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^+95] 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 [PGMU96, S^+], which provide an integrated view of the data exported by wrappers.

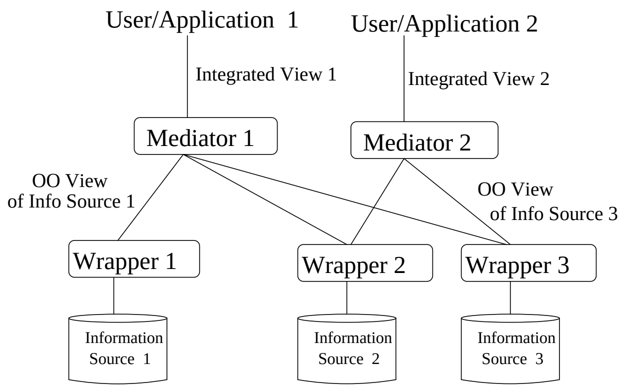

Figure 1: The TSIMMIS architecture

The architecture of Figure 1 is common in many integration projects [PGMW95, C^+95, 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" 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, possibly removing redundancies, or perhaps resolving inconsistencies between

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

sources in favor of the most reliable source.

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 declarative, logic rules. Each rule maps 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. 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)* [PGMU96]. However, the original MSL did not include semantic object-id's, hence making the specification of fusion much harder. We extend MSL with semantic object-id's, specified as skolem functions, that allow rules to specify object fragments that can be fused together. Construction of object-id's as skolem functions has been extensively studied by the deductive and object-oriented database systems community, hence providing us with a clear theoretical foundation.

Our first contribution is to show that the single concept of "object-id's as skolem functions," cast in the appropriate practical framework, significantly increases the power of the specification language. This makes it relatively easy to stipulate powerful fusion mediators. For example, one of the most challenging aspects of fusion is the integration of source objects that contain references to other source objects. We will show how semantic object-ids can be used to translate source references into semantically meaningful references at the mediator, allowing the integration of nested and cross-referenced objects such as those found on the Web.

A second contribution is the adaptation of *resolution* and *subsumption* for the purpose of efficient query processing against mediators that perform fusion.

Our last contribution is a set of novel optimization strategies that are specific to information fusion in an environment of semistructured sources. For example, with heterogeneous semistructured sources , it may not be known in advance what source a particular condition should be pushed to. In this case traditional techniques force the query processor to explore an exponential number of options for pushing selections to the sources. We demonstrate alternative query processing policies that most often perform substantially better. For example, we precede the information fusion phase with an information finding phase that attempts to rule out sources that can not contribute some particular data required by the query. We also 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 does not query `s2` for information that is already provided by `s1`.

Before proceeding, we make a few remarks that may help position our work. The first is that our specification language, MSL, is not intended as an end-user language. As a matter of fact, our goal has been to have a few simple but powerful language constructs. This simplicity has turned out to be essential for the development of efficient optimizers and execution strategies.

Second, the query processing algorithms, with the optimizations described here, have been implemented as part of our TSIMMIS system. The need for sophisticated processing algorithms became apparent as the system was being built and tested, and most of the techniques presented here were developed after an initial version [PGMU96] identified important weaknesses.

Third, since there is not enough space to go into the full details, in this paper we mainly use examples to demonstrate the power of MSL for fusion problems and to explain the key ideas of our query decomposition and optimization techniques. The precise syntax and semantics of the language can be found in [PGM]. Further details on the underlying algorithms can be found in [PGM] or in our implementation that will soon be ftp-available.

The outline for our paper is as follows. In Section 2, we give a brief overview of the OEM model. Section 3 covers typical fusion problems, and shows how MSL and semantic ids can address them. The query evaluation and optimization strategies are then described in Sections 4 and 5. In Section 6 we briefly survey the prior work that our system builds upon.

2 The OEM Model

Most applications that have to deal with semistructured information use a *self-describing* model, where each data item has an associated descriptive label. Applications include tagged file systems, Lotus NOTES, electronic mail, RFC1532 bibliographic records, and many more. In [PGMW95] we have defined a self-describing data model, 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 below (one object per line.) Notice how the schema information has now been incorporated into the labels of individual OEM objects.

```
<&r1,report,set,{&r1n,&r1a,&r1t,&r1r}>
  <&r1n,rn,string,'AB-123'>
  <&r1a,authors,set,{&r1a1}>
        <&r1a1,author,string,'John Patriot'>
  <&r1t,title,string,'UN Conspiracies'>
  <&r1r,rel,set,{&r2}>
<&r2,report,set,{&r2n,&r2a,&r2t,&r2r}>
```

Each OEM object consists of an *object-id* (e.g., `&r1n`), a *label* that explains its meaning (e.g., `rn` that stands for report number), a *type* (e.g., `string`), and a *value* of the specified type (e.g., `'AB-123'`.) Labels are strings that are meaningful to applications or end-users. Labels may have different meanings at different sources. Values may be either of an atomic type (e.g., `'John Patriot'`), or be a set of sub-objects object-id's (e.g., the value of the `rel` 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 strings (starting with `&`) that are used to link objects with their sub-objects (e.g., `&r1t` links the `report` to its `title`). We may use semantically meaningful object-id's to facilitate the integration tasks. For example, if the report number `rn` of the report objects is a key and can be used to identify this report with other reports that should be considered the same entity, then we can use `&AB-123` as the id for the report (instead of `&r1`). Furthermore, if this report object originally came from another source `sss`, then we extend the id to `&AB-123@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 to avoid confusion.

Some OEM objects (e.g., the objects identified by `&r1`, `&r2`) are "root" or *top-level* objects and are shown with left-most indentation. They represent the starting point for queries to the sources.

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.

3 Object-Identity Based 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 then present examples that illustrate a variety of fusion operations.

3.1 A Simple Example

Let us consider a mediator called `m` that exports technical report objects with label `tr`. The `tr` 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 `tr` object contains the corresponding `title`. If source `s2` contains the postscript for this report, then a `postscript` subobject is also included in the `tr`. Note, the specification of the `tr` object uses two rules. Each rule describes the contribution of only one of the sources.

```
<trep(RN) tr {<title T>}>@m :-          (MS1) (R1.1)
     <report {<rn RN> <title T>}>@s1
<trep(RN) tr {<postscript P>}>@m :-           (R1.2)
     <report {<rn 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. (The id's are invented using skolemization [PGM].)

Going back to our example, rule (R1.1) 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 `rn` subobject with value r and a `title` subobject with value t, *then* mediator `m` exports a `tr` object, with object-id `trep`(r), that has a `title` subobject with value t and a unique system-generated object-id.

The semantics of rule (R1.2) are defined accordingly. Notice how `tr` 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 (R1.1) does not prevent the `tr` with object-id `trep`(r) to have subobjects other than `title`, thus allowing the second rule to add more subobjects to the same `tr` 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 this example we assumed that source objects had some semantic key (like rn) that could be used for fusion. Often keys exist but are represented differently at sources. As a trivial example, report numbers could be represented as integers at s1 whereas s2 may represent them in the string format. In this case, we can build and use an *external predicate* [PGMU96] that converts the string format to integer to map the key in s2 into the form used in the semantic id. In the rest of our examples, we continue to assume that matching keys already exist, but keep in mind that this is equivalent to saying that keys can be converted to a canonical form.

3.2 Merging Information

It is not necessary to know the structure of the source reports in order to fuse them. Specification (MS2) demonstrates that we can group all information about reports into tr objects, without knowing the structure and contents of the reports subobjects.

```
<trep(RN) tr V>@all :-                    (MS2) (R2.1)
     <report V:{<rn RN>}>@s1
<trep(RN) tr V>@all :-                          (R2.2)
     <report V:{<rn 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 rn. Then, every object of the set value becomes a subobject of the tr object, 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 simultaneous 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 show next how this can be done. The person writing the mediator specification can decide if redundancies or inconsistencies are allowed.

3.3 Removing Redundancies

(MS2) generates one redundancy that is not useful: each tr object contains two rn subobjects with identical values but different object-id's. This redundancy can be eliminated as shown by mediator (MS3). It assigns the semantic object-id rnOID(RN) to the rn subobjects with value RN. In this way, the rn subobjects that have the same value are assigned the same object-id and hence they degenerate into the same rn object.

```
<trep(RN) tr {<rnOID(RN) rn RN>   (MS3) (R3.1)
              <O1 L1 X1>}>@n :-
 <report {<rn RN><O1 L1 X1>}>@s1 & NOT L1=rn
<trep(RN) tr {<rnOID(RN) rn RN>         (R3.2)
              <O2 L2 X2>}>@n :-
 <report {<rn RN><O2 L2 X2>}>@s2 & NOT L2=rn
```

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 rn. Then, the subobjects that are bound to <O1 L1 X1> or <O2 L2 X2> become subobjects of the tr objects. (If we did not have explicit NOT conditions the pattern <O1 L1 X1> and <O2 L2 X2> would also match with rn 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 tr 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. This number 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

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. We also 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 s that collects from s2 only information about reports that do not appear in s1.

```
<trep(RN) tr V>@s :-                          (MS4)
   <report V:{<rn RN>}>@s1
provides(RN) :- <report {<rn RN>}>@s1
<trep(RN) tr V>@s :-
   <report V:{<rn RN>}>@s2 & NOT provides(RN)
```

The first rule declares that every `report` of `s1` becomes a `tr` of `s`. Then the second rule 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 (e.g., `<provides RN>@t`, where `t` is an "intermediate" mediator) but the use of relations often makes the specification clearer.

Finally, the third rule exports a `tr` for every `report` of `s2` unless the report appears in the relation `provides`. Note, we use traditional "negation as 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 5.3 we demonstrate techniques used by the query optimizer that prevent the mediator from retrieving "blocked" data from the wrappers.

There are many variations for blocking data. In [PGM] we present an example where an external predicate assigns "reliability degrees" to the source reports and then we create a view where only the most reliable copies appear.

3.5 Removing Inconsistencies

In Section 3 we showed that specifications such as (MS1) may cause the same `tr` 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, (MS5) 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.[1]

```
<trep(RN) tr {<f(RN,F) F V>}>@p   (MS5) (R5.1)
     :- <report {<rn RN> <F V>}>@s1
provides(RN,F) :- <report {<rn RN> <F V>}>@s1
<trep(RN) report {<f(RN,F) F V>}>@p      (R5.3)
     :- <report {<rn RN> <F V>}>@s2
        & NOT provides(RN,F)
```

The subgoal `NOT provides(RN,F)` blocks (R5.3) from exporting any subobject with label f of a `report` with number 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`.

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 `rel` whose value is a set containing the `s1` object ids of the referenced reports[2] (see example OEM structure of Section 2.) If we are not careful when we import `rel` into the mediator, we will end up with object references that point to the original objects of `s1` and not to the corresponding fused `tr` objects.

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

The first solution is implemented by (MS6). Rule (R6.1) puts in the `tr` objects all information of the source reports with the exception of the `rel` subobject and it also adds a semantic object-id, namely `trep(RN)`, that is used for object fusion. Rule (R6.2) creates `rel` objects and inserts in them the corresponding `tr` objects. For simplicity we omit the corresponding rules for `s2`.

```
<trep(RN) tr {<L X>}>@a :-          (MS6) (R6.1)
  <report {<rn RN> <L X>}>@s1 & NOT L=rel
<trep(RN) tr                               (R6.2)
          {<rel {<trep(REL) tr {}>}>}>@a
 :- <report {<rn RN>
             <rel {<report {<rn REL>}>}>}>@s1
```

Our second solution does not rely on knowing which subobjects refer to source objects. The basic idea is to create two virtual objects for each `tr`. The first virtual object (as before) has the id `trep(RN)` and its `rel` 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

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

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

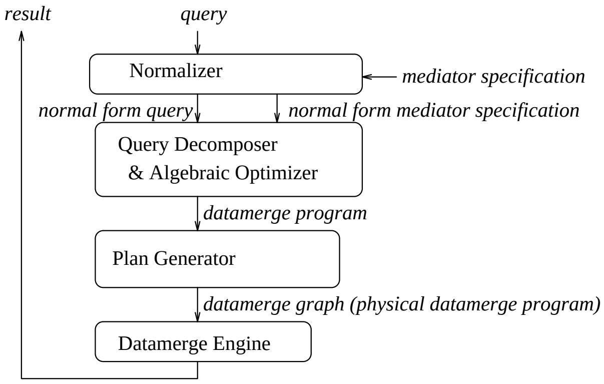

Figure 2: The basic architecture of MSI

to combine fragments from other sources. The second copy is simply used so that ids in the first refer to valid mediator objects.

```
<trep(RN) tr {<L X>}>@a :-                    (R7.1)
      <report {<rn RN> <L X>}>@s1
<O tr V>@a :- <tr V:{ <rn RN> }>@a            (R7.1)
      & <O report {<rn RN>}>@s1
```

The first rule (and the analogous one for s2 that is not shown) generates the first copy of each tr 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 tr objects; these "reuse" the same subobjects, such as title. Furthermore, the copies are virtual.

In summary, this section illustrated how fusion can be specified with MSL. In [PGM] we give additional examples and discuss how fusion can be done when there are no keys (like report number) to aid us.

4 Query Processing

In [PGMU96] we describe how the *Mediator Specification Interpreter (MSI)* processes queries in the absence of object-id's and fusion. In this section we focus on the system extensions for processing specifications with object-id's (and hence fusion). The extended MSI has the following four components (see Figure 2):

1. The *Normalizer* reduces the query and the specification into a normal form that facilitates the next steps when semantic ids are involved.

2. The *Query Decomposer and Algebraic Optimizer (QD&AO)* reads the query and the mediator specification and produces a *logical datamerge program* that determines, at a logical level, how the source objects are combined to construct the required query.

3. The *Plan Generator* develops the *physical datamerge program*, which specifies in detail the execution strategy that will be used, i.e., what queries will be sent to the sources, and so on.

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

In the rest of this section we show how the first three components operate, using extensions of resolution, unification and subsumption from classical deductive systems. We do not discuss here how the fourth component (datamerge engine) works, since object-id based fusion does not require any major changes. In Section 5 we discuss novel optimization techniques for the algebraic optimizer and plan generator. These techniques are specific to dealing with fusion of semi-structured information. As stated earlier, our goal here is to explain the fundamental ideas mainly through examples, leaving the full details for [PGM] (or in the actual implementation code that will be ftp-available soon).

4.1 Normalization, Query Decomposition, and Algebraic Optimization

The top two components of our system formulate a *logical datamerge program* from a query and a mediator specification. Recall, the query refers to mediator objects. The QD&AO transforms the query into a logical datamerge program that refers to source objects only. More precisely, a logical datamerge program is a collection of rules whose tails refer to the source object structures and whose heads describe the object structure of the answer objects.

Before the QD&AO is invoked, the normalizer transforms queries and mediators into *normal form MSL*. Normal form MSL patterns always have three fields and certain constructs (like V:{<title 'a'>}) 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 (Q8) into the query (Q9).

```
<X tr V> :- <X tr V:{<title 'a'>}>@m          (Q8)
<X tr {<Void Vl Vv>}> :-                       (Q9)
    <X tr {<T2 title 'a'> <Void Vl Vv>}>@m
```

Then 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. 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.[3]

To illustrate consider the following mediator (MS10) that contains a single rule.

```
<trep(RN) tr {<O L X>}>@m1 :-          (MS10) (R10.1)
     <r {<rn RN> <O L X>}>@s1
```

Let us now consider the query (Q11) that retrieves the `tr` objects where the object-id is `trep('123')`.

```
<trep('123') tr V> :-                          (Q11)
     <trep('123') tr V>@m1
```

The match of this query and specification (MS10) results in the single unifier θ where

$$\theta = [(R10.1) : \mathtt{RN} \mapsto '\mathtt{123}', \mathtt{V} \mapsto \{< \mathtt{O}\ \mathtt{L}\ \mathtt{X} >\}]$$

θ 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 our unifiers from unifiers of first-order logic. The unifier also contains the name (R10.1) 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 (R10.1). When θ is applied to the query head `V` is substituted by `{<O L X>}`. Similarly, applying θ to the rule tail of (R10.1), we replace `RN` by `'123'`. Thus, we derive datamerge rule (DR1).

```
<trep('123') tr {<O L X>}> :-                  (DR1)
     <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, where each set pattern contains at most one subobject pattern, *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.)

[3] 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. 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.

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 $\{e_1^1\}$
  then the value field of e_2 is $\{e_2^1, \ldots, e_2^m\}$
    and there is a pattern $e_2^j, j = 0, \ldots, m$ such that
    e_1^1 is included in e_2^j.
  else e_1 and e_2 have the same value field
  or the value field of e_1 is $\{\}$ and
    the value field of e_2 is a set pattern

□

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 described later.

4.2 Resolution in the Presence of 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. We have also abbreviated some labels; this is just to have more compact patterns in this paper.

```
<trep(RN1) tr {<T1 title T>}>@m (MS12) (R12.1)
  :- <Ro1 r {<RNo1 rn RN1> <T1 title T>}>@s1
<trep(RN2) tr {<Poid postscript P>}>@m (R12.2)
  :- <Ro2 r {<RNo2 rn RN2>
            <Poid postscript P>}>@s2
```

Rules (R12.1) and (R12.2) 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 (R12.1) 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 (Q9) which asks for all the subobjects of the tr objects where the title is 'a'. Since the subobjects of the query may come from different rules, the normalizer rewrites query (Q9) as (Q13):

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

Now we can match the two patterns that appear in the (Q13) query tail to different rule heads. Suppose that we start by matching the first pattern of the tail, i.e., <X r {<T2 title 'a'>}>. It matches only with the head of (R12.1). This produces the unifier $\theta_1 = [(R12.1) : \mathtt{X} \mapsto \mathtt{trep(RN1)}, \mathtt{T1} \mapsto \mathtt{T2}, \mathtt{T} \mapsto \text{'a'}]$. Applying θ_1 to the query and the rule and replacing the query condition, we produce

```
<trep(RN1) tr {<Void Vl Vv>}> :-            (Q14)
   <Ro1 r {<RNo1 rn RN1> <T1 title 'a'>}>@s1
   & <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 (R12.2) thus producing the unifier $\theta_2 = [(R12.2) : \mathtt{RN2} \mapsto \mathtt{RN1}, \mathtt{Void} \mapsto \mathtt{Poid}, \mathtt{Vl} \mapsto \mathtt{postscript}, \mathtt{Vv} \mapsto \mathtt{P}]$. Second, <trep(RN1) r {<Void Vl Vv>}> matches with (R12.1). Since we have already used (R12.1) for matching the first condition of the query tail, we must not use (R12.1) again for matching the second condition. Thus, we introduce an instance of (R12.1) with renamed variables (see rule (R12.1.b) below) and we match <trep(RN1) r {<Void Vl Vv>}> against it, producing the unifier $\theta_3 = [(R12.1.b) : \mathtt{RNb} \mapsto \mathtt{RN1}, \mathtt{Void} \mapsto \mathtt{T1b}, \mathtt{Vl} \mapsto \mathtt{title}, \mathtt{Vv} \mapsto \mathtt{T1b}]$.

```
<trep(RNb) tr {<T1b title Tb>}>@m          (R12.1.b)
   :- <Ro1b r {<RNo1b rn RNb>
               <T1b title Tb>}>@s1
```

Finally, for each one of the two unifiers θ_2 and θ_3 we develop one datamerge rule, shown below in datamerge program (DP15). Rule (DR15.1) is obtained by replacing the m condition of (Q14) with the rule tail of (R12.2) and subsequently applying θ_2. Similarly, (DR15.2) is derived using the rule tail of (R12.1.b) and unifier θ_3.

```
<trep(RN1) tr                        (DP15) (DR15.1)
           {<Poid postscript P>}> :-
 <Ro1 r {<RNo1 rn RN1><T2 title 'a'>}>@s1 &
 <Ro2 r {<RNo2 rn RN1>
        <Poid postscript P>}>@s2
<trep(RN1) tr {<T1b title Tb>}> :-    (DR15.2)
 <Ro1 r {<RNo1 rn RN1> <T2 title 'a'>}>@s1 &
 <Ro1b r{<RNo1b rn RN1> <T1b title Tb>}>@s1
```

In this particular case one query condition matched with only 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 Section 5. The extended version [PGM] presents 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. First, QD&AO *eliminates* any datamerge rule that produces data that are subsumed by the data produced by another rule. Second, QD&AO *reuses* rules. That is, we may avoid issuing a query if all of its bindings for *necessary* variables are obtained by another rule. Note, not all variables are necessary for constructing the fused object. Only variables that appear in the rule head, or variables that join conditions in the tail, are *necessary*.

Due to space limitations we illustrate only *rule reuse* and not *rule elimination*. Consider datamerge rule (DR15.1). To evaluate it, we need to send a query to s1 to evaluate the condition <Ro1 r {<RNo1 rn RN1> <T2 title 'a'>}>@s1. This query only contains one necessary variable, RN1. Notice that all RN1 bindings in the above condition are also bindings of RN1 in rule (DR15.2). Hence, instead of accessing s1 twice, we can reuse the bindings retrieved for (DR15.2) by rewriting the datamerge program as follows. Note, (DR15.2.b) and (DR15.1.b) correspond to the rewritten versions of (DR15.2) and (DR15.1).

```
[<trep(RN1) tr {<T1b title Tb>}>      (DR15.2.b)
 bind1(RN1)] :-
  <Ro1 r {<RNo1 rn RN1> <T2 title 'a'>}>@s1
  & <Ro1 r {<RNo1b rn RN1><T1b title Tb>}>@s1
<trep(RN) tr {<Poid postscript P>}> (DR15.1.b)
 :- bind1(RN1)
```

```
    & <Ro2 r {<RNo2 rn RN1>
              <Poid postscript P>}>@s2
```

The notation [...] specifies multi-head rules. Thus, the data retrieved from the tail of (DR15.2.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 (DR15.1.b).

We can detect the applicability of subsumption using unifiers. In particular, a datamerge rule condition c can reuse a datamerge rule r if there is a unifier θ such that every condition c_i of the tail of r is included in c (after we apply θ) and every useful variable of c appears in the head of r.

Subsumption-based Optimizations in Plan Generation: Some subsumption-based optimizations are applied during or after physical plan generation. Thus, we start by briefly describing how physical plans are obtained. Then, in the remainder of the paper we discuss subsumption-based optimizations to the physical plans.

Let us consider mediator (MS16) (that also appeared in non-normal form as (MS2) in Section 3). (MS16) integrates documents without explicitly mentioning their non-key attributes.

```
<trep(RN1) tr {<O1 A1 X1>}>@all (MS16) (R16.1)
   :- <Ro1 r {<RNo1 rn RN1> <O1 A1 X1>}>@s1
<trep(RN2) tr {<O2 A2 X2>}>@all          (R16.2)
   :- <Ro2 r {<RNo2 rn RN2> <O2 A2 X2>}>@s2
```

Let us assume that query (Q17) is sent to mediator (MS16).

```
<X tr {<Void Vl Vv>}> :-                   (Q17)
   <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 (DP18) of Figure 3.

The cost based optimizer receives the logical 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 (DP18). (PDP19), in Figure 4, is one of the possible physical datamerge programs (from now on referred as physical programs) for (DP18).

The notation @(Q20,s1) in physical rule (PDR19.1) indicates that query (Q20) 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 (PDR19.1) then saves the retrieved reports and stores the RN bindings in bind1.

The => annotation in rule (PDR19.2) indicates that we perform a nested-loops join of bind1(RN) and

```
[<trep(RN1) tr {<O1 A1 X1>}>   (DP18) (DR18.1)
 bind1(RN1)] :-
  <Ro1 r {<RNo1 rn RN1> <Y year '95'>}>@s1
  & <Ro1b r {<RNo1b rn RN1> <O1 A1 X1>}>@s1
<trep(RN1) tr {<O2 A2 X2>}> :-        (DR18.2)
  bind1(RN1)
  & <Ro2 r {<RNo2 rn RN1> <O2 A2 X2>}>@s2
[<trep(RN2) tr {<O2 A2 X2>}>          (DR18.3)
  bind2(RN2) ] :-
    <Ro2 r {<RNo2 rn RN2> <O2 A2 X2>}>@s2
    & <Ro2b r {<RNo2b rn RN2>
               <Y year '95'>}>@s2
<trep(RN2) tr {<O1 A1 X1>}> :-        (DR18.4)
     bind2(RN2)
     & <Ro1 r {<RNo1 rn RN2> <O1 A1 X1>}>@s1
```

Figure 3: Datamerge Program

<trep(RN) L V>@(Q21,s2). That is, for every binding r of RN in bind1, we instantiate a parameterized query (Q21), by replacing RN with r, and we send the instantiated query to s2. Similarly, the local annotation that appears in (PDR19.4) indicates that we perform a local join of bind2(RN) with <trep(RN) L V>@(Q23,s1). The join policy decision is made by estimating the cost of each option using information about the sources. We will not deal in this paper with these cost-based optimization problems.

Applying Query Subsumption on the Plan: Earlier 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 (Q20) is subsumed by query (Q23) because (Q23) retrieves all the reports of s1 whereas (Q20) retrieves only the reports with year '95'. Furthermore, once we have the result of (Q23) we may locally apply the condition on year and hence compute the result of (Q20). The optimizer captures this relationship between (Q20) and (Q23), eliminates (Q20), and modifies rule (PDR19.1) to use the subsuming query (Q23). Note the condition on year that is applied on the result of (Q23).

```
[<trep(RN) L V>, bind1(RN)] :-     (PDR19.1.b)
      <trep(RN) L {<Y year '95'>}>@(Q23,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.

```
[<trep(RN) L V>, bind1(RN)] (PDP19) (PDR19.1)
   :- <trep(RN) L V>@(Q20,s1)
<trep(RN) L V> :-                     (PDR19.2)
   bind1(RN)
   &(=>) <trep(RN) L V>@(Q21,s2)
[<trep(RN) L V>, bind2(RN)] :-        (PDR19.3)
   <trep(RN) L V>@(Q22,s2)
<trep(RN) L V> :-                     (PDR19.4)
   bind2(RN)
   &(local) <trep(RN) L V>@(Q23,s1)
<trep(RN) tr {<O1 A1 X1>}> :-             (Q20)
   <r {<rn RN> <Y year '95>}>@s1
   & <r {<rn RN> <O1 A1 X1>}>@s1
<trep(RN) tr {<O2 A2 X2>}> :-             (Q21)
   <r {<rn $RN> <O2 A2 X2>}>@s2
<trep(RN) tr {<O2 A2 X2>}> :-             (Q22)
   <r {<rn RN> <Y year '95'>}>@s2
   & <r {<rn RN> <O2 A2 X2>}>@s2
<trep(RN) tr {<O1 A1 X1>}> :-             (Q23)
   <r {<rn RN> <O1 A1 X1>}>@s1
```

Figure 4: Physical Datamerge Programs

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. The subsumption optimizations of QD&AO 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 in QD&AO as many subsumption optimizations as possible, then generate plans, and finally perform the remaining subsumption optimizations.

5 Advanced Optimization Techniques

In this section we describe further optimizations that are performed by the algebraic optimizer and the plan generator. The first two techniques (Section 5.1) are useful for reducing the exponential number of datamerge rules that can be generated when there is incomplete information about the sources. The third technique (Section 5.3) deals with negation and blocking operations.

5.1 Reducing the number of datamerge rules

Mediator (MS16) and query (Q17) illustrated the problem of exploding conditions. Since the label `year` could come either from `s1` or `s2`, the condition on `year` had to be pushed to both `s1` and `s2`. One of the nice features of MSL is that it does not *force* us to specify where data might come from, but this flexibility then forces the system to explore all possible options.

In the general case we may have a query requiring fused objects satisfying a conjunction of conditions $c_1, ..., c_n$, where the object components come from sources $s_1, \ldots, s_m$. If the specification does not state what conditions should be pushed to what sources, resolution leads to an exponential number of datamerge rules, corresponding to all possible ways of splitting $c_1, ..., c_n$ between the sources. The subsumption techniques of Section 4 can help reduce the amount of work, but we may still have too many queries for the sources.

One way to reduce the number of queries is to determine in advance those that will return an empty answer because no source objects could ever match. For instance, in our example, if we know that `year` information may not come from `s1` then we can remove rules (DR18.1) and (DR18.2) from (DP18) since they both require that `year` is found at `s1`.

Even though the mediator does not have a "schema" dictating what type of information a source may have, it could achieve the same effect by *asking at run time* if source `s1` has any objects with `year` label. If no such objects exist at `s1`, the mediator can 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; [PGM] presents some.

5.2 Refraining from Simultaneously Pushing Conditions

We now briefly consider a second technique for reducing the number of datamerge rules. The key observation is that pushing *all* conditions to the sources may not be (and most often is not) the best plan. At first sight this appears counter-intuitive because in conventional query optimization it is always beneficial to evaluate selection conditions as early as possible. However, the absence of complete knowledge about the structure of the sources prompts us to try all possible ways of splitting the conditions among the sources, hence producing a potentially inefficient plan.

Let us present an alternative for processing a query with n conditions to m sources without pushing all conditions. We can answer the query by (i) fetching from each source all objects satisfying any one of the c_i conditions and (ii) fusing these objects at the mediator, and (iii) selecting those fused objects that satisfy all the c_i conditions. For example, suppose that in a query like (Q17) we look both for reports with `year` equal to 95 and `topic` equal to "databases." Our original policy, that we call conjunctive, would involve datamerge rules that split the two conditions among the sources in all possible ways. Instead, we send a single query to each source asking for all documents that either have `year` 95 or `topic` databases. After constructing the fused objects, we filter out those that do not meet both conditions. This policy, which we call disjunctive (we are sending disjunction of atomic conditions to the sources), beats the original conjunctive policy when the number of sources is larger than the number of conditions.

In [PGM] we present more alternatives. Currently we are working on the evaluation of these alternatives and the implementation of an optimizer that efficiently chooses among these alternatives.

5.3 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 (MS4) 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 (PDR24). The important point is that we evaluate the `NOT provides('123')` condition of (PDR24) 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`.

```
<trep(RN) tr {<O2 A2 X2>}> :-          (PDR24)
     NOT provides('123') &
     <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 (Q9) that requests reports with title `'a'`. The best strategy here depends on the expected number of matching reports at each site. For instance, assume that the number of `'a'` reports retrieved from `s1` is not large. To be specific, say that only reports `'123'` and `'136` have title `'a'`. In this case the best strategy is probably to send to `s2` query (Q25) 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`.)

```
<trep(RN) tr {<O2 A2 X2>q}> :-         (Q25)
   <Ro2 r {<RNo2 rn RN> <O2 A2 X2>}>@s2
   & NOT RN='123' AND & RN='136'
```

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 `'a'` 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`.

6 Discussion and Related Work

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 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. 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.

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, A^+91, C^+95, S^+, HM93, TRV95, K^+93].) Most of them base fusion on a precise description of the schemas exported by the sources and present classifications and resolution techniques for the various schematic and semantic conflicts that may be found in the schemas [K^+93]. Unlike these approaches, in the present paper, we assume minimal knowledge of the structure and contents of the sources.

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 [Ull89]). 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]. Automatic creation and manipulation of object-id's based on Skolem functors are considered in depth in [HY90]. [LSS93, KLK91] propose logics and languages 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.

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.

Implementation Status: 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 5.3. 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. At the same time we are evaluating the performance of the proposed plans.

References

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

[C+95] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. RIDE-DOM Workshop*, pages 124–31, 1995.

[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.

[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.

[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.

[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 database 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.

[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.

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

[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.

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