Paper Number 530

Object Fusion in Mediator Systems*

Yannis Papakonstantinou, Serge Abiteboul, Hector Garcia-Molina
Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{yannis,abitebou,hector}@db.stanford.edu

Abstract

One of the main tasks of mediators is to fuse information from heterogeneous information sources. This may involve, for example, removing redundancies, and resolving inconsistencies in favor of the most reliable source. The problem becomes harder when the sources are unstructured/semistructured and we do not have complete knowledge of their contents and structure. In this paper we show how many common fusion operations can be specified non-procedurally and succinctly. The key to our approach is to assign semantically meaningful object ids to objects as they are "imported" into the mediator. These semantic ids can then be used to specify how various objects are combined or merged into objects "exported" by the mediator. In this paper we also discuss the implementation of a mediation system based on these principles. In particular, we present key optimization techniques that significantly reduce the processing costs associated with information fusion.

1 Introduction

The TSIMMIS system provides integrated access to heterogeneous information, stored not only in conventional databases but also in file systems, the Web, and legacy systems. The TSIMMIS architecture is shown in Figure 1. *Wrappers* [C$^+$94, FK93] (or *translators*) convert data from each source into a common model and also provide a common query language. Applications can access data directly through wrappers, but they may also go through mediators [PGMU, Wie92, S$^+$], which provide an integrated view of the data exported by the wrappers.

The architecture of Figure 1 is common in many integration projects [PGMW95, C$^+$94, A$^+$91, S$^+$, LMR90, K$^+$93]. However, the focus of our project is on semi-structured and/or unstructured information. This is information that may not conform to a rigid schema fixed in advance, and is frequently found, for instance, in the World-Wide-Web, SGML documents, semi-structured repositories such as ACeDB [TMD92] (very popular among biologists in the Human Genome Project), and Lotus NOTES. To represent such data, we use a "schema-less"(or self-describing [MR87]) object-oriented model, called *Object Exchange Model (OEM)* [PGMW95].

Mediators play the central role in information integration, and their most important task is to perform *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

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. This work was also supported by an equipment grant and a fellowship from IBM Corporation.

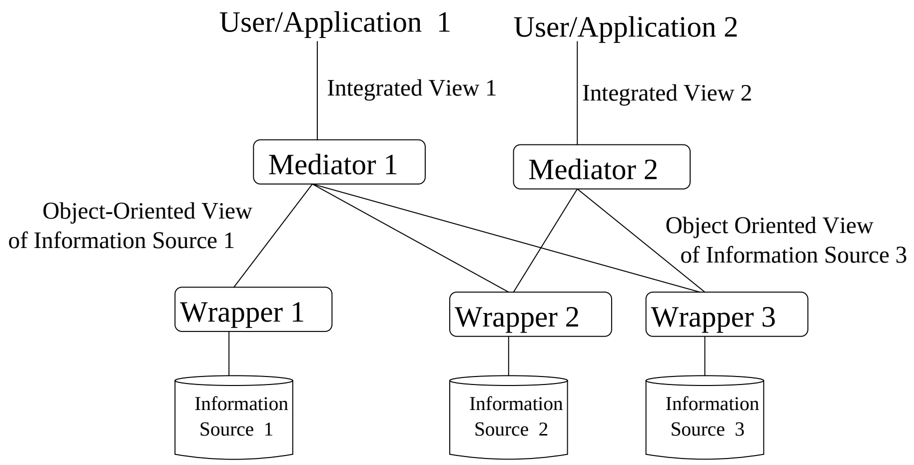

Figure 1: The TSIMMIS architecture for integration

information by removing redundancies, resolving inconsistencies between sources in favor of the most reliable source, and so on. A mediator may also have to avoid accessing a particular source if another source provides similar information at a lower cost (either financial or computational).

In this paper we present an 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, that 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.

For our specifications we use *MSL (Mediator Specification Language)*, originally introduced in [PGMU]. However, in the original MSL, object fusion could not be performed. We have extended MSL to allow rules to specify object fragments that can be fused together. As we will see in this paper, this single concept 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 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.

However, this added power also places a new burden on the query processing unit *(the Mediator Specification Interpreter, or MSI)*. In particular, query processing is complicated because it may not be clear in advance which sources contribute to the content of a single mediator object. Furthermore, nontrivial computations may have to be performed for refining the integrated objects. To address these problems, in this paper we present a set of optimization strategies that can significantly reduce the number of queries sent to sources, and can as well cut down on the volume of data that are fetched. This is achieved using the following techniques:

- We reconsider standard algebraic optimization techniques such as "pushing selections to the wrapper" that have been used in [PGMU]. Object-id based fusion makes these optimizations much harder. Nevertheless, we achieve significant gains by using a modification of *resolution-based* techniques from deductive systems.

- We develop an MSL variant of the well-known subsumption technique [Ull89] to limit the number of queries sent to sources.

- We present intelligent schemes to avoid retrieving information that will eventually be discarded. For example, if two sources `s1` and `s2` provide conflicting information about some object and the specification indicates that the conflicts are resolved in favor of `s1`, the mediator will not query `s2` for information that is already provided by `s1`.

- Because the structure and contents of the information at sources may not be known in advance, it is sometimes hard to discover which source contributes specific pieces of information to a fused object. If MSI is not careful it may end up requesting some source for information that can not be provided by this source. MSI resolves this problem by preceding the information fusion phase with an information finding phase that rules out sources that can not contribute to the fused objects.

These strategies have been successfully used in MedMaker, an integration system implemented at Stanford. These optimization strategies and the MedMaker components that deal with object fusion are described in Section 4. Before that, we give a brief overview of the OEM model in Section 2, and in Section 3 we give a sequence of examples that illustrate the power of MSL and of fusion based on semantic object identifiers.

2 The OEM Model

Most 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.

To illustrate the OEM model, consider a source that contains bibliographic information. A wrapper, named `s1`, exports this information as a set of OEM objects, some of which are shown in Figure 2 (one object per line.) Notice how the schema information has now been incorporated into the labels of individual OEM objects.

Each OEM object consists of an *object-id* (e.g., `&r1n`), a *label* that explains its meaning (e.g., `report_num`), a *type* (e.g., `string`), and a *value* of the specified type (e.g., `'AB-123-456'`.) Labels are strings that are meaningful to applications or end-users. Labels may have different meanings at different sources. Indeed, it will be the job of mediators to resolve these conflicts. Values may be either of an atomic type (e.g., `'John Patriot'` is of type string), or be a set of sub-objects object-id's (e.g., the value of the `related` object is {`&r2`}). To simplify the presentation, in the rest of this paper, we assume that the type of all *atomic* objects is `string` and we omit type information from objects.

From the point of view of the OEM model, object ids are arbitrary strings (starting with `&`) that are used to link objects with their sub-objects (e.g., `&r1t` links the report of Figure 2 to its title). Some OEM objects (e.g., the objects identified by `&r1`, `&r2`) are *top-level* objects. In our presentation they are the leftmost indented objects. For performance reasons, clients query object structures starting, by default, from the top-level objects. For example, a simple query may ask for top-level `report` objects that have a `year` sub-object with value `1988`. Nevertheless, the

```
<&r1, report, set, {&r1n, &r1a, &r1t, &r1y, &r1r}>
      <&r1n, report_num, string, 'AB-123-456'>
      <&r1a,authors, set, {&b1a1}>
            <&r1a1, author, string, 'John Patriot'>
      <&r1t, title, string, 'UN Conspiracies'>
      <&r1y, year, string, '1995'>
      <&r1r, related, set, {&r2}>
              ⋮
<&r2, report, set, {&r2n, &r2a, $r2t, &r2y, &r2r}>
```

Figure 2: The OEM object structure of `s1`

client is not restricted to query the object structure starting from top-level objects, as is explained in [PGM].

When we represent OEM objects within a mediator, we may use semantically meaningful object-id's to facilitate with the integration tasks. For example, if the `report_num` number of the report in Figure 2 is a key and can be used to match up this report with other reports that should be considered the same entity, then we can use `&AB-123-456` as the id for the report. Furthermore, if this report object originally came from another source `sss`, then we extend the id to `&AB-123-456@sss`. This convention is easy to implement and simplifies fusion: Objects that need to be fused can be identified by their ids, yet the source of the information is clearly noted as to avoid confusion.

Finally, note that OEM poses no restrictions on the labels of sub-objects. For example, some `report` objects have a single `title` object, others may not have any `title`, and others may have multiple `title`s. In this way, OEM allows us to represent and integrate information from unstructured sources. Note also that OEM objects may be arbitrarily nested.

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 MSL and demonstrates the basic principle of id based fusion. We than 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.

3.1 A Simple Example

Let us consider a mediator called `simple` 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.

```
(MS1) (R1.1) <trep(RN) techreport {<title T>}>@simple :-
                    <report {<report_num RN> <title T>}>@s1
```

```
(R1.2) <trep(RN) techreport {<postscript P>}>@simple :-
                  <report {<report_num RN> <postscript P>}>@s2
```

A specification consists of *rules* that define the view exported by the mediator. Each rule consists of a head followed by a :- and a tail. The head describes view objects, whereas the tail describes conditions that must be satisfied by the source objects. In general, the heads and tails are based on patterns of the form <*object-id label value*>. We may omit the object-id field when it is irrelevant. If it is missing from a tail pattern it means that we do not care about the object-id appearing at the source. If it is missing from a head pattern it means that the mediator has to invent an arbitrary, yet unique, object-id for the "generated" object.

Going back to our example, 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 simple 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 (MS1.1) does not prevent the techreport with object-id trep(r) to have 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.

To illustrate how MSL specifications are used at run time, assume that a client of simple wants to retrieve all data of techreport's with object-id trep('123'). As our query language, in this paper, we use MSL itself (with minor modifications discussed below). The use of MSL simplifies our discussion, and furthermore, MSL is convenient because of its expressive power.[1] Using MSL our query can be expressed as:

```
(Q2) <trep('123') techreport V> :- <trep('123') techreport V>@simple
```

The object pattern (or patterns in the general case) that appears in the query tail is evaluated against the object structure of the mediator in exactly the same way that the mediator specification rule tails are evaluated against the object structures of the wrappers. The object pattern of the query head does not include the usual @ notation because it is implied that the objects described by the query head refer to the result that will be materialized at the client.

Given the sample query (Q2) and the mediator specification (MS1), the Mediator Specification Interpreter (MSI) develops a plan (through a series of steps described in Section 4) that specifies the queries that will be sent to the sources and how the received results will be transformed and combined into the answer objects. In our example, MSI sends to s1 and s2 the queries (Q3) and (Q4).

[1]The TSIMMIS project at Stanford also uses a different query language called *LOREL* $[Q^+]$. It is an object-oriented extension to SQL and is oriented to end-users. LOREL queries are translated to MSL queries before being processed by mediators.

```
(Q3) <trep('123') techreport {<title T>}> :-
                              <report {<report_num '123'> <title T>}>@s1
(Q4) <trep('123') techreport {<postscript P>}> :-
                              <report V:{<report_num '123'> <postscript P>}>@s2
```

The two answer objects received from `s1` and `s2` are then merged into a single `techreport` object. Note, MSI then copies the fused answer to the client's memory. The subobjects of the `techreport` objects are also copied, and so on recursively up to some maximum depth (imposed by the size of the client's memory).

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 (MS5) demonstrates that we can group all information about reports into `techreport` objects, without knowing the structure and contents of the reports subobjects.

```
(MS5) (R5.1) <trep(RN) techreport V>@all :- <report V:{<report_num RN>}>@s1
      (R5.2) <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 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.

3.3 Removing Redundancies

The example of Section 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 (MS6). 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.

```
(MS6) (R6.1) <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
      (R6.2) <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. MSL contains a variant of outerjoin (see extended version [PGM]) that could be used to implement the example above. 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.

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 (MS7), that collects from `s2` only information about reports that do not appear in `s1`.

```
(MS7) (R7.1) <trep(RN) techreport V>@save :- <report V:{<report_num RN>}>@s1
      (R7.2) provides(RN) :- <report {<report_num RN>}>@s1
      (R7.3) <trep(RN) techreport V>@save :-
                 NOT provides(RN) AND <report V:{<report_num RN>}>@s2
```

Rule (R7.1) declares that every `report` of `s1` becomes a `techreport` of `save`. Then (R7.2) 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, (R7.3) 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 4.6 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 (MS7). 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 (R7.2) 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 [PGMU] 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.

```
(MS8) (R8.1) <trep(RN) techreport V>@med :- <O report V:{<report_num RN>}>@s1
                                            AND NOT notbest(O,RN)
      (R8.2) <trep(RN) techreport V>@med :- <O report V:{<report_num RN>}>@s2
                                            AND NOT notbest(O,RN)
      (R8.3) provides(O,RN,P) :- <O report {<report_num RN> <date D>}>@s1
                                 AND calc(D,P)
      (R8.4) provides(O,RN,P) :- <O report {<report_num RN> <date D>}>@s2
                                 AND calc(D,P)
      (R8.5) 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 (R8.1) and (R8.2) do not retrieve `report` objects whose object-id's `O` appear in the `notbest` relation.

3.5 Removing Inconsistencies Using Fine-Grained Blocking

In Section 3 we showed that specifications such as (MS1) 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, (MS9) 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.[2]

```
(MS9 (R9.1) <trep(RN) techreport {<field(RN,F) F V>}>@med :-
                 <report {<report_num RN> <F V>}>@s1
     (R9.2) provides(RN,F) :- <report {<report_num RN> <F V>}>@s1
     (R9.3) <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 (R9.3) 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`.

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

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[3] (see example OEM structure of Figure 2.) 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 (MS10). Rule (R10.1) puts in the `techreport` objects all information of the source reports with the exception of the `related` subobject. Rule (R10.2) creates a `related` object and inserts it into the appropriate `techreport`. For simplicity we omit the corresponding rules for `s2`.

```
(MS10) (R10.1) <trep(RN) techreport {<L X>}>@all-with-ref :-
                   <report {<report_num RN> <L X>}>@s1 AND NOT L=related
       (R10.2) <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.

```
(MS11) (R11.1) <trep(RN) techreport {<L X>}>@all-with-ref :-
                   <report {<report_num RN> <L X>}>@s1
       (R11.2) <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.

3.7 Other Types of Fusion

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

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

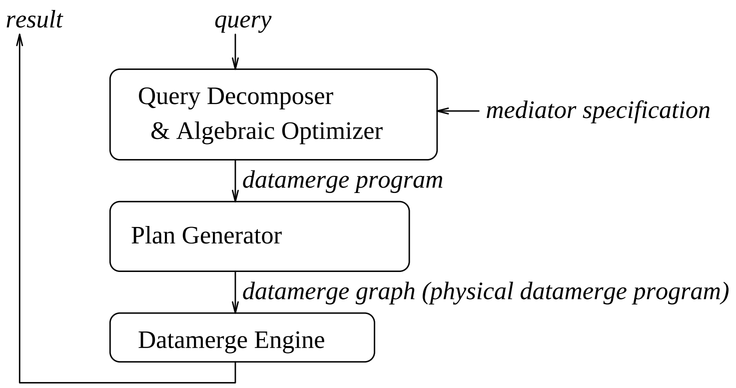

Figure 3: The basic architecture of MSI

- *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 paper. 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 or maximum temperature for the fused object. Such functions are incorporated into MSL specifications as external predicates [PGMU]. However, it is important to note that it becomes very expensive to search over such fused data. For example, if we want to search for objects with average temperature of 80 degrees, it is not easy to push any condition on temperature to the source. Since our focus is on efficient searching (see next section), we concentrate here on mediator specifications that allow MSI to efficiently process the query.

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 Architecture and Implementation of MSI

The Mediator Specification Interpreter (the run-time component of MedMaker) processes a query using the following three components (see Figure 3):

1. The *Query Decomposer and Algebraic Optimizer (QD&AO)* reads the query and the mediator specification and determines the conditions that source objects must satisfy to contribute to the answer. It also determines how the source objects are combined to construct the required query result. QD&AO tries to minimize the number and result sizes of queries sent to the sources. QD&AO produces a *datamerge program*.

2. The *Plan Generator* develops an *execution plan* for obtaining and combining the objects specified by 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. The plan generator produces a *physical datamerge program*.

3. The *Datamerge Engine* executes the physical datamerge program and produces the result.

In [PGMU], we described how the above components work when the MSL specifications do not contain semantic object-id's. Object-id based fusion presents a number of new challenges, and in this paper we focus on handling them using unification and resolution concepts. Subsection 4.1 reviews the basic QD&AO algorithm, while the necessary extensions for object fusion are described in Subsection 4.2. The remaining subsections describe various optimization techniques that are particularly effective for object fusion.

4.1 The Basic Query Decomposition and Algebraic Optimization Algorithms

In this section, we show how QD&AO formulates a *datamerge program* from a query and a mediator specification. Recall, the query refers to mediator objects. It is first transformed into a datamerge program that refers to source objects only. More precisely, a datamerge program is a collection of rules whose tails refer only to the source object structures and whose heads describe the object structure of the answer objects. The datamerge rules push as many conditions as possible to sources.

As a first step QD&AO transforms queries and mediators into *normal form MSL*. Normal form MSL is very similar to full MSL except that patterns always have three fields and certain constructs (like `V:{<title 'abc'>}`) are not allowed. Having fewer and more regular constructs simplifies the query processing work that follows. In the extended version of the paper [PGM], we give the syntax of full and reduced MSL and present an algorithm for converting expressions into normal form MSL. As an example, the algorithm converts the query

(Q12) `<X tr V> :- <X tr V:{<title 'abc'>}>@m`

into the query

(Q13) `<X tr {<Void Vl Vv>}> :- <X tr {<T2 title 'abc'> <Void Vl Vv>}>@m`

As the second and main step, QD&AO generates a logical datamerge program by matching the query tail conditions with rule heads. The process considers each condition c in the query tail, starting from the leftmost. (In our initial example there is only one condition, but later examples will show multiple conditions.) Condition c is compared against rule heads; c matches a rule r if the rule can produce objects that satisfy the condition. Each successful match produces a *unifier* that describes the match between c and r. For each unifier, we replace the condition c by conditions on the sources specified in r (see below). In the tail of this datamerge rule we still have the remaining query tail conditions which may refer to mediator objects. For each of these, we repeat the process of unifying them against some mediator rule until the tail of the datamerge rule only refers to objects at the sources.[4]

To illustrate consider the following mediator (MS14) that contains a single rule. (Note, there is no information fusion in this example.)

(MS14) (R14.1) `<trep(RN) tr {<O L X>}>@m1 :- <r {<rn RN> <O L X>}>@s1`

Let us now consider the query (Q15) that retrieves the `tr` objects where the object-id is `trep('123')`.

(Q15) `<trep('123') tr V> :- <trep('123') tr V>@m1`

The match of this query and specification (MS14) results in the single unifier θ where

$$\theta = [(R14.1) : \texttt{RN} \mapsto '123', \texttt{V} \mapsto \{< \texttt{O L X} >\}]$$

The above unifier maps the variables to the left of $\mapsto$ to the constructs to the right of $\mapsto$. In general, variables map to constants, variables, terms, or set patterns of the form $\{< o_1\ l_1\ v_1 > \ldots < o_n\ l_n\ v_n >\}$. (Note, the latter case (mapping to set patterns) differentiates between our unifiers and unifiers of first-order logic.) The unifier also contains the name (R14.1) of the rule that matched to the query.

After the unification, we apply θ to the query and the rule and we replace the transformed query condition with the transformed rule tail of (R14.1). When θ is applied to the query head `V` is substituted by `{<O L X>}`. Similarly, applying θ to the rule tail of (R14.1), we replace `RN` by `'123'`. Thus, we derive datamerge rule (DR1).

(DR1) `<trep('123') tr {<O L X>}> :- <r {<rn '123'> <O L X>}>@s1`

Formal Specification of Unifiers. To define the matching process more precisely, we give a few additional details. The notation $\theta(e)$ represents the expression e where the substitutions indicated by unifier θ have been performed. A condition e_1 *matches with the head* e_2 of rule r if there is *a unifier* θ *from* e_1 *to* e_2, as described by Definition 4.1 below. (Note, both e_1 and e_2 are MSL patterns.)

Definition 4.1 (Unifier θ from e_1 to e_2) A mapping 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 4.2. □

Definition 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 $\{e_1^1, \ldots, e_1^n\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ and for every pattern $e_1^i, i = 0, \ldots, n$
there is a pattern $e_2^j, j = 0, \ldots, m$ such that e_1^i is included in e_2^j.
else e_1 and e_2 have the same value field. □

[4] It is easy to see that in the absence of recursion this process terminates. In the presence of recursion more complex resolution strategies are required [GN88]. 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 [GN88].

For example, the condition of query (Q15) matches with (R14.1) because the unifier θ maps the condition to the rule head. This is because the pattern θ(`<trep('123') tr V>`) =`<trep('123' tr {<O L X>}>` is included in the pattern θ(`<trep(RN) tr {<O L X>}>`) = `<trep('123') tr {<O L X>}>`. (In this example the patterns are identical but this is not necessary in general.)

The algorithm for computing the unifiers from a pattern s to a pattern r and the algorithm for applying a unifier θ to a pattern p are given in the extended version of this paper [PGM]. Using these algorithms it is straightforward to develop datamerge programs (as described above). Note, computing unifiers is important not only for developing datamerge programs but also for performing the subsumption based optimizations of Sections 4.3 and 4.5.

4.2 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. This is more challenging because we have to simultaneously match the query tail conditions with the heads of more than one rule. In this section, we generalize our QD&AO algorithm to cover this case.

Let us consider mediator (MS1) of Section 3 that merges information from sources `s1` and `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; this is to 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 paper.)

```
(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` query (Q13) 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, we rewrite the query (Q13) as (Q17):

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

In this transformed form, we break up the tail so that every set pattern `{ ... }` contains exactly one object pattern `< ... >`. Such a transformation is straightforward.[5]

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 r {<T2 title 'abc'>}>`.[6]

[5]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.

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

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> <T1 title 'abc'>}>@s1 AND
                    <trep(RN1) r {<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) r {<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) r {<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 (R16.1) again for the matching the second condition. Thus, we introduce a second instance of (R16.1) (see rule (R16.1.b) below) and we match <trep(RN1) r {<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
```

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

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 mediator specification has variables in label positions. We will study techniques for reducing the number of datamerge rules in the following sections. The extended version [PGM] presentes formally the general query decomposition and unification steps necessary for object fusion.

4.3 Subsumption-Based 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. 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.

Due to space limitations we only illustrate query reuse and not rule elimination. 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
```

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 are 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.

4.4 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

```
(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 4: Datamerge program

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 (MS5) in Section 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 4.1) that results in the datamerge program (DP22) of Figure 4.

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).

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: 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 many conditions down we restrict the amount of retrieved data but we increase the number of rules and queries. If we push fewer conditions, 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.

4.5 Plan Generation

Our remaining logical optimizations are applied during or after physical plan generation. Thus, we start by briefly describing how physical plans are obtained. ([PGMU] explains this in more detail.) Then, in the remainder of the paper we discuss logical optimizations to the physical plans.

The cost-based optimizer receives the datamerge program and creates 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, is one of the possible physical datamerge programs (from now on referred as physical programs) for (DP22). The notation `@(Q24,s1)` in physical rule (PDR23.1) indicates that query

```
(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: A Physical Datamerge Program

(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?", "what is the expected cardinality of `bind1(RN)`?", and so on.) We will not deal in this paper with these cost-based optimization problems.

Query Subsumption Optimization In Section 4.3 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.

4.6 Optimization of Negation Operations

In Section 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 (MS7) 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 (Q13) 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 last case, blocking could not really be exploited to reduce the data retrieved from `s2`.

5 Discussion and Related Work

Object fusion is an important data integration task. It involves collecting and grouping information about the same real-world entity, and removing inconsistencies and redundancies. The task becomes harder when we do not have complete knowledge of the structure of the underlying data, as in the World Wide Web, for instance.

In this paper we have shown that the OEM data model and the MSL mediator specification language, each extended with semantic object-ids, provide a conceptually simple yet powerful framework for object fusion. The key features of this framework are:

- *Conceptual simplicity:* MSL is an object-oriented logic. The rules use a small but general set of features that consists of: (a) semantic object-id's, (b) negation, and (c) variables that can range over object-id's, labels, and values.

- *Groupings of fused information:* Multiple MSL rules can monotonically and independently add information to fused objects by specifying the semantic object-id of the fused object. The combined use of negation and attribute variables allows the specification of complex conflict resolution schemes.

- *Adaptability:* Fusion specifications can be adapted to various levels of knowledge about the sources. At one end, the designer knows little about the contents and structure (if any) of a source, so he writes "generic" specifications that can handle any structure. At the other end, the source has some known regularity, so the designer can improve the quality and run-time efficiency of object fusion.

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, $A^{+}91$, $C^{+}94$, 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, in the present paper, we assume minimal knowledge of the structure and contents of the sources.

Querying and integrating semistructured data is also considered in [$C^{+}94$, FK93, Q^{+}, PGMU, 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 sometimes is a handicap [$C^{+}94$, $BDH^{+}95$]. 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]).

MSL is an object-oriented logic, but has certain simplifying features. In particular, a number of problems are avoided by not considering sets as first-class citizens. (Variables may explicitly refer only to existing sets of objects.) Indeed, in absence of negation and semantic object-id's, MSL can be viewed simply as a variant of datalog (see [Ull88]). In the extended version of paper [PGM] we present the reduction of MSL to datalog with function symbols and negation. In absence of recursion, MSL can be viewed as a variant of OQL [Cat94]. However, unlike datalog and OQL, MSL makes it possible to handle both unstructured and structured data.

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 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).

[LSS93] proposes a logic with higher-order syntax and first-order semantics for schema integration and evolution and also demonstrates the need for higher order views. MSL achieves the same effects with the use of label variables. ([KLK91] achieves the same objective by assuming special virtual relations that contain the metadata information of the repositories relation.)

Finally, 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. In this paper we presented 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. Beyond this, we also investigated optimizations needed for reducing the number and volume of data that are retrieved from the sources during object fusion.

As a concluding remark, we note that we have developed a prototype system that fully implements the query decomposition and evaluation algorithm described here. It also features the subsumption based optimizations and some of the optimization techniques of Section 4.6. The system has been demonstrated on a collection of heterogeneous bibliographic sources. We are currently working on including the rest of the optimization techniques we have described into the prototype.

References

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

[ACHK93] Y. Arens, C.Y. Chee, C.-N. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *International 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*, 1993.

[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.

[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.

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

[Cat94] R.G.G Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994. with contributions from Tom Atwood et.al.

[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.

[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.

[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.

[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.

[HM93] J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *International 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.

[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, San Diego, California, June 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.

[LSS93] L. Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous datbase systems. In *Proc. DOOD*, pages 81–100, 1993.

[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, Sept 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 *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[PDS95] P.Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proc. DBPL*, 1995.

[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`.

[PGMU] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. To appear in ICDE96. Available by anonymous ftp at `db.stanford.edu` as the file `/pub/papakonstantinou/1995/medmaker.ps`.

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

[Q+] D. Quass et al. Querying semistructured heterogeneous information. To appear in DOOD95. Available by anonymous ftp at `db.stanford.edu` as the file `/pub/quass/1994/querying-submit.ps`.

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

[SAD94] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proc. EDBT, Cambridge*, 1994.

[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.

[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.