Schema Mapping as Query Discovery

Renée J. Miller*
University of Toronto
miller@cs.toronto.edu

Laura M. Haas Mauricio A. Hernández
IBM Almaden Research Center
{laura,mauricio}@almaden.ibm.com

Abstract

To enable modern data intensive applications including data warehousing, global information systems and electronic commerce, we must solve the *schema mapping* problem in which a source (legacy) database is mapped into a different, but fixed, target schema. Schema mapping involves the discovery of a query or set of queries that transform the source data into the new structure. We introduce an interactive mapping creation paradigm based on *value correspondences* that show how a value of a target attribute can be created from a set of values of source attributes. We describe the use of the value correspondence framework in *Clio*, a prototype tool for semi-automated schema mapping, and present an algorithm for query derivation from an evolving set of value correspondences.

1 Introduction

Many modern applications such as data warehousing, global information systems and electronic commerce need to take existing data with a particular structure or schema, and re-use it in a different form. These applications start with an understanding of how data will be used and viewed. That is, they start by determining a target schema. They then must create mappings between this target and the schemas of the underlying data sources. Creating those mappings is today a largely manual (and extremely difficult) process. Transformation of the data is accomplished by complex programs, hand-written or pieced together by specialized tools (*e.g.*, for data warehouses), and these programs must then be carefully tuned to get reasonable performance. While the time required to generate and optimize these programs may be justified for data warehouses, it is unacceptable for e-commerce, where applications must evolve much more quickly, and it is awkward for applications which require direct access to source data (such as global information systems and e-commerce).

We show how the transformation process can be simplified and made more efficient and flexible by using database management systems as transformation engines. Data independent transformations, specified in SQL, can then be automatically optimized and parallelized by the DBMS for better performance. As many DBMS today can process queries over data they do not manage ([IBM97, CHS+95, Ora]), the DBMS can effectively handle the inter-source scheduling and data movement needed for transformations as well. Creating mappings becomes a process of query discovery: finding the queries or views that correctly transform the data to the desired schema.

By simplifying the task of mapping creation, we make it possible for DBMS to play a broader role in new applications, not merely as a provider of data, but as a manager of the transformations themselves. Modern DBMS are not only data management tools, they are query management tools. They incorporate a wealth of sophisticated knowledge about queries and query manipulation. While this knowledge has been targeted to the problem of query optimization to produce efficient execution plans, we show how the same infrastructure and similar reasoning can be applied to the problem of query discovery for integrating and transforming data. For both tasks, we are reasoning about the relationships between and equivalences of queries and schemas.

Clio [HMN+99] is a research prototype of a schema mapping creation tool. Clio produces view definitions that allow applications to directly access source data using a middleware query engine. Queries posed on

*Supported by an IBM University Partnership Grant and the Presidential Early Career Award for Scientists and Engineers (PECASE) under NSF Award # 9702974.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

these views can be optimized normally by the query engine, so that only the data needed for a particular query is converted. Clio produces the SQL queries for the user, providing users with data samples and other feedback to allow them to understand the mappings produced.

In Section 2, we present a framework for mapping creation based on the notion of *value correspondences*. Value correspondences are an intuitive way of recording the relationships between source and target schemas. Given a set of value correspondences, we show how to compute the query or queries needed to perform the implied transformations (Section 3). Section 4 illustrates the use of our mapping algorithm for a data warehouse. We briefly discuss related work in Section 5 and conclude in Section 6.

2 A Framework for Query Discovery

The focus in schema mapping is on query discovery. By contrast, classical schema integration (including both view integration and database integration) is the activity of integrating a set of schemas into a unified representation [RR99]. Schema integration techniques typically distinguish two key tasks: creation of the integrated schema and creation of queries (mappings) between schemas. In the applications we consider, the target schema does not depend for its definition on the identity and structure of the sources. Hence, the problem of creating the integrated schema is no longer relevant. However, the need to create mappings between the source and the integration remains. Yet the problem of mapping generation between an integrated schema and the source schemas used to derive the integration is inherently different from that of deriving mappings between independently created schemas. In the former problem, the mapping is implicit to the derivation process. Indeed in their comprehensive schema integration survey, Ram and Ramesh devote only a single paragraph to mapping generation [RR99]. This is not an oversight on their part, but rather a true reflection of the methodologies they survey.

As with schema integration, the schema mapping task cannot be fully automated since the syntactic representation of schemas and data do not completely convey the semantics of different databases. For example, it is not possible to know with complete certainty from the schema and data alone whether the `Emp` relation in one schema has the same meaning as the `Employee` relation in another. As a result, for both schema mapping and schema integration, we must rely on an outside source to provide some information about how different schemas (and data) correspond.

However, the different nature and goals of these two tasks necessitate the use of different types of correspondences. For the schema integration, which is predominantly a schema design problem, design level assertions detailing how schema constructs relate are appropriate [RR99]. These assertions state how the **set** of values of a construct in one source schema relate to the **set** of values of a construct in another source schema. For the mapping problem, we claim that a different type of assertion is both more informative and easier to elicit from a user. We call this new type of assertion a *value correspondence*.

2.1 Overview of Value Correspondences

Informally, a value correspondence is a pair, consisting of (1) a function defining how a value (or combination of values) from a source database can be used to form a value in the target, and (2) a filter, indicating which source values should be used. For example, a string concatenation function can be used to indicate that a value of the staff-id attribute of the target schema is formed by concatenating the letter 'E' to an employee number from the source, along with a filter that selects only active employees. Similarly, a value of the appellation attribute may be formed by concatenating together a title and name value from the source. There might be a filter on title, or any other attribute(s), or the filter might be "True". From these examples, it should be clear that schema assertions and value correspondences are related. An attribute assertion that an Attribute A is a subset of Attribute D may imply the use of the identity function and some filter as a value correspondence to map values of A to values of D [RR99]. However, the main focus of schema assertions is on specifying how the values of one attribute (or other schema constructs) **as a set** relate with the set of values of another attribute. It is this set relationship that drives the integration algorithms [RR99].

In contrast, in Clio, the value correspondences drive the integration. This distinction is important for two reasons. First, we argue that it is natural for a DBA to be able to specify value correspondences indicating the form in which a source value should appear in the target. Even DBAs with incomplete knowledge of the schema can specify the correspondences for those values they understand. To be accurate, the set relationships of attribute assertions require a more complete knowledge of the schema and relationships between components of the schema. Inaccurate or imprecise assertions (for example, asserting that two attributes overlap when there is actually a set containment relationship) will lead to incorrect integrations. Second, the knowledge provided by these two different types of statements is very different. This difference gives rise to a new approach to reasoning about and creating schema assertions that has not previously been explored. Specifically, we propose an iterative *integration-by-example* paradigm under which a DBA specifies how example values are mapped and the tool attempts to deduce a likely schema mapping. In the process, the DBA may be prompted for information relevant to choosing between alternative mappings.

This information may sometimes include information about the set relationships (but only if this information is necessary for disambiguating between different mappings).

Note that we are not arguing that the information provided by schema assertions is irrelevant. On the contrary, we are arguing it may not be required to deduce all mappings and that it may be impossible for a DBA to specify *a priori* without having seen even a partial or potential mapping. Furthermore, we do not use schema level assertions to drive the mapping derivation process. Rather, we make use of reasoning about schemas (and queries) and about possible alternative schemas (and queries) to drive this process.

Our thesis is two-fold.

- **Value correspondences are an appropriate abstraction for eliciting information from the user or DBA.** A DBA may easily be able to indicate that distance values are formed by multiplying rate times time. However, (s)he may not readily be able to specify the possibly complex query required to indicate how a specific rate value is paired with a specific time value (perhaps through a complex query involving many relations) without some help or prompting from the mapping tool.

- **Using reasoning about queries and query containment, we can effectively and efficiently help the user derive correct schema mappings.** Specifically, we will employ the same reasoning about queries (and alternative queries) already used in DBMS to do query optimization and semantic query optimization. Traditionally, this knowledge is buried deep within the optimizer and highly tuned to the problem of finding a low cost query plan. To our knowledge, this is the first principled attempt to expose this sophisticated reasoning about queries to a user to help in the schema mapping task.

Value correspondences may be entered by a user or may be suggested using linguistic techniques applied to the data and meta-data such as the names of schema components [BHP94, Joh97]. In Clio, we use a graphical interface that facilitates schema and data browsing to elicit value correspondences from users [HMN+99]. Other data-centric interfaces, including the scalable spreadsheet paradigm proposed by Raman, Chou and Hellerstein [RCH99], would also be appropriate for eliciting the correspondences that drive our algorithms.

2.2 Constructing Schema Mappings

We now turn to the question of constructing a schema mapping from a set of value correspondences. The construction process is one of searching for the most

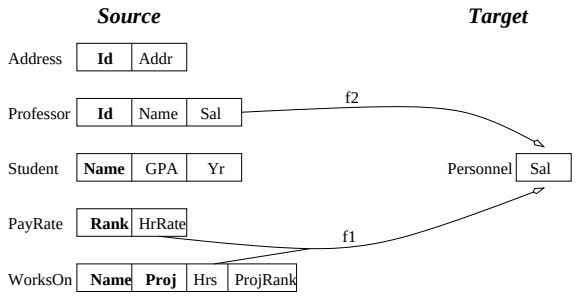

Figure 1: Example schemas to be mapped.

reasonable mapping based on the properties of the correspondences, the properties of the schemas, and the schema or structuring cues that lie buried in the data. We begin with an example that explains intuitively the type of reasoning we will employ.

Example 2.1 *Consider the two schemas of Figure 1. Suppose a user has indicated that the product of the values in the PayRate(HrRate) and WorksOn(Hrs) attributes should also appear in Personnel(Sal). This value correspondence is represented by the function f_1. For this example, we will assume all filters are "True".*

f_1 : *PayRate(HrRate)*WorksOn(Hrs)* $\rightarrow$ *Personnel(Sal)*

This correspondence indicates how two values from the source can be combined into a target attribute. However, it does not indicate which values should be combined. Intuitively, if HrRate and Hrs belonged to the same relation, then the most likely interpretation of the correspondence is to combine values from the same tuple. However, in general, particularly when HrRate and Hrs belong to different relations, we must define a query that produces pairs of values to be combined.

In this example, to produce a schema mapping we must determine a way of associating a specific tuple of PayRate with a tuple of WorksOn. If ProjRank is a foreign key of PayRate, then the natural way of doing this is through a join on Rank = ProjRank. This produces the following mapping.

```
q1: SELECT  P.HrRate*W.Hrs
    FROM    PayRate P, WorksOn W
    WHERE   P.Rank = W.ProjRank
```

However, suppose this foreign key is not declared but instead WorksOn.Name is declared as a foreign key of Student and Student.Yr is declared as a foreign key of PayRate. (That is, there is a different HrRate value for Sophomores than for Juniors, etc.*) Then the foreign key path* $WorksOn \bowtie Student \bowtie PayRate$ *would be a better join path to use in the schema mapping.*

```
q1': SELECT  P.HrRate * W.Hrs
     FROM    PayRate P, WorksOn W, Student S
     WHERE   W.Name = S.Name AND S.Yr = P.Rank
```

Note that if, in fact, ProjRank is also declared as a foreign key of PayRate, it is then not clear which join path is better. In some circumstances, the filter of the

value correspondence may provide a clue. For example, if our filter were "Student.Yr > 2", the join through Student would make more sense. In the absence of such clues, user input is required. A tool such as Clio can still help, however, by enumerating the options and providing "samples" (that is, instances of the target schema) that are the results of different mappings.

Implicit to the process of deriving the mapping is our intuition that for each HrRate value, there is somewhere in the source database **a** value for the Hrs attribute that can be used to derive **a** value of the Sal attribute in the target. It is certainly possible that a user wished to take the cross product of HrRate and Hrs and form salaries from every pair of these source values. However, this possibility is unlikely, particularly if there is a natural way to pair HrRates with specific Hrs values. So Clio makes use of reasoning about schemas and the semantics conveyed by constraints, such as foreign keys, to deduce likely mappings.

Example 2.2 *Continuing this example, suppose that the user has provided a second value correspondence indicating that values of the Professor(Sal) attribute should appear in Personnel(Sal) in the target.*

$$f_2 : Professor(Sal) \rightarrow Personnel(Sal)$$

Certainly, one interpretation of these correspondences is that we should take the join of salary values produced by f_1 and those produced by f_2 to populate the target. However, this is not the most intuitive mapping since it would mean that many (or perhaps even most) of the source values for salary would not appear in the target. Rather, it is more likely that the user intended the mapping to be a union of these values. The salary for personnel may be derived either from professor salaries or from student pay rates and hours. That is, a better mapping would be the following.

```
q2: SELECT  P.HrRate * W.Hrs
    FROM    PayRate P, WorksOn W, Student S
    WHERE   W.Name = S.Name AND S.Yr = P.Rank
    UNION ALL
    SELECT  Sal
    FROM    Professor
```

While these examples may seem heuristic, there is some principled reasoning going on under the covers. To guide the mapping construction, we are following two key principles. First, if possible, all values in the source appear in the target. This principle guided our decision to use a union rather than a join in the example when two different value correspondences were given for the same attribute. Second, if possible, a value from the source should only contribute once to the target. In other words, associations between values that exist in the source should not be lost. This principle guided our choice to use a join rather than the cross product to compute a salary value using the correspondence f_1.

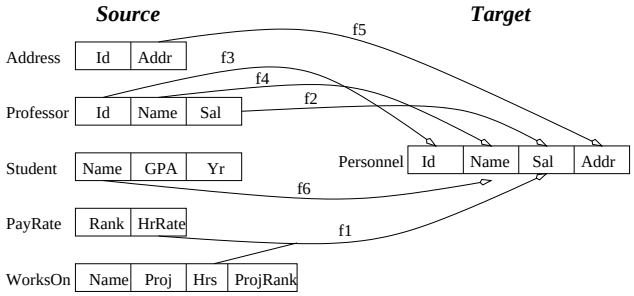

Figure 2: Value Correspondences.

Note that these principles are restatements of common data design principles such as "one fact in one place" [Dat95]. Even in the presence of filters, we try to uphold these principles for those values selected by the filter. Since our goal is schema mapping rather than schema design, we do permit a user to override these principles. For example, in publishing information for a "What-If" scenario, a user might want a cross-product so that (s)he could evaluate all possibilities.

We use these principles to derive an initial mapping, one that preserves, to the extent possible, the information in the source. A user may examine target data derived under this mapping and decide whether to restrict or modify the mapping.

Example 2.3 *To complete our running example, consider the extended schemas of Figure 2 and the following additional value correspondences.*

$$\begin{aligned} f_3&: Professor(Id) \rightarrow Personnel(Id) \\ f_4&: Professor(Name) \rightarrow Personnel(Name) \\ f_5&: Address(Addr) \rightarrow Personnel(Addr) \\ f_6&: Student(Name) \rightarrow Personnel(Name) \end{aligned}$$

Intuitively, these correspondences divide naturally into two groups that coincide with the two different ways in which a Personnel tuple can be created. The first group includes the correspondences from Professor and Address, namely f_2, f_3, f_4, f_5. A Personnel tuple can be created by joining together a Professor tuple and an Address tuple. Such a mapping is suggested by the presence of foreign key constraints between these relations or by the presence of a source query workload that includes a join of these two relations or even by the data itself (if the Id values in the two relations overlap). Depending on the constraint information, we may choose an outer-join, rather than a join to avoid losing information (but we use a join here, to keep our example simple). The second group includes the correspondences from Student, PayRate and WorksOn, namely f_1 and f_6. A Personnel tuple can be created by joining together Student, PayRate and WorksOn. Hence, the most reasonable schema mapping given these specific constraints in the source is the following.

```
s1: SELECT P.Id, P.Name, P.Sal, A.Addr
    FROM   Professor P, Address A
    WHERE  A.Id = P.Id
    UNION ALL
    SELECT NULL as Id, S.Name, P.HrRate*W.Hrs,
           NULL as Addr
    FROM   Student S, PayRate P, WorksOn W
    WHERE  S.name = W.name AND S.Yr = P.Rank
```

Notice that this is not the only possible mapping. Another option would be to take the outer-union of all the relations in the source and project out attributes from the source that do not participate in any value correspondence. The outer-union is the union where any missing attributes are set to null.

```
s1': SELECT NULL as Id, NULL as Name, NULL as Sal, Addr
     FROM   Address A
     UNION ALL
     SELECT P.Id, P.Name, P.Sal, NULL as Addr
     FROM   Professor P
     UNION ALL
     SELECT NULL as Id, Name, NULL as Sal, NULL as Addr
     FROM   Student S ...
```

While possible, it is clear from our understanding of the semantics of the source schema that this would not be a particularly natural mapping. Such a mapping loses associations between data values present in the source. For example, in the source, we can determine the address of a professor (assuming the Id of a professor appears in the Address relation). This would not be true in the target using a mapping based on outer-unions.

In the example, we described the intuition behind the mapping derivation. This intuition, while seeming natural to anyone who has worked with databases, actually has a formal basis that dates back to early theoretical work on database design. Simply put, our goal is to find mappings that do not lose information, or at least lose as little information as possible. In reasoning about mappings, we will consider alternative ways of combining value correspondences to produce mappings. We use formal reasoning about schemas to choose among these alternatives. Due to the vagaries of semantics, we will not always be right. But our goal is not to fully automate this process. Rather, our goal is to present users with a reasonable mapping as a starting point which can be refined. By providing an example mapping, the user can see target values produced by this mapping and identify data values that are missing (or have been included in error). Hence, the mapping refinement process is data- or value-driven. The user does not have to edit SQL to refine the mapping.

We have used a very simple example and enumerated only a few of the possible mappings that would need to be considered. We have not had space to overview the complexities introduced by considering aggregations, groupings or even outer-joins, all of which are important constructs for integrating information. In Example 2.3, if no foreign keys had been specified, we would need to use outer-joins rather than joins to avoid losing information. Because outer-joins are not associative, the differences between alternative outer-join orders can be subtle yet these differences are extremely important in obtaining a semantically correct mapping. There is a considerable literature on these subtleties alone [GL94, RU96, GLR97]. Given this inherent complexity, a systematic search through the large search space of alternative mappings is a job best done by a tool that can eliminate unlikely mappings and identify correct mappings a user might not otherwise have considered.

2.3 Search Space

Having defined the mapping discovery problem as a search through a set of alternative mappings, an important characteristic of the approach is the set of possible mappings considered. We begin by considering mappings that preserve information capacity dominance or equivalence [Hul86]. Such mappings are important in information integration [MIR93]. We are able to take advantage of a solid literature enumerating such mappings [MIR94, RU96, RR94] and providing search procedures for finding such mappings [AH88, MIR94]. From this foundation, we extend the search space in two ways. First, we consider a larger class of mappings, including queries for which the equivalence problem is not decidable. As a result, our algorithm is not complete in that it may not consider all possible mappings. However, this extension is required to consider mappings between schemas with constraints or dependencies. Second, we consider non-equivalence (or dominance) preserving mappings. This extension is a necessity since in practice, the source and target schemas will not represent the same information. To keep our search problem tractable, we attempt to find mappings that minimize the information loss. A formal description of the search space is beyond the scope of this paper. Informally, the mappings we consider can be broadly classified into two groups.

Vertical Compositions Facts or tuples can be combined using the join operator. To avoid having a tuple combine or join with multiple tuples (that is, to avoid having a single tuple contribute multiple times to the result), we favor performing joins where there is a functional (N:1) relationship between the tuples. Dependency theory tells us this can be accomplished using joins across foreign keys. (Indeed this same intuition motivates the relational normal forms.) In addition, to minimize information loss, we use outer-joins unless the constraints in the mapping imply that the outer and inner-joins would be equivalent or unless we can determine that the tuples that could be lost by using a join are included elsewhere in the mapping. Obviously, we will not always be able to determine

this since this problem is undecidable for the general constraints we consider. In composing outer-joins, we favor full disjunctions to ensure all information for a single fact is collected in a single tuple [RU96, GLR97]. Note that using an outer-join over a foreign key, we have a mapping that corresponds to the composition transformation of [MIR94]. Such a transformation preserves information (that is, information capacity dominance) in the sense of [Hul86]. We also have an algorithm for determining if such a mapping exists between two schemas and for finding such mappings [MIR94].

Horizontal Compositions Facts or tuples can also be combined using set operators. When we have multiple value correspondences to the same value in the target, we begin by using union to combine the values. To accomplish our information preservation principles, we favor using (multi-set) unions as a starting point, over other set operations such as intersections. If we can determine the sets being unioned are disjoint, a regular (set) union is used. For example, meta-data is often used to create a tag to distinguish a tuple coming from one place in the schema from a tuple coming from a different location. Indeed, the mappings resulting from schematic (or meta-data) heterogeneity between the source and target schemas can often be represented using tagged unions [Mil98].

This framework is an extensible one. Additional classes of mappings and additional heuristics for selecting between mappings can easily be integrated. Furthermore, this framework and the algorithms described below can be used both for traditional applications where the target schema is a (virtual) view specified over one or more (materialized) data sources and for applications where the source schemas are considered to be (materialized) views defined on a (virtual) target schema [LMSS95]. The former is sometimes referred to as the global-as-view approach in the literature and the latter as the local-as-view approach. In the former case, the mapping is a query or set of queries on the source schema that creates an instance of the target. In the latter case, the mapping is a query on the target. A more detailed discussion of this issue and its implications for the mapping can be found in the full version of this paper [MHH00].

3 Query Discovery Algorithm

We now present our mapping construction algorithm. To keep the notation simple, we assume the source and target schemas are representated in the relational model. We discuss generalizations to other models, including semi-structured models such as XML in Section 3.4.

3.1 Notation

Before presenting our algorithm, we outline the notation we will be using.

- Let $S_1, ..., S_n$ represent the n source relations.
- Let $T_1, ..., T_m$ represent the m target relations.
- We use the (possibly subscripted) symbol A to denote source attributes. The domain of an attribute A is denoted $dom(A)$.
- We use the (possibly subscripted) symbol B to denote target attributes.

Each attribute of the source will have associated meta-data. The meta-data includes the attribute name, the relation name, the schema name, the database name, the domain name, statistics such as high and low values of the attribute, and possibly additional annotations provided by a DBA. Hence, the meta-data is extensible. For an attribute A, $\mu(A)$ denotes the meta-data associated with A. Formally, $\mu(A)$ is a tuple $(\mu_1(A), \mu_2(A), ..., \mu_m(A))$ of values. For convenience, we give names to some of these values. The attribute name is denoted $attrname(A)$ and the relation name is denoted $relname(A)$.

We will represent a *value correspondence* as a tuple $v_i = \langle f_i, p_i \rangle$, where f_i is the correspondence function denoting the value substitution and p_i a filter.

When defining a correspondence function f_i, the DBA selects a number of source attributes (and, possibly, meta-data associated with those attributes) and **one** target attribute. Let $Attrs(f_i) = \{A_1, \ldots, A_q\}$ be the set of all source attributes used in f_i, and $TargetAttr(f_i) = B$ be the (one) target attribute. The correspondence function, f_i, can be expressed as follows.

$$f_i : dom(A_1) \times \ldots \times dom(A_q) \times \mu(A_1) \times \ldots \times \mu(A_q) \rightarrow dom(B)$$

Example 3.1 *The following correspondence indicates that values of the Distance attribute of the target can be formed by multiplying the Rate value by the Time value and dividing by 1.6 to convert kilometers to miles.*

$$f_1 : Rate * Time/1.6 \rightarrow Distance$$

Example 3.2 *The next correspondence indicates that company codes are formed by concatenating the ticker code with the relation name (the name of the stock exchange).*

$$f_2 : concat(relname(Ticker), Ticker) \rightarrow CompanyCode$$

Each value correspondence function f_i has an associated filter p_i that determines which subset of values from the source relations will be used by f_i. If we define $Attrs(p_i) = \{A_1, \ldots, A_r\}$ to be the set of all source attributes used in p_i, we can express p_i as follows.

$$p_i : dom(A_1) \times \ldots \times dom(A_r) \times \mu(A_1) \times \ldots \times \mu(A_r) \rightarrow boolean$$

By default, p_i is the predicate $True$, indicating the value correspondence is defined for all values in the domain. Note that $Attrs(p_i)$ is not necessarily the

same as $Attrs(f_i)$. In the first example above, we could define a p_1 : $Rate \leq 100$ which indicates that the correspondence only holds for small rate values. In the second example, we could have p_2 : $Exchange(Country)$ = "Canada" which would indicate that the correspondence only holds for stocks listed on Canadian exchanges. Here, even though the values involved in the correspondence come from the data and meta-data of a single relation (Ticker), the attributes of the correspondence will also include Exchange(Country). As described below, the algorithm will determine a join path between Exchange and Ticker (for example, $relname(Ticker) \bowtie Exchange(Name)$) to use when applying the filter.

Either the correspondence function or the filter may include aggregate functions. The aggregate is taken as a cue to perform a grouping in the schema mapping. To determine the grouping attributes, we must consider all the value correspondences for a target relation as described in the next section.

3.2 The Core Algorithm

For each target relation T_k we want to construct a query q_k that specifies what values to include in the relation. To do this, we consider the value correspondences $\mathcal{V}_k$ defining attribute values of T_k (*i.e.*, $\mathcal{V}_k = \{v_i = \langle f_i, p_i \rangle \mid TargetAttr(f_i) \in T_k\}$).

The idea behind this algorithm is to divide the set of value correspondences $\mathcal{V}_k$ into subsets of $\mathcal{V}_k$, each of which determines one way of computing the values of T_k. Each of these *candidate sets* can be mapped into a single *candidate SQL query* (that is, a query with a single `select-from-where-group-by` clause). The query q_k is then the horizontal composition (*i.e.*, the application of set operations such as UNION ALL) of these candidate queries.

We present the algorithm for a single target relation T and, thus, $\mathcal{V} = \mathcal{V}_k$ includes all value correspondences. When more that one target relation exists, we repeat the algorithm for each $\mathcal{V}_k$ possibly reusing computations from previous targets.

We divide the algorithm's tasks into four phases (see Figure 3). In the first phase, the value correspondences in $\mathcal{V}$ are partitioned into sets $\{c_1, \ldots, c_p\}$ that contain *at most one* correspondence per attribute of T. We call each such set a *potential candidate set*. In essence, each c_j represents one possible way of mapping the attributes of T. A potential candidate set is *complete* if it includes a value correspondence for every attribute in the target. Potential candidate sets are not necessarily disjoint since the same value mapping can appear in multiple potential candidate sets.

For clarity of exposition, we describe this phase of the algorithm as searching every potential candidate set derived from $\mathcal{V}$ independently (though the computations can be reused across subsets). Although this implies a large search space, potential candidate sets are generated on demand from the next phase of the algorithm (*i.e.*, pipelined). The order in which potential sets are passed to the next phase is, thus, important. As a heuristic, we give preference to complete potential sets whose value correspondences use the smallest set of source relations. Also, if a particular potential candidate set c_j is selected for use in the schema mapping, we can heuristically prune potential candidates that are proper subsets of c_j since they are unlikely to also appear in the mapping.

Example 3.3 *Consider the following value correspondences (and assume some filter p_i has been defined for each).*
$f_1 : S_1.A \rightarrow T.C$ $\quad f_2 : S_2.A \rightarrow T.D$ $\quad f_3 : S_2.B \rightarrow T.C$
The collection of complete potential candidate sets is $\mathcal{P} = \{\{v_1, v_2\}, \{v_2, v_3\}\}$. *The singleton sets* $\{v_1\}$, $\{v_2\}$, $\{v_3\}$ *are also potential candidate sets.*

It is important to note that we consider potential candidate sets that are not complete. There are two reasons for this. First, as shown in Example 2.3, in the final query mapping, there may not be a value correspondence for every target attribute. Second, we will be using our algorithm incrementally on perhaps incomplete sets of correspondences. We want to permit a user to specify a partial set of correspondences, and have Clio derive a possible mapping. Using the mapping, example tuples in the target can be derived. These tuples can be used by a user to understand how the data is fitting into the target.

The result of the first phase of the algorithm is a collection $\mathcal{P} = \{c_1, \ldots, c_q\}$, where each $c_j \subseteq \mathcal{V}$ represents a different possible way of mapping the attributes in the target relation T.

In the second phase of the algorithm, we prune from the set of potential candidate sets those sets that cannot be mapped into a good query. In particular, if the value correspondences in the potential candidate set map values from several source relations, we need to find a vertical composition (*i.e.*, a way of joining the tuples) of those relations. This composition will satisfy the criteria established in Section 2.3. We search for foreign key paths between these relations.[1] Often there will be at most one such path. If, however, there are multiple paths, we favor the path for which the estimated difference in size of the outer and inner join is the smallest.[2] This heuristic favors (outer-)join paths that produce the fewest dangling tuples. For any ambiguities that remain, we ask the user to choose one of the available join paths. To help in this process, we show the user example target tuples produced by

[1] Actually, the search is done only once for all potential candidates and the results of the search reused over different potential candidates.

[2] Note that this measure can be evaluated using common meta-data such as the number of distinct values of an attribute.

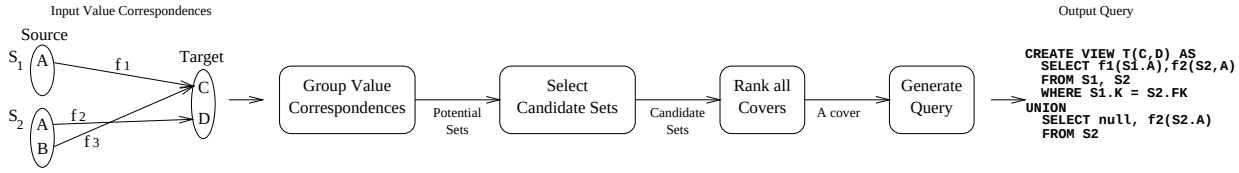

Figure 3: Mapping Algorithm

each of the alternative paths. In the absence of foreign key paths, we could employ data mining techniques to determine if there is an (approximate) foreign key relationship between the relations in question [Bel97, KMRS92] or permit the user to suggest appropriate join paths. If no acceptable join path can be found, the potential candidate set is removed from further consideration. Any potential candidate set that survives this pruning is a *candidate set*.

The result of this second phase is a set $\mathcal{G} \subseteq \mathcal{P}$ of candidate sets. Value correspondences in a candidate set either map attributes from only one source relation, or map attributes from multiple source relations and a join path among those relations is known.

In the third phase of the algorithm, we attempt to find a subset Γ of the candidate sets ($\Gamma \subseteq \mathcal{G}$) that covers all value correspondences in $\mathcal{V}$ (that is, every value correspondence in $\mathcal{V}$ appears at least once in Γ). We permit correspondences to participate in multiple candidates within a cover, but we do not consider a set of candidates Γ if we can remove a candidate set and still have a cover. For instance, in Example 3.3, $\mathcal{G} = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1\}, \{v_2\}, \{v_3\}\}$. Possible covers include $\Gamma_1 = \{\{v_1\}, \{v_2, v_3\}\}$ and $\Gamma_2 = \{\{v_1, v_2\}, \{v_2, v_3\}\}$ since all defined value correspondences appear at least once.

If there is more than one cover, Clio ranks them in reverse order of the number of candidate sets in the cover. Since the number of candidates in a cover is the number of candidate SQL queries needed to compute the mapping, we prefer smaller covers which will produce simpler mappings. When two or more covers have the same number of candidate sets, we prefer those that use the largest number of target attributes in all candidate sets and, thus, minimize the number of "null" values in the target. The ranked covers are presented as alternative mappings for the user to evaluate.

The final step is to build the query q from the selected cover. For each candidate set c_j in the selected cover, we create a candidate SQL query such that all correspondence functions f_i mentioned in c_j appear in the `SELECT` clause, all source relations are mentioned in the `FROM` clause, and all predicates p_i appear as a conjunction in the `WHERE` clause. Any join path determined in the second step for this candidate set will be used to determine the appropriate source relations for the `FROM` clause. The join predicates are also added to the `WHERE` clause. For each candidate set that includes aggregate functions (in either the correspondence or the filter), we select grouping attributes. All attributes (or functions on attributes) in the select clause that are not within the aggregate are selected as the grouping attributes. If the aggregate is in the correspondence function, the aggregate is placed in the select clause. If the aggregate is in the filter, the aggregate is placed in the `HAVING` clause. (We provide an example using aggregation in Section 4.) All candidate SQL queries are then combined into one large query using the multiset `UNION ALL`.

As in query optimization, the search space we consider in this algorithm is exponential. Nevertheless, we are able to provide heuristics that can guide the search towards likely covers and, thus, a correct schema mapping.

3.3 Making the Algorithm Incremental

Often, users will provide value correspondences incrementally and wish to see partial results before adding additional correspondences. We therefore provide an incremental version of the above algorithm. The algorithm takes as input a cover Γ_i and a single change $\Delta\mathcal{V}$ to the set of input value correspondences $\mathcal{V}$. The change $\Delta\mathcal{V}$ can be the addition (denoted $+v$) or the deletion (denoted $-v$) of a single value correspondence v. As output, the user is presented with a ranked set of possible next covers that are produced by the application of $\Delta\mathcal{V}$ to Γ_i. The cover selected by the user becomes the next cover Γ_{i+1}.

The incremental algorithm is divided into three phases (see Figure 4). The first phase does the work of the first and second phases of the batch algorithm presented in the previous section. Given a $+v$, the algorithm tries to insert v into all candidate sets of Γ_i. If the addition of v changes the set of source relations of a candidate set, a new join condition is sought (using the current join condition, if any, as seed for this search). The same heuristics discussed in the previous section to obtain a vertical composition are used here. If no candidate set in the cover can accept v, a new candidate set is created. For a deletion $-v$, the algorithm removes v from all candidate sets where it appears. Candidate sets that become empty, are removed from the cover. The result of this first phase is a set of changes $\Delta\Gamma$ that can be applied to the candidate sets in the current cover Γ_i.

The second phase of the algorithm applies each change in $\Delta\Gamma$ to Γ_i, producing a set of tentative covers Γ_{i+1}. Since the first phase limits the number of

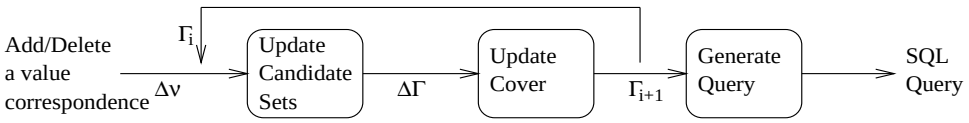

Figure 4: Incremental Mapping Algorithm

changes per candidate set to at most one change, the number of possible new covers is bounded by the number of candidate sets in the cover. This set of new covers are ranked as described in the previous section and presented to the user. The user selects one cover as the next Γ_{i+1}.

The third phase of this algorithm is identical to the fourth phase of the previous algorithm. Given a cover Γ_i, this phase produces an SQL query.

3.4 Nested-Sets in Target Relations

In addition to flat relational schemas, Clio can produce mappings to nested relational targets. Such mappings can be used to populate semi-structured schemas, including XML Schemas [W3C99]. For example, assume one of our target relations is `DeptInfo(number, name, staff:set of row(ename, eaddress))` where `staff` is a set of rows containing the name and address of each staff member. Given source relations `Department(dno, dname)` and `Professors(ssn, name, address, salary, dno)`, we could expect users to map values from `Department` into the outer-level of `DeptInfo` and values from `Professors` into `staff`. This mapping implies a join condition between the source relations `Department` and `Professor`.

Clio considers each target relation as an instance of a collection (or set) of row types. These row types can, in turn, contain collections of other row types. *Candidate sets* represent a possible mapping of the attributes of a particular collection and are maintained for each target collection (including nested collections). This forms a tree of candidate sets. For instance, in the example above, there is one candidate set that defines the mapping for `DeptInfo` and a candidate set under it that defines the maping for `staff`.

Join conditions for nested candidate sets include the extra step of finding (if needed) a way of joining the source relations of a particular candidate set with the source relations of each nested candidate set under it. In the example above, no join condition is needed for the candidate set of `staff`. However, a join condition between the source relations of that candidate set (`Employee`) and the source relation (`Department`) of the candidate set of the parent is needed.

Given a nested cover Γ, we can use a modified version of the procedure described in Section 3.2 to produce an SQL mapping query. The reasoning is similar to that used for flat relations and incorporates explicit knowledge about when nesting preserves the desired information. A nested query is added to the `FROM` clause of the candidate set's query for each of its nested candidate sets. To generate these nested queries, a recursive call is made to this procedure using the nested candidate sets as input. In the example used in this section, the expected output query is the following query.

```
SELECT DI.dno as number, DI.dname as name,
       EmpTable.EmpSet as staff
FROM DeptInfo DI,
     (SELECT SET OF(ROW(E.name, E.address)) AS EmpSet
      FROM   Employee E
      WHERE  E.dno = DI.dnumber) AS EmpTable
```

4 A Data Warehousing Example

We use an example based on a proposed software engineering warehouse for storing and exchanging information extracted from computer programs [BGH99]. Such warehouses have been proposed both to enable new program analysis applications, including data mining applications [MG99], and to promote data exchange between research groups using different tools and software artifacts for experimentation [HMPR97]. Figure 5 depicts a portion of a warehouse schema for this information. This schema has been designed to represent data about a diverse collection of software artifacts that have been extracted using different software analysis tools. The warehouse schema was designed to be as flexible as possible. As a result, it uses a very generic representation of software data as labeled multi-graphs. Conceptually, software artifacts (for example, functions, data types, macros, *etc.*) form the nodes of the graph. Associations or references between artifacts (for example, function calls or data references) form the edges. Two of the main tables for artifacts and references are depicted in the figure. Both tables are specialized with subtables containing specific types of software artifacts and references.

As new software analysis tools are developed, the data from these tools must be mapped into this integrated schema. In Figure 5, we also give a relational representation of facts extracted from the Rigi parser [MOTU93]. This schema may be supported by a wrapper built on top of Rigi [RS97]. Foreign keys are depicted by dashed lines. To map the Rigi data into the warehouse, the correspondences of Figure 6 may be used. In the Schema S, function and data type names are sufficient to disambiguate values within a software system. Within the warehouse, the information must be combined with meta-data describing the software system (for example, the program name and version). In Rigi, the program name and version are given in a header of a text file containing the set of all facts for

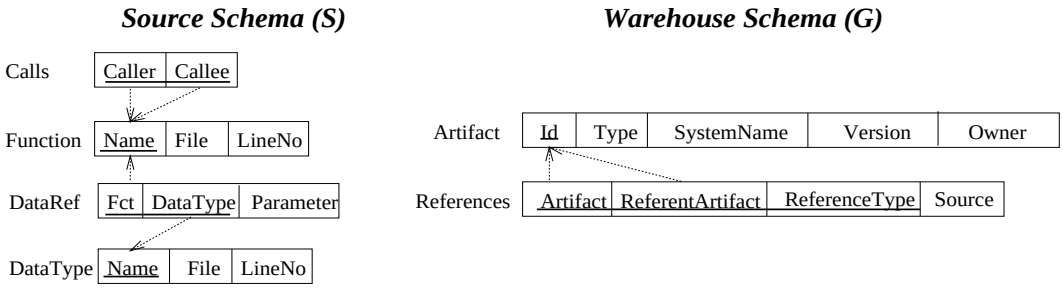

Figure 5: Schema of Rigi Source Database and a Software Engineering Warehouse

the program. The wrapper exposes this information using the meta-data functions dbname and dbversion. The correspondence f_1 is given below and the other correspondences are defined similarly. The function Id is a Skolem Function that produces a unique id for each unique set of values on which it is invoked [HY92]. Note that correspondences f_4 and f_5 map the relation name into the ReferenceType value, effectively transforming schema to data.

f_1: *Id(dbname(), dbversion(), Calls(Caller))* $\rightarrow$ *References(Artifact)*

The grouping algorithm of Clio uses the foreign key information in the source to create several candidate subsets. One contains the four correspondences $\{f_1, f_2, f_3, f_4\}$. Note that there are two foreign key join paths between the source relations involved in these correspondences. The first populates the Source attribute of the target with the File attribute of the caller function (Mapping S_1). The second populates the Source attribute of the target with the File attribute of the called function (Mapping S_2).

```
S1: SELECT Id(dbname(),dbversion(),C.Caller),
           Id(dbname(),dbversion(),C.Callee),
           relname(C), Id(dbname(),dbversion(),F.File)
    FROM   Calls C, Function F
    WHERE C.Caller = F.Name

S2: SELECT Id(dbname(),dbversion(),C.Caller),
           Id(dbname(),dbversion(),C.Callee),
           relname(C), Id(dbname(),dbversion(),F.File)
    FROM   Calls C, Function F
    WHERE C.Callee = F.Name
```

If we cannot distinguish these paths using the data, both will be presented to the user. The user is given some example values to help evaluate which of the join paths is correct (Figure 7). Based on the data, the user can pick the desired mapping. In this example, the user would choose the first since the source location of a program call is the location of the caller function.

A second candidate subset contains the four correspondences $\{f_5, f_6, f_7, f_3\}$. Note that Clio chooses to use f_3 in both candidates since there is a good foreign key path to use for both candidates. These two correspondences form a cover. Clio combines the Mapping S_1 and the mapping produced for this second candidate subset to produce the following complete schema

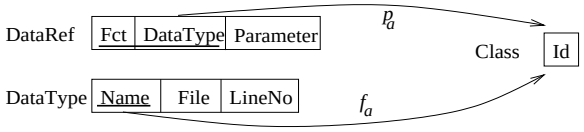

Figure 8: Aggregate filter in a value correspondence.

mapping. Since Clio favors grouping correspondences from the same relation, the other covers possible in this example are eliminated.

```
S: SELECT Id(dbname(),dbversion(),C.Caller),
          Id(dbname(),dbversion(),C.Callee),
          relname(C), Id(dbname(),dbversion(),F.File)
   FROM   Calls C, Function F WHERE C.Caller = F.Name
   UNION ALL
   SELECT Id(dbname(),dbversion(),D.Fct),
          Id(dbname(),dbversion(),D.DataType),
          relname(D), Id(dbname(),dbversion(),F.File)
   FROM DataRef D, Function F WHERE D.Fct = F.Name
```

To extend this example, consider the correspondence and filter used to define the Class table (a subtable of the Artifact table). Although the Rigi facts from Figure 7 represent C programs, the warehouse may contain tables like Class for storing information about object-oriented classes. C programs might be "reverse engineered" into C++ programs by grouping together into a class all functions that access a particular data type or set of data types. For brevity, we assume the Class table has a single Id attribute indicating the data type of the class (Figure 8).[3]

f_a : *Id(dbname(),dbversion(),DataType(Name))*$\rightarrow$*Class(Id)*
p_a : *count(DataRef(Fct))* > 5

The correspondence f_a maps data types to the Class table. The user also provides a filter p_a restricting the mapping to data types referenced by more than 5 functions. Clio discovers the join paths between DataRef and DataType. Given the aggregate function in the filter, the discovered mapping includes a group by and is depicted below.

```
Sa: SELECT Id(dbname(),dbversion(),T.Name)
    FROM   DataType T, DataRef R
    WHERE  T.Name = R.DataType
    GROUP BY Id(dbname(),dbversion(),T.Name)
    HAVING count(R.Fct) > 5
```

[3] While we are over-simplifying the reasoning behind reverse engineering methodologies, we are being faithful to the way these groups can be represented in SQL [MG99].

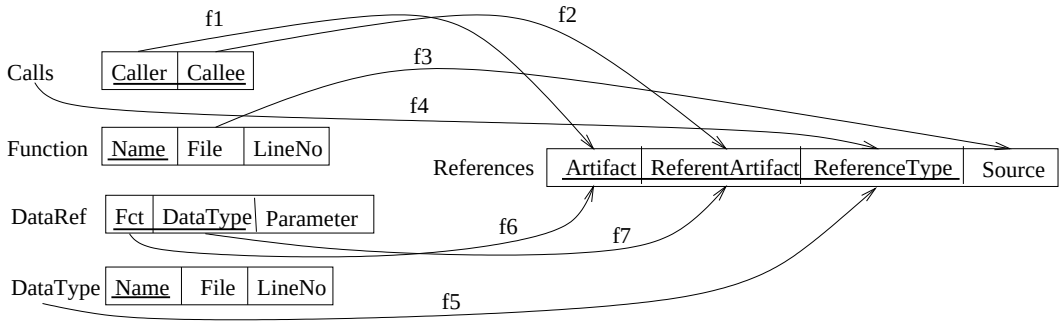

Figure 6: Value correspondences used to map the Rigi Schema to the Warehouse

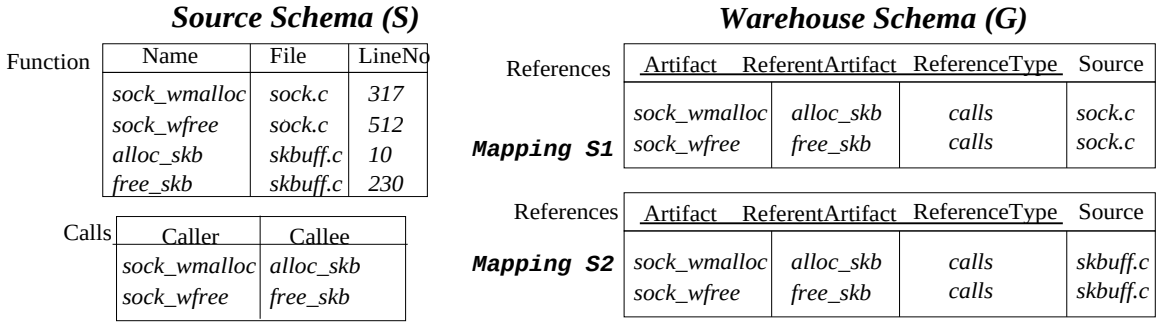

Figure 7: Discovered alternative schema mappings are used to derive example target data. The facts depicted are example facts from Rigi's analysis of the Linux software system. The files sock.c and skbuff.c contain the socket management and socket buffer support, respectively, for the network subsystem of Linux.

Additional real world examples, including an example of using Clio to produce a mapping to an XML schema, are included in the full version of this paper [MHH00].

5 Related Work

We have already described the differences between classical schema integration [RR99], which is primarily a schema design problem, and the schema mapping problem we have addressed here.

Related language-based approaches provide tools for the specification and implementation of data and schema translations. The YAT conversion language [CDSS98] permits the specification of data and schema matching and restructuring operations. The correspondence rules of [ACM97] are another example. These tools also include the schema matching techniques of [MZ98] for simplifying the specifications of matching rules. Our techniques complement and extend these language-based approaches to consider the general problem of query discovery. Finally, the search problem we consider is closely related to the problem of finding the set of all views that can be used to answer a query [LMSS95, DPT99].

6 Conclusions

In this paper, we identified a new problem, *targeted schema mapping*, that is of critical importance to several increasingly common classes of applications. We distinguished schema mapping from the well-known problem of schema integration, and discussed the similarities and differences between the two. By using queries to represent a mapping, we allow DBMSs to play an expanded role as data transformation engines, as well as data stores. Additionally, we find expanded uses for many techniques from query optimization, as we apply them to the new task of query discovery or mapping creation. Our framework for schema mapping uses *value correspondences* that describe how to populate a single attribute of the target schema. Given a set of value correspondences, we must discover the mapping query needed to transform source data to target data. We presented our algorithm for this often complex task, and introduced Clio, a tool that helps users create a schema mapping. Finally, we showed through extensive examples based on real applications how Clio would process a set of value correspondences to arrive at the mapping query.

Acknowledgements

Discussions with Bartholomew Niswonger helped shape the direction of Clio. Peter Schwarz contributed to the recognition of the importance of value mappings. We thank Periklis Andritsos for providing the Linux data used in our examples, and Ling Ling Yan and anonymous referees for their helpful comments.

References

[ACM97] S. Abiteboul, S. Cluet, and T.. Milo. Correspondence and Translation for Heterogeneous Data. In *Proc. of the Int'l Conf. on Database Theory (ICDT)*, pages 351–363, 1997.

[AH88] S. Abiteboul and R. Hull. Restructuring Hierarchical Database Objects. *Theoretical Computer Science*, 62:3–38, 1988.

[Bel97] Siegfried Bell. Dependency mining in relational databases. In *Proc. of the First Int'l. Joint Conf. on Qualitative and Quantitative Practical Reasoning*, volume 1244 of *LNAI*, pages 16–29, Berlin, June9–12 1997. Springer.

[BGH99] I. Bowman, M. Godfrey, and R. Holt. Connecting Software Architecture Recovery Frameworks. In *Proc. of the First Int'l Symposium on Constructing Software Engineering Tools (CoSET'99)*, Los Angeles, May 17-18 1999.

[BHP94] M. W. Bright, A. R. Hurson, and S. Pakzad. Automated Resolution of Semantic Heterogeneity in Multidatabases. *ACM TODS*, 19(2):212–253, June 1994.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion. In *ACM SIGMOD Conference*, pages 177–188, 1998.

[CHS+95] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proc. of the Fifth Int'l IEEE Wksp. on Research Issues in Data Eng. (RIDE-95): Distributed Object Mngmt.*, March 1995.

[Dat95] C. J. Date. *An Introduction to Database Systems*. Addison Wesley, 1995.

[DPT99] A. Deutsch, L. Popa, and V. Tannen. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *Proc. of the Int'l Conf. on VLDB*, pages 459–470, 1999.

[GL94] César A. Galindo-Legaria. Outer-joins as Disjunctions. In *ACM SIGMOD Conference*, pages 348–358, 1994.

[GLR97] César A. Galindo-Legaria and Arnon Rosenthal. Outer-join Simplification and Reordering for Query Optimization. *ACM TODS*, 22(1):43–73, 1997.

[HMN+99] L. M. Haas, R. J. Miller, B. Niswonger, M. Tork Roth, P. M. Schwarz, and E. L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[HMPR97] M. Harrold, R. J. Miller, A. Porter, and G. Rothermel. A Collaborative Investigation of Program-Analysis-Based Testing and Maintenance. In *International Workshop on Experimental Studies of Software Maintenance*, pages 51–56, Bari, Italy, October 1997.

[Hul86] R. Hull. Relative Information Capacity of Simple Relational Database Schemata. *Society for Industrial and Applied Mathematics (SIAM) Journal of Computing*, 15(3):856–886, August 1986.

[HY92] R. Hull and M. Yoshikawa. Object Identity and Query Equivalence. In J. D. Ullman, editor, *Theoretical Studies in Computer Science*, pages 253–286. Academic Press, Boston, MA, 1992.

[IBM97] IBM. DB2 DataJoiner Application Programming and SQL Reference Supplement. Technical report, IBM Corporation, 1997.

[Joh97] P. Johannesson. Linguistic Support for Analysing and Comparing Conceptual Schemas. *Data and Knowledge Engineering*, 21(2):165–182, 1997.

[KMRS92] M. Kantola, H. Mannila, K.-J. Rih, and H. Siirtola. Discovering Functional and Inclusion Dependencies in Relational Databases. *International Journal of Intelligent Systems*, 7(7):591–607, September 1992.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, San Jose, CA, May 1995.

[MG99] R. J. Miller and A. Gujarathi. Mining for Program Structure. *Int'l Journal on Software Eng. and Knowledge Eng.*, 9(5):499–517, 1999.

[MHH00] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. Technical Report CSRG-412, University of Toronto, Department of Computer Science, 2000.

[Mil98] R. J. Miller. Using Schematically Heterogeneous Structures. *ACM SIGMOD Conference*, 27(2):189–200, June 1998.

[MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proc. of the Int'l Conf. on VLDB*, pages 120–133, Dublin, Ireland, August 1993.

[MIR94] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems*, 19(1):3–31, 1994.

[MOTU93] H. A. Müller, M. A. Orgun, S. R. Tilly, and J. S. Uhl. A Reverse Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[MZ98] T. Milo and S. Zohar. Using Schema Matching to Simplify Heterogeneous Data Translation. In *Proc. of the Int'l Conf. on VLDB*, pages 122–133, NY, NY, 1998.

[Ora] Oracle. Rdb Distributed Technology Suite. http://oracle.com/rdb/download/rdb7/disttech/sy_dist.pdf.

[RCH99] V. Raman, A. Chou, and J. M. Hellerstein. Scalable Spreadsheets for Interactive Data Analysis. In *ACM-SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, Philadelphia, PA, May 1999.

[RR94] A. Rosenthal and D. Reiner. Tools and Transformations - Rigorous and Otherwise - For Practical Database Design. *ACM TODS*, 19(2), June 1994.

[RR99] S. Ram and V. Ramesh. Schema Integration: Past, Current and Future. In A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors, *Management of Heterogeneous and Autonomous Database Systems*, pages 119–155. Morgan Kaufmann Publishers, 1999.

[RS97] M. Tork Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. of the Int'l Conf. on VLDB*, pages 266–275, Athens, Greece, August 1997.

[RU96] A. Rajaraman and J. D. Ullman. Integrating Information by Outerjoins and Full Disjunctions. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 238–248, Montréal, Canada, 1996.

[W3C99] Xml schema part 1: Structures. http://www.w3.org/TR/xmlschema-1, December 1999.