheless, the integration system could obtain the answer to the query by retrieving the full file, computing the average increase, and finding the stocks that have increased more than the average.

  In light of this challenge, we developed algorithms that, given descriptions of the query capabilities of the sources, allow the integration system to adapt to the sources' capabilities.

Before we proceed with a brief presentation of our systems' architecture we make a few remarks that may help position our work. First, the thesis focuses on query processing issues pertaining to an integration system. There are many other aspects of an integration system that are equally important (e.g., updates, user interface, and so on) but are not discussed in this thesis. Second, most of the work described in this thesis has been implemented as part of the TSIMMIS system. Indeed, most of the problems that we solve became apparent as the system was being built and tested.

1.0.1 Basic Architecture, Terminology, and System Operation

The TSIMMIS data-integration system provides integrated access via an architecture (see Figure 1.1) that is common in many projects: *Wrappers* [C$^+$95, FK93] (also called *translators* [PGMW95]) convert data from each source into a common object-oriented model, as illustrated in Figure 1.1. The wrappers also provide a common query language and common data model for extracting information. When they receive a query they translate it into a source-specific query or command that is issued to the source. On the way back, they translate the source result into the common model. Applications can access data directly through wrappers, but they may also go through *mediators* [PGMW95, Ire, Wie92]. A mediator combines, integrates, or refines data from wrappers, providing applications with a "cleaner" integrated view. For example, a mediator for Computer Science publications could provide access to a set of bibliographic sources that contain relevant materials. Users accessing the mediator would see a single collection of materials, with, for example, duplicates removed and inconsistencies resolved (e.g., all authors names would be in the format last name, first name.)

The main focus of this thesis is the development of technologies that allow the implementation of wrappers and mediators from high level specifications of their functionality, as shown in Figure 1.2. We develop a mediator by providing a declarative *mediator specification* to a generic mediator specification *interpreter* that

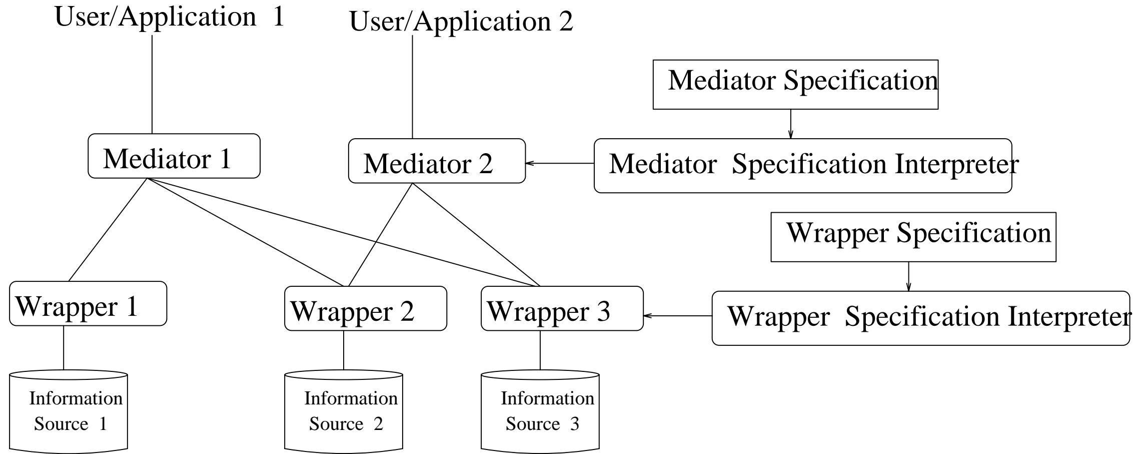

Figure 1.2: Wrapper and Mediator Generation

we developed. Similarly, we develop a wrapper by providing a *wrapper specification* to a generic *wrapper generator* that we developed. The wrapper specification shows how queries expressed in the common query language are translated into queries expressed in the native query language of the underlying sources.

During run time, when the mediator receives a client query it decomposes it into common language queries which are sent to the wrappers. These queries are translated by the wrappers into queries of the underlying system. However, given the limited capabilities of the sources, it is not easy to derive a native query which is actually supported by the underlying system. There are two approaches to this problem:

- **The TSIMMIS approach:** The wrappers can indirectly support queries which are not directly supported by the underlying system. Then the mediator decomposes client queries assuming that the wrappers are powerful enough to handle any query they are given. Hence, the query decomposition module of the mediator is not restricted to using the limited set of queries which are supported by the underlying source. This is the approach that has been actually implemented in TSIMMIS and is presented in detail in Chapters 3 to 7.

- **The Garlic approach:** The wrappers may support only a limited set of queries which are either directly supported by the underlying system or they are "easy" to implement at the wrapper level. Then the mediator adapts to the limited capabilities of the wrappers. The Garlic approach is described in Chapter 8.

The Garlic approach is superior to the TSIMMIS approach in that there are client queries which can be handled using the Garlic approach but can not be handled using the TSIMMIS approach. This power difference will become explicit in Chapters 7 and 8. On the other hand the TSIMMIS approach leads to a simpler separation of tasks. In particular, TSIMMIS mediators — unlike Garlic mediators — do not handle query optimization in parallel with the limited query capabilities issues. This separation of tasks simplifies the algorithms.

Overview

In Chapter 2 we present related work from database integration and other fields that is instrumental to the wrapper and mediator technologies. Note, we also discuss work that is related to each chapter at the end of it. In Chapter 3 we describe the data model on which we build our integration system and we explain our motivation for choosing this model for our system. We also describe the client-server architecture of our system. Chapter 4 presents a mediator specification language that allows the implementor to declaratively

specify how the data exported by wrappers are integrated into a single view. Chapter 5 discusses the query processing algorithms of the mediator specification interpreter and Chapter 6 assesses the performance of the interpreter's algorithms. In Chapter 7 we describe the wrapper generation technology and discuss a solution to the limited query capabilities problem. Chapter 8 presents a more powerful approach to the problem of adapting to limited and different source capabilities. Appendices provide the formal semantics of languages that have been presented in this thesis and the full description of algorithms that are too complex to be formally described in the main body of the thesis.

Chapter 2

Related Work

We summarize in the following paragraphs work that is related to the wrapper and mediator technologies described in the subsequent chapters. Note, detailed comparisons between specific topics of this thesis and related work are found in the corresponding chapters.

Earlier database integration works [A^{+}91, K^{+}93, BLN86, LMR90, T^{+}90, Gup89, FLNS88, ACHK93] focused on the integration of well-structured databases that support powerful query languages. As described in the introduction, this thesis focuses on extending/adapting these technologies for integrating arbitrary information sources. Hence, we revisit many concepts of database technology.

We classify the related work in four categories. First, we describe data modelling work that affected our development of the TSIMMIS object-oriented data model. Second, we describe view specification and query language works and we compare them with our mediator specification language which can be viewed as an object-oriented query/view language. Third, we describe query processing techniques that have been used or extended in our system. Finally, we present work related to wrapper generation and the problem of limited query capabilities.

Data Modelling and Semistructured Data Our focus is on semistructured data, which is information that may not conform to a rigid schema fixed in advance. It is frequently found, for instance, in the World-Wide-Web, SGML documents, semistructured repositories such as ACeDB [TMD92] (very popular among biologists in the Human Genome Project), and Lotus NOTES[Mar93]. To represent such data, we use a "schemaless"(or self describing [MR87]) object-oriented model. Indeed, many applications that have to deal with semi-structured information use a *self-describing* model, where each data item has an associated descriptive label. Applications include tagged file systems [Wie87], Lotus NOTES [Mar93], the Teknekron's Information Bus [O^{+}93], LOOM frames [MY89], electronic mail, RFC1532 bibliographic records, and many more. In [PGMW95] we have defined a self-describing data model [MR87], called the *Object Exchange Model (OEM)*, that captures the essential features of the self-describing models used in practice and also generalizes them to allow nesting and to include object identity. Many projects have used object-oriented models for integration purposes [C^{+}95, A^{+}91]. The main difference between OEM and these strongly-typed object-oriented data models [Cat94, KKS92] is the absence of a predefined schema in OEM.

Mediation, Object-Oriented View Definition and Query Languages Our work builds on many prior results and experiences, and we briefly review here some of them. Many projects have dealt with data integration and fusion (e.g., [LMR90, Gup89, A^{+}91, C^{+}95, S^{+}, FK93, HM93, H^{+}92, TRV95, K^{+}93].) Most of them base fusion on a precise description of the schemas exported by the sources, along with designer provided descriptions of the semantic connections between the entities of the schemas ([HM93, H^{+}92] are representative of this approach.) [MIR94, MI93] consider the problem of schema integration from the perspective of *information capacity*. Also, CASE tools have adopted the approach of thoroughly modelling the source data semantics (e.g., [DIS, Inc]). Thorough classifications have been developed for the various

schematic and semantic conflicts that may be found in schemas, and corresponding techniques are suggested for conflict resolution [BLN86, DH86, ME84, K+93]. Some approaches go one step further by modeling the sources as knowledge bases (see e.g., [EL85, ACHK93]) and use this knowledge to perform integration. Unlike these approaches we assume minimal knowledge of the structure and contents of the sources.

Querying and integrating semistructured data is also considered in [FK93, QRS+95, PGMU96, ACM93, PDS95]. We believe that approaches based on the relational model (and corresponding view definition languages) are not applicable here. (Indeed, one may argue that they are insufficient even for relational database integration [KLK91].) Object-oriented database systems do provide more flexibility. However, their strict type system [C+95, BDH+95] sometimes is a handicap. Also, primitives for data integration found in these systems are still quite limited, with some exceptions like work on views in the context of OQL (e.g., [SAD94]).

The mediator specification language is essentially an object-oriented logic, but has certain simplifying features that we present in Chapter 4. Indeed, in the absence of negation and semantic object-id's, MSL can be viewed simply as a variant of Datalog [Ull88]. However, unlike Datalog, MSL makes it possible to handle both unstructured and structured data.

Object identity plays an instrumental role in the specification of mediators that fuse objects. In particular, MSL's handling of object identity is influenced by [Mai86, KKS92, CKW93, KL89, HY90, AK89]. Further discussion on this topic appears in Section 4.5. Finally, MSL can handle schema components and can be compared in this aspect to [LSS93, KLK91] (details in Section 4.5).

Query Processing Though MSL can be reduced to a variant of Datalog [Ull89], query execution against mediators cannot be achieved by a simple modification of datalog evaluation mechanisms because the environment (i.e., remote heterogeneous sources) is radically different from a conventional database. We present a variant of top-down depth-first resolution [GN88] to formulate the queries that will be sent to the sources and also push conditions to the sources, hence implementing the well known algebraic optimization technique. Beyond this, we also investigate optimizations needed for reducing the volume of data that is retrieved from the sources during object fusion and we show that our technique outperforms traditional optimization algorithms [OV91].

Wrapper Generation and the Problem of Limited Query Capabilities Our wrapper generation technology is reminiscent of the Yacc parsing facility. Earlier wrapper- generation efforts have either used brute force [EH86, FK93] or have focused on specific translations. For example, there are algorithms 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)[A+91]. Our query translation methodology is more general and it also handles the case where the source has 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 [RSU95, C+95] 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 how to process a query which is not directly supported by the underlying sources 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, LMSS95, RSU95]. These works use a fixed set of prespecified views to answer a query. However, we use an infinite set of views that are specified via templates. [LRU96] provides some important theoretical results on using Datalog programs for describing the limited capabilities of sources.

Chapter 3

Information Exchange in TSIMMIS

The components of our integration architecture need to exchange data objects (units of information), either for examination by an end user or for integration with other data objects. For this, there needs to be an agreement as to how objects will be requested, how they will be represented, what the semantic meaning of each object (and its components) is, and how objects are actually transported over the network. This brings up the question "Which are the properties of a model appropriate for integration ?" In this chapter we address this question and we discuss the data model and the communication protocol that form the information exchange protocol between any two components of our architecture.

In Section 3.1 we present an "object exchange model" (OEM) that we believe is well suited for information exchange in heterogeneous, dynamic environments. OEM is flexible enough to encompass all types of information, yet it is simple enough to facilitate integration; OEM also includes semantic information about objects. In Section 3.2 we present a pair of general-purpose libraries we have implemented that support OEM object exchange between any client and server processes. The procedures in these libraries are linked with the server program or the client program, thus allowing the development of client programs with *embedded* calls for the exchange of objects. They provide communication services, session handling, object memory management, and partial object fetches (since the result of a query may be large). In Section 3.3 we compare our information exchange protocol to other information exchange protocols.

3.1 Object Exchange Model

The first question to be addressed is: with so many data models around, why do we need another one? In fact we do *not* need another new data model. Rather, we adopt an essential set of data model concepts that have been used for more than 30 years. Using these concepts we formally cast a data model that facilitates information exchange in heterogeneous systems.

The basic idea is very simple: each value we wish to exchange is given a *label* (or *tag*) that describes its meaning. For example, if we wish to exchange the temperature value 80 degrees Fahrenheit, we may describe it as:

⟨temperature-in-Fahrenheit, integer, 80⟩

where the string "temperature-in-Fahrenheit" is a human-readable label, "integer" indicates the type of the value, and "80" is the value itself. If we wish to exchange a complex object, then each component of the object has its own label. For example, an object representing a set of two temperatures may look like:

⟨set-of-temperatures, set, $\{cmpnt_1, cmpnt_2\}$⟩
 $cmpnt_1$ is ⟨temperature-in-Fahrenheit, integer, 80⟩
 $cmpnt_2$ is ⟨temperature-in-Celsius, integer, 20⟩

A main feature of OEM is that it is *self-describing*. We need not define in advance the structure of an object, and there is no notion of a fixed schema or object class. In a sense, each object contains its own schema. For example, "temperature-in-Fahrenheit" above would play the role of a column name, were this object to be stored in a relation, and "integer" would be the domain for that column.[1]

Note that, unlike in a database schema, a label here can play two roles: identifying an object (component), and identifying the meaning of an object (component). To illustrate, consider the following object:

⟨person-record, set, {$cmpnt_1$, $cmpnt_2$, $cmpnt_3$}⟩
  $cmpnt_1$ is ⟨person-name, string, "Fred"⟩
  $cmpnt_2$ is ⟨office-number-in-building-5, integer, 333⟩
  $cmpnt_3$ is ⟨department, string, "toy"⟩

Like a column name in a relation, the label "person-name" identifies which component in the person's record contains the person's name. In addition, the label "person-name" identifies the meaning of the component—it is the name of a person. We would not expect to find a dog's name "Fido" or "Spot" in this component.

Thus, we assume that labels are as descriptive as possible. (For instance, in our example above, replacing "person-name" by "name" would not be advisable.) In addition, if an information source exports objects with a particular label, then we assume that the source can answer the question *What does this label mean?*. The answer should be a human-readable description—a type of "man page" (similar in flavor to Unix Manual pages). For example, if we ask the source that exports the above object about "person-name," it might reply with a text note explaining that this label refers to names of employees of a certain corporation, the names do not exceed 30 characters, and upper vs. lower case is not relevant.

It is particularly important to note that labels are relative to the source that exports them. That is, we do not expect labels to be drawn from an ontology shared by all information sources. For example, a client might see the label "person-name" originating from two different sources that provide personnel data for two different companies, and the label may mean something different for each source; the client is responsible for understanding the differences. If the client happens to be a mediator that exports combined personnel data for the two companies, then the mediator may choose to define a new label "generic-person-name" (along with a "man page"), to indicate that the information is not with respect to a particular company.

We believe that a self-describing object exchange model provides the flexibility needed in a heterogeneous, dynamic environment. For example, personnel records could have fewer or more components than the ones suggested above; in our temperatures set, we could dynamically add temperatures in Kelvin, say. In spite of this flexibility, the model remains very simple.

As mentioned earlier, the idea of self-describing models is not new—such models have been used in a variety of systems (see Section 3.3 for a discussion of these models and systems). Nevertheless, it is useful to formally cast a self-describing model in the context of information exchange in heterogeneous systems (something that has not been done before, to the best of our knowledge), and to extend the model to include object nesting as illustrated above. To do this, a number of issues must be addressed, as will be seen in the following section.

3.1.1 Specification

Each object in OEM has the following structure:

Object-Id	Label	Type	Value

where the four fields are:

[1]Of course, if we are exchanging a set of objects where each object has the same structure and labels, then it would be redundant to transmit labels with every member of the set. We view this as a "data compression" issue and do not discuss it further here. From a logical point of view, we assume that each object in our model carries its own label.

- **Label:** A variable-length character string describing what the object represents.
- **Type:** The data type of the object's value. Each type is either an *atom* (or *basic*) type, such as `integer`, `string`, `real number`, etc., or the type `set`, or the type `list`. The possible atom types are not fixed and may vary from information source to information source [2].
- **Value:** A variable-length value for the object.
- **Object-ID:** A unique variable-length identifier for the object.

In denoting an object, we often drop the Object-ID field, i.e. we write ⟨label,type,value⟩, as in the examples above.

Object identifiers (henceforth referred to as OID's) may appear in set and list values as well as in the Object-ID field. We provide a simple example to show how sets (and similarly lists) are represented without OID's, and to motivate the kind of OID's that are used in OEM. Then we discuss OID's in set and list values.

Suppose an object representing an e-mail message has label "message" and a set value. The set consists of three subobjects, an "author", a "title" and a "text". All four objects are exported by an information source IS through a wrapper, and they are being examined by a client C. C can retrieve the "message" object by posing a query (see Chapter 4 or [QRS+95]) that returns the object as an answer.

Assume for the moment that the "message" object is fetched into C's memory along with its three subobjects. The value field of the "message" object will be a set of *object references*, say $\{o_1, o_2, o_3\}$. Reference o_1 will be the *memory location* for the "author" subobject, o_2 for the "title," and o_3 for the "text." Thus, on the client side, the retrieved object will look like:

⟨message, set, $\{o_1, o_2, o_3\}$⟩
  o_1 is location of ⟨author, string, "author's name"⟩
  o_2 is location of ⟨title, string, "some title"⟩
  o_3 is location of ⟨text, string, "a long string"⟩

On the information source side, the "message" object may map to a real object of the same structure, or it may be an "illusion" created by the wrapper from other information. Suppose IS is an object database, and the "message" object is stored as four objects with immutable OID's id_0 (message), id_1 (author), id_2 (title), and id_3 (text). In this case, the retrieved object on the client side would have id_0 in the Object-ID field for the message object, id_1 in the Object-ID field for the author object, and so on. The Object-ID fields tell client C that the objects it has correspond to identifiable objects at IS. Such OID's can be used to retrieve again from the IS the objects that they identify, at any time.

However, in many cases it is complicated, inneficient or even impossible to create a meaningful OID for some object. For example, suppose that the IS is a common mailbox file that contains a sequence of mail messages. In this case the wrapper places an arbitarry string starting with `&` in the OID field for the "message", "author", "title" and "text" objects.

So far we have assumed that the client retrieves the "message" object and all of its subobjects. However, for performance reasons, the wrapper may prefer not to copy all subobjects. For example, if the "text" subobject is a very large string, it may be preferable to retrieve the "author" and "title" subobjects in their entirety, but retrieve only a "placeholder" for the "text" object. In this case, the value field for the "message" object at the client will contain $\{o_1, o_2, id_3\}$. This indicates that the "author" and "title" subobjects can be found at memory locations o_1 and o_2, but the "text" subobject will be explicitly retrieved using OID id_3, if the client judges that the "text" will be interesting to him. Note if it is difficult to construct meaningful object-id's it is a heavy burden for the wrapper to keep track of all arbitrary object-id's it has ever generated. Hence, arbitrary object-id's expire once the next query is given to the wrapper.[3]

Note that, regardless of the representation used in set and list values, the wrapper always gives the client the illusion of an object repository. Thus, we can think of our "message" object as:

[2] For example, a multimedia source may offer a wide variety of non-conventional data types.

[3] Another option is to allow arbitrary object-id's to be valid for the whole client-server session

⟨message, set, {$cmpnt_1$, $cmpnt_2$, $cmpnt_3$}⟩
 $cmpnt_1$ is ⟨author, string, "author's name"⟩
 $cmpnt_2$ is ⟨title, string, "some title"⟩
 $cmpnt_3$ is ⟨text, bits, "a long string"⟩

where each $cmpnt_i$ is some mnemonic identifier for the subobject. (Recall, if $cmpnt_i$ is not meaningful it starts with a `&`.) Indeed, in the rest of the thesis we'll use the following notation for objects:

⟨$cmpnt_0$, message, set, {$cmpnt_1$, $cmpnt_2$, $cmpnt_3$}⟩
 ⟨$cmpnt_1$,author, string, "author's name"⟩
 ⟨$cmpnt_2$,title, string, "some title"⟩
 ⟨$cmpnt_3$,text, bits, "a long string"⟩

A final issue regarding OEM is that of duplicate objects at the client. Suppose, for example, that set object A at the information source has B and C as subobjects. Both B and C are of set type, and both have as subobjects the same object D. A query at a client retrieves A and all of its subobjects. Will the client have a single copy of object D, or will objects B and C point to different copies of D?

Our model does *not* require a single copy of D at the client, since this would place a heavy burden on wrappers that are not dealing with real objects at the information source. However, if both copies of D have the same Object-ID field, then the client can discern that the two objects correspond to the same object at the source. Also note that we do not require wrappers to discover cyclic objects at the source. Suppose, for example, that A has B as a subobject and B has A as a subobject. If the client fetches A from a "smart" wrapper, the wrapper would return only two objects, a copy of A and a copy of B. Each object's set value would be a reference for the other object. However, a "dumb" wrapper is free to return, say, four objects, A_1, B_1, A_2, B_2, where A_1 references B_1, B_1 references A_2, A_2 references B_2, and B_2 contains only the object-id's for the subobjects.

3.2 The OEM Support Libraries

OEM and OEM-QL are designed for a *client* to send queries and obtain corresponding answer objects from a *server*. The server may be a wrapper or a mediator, while the client may be a mediator or an end-user program (such as the Mobie browser described in [H$^+$95]). We have implemented general-purpose OEM Support Libraries that provide the common functionality needed for object and query exchange. There are two main components: the *Client Support Library* (CSL) and the *Server Support Library* (SSL).

Figure 3.1 illustrates how the Support Libraries are used. The implementor of client applications links CSL with the client program in order to create programs with *embedded* CSL calls; CSL calls are used to establish connections with TM servers, to send queries, and to receive OEM objects. CSL procedures handle all low level communications, and deposit retrieved objects in a main memory object buffer. At the server side, the SSL handles incoming connections, buffer management, and management of "slave" processes to execute queries. Note that if a server S obtains its information from another wrapper or mediator, then S also acts as a client, so it also uses the CSL.

Our Support Libraries expedite the implementation of mediators, wrappers, and end-user programs. In addition, implementing these libraries has brought to the surface a number of interesting issues regarding the exchange of objects when one or more participants are not inherently object-oriented. As far as we know, these issues do not arise in conventional, homogeneous object-oriented systems (or at least not in quite this way). Here we discuss one of the most important issues that has arisen, namely that of *partial object fetches*.

In many cases it is extremely inefficient to send the complete answer object to the client in one step. In particular:

1. The client has to wait until the full answer is retrieved from the information source before examining the object. This prevents "pipelined" operation, where the client starts processing subobjects as they arrive. The problem is exacerbated if we have a string of mediators between the source and the client:

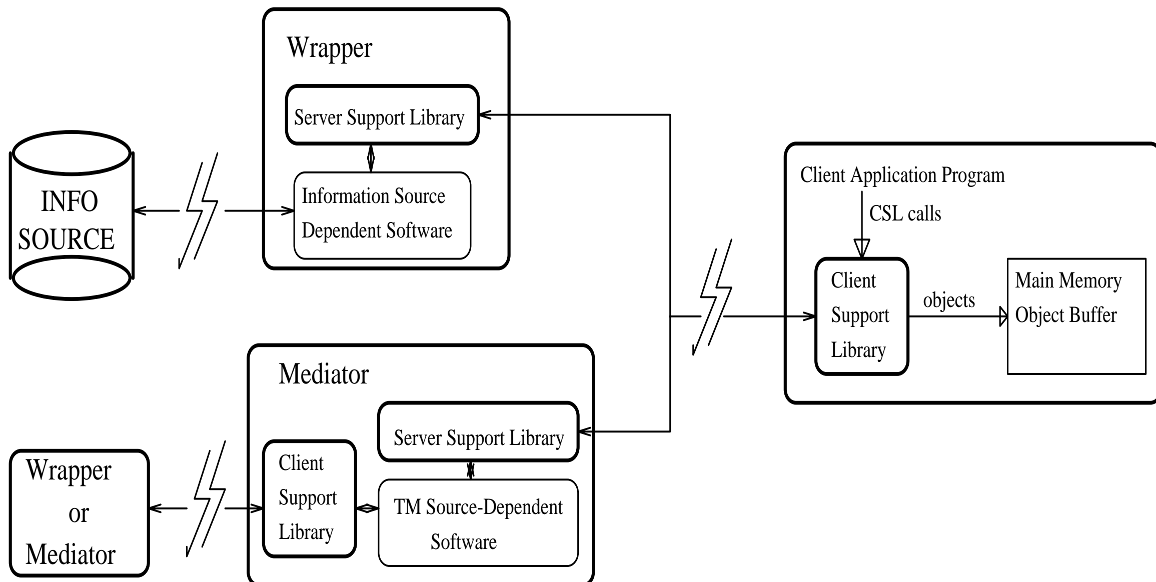

Figure 3.1: Use of the OEM Support Libraries

the client cannot begin processing the answer until all of the intermediate TM's have completed their work.

2. The answer object may be very large. Once a client inspects part of the answer object, the client may determine that it does not need some portions of the answer object, or perhaps does not need the object at all.

To avoid these problems, the Support Libraries provide a *partial fetch* mechanism that enables clients to retrieve only parts of the answer object. The mechanism is used as follows. When the client wishes to request an object, it calls a `query()` function, passing the query as a parameter. The client can then fetch either the full answer object (including subobjects) by calling the `getFullObject()` function, or the client can fetch only the root of the answer object by calling the `getRootObject()` function. In the latter case, additional `getFullObject()` and/or `getRootObject()` calls are used to fetch the subobjects.

Calls to the `getRootObject()` function lead to *incomplete* objects in the client's memory. To illustrate, consider an answer object A whose value is a set of three subobjects, B, C, and D. If a copy of A is fetched in the client memory with a `getRootObject()` call, then the value of A will be a set of three references u_1, u_2, u_3. Indeed, even if the full object A is requested (using a `getFullObject()`), if a subobject of A, say C, is very large the server may decide not to fetch C in the client's memory. Thus, we see that during a client-server session either the client or the server may decide not to fetch some (or all) subobjects of an object A, upon fetching A.

Consider now what happens when a client wants to examine an unfetched object. One option is to have the server provide object references u_1, u_2, u_3 to the unfetched objects. Then the client can issue fetch calls such as `getFullObject(`u_2`)` or `getRootObject(`u_2`)`. Furthermore, the object references may contain semantic information that guides the client in traversing the OEM structure. For example, the object references could contain the label of the referenced object.

However, in many cases the server can not export object references for the subobjects because the information source does not provide any good way to refer to subobjects. For example, the Folio bibliographic source returns a stream of documents and the wrapper has no control over the order of the output. Had the wrapper wanted to export meaningful object references it would have to first retrieve the documents. But such an action defies the original motivation of partial fetch. For these cases the server can output an

iterator which can be used from the client for "getting the next object."

Our goal has not been a full description of the Support Libraries or a major contribution in information exchange. Instead we provide an illustration of the challenging practical issues that arise when the "fetch abilities" of the sources vary, and of the simple solutions which often are sufficient. A more powerful approach to querying and browsing is presented in [CHMW96] where querying and browsing is integrated instead of being two separate phases.

3.3 Discussion and Related Work

In this section we contrast OEM with other similar models and systems. We focus particularly on the differences between OEM and more conventional object-oriented models, and we discuss the motivation behind our design of OEM.

Labeled fields are used as the basis of several data models or data formatting conventions. For example, a *tagged file system* [Wie87] uses labels instead of positions to identify fields; this is useful when records may have a large number of possible fields, but most fields are empty. Electronic mail messages consist of label-value pairs (e.g. label "From" and value "yannis@cs.stanford.edu"). More recently, Lotus Notes [Mar93] has used a label-value model to represent office documents, and Teknekron Software Systems [O+93] has used a self-describing object model for exchange of information in their stock trading systems. Also [LMR90] and [MR87] recognize that increased flexibility is required in heterogeneous system and as a solution they propose methods for the development of self-describing databases .

Recent projects on heterogeneous database systems (e.g., [A+91, Ber91, K+93]) have applied object-oriented (OO) data models to the problem of database integration. OEM differs from these and other OO data models in several ways. First, OEM is an information *exchange* model. OEM does not specify how objects are stored at the source. OEM does specify how objects are received at a client, but after objects are received they can be stored in any way the client likes. OEM explicitly handles cross-system OID's (e.g., in Section 3.1.1 a "message" object at the client points to a "text" object at the source). In a conventional OO system there may also be client copies of server objects, but there the client copy is logically identical to the server copy and an application program at the client is not aware of the difference.

A very important difference between OEM and conventional OO models is that OEM is much simpler. OEM supports only *object nesting* and *object identity*; other features such as classes, methods, and inheritance are omitted. (Incidentally, [Cat91] claims that the only two essential features of an OO data model are nesting and object identity.) Our primary reason for choosing a very simple model is to facilitate integration. As pointed out in [BLN86], simple data models have an advantage over complex models when used for integration, since the operations to transform and merge data will be correspondingly simpler. Meanwhile, a simple model can still be very powerful: advanced features can be "emulated" when they are necessary. For example, if we wish to model an employee class with subclasses "active" and "retired," we can add a subobject to each employee object with label "subclass" and value "active" or "retired." Of course this is not identical to having classes and subclasses, since OEM does not force objects to conform to the rules for a class. While some may view this as a weakness of OEM, we view it as an advantage, since it lets us cope with the heterogeneity we expect to find in real-world information sources.

Furthermore, the simple and schemaless nature of OEM often allows us to model complex features of the source in a user-friendly way. For example, consider a relational (or deductive) database that contains a `parent-table(parent, child)` table. If we wish to provide an OEM model of this source that will make easy to find the ancestors of a person, we can make the object that corresponds to any person contain as subobjects the objects that correspond to his/her parents. Then, it is easy for the user to place a query that retrieves all ancestors of a person. On the same time, the user can easily browse the genealogical tree of a person using the browsing facility described in [H+95].

Also, the schemaless nature of OEM is particularly useful when a client interfaces with a source whose structure is unknown. In traditional object oriented and relational databases a client must know the schema in order to indicate where the data, which is fetched from the database, is placed in the client program's

memory. On the other hand, using OEM a client program does not have to know anything about the objects to be fetched. This feature has greatly simplified the development of programs like the general-purpose MOBIE browser [H+95] which naturally can not know in advance the structure of the underlying source.

Someone might wonder how do we start querying a source with unknown schema and structure? How can we be sure that we did not find "employees" because there are no "employees" as opposed to not finding any because they are actually labeled as "emp". The answer is that we can always issue queries that retrieve labels and structure. Then using the preliminary fashion about structure and labels we can query.

The simplicity of using OEM in this context is evident even when it is compared against standardization proposals that also address the problem of object exchange in heterogeneous environment, *e.g.*, CORBA's Object Request Broker.[4]

A final distinct difference between OEM and conventional OO models is the use of labels in place of a schema. Clearly, it would be trivial to add labels to a conventional OO model (e.g., all objects could have an attribute called "label"). The only difference then is that in OEM labels are first-class citizens. We believe this small change makes interpretation and manipulation of objects more straightforward, as discussed in the next section.

[4]Other standards, like ODMG's Object Database Standard [Cat94] are more directed to facilitating interoperability and portability, rather than facilitating object exchange in heterogeneous environment

Chapter 4

The Mediator Specification Language

Given a set of sources with wrappers which export OEM objects, we would like to build mediators to integrate and refine the information. The significant programming effort involved in the hard-coded development of TSIMMIS mediators [Ire] suggests the need for development of systems that facilitate mediator development. Our mediation system, *MedMaker*, provides a high level language, called the *Mediator Specification Language (MSL)*, which allows the declarative specification of mediators. MSL can also serve as a query language and we will indeed use it to simplify the discusiion of query processing.

At run time, when the mediator receives a query, MedMaker's *Mediator Specification Interpreter (MSI)* collects and integrates the necessary information from the sources, according to the specification. The process is analogous to expanding a query against a conventional relational database view. Indeed, MSL can be seen as a view definition language which is targeted to the OEM data model and the functionality needed for integrating heterogeneous sources. The special requirements of integration led to the introduction of a number of useful concepts and properties that are not found in conventional view definition languages:

- MSL mediator specifications can handle some schema evolution of the underlying sources without a need for rewriting of the specification (see Section 4.1).
- MSL can handle structure irregularities of the sources without producing erroneous or unexpected results.
- MSL can integrate sources for which we do not fully know the object structures (see Section 4.1).
- MSL can manipulate both the values and the descriptive semantic labels in the same fashion, getting around problems such as schematic discrepancies [KLK91] (see Section 4.1). Furthermore, MSL can also manipulate object-id's in the same fashion as values, providing object identification on top of value-oriented sources. (Indeed, datalog terms can be used for object-id's.)
- MSL provides a flexible technique (based on object identity) for grouping of source objects that relate to the same entity.
- MSL facilitates *object fusion*. This involves grouping together information (from the same or different sources) about the same real-world entity. In doing this fusion, the mediator may also "refine" the information by removing redundancies, resolving inconsistencies between sources in favor of the most reliable source, and so on.

The above capabilities are "packaged" in a high-level declarative language that combines power with simplicity and conciseness, thus allowing the client of a heterogeneous system to easily define an integrated view. Note, we also present a novel approach to object fusion that is based on semantic object identifiers. The basic idea is as follows. The mediator is specified by a set of non-procedural, logic rules. Each rule maps a set of objects at a source, which pertain to some identifiable real world entity, into a "virtual" object at

the mediator. The virtual object is assigned a semantically meaningful object identifier. Mediator objects that have the same object-id are then fused together, in a way that is also specified by the rules. The above description is conceptual; no objects are fused until a user query arrives at the mediator. (The mediator specification is like a database view.) Only when a query arrives, are the sources queried for the object fragments that are necessary for composing the selected fused objects.

As we will see, the single concept of semantic object-id's significantly increases the power and flexibility of the language, and makes it relatively easy to specify the most common fusion operations. It also makes it possible to integrate objects that contain references to other source objects. These "remote references" are translated into semantically meaningful references at the mediator, allowing the integration of nested and cross-referenced objects such as those found on the Web.

First, we present an extended example that illustrates the MSL language and some of its integration capabilities. The example is continued in Section 4.2.1, where we introduce the query processing mechanisms of MedMaker, though we postpone the complete discussion of MedMaker's query processing until Chapter 5. Section 4.3 presents specifications that accomplish complicated tasks such as removing redundancies, resolving inconsistencies, etc. An introduction to the formal semantics of MSL is provided in Section 4.4. Section 4.5 compares the mediator specification language to other languages that have been (or could be) used for integration for the integration of heterogeneous information sources. The complete syntax of MSL is provided in Appendix A and the complete semantics appear in Appendix B.

4.1 A Mediator Specification Example

For our extended example, we consider two sources that contain information on the staff of a Computer Science department. The first source is a relational database containing two tables with schemas

```
employee(first_name, last_name, title, reports_to)
student(first_name, last_name, year)
```

A wrapper, named `cs`, exports this information as a set of OEM objects, some of which are shown in Figure 4.1. Notice how the schema information has now been incorporated into the individual OEM objects.[1]

A second source is a university "whois" facility that contains information about employees and students. A wrapper `whois` provides access to this source; several sample objects are shown in Figure 4.2. Notice that in this case there can be irregularities. For instance, object `&p1` contains an email subobject while `&p2` does not.

Let us now consider a mediator, called `med`, that has access to wrappers `cs` and `whois` and exports a set of "cs_person" objects. Our goal in this example is that each "cs_person" object represents a person appearing in both sources. The subobjects of each "cs_person" object should represent the combined information about this person. For example, since an object with information about Joe Chung is exported from both `cs` and `whois`, `med` combines this information and exports the object of Figure 4.3.

4.1.1 Problems in Mediator Specification

Creating the integrated view from the wrapper views requires the resolution of a number of problems:

- *schema-domain mismatch*: The `whois` source represents names by a long string that contains both the first and the last name, while the `cs` database represents names using the "last_name" and "first_name" subobjects.

[1] Two minor points: (1) After translation, we have lost knowledge that objects at this source *must* have a regular structure. If this information is important to the applications, it could be exported as additional facts about this source. (2) One could consider it inefficient to repeat the schema in all objects, in this case where there is a regular pattern to objects. This problem can easily be addressed by data compression when objects are exported. Conceptually, we believe it is easier to think of each object as having its own labels.

```
<&e1,employee, set, {&f1,&l1,&t1,&rep1}>
     <&f1, first_name, string, 'Joe'>
     <&l1, last_name, string, 'Chung'>
     <&t1, title, string, 'professor'>
     <&rep1, reports_to, string, 'John Hennessy'>
<&e2,employee, set, {&f2,&l2,&t2}>
     <&f2, first_name, string, 'John'>
     <&l2, last_name, string, 'Hennessy'>
     <&t2, title, string, 'chairman'>
⋮
<&s3,student, set, {&f3,&l3,&y3}>
     <&f3, first_name, string, 'Pierre'>
     <&l3, last_name, string, 'Huyn'>
     <&y3, year, integer, 3>
⋮
```

Figure 4.1: The OEM object structure of the `cs` wrapper

```
<&p1,person, set, {&n1,&d1,&rel1,&elm1}>
     <&n1, name, string, 'Joe Chung'>
     <&d1, dept, string, 'CS'>
     <&rel1, relation, string, 'employee'>
     <&elm1, e_mail, string, 'chung@cs'>
<&p2,person, set, {&n2,&d2,&rel2}>
     <&n2, name, string, 'Nick Naive'>
     <&d2, dept, string, 'CS'>
     <&rel2, relation, string, 'student'>
     <&y2, year, integer, 3>
⋮
```

Figure 4.2: The OEM object structure of `whois`

- *schematic discrepancy*: Data in one database correspond to metadata of the other. In particular, the status of a person – `employee` or `student` – appears as a value in `whois` (it was part of a relational table), while it appears in the schema of `cs` (it was part of the relational schema).
- *schema evolution*: The format and contents of the sources may change over time, often without notification to the mediator implementor. For example, an attribute "birthday" may appear in either of the two sources, or the "e_mail" attribute may be dropped. We would like our mediator specification to be insensitive to as many of these changes as possible. For example, if "birthday" is included or dropped, it should be automatically included or dropped from the `med` view, without need to change the mediator specification.
- *structure irregularities:* Source `whois` does not follow a regular schema (i.e., it is a semistructured source.)

4.1.2 The Mediator Specification of `med`

The following MSL specification MS1 defines the `med` mediator we have described, resolving the integration problems we have discussed above. We will explain this specification in the paragraphs that follow.

(MS1) `Rules:`

```
<&cp1,cs_person, {&mn1,&mrel1,&t1,&rep1,&elm1}>
      <&mn1, name, string, 'Joe Chung'>
      <&mrel1, relation, string, 'employee'>
      <&t1, title, string, 'professor'>
      <&rep1, reports_to, string, 'John Hennessy'>
      <&elm1, e_mail, string, 'chung@cs'>
```

Figure 4.3: Object exported by `med`

```
<cs_person {<name N> <relation R> Rest1 Rest2}>@med
    :- <person {<name N> <dept 'CS'> <relation R> | Rest1}>@whois
       AND decompose_name(N, LN, FN)
       AND <R {<first_name FN> <last_name LN> | Rest2}>@cs
External:
decompose_name(string,string,string)(bound,free,free) impl by name_to_lnfn
decompose_name(string,string,string)(free,bound,bound) impl by lnfn_to_name
```

A specification consists of

1. rules that define the view provided by the mediator, and
2. declarations of functions that will be called upon for translating objects from one format to another.

Each rule (the above specification has only one rule) consists of a *head* and a *tail* that are separated by the `:-` symbol. The tail describes the patterns of objects that must be found at the sources, while the head describes the pattern of the top-level objects of the integrated view.

Intuitively, we may think of the process of "creating" the virtual objects of the mediator as pattern matching. First, we match the patterns that appear in the tail against the object structure of `cs` and `whois`, trying to bind the *variables* (represented by identifiers starting with a capital letter, such as `N`, `Rest1`, etc.) to object components of `cs` and `whois`. Then we use the bindings to "construct" the objects specified in the head of the rule.

The specification is based on patterns of the form <*object-id label type value*>, where we may place constants or variables in each position. For simplicity we can drop some of the fields when they are irrelevant. If one field is dropped, we assume it is the type, so we have a pattern of the form <*object-id label value*>. If two fields are dropped, we assume they are the type and the object-id. When the object-id is missing in a tail pattern, it means that we do not care about the object-id's appearing at the sources. When an object-id is missing from a head pattern, it means we do not care what object-id the mediator uses for the "generated" object.

When the *label* (*value*) field contains a constant, the pattern matches successfully only with OEM objects that have the same constant in their *label* (*value*) field. On the other hand, when the *label* (*value*) field contains a variable, the pattern can successfully match with any OEM object, regardless of the label (value) of the object. For example, the pattern `<name N>` can match with OEM objects `<&1, name, string, 'Fred'>` or `<&2, name, string, 'Tom'>`. As a result of a successful matching, the variable `N` will bind to the value of the specific OEM object (either `'Fred'` or `'Tom'` in the example).

Returning to our mediator specification example, we match the patterns of the tail against the top-level objects of the corresponding sources, trying to bind the variables of the tail to appropriate object components. In particular, we match the pattern

```
<person {<name N> <dept 'CS'> <relation R> | Rest1}>
```

against the objects of source `whois`, trying to bind the variables `N`, `R`, and `Rest1` to appropriate object components. That is, we try to find top-level "person" objects that have a "name" subobject, a "dept" subobject with value `'CS'`, and a "relation" subobject. The object identified by `&p1` (see Figure 4.2) satisfies

these requirements. As a result, `N` binds to `'Joe Chung'`, `R` binds to `'employee'`, and `Rest1` binds to the remaining subobjects, i.e., it binds to {`<&elm1, e_mail, string, 'chung@cs'>`} Let us name this set of bindings $b_{w,1}$. Other objects may also satisfy these conditions and produce other bindings for `N`, `R`, and `Rest1`. For instance, `N` can bind to `'Nick Naive'`, `R` to `'CS'`, and `Rest1` to {`<&y2, year, integer, 3>`}.

The specification also indicates that we match the pattern

```
<R {<first_name FN> <last_name LN> | Rest2}>
```

against the objects at source `cs`, obtaining bindings for the variables `R`, `FN`, `LN`, and `Rest2`. Referring to Figure 4.1, we see that one of these binding, call it $b_{c,1}$, will bind `R` to `'employee'`, `FN` to `'Joe'`, `LN` to `'Chung'`, and `Rest2` to {`<&t1, title, string, 'professor'> <&rep1, reports_to, string, 'John Hennessy'>`}.

The next step is to match the two sets of bindings. A binding $b_{w,i}$ from `whois` matches a binding $b_{c,i}$ from `cs` if the following conditions hold:

1. The two bindings agree on the values assigned to common variables, in this case, `R`.

2. The name `N` found in `whois` "corresponds" to the last name, first name pair `LN, FN` found in `cs`.

For example, we see that binding $b_{w,1}$ matches $b_{c,1}$ because they both bind `R` to `'employee'` and the name `N = 'Joe Chung'` corresponds to last name `LN = 'Chung'` and first name `FN = 'Joe'`.

4.1.3 External Predicates

The correspondence between names and first, last name pairs is given by the predicate

```
decompose_name(N,LN,FN)
```

Conceptually, we can think of `decompose_name` as a predicate that evaluates to true if `N` is a valid decomposition of last, first names `LN, FN`. In practice, `decompose_name` is implemented as a pair of functions, `name_to_lnfn` and `lnfn_to_name` (in principle written in any programming language), and defined in the mediator specification. For example, the line

```
decompose_name(string,string,string)(bound,free,free) implemented by name_to_lnfn
```

indicates that `name_to_lnfn` can be called with a full name (the first bound parameter); the function decomposes the name and returns the last and first names (second and third free parameters). Similarly, `lnfn_to_name` can compose a last, first name pair and produce a full name. Thus, operationally, to check if `decompose_name('Joe Chung', 'Chung', 'Joe')` is true, we can call `name_to_lnfn` with input parameter `'Joe Chung'` and see if it returns `'Joe'` and `'Chung'`. If it does, the predicate holds. Equivalently, we can call `lnfn_to_name` to perform the check.[2]

Having more than one function for `decompose_name` gives flexibility at execution time. For example, using `name_to_lnfn` we can first obtain `person` objects of `whois` and for each full name we find the first and last names and send a query to `cs` to find persons with this first name and last name. Using `lnfn_to_name` we can pass bindings in the opposite way — from to `cs` to `whois`.

4.1.4 Creation of the Virtual Objects

For each set of matching bindings from the tail patterns, we conceptually create an object in the `med` view.[3] (We stress that objects are not really materialized by the mediator specification.) The head of the rule tells

[2] Of course, if the implementor had provided a function `check_name_lnfn` that is called with all three parameters bound, we would simply call `check_name_lnfn` with input parameters `'Joe Chung'`, `'Chung'`, and `'Joe`.

[3] In reality, we first project the bindings of the variables of the tail, into bindings of the variables that appear in the head of the rule. Then we eliminate *duplicated* bindings, and finally we create an object of `med` for each set of bindings of the variables of the head.

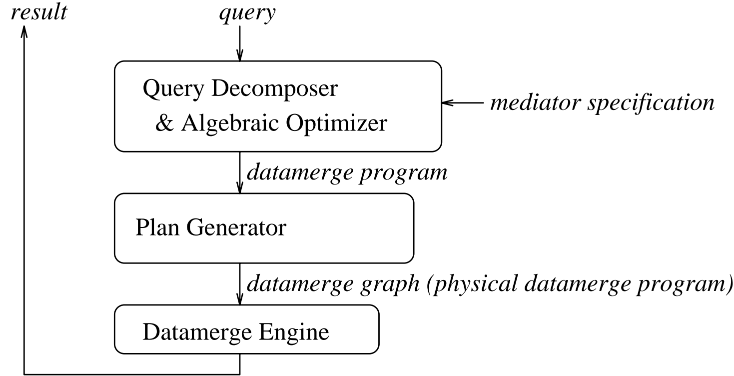

Figure 4.4: The basic architecture of MSI

us how to construct the view objects. For example, the matching bindings $b_{w,1}$ and $b_{c,1}$ result in the object of Figure 4.3.

Note that even though `Rest1` and `Rest2` are bound to sets of objects, and `<name N>` and `<relation R>` are bound to single objects, we can include all four inside the curly braces that define the subobjects for a "cs_person" object. In general, when variables that have been bound to sets appear inside curly braces {} in a rule head, the first level of their contents is "flattened out" and included in the set value that is described by the curly braces pattern.

Note also that our sample head did not specify any types or object-id's for the view objects. The types, of course, are simply set to the types of the bound variables (`string` in our case.) For the object-id's, any arbitrary unique strings can be used (e.g., `&cp1, &mn1, ...` are used in Figure 4.3.)

4.1.5 MSL's Solutions to Mediator Specification Problems

The specification of `med` solves the integration problems mentioned earlier, mainly by exploiting the free use of variables in the Mediator Specification Language, and the schema/data combination ability of OEM. For example, we were able simultaneously to bind variable `R` to a value in `whois` and a label in `cs`, thus addressing the schematic discrepancy. The schema evolution problem is handled by the use of variables `Rest1` and `Rest2`. If, say, new attributes such as "birthday" are added to `cs`, no change is required to the mediator specification. The new attribute will be included with `Rest1` and propagated to the integrated view. At the same time, the bindings of variables `Rest1` or `Rest2` are not required to carry homogeneous sets of objects. For example, binding $b_{w,1}$ binds `Rest1` to {`<&elm1, e_mail, string, 'chung@cs'>`} while $b_{w,2}$ binds `Rest1` to {}. In this way, MSL can handle the integration of unstructured sources that do not have a regular schema. Finally, the ability to use external predicates allows us to process atomic values in any desirable way.

One apparent limitation of the integrated view we have defined for `med` is that it only includes information for people that appear in both `cs` and `whois`. In particular, we may wish to include information in `med` even if it appears in a single source. This will be one of the fusion examples that we will demonstrate in Section 4.3. However, before doing so, in the following section we sketch how our Mediator Specification Interpreter (MSI) would process an incoming query against the sample definition we have given.

4.2 Introduction to Architecture and Implementation of MSI

The Mediator Specification Interpreter (the run time component of MedMaker) processes a query using a pipeline with the following three components (see Figure 4.4):

1. The *Query Decomposer and Algebraic Optimizer (QD&AO)* reads the query and the mediator specification and discovers which objects it must obtain from each source. Furthermore, it determines the conditions that the obtained source objects must satisfy.
2. The *cost-based optimizer* develops a plan for obtaining and combining the objects specified by the QD&AO. The plan specifies what queries will be sent to the sources, in what order they will be sent, and how the results of the queries will be combined in order to derive the result objects.
3. The *datamerge engine* executes the plan and produces the required result objects.

In the following subsection we use an example to overview MSI's query processing. The remaining subsections describe each MSI component.

4.2.1 Query Processing Overview

Let us assume that a client of mediator `med` wants to retrieve all the data for 'Joe Chung.' We use MSL (with one minor modification discussed below) as our query language.[4] The use of MSL simplifies our discussion, and furthermore, MSL makes a good query language because of its power and simplicity. Using MSL, our query can be expressed as:

```
(Q1) JC :- JC:<cs_person {<name 'Joe Chung'>}>@med
```

The object pattern (or object patterns) that appears in the tail of the query are matched against the object structure of `med` in exactly the same manner that tail patterns of MSL rules are matched against the sources. One new MSL feature that appears in the tail of our sample query is the *object variable* `JC`. The operator `:` indicates that `JC` must bind to "cs_person" objects that have a "name" subobject with value `'Joe Chung'`.

The query head indicates that every object that `JC` binds to is included in the result. Unlike mediator specification, when MSL is used for querying, the objects specified by the query rule head are materialized at the client.[5]

Query Decomposition Given our sample query, the QD&AO replaces the object pattern of the query tail with object patterns that refer to objects of the sources, thus deriving the datamerge rule R2:

```
(R2) <cs_person {<name 'Joe Chung'> <rel R>
                Rest1 Rest2}>
      :- <person {<name 'Joe Chung'> <dept 'CS'>
                 <relation R> | Rest1}>@whois
         AND decomp('Joe Chung', LN, FN)
         AND <R {<first_name FN> <last_name LN>
                 | Rest2}>@cs
```

Intuitively, the MSI derived the above rule by unifying the pattern `JC:<cs_person ...>@med` of the query tail against the head of the mediator specification rule of `med`.[6] After the unification, we generate a datamerge rule whose head is the head of the query and whose tail is the mediator specification rule's tail.

Execution Plan Now that the MSI knows what objects it has to find at the sources, the cost-based optimizer builds a physical datamerge program that specifies what queries should be sent to the sources, in what order they should be sent, and how the results of the queries should be combined in order to produce the query result. Here we informally describe a possible (and efficient) plan for our running example:

[4] The TSIMMIS project at Stanford is also exploring a different query language, called *LOREL*. It is an object-oriented extension to SQL and is oriented to the end-user. LOREL is described in [QRS+95]. MSL is more powerful than LOREL (e.g., MSL allows the specification of recursive views) and is targeted to mediator specification.

[5] Here we do not address the problem of materializing OEM objects at clients. The various issues and strategies have been discussed in Section 3.1.

[6] If there were several rules, the MSI would look for one or more matching rule heads. If more than one head matches, then more than one rule will be considered; resulting objects will be added to the result.

1. Bindings for the variables `R` and `Rest1` are obtained from the source `whois`. The bindings are obtained in two steps. First the following query is sent to `whois`:

   ```
   <bind_for_whois {<bind_for_R R>
                    <bind_for_Rest1 Rest1>}>
     :- <person {<name 'Joe Chung'> <dept 'CS'>
                <relation R> | Rest1}>@whois
   ```

   Labels `bind_for_whois`, `bind_for_R` and `bind_for_Rest1` are simply place-holders that allow the MSI to conveniently pick out the desired information from the returned result objects.

2. Bindings for `LN` and `FN` can be obtained from one of the `decomp` functions, i.e., from `name_to_lnfn`. We call it with bound parameter `N = 'Joe Chung'` and obtain `LN = 'Chung'` and `FN = 'Joe'`.

3. For each of the `R` binding of step (1), we combine it with the single binding of step (2), and submit a query to `cs` to obtain a binding for `Rest2`. For example, for the binding `R = 'employee'` we send the following query to `cs`:

   ```
   <bind_for_cs {<bind_for_Rest2 Rest2>}>
    :-<employee {<first_name 'Joe'>
                 <last_name 'Chung'>|Rest2}>@cs
   ```

4. Once MSI obtains bindings for `Rest2` as well, it generates objects that follow the pattern of the head of (R2). For example, considering the bindings we have illustrated so far, the MSI would generate the object of Figure 4.3.

4.2.2 Other MSL features

In this Section we present various MSL features that enhance the capabilities of the language.

First, the keyword `isatom` may precede an object pattern of the MSL rule tail, forcing the pattern to bind to atomic objects only.

Second, the pattern $*\langle variable \rangle$ may appear in the head of an MSL rule, in place of an object pattern or an object variable. For example, the rule

```
*JC :- <JC cs_person {<name 'Joe Chung'>}>@source
```

declares that the mediator exports, as top-level objects, the `source` objects identified by `*JC`. Intuitively, `*` acts as a dereferencing operator. If the pattern $*\langle variable \rangle$ appears nested inside an object pattern then the objects identified by $\langle variable \rangle$ become subobjects of the objects "created" by the pattern. For example, the rule

```
<JC cs_person {*R}> :- <JC cs_person {<R L V>}>@source
```

declares that every object identified by `R` must be exported as a subobject of a "cs_person" object identified by corresponding bindings of `JC`.

4.3 Object-Identity-Based Information Fusion

In this section we explain how object fusion can be achieved with semantic object ids. We start with a simple example that introduces semantic object-id's and demonstrates the basic principle of id based fusion. We then present examples that illustrate a variety of fusion operations. These examples are not exhaustive; they simply show how semantic ids can help in a variety of fusion scenarios.

4.3.1 A Simple Example

Let us consider a mediator called `s` that exports objects with label `techreport`. The `techreport` objects fuse information about reports that have the same report number and are exported by the sources `s1` and `s2`. In particular, if source `s1` contains a report and its title, the exported `techreport` object contains the corresponding `title`. If source `s2` contains the postscript for the report, then a `postscript` subobject is also included in the `techreport`. Note, the specification of the `techreport` object appears in two rules. Each rule describes the contribution of only one of the sources.

```
(MS3) (R3.1) <trep(RN) techreport {<title T>}>@s :-
                       <report {<report_num RN> <title T>}>@s1
      (R3.2) <trep(RN) techreport {<postscript P>}>@s :-
                       <report {<report_num RN> <postscript P>}>@s2
```

The first rule declares that:

- **if** there is a pair of *bindings* t and r for variables `T` and `RN` (variables are identifiers starting with a capital letter) such that `s1` contains a `report` top-level object that has a `report_num` subobject with value r and a `title` subobject with value t,
- **then** mediator `s` exports a `techreport` object, with object-id `trep(`r`)`, that has a `title` subobject with value t and a unique system-generated object-id.

The semantics of the second rule are defined accordingly. Notice how `techreport` objects at the mediator are assigned the semantic object id `trep(RN)`. (We add the function symbol `trep` to the report number obtained from the source to uniquely identify how this id was generated.) Observe that (MS3.1) does not prevent the `techreport` with object-id `trep(`r`)` from having subobjects other than `title`, thus allowing the second rule to add more subobjects to the same `techreport` objects. In general this is how object fusion is achieved: MSL allows rules to incrementally and independently insert information into a semantically identified mediator object. In the examples that follow we will show how this feature provides significant power and flexibility to mediator specifications. Incidentally, note that the simplicity of OEM facilitates such id based fusion. In particular, if objects had rigid schemas it would not be as natural to combine object fragments. If more than one sources contribute subobjects with same label we would have to introduce multiple tuples for the same object. Furthermore, we avoid generating NULL values as it would be done in an SQL system.

4.3.2 Merging Information with Incomplete Knowledge of the Source Contents

It is not necessary to know the structure of the source reports in order to fuse them. Specification (MS4) demonstrates that we can group all information about reports into `techreport` objects, without knowing the structure and contents of the reports subobjects.

```
(MS4) (R4.1) <trep(RN) techreport V>@all :- <report V:{<report_num RN>}>@s1
      (R4.2) <trep(RN) techreport V>@all :- <rep V:{<report_num RN>}>@s2
```

Variable `V` binds to set values that contain all subobjects of `report` provided that at least one of the subobjects has the label `report_num`. Then, every object of the set value becomes a subobject of the `techreport`, regardless of whether the other source also provides the same piece of information.

Note, OEM provides the flexibility to integrate information without having to worry about the presence of subobjects with same label. In some cases this may be desirable. For instance, say each source contains a different `title` for the same report. We may want to record these two potentially different titles in the fused object. In other cases, however, we may wish to eliminate one of the titles. We will show how this can be done in Section 4.3.5. Fortunately, the OEM model does not force a decision on us: the person writing the mediator specification can decide if redundancies or inconsistencies are allowed.

4.3.3 Removing Redundancies

The example of Section 4.3.2 does generate one redundancy that is not very useful: each techreport object contains two report_num subobjects with identical values but different object-id's. This redundancy can be eliminated as shown by mediator (MS5). It assigns the semantic object-id rnOID(RN) to the report_num subobjects with value RN. In this way, the report_num subobjects that have the same value are assigned the same object-id and hence they degenerate into the same report_num object.

```
(MS5) (R5.1) <trep(RN) techreport {<rnOID(RN) report_num RN> <O1 L1 X1>}>@nored :-
                 <report {<report_num RN> <O1 L1 X1>}>@s1 AND NOT L1=report_num
      (R5.2) <trep(RN) techreport {<rnOID(RN) report_num RN> <O2 L2 X2>}>@nored :-
                 <report {<report_num RN> <O2 L2 X2>}>@s2 AND NOT L2=report_num
```

Note, the variables L1 and L2 that appear in label positions allow the patterns <O1 L1 X1> and <O2 L2 X2> to match with any subobject of the reports of s1 and s2, provided that L1 and L2 are not equal to report_num. Then, the subobjects that are bound to <O1 L1 X1> or <O2 L2 X2> become subobjects of the mediator techreports. (If we did not have explicit NOT conditions the pattern <O1 L1 X1> and <O2 L2 X2> would also match with report_num objects.)

Comparison of object-id based fusion with outerjoin Outerjoin has also been suggested as a way to join information from sources that may or may not contribute to the joined object. Using outerjoin we could, in a single rule, create a techreport virtual object with report number r if there is a report with number r at s1 or s2. However, we believe that the object-id based fusion scheme we illustrated above is more powerful. In particular, with object-id based fusion we can easily join objects from the same source. The need for this arises if, for example, s1 has multiple report objects that refer to the same real-world report. To do the same with outerjoin, we would have to know the maximum number of outerjoins that we may need to apply, something that is data-dependent. Furthermore, object-id based fusion is a more modular solution: If we want to add one more source we simply introduce one more rule.

4.3.4 Blocking Sources and Resolving Inconsistencies

More than one source may offer information about the same real world entity. If all sources offer roughly the same information we may want to avoid retrieving information about an entity from some source(s) if some other source provides us enough information about this entity. Information sources that charge their users make this scenario particularly important; if we can retrieve enough information from some "cheap" source, we want to avoid retrieving similar information from an "expensive" source. In this section we show specifications where the presence of some data "blocks" the retrieval of other data. In the next subsection we show that MSL's flexibility allows blocking at various levels of granularity, from blocking entire objects to selectively blocking subobjects that meet various conditions.

As our example, assume that source s1 can be accessed for free whereas s2 charges a fee for providing information. In this case, we may wish to have mediator save, defined by (MS6), that collects from s2 only information about reports that do not appear in s1.

```
(MS6) (R6.1) <trep(RN) techreport V>@save :- <report V:{<report_num RN>}>@s1
      (R6.2) provides(RN) :- <report {<report_num RN>}>@s1
      (R6.3) <trep(RN) techreport V>@save :-
                 NOT provides(RN) AND <report V:{<report_num RN>}>@s2
```

Rule (R6.1) declares that every report of s1 becomes a techreport of save. Then (R6.2) collects in *relation* provides the report numbers RN of all reports that come from s1. In general, MSL specifications may define and use relations that serve as "intermediate" results. We could as well use OEM objects for storing intermediate results, but we believe that sometimes the use of relations makes the specification clearer.

Finally, (R6.3) exports a `techreport` for every `report` of `s2` unless the report appears in the relation `provides`. Note, we use traditional "negation by failure" semantics. In effect, the relation `provides` prevents (or blocks) `s2` from exporting a report via the third rule if the "same" report has been exported by `s1` via the first rule. In Section 5.6.3 we demonstrate techniques used by the query optimizer that prevent the mediator from retrieving "blocked" data from the wrappers.

There are many variations to the blocking scheme of (MS6). Just to illustrate one, let us assume that if `s1` does not provide `author` and `title` for a report then we retrieve this report from `s2` also. In this case, all we have to do is replace (R6.2) with the following rule.

```
provides(RN) :- <report {<report_num RN> <title T> <author A>}>@s1
```

Calculated Priorities So far, we have assigned priorities in a static way. For example, source `s1` has priority over `s2`. However, the computing power of MSL allows the expression of arbitrarily complex blocking schemes. For example, we may assign priorities to the various pieces of information in a dynamic way. The implementor may provide an *external predicate* called `calc` that calculates how credible are the pieces of information provided by each source. (We have described in [PGMU96] how external predicates interface with MSL.) To illustrate this, let us assume that reports with most recent `date` subobject are given the highest priority, i.e., if there are multiple `report` objects that refer to the same report we retain only the `report` with the most recent date. The predicate `calc` is given the value of the `date` subobject and returns an integer that is the priority `P` of the specific piece of information.

```
(MS7) (R7.1) <trep(RN) techreport V>@med :- <O report V:{<report_num RN>}>@s1
                                            AND NOT notbest(O,RN)
      (R7.2) <trep(RN) techreport V>@med :- <O report V:{<report_num RN>}>@s2
                                            AND NOT notbest(O,RN)
      (R7.3) provides(O,RN,P) :- <O report {<report_num RN> <date D>}>@s1
                                 AND calc(D,P)
      (R7.4) provides(O,RN,P) :- <O report {<report_num RN> <date D>}>@s2
                                 AND calc(D,P)
      (R7.5) notbest(O1,RN) :- provides(O1,RN,P1) AND provides(O2,RN,P2) AND P1<P2
```

If relation `provides(O,RN,P)` contains the tuple (o, r, p), then there is a `report` object identified by o (the object-id indicates whether it comes from `s1` or `s2`) that has report number r and priority p. If `notbest(O1,RN)` contains a tuple (o_1, r) then there is a report object identified by o_2 that has the same report number r with o_1 and greater priority p_2 than the priority p_1 of o_1. Hence, o_1 is not the most credible `report` object for the specific report. Rules (R7.1) and (R7.2) do not retrieve `report` objects whose object-id's `O` appear in the `notbest` relation.

4.3.5 Removing Inconsistencies Using Fine-Grained Blocking

In Section 4.3 we showed that specifications such as (MS3) may cause the same `techreport` to have multiple `title` objects. In this section we show that using negation and label variables we may block subobjects that come from one source (presumably the less reliable source) in favor of subobjects that come from the other source (the more reliable). In effect, we use fine-grained blocking, i.e., blocking where we individually access each subobject (using label variables) and decide whether it must be blocked or not.

For example, (MS8) resolves all inconsistencies in favor of `s1`, i.e., if `s1` provides some report subobject with label `F`, then `s2` should not provide a subobject with the same label. Note, in this example we assume that no report has two subobjects with the same label and different values.[7]

```
(MS8 (R8.1) <trep(RN) techreport {<field(RN,F) F V>}>@med :-
                <report {<report_num RN> <F V>}>@s1
```

[7] In [PGM] we generalize MSL to handle the case where multiple subobjects with the same label exist.

```
(R8.2) provides(RN,F) :- <report {<report_num RN> <F V>}>@s1
(R8.3) <trep(RN) report {<field(RN,F) F V>}>@med :-
           NOT provides(RN,F) AND <report {<report_num RN> <F V>}>@s2
```

The subgoal NOT provides(RN,F) blocks (R8.3) from exporting any subobject with label f of a report identified by r if the tuple (r, f) is in provides, i.e., if data about the f subobject of the report with number r can be found in s1.

4.3.6 Handling References

When we import objects from sources and fuse them into mediator objects we must be careful with the object references that are imported. For example, assume that reports stored in s1 have references to related reports, also stored in s1. From an OEM point of view, each report contains a subobject related, the value of which is a set containing the s1 object ids of the referenced reports.[8] If we are not careful when we import related into the mediator, we will end up with object references that point to the original objects of s1 and not to the corresponding fused techreport objects.

In this section we show two ways to resolve this problem. The first solution is more efficient but assumes that we know which are the subobjects that contain references to fused objects (the subobject related in our example.) The second one is less efficient but it works even if we do not know which objects contain references. The latter solution is very useful when we integrate structures that are deeply nested and we do not have complete information about their structure (as is the case with World-Wide-Web).

The first solution is implemented by (MS9). Rule (R9.1) puts in the techreport objects all information of the source reports with the exception of the related subobject. Rule (R9.2) creates a related object and inserts it into the appropriate techreport. For simplicity we omit the corresponding rules for s2.

```
(MS9) (R9.1) <trep(RN) techreport {<L X>}>@all-with-ref :-
              <report {<report_num RN> <L X>}>@s1 AND L!=related
      (R9.2) <trep(RN) techreport {<related {<trep(REL) techreport {}>}>}>@s1
              <report {<report_num RN> <related {<report {<report_num REL>}>}>}>@s1
```

Our second solution does not rely on knowing what subobjects may refer to s1 objects that are fused. The basic idea is to create two virtual objects for each techreport. The first virtual object (as before) has the id trep(RN) and its related subobject contains s1 object-ids. The second mediator object contains the same information except that its object-id is identical to the object-id in s1. The first copy is needed for fusion, since its semantic id is used to combine fragments from other sources. The second copy is simply used so that ids in the first are to valid mediator objects.

```
(MS10) (R10.1) <trep(RN) techreport {<L X>}>@all-with-ref :-
                   <report {<report_num RN> <L X>}>@s1
       (R10.2) <O techreport V>@all-with-ref :-
                     <techreport V:{ <report_num RN> }>@all-with-ref
                     AND <O report {<report_num RN>}>@s1
```

The first rule (and the analogous one for s2 that is not shown) generates the first copy of each techreport fused object. (Note that these objects contain s1 ids.) The second rule generates the copy objects and simply changes the id. If fused objects are expected to contain s2 ids, then another rule would be needed to generate virtual copies with s2 ids. Note that we only create copies of the top-level techreport objects; these "reuse" the same subobjects, such as title. Furthermore, the copies are virtual and hence not materialized at the mediator unless necessary.

[8] OEM allows top-level objects to be subobjects as well.

4.3.7 Other Types of Fusion

We have only presented a few representative fusion examples. There are of course many others. In closing this section we briefly comment on some interesting cases.

- *Fusion with Canonical Forms.* In our examples we assumed that source objects had some semantic key (like `report_num`) that could be used for fusion. Often keys exist but are represented differently at sources. As a trivial example, phone number may uniquely identify customers, but one source may represent a phone number as (415) 555-1111 whereas another source may represent the same number as +1.415.555.1111. In these cases, key values can be mapped (via external predicates) to a canonical form that can then be used to form the semantic id.
- *Fusion with No Keys.* Often, when dealing with heterogeneous and autonomous sources, objects have no well defined keys. So, to decide if two customer records represent the same person, we need to apply a complex function that compares their names, addresses, and phone numbers, say. The output is not a canonical key, but simply a fused record that somehow combines the information. We can use MSL (and external predicates) to define this type of fusion, but it introduces many problems that are beyond the scope of this thesis. Just to mention one, the fusion process can have an unbounded number of steps, each quite expensive. That is, after we fuse two customer records, we have generated a new record. Now this record must be compared against all other customer records for a potential match, generating even more records.
- *Complex Fusion.* When fusing fragments into a single mediator object, we have used relatively simple schemes to combine the data, for example, selecting one `title` over another. It is of course possible to have more complex functions. For instance, if each fragment contains a `temperature` subobject, we could compute an average temperature for the fused object. Such functions involve aggregation and are not considered by MSL, which is a purely conjunctive language.

As a final comment on MSL, we stress that MSL is not a language for the end-user. It is a language for succinctly describing mediators using very few primitives.

4.4 Introduction to Semantics of MSL Specifications

In this section we present the model-theoretic semantics of MSL. Besides its usefulness in formally defining the meaning of mediator specifications and queries, they also show the connection between MSL and first-order logic and they provide insight into the implementation of MedMaker that will be discussed in the next chapter.

The formal semantics is based on the intuition that OEM and MSL are essentially a form of logic that has been appropriately reformulated in order to express object nesting, object identity, and the combination of schema and data information. We substantiate this intuition by reducing OEM data and MSL mediator specifications[9] and queries into *first-order* formulas. In particular, we first describe how OEM objects can be reduced to an equivalent *relational representation* that is based on first-normal-form relations. Then, we show how mediator specifications can be reduced to datalog rules that contain function symbols. The minimal model of the datalog rules describes the relational representation of the object structure that is exported by the mediator. The same idea applies to the specification of query semantics, since we can view queries as mediator specifications that define the query result.

The transformation of an arbitrary MSL specification to a logic program is accomplished in two steps:

1. we reduce every MSL rule into one or more *normal-form* MSL rules, that is, MSL rules conforming to the following restrictions:

[9] We assume that the MSL specification satisfies the obvious safety and typing constraints, such as “no object variable can appear where a non-object variable is expected.” A complete list of the safety and typing restrictions appears in Appendix B.0.1.

(a) they contain only 3-field patterns that explicitly specify the object-id, label, and value of the source and mediator objects. (In accordance with the simplifying assumption set forth in Chapter 3 we omit the type field.)

(b) they do not contain object variables or rest variables such as the variables `Rest1` and `Rest2` of `med`'s specification (see Section 4.1, page 4.1.2).

A formal syntax of MSL normal form appears in Appendix A.

2. we reduce every normal-form MSL rule into one or more datalog rules.

Section 4.4.1 provides the rules for computing the relational representation of the object structure exported by an OEM source (i.e., translator or mediator). Section 4.4.3 illustrates the reduction of a normal-form MSL rule to datalog rules. Section 4.4.4 illustrates the reduction of an MSL rule to a normal-form MSL rule. The full algorithm for reducing a MSL rule into a normal-form MSL rule will be presented in Section 5.5 because it is actually used by MedMaker.

4.4.1 Relational Representation of OEM Objects

We can represent the OEM object structure that is exported by an OEM source using three relations, named *top*, *object*, and *member*. For example, Figure 4.5 demonstrates the relational representation of part of the object structure of translator `cs` (see Figure 4.1 for the corresponding part of the object structure). The relational representation of an object structure can be derived mechanically by the following three straightforward rules:

1. If at the source s there is an object with object-id oid, label l, and atomic value (i.e., non-set value) v, then we introduce a tuple

$$object(s, oid, l, v)$$

2. If at the source s there is an object o with object-id oid, label l, and a set value that consists of the objects $o_1, o_2, \ldots, o_n$ that are identified by $oid_1, oid_2, \ldots, oid_n$ then we introduce the tuples

$$\begin{array}{l} object(s, oid, l, set) \\ member(s, oid, oid_1) \\ member(s, oid, oid_2) \\ \quad\vdots \\ member(s, oid, oid_n) \end{array}$$

where *set* is a special value indicating that the object identified by oid is a set object.

3. If at the source s the object identified by oid is a top-level object, we introduce the tuple

$$top(s, oid)$$

We treat structured object-id's in the same way that we treat any object-id. For example, if at source `src` the object identified by `h1(a52,'Smith')` is a member of the object identified by `h2('Jones')` we introduce the tuple:

$$member(\texttt{src}, \texttt{h2}('\texttt{Jones}'), \texttt{h1}(\texttt{a52}, '\texttt{Smith}'))$$

The above rules can be used in reverse as well, i.e., given the relational representation of a set of sources we can use the above rules to derive the OEM object structures of the sources. Note that the relational

top

SRC	*OID*
cs	&e1
cs	&e2
cs	&s3
⋮	

object

SRC	*OID*	*Label*	*Value*
cs	&e1	employee	*set*
cs	&f1	first_name	'Joe'
cs	&l1	last_name	'Chung'
cs	&t1	title	'professor'
cs	&rep1	report_to	'John Hennessy'
cs	&e2	employee	*set*
cs	&f2	first_name	'John
cs	&l2	last_name	Hennessy'
cs	&t2	title	'chairman'
⋮			

member

SRC	*OID*	*SubOID*
cs	&e1	&f1
cs	&e1	&l1
cs	&e1	&t1
cs	&e1	&rep1
cs	&e2	&f2
cs	&e2	&l2
cs	&e2	&t2
⋮		

Figure 4.5: Relational representation of `cs`'s exported object structure

representation must satisfy obvious constraints (fully described in Subsection 4.4.2) such as the constraint that the existence of a tuple

$$member(s, oid, oid_1)$$

implies the existence of a tuple

$$object(s, oid, l, set, set)$$

for some label l. Indeed, the mediator specification must be such that for every query Q the relational representation of the answer satisfies the constraints for being reducible to an OEM answer object. If the required constraints are not satisfied the system informs the user of the constraint violation. External predicates and predicates exported by the sources and/or the mediator can also be incorporated in the relational representation of sources.

4.4.2 Constraints on the Relational Representation of an OEM Object Structure

The *top*, *member*, and *object* relations that represent an OEM object structure must satisfy the following constraints:

1. The "source" and "object-id" attributes of the *object* relation constitute a key for the tuples of *object*.
2. If there is a tuple
$$member(s, oid, oid_1)$$
there must also be a tuple
$$object(s, oid, l, set)$$
3. If there is a tuple
$$object(s, oid, l, c)$$
where $c \neq set$, then there is no tuple
$$member(s, oid, oid_1)$$

4. If there is a tuple

$$member(s, oid, oid_1)$$

there is also a tuple

$$object(s, oid_1, l, v)$$

4.4.3 Example of Reducing Normal-Form MSL Rules to Logic Rules

Now we illustrate how a normal-form mediator specification rule can be reduced to a Datalog rule such that the relational representation of the OEM structure created by the MSL rule is identical to the relations computed by the Datalog rule.

For example, consider the following specification of mediator `simple` that for every top-level object of source `src`, with object-id `O` and label `L`, exports a "top-label" object whose value is the binding of `L` and whose object-id is the binding of `O`. Note that the following rule happens to be a normal-form rule since it uses only 3-field object patterns, does not contain object or rest variables, and the value variable `V` appears only once.

```
<O top_label L>@simple :- <O L V>@src        (1)
```

The "informal" semantics of MSL described in Section 4.1 requires that first we obtain bindings for the variables that are used in the head of the MSL rule. We formalize in logic our notion of obtaining bindings for `L` and `O` by introducing a predicate *bind* that "collects" valid pairs of bindings for the pair `L` and `O`. In general, *bind* collects bindings for the variables that appear in the head of the MSL rule.

A logic rule that corresponds to the tail of the MSL rule defines the tuples of *bind*:

$$bind(O, L) : -top(\texttt{src}, O) \wedge object(\texttt{src}, O, L, V)$$

The tail of the above logic rule was created by applying the following two straightforward principles:

- if the MSL pattern

$$<\langle oid\rangle\ \langle label\rangle\ \langle value\rangle>$$

  appearing in the MSL rule tail refers to source $\langle src\rangle$,[10] and $\langle value\rangle$ is either a variable or a constant, we put in the tail of the datalog rule the *literal*

$$object(\langle src\rangle, \langle oid\rangle, \langle label\rangle, \langle value\rangle)$$

- if the MSL pattern

$$<\langle oid\rangle \ldots> @\langle src\rangle$$

  appears in the MSL rule tail, we put in the tail of the logic rule's tail the *literal*

$$top(\langle src\rangle, \langle oid\rangle)$$

Going back to the running example, now that we have obtained bindings for the variables that appear in the head, we write the rules that create the (relational representation) of the object structure of `simple`.

$$top(\texttt{simple}, O) : -bind(O, L)$$
$$object(\texttt{simple}, O, \texttt{top_label}, L) : -bind(O, L)$$

The above rules were created by applying two straightforward principles for reducing the MSL rule head in datalog rules:

[10] Note, the pattern either is followed by a `@`$\langle src\rangle$ or is nested inside a pattern that is followed by a `@`$\langle src\rangle$

- If the MSL pattern

$$<\langle oid\rangle\ \langle label\rangle\ \langle value\rangle>$$

appears in the head of the mediator specification of $\langle medname\rangle$, and $\langle value\rangle$ binds to constants only, we introduce the logic rule[11]

$$object(\langle medname\rangle, \langle oid\rangle, \langle label\rangle, \langle value\rangle) :- bind(\ldots)$$

Note, the arguments of the predicate *bind* are the variables that appear in the head of the MSL rule.

- If the top-level object

$$<\langle oid\rangle \ldots>$$

appears in the head of the mediator specification of $\langle medname\rangle$, then we introduce the datalog rule

$$top(\langle medname\rangle, \langle oid\rangle) :- bind(\ldots)$$

4.4.4 Example of Reducing MSL Rules to Normal-Form

Whenever a 2-field pattern appears at an MSL rule we have to replace it with a 3-field pattern. Let us consider the following mediator `fl` that exports an object "top_label" for every label of a top-level object that it finds at source `src`.

```
<top_label L>@fl :- <O L V>@src
```

The above rule is transformed into the normal-form MSL rule

```
<G top_label L> :- <O L V>@src AND genoid_{L→G}(L,G)
```

The type of the generated "top_label" objects is `string` because the values of the "top_label" objects come from labels. The object-id field is filled with a variable `G`. Then, for every valid binding of the variable `L` we have to generate one binding of `G`. This is accomplished by the predicate $genoid_{L \rightarrow G}$ that appears above. The predicate $genoid_{L \rightarrow G}(l, g)$ associates a unique g with each l.

Indeed MedMaker actually converts the MSL rules and the query into normal form before it matches them. Section 5.5 discusses the details of the conversion.

4.5 Related Work

We described in Chapters 1 and 3 the OEM features that make it suitable for integration of heterogeneous information systems. Because of OEM, MSL mediator specifications tend to be short and simple and avoid questions such as "what is the class of the view objects?", that complicate object-oriented view definition [Rud92, AB91]. In spite of its simplicity, MSL is quite powerful. For instance, it allows the construction of arbitrarily complex object structures (which XSQL [KKS92] does not) and allows creation and manipulation of object-id's (ODMG's query language [Cat94], for instance, does not allow explicit creation and manipulation of object-id's).

[11] The normal-form assumption guarantees that $\langle value\rangle$ does not bind to a set of objects, and thus it can be safely used in the Datalog rule.

MSL and OEM can be seen as a form of first-order logic. Indeed, we borrow many concepts from logic oriented languages such as datalog [Ull88, Ull89], HiLog [CKW93], O-Logic [Mai86], and F-Logic [KL89]. HiLog first proposed – under a logic framework – the idea of mixing schema and data information.[12] It also demonstrated the reduction of nested structures into first-order structures, a feature of OEM and MSL that is illustrated in [PGM]. Nevertheless, we should not see MSL as a front-end to a datalog system (the next Chapter will show the extra difficulties.)

Recently SchemaSQL [LSS96] extended SQL so that data and schema information can be simoultaneously accessed and manipulated. It is interesting to note that, unlike SchemaSQL, the simplicity of OEM and MSL allow the simoultaneous access of schema and data by introducing a single feature: variables in the label fields. (Of course this does not reduce the value of SchemaSQL, which solves the problem while being compatible with SQL.)

MSL's handling of semantic object-ids is based on a particular use of Skolem functions as first introduced in object-oriented systems in [Mai86] and refined in [KKS92, CKW93, KL89]. Automatic creation and manipulation of object-id's based on Skolem functors are considered in depth in [HY90]. It is observed in [AK89] that object-id based set formation (as provided by the object-id based fusion) can replace explicit (LDL-like [NT88]) grouping operators. They also advocate a *ptime* sublanguage by prohibiting recursion through object creation. The architecture we use prevents such a potentially dangerous/expensive form of recursion: objects are created in a mediator based on the objects in lower level sources (that can themselves be mediators).

A very important difference between MedMaker and other integration systems is that MedMaker can integrate conventional well-structured databases that have a static schema and at the same time can integrate sources that do not have a regular schema, or sources that have an often-changing schema. The ability to integrate all kinds of sources is due to:

1. OEM's absence of schema, which allows the intuitive representation of heterogeneous, semistructured, and changing information.

2. MSL's ability to exploit regularities and complete knowledge of the schema (the example of Section 5.1.1 demonstrated the tradeoff between performance and partial knowledge of the schema).

Though systems that integrate well-structured conventional databases exist (e.g., [A+91, K+93, BLN86, LMR90, T+90, Gup89, FLNS88, ACHK93]) and recently systems for the integration of sources with minimal structure have also appeared [Fre, RJR94, S+93], we do not know of view definition based systems ([A+91, Ber91, CWN94, FLNS88] and others) that handle the whole spectrum of information sources simultaneously, and with MSL's flexibility.

Note, MedMaker performs integration by "working" with the structures of the source objects. Semantic information is effectively encoded in the MSL rules that do the integration. There are many projects that follow MedMaker's "structural" approach [Ber91, DH86, B+86], as well as many projects that follow a semantic approach [HM93, H+92]. We believe that the power of the structural approach, along with the flexibility, generality, and conciseness of OEM and MSL make the "structural" approach a better candidate for the integration of widely heterogeneous and semistructured information sources.

[12] [KLK91] has also proposed an interesting mixing of schema and data information for the relational data model.

Chapter 5

Implementation and Algorithms of MedMaker

MedMaker's architecture (Figure 5.1) was informally introduced in Section 4.2. In this chapter, we discuss in detail each component of the architecture and its role in query processing.

For readability, we start by explaining query processing for the relatively simpler case of non-fusion mediators. For these mediators we do not need the *normalizer* module that appears on the top of our architecture. We first explain how the *Query Decomposition and Algebraic Optimization (QD&AO)* module matches each query condition with each rule and generates a datamerge program, i.e., a rewriting of the query that refers directly to the wrapper objects instead of the mediator objects.

The *optimizer* converts the datamerge programs into executable plans, called *datamerge graphs*, that specify the queries that will be sent to the sources in order to retrieve the objects required by the datamerge program, the merging of query results, and so on. Finally, the *datamerge engine* executes the datamerge graphs.

Query processing for fusion mediators poses the extra difficulty that multiple rules may "contribute" to the same mediator object. This property significantly complicates the condition/rule matching. The implemented solution consists of two steps: First, the normalizer reduces the query and the mediator specification into normal form (recall Section 4.4.) Using the normal form, the QD&AO implements the query condition/rule matching process as an extension of resolution and unification in first order logic.

In Section 5.1 we informally describe the QD&AO algorithms for specifications that do not involve object fusion. Section 5.2 describes the structure of the datamerge graphs and the datamerge engine. Section 5.3 describes the cost-based optimizer. Section 5.4 describes the additional challenges posed by object fusion and formally presents the unification algorithm employed for processing normal form queries and conditions. Section 5.5 formally describes the reduction of MSL into normal form. Section 5.6 describes a set of optimizations that are specific to the fusion problem. Finally, Section 5.7 compares MedMaker's query processing techniques to related query processing and optimization technologies.

5.1 Query Decomposition and Algebraic Optimization

The QD&AO matches the query against the mediator specification rules and rewrites the query so that references to the virtual mediator objects are replaced by references to source objects. The result is a *logical datamerge program* that is a set of MSL rules specifying the result. In Section 4.2.1 we illustrated the view expansion and algebraic optimization process. There, expression Q2 was the logical datamerge program. In the rest of this section we explain the QD&AO process in more detail. In general, the QD&AO formulates the logical datamerge programs in a sequence of the following steps:

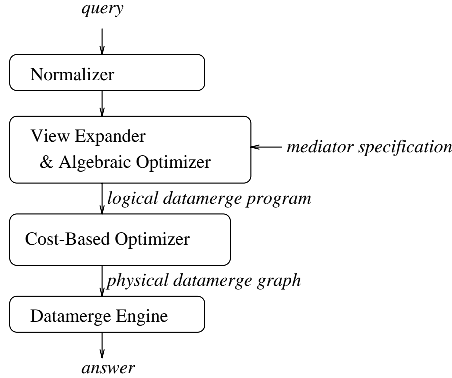

Figure 5.1: The full architecture of MSI

1. First it matches a query tail condition c with specification rule heads. The successful matches result in expressions called *unifiers*. Intuitively, our unifiers describe the match between the condition and the rule, the conditions that must be pushed to the sources, and other information necessary for the rewriting of the query. Note, they can be viewed as extensions of the unifiers used in resolution of first order clauses [GN88].
2. Then for every unifier a logical datamerge rule is formed. The rule head is formed by applying the unifier to the query head, while the datamerge rule tail is formed by
   (a) substituting the condition with the specification rule tail, and
   (b) applying the unifier to the new query condition
3. Steps 1 and 2 are repeated until no condition that refers to the mediator is left. The resulting rules are called *logical datamerge rules*. Their union describes the result of the query without referring to the mediator's virtual objects.

In the absence of recursion this process terminates. In the presence of recursion more complex resolution strategies and termination criteria are required [GN88] (not implemented yet.) Also, note that the matching of query conditions with rules corresponds to resolution of Horn clauses, whereas the unifiers that we use are extensions of unifiers of first-order clauses.

For example, consider the query Q1 (Section 4.2.1) and the specification MS1 of `med` which are repeated below:

```
(MS1) <cs_person {<name N> <relation R> Rest1 Rest2}>@med
          :- person {<name N> <relation R> | Rest1}>@whois
             AND decompose_name(N, LN, FN)
             AND <R {<first_name FN> <last_name LN> | Rest2}>@cs
```

```
(Q1) JC :- JC:<cs_person {<name 'Joe Chung'>}>@med.
```

The match[1] results in the unifier θ_1 where

$$\theta_1 = \left[\begin{array}{l} \texttt{N} \mapsto \texttt{'Joe Chung'}, \\ \texttt{JC} \mapsto < \texttt{cs_person}\ \{< \texttt{name 'Joe Chung'} > < \texttt{relation R} > \ \texttt{Rest1 Rest2}\} > \end{array} \right]$$

[1]Before we match a query with one or more rules we must rename the variables that appear in the query and the rules, so that no two rules, or a query and a rule have identically named variables.

The above unifier consists of two *mappings*. The application of θ_1 to the query head causes the substitution of `JC` by the structure following the $\Rightarrow$. Similarly, the application of θ_1 to the mediator rule tail causes the substitution of `N` by `'Joe Chung'`. Combining the transformed query head with the transformed mediator rule tail we obtain the logical datamerge rule Q2.

In general, a unifier may contain any number of mappings and/or definitions. When the QD&AO matches a query condition with a rule head it generates all unifiers θ such that

1. If we apply the mappings to the query condition and the mediator rule head, the transformed query condition pattern is *contained* in the rule head pattern. In the example, the transformed query condition `<cs_person {<name 'Joe Chung'>}>`[2] is contained in the transformed rule head `<cs_person {<name 'Joe Chung'> <relation R> Rest1 Rest2}>` because they have the same label `cs_person` and every subobject pattern of the query condition pattern (i.e., the pattern `<name 'Joe Chung'>`) is identical to a subobject pattern of the rule head. Containment guarantees that any mediator object generated by the transformed rule satisfies the query condition pattern.

2. There is a mapping for every object variable, "rest" variable, and value variable that appears in the head of the query. The mapping provides the expression that will replace the variable in the datamerge rule. For example the mapping of `JC` indicates that the object variable `JC` must be replaced with the structure `<cs_person {<name 'Joe Chung'> <relation R> Rest1 Rest2}>`.

Note, QD&AO does not really generate all possible unifiers, since some of them are trivial rewritings of other unifiers. For example, if we have a unifier $\theta = [\texttt{V1} \mapsto \texttt{V2}]$ it is useless to generate a unifier $\theta' = [\texttt{V2} \mapsto \texttt{V1}]$. The latter one will generate exactly the same datamerge rules with the former one module variable renaming.

5.1.1 Pushing Conditions to the Sources

The QD&AO pushes conditions such as "the name must equal `'Joe Chung'`" to the corresponding source. Indeed, QD&AO pushes to the sources all conditions that can be pushed, thus implementing the (well-known in relational DB's) "push selections down" algebraic optimization. In our environment with nested objects that may have unknown structure, algebraic optimization is substantially more challenging than in a relational environment. To illustrate this point, assume that the following query, which retrieves the data of 3rd year students, is sent to mediator `med` (specified by MS1):

```
S :- S:<cs_person {<year 3>}>@med
```

Mediator `med` joins data from two sources, and we cannot tell in advance whether the "year" object comes from one source or the other. In particular, when we match the query against the mediator specification MS1, the `<year 3>` pattern can be "pushed" either into `Rest1` or into `Rest2`. The two possibilities correspond to the unifiers τ_1 and τ_2:

$$\tau_1 = \left[\begin{array}{l} \texttt{Rest1} \mapsto \{< \texttt{year 3} >\}, \\ \texttt{S} \mapsto < \texttt{cs_person} \ \{< \texttt{name N} > \ < \texttt{relation R} > \ \texttt{Rest1 Rest2}\} > \end{array} \right]$$

$$\tau_2 = \left[\begin{array}{l} \texttt{Rest2} \mapsto \{< \texttt{year 3} >\}, \\ \texttt{S} \mapsto < \texttt{cs_person} \ \{< \texttt{name N} > \ < \texttt{relation R} > \ \texttt{Rest1 Rest2}\} > \end{array} \right]$$

The two unifiers give rise to the following two rules which constitute the logical datamerge program. Notice that we need two rules because it could be the case that both sources export `year` objects.

```
(Q11) <cs_person {<name N> <relation R> Rest1 Rest2}>
        :- <person {<name N> <dept 'CS'> <relation R> | Rest1:{<year 3>}}>@whois
           AND decompose_name(N, LN, FN)
```

[2] Before we check containment we reduce structures of the form "*variable*:*pattern*" to "*pattern*".

```
           AND <R {<first_name FN> <last_name LN> | Rest2}>@cs
(Q12) <cs_person {<name N> <relation R> Rest1 Rest2}>
         :- <person {<name N> <dept 'CS'> <relation R> | Rest1}>@whois
           AND decompose_name(N, LN, FN)
           AND <R {<first_name FN> <last_name LN> | Rest2:{<year 3>}}>@cs
```

Note, mappings of the form `Rest1` $\mapsto$ {< `year 3` >} cause the attachment of the conditions specified inside the {} to the specified variable (`Rest1` in the example). If `Rest1` has already some conditions S associated with it, QD&AO will produce the conjunction of the conditions S with the `<year 3>` condition.

If the desired `year` information is found at only one of the two sources, then the execution of one of Q11 or Q12 is redundant. Indeed, had we put n subobject conditions on the "cs_person" we would have to emit 2^n queries, most of which are probably redundant. We will describe in Section 5.6 solutions to this unecessary exponential explosion problem.

Note, the examples of the previous paragraphs dealt with single condition queries. Nevertheless, the demonstrated techniques can be easily extended for multiple condition queries. A more challenging task is the extension to fusion mediators, i.e., mediators that use object-id based grouping. In Section 5.4 we show how this extension can be easily handled by reducing both the query and the mediator specification into a normal form. Also note that we have not yet presented the complete algorithm for unification. A complete algorithm, which focuses on normal form MSL, will be presented in Section 5.4.2.

5.2 The Physical Datamerge Graph and the Datamerge Engine

The optimizer receives the logical datamerge program from the QD&AO and generates a *datamerge graph*. This graph specifies the queries to be sent to the sources as well as the mechanics for constructing the query result from the results received from the sources. The graph is then executed by the datamerge engine, which produces the query result.

In this section we illustrate datamerge graph execution through a detailed example. Our goal is not to describe the datamerge engine in full detail (this is done in [Yer]), but rather to show the capabilities of our datamerge engine and illustrate its similarity to relational database engines. As our starting point we use logical datamerge rule Q11. From it, the optimizer may generate the physical datamerge graph of Figure 5.2. This is a "dataflow" graph, where the nodes (rounded boxes) represent the operations to be executed by the engine. The rectangles next to the arcs of the graph represent tables that flow during a sample run of this graph. Typically, the tuples of the tables carry bindings for the logical datamerge program variables.

The datamerge engine executes the graph in a bottom-up fashion. First, the lower *query* node is executed. This causes query `Qw` to be sent to source `whois`, obtaining bindings for `N`, `R`, and `Rest1`. Query `Qw` is provided to the engine by the optimizer, and is defined as:

```
(Qw) <binding_for_whois {<binding_for_N N> <binding_for_R R>
                         <binding_for_Rest1 Rest1>}>
          :- <person {<name N> <dept 'CS'> <relation R> | Rest1:{year 3}}>@whois
```

The result of `Qw` is placed in the mediator's memory. In Figure 5.2 we show this result at the bottom of the figure. The numbers with a "x" prefix represent object addresses in the mediator's memory. For example, one result object is at address `x032`; it has label `binding_for_whois` and its value is a set containing the objects at locations `x036`, `x038` and `x040`. For readability, we omit the object-id and type fields of the objects from the figure.

The query operator produces a table where each line contains the address of a top-level result object (`x032` and `x056` in the example). For readability, we add a heading row to our tables (`Result of Qw` in this case), but these do not appear in practice.

The table is passed to the next operator in the graph, an *extractor* node that extracts bindings of the variables `N`, `R`, and `Rest1` (from the "binding_for_whois" objects) and outputs a table of corresponding (`N`,

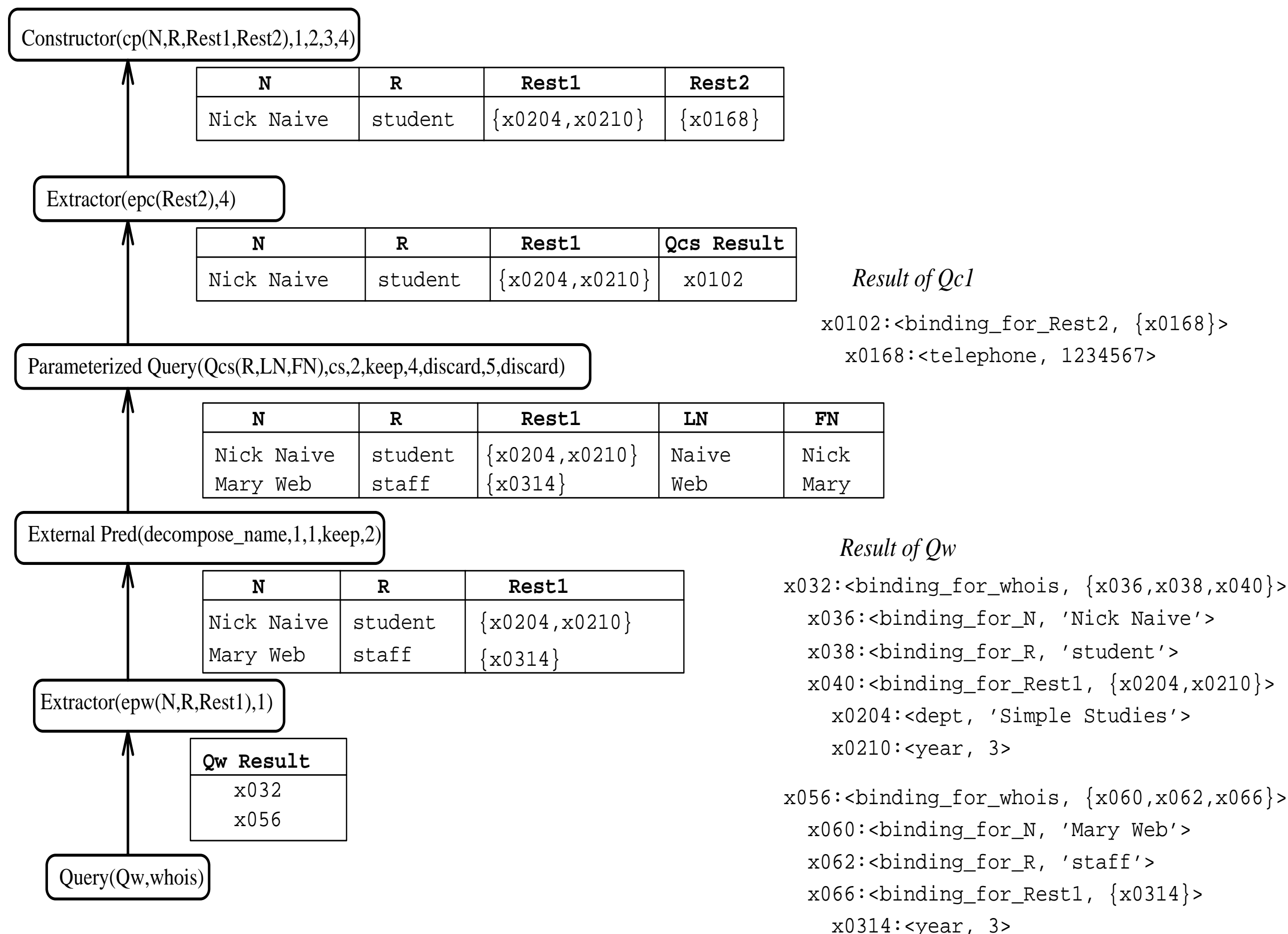

Figure 5.2: A physical datamerge graph

R, Rest1) tuples. The extractor node has two parameters: the first is the optimizer provided object pattern epw, defined by

<binding_for_whois {<binding_for_N N> <binding_for_R R> <binding_for_Rest1 Rest1>}>

epw indicates where the desired bindings are found in the result objects; the second parameter (1) indicates the column of the input table that contains the objects that are the subject of the extraction. Again, the heading row in the output table is only for readability. Also for reabability, in the N and R columns we write strings, while in reality we have pointers to the strings. Similarly, in the Rest1 column we write the full sets while in reality the column contains pointers to the indicated sets.

Then, for every tuple, the *external pred(-icate)* node invokes the predicate decompose_name. The other parameters for this node indicate: the number of arguments for decompose_name (1); the column of the input table containing the one input parameter (1); whether the input column is kept in the output table;[3] and the number of result arguments from decompose_name (2).

The next node is the *parameterized query* node. For each tuple of its input table, this node generates a query for source cs requesting bindings for Rest2 that are needed to construct final result objects. The query to send is defined by Qcs which is provided by the optimizer along with the graph:

[3] As opposed to extractor nodes that always dicard their input column (after using it).

```
(Qcs(R,LN,FN)) <binding_for_Rest2 Rest2>
                      :- <$R {<last_name $LN> <first_name $FN> | Rest2}>@cs
```

The values for query parameters `$R`, `$LN`, and `$FN` are taken from the 2nd, 4th, and 5th columns of the incoming table. (The `keep` and `discard` parameters again indicate if the inputs columns remain in the output table.) Thus, for our sample data, two queries `Qcs1` and `Qcs2` are emitted:

```
(Qc1) <binding_for_Rest2 Rest2>
           :- <student {<last_name 'Naive'> <first_name 'Nick'> | Rest2}>@cs
(Qc2) <binding_for_Rest2 Rest2>
           :- <staff {<last_name 'Web'> <first_name 'Mary'> | Rest2}>@cs
```

Let us assume that `Qc1` returns only the `x0102` "binding_for_Rest2" object and `Qc2` does not return anything. In this case, the parameterized query node outputs the table shown in Figure 5.2. After the upper extractor node extracts `Rest2` bindings from the results of the parameterized query node, the *constructor node* is activated and creates the final result objects. The form of these objects is defined by the pattern `cp(N,R,Rest1,Rest2)` where

```
cp(N,R,Rest1,Rest2) = <cs_person {<name N> <relation R> Rest1 Rest2}.
```

For each row in the input table, the constructor operator takes a row (1st, 2nd, 3rd, and 4th values), assigns them to the `N`, `R`, `Rest1`, and `Rest2` values in `cp`, creating one of the final result objects.[4]

Through this example we have illustrated how the entire mediation process can be described by a low level executable graph. The nodes of our datamerge graphs are the "machine language" of MedMaker which is run by our implementation of the datamerge engine. Indeed, it is interesting to compare them with relational algebra expressions: The *query* node is responsible for obtaining information from remote sites (unlike relational databases where all information resides at the same site), the *extract* node is reponsible for converting the OEM format into relational format, and the *construct* node translates the relational format into OEM. The other nodes operate on relations and produce relations as it would be done in a relational system.

In particular, the engine's design includes *select*, *project*, *join* (implemented by nested loops), *cartesian product*, *union*, *external predicate*, and *filter* nodes. The latter is the special case of external predicate, where the external predicate[5] has no output and it merely checks whether its input satisfies an implementor provided function. The engine offers both the filter node and the external predicate node because it is easier for the mediator implementor to interface a function with no output into a filter node than in an external predicate node.

5.3 Cost-Based Optimization

There is generally more than one physical datamerge graph that correspond to a logical datamerge program. The optimizer has to select the "optimal" graph. However, optimization in distributed and autonomous systems, such as TSIMMIS, is much harder [LOG93] than optimization in a conventional database because optimization criteria are different and cost estimation is harder. (See Section 5.3.1) for a discussion on these issues.) Furthermore the sources have limited and different query capabilities. This makes optimization even harder because a plan which would be optimal, if the sources had full capabilities, may involve emitting to the sources unsupported queries.

MedMaker does not handle the problem of limited and different capabilities. Instead, its cost-based optimizer formulates an "optimal" plan assuming that the sources have full query capabilities. If a query

[4] Our current implementation does not have a duplicate elimination feature, though the MSL semantics describe duplicate elimination in the OEM context.

[5] The current implementation does not include external predicate.

emitted by MedMaker is not directly supported by the underlying source the wrapper of this source may indirectly support the query using techniques described in Chapter 7. Apparently, this is not a complete solution of the problem; solutions on this problem are discussed in Chapter 8.

The cost-based optimizer uses simple heuristics to guide datamerge graph selection. This seems to work well when the alternative graphs are relatively simple (few joins) and have important "qualitative" differences; e.g., the optimal graph uses available indexes at sources while the alternatives do not. However, they do not perform well in the presence of large joins or complicated join hypergraphs. Hence, we do not consider the cost-based optimizer to be a contribution of our mediator work. However, we include it in the presentation because it is part of the MedMaker system and it also gives us the chance to point out future work in optimizers for mediator systems.

In Section 5.3.1 we discuss the main issues in heterogeneous cost-based optimization. In Section 5.3.2 we discuss the cost-based optimizer of MedMaker.

5.3.1 Issues in Cost-Based Optimization

Query optimization in information integration systems such as TSIMMIS is harder than optimization in conventional centralized systems. One of the hardest problems is the different and limited capabilities of the sources. Due to the complexity of this problem we independently discuss it in Chapters 7 and 8. The following issues should not be ignored either:

- *Optimization Criteria:* Relational system's optimizers attempt to minimize the time and resources needed for constructing the full query result. Though the subsequent sections focus on the same optimization criterion we should note that the predominant criterion could be the minimization of the financial cost of retrieving the information, the minimization of the "response time to the next piece of information", or the minimization of the time required for retrieval of a part of the answer. The latter two are especially important when the information is accessed by browsing.

- *Statistics and Cost Estimation:* Relational optimizers have access to statistics such as cardinalities of the base tables, select, project, and join selectivities, etc. They also have precise estimates of the cost of an operation because they know many of the details of the execution engine algorithms and the hardware on which the engine runs.

  Mediators often do not have the "luxury" of precise statistics and cost estimates for the queries they send to to the sources. The reason is that the information sources do not maintain statistics or they do not want to export them. In this case, the optimizer has to have its own source of statistics. In [ACPS96] we have considered inferring statistics and cost estimates from prior usage of the sources. This work has not been implemented and evaluated in the context of TSIMMIS and hence it is only briefly described in Section 5.7. Instead, the optimization component of TSIMMIS uses statistics and cost estimation information that is provided by the mediator specifier. We discuss the details in Section 5.3.2.

- *Typical Queries:* Conventional systems have mainly focused on optimizing conjunctive queries and views. However, integration systems will typically deal with queries or views that union and fuse information. The optimization of such queries will produce new challenges and opportunities. For example, Section 5.6 shows that pushing the maximum number of conditions to the sources is not the optimal policy for fusion mediators. The performance evaluation of Section 6 further illustrates this point. We have also formally proved our claim in [AGMPY].

The above three points will help us position MedMaker's cost-based optimizer described in the following sections.

5.3.2 Cost-Based Optimization in MedMaker

The optimization criterion used by MedMaker is the minimization of the time required to retrieve the full answer. To achieve this the optimizer chooses between different join policies and join orders for executing the conditions that appear in the tails of the datamerge rules. Other optimizations, such as specialized optimizations for fusion queries, have been tested but not implemented by MedMaker's optimizer yet. They will be separately described in Section 6.

MedMaker assumes that the sources do not provide any statistics. Instead, its optimizer uses information that the implementor attaches to the mediator specification. This information assists the mediator in choosing join policies and join orders. The following example illustrates MedMaker's cost-based optimization. In particular, it shows how given (i) a query, and (ii) a mediator specification, annotated with cost information about the sources, it chooses a plan.

EXAMPLE 5.3.1 Consider the following mediator MS2 that joins entries of `inspec`, a bibliographic system of Stanford University, with the articles of `biblio`, a list of bibliographic information files maintained by Stanford's DB group.

```
(MS2) <join_entry {<title T> <abstract B> <author A> <year Y>}>@med :-
                  <inspec_entry V:{<title T> <abstract B> <author A>}>@inspec
                  AND <biblio_entry {<title T> <year Y>}>}>@biblio
```

Then consider the following query which retrieves entries with a specific author name.

```
(Q13) <ans {<title T> <abstract B> <year Y>}> :-
            <join_entry {<title T> <abstract B> <author ''Joe''> <year Y>}>@med.
```

The datamerge program for the "Joe" query is the following.

```
(MS3) <ans{<title T> <abstract B> <year Y>}>@med :-
          <inspec_entry V:{<title T> <abstract B> <author ''Joe''>}>@inspec
          AND <biblio_entry {<title T> <year Y>}>@biblio
```

There are three datamerge graphs that execute the above rule and correspond to the three available join policies for doing the join.

- **local join:** Retrieve all "Joe" entries from `inspec` and all entries from `biblio` and do the join locally.
- **passing bindings from `biblio` to `inspec`:** retrieve all entries from `biblio` and for each title that appears send to `inspec` a query that retrieves the specific document.
- **passing bindings from `inspec` to `biblio`:** retrieve all "Joe" entries from `inspec` and for each one of them send to `biblio` a query that retrieves articles with the same title.

In order to evaluate the cost of these plans we have to consider the features of each source: `biblio` has no index structure; every query is answered by sequentially scanning all bibliographic entries (approximately five thousand) of the underlying source. Hence, the response time of a query is fixed. Source `inspec` is much larger and has many indexes that significantly speed up the search. The time required to answer an `inspec` query is roughly proportional to the size of the result.

Given this information about the sources the optimizer must avoid selecting the "passing bindings from `inspec` to `biblio`" plan. Passing bindings from `inspec` to `biblio` requires that the data of `biblio` are scanned n times, where n is the number of "Joe" entries in `inspec`. This plan is obviously extremely expensive when n is large. It is the best one only in the case where there is no "Joe" in `inspec`. Even in this case, its only benefit over the second best plan is that it avoids altogether scanning `biblio` whereas the local join plan described below will scan `biblio` once.

The decision between the other two plans is not straightforward. The "local join" plan sends only one query to inspec but it retrieves all "Joe" entries found in inspec — not only those which match with some entry of biblio. The "passing bindings from biblio to inspec" plan retrieves only the necessary data but it issues many queries — as many as the articles in biblio. The mediator implementor can assist the optimizer in this decision by providing explicit information about the policy that should be used for joining the conditions, as shown below.

```
(MS4) <join_entry {<title T> <abstract B> <author A> <year Y>}>@med :-
          <inspec_entry V:{<title T> <abstract B> <author A>[6]}>@inspec[2]
          AND <biblio_entry {<title T> <year Y>}>}>@biblio[10]
join_threshold=1
```

The numbers 10 and 2 can be seen as logarithmic metrics of the conditions' selectivity before we specify the value of any subobject condition. The number 8 specifies the selectivity increase of the inspec condition if the author is provided. For the datamerge program of our running example the overall selectivity of biblio is 10 and the overall selectivity of inspec is 6+2=8. If the difference between the conditions participating in the join is larger than the *join threshold* then the optimizer chooses to pass parameters from the "high" selectivity condition to the "low" selectivity condition. The intuition is that the parameterized queries will be only a few, they will "focus" more the high selectivity condition, and hence they'll significantly reduce the cost. Otherwise, local join is chosen. In our running example, we would have a local join if the author condition was more selective, i.e., if it was assigned an 8 instead of 6.

It is obvious that the implemented optimizer has a "crude" cost model. For example, the optimizer has no way to estimate the number of objects returned by each condition and, consequently, the number of parameterized queries that will be asked. The join selectivities, which may also be particularly important, are absent. We excluded these features from our optimizer to simplify it. Undoubtedly, better optimizers should include these features.

Indeed, in [ACPS96] we propose an optimizer that selects the optimal plan using information about the cardinalities and the response times of the queries that are sent to the sources. Furthermore, the optimizer estimates the cardinalities and response times by using information about the cardinalities and response times of queries that have been sent to the sources in the past. This optimizer has been implemented for the HERMES heterogeneous information system which is based on the relational data model. We believe that there is no inherent difficulty in adapting the optimization techniques of [ACPS96] to the OEM model.

5.4 Object Fusion

In Section 5.1 we informally described the resolution and unification algorithms run by QD&AO. Indeed, these algorithms were implemented for the first version of MedMaker, but we had to reconsider them because they do not handle object fusion and because their complexity indicated that the upgrade to object fusion would be extremely complicated. In this section we describe the changes that were required for handling object fusion. We also describe formally the resolution and unification algorithm that is run by QD&AO.

Section 5.4.1 describes the main enhancement for object fusion in QD&AO, namely the reduction of the queries and the specifications into the normal form MSL (informally described in Section 4.4.3). Section 5.4.2 formally describes the resolution and unification process which are significantly simpler due to the normal form. Section 5.5 formally describes "normalization".

5.4.1 Enhanced Query Decomposition and Algebraic Optimization

An important step in the handling of object fusion is the transformation of queries and mediator specifications into *normal form MSL*. Recall, normal form MSL (see also Section 4.4.3) is very similar to full MSL except that patterns always have three fields and certain constructs (such as V:{<title 'abc'>} are not allowed. Having fewer and more regular constructs simplifies the query processing work that follows. In Appendix A

we give the syntax of full and reduced MSL. In Section 5.5 we present an algorithm for converting MSL into normal form. As an example, the algorithm converts the query

```
(Q14) <X tr V> :- <X tr V:{<title 'abc'>}>@m
```

into the query

```
(Q15) <X tr {<Void Vl Vv>}> :- <X tr {<T2 title 'abc'> <Void Vl Vv>}>@m
```

In the second step QD&AO (the Query Decomposer and Algebraic Optimizer) generates a logical datamerge program by matching the query tail conditions with rule heads. Normal form significantly simplifies unification and furthermore it allows object fusion. However, the handling of object fusion requires extensions in our algorithms, as we discuss in the next section.

Query Processing with Fusion

Object-id based fusion introduces additional complexity to the QD&AO process because multiple rules or multiple instantiations of the same rule may contribute to the same mediator object. Hence the query decomposition algorithm that we informally described in Section 5.1.1 is not enough because we have to simultaneously match the query tail conditions with the heads of more than one rule. In this section, we informally generalize our QD&AO algorithm to cover this case and then we formally present the algorithms run by QD&AO in Section 5.4.2.

Let us consider the following mediator (MS3), which was first introduced in Section 4.3 and merges information from sources `s1` and `s2`.

```
(MS3) (R3.1) <trep(RN) techreport {<title T>}>@m :-
                       <report {<report_num RN> <title T>}>@s1
      (R3.2) <trep(RN) techreport {<postscript P>} >@m :-
                       <report {<report_num RN> <postscript P>}>@s2
```

The first step is to convert the rules to normal form MSL. At the same time we rename variables so that no two rules have common variables; using renaming we avoid confusion when rules are merged into a single datamerge rule. (We have also abbreviated some labels; this is just to have more compact patterns in this section.)

```
(MS16) (R16.1) <trep(RN1) tr {<T1 title T>}>@m :-
               <Ro1 r {<RNo1 rn RN1> <T1 title T>}>@s1
       (R16.2) <trep(RN2) tr {<Poid postscript P>}>@m :-
               <Ro2 r {<RNo2 rn RN2> <Poid postscript P>}>@s2
```

Rules (R16.1) and (R16.2) contribute information to the same `tr` objects. Furthermore, different instantiations of the same rule may contribute information to the same `tr` object. For example, assume that `s1` has two `r` objects for the same report number (the source may have duplicates for the same report). Then rule (R16.1) will have two different instantiations with the same `RN1` binding and possibly different `T` bindings. These two instantiations will both contribute information to the same `tr`.

Let us now submit to `m` the query (Q15) which asks for all the subobjects of the `tr` objects where the title is `'abc'`. Since the subobjects of the query may come from different rules, the QD&AO rewrites the query (Q15) as (Q17):

```
(Q17) <X tr {<Void Vl Vv>}> :- <X tr {<T2 title 'abc'>}>@m AND
                               <X tr {<Void Vl Vv>}>@m
```

In this transformed form, which we call *single path conditions* form, we break up the tail so that every set pattern { ... } contains exactly one object pattern < ... >. Such a transformation is straightforward.[6]

Now we can match the two patterns that appear in the (Q17) query tail to different rule heads. Suppose that we start by matching the first pattern of the tail, i.e., <X tr {<T2 title 'abc'>}>.[7] It matches only with the head of (R16.1). This produces the unifier:

$$\theta_1 = [(R16.1) : \texttt{X} \mapsto \texttt{trep(RN1)}, \texttt{T1} \mapsto \texttt{T2}, \texttt{T} \mapsto \texttt{'abc'}]$$

Applying the unifier θ_1 to the query and the rule and replacing the query condition, we produce the following rule.

```
(Q18) <trep(RN1) tr {<Void Vl Vv>}> :-
                    <Ro1 r {<RNo1 rn RN1> <T2 title 'abc'>}>@s1 AND
                    <trep(RN1) tr {<Void Vl Vv>}>@m
```

Observe that this new query has only one condition referring to mediator m. To complete the process, we match the remaining condition that refers to m with the mediator rules. Pattern <trep(RN1) tr {<Void Vl Vv>}>@m matches with either one of the rules of our specification.

First, it matches with rule (R16.2) thus producing the unifier θ_2

$$\theta_2 = [(R16.2) : \texttt{RN2} \mapsto \texttt{RN1}, \texttt{Void} \mapsto \texttt{Poid}, \texttt{Vl} \mapsto \texttt{postscript}, \texttt{Vv} \mapsto \texttt{P}]$$

Second, <trep(RN1) tr {<Void Vl Vv>}> matches with (R16.1). In this case we have to take into consideration that multiple instantiations of rule (R16.1) may contribute title subobjects to the same tr object. Since we have already used (R16.1) for matching the first condition of the query tail, we must not use the same copy of (R16.1) again for matching the second condition. Thus, we introduce a second instance of (R16.1) (see rule (R16.1.b) below) and we match <trep(RN1) tr {<Void Vl Vv>}> against it, producing the unifier θ_3. Note, the second instance of rule (R16.1) must not have the same variable names as the first one.

```
(R16.1.b) <trep(RNb) tr {<T1b title Tb>}>@m :-
                         <Ro1b r {<RNo1b rn RNb> <T1b title Tb>}>@s1
```

$$\theta_3 = [(R16.1.b) : \texttt{RNb} \mapsto \texttt{RN1}, \texttt{Void} \mapsto \texttt{T1b}, \texttt{Vl} \mapsto \texttt{title}, \texttt{Vv} \mapsto \texttt{Tb}]$$

Finally, for each one of the two unifiers θ_2 and θ_3 we develop one datamerge rule, shown below in datamerge program (DP19). Rule (DR19.1) is obtained by replacing the m condition of (Q18) with the rule tail of (R16.2) and subsequently applying θ_2. Similarly, (DR19.2) is derived using the rule tail of (R16.1.b) and unifier θ_3.

```
(DP19) (DR19.1) <trep(RN1) tr {<Poid postscript P>}> :-
                              <Ro1 r {<RNo1 rn RN1> <T2 title 'abc'>}>@s1
                              AND <Ro2 r {<RNo2 rn RN1> <Poid postscript P>}>@s2
       (DR19.2) <trep(RN1) tr {<T1b title Tb>}> :-
                              <Ro1 r {<RNo1 rn RN1> <T2 title 'abc'>}>@s1
                              AND <Ro1b r{<RNo1b rn RN1> <T1b title Tb>}>@s1
```

In this particular case one query condition matched only with one rule head. In the worst case each condition matches with many rule heads potentially yielding an exponential number of datamerge rules. More precisely, if each of the m query conditions unify with n rules, we produce n^m datamerge rules. This explosion can occur, for instance, if the fusion mediator specification has variables in label positions. Alternative query processing techniques for reducing the number of datamerge rules in fusion mediators are described in Section 5.6 and are evaluated in Section 6.

[6] It is sometimes possible to avoid this step, e.g., if QD&AO finds that no object id fusion is performed on objects with a given label.

[7] In general, the order in which we match conditions does not affect the final result.

5.4.2 Formal Specification of Resolution and Unification for Normal Form MSL

In this section we formally describe resolution and unification. We start by formally defining the unifier of a condition with a rule head. Then, we describe an algorithm that discovers all unifiers.

The notation $\theta(e)$ represents the expression e where the substitutions indicated by the mappings of unifier θ have been performed. A single path condition e_1 *matches with a head* e_2 of rule r if there is *a unifier θ from e_1 to e_2*, as described by Definition 5.4.1 below. Note, the following definition is also applicable for any normal form pattern e_2 and single path pattern e_1.

Definition 5.4.1 (Unifier θ from e_1 to e_2) θ is a unifier from e_1 to e_2 if the pattern $\theta(e_1)$ *is included in* the pattern $\theta(e_2)$, as described by Definition 5.4.2. □

Definition 5.4.2 (Pattern e_1 is included in e_2) A pattern e_1 is included in a pattern e_2 if and only if

(a) e_1 has identical object-id and label fields as e_2
(b) *if* the value field of e_1 is of the form $\{e'_1\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ and
there is a pattern $e_2^j, j = 0, \ldots, m$ such that e'_1 is included in e_2^j.
else if the value field of e_1 is of the form $\{\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ (m may be 0)
else e_1 and e_2 have the same value field.

□

For example, the first condition of query (Q17) matches with (R16.1) because the unifier θ_1 maps the condition to the rule head.

Computation of Unifiers: Apparently, unification of normal form patterns is closely related to term unification (Section 12.4 of [Ull89]). However, we must take into consideration the following differences:

- Each condition subobject may match with any rule head subobject. Because of this we may have multiple unifiers of a given condition with a given head.
- A variable that appears in value position may unify with a set pattern.

The unification algorithm is given in Figure 5.3. The input is one single path condition c and a rule head r. The output is the set of unifiers of c and r. Before we analyze the algorithm let us first provide a few explanations:

- The algorithm uses *term unification* for matching object-id's, labels, or atomic values. However we do not describe term unification. Descriptions can be found in [Ull89] or [GN88].
- The composition $\theta \circ \zeta$ of two unifiers θ and ζ is similar to composition of first-order logic unifiers. We first apply θ to all mappings of ζ and then we merge in a single unifier the mappings of both θ and ζ. For example, if $\theta = [X \mapsto a]$ and $\zeta = [Y \mapsto X]$ then $\theta \circ \zeta = [X \mapsto a, Y \mapsto a]$.

  The composition fails –and consequently the attempted unification also fails – if we attempt to have more than one mappings for a variable. For example, assume that we have already mapped the condition variable `V` to `{<a {<l X>}>}`. Then the unification of the condition `<o {<b V>}>` and the rule head `<o {<b {<g Y>}>}>` fails. Intuitively, the unification failed because `V` binds to sets and MSL prohibits deep set equality operations .

Let us now informally describe the unification algorithm of Figure 5.3. It is based on the recursive function **unify** that is initially called with arguments the condition c, the head r and the empty unifier. In general, **unify** is passed a unifier θ that carries all mappings that have already been developed either from matching another condition with the mediator specification or even from matching a part of the currently matched condition with the specification.

For example, consider the condition <f(X) X {<l(U) S U>}> and the rule head <f(Y) o {<l(W) s1 Y> <l(V) s2 V>}>. If the algorithm succesfully unifies the object-id's it formulates an intermediate unifier θ_{oid} which is $[\mathtt{X} \mapsto \mathtt{Y}]$ in the running example. Then it applies θ_{oid} to the labels and attempts to generate the unifier θ_{label}. In the running example applying θ_{oid} to the label X returns the label Y. The unification of the label Y with the label o results in

$$\theta_{label} = [\mathtt{Y} \mapsto \mathtt{o}]$$

If the value fields of c and r are terms then the algorithm applies

$$\theta_{label} \circ \theta_{oid} = [\mathtt{X} \mapsto \mathtt{o}, \mathtt{Y} \mapsto \mathtt{o}]$$

to the values and unifies them attempting to produce the unifier θ_{value}. If it succeeds it returns the unifier $\theta_{value} \circ \theta_{label} \circ \theta_{oid}$.

If the value fields are non empty sets, as is the case in the running example, then the algorithm recursively calls **unify**, passing the subobject of the condition <l(U) S U>, one of the rule head's subobjects, and the unifier $\theta_{label} \circ \theta_{oid}$ that has been formulated so far. Note **unify** makes n recursive calls, where n is the number of subobject patterns in the rule head. In the running example we have a call

```
unify(<l(U) S U>, <l(W) s1 Y>, [X ↦ o, Y ↦ o])
```

and a call

```
unify(<l(U) S U>, <l(V) s2 V>, [X ↦ o, Y ↦ o])
```

For every recursive call the algorithm produces a set of succesful unifiers. The union of the sets constitutes the result of **unify**. In the running example we end up with the unifiers

$$[\mathtt{X} \mapsto \mathtt{o}, \mathtt{Y} \mapsto \mathtt{o}, \mathtt{S} \mapsto \mathtt{s1}, \mathtt{U} \mapsto \mathtt{o}, \mathtt{W} \mapsto \mathtt{o}]$$

and

$$[\mathtt{X} \mapsto \mathtt{o}, \mathtt{Y} \mapsto \mathtt{o}, \mathtt{S} \mapsto \mathtt{s2}, \mathtt{U} \mapsto \mathtt{V}]$$

Hence the result produced by the algorithm is the set

$$\{[\mathtt{X} \mapsto \mathtt{o}, \mathtt{Y} \mapsto \mathtt{o}, \mathtt{S} \mapsto \mathtt{s1}, \mathtt{U} \mapsto \mathtt{o}, \mathtt{W} \mapsto \mathtt{o}], [\mathtt{X} \mapsto \mathtt{o}, \mathtt{Y} \mapsto \mathtt{o}, \mathtt{S} \mapsto \mathtt{s2}, \mathtt{U} \mapsto \mathtt{V}\}$$

The above discussion assumed that either the values of the condition pattern c and the rule head pattern r are both terms or they are both sets. Figure 5.3 also describes the cases where the value of c is a set and the value of r is a variable or vice versa. In the former case (where the value of c is a variable) we have to create a mapping from the value of c to the set pattern that stands for the value of r. For example, if the condition is of the form <l V> and the head is of the form <l {<a 1> <b 2>}> then we create the mapping

$$[\mathtt{V} \mapsto \{< \mathtt{a\ 1} >< \mathtt{b\ 2} >\}]$$

and add it to the unifier $\theta_{label} \circ \theta_{oid} \circ \theta$ that has been produced up to that point. For the sake of simplicity the pseudocode of Figure 5.3 does not handle the complications that arise if the value of c is a variable that

already appears in $\theta_{label} \circ \theta_{oid} \circ \theta$. We will illustrate some of these complications, which are actually handled by the implementation code, using examples.

Assume that we attempt to match the condition <X o X> with <Y o {<a 1>}>. First we map X to Y, which is a term, and then the unification fails because X cannot be a set and a term simultaneously. A more subtle case raises if the condition variable is already mapped to a set. For example, assume that V is already mapped to {<a 1> <b 2>}. Assume that we now want to match V with {<a 1>}. The algorithm will stop the processing of this query and will notify the user that it can not be done because it requires deep set equality — a feature not inncluded in the MSL semantics.

If the value of c is a set pattern and the value of r is a variable V then, intuitively, the mapping must force V to bind to subobjects that correspond to the patterns in c. Hence, the unification procedure maps V to the set pattern if V is not already mapped to anything. Otherwise, if V is already mapped to a set pattern the two sets are concatenated (see the pseudocode of Figure 5.3). For example, assume that we match the condition

```
<o {<a {<x 1>}> <b {<y 2>}>
```

with the head

```
<o {<a V> <b V>}>
```

First we map V to {<x 1>} because the a object must have a <x 1> subobject. Then we update the mapping of V to {<x 1> <y 2>} because the b object must have a <y 2> subobject and the a and the b objects have the same value.

The unification routine is the main component of the QD&AO algorithm of Figure 5.4. Recall that the QD&AO algorithm matches a query condition that refers to the mediator view with all possible rule heads. For every succesful matching a rule is derived by replacing the query condition with the tail of the corresponding rule. If the derived rule contains no reference to the mediator view then it becomes one of the datamerge rules that are the output of QD&AO. Otherwise the derived rule is placed in the queue L of the algorithm of Figure 5.4. The algorithm continues operating until the list L is empty. The generalization of the algorithm to queries that involve more than one rules is straightforward.

5.5 Normalization of MSL Rules

This section describes the reduction of MSL rules into normal form. This reduction is very important for simplifying the matching process, as we have seen in the example of Section 5.4.1. The reduction is implemented by the normalizer component of the interpreter (see Figure 5.1). Recall, the normal form has only three field object patterns and there are no object variables (*e.g.*the O in O:<l V>), subobject conditions on set variables (*e.g.*<l V:{<s 1>}>), or rest variables (*e.g.*<l {<s 1> | Rest}>).

Section 5.5.1 provides a categorization of the variables that appear in MSL rules. The categorization is essential for the reduction/elimination of object variables, rest variables, and value variables that bind to sets. Section 5.5 describes the steps of the reduction. It also contains examples for the steps that are not obvious.

5.5.1 Categorization of Variables

We use the following categorization of MSL variables in the reduction of arbitrary MSL rules to normal-form MSL rules.[8] Note that a variable may be classified in more than one of the following categories:

1. *atomic variables*, that bind to atomic entities only (e.g., object-id's, labels, atomic values) for any object structure exported by the wrappers. These are the variables that appear

[8] We also use the categorization in the expression of safety and typing constraints (see Appendix B.0.1.)

INPUT A single path condition c, and
a normal-form rule head r
OUTPUT A set S of unifiers θ such that $\theta(r)$ contains $\theta(c)$
METHOD Run the function $S =$ **unify**$(c, r, [])$

function **unify**(c, r, θ) *returns sets of unifiers*

apply θ to $c.oid$ and $r.oid$

if term unification of $c.oid$ and $r.oid$ results in θ_{oid} *unify object-id's*
apply $\theta oid \circ \theta$ to labels of c and r
else return empty set

if unification of $c.label$ and $r.label$ results in θ_{label} *unify labels*
apply $\theta_{label} \circ \theta_{oid} \circ \theta$ to values of c and r
else return empty set

if $c.value$ and $r.value$ are terms and their term unification results in θ_{value}
if atomic objects then unify the term values
return $\theta_{value} \circ \theta_{label} \circ \theta_{oid} \circ \theta$
else if $c.value$ is a variable and $r.value$ is a set
return $[c.value \mapsto r.value] \circ \theta_{label} \circ \theta_{oid} \circ \theta$
else if $c.value$ is a set $\{c_1 \ldots c_d\}$ and $r.value$ is a variable
if there is already definition of the form $r.value \mapsto \{s_1 \ldots s_p\}$
return $[r.value \mapsto \{c_1, \ldots, c_d, s_1, \ldots, s_p\}] \circ \theta_{label} \circ \theta_{oid} \circ \theta$
else
return $[r.value \mapsto \{c_1, \ldots, c_d\}] \circ \theta_{label} \circ \theta_{oid} \circ \theta$
else if $c.value$ is the empty set and $r.value$ is a set
return $\theta_{label} \circ \theta_{oid} \circ \theta$
else if $c.value$ has a subobject condition c' and $r.value$ is a set
for each subobject r_i or $r.value$
$S_i =$ **unify**$(c', r_i, \theta_{label} \circ \theta_{oid} \circ \theta)$
return the union $\cup_i S_i$ of the results of the **unify** calls above
else return empty set

Figure 5.3: The unification algorithm

INPUT A single rule query q and a set of specification rules, both in normal form
OUTPUT A datamerge program
METHOD

```
transform the tail of q into single path form
insert q in the queue L
initialize the datamerge program with the empty set of rules
while L is not empty
    extract a rule r_q from L
    extract a condition c from tail(r_q), such that c does not refer to a mediator view
    for each rule r of the mediator specification
        call Θ =unify(c, head(r), [])
        for each θ in ×
            create a rule r' such that
                the head of r' is θ(head(r_q))
                the tail of r' the conjunction of θ(tail(r)) and θ(tail(r_q))
            if a condition of r' refers to a mediator view
                transform the tail of r' into single path form
                insert r' in L
            else
                insert r' in the datamerge program
```

Figure 5.4: The QD&AO algorithm

- in the object-id and label fields of object patterns of the head and/or the tail of the MSL rule,
- in the arguments of predicates, either in the head or the tail of the MSL rule,

2. *object variables*, that bind to objects (*e.g.*`O` in `O:<l V>`. They appear in place of ⟨*object variable*⟩ in the ⟨*object condition*⟩ production of the syntax.

3. *rest variables*, that bind to sets of objects and appear in place of ⟨*set variable*⟩ in the ⟨*rest*⟩ production of the syntax. For example, the variables `Rest1` and `Rest2` of the mediator specification of `med` (see Section 4.1) are rest variables.

4. *non-atomic value variables*, which bind to sets of objects. They appear in place of value fields of patterns of the tail and are not atomic variables. Non-atomic value variables are further classified into the following two categories:

   (a) *value variables that bind to sets (of objects) only*: they are value variables that appear in place of a ⟨*set variable*⟩ in the ⟨*value condition*⟩ production of the syntax. For example, in the following MSL rule `V1` binds to sets only

   ```
   O:- <L V1:{<l1 X>}>@src
   ```

   (b) *value variables that bind to atomic constants or sets*: all the non-atomic value variables that can not be classified as value variables that bind to sets only.

EXAMPLE 5.5.1 Let us illustrate using an example the variable classification described above. Consider the following rule.

```
<X ans V>@med :- O:<I l V:{<s X> <t T>}>@src
```

- `X` is atomic because it appears in the object-id position.
- `O` is object variable.
- `V` is non-atomic value variable. Indeed, it is non-atomic value variable that binds to sets only.
- `T` is non-atomic value variable that binds to atomic constants or sets.

□

5.5.2 Reduction of Arbitrary MSL rules to Normal-Form MSL Rules

The reduction of arbitrary MSL rules to normal-form MSL rules proceeds in the following steps which must be executed in this specific order:

1. reduce all 2-field object patterns that appear in the MSL rule's tail to 3-field object patterns. In particular, we fill the missing object-id fields with invented unique variables.
2. for every variable V that is an object variable, rest variable, and value variable that may bind to sets find a set A_V of object-id variables that are sufficient for describing the bindings of the object, rest, and value variables. We will say that "A_V is the association set of V" or "V is associated with the variables in A_V". For example, the association set of `V` in example 5.5.1 is {`I`}. See Subsection 5.5.4 for details on computing association sets.
3. reduce every 2-field head object pattern, i.e. a pattern that does not specify the object-id of the generated object, to a 3-field pattern. We fill the missing object-id field with an invented term that stands for the automatically generated object-id. This is a very important step because the generated object-id directs how the result will be grouped. See Subsection 5.5.5 for the details of object-id generation. Note that this step has to be done after step 2 because the normalizer needs the association sets in order to generate the object-id's.
4. eliminate all object, rest, and non-atomic value variables (see Subsection 5.5.6).

5.5.3 Reduction of 2-field Tail Object Patterns to 3-field Patterns

The normalizer replaces all 2-field object patterns that appear in the MSL rule tail with 3-field patterns that explicitly specify object-id, label, and value. In particular every 2-field pattern

$$<\langle label\rangle\langle value\rangle>$$

is replaced by a 3-field pattern

$$<O\ \langle label\rangle\ \langle value\rangle>$$

where O is an invented variable that does not appear anywhere else in the MSL rule.

5.5.4 Associations of Non-Atomic Variables to Atomic Variables

The normalizer associates every "non-atomic" variable V with one or more atomic variables $V_1^a, \ldots, V_m^a$ that will be sufficient for "simulating" the behavior of V. The complete list of variable associations is the following:

1. if `O` is an object variable that appears in the MSL rule head, then we associate `O` with an object-id term $\langle oid \rangle$ such that the object pattern

$$\texttt{O} :< \langle oid \rangle\ \langle label \rangle\ \langle value \rangle >$$

   appears in the MSL rule's tail. The angles in $\langle oid \rangle, \langle label \rangle, \langle value \rangle$ signify that anything that could be present in an oid, label, or value field is acceptable in the pattern above.

2. if `R` is a rest variable that appears in the MSL rule head, then the normalizer includes in the association set of `R` the object-id term $\langle oid \rangle$ where the object pattern

$$< \langle oid \rangle \langle label \rangle\ \{...|\texttt{R}...\} >$$

   appears in the MSL rule's tail. We will name $\langle oid \rangle$ "the *oid-rest* term associated with `R`". Furthermore, the normalizer includes in the association set of `R` the object-id terms $\langle oid \rangle_i$ of the subobjects of $\langle oid \rangle$ that must be excluded from `R`. We name each $\langle oid \rangle_i$ "an *oid-excluded* term associated with `R`".

   For example, consider the mediator specification

```
<N L R> :- <O L {<I name N> | R}>@src
```

   The oid-rest term associated with `R` is `O`. The oid-excluded term – in this case only one – associated with `R` is `I`.

3. If the non-atomic value variable `V` appears in the MSL rule head associate `V` with an object-id term $\langle oid \rangle$ such that the object pattern

   <$\langle oid \rangle \langle label \rangle$ `V`>

   or

   <$\langle oid \rangle \langle label \rangle$ `V:{...}`>

   appears in the MSL rule tail.

EXAMPLE 5.5.2 Consider the rule

```
<ans V Rest>@med.  :- <L l V:{<s X> <T t {<A a Y>| Rest}>}>@s.
```

`V` is associated to the object-id `L` and `Rest` is associated with the oid-rest term `T` and the oid-excluded term `A`. □

5.5.5 Object-Id Generation

We replace every 2-field object pattern that appears in the MSL rule head with a 3-field pattern, by inventing an object-id variable that appears in place of the object-id field. For example, the normalizer assigns to the `ans` object of the rule in Example 5.5.2 the object-id `s1(L,T)` for reasons we will explain below.

In general, if the 2-field pattern

$$< \langle label \rangle \langle value \rangle >$$

appears in the MSL rule head, it is replaced by the 3-field pattern

$$< s(\bar{X})\ \langle label \rangle \langle value \rangle >$$

where s is an invented name that does not appear in the MSL specification or the wrapper data and $\bar{X}$ is a list of variables. Intuitively, the selection of variables in $\bar{X}$ guarantees that it is impossible to generate two objects that will be "accidentally" fused together.

1. we insert in $\bar{X}$ all atomic variables that appear in the MSL rule head.
2. if the object variable `O` appears in the MSL rule's head, then we insert in $\bar{X}$ the object-id term $\langle oid \rangle$ that is associated with `O`.
3. if the rest variable `R` appears in the MSL rule's head, then we insert in $\bar{X}$ the oid-rest term $\langle oid \rangle$ that is associated with `R`.
4. if the non-atomic value variable V that binds to sets only appears in the MSL rule's head, then we insert in $\bar{X}$ the object-id term $\langle oid \rangle$ that is associated with `V`.

   EXAMPLE 5.5.3 The normalizer assigns to the `ans` object of the rule in Example 5.5.2 the object-id `s1(L,T)` because `L` is the object-id associated with `V` and `A` is the oid-rest term associated with `Rest`. □

5. if the non-atomic value variable V, which may bind both to sets and atomic values appears in the MSL rule's head, the normalizer puts in $\bar{X}$ a unique variable Z and it also puts in the rule's tail the special predicate `var`(V, A, Z) where A is the object-id variable associated with V. The special predicate `var(Y,X,Z)` causes `Z=Y` if `Y` is atomic and `Z=X` if `Y` is set. Let us justify with an example this normalization step.

   EXAMPLE 5.5.4 Consider the following rule that simply copies `year` objects from the wrapper to the mediator.

   ```
   <ans Y>@med :- <X year Y>@src
   ```

   If the value of year is atomic the MSL semantics directs that the mediator exports one object for every possible value of `Y`. Hence, the normal form equivalent of the rule is

   ```
   <s(Y) ans Y>@med :- <X year Y>@src
   ```

   However, we can not be sure that `Y` is atomic. Indeed, if `Y` is not atomic the normal form equivalent of the rule is

   ```
   <s(X) ans Y>@med :- <X year Y>@src
   ```

   Using the `var` predicate the normal form equivalent of the above rule is

   ```
   <s(Z) ans Y>@med :- <X year Y>@src AND var(Y,X,Z)
   ```

   □

5.5.6 Elimination of Object, Rest, and Value Variables that Bind to Sets

We apply the following rules to eliminate all object, rest, and value variables that bind to sets. Let us first provide a short example which illustrates the transformations formally described later.

EXAMPLE 5.5.5 Let us consider again the rule

```
<ans {V Rest}>@med.  :- <L l V:{<s X> <T t {<A a Y>| Rest}>}>@s.
```

After the normalizer removes the variables `V` and `Rest` it derives the following rule.

```
<s1(L,T) ans {<Void Vl Vv> <Restoid Restl Restv>}>@med.  :-
            <L l {<Void Vl Vv> <s X> <T t {<A a Y> <Restoid Restl Restv>}>}>@s
            AND neq(A, Restoid)
```

Notice that the variable `V` has been replaced by the pattern `<Void Vl Vv>` which will bind to all objects that `V` would bind to. The rest variable `Rest` has been replaced by `<Restoid Restl Restv>`. In addition, the predicate `neq` enforces that `<Restoid Restl Restv>` does not bind to a subobjects of `l`. □

The formal description of the transformation follows:

1. if the object variable `O` appears in the MSL rule's head, and `O` is associated with the object-id term $\langle oid \rangle$ the normalizer replaces instances of `O` in the MSL rule's head with $\$\langle oid \rangle$, which stands for the object identified by $\langle oid \rangle$.[9] We remove `O` from the MSL rule's tail. This normalization has not been implemented yet because the QD&AO module does not process \$ structures.

2. if the rest variable `R` appears in the MSL rule's head, and `R` is associated with the oid-rest term $\langle oid \rangle$ and the oid-excluded terms $\langle oid \rangle_1, \ldots, \langle oid \rangle_n$ then

   (a) we replace the pattern

   $$<\langle oid \rangle \langle label \rangle \; \{\langle excluded\ subobjects \rangle \; |\mathtt{R}:[\{\langle rest \rangle\}]\} >$$

   of the MSL rule tail with

   $$<\langle oid \rangle \langle label \rangle \; \{\langle excluded\ subobjects \rangle <\mathtt{R}\ \mathtt{L}\ \mathtt{V}>[\langle rest \rangle]\} >$$

   where `L` and `V` must be invented unique variables.

   (b) we add in the MSL rule tail the following conjunction of external predicates

   $$\mathtt{neq}(R, \langle oid \rangle_1) \wedge \ldots \wedge neq(R, \langle oid \rangle_n)$$

   where the built-in predicate *neq* is satisfied if its two arguments are not identical.

   (c) we replace the object patterns

   $$<\langle oid' \rangle \langle label \rangle \; \{\langle subobjects\ list \rangle \; R \; \langle subobjects\ list \rangle'\} >$$

   of the MSL rule head with

   $$<\langle oid' \rangle \langle label \rangle \; \{\langle subobjects\ list \rangle \; \$\mathtt{R} \; \langle subobjects\ list \rangle'\} >$$

   (d) we replace the object patterns

   $$<\langle oid' \rangle \langle label \rangle \; \mathtt{R} >$$

   of the MSL rule head with

   $$<\langle oid' \rangle \langle label \rangle \; \{\$\mathtt{R}\} >$$

   (e) if `R` appears in place of the MSL rule head, we replace it with `$R`.

3. if the value variable `V` that binds only to sets appears in the MSL rule's head then

[9] In order to keep simple the syntax of Figure A.1 we allow only variables to follow the `$` symbol. If $\langle oid \rangle$ is not a variable, but it is a constant or a term we should introduce in the body of the rule an equality $V = \langle oid \rangle$ where V is an invented variable. Then we can place V after the `$`

(a) we replace the pattern

$$<\langle oid\rangle\langle label\rangle\ \mathtt{V} : \{\langle obj\ cond\ list\rangle\}>$$

of the MSL rule tail with

$$<\langle oid\rangle\langle label\rangle\ \{<\mathtt{V\ L\ Val}>\langle obj\ cond\ list\rangle\}>$$

where L, T, and Val must be invented unique variables.

(b) we replace object patterns

$$<\langle oid\rangle'\langle label\rangle\ \{\langle subobjects\ list\rangle\ \mathtt{V}\ \langle subobjects\ list'\rangle\}>$$

of the MSL rule head with

$$<\langle oid\rangle'\langle label\rangle\ \{\langle subobjects\ list\rangle\ \$\mathtt{V}\ \langle subobjects\ list'\rangle\}>$$

(c) we replace object patterns

$$<\langle oid'\rangle\langle label\rangle\ \mathtt{V}>$$

of the MSL rule head with

$$<\langle oid'\rangle\langle label\rangle\ \{\$\mathtt{V}\}>$$

(d) if V appears in place of the MSL rule head, we replace it with $V.

5.6 More Optimizations

Datamerge rules are evaluated by sending queries to the sources, yielding bindings for the rule variables. Since querying sources may be expensive, we want to reduce the number of queries to a minimum. This section presents a large number of techniques that are used for this purpose.

We start with optimizations that are based on rewritings of the datamerge rules using a modification of subsumption for MSL. QD&AO uses two subsumption-based optimizations for this purpose, *rule elimination* and *query reuse*.

Rule elimination: A datamerge rule can be eliminated if the data that it produces are subsumed by the data produced by another rule.

Query reuse: Each query generated by a datamerge rule obtains bindings for variables, but not all bindings are useful for constructing the fused object. Only variables that appear in the rule head, or variables that join conditions in the tail, are *useful*. We may avoid issuing a query if all of its bindings for useful variables are obtained by another query to the source.

We only illustrate query reuse and not rule elimination (that is very similar). Let us consider datamerge rule (DR19.1). To evaluate it, we need to send a query to s1 to evaluate the condition <Ro1 r {<RNo1 rn RN1> <T2 title 'abc'>}>@s1. This query only contains one useful variable, RN1. Notice that all RN1 bindings in the above condition are also bindings of RN1 in rule (DR19.2). Hence, instead of accessing s1 twice, we can reuse the bindings retrieved for (DR19.2) by rewriting the datamerge program as follows. Note, (DR19.2.b) and (DR19.1.b) correspond to the rewritten versions of (DR19.2) and (DR19.1).

```
(DR19.2.b) [ <trep(RN1) tr {<T1b title Tb>}>
             bind1(RN1) ] :- <Ro1 r {<RNo1 rn RN1> <T2 title 'abc'>}>@s1
                             AND <Ro1 r {<RNo1b rn RN1> <T1b title Tb>}>@s1
(DR19.1.b) <trep(RN) tr {<Poid postscript P>}> :- bind1(RN1)
                        AND <Ro2 r {<RNo2 rn RN1> <Poid postscript P>}>@s2
```

```
(DP22) (DR22.1) [ <trep(RN1) tr {<O1 A1 X1>}>
                  bind1(RN1) ] :- <Ro1 r {<RNo1 rn RN1> <Y year '95>}>@s1
                                  AND <Ro1b r {<RNo1b rn RN1> <O1 A1 X1>}>@s1
       (DR22.2) <trep(RN1) tr {<O2 A2 X2>}> :- bind1(RN1)
                                  AND <Ro2 {<RNo2 rn RN1> <O2 A2 X2>}>@s2
       (DR22.3) [ <trep(RN2) tr {<O2 A2 X2>}>
                  bind2(RN2) ] :- <Ro2 r {<RNo2 rn RN2> <O2 A2 X2>}>@s2
                                  AND <Ro2b r {<RNo2b rn RN2> <Y year '95'>}>@s2
       (DR22.4) <trep(RN2) tr {<O1 A1 X1>}> :- bind2(RN2)
                                  AND <Ro1 r {<RNo1 rn RN2> <O1 A1 X1>}>@s1
```

Figure 5.5: Datamerge program

The notation `[ ... ]` specifies multi-head rules. Thus, the data retrieved from the tail of (DR19.2.b) is used for constructing `<trep(RN) tr {<T1b title Tb>}>` objects, as well as collecting the `RN1` bindings in relation `bind1(RN1)` (the name `bind1` is a unique name generated by the QD&AO.) Then, the `RN1` bindings are used by (DR19.1.b).

We can detect the applicability of the "query reuse" and "rule elimination" rewritings by using unifiers. In particular, a datamerge rule condition c can reuse a datamerge rule r if there is a unifier θ from the tail of r to c and every useful variable of c appears in the head of r. Similarly, a datamerge rule r' can be eliminated if there is a datamerge rule r and a unifier θ such that θ maps the tail of r' to the tail of r and it also maps the head of r to the head of r'.

Note, subsumption based rewritings always improve the datamerge program. The rule elimination technique always improves a program because there are fewer rules to execute in the rewritten program. Furthermore, a rule elimination does not affect the applicability of the "query reuse" optimization because when we remove a rule r we still retain another one that generates data that is superset of the data generated by r. The query reuse rewriting always improves the datamerge program assuming that retrieving bindings from the mediator's storage is more efficient than retrieving them from the source.

5.6.1 Limiting the Number of Datamerge Rules

As mentioned earlier, query processing may yield an exponential number of datamerge rules. In this section we will study two techniques that can significantly reduce the number of rules and queries sent to the sources. Before discussing the techniques we give a concrete motivating example. Consider mediator (MS20) (that also appeared in non-normal form MSL as (MS4) in Section 4.3). (MS20) integrates documents without explicitly mentioning their non-key attributes.

```
(MS20)
  (R20.1) <trep(RN1) tr {<O1 A1 X1>}>@all :- <Ro1 r {<RNo1 rn RN1> <O1 A1 X1>}>@s1
  (R20.2) <trep(RN2) tr {<O2 A2 X2>}>@all :- <Ro2 r {<RNo2 rn RN2> <O2 A2 X2>}>@s2
```

Let us assume that query (Q21) is sent to mediator (MS20).

```
(Q21) <X tr {<Void Vl Vv>}> :- <X tr {<Y year '95'> <Void Vl Vv>}>@m
```

The label `year` may come either from `s1` or `s2`. This intuition is captured by the standard query/rule matching process (see Section 5.1) that results in the datamerge program (DP22) of Figure 5.5.

Observe that this simple query results in many datamerge rules and, consequently, in many queries sent to `s1` and `s2`. In general, if a query asks for reports with attributes `l1`,...,`ln` and the mediator specification does not indicate the origin of `l1`,...,`ln`, we must create and execute datamerge rules that correspond to all possible partitions of the set `l1`,...,`ln` between `s1` and `s2`, i.e., we need a number of rules that is exponential in n.

Learning about the sources

One of the strengths of MSL is its ability to integrate sources without having a "schema" that describes the type of information found there. However, this lack of schema may result in the large number of rules we have illustrated. In particular, a schema could help us rule out in advance queries that will never return an answer, and hence reduce the number of rules. For instance, in (DP22), if we know that `year` information may not come from `s1` then we can remove rules (DR22.1) and (DR22.2) since they both require that a `year` be found at `s1`.

Even though the mediator does not have a schema, it could achieve the same effect by *asking at run time* if source `s1` had any objects with `year` label. If no such objects existed at `s1`, the mediator could eliminate all datamerge rules that require a `year` at `s1`. (In practice, we can interleave query decomposition with this label checking, so the rules would never have to be created.)

The label queries we have described could be addressed to a *lexicon service* residing either at the source or the wrapper for the source. The service could answer label queries based on its knowledge of the domain (e.g., only medical terms defined in a known dictionary are used as labels in a given structure), based on index structures (e.g., the source provides a label index for speeding up queries), or based on a local schema if there happens to be one (e.g., the data at this source is stored in a relational database). We describe in Chapter 6 a performance evaluation of the effects of lexicon servers.

There are many variations to the idea of lexicon services; we only mention two briefly here. One variation is a service that answers more complex queries regarding the relationship between labels. For instance, we may want to ask if `s1` contains any top-level objects with label `r` that in turn contain a `year` subobject. If there are no such objects, then we can rule out `s1` queries even if `s1` has `year` labels somewhere. Another variation is to cache label information from previous queries at the mediator itself. In this case, the lexicon service would reside at the mediator, but its information could be out of date. Thus, this information could not be used to rule out sources, but could be used to order queries so we would first probe the sources most likely to have matching data. This is very useful if the end-user wants some results quickly or does not want to perform an exhaustive search.

Local evaluation of conditions

We now consider a second technique for reducing the number of datamerge rules. The key observation is that we are generating large numbers of queries because we are pushing all conditions to the sources. Thus, we may try to reduce the number of datamerge rules by pushing fewer conditions, i.e., *locally* evaluating some of the conditions.

For example, suppose that a query `Q` contains conditions on three labels `l1`, `l2`, and `l3`. Query `Q` is run against a mediator that merges data from sources `s1` and `s2`. Suppose that both sources know about these labels. We may reduce the number of datamerge rules by considering first the `l1` condition only. That is, we evaluate an intermediate query that retrieves data from the sources based on the `l1` condition only. The result of this intermediate query contains the objects in the result of `Q`, but may contain additional objects. Then, we use additional datamerge rules that apply the `l2` and `l3` conditions to the intermediate result. There are two benefits to this approach: First, the total number of datamerge rules is smaller. In general, if there are n labels in the conditions, we now generate a number of rules proportional to n, not exponential on n. Second, fewer of these rules generate queries for the sources; the rest can be evaluated at the mediator.

The tradeoff here is as follows: If we push down conditions with high total selectivity we restrict the amount of retrieved data but we increase the number of rules and queries. If we push fewer conditions, which have a lower total selectivity, we have fewer queries but we retrieve more data. Balancing this tradeoff is a cost-based optimization issue that is not currently addressed by the interpreter. The current implementation always pushes the maximum number of conditions to the source, under the assumption that simultaneous conditions on many labels are rare.

```
(PDP23) (PDR23.1) [ <trep(RN) L V>, bind1(RN) ] :- <trep(RN) L V>@(Q24,s1)
        (PDR23.2) <trep(RN) L V> :- bind1(RN)
                                    AND(=>) <trep(RN) L V>@(Q25,s2)
        (PDR23.3) [ <trep(RN) L V>, bind2(RN) ] :- <trep(RN) L V>@(Q26,s2)
        (PDR23.4) <trep(RN) L V> :- bind2(RN)
                                    AND(local) <trep(RN) L V>@(Q27,s1)
        (Q24) <trep(RN) tr {<O1 A1 X1>}> :- <r {<rn RN> <Y year '95>}>@s1
                                           AND <r {<rn RN> <O1 A1 X1>}>@s1
        (Q25) <trep(RN) tr {<O2 A2 X2>}> :- <r {<rn $RN> <O2 A2 X2>}>@s2
        (Q26) <trep(RN) tr {<O2 A2 X2>}> :- <r {<rn RN> <Y year '95'>}>@s2
                                              AND <r {<rn RN> <O2 A2 X2>}>@s2
        (Q27) <trep(RN) tr {<O1 A1 X1>}> :- <r {<rn RN> <O1 A1 X1>}>@s1
```

Figure 5.6: A Physical Datamerge Program

5.6.2 Applying Subsumption after Plan Generation

Our remaining logical optimizations are applied during or after physical plan generation. Thus, we start by briefly recalling how physical plans are obtained. (Section 5.2 explains this in more detail but here we focus more on multirule datamerge programs that are candidates for subsumption.) Then, in the remainder of the section we discuss logical optimizations to the physical plans.

The cost-based optimizer receives the datamerge program and creates a datamerge plan. An alternative representation of datamerge plans — that facilitates the discussions of this Section — is a *physical datamerge program* that consists of a list of (possibly parameterized) queries that will be sent to the sources, along with a description of how to combine query results. To illustrate, let us consider the datamerge program (DP22). (Assume that (DP22) could not be simplified any further using lexicons.) (PDP23), in Figure 5.6, is one of the possible physical datamerge programs (from now on referred to as physical programs) for (DP22).

The notation `@(Q24,s1)` in physical rule (PDR23.1) indicates that query (Q24) should be sent to `s1` and the result should be treated as a "data source" for the rule. The query obtains from `s1` all data about reports with year `'95'`. Rule (PDR23.1) then saves the retrieved reports and stores the `RN` bindings in `bind1`.

The `=>` annotation in rule (PDR23.2) indicates that we perform a nested-loops join of `bind1(RN)` and `<trep(RN) L V>@(Q25,s2)`. That is, for every binding r of `RN` in `bind1`, we instantiate a parameterized query (Q25), by replacing `RN` with r, and we send the instantiated query to `s2`. Similarly, the `local` annotation that appears in (PDR23.4) indicates that we perform a local join of `bind2(RN)` with `<trep(RN) L V>@(Q27,s1)`. The join policy decision is made by estimating the cost of each option using information about the sources (e.g., "does the source have an index on report number?") as we've seen in Section 5.3.

Query Subsumption Optimization In Section 5.6 we showed how to eliminate redundant rules from a datamerge program and how to reuse the results of some rules. We now revisit subsumption and demonstrate that once the actual queries have been formulated some query calls may be saved by reusing the results of other queries.

For example, query (Q24) is subsumed by query (Q27) because (Q27) retrieves all the reports of `s1` whereas (Q24) retrieves only the reports with year `'95'`. Furthermore, once we have the result of (Q27) we may locally apply the condition on `year` and hence compute the result of (Q24). The optimizer captures this relationship between (Q24) and (Q27), eliminates (Q24), and modifies rule (PDR23.1) to use the subsuming query (Q27). Note the condition on `year` that is applied on the result of (Q27).

```
(PDR23.1.b) [<trep(RN) L V>, bind1(RN)] :-
                  <trep(RN) L {<Y year '95'>}>@(Q27,s1)
```

Detecting query subsumption is again done through unifiers. In particular, a query q is subsumed by a query q' if there is a unifier θ that maps the tail of q' to the tail q and furthermore all variables that appear in $\theta(head(q))$ also appear in $\theta(head(q'))$. With a few extensions to the unification process, we can also derive the condition that has to be applied on the subsuming query.

Note that query subsumption optimization can only be performed after we know which queries will be sent to sources, i.e., after the physical plan is generated. Our earlier logical optimizations could also be performed at this latter stage, but it is much better to do them as early as possible to simplify plan generation. This leads to the following strategy: first do as many logical optimizations as possible, then generate plans, and finally perform the remaining optimizations.

5.6.3 Optimization of Negation Operations

In Section 4.3.4 we argued that information blocking is effective for removing inconsistencies and establishing priorities between information drawn from various sources. In general, all specifications involving information blocking contain `NOT` conditions that guide blocking. The performance challenge is to avoid issuing queries that retrieve information that is blocked. The interpreter can reduce to a minimum the number of queries sent to the sources and the amount of retrieved data, for a wide class of queries and information blocking mediator specifications. Due to space limitations, here we only sketch the techniques that are used.

Let us consider mediator specification (MS6) that exports all `s1` reports and `s2` reports with numbers that do not appear in `s1`. In the simplest case, the query specifies the required report number `RN`, say `'123'`. In this case we develop a physical datamerge program that contains (PDR28). The important point is that we evaluate the `NOT provides('123')` condition of (PDR28) before we emit the query `Q` that obtains data for `'123'` from `s2`. (We omit `Q`.) . Thus, if `'123'` is provided by `s1` we avoid sending `Q` to `s2`.

```
(PDR28) <trep(RN) tr {<O2 A2 X2>}> :- NOT provides('123') AND
                                      <trep(RN) tr {<O2 A2 X2>}>@(Q,s2)
```

In other cases, avoiding the retrieval of "blocked data" is more complicated or even impossible. For example, consider query (Q15) that requests reports with title `'abc'`. The best strategy here depends on the expected number of matching reports at each site. For instance, assume that the number of `'abc'` reports retrieved from `s1` is not large. To be specific, say that only reports `'123'`, `'136`, and `'253'` have title `'abc'`. In this case the best strategy is probably to send to `s2` query (Q29) with explicit negation conditions for each one of the `s1` reports. (In general it has a `NOT RN=`b for every b that is a member of `provides`.)

```
(Q29) <trep(RN) tr {<O2 A2 X2>}> :- <Ro2 r {<RNo2 rn RN> <O2 A2 X2>}>@s2
                 AND NOT RN='123' AND NOT RN='136' AND NOT RN='253'
```

If the number of reports retrieved from `s1` is large it may be preferable to ship relation `provides` to `s2` and then send to `s2` a query that requests all reports whose report numbers do not appear in `provides`. If `s2` is not willing to accept a full relation from the mediator, another option is to retrieve from `s2` all reports with title `'abc'` and test locally whether these reports are also provided from `s1`. If they are, the `s2` version can be discarded. In this case, blocking could not really be exploited to reduce the data retrieved from `s2`.

5.7 Related Work

Query processing in MedMaker adapts many relational processing techniques for the purpose of querying and integrating semistructured data. Indeed, we believe that the most important contribution is the reduction of many query processing problems into well-known relational query processing techniques hence avoiding the complexities of developing a whole new query processing technology, as is done in [BDHS96]. The normalizer is the module that is mainly responsible for making the reduction: MSL normal form is very similar to Datalog and/or SQL focused on conjunctive queries, hence making possible the adoption of many relational query processing techniques:

- The resolution and unification process implements the well-known "push selections/projections down" rule. It can also be viewed as magic predicates for non-recursive programs.
- The cost-based optimizer imitates the Wong-Yousefi algorithm that was used in INGRES.
- The use of subsumption imitates common subexpressions in relationa algebra trees.

Nevertheless, the reduction to Datalog is not "perfect" in the sense that we would like to work directly with Datalog. For example, we may have values that may be atomic or nested. In the latter case we are able to push subobject conditions on them. It would be very difficult to handle this case had we worked with Datalog.

Chapter 6

Performance Evaluation

Chapters 3 and 4 argued that MedMaker provides the appropriate functionality for integration and fusion of semistructured data coming from dynamic autonomous sources. Then Chapter 5 described MedMaker's algorithms, discussed their efficiency, and suggested alternative strategies for the cases where the algorithms are not expected to deliver the optimum performance. However, the complexity of MedMaker's algorithms makes difficult the analytical evaluation of their performance. This chapter experimentally evaluates MedMaker. The evaluation, though not complete, it shows MedMaker's efficiency in many common scenarios, reveals the weak spots, and illustrates the overall trends.

The performance metric we use in our study is the *total time* spent for processing a query. In particular, we consider the total CPU time spent at the mediator (recall that MedMaker does not use disk), the total network time, and the time spent at the sources processing the queries emitted by the mediator, i.e.,

$$Total = CPU + network + total\ sources$$

We study the following cases and reach the corresponding results:

- When MedMaker is used in a setting similar to a relational system — i.e., when the source data have regular known structures — its performance scales up linearly as the number of rules, sources, attributes, and conditions increase. We show this by a series of experiments where we study how the time spent by each module of MedMaker increases as a function of the number of rules, sources, attributes of the source objects, and conditions in the query (see Section 6.2).
- Then we study MedMaker's performance in object fusion scenarios where the source schemas are unknown or scenarios where we have to send every condition to every source. The experiments illustrate the disadvantages of trying to push the maximum number of conditions down to the sources. The alternatives of Section 5.6 (i.e., the use of lexicon servers and pushing only one condition in every query) are evaluated and are shown to outperform the "push maximum" policy (see Section 6.3).

Overall, the network cost — i.e., the time spent on network operations — dominates by orders of magnitude the time spent at any component of MedMaker. The reason is that MedMaker's algorithms use only CPU and main memory and do not perform disk operations. However, we still want to check the performance of each MedMaker's algorithm to understand how they scale. Hence we do not only measure the overall cost of a query (which is the sum of network cost and processing cost at the sources) but we also measure the cost of each module, i.e., the time spent by each module. We present the setting of the experimentation in the next section. Then we present in detail results in Sections 6.2 and 6.3.

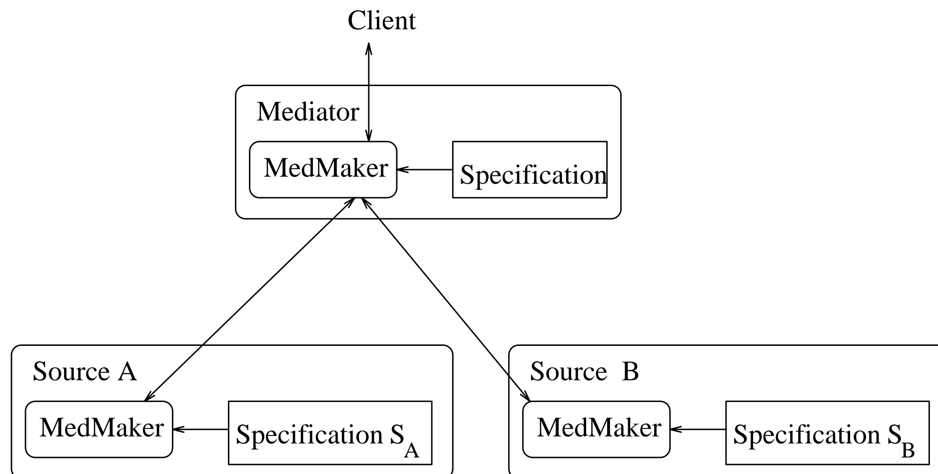

Figure 6.1: The architecture used in the experiments

6.1 Experimental Setting

Before we describe the measured costs and how we measure them (Section 6.1.2) we explain why we needed synthetic sources for the experiments and how we constructed them.

6.1.1 Synthetic Sources

The architecture (see Figure 6.1) used in the experiments differs from the typical TSIMMIS architecture in that it does not use wrappers and sources as they have been described so far. Instead mediators connect to synthetic OEM sources which are actually MedMaker programs. We can specify each object o of a synthetic source, say A (see Figure 6.1), by including in the specification S_A of the corresponding MedMaker program the rule

$$o \; :- \; \texttt{true}.$$

For example, if we want a synthetic source to export the object

```
<&o1 top {&a11, &a12}>
          <&a11 attr1 v1>
          <&a12 attr2 v2>
```

we include the rule

```
<&o1 top {<&a11 attr1 v1>
          <&a12 attr2 v2>
          }> :- true.
```

Note that no special machinery is required for MedMaker to handle rules of the above form. Matching queries with such specifications results in plans that consist of constructors only (one constructor for every object of the answer).

We use synthetic sources instead of wrappers and real sources because MedMaker emits a large variety of queries and many of them are too complex to be supported by TSIMMIS wrappers.[1] On the other hand

[1] Even if the underlying source is a relational database which supports all queries over its schema it is impossible to generate, using the techniques of Chapter 7, a wrapper that supports every query.

Cost	*Definition*
QD&AO cost	time spent by the QD&AO module
parse cost	time spent by the parser module
optimizer cost	time spent by the optimizer module
net engine cost	time spent by the engine in operations other than querying the sources
total MedMaker cost	CPU time spent at MedMaker's site $= QD\&AO + parse + optimizer + net\ engine$
source querying cost	time spent by the sources for answering the queries emitted by the engine
network cost	time spent on the network for transfering queries and results
plan execution cost	total time spent for executing the datamerge graph $= net\ engine + source\ querying + network$
total cost	time spent by MedMaker, sources, and network $= QD\&AO + parse + optimizer + plan\ execution$ $= total\ MedMaker + source\ querying + network$

Figure 6.2: Costs measured

the synthetic sources support every MSL query. At the same time they facilitate experimentation because they allow us to easily vary the form and size of the source data.

One might be skeptical about the value of the following experiments, or even about the value of MedMaker, since they do not run on top of real sources. There are two counterarguments to this skepticism:

1. If the sources support every query over their schema, e.g., they are relational databases, and the wrappers can translate every MSL query into a corresponding source query then all the results presented in this chapter hold.

2. If a source does not directly support the queries formulated by the QD&AO and the optimizer then the wrapper or the mediator must find indirect ways to support the queries. Indirect support is discussed in Chapters 7 and 8. At any rate, most of the query processing techniques we evaluate in this chapter will still have to be considered. After all it is impossible to solve the query optimization problem for sources with limited capabilities before we first understand the solution for sources with full capabilities.[2]

6.1.2 Experimentation Environment and Measurements

The experiments were conducted using a modified MedMaker that measures, using the `times` function of `sys/time.h`, the time spent by each client query in each module of MedMaker, the time spent on network operations, and the time spent at the sources for processing queries emitted by the mediator. We refer to all of these times as *costs*. Computing the network cost and the datamerge engine cost is more complicated and requires a few explanations. For the datamerge engine module we compute two costs:

1. The *plan execution cost* is the total time required for evaluating the datamerge graph. It includes the time the engine waited for the sources to process the queries sent by MedMaker.

2. The *net engine cost* is the time spent by the engine in operations other than querying the sources. The net engine cost is obtained as follows:

   (a) Measure the time t spent for the execution of the query operators of the datamerge graph.

[2]This statement does not imply that the best query processing strategy is to first find an "optimal" plan assuming the sources can handle every query and then find indirect ways to support the queries that are not directly supported. Recently [KHWY] proposed an algorithm that "mixes" the two phases producing very good results. The problem of optimization for sources with limited capabilities is hard and still open.

Parameter	Base	Range
Number of Rules	10	5—40
Number of Sources	5	5—25
Number of Attributes	10	5—25
Number of Query Conditions	1	1—5
Query Result Size	25 objects	NA

Figure 6.3: Experimentation parameters: their base values and their range

(b) Subtract t from the plan execution cost.

Note that the net engine cost underestimates the time spent by the datamerge engine because sending the query, fetching the data from the sources, and storing them in the mediator's memory definitely has a cost on the CPU time of the machine where the mediator operates. This cost should ideally be accounted in the net engine cost. Nevertheless the observed trends will be the same.

The calculation of network cost is based on the fact that MedMaker's engine executes operations sequentially. In particular, queries are sent to the sources sequentially and no query is sent until the result of the previous query is received. Under this execution model the time required for receiving the result to a query is the sum of the time required for query processing at the source and the time spent on the network, assuming that there is no other simultaneous significant activity on the network and on the processing machine. This is the case during our experimentation.

We also report the total MedMaker cost that is the CPU time spent at MedMaker's site and the total cost. Figure 6.2 presents the definitions of the measured costs and some important equations regarding the relationship of the measured costs.

The experiments were guided by C shell scripts that start the client, the mediator, and the sources, giving them every time a series of queries and specifications where either the number of conditions, or rules, or participating sources, or attributes ranges within the space reported in Figure 6.3. Then the scripts collect cost profiles from the sources and the mediators and compute the costs described above. Since random factors also affect the measurements every experiment has been performed five times and we report the average of each set of experiments.

With 90% confidence the actual values are within 11% of the reported average value for all experiments where the reported value is greater than 200ms. This means that if, for example, an experiment returns a value of 1000ms for some cost then with 90% confidence the actual value is between 890ms and 1110ms. For costs that have smaller values the intervals are larger because the rounding errors, introduced by the 10ms granularity of the clock, are greater. In particular, with 90% confidence for reported average values greater than 60ms and less than 200ms the actual value is within 26% of the reported value. The only exception is the net engine cost where the random errors are larger (because the net engine cost is a sum and the rounding errors accumulate) and the actual value is within 34% of the reported value. Note that the less credible results are the ones that refer to "small" costs and hence are less influential anyway.

The clients, mediators, and synthetic sources of the experiments were run on `sponge.stanford.edu` which is an IBM RS6000 370 workstation having 64M memory and running AIX 4.1.4 on a 75MHz CPU.[3]

6.2 Performance of MedMaker when the Schema is Provided in the Specifications

Our first set of experiments evaluates the performance of MedMaker in situations that resemble view specification in relational systems, i.e., when the source data are regular and their regularities are explicitly

[3] We are thankful to IBM Corp. for providing us the equipment.

```
<&j(J) topmed { <&XA2 attr2 VA2> <&XA1 attr1 VA1> }> :-

     <&X top1 {<&JOID key J> <&XA2 attr2 VA2> <&XA1 attr1 VA1> }>@src1 .

<&j(J) topmed { <&XA2 attr2 VA2> <&XA1 attr1 VA1> }> :-

     <&X top1 {<&JOID key J> <&XA2 attr2 VA2> <&XA1 attr1 VA1> }>@src2 .

<&j(J) topmed { <&XA2 attr2 VA2> <&XA1 attr1 VA1> }> :-

     <&X top1 {<&JOID key J> <&XA2 attr2 VA2> <&XA1 attr1 VA1> }>@src3 .

<&j(J) topmed { <&XA2 attr2 VA2> <&XA1 attr1 VA1> }> :-

     <&X top2 {<&JOID key J> <&XA2 attr2 VA2> <&XA1 attr1 VA1> }>@src1 .

<&j(J) topmed { <&XA2 attr2 VA2> <&XA1 attr1 VA1> }> :-

     <&X top2 {<&JOID key J> <&XA2 attr2 VA2> <&XA1 attr1 VA1> }>@src2 .

<&j(J) topmed { <&XA2 attr2 VA2> <&XA1 attr1 VA1> }> :-

     <&X top2 {<&JOID key J> <&XA2 attr2 VA2> <&XA1 attr1 VA1> }>@src3 .
```

Figure 6.4: The structure of the fusion mediator specification for the reference experiment. For space reasons we show only two sources, four rules, and two attributes

provided in the MSL specification. We show that MedMaker scales up linearly as a function of the parameters of Figure 6.3 and its additional flexibility (i.e., the handling of semistructured data and unknown schemas) does not come at the cost of efficiency in this conventional scenario.

In particular we test how fusion mediators perform as a function of the number of rules, attributes, sources, and query conditions. We start by presenting the experiments that show the performance of a fusion mediator as a function of the number of rules participating in the fusion (see Figure 6.5).

The fusion mediator operates on top of five wrappers which are named `src1`, `src2`, etc. Each wrapper conceptually exports 25 objects with label `top1`, 25 objects with label `top2`, and so on up to `top30`. The ith `top1` object of each wrapper has a "key" subobject with value i. The fusion mediator specifications of the experiments use the values of the `key` objects to fuse together information. Figure 6.4 shows a fusion mediator which fuses together information of the `top1` and `top2` objects of wrappers `src1`, `src2`, and `src3`. In particular, the first three rules fuse information from the `top1` objects of `src1`, `src2`, and `src3`. The fourth, fifth, ans sixth rule fuse information from the `top2` objects.

In addition to the key the ith `top`j object also has object-id `o`j`(`i`)`. It also has subobjects `<&a`j`_1(`i`) attr1 1>`, `<&a`j`_2(`i`) attr2 2>`, ..., `<&a`j`_30(`i`) attr30 30>`. However, the fusion mediator does not propagate all source "attr" objects to the mediator view. For example, the specification of Figure 6.4 accesses only the `attr1` and `attr2` objects.

For the experiment of Figure 6.5 we have varied the number of rules used in the mediator specification from 5 to 40 in steps of 5 rules at a time. For the experiment where the fusion specification has x rules the first five of them fuse information from the `top1` objects of `src1`,...,`src5`, the second five fuse information from the `top2` objects, and so on. The last five fuse information from the `top`$x/5$ objects. All rules access and copy to the mediator view the subobjects `attr1` to `attr10` of the source "top" objects. (For this experiment we keep the number of attributes to 10 as Figure 6.3 suggests.)

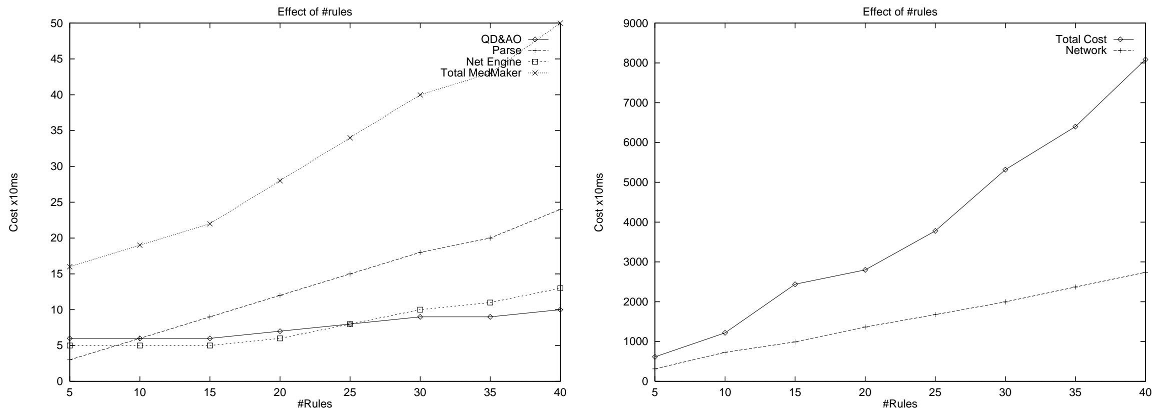

Figure 6.5: Performance of fusion mediator as a function of number of rules

The following single condition query that collects the object-id's of the top-level mediator objects is sent to the mediators.

```
<ans X> :- <X topmed {<attr1 1>}>.
```

MedMaker matches the query with the mediator specifications and develops a plan which sends to each source $x/5$ queries. The first query going to a source, say `src1`, retrieves the "key" subobjects of the `top1` objects that have an `attr1` subobject with value `1`. The second one retrieves the keys of the `top2` objects that have an `attr1` subobject with value `1`, and so on. Each one of these queries retrieves 25 keys because each wrapper exports 25 `top1` objects, 25 `top2` objects, etc, and every "top" object has a "key" and an `attr1` subobject with value `1`.

Figure 6.5 shows the performance of the fusion mediator described above as a function of the number of rules. The left diagram presents the cost of the QD&AO, the parser, the net engine cost (recall, the net engine cost does not include the cost of querying the sources), and the total mediator cost. As expected, the QD&AO and the net engine cost roughly follow a function of the the form $a + b(\textit{number of rules})$ where a and b are constants. The factor a is due to initialization costs. The factor b is justified because as the number of rules increases QD&AO has to do more matches and the datamerge graphs are bigger.

Note that we also plot the parse cost in Figure 6.5. Of course, not a lot can be said about the implementation of MedMaker's very typical Yacc based parser. However, it provides a point of reference for the cost of the other modules. For example, it is interesting to note that parsing is in general more expensive than QD&AO. Since the most expensive activity in both parsing and QD&AO is the allocation of memory for representing and transforming the query and the mediator specification we conclude that QD&AO does fewer memory allocation operations than the parser. Note that QD&AO does fewer memory allocations despite the fact that its output, i.e., the datamerge graph, is almost as large as the output of the parser, i.e., the datamerge graph has almost as many nodes as the parse tree of the mediator specification.

The right diagram of Figure 6.5 illustrates the network cost and the total cost, i.e., the sum of mediator cost, network cost, and source processing cost. Note that the network cost is almost linear in the number of rules. This is due to the fact that the network cost is proportional to the number of queries emitted by the mediator which in turn is equal to the number of rules of the mediator. Hence, the performance of MedMaker scales linearly with the number of rules in the common scenario tested by the experiment.

Before we proceed to experiments illustrating the performance as a function of the number of sources, attributes, or query conditions we note the following two conventions followed by our experiments and their presentation:

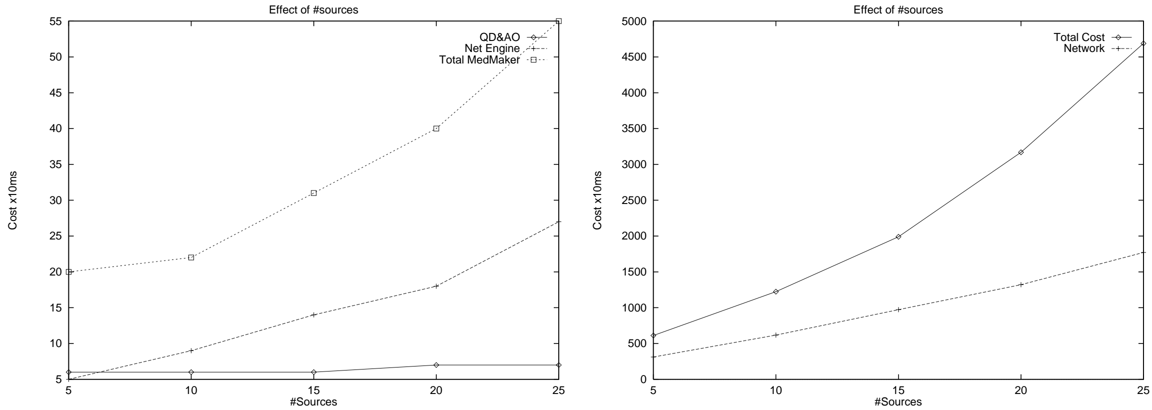

Figure 6.6: Performance of fusion mediator as a function of the number or participating sources

1. We do not plot in any of the figures the optimization cost because it is very small (always below 20ms). This is expected because the optimizer does not generate alternative plans.[4] Furthermore, we do not plot the parsing cost in any figure other than 6.5. Nevertheless, the optimization and parsing cost are parts of the "total MedMaker" and "total" costs.

2. We do not perform experiments with more than 40 rules because all our experiments have to observe a limit of around 60 queries to the sources for every plan. This is because the server support library dedicates a separate process to every query (so that the system does not fail when a query fails) but the current implementation does not kill the query service process until the session is finished.

Now we perform experiments for studying the effect of the number of sources, query conditions, and attributes on the query processing cost.

Effect of number of sources: Figure 6.6 corresponds to a fusion mediator with x sources and x rules, where x ranges from 5 to 25. The ith rule collects the `top1` objects of the ith source. Costs scale up almost linearly as the number of sources increases. Furthermore, by comparing Figure 6.5 with Figure 6.6 we see that there is no important difference in the network cost whether x rules access information from x sources or $x/5$ sources. This is expected because in both cases x conjunctive queries are executed sequentially. Had the mediator used disjunction to group together conjunctive queries directed to the same source it would be more efficient to have fewer sources.

Notice that the total cost is higher when x rules access x sources instead of $x/5$ sources. The reason is that in the former case more processes run, inducing more context switches. Had the synthetic sources run in different machines we would not have this performance discrepancy.

Effect of number of attributes: For this experiment we fix the number of rules at 10 and the number of sources at 5. Then the number of `attr` objects accessed by the rules and propagated to the "topmed" objects ranges from 5 to 25 in steps of 5. For example, in the step where the number of attributes is 5 the first rule of the fusion mediator specification (see Figure 6.4) is modified as shown in Figure 6.8 and in the step where the number of attributes is 10 is is modified as shown in Figure 6.9.

Figure 6.7 shows that the effect of the number of attributes on the performance of QD&AO is minimal. In particular, the QD&AO cost increases by 40% when the number of attributes grows from 5 to 25. The reason is that when the object patterns of the mediator specification mention many attributes they are larger

[4] Indeed, the reader may wonder why the optimizer takes 20ms to complete its straightfoward (relatively to the QD&AO) task. The reason is that the query strings that are sent to the sources are formulated in an inneficient way which involves many string copies.

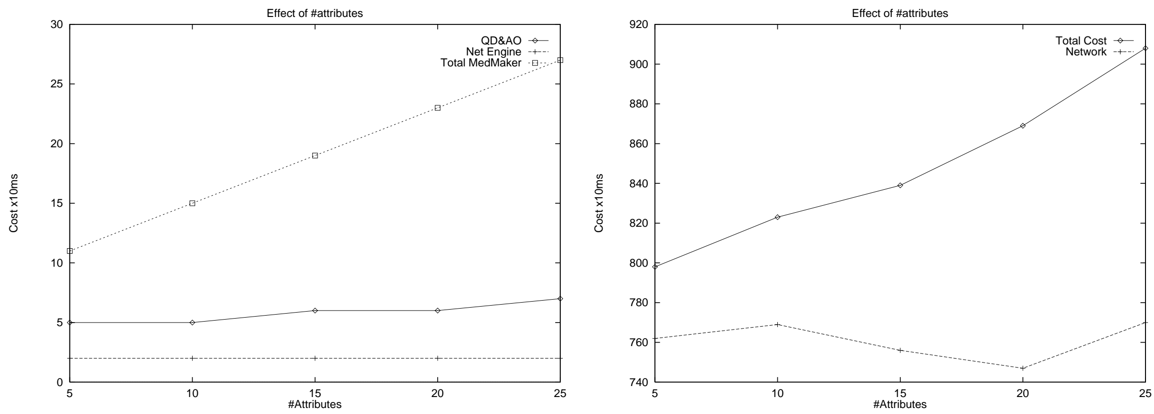

Figure 6.7: Performance of fusion mediator as a function of number of attributes

```
<&j(J) topmed { <&XA5 attr5 VA5>
                <&XA4 attr4 VA4>
                <&XA3 attr3 VA3>
                <&XA2 attr2 VA2>
                <&XA1 attr1 VA1>
              }> :-
<&X top1 {<&JOID key J>
                <&XA5 attr5 VA5>
                <&XA4 attr4 VA4>
                <&XA3 attr3 VA3>
                <&XA2 attr2 VA2>
                <&XA1 attr1 VA1>
              }>@src1.
```

Figure 6.8: Specification rule with 5 attributes

```
<&j(J) topmed { <&XA10 attr10 VA10>
                <&XA9 attr9 VA9>
                <&XA8 attr8 VA8>
                <&XA7 attr7 VA7>
                <&XA6 attr6 VA6>
                <&XA5 attr5 VA5>
                <&XA4 attr4 VA4>
                <&XA3 attr3 VA3>
                <&XA2 attr2 VA2>
                <&XA1 attr1 VA1>
              }> :-
<&X top1 {<&JOID key J>
                <&XA10 attr10 VA10>
                <&XA9 attr9 VA9>
                <&XA8 attr8 VA8>
                <&XA7 attr7 VA7>
                <&XA6 attr6 VA6>
                <&XA5 attr5 VA5>
                <&XA4 attr4 VA4>
                <&XA3 attr3 VA3>
                <&XA2 attr2 VA2>
                <&XA1 attr1 VA1>
              }>@src1.
```

Figure 6.9: Specification rule with 10 attributes

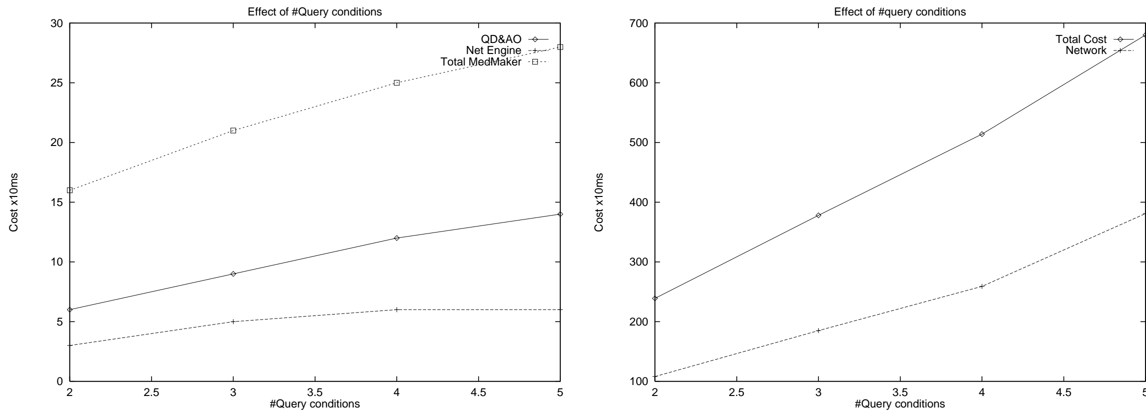

Figure 6.10: Performance of fusion mediator as a function of number of query conditions

and hence applying a unifier on them is more expensive. The total cost increases by 13% when the number of attributes grows from 5 to 25. The increase is primarily due to the fact that the queries sent to the sources mention more attributes and hence induce an increased "matching" activity. On the other hand the network cost is essentially unaffected by the number of attributes. (The fluctuations observed in Figure 6.7 are within an unimportant 3% of the ideal flat curve (though they look bigger because the offset of the vertical axis is 740) and are most probably due to time quantumization errors.) A flat network is expected because neither the number of queries emitted to the sources nor the size of the query results depend on the number of attributes. (Recall, the queries retrieve keys only.)

Overall the increase of the number of attributes causes small (if any) performance increases and we conclude that mediators with large numbers of attributes are efficient.

Effect of number of conditions: For this experiment we fix the number of sources to 5, the number of rules to 10, and the number of attributes to 10 and then we issue the following four queries which contain an increasing number of subobject conditions:

```
<ans X> :- <X topmed {<attr1 1> <attr2 2>}>.

<ans X> :- <X topmed {<attr1 1> <attr2 2> <attr3 3>}>.

<ans X> :- <X topmed {<attr1 1> <attr2 2> <attr3 3> <attr4 4>}>.

<ans X> :- <X topmed {<attr1 1> <attr2 2> <attr3 3> <attr4 4> <attr5 5>}>.
```

Note that each condition can match with more than one rule head, i.e., each condition can be satisfied by more than one sources, hence leading to the fusion query processing problems discussed in Chapter 5. We avoid these problems by modifying the mediator specification so that each subobject condition can match with a subobject of only one rule. In particular, only the first rule exports an `attr1` subobject, only the second rule exports an `attr2` subobject and so on up to `attr5`. Hence the mediator specification is modified as shown in Figure 6.11. In this way we avoid the fusion query processing problems (fusion query processing techniques are evaluated in the next section) and we study the effect of the number of conditions in a setting which is more similar to a conventional system where the schema information indicates where each condition should be pushed.

Figure 6.10 shows that the total cost and the network cost are linear to the number of query conditions. This is expected because the number of queries emitted to the sources is equal to the number of subobject

```
<&j(J) topmed {<&XA1 attr1 VA1>}>

        :- <&X top1 {<&JOID key J> <&XA1 attr1 VA1>}>@src1.

<&j(J) topmed {<&XA2 attr2 VA2>}>

        :- <&X top1 {<&JOID key J> <&XA2 attr2 VA2>}>@src2.

<&j(J) topmed {<&XA3 attr3 VA3>}>

        :- <&X top1 {<&JOID key J> <&XA3 attr3 VA3>}>@src3.

<&j(J) topmed {<&XA4 attr4 VA4>}>

        :- <&X top1 {<&JOID key J> <&XA4 attr4 VA4>}>@src4.

<&j(J) topmed {<&XA5 attr5 VA5>}>

        :- <&X top1 {<&JOID key J> <&XA5 attr5 VA5>}>@src5.
```

Figure 6.11: The fucion mediator specification for testing the effects of query conditions number

conditions in the query. In particular, the query retrieving `attr1` is sent to `src1`, the query retrieving `attr2` is sent to `src2` and so on. The QD&AO cost also increases when the number of conditions increases because every condition has to be separately matched to the rule heads. However, all cost increases are propotional or less than proportional to the number of conditions.

Overall, the presented experiments illustrate that MedMaker is efficient when the mediator specification indicates to which source each condition should be pushed. In particular, the costs scale up linearly (or less than linearly) as the number of rules, sources, attributes and query conditions increases. This scenario is similar to query decomposition in relational databases whereas the schema information indicates to which source each condition should be pushed. In the next section we study MedMaker's performance in cases where every condition may be pushed to more than one sources.

6.3 Evaluation of Optimizations for Fusion and Unknown Schema

Fusion mediators are inefficient when every condition has to be pushed at many sources. This is the case when we have mediators such as the one of Figure 6.12 (which is actually used for our experimentation.) In this section we show that MedMaker's policy to push all conditions to the sources is indeed more expensive than the two alternatives suggested in Section 5.6. For convenience, we summarize the two alternatives again and we also specify the way in which we tested each of them:

- **Use of a lexicon service:** Assume that the mediator can ask a lexicon service of a wrapper whether the wrapper exports (sub)objects with some label l. If the wrapper gives a negative answer then no query that asks for l is sent to this wrapper.

  MedMaker's implementation does not yet include lexicon services as described above. For the purposes of the experimentation we included in the QD&AO a function f, which is called whenever a condition is pushed to a wrapper that may not satisfy this condition, and decides whether a condition can be

```
<&X topmed {<&V1 V2 V3>}> :- <top1 {<&V1 V2 V3> <key X>}>@src1.

<&X topmed {<&V1 V2 V3>}> :- <top2 {<&V1 V2 V3> <key X>}>@src2.
```

Figure 6.12: Fusion mediators with two sources of unknown schema

pushed to this wrapper. For example, consider matching the condition `<topmed {<l 1>}>` with the first rule of Figure 6.12. From the structure of the rule it is not clear whether `<l 1>` should be pushed to `src1`. The function f will be called by MedMaker before the unification routine[5] matches the query condition label `l` to the label variable `V2` that appears in the head of a rule. In particular, the label `l` and the wrapper name `src1`, where `V2` comes from, will be given to f which decides whether it makes sense to push `l` to `src1`. If the decision is negative MedMaker fails the match of the condition with the first rule.

- **Push only one condition in every query:** Even after the use of the lexicon, every condition may have to be pushed to many sources. However, trying to push as many conditions as possible in every query leads to an exponential explosion described in Section 5.6. It is more efficient to send to each source one query for every condition. The rest of the fusion can be done at the mediator. The described "push only one condition" optimization is also not implemented but we simulated it for the purposes of the experimentation as explained below.

For comparing the three policies we use mediators such as those of Figure 6.12. We vary the number of sources from two (which is actually the one shown in Figure 6.12) to five. Correspondingly, the number of rules varies from two to five. Then the following query is issued

```
<ans X> :- <X topmed {<attr1 1> <attr2 2>}>.
```

Note that the `attr1` and `attr2` conditions have to be pushed to all sources. For comparing the three policies we set the sources so that label `attr1` appears only at the first source and the label `attr2` appears only at the second source and so on up to `attr5`. This information is known to the lexicon service functions but it is not explicitly shown in the specifications.

Performance of the "push maximum conditions" policy The implemented policy leads to x^2 datamerge rules where x is the number of sources (see Section 5.6 for a discussion on why so many datamerge rules are derived.) Each datamerge rule has two conditions; an `attr1` condition andan `attr2` condition. The evaluation of each datamerge rule requires one query, if both conditions are pushed to the same source, or two queries if they are pushed to different sources. Overall, $2x(x-1)+x$ queries are emitted, i.e., the number of emitted queries is quadratic in the number of sources. Note that only 2 queries (2 is the number of query conditions) return nonempty answers. Nevertheless the other $2x(x-1)+x-2$ queries also have a processing and network cost and hence the network cost and the total cost look like quadratic curves. The net engine cost also has a quadratic component in it the cost of processing the 2 nonempty results dominates the quadratic component. (Indeed, the quadratic component can be seen if we take a 60ms offset.)

The quadratic nature of these costs may prohibit scaling to large numbers of sources. We show next that the lexicon approach and the "push only one condition" approach scale do not have such scaling problems.

[5] see algorithm in Figure 5.3 of Chapter 5

```
<&X top V> :- <&X top V>@middle1

<&X top V> :- <&X top V>@middle2<&X top V> :- <&X top V>@middle1
```

Figure 6.13: Mediator `top`

Performance of the lexicon approach The lexicon approach (see Figure 6.15) has essentially a constant cost. (The fluctuations of the network cost are within 6% of the average and should be explained to errors in the measurements.) The constant cost is justified because regardless of the number of involved sources, MedMaker emits only two queries; the first query retrieves the `src1` objects with an `attr1` subobject and the second query retrieves the `src2` objects with an `attr2` subobject. We should keep in mind that the performance of the lexicon approach depends on our assumption that only one source has `attr1` and only has `attr2`. If both sources had both attributes then the performance of the lexicon policy would be the same with the performance of the "push maximum".

Performance of the "push only one condition" approach We simulate the "push only one condition" policy, for the purposes of the experiment, by splitting the mediator into three mediators. The two of them, named `middle1` and `middle2` run the specification of Figure 6.12. The third mediator, named `top`, uses the other two as sources, as shown in Figure 6.13, and it also has a lexicon service function saying that `attr1` comes only from `middle1` and `attr2` comes only from `middle2`. After we give the query, because of the lexicon service, only the datamerge rule that pushes `attr1` to `middle1` and `attr2` to `middle2` "survives". Hence, a `attr1` query is sent to `middle1` which decomposes it into `attr1` queries sent to the sources. Similarly `attr2` is pushed to `middle2`. Observe that the sources receive exactly the queries they would receive by the "push only one condition" policy. We calculate the cost of the policy by summing up the processing time at `src1` to `src5`, and the network costs in the communication between the "middle" mediators and `src1` to `src5`. Note that the network cost between the top mediator and the "middle" mediators is ignored because in an actual implementation of the "push only one condition" this communication would be done inside the mediator. We do not measure QD&AO costs, net engine, and optimizer cost because they will be very different in an actual implementation of the "push only one condition" policy.

The total and network cost of the "push only one condition" approach are proportional to the number of sources (see Figure 6.16.) This is expected because $2x$ queries are emitted by the mediator, where x is the number of sources.

Figure 6.17 compares the three policies. The lessons learned are the following:

- The lexicon policy is superior if each condition is satisfied by only one source. However, it presumes the existence of a lexicon service that knows which sources may satisfy a given condition.
- The "push only one condition" approach is a clear winner because it is applicable in all fusion scenarios. It is an open issue to seamlessly integrate the "push only one condition" policy, which is optimal for fusion queries, with the "push maximum conditions" policy which is optimal for conventional conjunctive queries and views.

We should keep in mind that the above results refer to a limited class of queries and mediator specifications as well as a very specific cost measure. Future work should address more complex queries and mediator specifications.

Note also that the experimental results are based on the assumption that the sources can support any query which is given to them. This will often not be the case. The next two chapters show how a query can be answered even though it may not be directly supported by the corresponding source. Nevertheless, we

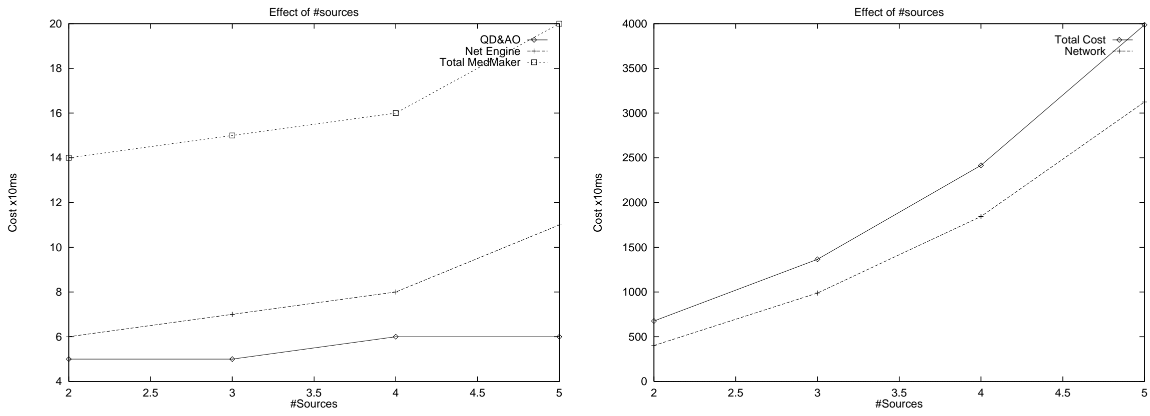

Figure 6.14: Performance of fusion mediator with unknown schema and two query conditions

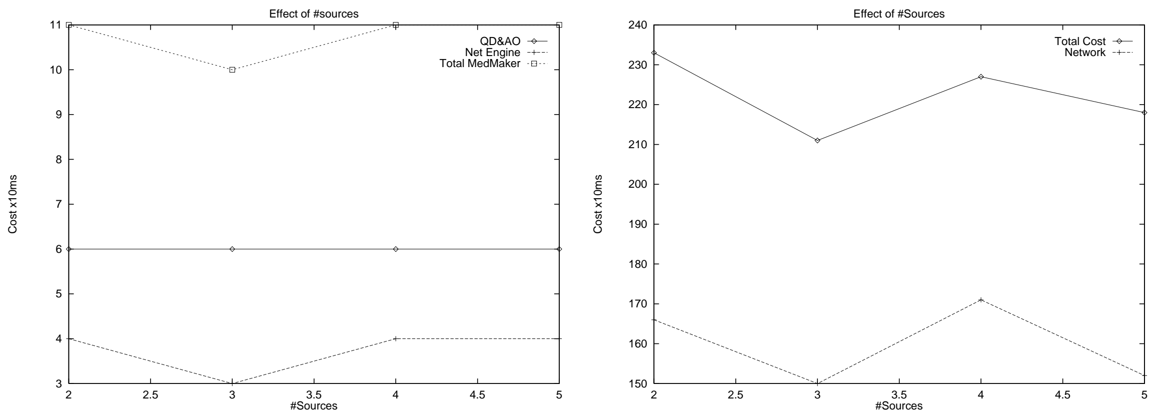

Figure 6.15: Performance of fusion mediator with lexicon service and two query conditions

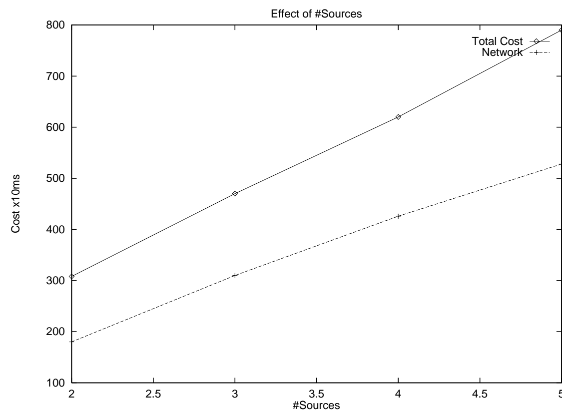

Figure 6.16: Performance of fusion mediator that pushes one condition in each query (total and net cost only)

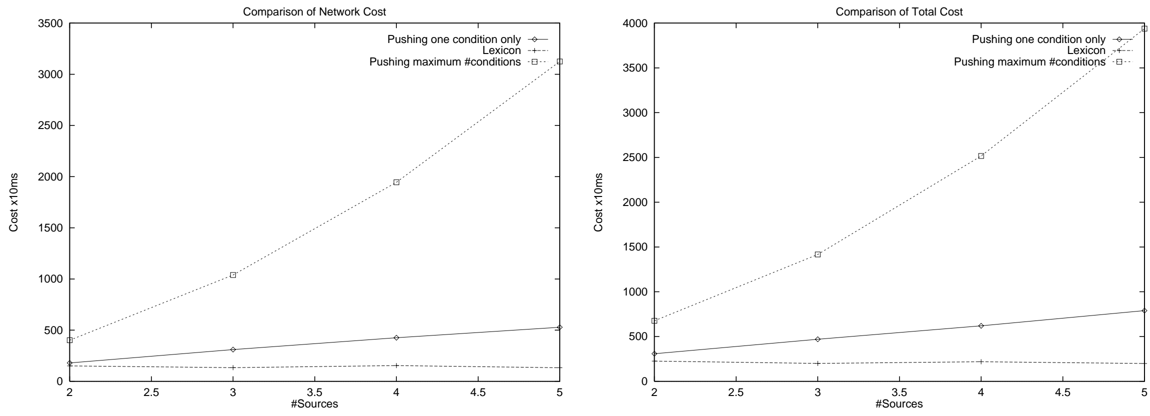

Figure 6.17: Comparison of the three alternatives for query processing with unknown schemas

provide limited guarantees about the performance in scenarios where the sources have limited capabilities. Future work should address the issue of performance in the presence of limited capabilities.

Chapter 7

Wrapper Generation

In this chapter we describe a wrapper generation system for TSIMMIS. We recall from Chapter 3.1 that *wrappers* are the software components that convert data and queries from one model to another. Wrappers receive common model queries (MSL in our case) from mediators or from the applications. 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 model understood by the application.

During the first stages of the TSIMMIS project [PGMW95] we developed hard-coded wrappers for a variety of sources, including legacy systems (see Chapter 3.1. We have observed, like everyone who has built a wrapper, that writing them involves a lot of effort [A+91, C+95, 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 developed a *wrapper implementation toolkit* for rapidly building wrappers. The toolkit contains a library of commonly used functions, such as for receiving queries from the application and packaging results. We've already discussed some of these functions in Chapter 3.1. The toolkit also contains a facility for translating queries into source-specific commands and queries, and for translating results into a model useful to the application.

We focus on the query translation component of the toolkit, which we refer to as the *converter*. (In Section 7.4 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 native query may refer to any program used to access and retrieve information from the underlying source.

EXAMPLE 7.0.1 Note, the ideas behind wrapper generation are not specific to the MSL query language which is used as the query language for TSIMMIS. Indeed, the introductory examples assume that we use the relational data model and the SQL query language.

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:

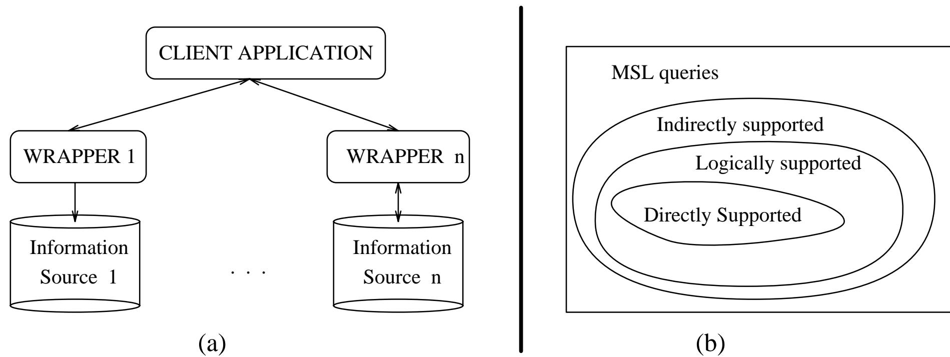

Figure 7.1: (a) Accessing information through wrappers (b) Supported queries.

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

This query matches the template with the bindings `$X = "emp"`, `$Y = "name"` and `$Z = "'toy'"`. Given the match, the actions associated with the template would then generate the necessary native query to do the actual search on the source. For example, if the underlying source were a file system the actions could produce a "grep" command to search for the string `$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 7.0.1 illustrates a very simple template matching facility that could be easily implemented using Yacc-like tools. 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" the queries which they are translating, then they can successfully handle many more queries.

EXAMPLE 7.0.2 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 7.0.3 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 might be able to 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 7.0.2 the converter must understand the notion of selection and conjunctive logical expressions. In Example 7.0.3 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.

Of course, our converters are targeted for the MSL query language. (SQL was only used in our initial examples to motivate our ideas.) The converter is configured with templates written in a *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 3.1.) 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 (7.1.b) graphically shows the types of accepted queries. Thus, though QDTL descriptions look like Yacc grammars – suitably modified for the description of queries – our converter handles a much larger class of queries than the class of directly supported queries that would be handled by a traditional parsing facility such as Yacc. Furthermore, our converter introduces the following innovations:

- A designer can define the functionality of each source succinctly and clearly 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 7.3.)
- 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, may permit only limited sets of queries. The automatic extension of query abilities allows us to bring different sources to the same level of functionality and then to integrate them more easily.
- The converter, together with the other functions of the toolkit, makes it possible to implement wrappers rapidly.

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

In Section 7.1 we give a detailed example that shows how QDTL is used and the types of queries it can handle. Indirectly supported queries and the notion of query subsumption are further discussed in Section 7.2, while Section 7.3 introduces additional powerful QDTL features such as nonterminal templates and metapredicates. In Section 7.4 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 7.5 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. Finally, Section 7.6 discusses related work.

7.1 A Detailed Example

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.,

   (L30) `lookup -ln Smith`

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

   (L31) `lookup -ln Smith -fn John`

3. Retrieve all person records by issuing the command

   (L32) `lookup`

The queries accepted by the lookup facility can be described in our Query Description and Translation Language (QDTL). As discussed earlier, a QDTL description consists of a set of templates with associated actions. Below we present description D1 which consists of three *query templates* QT1.1, QT1.2, and QT1.3. For simplicity, we do not yet give 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 Q33 is directly supported by template QT1.1 by substituting `P` for `O` and `'Smith'` for `$LN`.

  ```
  (Q33) *P :- <P person {<L last_name 'Smith'>}>
  ```

- *Logically supported queries.* A query q is logically supported by template t if q is logically equivalent to some query q' which is 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 of these queries are equivalent to the following query Q34, which is directly supported by template QT1.2:

(Q34) `*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 7.2; for now we present an example. Consider the following query:

  (Q35) `*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

  (Q36) `*Q :- <Q person {<last_name 'Smith'>}>`

  This means that the answer to Q36 contains all the information necessary to answer Q35. Thus, the converter matches Q35 to template QT1.1 as if it were Q36, 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 Q36 to produce the result of query Q35.

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

7.1.1 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 L30, L31, and L32.

```
(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 Q34:

```
*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 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.

7.2 Query Subsumption

In Section 7.1 we said that query Q35 was subsumed by Q36 because the former had an additional condition on the "role" subobject. Thus query Q35 selects a subset of the objects obtained by the subsuming query Q36.

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 Q37 that retrieves the `first_name` subobjects of `person` objects with last name `'Smith'`

```
(Q37) *F :- <O person {<F first_name X> <last_name 'Smith'>}>
```

Query Q37 is subsumed by the following query Q38, which retrieves the full `person` objects of persons with last name `'Smith'` and an unspecified first name.

```
(Q38) *O :- <O person {<F first_name X> <last_name 'Smith'>}>
```

Notice that Q37 and Q38 have exactly the same conditions. However, Q38 subsumes Q37 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.[1]

Definition 7.2.1 (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'.

□

Definition 7.2.2 (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] □

Definition 7.2.3 (Indirect support) A query q is indirectly supported by a query q' if

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

□

[1] Note, we have already defined containment using MSL patterns in Chapter 5. However, the following definitions are independent of MSL because we want to make clear the applicability of these ideas to any (object) model.

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

A filter query is formally defined as follows:

Definition 7.2.4 (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 $\langle cond(Y) \rangle$ is reachable from Y, and
- $\texttt{answer}_f(x)$ holds if and only if $\texttt{answer}(x)$ also holds

□

We will say that a template t indirectly supports a query q if t directly supports a query q' that indirectly supports q. Note, query subsumption does not necessarily imply indirect support. For example, consider the following query

(Q39) `*F :- <person {<F first_name X>}>`

that subsumes Q37, since it retrieves all `first-name` objects. However, Q39 does not indirectly support Q37, since given a `first-name` object in the result of Q39, we can not tell whether it is a subobject of a `person` with `last_name 'Smith'`.

7.2.1 Maximal Supporting Queries

Notice that given a query q there may be more than one query that supports q, and these queries may not be logically equivalent. For example, query Q35 on page 76 is supported by query Q36 and also by the query

(Q40) `*O :- <O person V>`

that retrieves all `person` objects.

Note, query Q40 also subsumes query Q36. Thus, Q36 derives fewer unnecessary answers than Q40. From a performance point of view it is better for the wrapper to send Q36 to the source (after the necessary transformation to a native query) rather than Q40, because the former contains more conditions of the original query Q35. Indeed, for our example, query Q36 is the best query directly supported by description D1 that supports query Q35 because Q36 pushes to the source as many conditions as possible. We will say that Q36 is a *maximal supporting query* for Q35.

Definition 7.2.5 (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.

□

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 Q34. This query has two maximal supporting queries:

```
(Q41) *O :- <O person {<last_name 'Smith'>}>
(Q42) *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. Once the converter selects a maximal supporting query, it executes the actions associated with that particular maximal query. 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.

7.3 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 template that define n. For example, the query

```
(Q43) *O :- <O person {<last_name 'Smith'> <role 'professor'>}>
```

is directly supported by template QT4.1 because Q43 matches with the expansion

```
(E44) *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.

7.3.1 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 associates 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) ; }
(NTAC5.5) _OptFN ::= { $_OptFN = ''; }
(NT5.6) _OptRole ::= <role $R>
(AC5.6)                 { sprintf($_OptRole, '-role %s',$R) ; }
(NTAC5.7) _OptRole ::= { $(_OptRole.role) = '' ; }
```

As discussed earlier, query Q43 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.

7.3.2 Recursion

Nonterminal templates may contain nonterminals recursively. This flexibility allows us to describe infinite sets of expansions. The following description — which 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 Q43. 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 Q43.

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.

7.3.3 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 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 nonterminal template NT6.2:

```
(NT9.2) _Cond ::= <$Label $Value> _Cond personsub($Label)
```

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 QT6.1 with the nonterminal template NT9.2 and then with the nonterminal template NT6.3 we get:

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

Matching this expansion with query Q30 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.

7.4 Wrapper Architecture

Figure 7.2 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 7.2. 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 7.2). 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 7.4.1 discusses the converter architecture in more detail. Then, Subsection 7.4.2 discusses the extractor, while Subsection 7.4.3 discusses the filter processor.

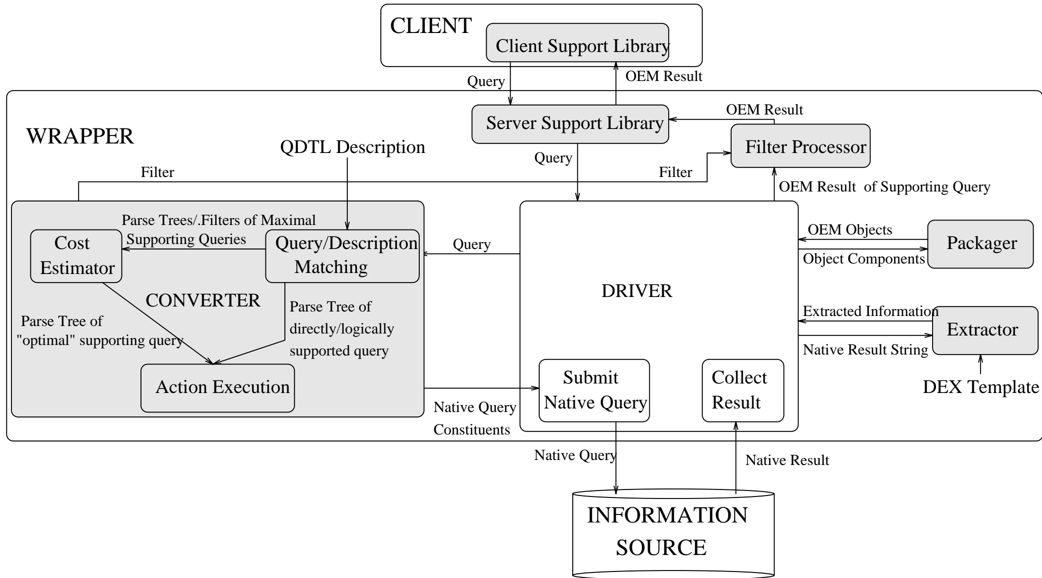

Figure 7.2: The Architecture of the Translator

7.4.1 Converter Architecture

To illustrate, let us assume that the converter is given description D5 which directly supports query Q43 (see Section 7.3). The *query/description matching* component of the converter produces the parse tree of Figure 7.3 which 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 query, 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 7.3 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, NTAC5.5, 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 which chooses one of the parse trees either by an *arbitrary choice* or by *cost-based* selection. The latter 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.

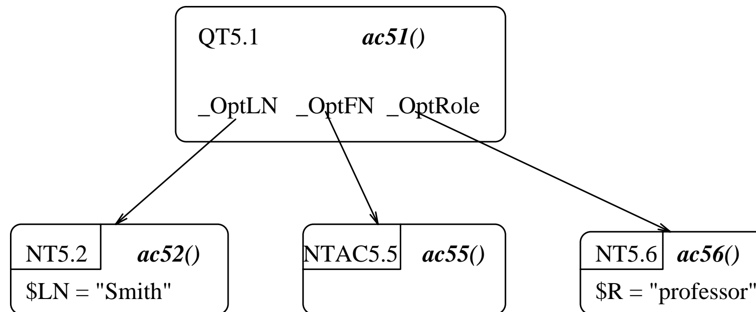

Figure 7.3: The parse tree

7.4.2 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` (which 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.

7.4.3 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.

7.5 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.

Our solution is based on the OEM-to-relational reduction that was presented in Section 4.4.1. In particular, the MSL queries and QDTL descriptions are actually converted to relational terms using the relations *top*, *object*, and *member*. Of course, the objects in the underlying sources are not converted. We recall in the following section the basic rules of reducing OEM to relational and we enhance them for translating QDTL descriptions to Datalog programs. Note, we drop the *SRC* field (see Section 4.4.1) because the MSL query and the template refer to the same source always.

7.5.1 Reduction of OEM to Relational

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

`object(oid, l, v)`

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

`object(oid, l, set)`

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

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

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

`top(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 7.5.1 Consider the query

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

The above query selects all 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 a normal-form MSL query to a relational form is given in Appendix B. A QDTL description is similarily reduced to relational description. We illustrate the translation via an example.

EXAMPLE 7.5.2 Consider description D6 from Subsection 7.3.2. 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`. □

7.5.2 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 if a corresponding filter query exists. Thus, a supporting query is found in two steps:

1. find a subsuming query, and
2. find the corresponding filter.

We extend an existing algorithm [Ull89] that checks containment, to answer the first step. We refer to the containment algorithm from [Ull89] as QinP. We extend the algorithm to handle the second step.

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 and also the native query constituents for the underlying source. We describe in detail the extended algorithm X-QinP in the Appendix C. We continue with examples to illustrate the required extensions.

EXAMPLE 7.5.3 (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 Q45 which selects all `person` objects that have a subobject with label `last_name` and value '`Smith`':

```
(Q45)  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.

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

(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 (E46).

(E46): `answer(P) :-top(P), object(P, person, set), member(P,X), object(X,$L,$V)`

(E46) is identical to query Q45 by substituting appropriately variables and place holders. Thus, $D11$ directly supports $Q45$.

Alternatively, consider query Q47 that picks `person` objects with specified values of subobjects `last_name` and `ssn`.

```
(Q47) answer(O) :- top(O), object(O, person, set), member(O,L),
                   object(L, last_name, 'Smith'), member(O,S),
                   object(S, ssn, '123')
```

Description D11 does not directly support query Q47 because the query imposes selection conditions on two subobjects whereas the description supports queries with only single subobject selections. However, E46 produces two queries that indirectly support Q47:

- E48 enforces the selection condition on subobject `last_name`.

```
(E48): answer(O) :- top(O), object(O, person, set), member(O,L),
                    object(L, last_name, 'Smith')
```

- E49 enforces the selection condition on subobject `ssn`.

```
(E49): 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.

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 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 $Q45$ and expansion (E46) in Example 7.5.3 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

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

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. Section 14.5 in [Ull89] gives an algorithm (Algorithm QinP) to answer exactly this question.

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.

EXAMPLE 7.5.4 (Applying Algorithm QinP) Consider query Q47 from Example 7.5.3.

```
(Q47)  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 Q47, Algorithm QinP does the following: First the algorithm "freezes" Q47, i.e., it replaces each variable in each subgoal of Q47 by a corresponding "frozen" constant and puts the resulting frozen facts in a database DB(Q47). 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 overbars 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(Q47) to check if the program derives the frozen head of Q47, 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, ($E48$) in Example 7.5.3). The algorithm does not find the selection conditions that are not enforced by each subsuming query (for instance, ($E48$) 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 7.5.5 (Multiple Subsuming Queries) Example 7.5.3 shows that query Q47 is indirectly supported by Description D11 (page 85) via two subsuming queries ($E48$) and ($E49$). We discuss in more detail how to obtain ($E48$).

```
(E48):  answer(O) :- top(O), object(O, person, set), member(O,L),
                     object(L, last_name, 'Smith')
```

[5] The containment results hold in the presence of constant placeholders.

($E48$) is obtained by algorithm X-QinP, because program P(D11) derives the frozen head of query Q47 using frozen base facts `top(ō)`, `object(ō,person,set)`, `member(ō,l̄)`, and `object(l̄,last_name,'Smith')`. ($E49$) is obtained similarly. As guaranteed by extended algorithm X-QinP, ($E48$) and ($E49$) are *maximal*. □

Note, in Example 7.5.5 the subsuming queries ($E48$) and ($E49$) do not use all the frozen facts obtained by freezing the target query Q47. 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 ($E48$) 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 7.5.6 (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 consists of all the conditions in the input query, of which some conditions could 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 ($E48$).

```
*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 description 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.

7.6 Related Work

Integration of heterogeneous information sources has attracted great interest from the database community[Wie92, LMR90, T+90, Gup89, A+91, C+95, 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 requiring 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" (*sic*). Indeed, there are existing algorithms 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 not even 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 [RSU95, C+95] 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 and it is discussed in the related work Section of the next Chapter.

Chapter 8

A Powerful Capabilities-Based Rewriting Algorithm

The converter presented in Chapter 7 has the ability to indirectly support a query by finding an appropriate filter and a subsuming query that is supported by the source. However, it is often the case that the indirectly supported query cannot be computed unless we combine multiple directly supported queries. The main contribution of this chapter is the enhancement of the capabilities extension technology so that an indirectly supported query may be computed using a set of supported queries.

In particular, this chapter presents a *Capabilities-Based Rewriter* (CBR), which, given a *target query* and descriptions of the wrappers query capabilities, it can combine multiple supported queries in order to compute the target query. Note, the indirect query support abilities of the algorithms of Chapter 7 are not needed in a system that features the algorithms of this chapter. We also introduce a capabilities description language that describes relational queries and is strictly more expressive than QDTL (modulo the data model differences.) The additional expressiveness results in exact representations of the source power and consequently in computations that take full advantage of the sources power. Note, we use the relational model in this chapter for three reasons:

1. As was already shown in Section 7.5, the problem of indirect support in OEM is easily reduced to the problem of indirect support in the relational model. Hence all the results of this chapter can be easily adapted for the OEM model.
2. The increased complexity of the algorithms in this chapter necessitates that we present them using the relational model which is the model that is actually used by the algorithms.
3. The work described in this chapter was done for IBM's Garlic system, which integrates multiple multimedia information sources, and is based on an extension of the relational model.

8.1 The Relational Query Description Language(RQDL)

In this section, we present a language to describe the supported queries for each wrapper. The language describes conjunctive queries in the relational model (unlike QDTL which describes MSL queries.) We inherit from QDTL the use of nonterminals for the description of arbitrarily long queries. There are also a number of novel issues:

- RQDL can express projection capabilities.
- RQDL can describe sets of supported queries without referring to a specific schema. In contrast, the reduction of QDTL to the relational model (see Section 7.5.1) refers to a schema consisting of the

object, member, and top relations. The fact that QDTL is built on top of a specific schema — though it does not look so at first sight — is the reason for some of its shortcomings, *e.g.*, the inability to describe projection capabilities.

- RQDL nonterminals have parameters which stand for arbitrarily large sets of variables and greatly increase the expressive power of the language. For example, RQDL can describe the capabilities of a powerful SQL system that supports all conjunctive queries over its schema while QDTL can not [PV].

We introduce the basic language features in Section 8.1.1, followed in Sections 8.1.2 and 8.1.3 with the extensions needed to describe infinite query sets and to support schema-independent descriptions. Section 8.1.4 introduces a normal form for queries and descriptors that increases the expressiveness of the language. The complete syntax and semantics of the language appears in Appendix D.4.

8.1.1 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, let us consider a modification of the "lookup" facility that was introduced in Chapter 7. It 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':

(Q50) answer(FN,LN,D,O,M) :- emp(FN,LN,D,O,M), prefix(FN,'Rak'), prefix(LN,'Aggr')

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: *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 (Q50) the distinguished variables are FN, LN, D, O and M and the join variables are FN and LN.

Description (D51) is an RQDL specification of lookup's query capabilities. The identifiers starting with $ ($FP and $LP) are constant placeholders — identical to QDTL's 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 (QT51.3) describes query (Q50) 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'.

(D51) (QT51.1) answer(F,L,D,O,M) :- emp(F,L,D,O,M)
(QT51.2) answer(F,L,D,O,M) :- emp(F,L,D,O,M), prefix(L, $LP), _isalpha($LP)

[1] The difference from the "lookup" as described in Chapter 7 is that now we also consider prefix searches.

[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.

```
(QT51.3) 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. Optionally *reorder* the template's subgoals.

8.1.2 Descriptions of Large and Infinite Sets of Supported Queries

RQDL can describe arbitrarily large sets of templates (and hence queries) when extended with nonterminals similar 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 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 (Q52), which retrieves the data for employees whose office contains the strings `'alma'` and `'B'`.

```
(Q52) answer(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,'alma'), substring(O,'B')
```

(D53) uses the nonterminal `_Cond` to describe the supported queries. In this description the query template (QT53.1) is supported by nonterminal templates such as (NT53.1).

```
(D53) (QT53.1) answer(F,L,D,O,M) :- emp(F,L,D,O,M), _Cond(F,L,D,O,M)
      (NT53.1) _Cond(First,L,D,O,M) : substring(First, $FS), _Cond(First,L,D,O,M)
      (NT53.2) _Cond(F,Last,D,O,M) : substring(Last, $LS), _Cond(F,Last,D,O,M)
      (NT53.3) _Cond(F,L,Dept,O,M) : substring(Dept, $DS), _Cond(F,L,Dept,O,M)
      (NT53.4) _Cond(F,L,D,Office,M) : substring(Office, $OS), _Cond(F,L,D,Office,M)
      (NT53.5) _Cond(F,L,D,O,Mgr) :  substring(Mgr, $MS), _Cond(F,L,D,O,Mgr)
      (NT53.6) _Cond(F,L,D,O,M) :
```

To see that description (D53) describes query (Q52), we expand `_Cond(F,L,D,O,M)` in (QT53.1) with the nonterminal template (NT53.4) and then again expand `_Cond` with the same template. The variable `Office` of (NT53.4) unifies with `O` during the expansion. The `_Cond` subgoal in the resulting expansion is expanded by the empty template (NT53.6) to obtain the expansion (E54).

```
(E54) answer(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,$OS),
                           substring(O,$OS1)
```

Note, before a template is used for expansion, all its variables are uniquely renamed. Hence, the second occurrence of placeholder `$OS` of template (NT53.4) is renamed to `$OS1` in the above expansion. The above expression describes the query (Q52), *i.e.*, the placeholders and variables of (E54) can be mapped to the constants and variables of (Q52).

8.1.3 Schema Independent Descriptions of Supported Queries

Description (D53) 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. For example, if we match the template

`answer(_R) :- r(_A)`

with the the query

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

the vector _*R* matches with the variable list `[X,Y,Z]` and _*A* matches with `[X,Y,Z,W]`. We represent vectors in our examples by identifiers starting with an underscore (_) followed by a capital letter.[3] In addition, we provide two built-in metapredicates to relate vectors and attributes: **_subset** and **_in**. The metapredicate **_subset**(_*R*,_*A*) succeeds if each variable in the list that matches with _*R* appears in the list that matches with _*A*. The metapredicate **_in**(X, _*A*) succeeds if the variable that matches with X appears in the variable list that matches with _*A*.

For example, consider a wrapper called `file-wrap` that accesses plain UNIX files and translates them into tables. 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. (D55) 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.

```
(D55) (QT55.1) answer(_R) :- $Table(_A), _Cond(_A), _subset(_R, _A)
      (NT55.1) _Cond(_A) :  _in(X, _A), substring(X, $S), _Cond(_A)
      (NT55.2) _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 (Q56) is described by (D55).

```
(Q56) answer(L,D) :- emp(F,L,D,O,M), substring(O,'alma'), substring(O,'B')
```

First, we expand (QT55.1) by replacing the nonterminal `_Cond` with (NT55.1). Then again we expand the result with (NT55.1), and finally with (NT55.2), thus obtaining expansion (E57).

```
(E57) answer(_R) :- $Table(_A), _in(X,_A), substring(X,$S),
                    _in(X1,_A), substring(X1,$S1), _subset(_R,_A)
```

Expansion (E57) describes query (Q56) because there is mapping of variables, vectors, and placeholders of (E57) 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]`, `$S` to `'alma'`, `$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**(O, [F,L,D,O,M]) succeeds because `O` is in the variable list, and **_subset**([L,D], [F,L,D,O,M]) succeeds because `[L,D]` is a "subset" of `[F,L,D,O,M]`.

[3]Nonterminals also start with an underscore and a capital letter. Vectors are distinguished from nonterminals by their position in the template.

Vectors are useful even when the schema is known as the specification may be repetitious, as in description (D53). In our running example, even if the attributes of the tables are known, we save effort by not having to mention explicitly the columns and the table names when saying that a substring condition can be placed on any column of any tuple. Furthermore, we do not have to update the specification whenever the schema changes.

8.1.4 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 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 (Q58) 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`.

(Q58) `answer(FL,D) :- emp(FL,FL,D,O,'Y')`

(Q58) should not be described by the description (D55). Nevertheless, we are still able to construct the expansion (E59) that erroneously matches with the query (Q58) if we map $_A$ to `[FL,FL,D,O,'Y']` and $_R$ to `[FL,D]`.

(E59) `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 (Q60) is the normal form of (Q58).

(Q60) `answer(FL1,D) :- employee(FL1,FL2,D,O,M), FL1=FL2, M='Y'`

Description (D55) does not describe (Q60) because (D55) does not support the equality conditions that appear in (Q60). Description (D61) supports equality conditions on any column and equalities between any two columns: (NT61.2) describes equalities with constants and (NT61.3) describes equalities between the columns of our table.

```
(D61) (QT61.1) answer(_R) :- $Table(_A), _Cond(_A), _subset(_R, _A)
      (NT61.1) _Cond(_A) :  _in($Position,X,_A), substring(X, $S), _Cond(_A)
      (NT61.2) _Cond(_A) :  _in($Position1,X,_A), X=$C, _Cond(_A)
      (NT61.3) _Cond(_A) :  _in($Pos1,X,_A), _in($Pos2,Y,_A), X=Y, _Cond(_A)
      (NT61.4) _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.

8.2 The Capabilities-Based Rewriter

The Capabilities-Based Rewriter (CBR) determines whether a target query q is directly supported by the wrapper description. If not, the CBR determines whether q can be computed by combining a set of supported queries using selections, projections and joins. In this case, the CBR will produce a set of plans for evaluating the query. Note, CBR is more powerful than the query converter of TSIMMIS (described in Chapter 7) because it can indirectly support a query using a combination of multiple supported queries.[4]

[4] Indeed, all the examples presented in this section cannot be handled by the query converter of Chapter 7.

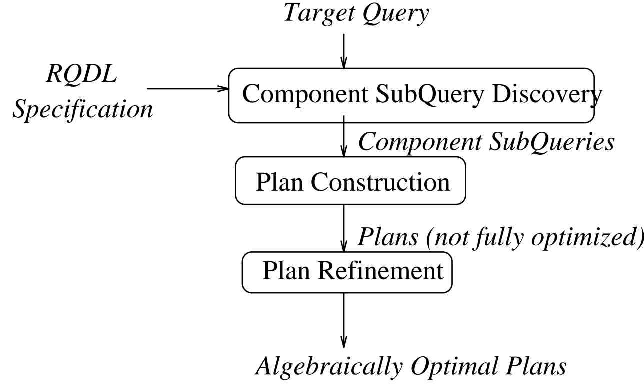

Figure 8.1: The CBR's components

The CBR consists of three modules, which are invoked serially (see Figure 8.1):

- **Component SubQuery (CSQ) Discovery:** finds supported queries that involve one or more subgoals of q and export one or more of the variables that appear in the subgoals. The CSQs that are returned contain the largest possible number of selections and joins, and do no projection. In this way the CBR pushes as much work as possible to the sources and exploits their functionality. All other CSQs are pruned. This prevents an exponential explosion in the number of CSQs.
- **plan construction:** produces one or more plans that compute q by combining the CSQs exported by CSQ Discovery. The plan construction algorithm is based on query subsumption and has been tuned to perform efficiently in the cases typically arising in capabilities-based rewriting.
- **plan refinement:** refines the plans constructed by the previous phase by pushing as many projections as possible to the wrapper.

EXAMPLE 8.2.1 Consider query (Q62), 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 (D61). Recall that for presentation purposes we use the more compact unnormalized forms of queries and descriptions.

(Q62) `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:

(Q63) $\texttt{answer}_{63}$`(F0,L0,D,O0) :- emp(F0,L0,D,O0,'Y')`
(Q64) $\texttt{answer}_{64}$`(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 (Q63) but export a different subset of the variables `F0`, `L0`, `D`, and `O0` (likewise for (Q64). The CSQs that export fewer variables are pruned hence reducing significantly the run time of the algorithm.

The plan construction module detects that a join on `D` of $\texttt{answer}_{63}$ and $\texttt{answer}_{64}$ produces the required `answer` of (Q62). Consequently, it derives the plan (P65).

(P65) `answer(F0,L0,O1) :-` $\texttt{answer}_{63}$`(F0,L0,D,O0),` $\texttt{answer}_{64}$`(F1,L1,D,O1,M1)`

Finally, the plan refinement module detects that variables `O0`, `F1`, `L1`, and `M1` in $\texttt{answer}_{63}$ and $\texttt{answer}_{64}$ are unnecessary. Consequently, it generates the more efficient plan (P68).

```
(Q66) answer66(F0,L0,D) :- emp(F0,L0,D,O0,'Y')
(Q67) answer67(D,O1) :- emp(F1,L1,D,O1,M1), substring(O1, 'B')
(P68) answer(F0,L0,O1) :-  answer66(F0,L0,D),  answer67(D,O1)
```

□

The CBR's goal is to produce all *algebraically optimal* plans for evaluating the query. An algebraically optimal plan is one in which any selection, projection or join that can be done in the wrapper is done there, and in which there are no unnecessary queries. Note, algebraic optimality generalizes the notion of maximal supporting queries (see Section 7.2.1) for the case that we combine multiple supported queries. The formal definition of algebraic optimality is the following:

Definition 8.2.1 (**Algebraically Optimal Plan** P) A plan P is algebraically optimal if there is no other plan P' such that for every CSQ s of P there is a corresponding CSQ s' of P' such that the set of subgoals of s' is a superset of the set of subgoals of s (*i.e.*, s' has more selections and joins than s) and the set of exported variables of s is a superset of the set of exported variables of s' (*i.e.*, s' has more projections than s.) □

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

8.3 CSQ Discovery

The CSQ discovery module takes as input a target query and a description. It has many similarities to the QDTL converter: it is essentially a rule production system where the templates of the description are the production rules and the subgoals of the target query are the base facts. The CSQ discovery module uses bottom-up evaluation because it is guaranteed to terminate even for recursive descriptions [Ull89]. However, bottom-up derivation often derives unnecessary facts, unlike top-down. We use a variant of *magic sets rewriting* [Ull89] to "focus" the bottom-up derivation. Furthermore, the vectors greatly increase the complexity of the algorithm as well as the size of its output, *i.e.*, the set of derived CSQs. To further reduce the set of derived CSQs we develop two CSQ pruning techniques as decsribed in Sections 8.3.2 and 8.3.3. Reducing the number of derived CSQs makes the CSQ discovery more efficient and also reduces the size of the input to the plan construction module.

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

EXAMPLE 8.3.1 Consider query (Q52) and description (D53) from page 92. 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 query variables, e.g. `F,L,D,O,M`, into similarly named constants, e.g. `f,l,d,o,m`. Actual constants like `'B'` are in single quotes.

In the first round of derivations template (NT53.6) derives fact `_Cond(F,L,D,O,M)` without using any base fact (since the template has an empty body). Hence, the set of facts underlying the derived fact is empty. Variables are allowed in derived facts for nonterminals. The semantics is that the derived fact holds for any assignment of frozen constants to variables of the derived fact.

In the second round many templates can fire. For example, (NT53.4) 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 (NT53.6) again and produce _Cond(F,L,D,O,M) but we do not retain it since its set of underlying facts is equal to the version of _Cond(F,L,D,O,M) that we have already produced.

Eventually, we generate answer(f,l,d,o,m) with set of underlying facts {emp(f,l,d,o,m), substring(o, 'alma'), substring(o,'B')}. Hence we output the CSQ (Q52), which, incidentally, is the target query.

The above process can produce an exponential number of facts. For example, we could have proved _Cond(o,L,D,O,M), _Cond(F,o,D,O,M), _Cond(o,o,D,O,M), and so on. In general, assuming that emp has n columns and we apply m substrings on it we may derive n^m facts. Magic-sets can remove this source of exponentiality by "focusing" the nonterminals. Applying magic-sets rewriting and the simplifications described in Chapter 13.4 of [Ull89] we obtain the following equivalent description. We show only the rewriting of templates (NT53.4) and (NT53.6). The others are rewritten similarly.

```
(D69) (QT69.1) answer(F,L,D,O,M) :- emp(F,L,D,O,M), _Cond(F,L,D,O,M)
      (NT69.4) _Cond(F,L,D,Office,M):mg_Cond(F,L,D,Office,M),substring(Office,$OS),
                                     _Cond(F,L,D,Office,M)
      (NT69.6) _Cond(F,L,D,O,M) : mg_Cond(F,L,D,O,M)
      (MS69.1) mg_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 in a way similar to a top-down strategy and thus derives only relevant facts. □

8.3.1 Derivations Involving Vectors

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

For example, let us fire template (NT55.1) of (D55) on the base facts produced by query (Q52). Assume also that (NT55.2) 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 (D55).

```
(D70) (QT70.1) answer(_R) :- $Table(_A), _Cond(_A), _subset(_R, _A)
      (NT70.1) _Cond(_A):mg_Cond(_A),_in($Position,X,_A),substring(X,$S),_Cond(_A)
      (NT70.2) _Cond(_A) :  mg_Cond(_A)
      (MS70.1) mg_Cond(_A) :  $Table(_A)
```

When rules (NT70.1) and (NT70.2) fire, the first subgoal instantiates variable _A to [f,l,d,o,m] and they derive only _Cond([f,l,d,o,m]). Thus, magic-sets caused _A to be bound to the only vector of interest, namely [f,l,d,o,m]. Note a program that derives facts with unbound vectors may not be problematic because no metapredicate may use the unbound vector variable. However we take a conservative approach and consider only those programs that produce facts with only bound vector variables. Magic-sets rewriting does not always ensure that derived facts have bound vectors. In the rest of this section we describe sufficient conditions for guaranteeing the derivation of facts with bound vectors only. First we provide a condition (Theorem 8.3.1) that guarantees that a program (that may be the result of magic rewriting) does not derive facts with unbound vectors. Then we describe a class of programs that after being magic rewriteen satisfy the condition of Theorem 8.3.1.

Theorem 8.3.1 *A program will always produce facts with bound vector variables if in all rules "*$_\mathbf{H}(_V)$ *:* $-$`tail`*"* `tail` *has a non-metapredicate subgoal that refers to* $_V$*, or in general* $_V$ *can be assigned a binding if all non-metapredicate subgoals in* `tail` *are bound.* □

Intuitively, after we magic-rewrite a program it will keep deriving facts with unbound vectors only if a nonterminal of the initial program derives uninstantianted vectors and in the rules that are used it does not share variables with predicates or nonterminals s that bind their arguments (otherwise, the magic predicate will force the the rules that produce uninstantianted vectors to focus on bindings of s.) For example, specification (MS55) does not derive uninstantianted vectors because the nonterminal `_Cond`, that may derive uninstantianted variables, shares variables with `$Table(_A)`. The following rules formalize the above intuition. Their complexity is due to the fact that a nonterminal may bind the arguments of another nonterminal that initially was not bound.

Definition 8.3.1 (**Grounded Subgoal in a Rule** R) A subgoal that uses a predicate from the target query, is grounded because target query subgoals instantiate their arguments using frozen constants. A nonterminal subgoal is grounded as defined by Definition 8.3.3. 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 8.3.2 (**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 8.3.3 (**Grounded nonterminal**) A nonterminal **_N** is grounded if each rule defining **_N** is grounded.[5] 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 8.3.2 *If each nonterminal in a description* D *is grounded then a bottom-up evaluation of magic-sets rewritten* D *produces no fact that has vector variables.* □

The following algorithm derives CSQs given a target query and a description that has all nonterminals ground.

Algorithm 1
Input: Target query Q and Description D
Output: A set of CSQs $s_i, i = 1, \ldots, n$
Method:
  Check if every nonterminal in D grounded (see definition 8.3.3)
  Reorder each template R in D such that
    All predicate subgoals occur in the front of the rule
    A nonterminal **_N** appears after **_M** if **_N** depends on **_M** for grounding.
    Metapredicates appear at the end of the rule
  Rewrite D using Magic-sets
  Evaluate bottom-up the rewritten description D as described in Appendix D.3

Template R can be reordered because of the following theorem.

Theorem 8.3.3 *Nonterminals of a program can be completely ordered such that nonterminal* **_N** *in position* i *depends for its groundings only on nonterminal in positions* $1 \ldots i-1$*.* □

[5] We could relax this assumption by assuming that there is at least one grounded rule defining **_N**. However, this makes the other rules useless and meaningless.

8.3.2 Retaining Only "Representative" CSQs

A large number of unneeded CSQs are generated by templates that use vectors and the **_subset** metapredicate. For example, template (QT61.1) describes for a particular **_A** all CSQs that have in their head any subset of variables in **_A**. It is not necessary to generate all possible CSQs. Instead, for all CSQs that are derived from the same expansion e, of some template t, where e has the form

`answer(`*_V*`) :-` ⟨*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*.[6] 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 (Q64) and the represented CSQ (Q67) both are derived from the expansion (E71) of template (QT61.1).

(E71) `answer(`*_R*`):-$Table(`*_A*`),`**_in**`($Position,X,`*_A*`),substring(X,'B'),`**_subset**`(`*_R*`,`*_A*`)`

The CSQ discovery module generates only (Q64) and not (Q67) because (Q64) has fewer attributes than (Q67) and is derived by by mapping the vector *_R* to the same vector with *_A*, *i.e.*, to `[F1,L1,D,O1,M1]`. Representative CSQs often retain unneeded attributes and consequently *Representative plans*, *i.e.*, plans containing representative CSQs, retrieve unneeded attributes. The unneeded attributes are projected out by the plan refinement module.

Theorem 8.3.4 *Retaining only representative CSQs does not lose any plan,* i.e.*, if there is an algebraically optimal plan p_s that involves a represented query s then p_s will be discovered by the CBR.* □

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.

Retaining only a representative CSQ of head arity a eliminates $2^a - 1$ represented CSQs thus eliminating an exponential factor from the execution time and from the size of the output of the CSQ discovery module. Still, one might ask why the CSQ discovery phase does not remove the variables that can be projected out. The reason is that the "projection" step is better done after plans are formed because at that time information is available about the other CSQs in the plan and the way they interact (see Section 8.5). Thus, though postponing projection pushes part of the complexity to a later stage, it eliminates some complexity altogether. The eliminated complexity corresponds to those represented CSQs that in the end do not participate in any plan because they retain too few variables.

8.3.3 Pruning Non-Maximal CSQs

Further efficiency can be gained by eliminating any CSQ Q that has fewer subgoals than some other CSQ Q' because Q checks fewer conditions than Q'. A CSQ is maximal if there is no CSQ with more subgoals and the same set of exported variables, modulo variable renaming. We formalize maximality in terms of subsumption [Ull89].

Definition 8.3.4 (Maximal CSQs) A CSQ s_m is a *maximal CSQ* if there is no other CSQ s that is subsumed by s_m. □

[6] 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. We will not describe any further the extensions needed for the handling of this case.

Note, there are two subtle differences between the definition 7.2.5 in Section 7.2.1 and the above definition of maximality:

- Definition 7.2.5 requires that the maximal query subsumes the client query.
- Even more, Definition 7.2.5 requires that there is a filter that can be applied on the maximal query and produce the client query.

In general, the CSQ discovery module generates only *maximal* CSQs and prunes all others. This pruning technique is particularly effective when the CSQs contain a large number of conditions. For example, assume that g conditions are applied to the variables of a predicate. Consequently, there are $2^g - 1$ CSQs where each one of them contains a different proper subset of the conditions. By keeping "maximal CSQs only" we eliminate an exponential factor of 2^g from the output size of the CSQ discovery module.

Theorem 8.3.5 *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.

8.4 Plan Construction

In this section we present the plan construction module (see Figure 8.1.) 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 8.4.1 addresses (i) with the notion of "subgoal consumption." Section 8.4.2 checks (ii), *i.e.*, checks join variables. Section 8.4.3 checks (iii) by ensuring bindings are available. Finally, Section 8.4.4 summarizes the conditions required for constructing a plan and provides an efficient plan construction algorithm.

8.4.1 Set of Consumed Subgoals

We associate witheach CSQ a set of consumed subgoals that describes the CSQs contribution to a plan. Loosely speaking, a CSQconsumes a subgoal if it extracts all the required information from that subgoal. Our goal is to find a combination of CSQs that consume all subgoals of the target query. This task is not so simple because a CSQ does not necessarily consume all its subgoals. For example, consider a CSQ s_e that semijoins the `emp` relation with the `dept` relation to output each `emp` tuple that is in some department in relation `dept`. Even though this CSQ has a subgoal that refers to the `dept` relation it may not always consume the `dept` subgoal. In particular, consider a target query Q that requires the names of all employees and the location of their departments. CSQ s_e does not output the location attribute of table `dept` and thus does not consume the `dept` subgoal with respect to query Q. Before we proceed to a formal definition of the set of consumed subgoals associated with a CSQ we provide an example that illustrates the conditions that must be met in order to include a subgoal in the consumed set of a CSQ.

EXAMPLE 8.4.1 Consider a wrapper, called `semijoin`, that supports queries that may include any set of predicates, with arbitrary join conditions between them (unlike the wrapper used in Example 65,) but can export data from only one of the predicates. This is the case in some object-oriented databases, such as ObjectStore, which export information about 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 (Q72) and (Q73), which allow us to compute (Q62) using plan (P74).[7] ((Q72) and (Q73) are representative CSQs.)

[7] `semijoin` does not directly support the query (Q62) because (Q62) exports variables from both its `emp` subgoals.

```
(Q72) answer72(F0,L0,D,O0) :- emp(F0,L0,D,O0,'Y'), emp(F1,L1,D,O1,M1),
                              substring(O1,'B')
(Q73) answer73(F1,L1,D,O1,M1) :- emp(F0,L0,D,O0,'Y'), emp(F1,L1,D,O1,M1),
                                 substring(O1,'B')
(P74) answer(F0,L0,O1) :-  answer72(F0,L0,D,O0), answer73(F1,L1,D,O1,M1)
```

In the above example (Q72) consumes only the subgoal `emp(F0,L0,D,O0,'Y')` while (Q73) consumes the subgoals `emp(F1,L1,D,O1,M1)` and `substring(O1,'B')`. Intuitively, (Q72) 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 8.4.1 (**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 8.4.1 *Each CSQ has a unique* maximal *set of consumed subgoals that is a superset of every other set of consumed subgoals.* □

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

Algorithm 2
Input: A target query Q and a CSQ s for 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 8.4.2 (**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 8.4.2) produces each plan in time polynomial in the number of CSQs produced by the CSQ discovery module. However, there are scenarios (see the extended version [PGH]) where the relevance criteria may erroneously prune out a CSQ that could be part of a plan.

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

In summary, the computation of the set of consumed subgoals is an important step of plan construction because in order for a set of CSQs to form a plan it is a necessary condition that the union of their sets of consumed subgoals includes every subgoal of the target query. However, this condition is not sufficient. The join variables condition must also be satisfied, as we discuss in the next section.

8.4.2 Join Variables Condition

Finding sets of CSQs that consume every subgoal of the target query does not complete the task of the plan construction module. 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 joining of the CSQs forms a plan. In this section we describe a condition, which is tested by the plan construction module, and ensures that a set of CSQs can be joined to form a plan.

In order to explain the condition we first present an example where the condition does not hold and consequently we cannot form a plan. Consider an online employee database that can be queries for the names of all employees in a given division. The database can also be queried for the names of all employees in a given location. Further, the name of an employee is not uniquely determined by their location and division. The employee database cannot be used to find employees in a given division and in a given location by joining the results of two queries - one on division and the other on location. To see this, consider a query that looks for employees in "CS" in "New York". Joining the results of two independent queries on division and location will incorectly return as answer a person named "John Smith" if there is a "John Smith" in "CS" in "San Jose" and a different "John Smith" in "Electrical" in "New York".

Intuitively, the problem arises because the two independent queries do not export the information necessary to correctly join their results. We can avoid this problem by checking that CSQs are combined only if they export the join variables necessary for their correct combination. The theorem of Section 8.4.4 formally describes the conditions on join variables that guarantee the correct combination of CSQs.

8.4.3 Passing Required Bindings via Nested Loops Joins

Computing the sets of consumed subgoals and testing for the join variables condition is sufficient for formulating plans that combine queries using joins, selections, and projections. However, in heterogeneous systems we can combine CSQs by passing join variable bindings from one CSQ to the other. In effect, the CBR emulates nested loops joins in this way. We first illustrate nested loops joins using an example and consequently we discuss the additional requirements posed on CBR for the discovery of plans utilizing nested loops.

We may compute (Q62) by the following steps: first we execute (Q75); then we collect the department names (*i.e.*, the `D` bindings) and for each binding d of `D`, we replace the `$D` in (Q76) with d and send the instantianted query to the wrapper. We use the notation `/$D` in the nested loops plan (P77) to denote that (Q76) receives values for the `$D` placeholder from `D` *bindings* of the other CSQs – (Q75) in this example.

(Q75) `answer`$_{75}$`(F0,L0,D,O0) :- emp(F0,L0,D,O0,'Y')`
(Q76) `answer`$_{76}$`(F1,L1,O1,M1) :- emp(F1,L1,$D,O1,M1)`
(P77) `answer(F0,L0,O1) :- answer`$_{75}$`(F0,L0,D,O0), answer`$_{76}$`(F1,L1,O1,M1)/$D`

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

- **CSQ discovery:** A subgoal of a CSQ s may contain placeholders `/$`$\langle var \rangle$, such as `$D`, in place of corresponding join variables (`D` in our example.) Whenever this is the case, we introduce the structure `/$`$\langle var \rangle$ next to the `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, $\mathcal{B}_{76}$ = {`D`}

and $\mathcal{B}_{75} = \{\}$. CSQ discovery previously did not use bindings information while deriving facts. Thus, the algorithm derives useless CSQs that need bindings not exported by any other CSQ.

The optimized derivation process uses two sets of attributes and proceeds iteratively. Each iteration derives only those facts that use bindings provided by existing facts. In addition, a fact is derived if it uses at least one binding that was made available only in the very last iteration. Thus, the first iteration derives facts that need no bindings, that is, for which $\mathcal{B}_s$ is empty. The next iteration derives facts that use at least one binding provided by facts derived in iteration one. Thus, the second iteration does not derive any subgoal derived in the first iteration, and so on. Algorithm 5 in Appendix D.3 formalizes this intuition.

The bindings needed by each CSQ of a plan impose order constraints on the plan. For example, the existence of `D` in $\mathcal{B}_{76}$ requires that a CSQ that exports `D` is executed before (Q76). It is the responsibility of the plan construction module to ensure that the produced plans satisfy the order constraints.

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

8.4.4 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 8.4.2 *Given CSQs $s_i, i = 1, \ldots, n$ with corresponding heads* $\texttt{answer}_i(V_1^i, \ldots, V_{v_i}^i)$*, sets of maximal consumed subgoals $\mathcal{C}_i$ and sets of needed bindings $\mathcal{B}_i$, the plan*

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

is correct if

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

Algorithm 3 in Appendix D.1 for plan construction is based on Theorem 8.4.2. The algorithm takes as input a set of CSQs derived by the CSQ discovery process described later, and the target query Q. At each step the algorithm selects a CSQ s that consumes at least one subgoal that has not been consumed by any CSQ s' considered so far and for which all variables of $\mathcal{B}_s$ have been exported by at least one s'. Assuming that the algorithm is given m CSQs (by the CSQ discovery module) it can construct a set that satisfies the consumed sets and the bindings passing conditions in time polynomial in m. Nevertheless, if the join variables condition does not hold the algorithm takes time exponential in m because we may have to create exponentially many sets until we find one that satisfies the join variables condition. However, the join variables condition evaluates to true for most wrappers we find in practice (see following discussion) and thus we usually construct a plan in time polynomial in m.

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

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

The plan construction algorithm first sorts the input CSQs in a total order that respects the PO $\overset{b}{\prec}$. Then it procedes by picking CSQs and testing the conditions of Theorem 8.4.2 until it consumes all subgoals of the target query. The algorithm capitalizes on the assumption that in most practical cases every CSQ consumes at least one subgoal and the join variables condition holds. In this case, one plan is developed in time polynomial in the number of input CSQs. The following lemma describes an important case where the join variables condition always holds.

Lemma 8.4.1 *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 8.4.1 holds for typical wrappers of bibliographic information systems and lookup services (wrappers that have the structure of (D61)), 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 8.4.1 cover essentially any wrapper of practical importance.

8.5 Plan Refinement

The plan refinement module filters and refines constructed plans in two ways.

- Eliminate algebraically non-optimal plans.
- Project out unnecessary variables from representative CSQs.

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

Projecting out unnecessary variables First we define the notion 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, the definition is not restricted to maximal consumed sets:)

Definition 8.5.1 (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. □

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

Unecessary variables cannot always be projected out when the maximal consumed sets of the CSQs intersect. For example, consider a wrapper that exports predicates `emp` and `substring`. Every supported query has exactly one `emp` subgoal, at most one `substring` subgoal, and may export any subset of the `emp` variables. The target query (Q78) can be computed by plan (P81).

(Q78) `answer(F,L) :- emp(F,L,D,O,M), substring(D,'data'), substring(O,'B')`
(Q79) $\texttt{answer}_{79}$`(F,L,D,O,M) :- emp(F,L,D,O,M), substring(D,'data')`
(Q80) $\texttt{answer}_{80}$`(F,L,D,O,M) :- emp(F,L,D,O,M), substring(O,'B')`
(P81) `answer(F,L) :-` $\texttt{answer}_{79}$`(F,L,D,O,M),` $\texttt{answer}_{80}$`(F,L,D,O,M)`

Having both queries export all the variables is useless. An obvious optimization is to replace (Q80) with (Q82), which exports only the distinguished variables `F` and `L` and the join variable `D`.

(Q82) $\texttt{answer}_{82}$`(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 (Q79) with CSQ (Q83) we construct the erroneous plan (P84). (P84) violates the join variables condition.

(Q83) $\texttt{answer}_{83}$`(F,L,O) :- emp(F,L,D,O,M), substring(D,'data')`
(P84) `answer(F,L) :-` $\texttt{answer}_{82}$`(F,L,D),` $\texttt{answer}_{83}$`(F,L,O)`

The problem arises because the maximal consumed sets of (Q79) and (Q80) intersect. It can be solved as follows: Since CSQ (Q82) consumes the subgoals `emp(F,L,D,O,M)` and `substring(O,'B')` we can modify the exported variables of the representative CSQ (Q79) so that it consumes only the subgoal `substring(D,'data')`. Thus, we can replace the representative CSQ (Q79) with the CSQ (Q85) that exports only the necessary variables of the set {`substring(D,'data')`}, *i.e.*, `D`. Consequently, we can construct the plan (P86).

(Q85) $\texttt{answer}_{85}$`(D) :- emp(F,L,D,O,M), substring(D,'data')`
(P86) `answer(F,L) :-` $\texttt{answer}_{82}$`(F,L,D),` $\texttt{answer}_{85}$`(D)`

Symmetrically, we may assume that (Q79) consumes `emp(F,L,D,O,M)` and `substring(D,'data')` in which case (Q80) consumes only `substring(O, 'B')` and hence we can produce the plan (P88).

(Q87) $\texttt{answer}_{87}$`(O) :- emp(F,L,D,O,M), substring(O,'B')`
(P88) `answer(F,L) :-` $\texttt{answer}_{83}$`(F,L,O),` $\texttt{answer}_{87}$`(O)`

Intuitively, the plans (P86) and (P88) 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 outline an algorithm to add projections to one representative CSQ s that participates in a plan P for query Q (the algorithm is formally stated in Appendix D.2). The algorithm produces possibly multiple plans with s replaced by a CSQ with fewer distinguished attributes. The algorithm adds projections to s by pruning the set of maximal consumed subgoals of s as computed by Algorithm 2 and then checking if with the pruned set s still forms a plan to compute query Q. A smaller consumed set for a CSQ results in fewer necessary variables and thus fewer distinguished attributes. Thus, if a smaller

Technique	Exponential factor eliminated
Pruning represented CSQs	2^a, where a is the arity of the representative CSQ (see **Evaluation** paragraph of Section 8.3.2)
Pruning non-maximal CSQs	2^g, where g is the number of "optional" subgoals in the maximal CSQ (see Section 8.3.3)
Pruning CSQs with inappropriate bindings	2^n, where n is the number of variables that may be or may not be bound
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 8.3.1)

Figure 8.2: Sources of Exponentiality Eliminated by CBR

consumed set for s is found to be sufficient to build a plan P, then the set of distinguished attributes of s can be reduced thereby adding projections. The algorithm can be iteratively applied to each representative CSQ in a plan.

Algorithm 4 is exponential in the size of the consumed subgoal set $\mathcal{C}_s$. 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 has to be present in all legal consumed subsets of s. 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.

8.6 Completeness and Performance of CBR

The CBR algorithm employs many techniques to eliminate sources of exponentiality that would otherwise arise in many practical cases. Figure 8.2 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 8.4.4. Of course, the complexity of the full CBR remains exponential because the output of the CSQ discovery phase is in general exponential in the size of the query and the description.

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 8.4.2 and the discussion of Section 8.4.1.) On the other hand, the techniques for eliminating exponentiality described in Table 8.2 preserve completeness, in that we do not miss any plan through applying one of these techniques (see justifications in Sections 8.3.2, and 8.3.3).

8.7 Related Work

Significant results have been developed for the resolution of semantic and schematic discrepancies while integrating heterogeneous information sources. However, most of these systems [S$^+$, HM93, A$^+$91, Gup89] do not address the problem of different and limited query capabilities in the underlying sources because they assume that those sources are full-fledged databases that can answer any query over their schema.[8] The

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

recent interest in the integration of arbitrary information sources, including databases, file systems, the Web, and many legacy systems, invalidates the assumption that all underlying sources can answer any query over the data they export and forces us to resolve the mismatch between the query capabilities provided by these sources. Only a few systems have addressed this problem.

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

DISCO [TRV95] describes the set of supported queries using context-free grammars. This technique reduces the efficiency of capabilities-based rewriting because it treats queries as "strings."

The Information Manifold [LRO96] develops a query capabilities description that is attached to the schema exported by the wrapper. The description states which and how many conditions may be applied on each attribute. RQDL provides greater expressive power by being able to express schema-independent descriptions and descriptions such as "exactly one condition is allowed." Nevertheless, there is an interesting similarity between the CBR algorithms and the information finding algorithms of the Information Manifold: The former find possible ways to execute a client query, i.e., they find combinations of supported queries that, when combined, they are equivalent to the client query. Information Manifold finds combinations of source supported queries which, when combined, are subsumed by the client query and hence they are a possible solution to the problem.

TSIMMIS suggests an explicit description of the wrapper's query capabilities [PGGMU95], using the context-free grammar approach of the current paper. (The description is also used for query translation from the common query language to the language of the underlying source.) However, TSIMMIS considers a restricted form of the problem wherein descriptions consider relations of prespecified arities and the mediator can only select or project the results of a single CSQ.

This paper enhances the query capability description language of [PGGMU95] to describe queries over arbitrary schemas, namely, relations with unspecified arities and names, as well as capabilities such as "selections on the first attribute of any relation." The language also allows specification of required bindings, *e.g.*, a bibliography database that returns "titles of books given author names." We provide algorithms for identifying for a target query Q the algebraically optimal CSQs from the given descriptions. Also, we provide algorithms for generating plans for Q by combining the results of these CSQs using selections, projections, and joins.

The CBR problem is related to the problem of determining how to answer a query using a set of materialized views [LY85, LMSS95, RSU95, Qia96]. 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 a particular evaluation order due to binding requirements.

[LMSS95] shows that rewriting a conjunctive query is in general exponential in the total size of the query and views. [Qia96] shows that if the query is acyclic we can rewrite it 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). Recently [LRU96] proposed Datalog for the description of supported queries. It also suggested an algorithm that essentially finds what we call maximal CSQs.

Appendix A

MSL Syntax

A complete syntax for the MSL language appears in Figure A.1. Figure A.2 provides the syntax of normal form MSL. Note, that the definitions of $\langle object \rangle$, $\langle object\ condition \rangle$, and $\langle value\ condition \rangle$ are much simpler in the normal form and $\langle rest \rangle$ has been eliminated.

```
(0)  ⟨specification⟩        ::=  Rules ⟨rules⟩ [External ⟨external pred defs⟩]
(1)  ⟨rules⟩                ::=  ⟨rules⟩ ⟨rule⟩
                              |  ϵ
(2)  ⟨rule⟩                 ::=  ⟨head⟩ :− ⟨condition list⟩ .
(3)  ⟨head⟩                 ::=  ⟨object⟩
                              |  ⟨predicate name⟩ ( ⟨argument list⟩ )
(4)  ⟨object⟩               ::=  ⟨variable⟩
                              |  < [⟨object id⟩] ⟨label⟩ ⟨value⟩ >
(5)  ⟨object id⟩            ::=  ⟨term⟩
(6)  ⟨term⟩                 ::=  ⟨function symbol⟩ ( ⟨term list⟩ )
                              |  ⟨simple term⟩
(7)  ⟨term list⟩            ::=  ⟨term list⟩ , ⟨term⟩
                              |  ⟨term⟩
(8)  ⟨simple term⟩          ::=  ⟨variable⟩
                              |  ⟨constant⟩
(9)  ⟨label⟩                ::=  ⟨simple term⟩
(10) ⟨value⟩                ::=  ⟨simple term⟩
                              |  { ⟨subobjects list⟩ }
(11) ⟨subobjects list⟩      ::=  ⟨subobjects list⟩ ⟨object⟩
                              |  ϵ
(12) ⟨argument list⟩        ::=  ⟨argument list⟩ , ⟨term⟩
                              |  ⟨term⟩
(13) ⟨condition list⟩       ::=  ⟨condition list⟩ AND ⟨condition⟩
                              |  ⟨condition⟩
                              |  TRUE
(14) ⟨condition⟩            ::=  ⟨object condition⟩ @ ⟨site⟩
                              |  ⟨predicate name⟩ ( ⟨argument list⟩ )
                              |  ⟨predicate name⟩ ( ⟨argument list⟩ ) @ ⟨site⟩
(15) ⟨object condition⟩     ::=  [⟨object variable⟩ :]
                                   < [⟨object id⟩] ⟨label⟩ ⟨value condition⟩ >
(16) ⟨value condition⟩      ::=  ⟨simple term⟩
                              |  [⟨set variable⟩ :] { ⟨obj cond list⟩ [| ⟨rest⟩] }
(17) ⟨rest⟩                 ::=  ⟨set variable⟩ [: { ⟨obj cond list⟩ [| ⟨rest⟩] }]
(18) ⟨obj cond list⟩        ::=  ⟨object cond list⟩ ⟨object condition⟩
                              |  ϵ
(19) ⟨site⟩                 ::=  ⟨simple term⟩
(20) ⟨external pred defs⟩   ::=  ⟨external pred defs⟩ ⟨external pred def⟩
                              |  ϵ
(21) ⟨external pred def⟩    ::=  ⟨pred name⟩ ( ⟨type list⟩ ) ( ⟨binding list⟩ )
                                   impl by ⟨C function name⟩
(22) ⟨type list⟩            ::=  ⟨type list⟩ , ⟨type⟩
                              |  ⟨type⟩
(23) ⟨binding list⟩         ::=  ⟨binding list⟩ , ⟨binding⟩
                              |  ⟨binding⟩
(24) ⟨binding⟩              ::=  bound|free
```

Figure A.1: MSL's Syntax

	⟨*specification*⟩	::=	`Rules` ⟨*rules*⟩ [`External` ⟨*external pred defs*⟩]
	⟨*rules*⟩	::=	⟨*rules*⟩ ⟨*rule*⟩
		|	ϵ
	⟨*rule*⟩	::=	⟨*head*⟩ :− ⟨*condition list*⟩ .
	⟨*head*⟩	::=	⟨*object*⟩
		|	⟨*predicate name*⟩ (⟨*argument list*⟩)
(25)	⟨*object*⟩	::=	< ⟨*object id*⟩ ⟨*label*⟩ ⟨*value*⟩ >
		|	∗ ⟨*variable*⟩
	⟨*object id*⟩	::=	⟨*term*⟩
	⟨*term*⟩	::=	⟨*function symbol*⟩ (⟨*term list*⟩)
		|	⟨*simple term*⟩
	⟨*term list*⟩	::=	⟨*term list*⟩ , ⟨*term*⟩
		|	⟨*term*⟩
	⟨*simple term*⟩	::=	⟨*variable*⟩
		|	⟨*constant*⟩
	⟨*label*⟩	::=	⟨*simple term*⟩
	⟨*value*⟩	::=	⟨*simple term*⟩
		|	{ ⟨*subobjects list*⟩ }
	⟨*subobjects list*⟩	::=	⟨*subobjects list*⟩ ⟨*object*⟩
		|	ϵ
	⟨*argument list*⟩	::=	⟨*argument list*⟩ , ⟨*term*⟩
		|	⟨*term*⟩
	⟨*condition list*⟩	::=	⟨*condition list*⟩ `AND` ⟨*condition*⟩
		|	⟨*condition*⟩
		|	`TRUE`
	⟨*condition*⟩	::=	⟨*object condition*⟩ @ ⟨*site*⟩
		|	⟨*predicate name*⟩ (⟨*argument list*⟩)
		|	⟨*predicate name*⟩ (⟨*argument list*⟩) @ ⟨*site*⟩
(26)	⟨*object condition*⟩	::=	< ⟨*object id*⟩ ⟨*label*⟩ ⟨*value condition*⟩ >
(27)	⟨*value condition*⟩	::=	⟨*simple term*⟩
		|	{ ⟨*object cond list*⟩ }
	⟨*obj cond list*⟩	::=	⟨*object cond list*⟩ ⟨*object condition*⟩
		|	ϵ
	⟨*site*⟩	::=	⟨*simple term*⟩
	⟨*external pred defs*⟩	::=	⟨*external pred defs*⟩ ⟨*external pred def*⟩
		|	ϵ
	⟨*external pred def*⟩	::=	⟨*pred name*⟩ (⟨*type list*⟩) (⟨*binding list*⟩) `impl by` ⟨*C function name*⟩
	⟨*type list*⟩	::=	⟨*type list*⟩ , ⟨*type*⟩
		|	⟨*type*⟩
	⟨*binding list*⟩	::=	⟨*binding list*⟩ , ⟨*binding*⟩
		|	⟨*binding*⟩
	⟨*binding*⟩	::=	`bound`|`free`

Figure A.2: Syntax of Normal Form MSL

Appendix B

MSL Semantics

B.0.1 Safety and Typing Restrictions

MSL enforces the safety restriction that all variables appearing in the head of an MSL rule must also appear in its tail. MSL also enforces the following typing restrictions on the use of variables. The expression of the typing restrictions is based on the variable's categorization that was described in Section 5.5.1:

1. No object, rest, or value variable that binds to sets only can also be an atomic variable. For example, the following MSL rule is prohibited because the variable `O` appears as an object variable (in the head) and as an atomic variable (in the tail).

```
O :- <O V>@src
```

2. No object variable, rest variable, or value variable that binds to sets only may appear two times in the MSL rule tail. For example, the following rule is prohibited

```
O:- <l1 V:{<l2 v1>}>@src AND <l3 V>@src
```

   Whenever comparison of non-atomic variables is required appropriate set comparison (or object comparison) predicates should be used.

B.0.2 Reduction of Normal-Form MSL Rules to Datalog Rules

In this section we describe the reduction of an MSL specification that consists of normal-form MSL rules to a Datalog program (that may contain function symbols). Every normal-form MSL rule $R_i^N, 1 \leq i \leq r$, where r is the number of normal-form MSL rules, reduces to a set of datalog rules R_i^D. The concatenation of all the Datalog rules of sets R_i^D, together with three generic rules, that are specified in Subsection B.0.2, constitute the Datalog program that specifies the semantics of the MSL specification. Note, from now on whenever we write "MSL rule" we will imply "normal-form MSL rule".

For every MSL rule R_i^N of the mediator specification we generate a set R_i^D of datalog rules. R_i^D contains:

1. one rule of the form

$$bind_i(\bar{W}) :- \langle datalog\ condition \rangle$$

   where $\bar{W}$ is the set of variables that appear in the MSL rule head. The $\langle datalog\ condition \rangle$ is derived by reducing the MSL rule's tail to a conjunction of datalog literals, following the steps described in Subsection B.0.2. Note, whenever there is no confusion about which rule we are talking about we will refer to $bind_i$ simply as $bind$.

2. l rules of the form

$$head_j(\bar{Y}) :- bind(\bar{W})$$

where the $head_j$ datalog rule heads are derived by reducing the MSL rule's head to datalog, following the steps described in Subsection B.0.2.

Reduction of the MSL Rule's Tail

The MSL rule's tail is translated into a conjunction of Datalog literals, i.e., a conjunction of one or more *object*, *member*, *top*, external predicates, and source specific predicates, that specify which are the acceptable bindings for the set $\bar{W}$ of interesting variables. The following rules formalize the reduction of the MSL rule tail into a conjunction of literals, by associating reduction actions with the productions of the MSL syntax (Figure A.1):

- $\langle$*condition list*$\rangle$: Every condition c_i , $1 \leq i \leq m$ of the condition list of an MSL rule tail results in a conjunction C_i of one or more datalog literals. The tail of the datalog rule is the conjunction of all the literals that appear in all C_i , $1 \leq i \leq m$.
- $\langle$*condition*$\rangle$: An MSL condition c results in a conjunction C of one or more datalog literals, depending on whether the condition is an object condition or a predicate:
  1. if $\langle$*condition*$\rangle$ is a *"top object condition"* (see Figure A.1), i.e. has the form

     <$\langle oid \rangle$...>@$\langle src \rangle$

     the condition reduces to a conjunction C of the literals resulting from the object condition <$\langle oid \rangle$...> (see action associated with $\langle$*object condition*$\rangle$) and the literal

     $$top(\langle src \rangle, \langle oid \rangle)$$

  2. if $\langle$*condition*$\rangle$ is an *"external predicate condition"* (see Figure A.1) that has the form

     pred($\langle arg \rangle_1$, $\langle arg \rangle_2$, ..., $\langle arg \rangle_n$)

     where $\langle arg \rangle_i$, $1 \leq i \leq n$ are the arguments of the predicate, the predicate condition results in the datalog literal (that can be seen as a trivial conjunction)

     $$\texttt{ext_pred}(\langle arg \rangle_1, \langle arg \rangle_2, \ldots, \langle arg \rangle_n)$$

     We changed the name of the predicate from `pred` to `ext_pred` to avoid confusing it with some source predicate with the same name.
  3. if $\langle$*condition*$\rangle$ is a *"source predicate condition"* that has the form

     pred($\langle arg \rangle_1$, $\langle arg \rangle_2$, ..., $\langle arg \rangle_n$)@$\langle src \rangle$

     where $\langle arg \rangle_i$, $1 \leq i \leq n$ are the arguments of the predicate and $\langle src \rangle$ is a simple term that stands for the source, the source predicate reduces to the datalog literal (that can be seen as a trivial conjunction):

     $$\texttt{pred}(\langle src \rangle, \langle arg \rangle_1, \langle arg \rangle_2, \ldots, \langle arg \rangle_n)$$
- $\langle$*object condition*$\rangle$: The reduction of the object condition to a literal conjunction depends on whether the value of the object condition binds to atomic constants only, sets only, or anything:
  1. if the object condition has the form

     <$\langle oid \rangle$ $\langle label \rangle$ $\langle constant \rangle$>

     it reduces to the datalog literal

     $$object(\langle src \rangle, \langle oid \rangle, \langle label \rangle, \langle constant \rangle)$$

     where $\langle src \rangle$ is the term that describes the source against which the object condition is evaluated.

2. if the object condition has the form

   `isatom:`<$\langle oid \rangle$ $\langle label \rangle$ $\langle variable \rangle$>

   it reduces to the conjunction of datalog literals

$$object(\langle src \rangle, \langle oid \rangle, \langle label \rangle, \langle variable \rangle) \wedge ext_neqset(\langle variable \rangle)$$

   where $\langle src \rangle$ is the term that describes the source against which the object condition is evaluated and *ext_neqset* is a predicate that is satisfied when its argument does not have the value *set*.

3. if the object condition has the form

   <$\langle oid \rangle$ $\langle label \rangle$ {<$\langle oid_1 \rangle \ldots$> <$\langle oid_2 \rangle \ldots$> $\ldots$ <$\langle oid_n \rangle \ldots$>}>

   then we introduce in the datalog condition

   (a) the literal

$$object(\langle src \rangle, \langle oid \rangle, \langle label \rangle, set)$$

   (b) the literals that describe that the object identified by $\langle oid \rangle$ has subobjects that are identified by $\langle oid \rangle_i$, $1 \leq i \leq n$

$$\begin{array}{l} member(\langle src \rangle, \langle oid \rangle, \langle oid \rangle_1) \\ member(\langle src \rangle, \langle oid \rangle, \langle oid \rangle_2) \\ \vdots \\ member(\langle src \rangle, \langle oid \rangle, \langle oid \rangle_n) \end{array}$$

   (c) the literals that result from the reduction of the subobject patterns <$\langle oid_1 \rangle \ldots$> $\ldots$ <$\langle oid_n \rangle \ldots$> to conjunctions of datalog literals.

Reduction of MSL Rule's Head

We reduce every MSL rule's head to a set of datalog rules heads and we attach to each datalog rule head the tail $bind(\bar{W})$ thus getting a set of datalog rules that correspond to the MSL rule.

- if the head is an "exported predicate" (see Figure A.1) that has the form

$$\texttt{pred}(\langle arg_1 \rangle, \ldots, \langle arg_n \rangle)$$

  where $\langle arg_1 \rangle, \ldots, \langle arg_n \rangle$ are the arguments, we create the datalog rule

$$\texttt{pred}(\langle medname \rangle, \langle arg \rangle_1, \ldots, \langle arg_2 \rangle) : -bind(\bar{W})$$

  where $\langle medname \rangle$ is the name of the mediator we specify.

- if the head has the form

$$\$\langle oid \rangle$$

  then we introduce the datalog rules

$$\begin{array}{l} top(\langle oid \rangle) : -bind(\bar{W}) \\ pick(\langle medname \rangle, \langle src \rangle, \langle oid \rangle) : -bind(\bar{W}) \end{array}$$

  where $\langle src \rangle$ is a simple term (variable or constant) that describes the source of the objects identified by $\langle oid \rangle$ ($\langle src \rangle$ can be trivially found by inspecting the MSL rule's tail). The predicate *pick* fetches the object identified by $\langle oid \rangle$ together with all its subobjects.

- if the head is a "generated object" that has the form

  <$\langle oid \rangle \langle label \rangle \langle type \rangle$ $\langle generated\ value \rangle$>

and results in one or more datalog rules depending on the form and properties of the $\langle$*generated value*$\rangle$:

- if $\langle$*generated value*$\rangle$ is a constant `c` or a variable `V` (that, by now, we can be sure that it binds to atomic constants only) then we introduce the datalog rule

$$object(\langle medname\rangle, \langle oid\rangle, \langle label\rangle, \langle generated\ value\rangle) :- bind(\bar{W})$$

- if $\langle$*generated value*$\rangle$ is a subobjects list that has the form

$$\{<\langle oid_1\rangle \ldots> \ \ldots \ <\langle oid_n\rangle \ldots> \ \$\langle refoid_1\rangle \ \ldots \ \$\langle refoid_m\rangle\}$$

[1] we introduce:

1. the datalog rule

$$object(\langle medname\rangle, \langle oid\rangle, \langle label\rangle, set, set)$$

2. the datalog rules that specify that the objects identified by

$$\langle oid_1\rangle, \ldots, \langle oid_n\rangle, \langle refoid_1\rangle, \ldots, \langle refoid_m\rangle$$

are subobjects of the object identified by $\langle oid\rangle$

$$member(\langle medname\rangle, \langle oid\rangle, \langle oid_i\rangle) :- bind(\bar{W})$$
$$member(\langle medname\rangle, \langle oid\rangle, \langle refoid_j\rangle) :- bind(\bar{W})$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$.

3. the datalog rules that result from the reduction of the head object patterns

$$<\langle oid_1\rangle \ldots> \ldots <\langle oid_n\rangle \ldots>$$

to datalog conditions.

4. the datalog rules that "copy" the objects identified by $\langle refoid_1\rangle, \ldots, \langle refoid_m\rangle$ from $\langle src\rangle$ to $\langle medname\rangle$

$$pick(\langle medname\rangle, \langle src\rangle, \langle refoid_j\rangle) :- bind(\bar{W})$$

where $1 \leq j \leq m$.

Copying Objects from Source to Destination

The following three generic rules are attached to every datalog program and are used for copying an object together with its subobjects from a source S to a mediator M:

$$object(D, O, L, V) :- pick(D, S, O) \wedge object(S, O, L, V)$$
$$member(D, O_1, O_2) :- pick(D, S, O_1) \wedge member(S, O_1, O_2)$$
$$pick(D, S, O_2) :- pick(D, S, O_1) \wedge member(S, O_1, O_2)$$

B.0.3 Evaluable External Predicates

Rules that contain external predicates may not be evaluable. For example, consider the external predicate **decompose_name** that was described in Section 4.1. We may not have a rule such as

```
<name {<last_name LN> <first_name FN>}> :- decompose_name(N, LN, FN)
```

because the predicate **decompose_name** can not produce bindings for `LN` and `LN` without first being given a binding for `N`.

We formalize the notion of evaluable rules that contain external predicates, as follows: A rule is evaluable if there is a permutation of the conditions of the rule tail, such that if the external predicate p appears as the i-th condition of the tail, then there is an implementation of p such that in the place of bound arguments appear terms that contain constants and variables that appear in the conditions that precede p.

[1] Without loss of the generality we assume that the $ patterns follow the <...> patterns.

Appendix C

Extended Algorithm QinP

This appendix provides the formal description of the algorithm X-QinP that finds the maximal supporting queries of a given query q, given a QDTL description d. We recall that X-QinP is based on the Algorithm QinP (Section 14.5 of [Ull89]) that 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. Note, we assume q has been translated to a conjunctive query (using the OEM-to-relational reduction presented in Appendix B) and d has been translated to a Datalog program as illustrated in Section 7.5.1.

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

   (Q89) `answer(O) :- top(O), object(O,L,V)`

   that supports the query

   (Q90) `answer(O) :- top(O), object(O,person,V)`

   Note, we have to filter the result of Q89 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 Q90 in the residue, though it maps to the `object` subgoal of Q89.
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. We also provide the following useful definition

Definition C.0.1 (**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 $I_{t'} \subset I_t$. □

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 θ 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 θ 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

Appendix D

CBR Appendix

(28) $\langle description \rangle$	::=	$(\langle query\ template \rangle \mid \langle nonterminal\ template \rangle)*$
(29) $\langle query\ template \rangle$	::=	`answer`($\langle predicate\ arguments \rangle$) $:-$ $\langle subgoal\ list \rangle$
(30) $\langle nonterminal\ template \rangle$	::=	$\langle nonterminal\ name \rangle$ ($\langle arguments \rangle$) $\langle subgoal\ list \rangle$
(31) $\langle subgoal\ list \rangle$	::=	$\langle subgoal \rangle$ (, $\langle subgoal \rangle)*$
(32) $\langle subgoal\ list \rangle$	::=	ϵ %*subgoal list may be emtpy*
(33) $\langle subgoal \rangle$	::=	$\langle predicate \rangle$ ($\langle arguments \rangle$) %*predicate*
(34) $\langle subgoal \rangle$	::=	$\langle metapredicate\ name \rangle$ ($\langle arguments \rangle$) %*metapredicate*
(35) $\langle subgoal \rangle$	::=	$\langle nonterminal\ name \rangle$ ($\langle arguments \rangle$) %*nonterminal*
(36) $\langle arguments \rangle$	::=	$\langle vector \rangle \mid \langle variable \rangle$ (, $\langle variable \rangle)*$
(37) $\langle predicate\ name \rangle$	::=	$\langle identifier \rangle \mid \langle placeholder \rangle$
(38) $\langle metapredicate\ name \rangle$	::=	$\langle identifier \rangle$
(39) $\langle nonterminal\ name \rangle$	::=	$\langle identifier \rangle$

Figure D.1: Normal-form RQDL syntax

D.1 Algorithm for Plan Construction

Algorithm 3

Input: A set of CSQs $\{s_1, \ldots, s_m\}$
    A target query Q
Output: A set of plans that satisfy Theorem 8.4.2 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* $\overset{b}{\prec}$
    Invoke procedure *plan*(L_0,$\{\}$)

Procedure *plan*(L, P)
% P is list of CSQs that form part of a plan (the first CSQs of the plan's tail)
% L is a sorted list of CSQs that are considered for generating P
% $sub(P)$ *refers to the union of the consumed sets* $\mathcal{C}_i$ *of the CSQs* s_i *of the set P*
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 8.4.2
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*

D.2 Algorithm for Plan Refinement

Algorithm 4
Input: Plan P involving representative CSQ s.
Output: One or more plans with s replaced by a CSQ with fewer distinguished attributes
Method:
% Prune the set of maximal consumed subgoals of s
For each subset M of the set of maximal consumed subgoals of s
Replace annotation $\mathcal{C}_s$ by M
% Check that the resulting plan is legal
% sub(P) refers to the union of the maximal consumed sets of plan P
If $sub(P)$ contains all subgoals of Q then proceed else discard M
% consumes all subgoals
Compute set of necessary variables V of s as per Definition 8.5.1.
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 8.2.1) than P''
discard P' and exit loop
else if P'' is algebraically worse than P'
discard P''

D.3 Bottom-up Algorithm for deriving CSQs

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

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)

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

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 fact n'
do not add n to set of derived facts.

D.4 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 8.1.4.)

The syntax appears in Figure D.1. Furthermore, we restrict to descriptions where there is a nonterminal template, with matching arity, for every nonterminal that appears in a template. Additionally, for the implementation reasons described in Section 8.3 we restrict to descriptions where all nonterminals are grounded (see Definition 8.3.3).

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 D.4.1 (**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

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

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

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

and a nonterminal template of the form

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

where

(a) the nonterminal template and the expansion e have no common variable,
(b) there is a collection of mappings θ such that $\theta(N(\langle arguments \rangle))$ is identical to $\theta(N(\langle definition\ arguments \rangle))$. We call θ a *unifier*. Definition D.4.2 formally defines the application of a unifier on an RQDL expression.

□

Definition D.4.2 (**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 $\langle placeholder \rangle \mapsto \langle constant \rangle$, or $\langle variable \rangle_1 \mapsto \langle variable \rangle_2$, or $\langle vector \rangle_1 \mapsto \langle vector \rangle_2$ then replace all instances of $\langle placeholder \rangle$, $\langle variable \rangle_1$, and $\langle vector \rangle_2$ with $\langle constant \rangle$, $\langle variable \rangle_2$, or $\langle vector \rangle_2$ respectively.

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

□

Definition D.4.3 (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 D.4.4 (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.

Bibliography

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

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference*, pages 238–47, Denver, CO, May 1991.

[ACHK93] Y. Arens, C.Y. Chee, C.-N. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *Intl Journal of Intelligent and Cooperative Informations Systems*, 2:127–58, June 1993.

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. VLDB*, pages 73–84, 1993.

[ACPS96] S. Adali, S. C. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. SIGMOD*, pages 137–48, 1996.

[AGMPY] S. Abiteboul, H. Garcia-Molina, Y. Papakonstantinou, and R. Yerneni. Fusion query optimization. Available at `http://www-cse.ucsd.edu/ yannis/papers/fqo.ps`.

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conference*, pages 159–73, Portland, OR, May 1989.

[B+86] Y.J. Breibart et al. Database integration in a distributed heterogeneous database system. In *Proc. 2nd Intl. IEEE Conf. on Data Engineering*, Los Angeles, CA, February 1986.

[BDH+95] P. Buneman, S. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. VLDB*, 1995.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD*, 1996.

[Ber91] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. In *Proc Intl Workshop on Interoperability in Multidatabase Systems*, pages 22–29, Kyoto, Japan, 1991.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[C+95] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. RIDE-DOM Workshop*, pages 124–31, 1995.

[Cat91] R. G. G. Cattell. *Object Data Management.* Addison-Wesley, 1991.

[Cat94] R.G.G Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994. with contributions from Tom Atwood et.al.

[CHMW96] M. Carey, L. Haas, V. Maganty, and J. Williams. PESTO:An integrated query/browser for object databases. In *Proc. VLDB*, 1996.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. Hilog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, February 1993.

[CWN94] S. Chakravarthy, Whan-Kyu Whang, and S.B. Navathe. A logic-based approach to query processing in federated databases. *Information Sciences*, 79:1–28, 1994.

[DH86] U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. In *Proc. IEEE Workshop on Object-Oriented DBMS*, Asilomar, CA, September 1986.

[DIS] Corp. Data Integration Solutions. Integration Works. 18726 S. Western Avenue, Suite 405, Gardena, CA 90248.

[EH86] A. K. Elmagarmid and A. A. Helal. Heterogeneous database systems. Technical Report TR-86-004, Program of Computer Engineering, Pennsylvania State University, University Park, PA, 1986.

[EL85] C.F. Eick and P.C. Lockemann. Acquisition of terminological knowledge using database design techniques. In *Proc. SIGMOD*, pages 84–94, 1985.

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

[FLNS88] P. Fankhauser, W. Litwin, E.J. Neuhold, and M. Screfl. Global view definition and multidatabase languages: two approaches to database integration. In *Research into Networks and Distributed Applications. European Teleinformatics Conf.*, pages 1069–1082, Vienna, Austria, April 1988.

[Fre] M. Freedman. WILLOW: Technical overview. Available by anonymous ftp from `ftp.cac.washington.edu` as the file `willow/Tech-Report.ps`, September 1994.

[GN88] M.R. Genesereth and N.J. Nillson. *Logical Foundations of Artificial Intelligence.* Morgan Cauffman, 1988.

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

[H+92] M. Huhns et al. Enterprise information modeling and model integration in Carnot. Technical Report Carnot-128-92, MCC, 1992.

[H+95] J. Hammer et al. Information translation, mediation, and mosaic-based browsing in the TSIMMIS system. In *Proc. ACM SIGMOD Conf.*, page 483, May 1995.

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

[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.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. VLDB Conference*, pages 455–68, Brisbane, Australia, August 1990.

[Inc] CGI Systems Inc. PACBASE. One Blue Hill Plaza, P.O.Box 1645, Pearl River, NY 10965.

[Ire] K.L. Ireland. TSIMMIS union and join mediators detailed design specification. Available by anonymous ftp at `db.stanford.edu` as the file `/pub/ireland/mediator_design.ps`.

[K+93] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD*, pages 59–68, 1992.

[KL89] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf.*, pages 134–46, Portland, OR, June 1989.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of heterogeneous databases with schematic discrepancies. In *Proc. ACM SIGMOD*, pages 40–9, Denver, CO, May 1991.

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

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

[LOG93] H. Lu, B.-C. Ooi, and C.-H. Goh. Multidatabase query optimization: Issues and solutions. In *Proc. RIDE-IMS '93*, pages 137–43, Vienna, Austria, February 1993.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Query processing in the information manifold. In *Proc. VLDB*, 1996.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external processors. In *Proc. PODS*, pages 227–37, 1996.

[LSS93] L. Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proc. DOOD*, pages 81–100, 1993.

[LSS96] L. Lakshmanan, F. Sadri, and I.N. Subramanian. Schema SQL — a language for interoperability in relational multidatabase systems. In *Proc. VLDB*, 1996.

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

[Mai86] D. Maier. A logic for objects. In J. Minker, editor, *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, Washington, DC, USA, August 1986.

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

[ME84] M.V. Mannino and M.V. Effelsberg. Matching techniques in global schema design. In *Proc. IEEE COMPDEC*, pages 418–25, 1984.

[MI93] R. Miller and Y. Ioannidis. The use of information capacity in schema integration and translation. In *Proc. VLDB*, pages 120–33, 1993.

[MIR94] R. Miller, Y. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: bridging theory and practice. In *Proc. EDBT*, pages 73–80, 1994.

[MR87] L. Mark and N. Roussopoulos. Information interchange between self-describing databases. *IEEE Data Engineering*, 10:46–52, 1987.

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

[NT88] S. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases.* Computer Science Press, 1988.

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

[OV91] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[PDS95] P.Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proc. DBPL*, 1995.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for the rapid implementation of wrappers. In *Proc. DOOD Conf.*, pages 161–86, 1995.

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

[PGM] Y. Papakonstantinou and H. Garcia-Molina. Object fusion in mediator systems (extended version). Available by anonymous ftp at `db.stanford.edu` as the file `/pub/papakonstantinou/1995/fusion-extended.ps`.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Proc. ICDE Conf.*, pages 132–41, 1996.

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

[Qia96] Xiaolei Qian. Query folding. In *Proc. ICDE*, pages 48–55, 1996.

[QR95] X. Qian and L. Raschid. Query interoperation among object oriented and relational databases. In *Proc. ICDE*, pages 271–8, 1995.

[QRS+95] D. Quass, A. Rajaraman, S. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proc. DOOD*, pages 319–44, 1995.

[RJR94] R. Rao, B. Janssen, and A. Rajaraman. GAIA technical overview. Technical report, Xerox Palo Alto Research Center, 1994.

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

[Rud92] E. Rudensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. VLDB Conference*, pages 187–98, Vancouver, Canada, August 1992.

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

[S^+93] K. Shoens et al. The Rufus system: Information organization for semistructured data. In *Proc. VLDB Conference*, Dublin, Ireland, 1993.

[SAD94] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proc. EDBT*, pages 81–94, 1994.

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

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the acedb data base manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, 1992.

[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.

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

[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.

[Yer] R. Yerneni. The datamerge engine of medmaker. Available by ftp at `http://www-cse.ucsd.edu/ yannis/papers/ramana-engine.ps`.