Leveraging Query Logs for Schema Mapping Generation in U-MAP[*]

Hazem Elmeleegy
AT&T Labs - Research
hazem@research.att.com

Ahmed Elmagarmid
Qatar Computing Research Institute
Qatar Foundation
aelmagarmid@qf.org.qa

Jaewoo Lee
Purdue University
lee748@cs.purdue.edu

ABSTRACT

In this paper, we introduce U-MAP, a new system for schema mapping generation. U-MAP builds upon and extends existing schema mapping techniques. However, it mitigates some key problems in this area, which have not been previously addressed. The key tenet of U-MAP is to exploit the *usage* information extracted from the *query logs* associated with the schemas being mapped. We describe our experience in applying our proposed system to realistic datasets from the retail and life sciences domains. Our results demonstrate the effectiveness and efficiency of U-MAP compared to traditional approaches.

Categories and Subject Descriptors

H.2.5 [**Heterogeneous Databases**]: Data translation

General Terms

Algorithms, Design

Keywords

schema mapping, query logs, usage, data exchange, data integration

1. INTRODUCTION

One of the most common data management tasks is the transformation of data residing in one data source according to a given schema into the schema of another source. The problem of automatically discovering the necessary transformation rules to carry out this task is known as *schema mapping*, and the discovered rules are usually referred to as *mappings*.

There are multiple scenarios in which schema mapping is necessary. The first scenario is *data exchange* (e.g., [23]), where the data needs to be physically moved from one source to another, perhaps for data migration purposes. The second scenario is *data integration* (e.g., [29]) where it is required to answer queries expressed based on some mediated schema, while the result is obtained from some underlying sources, usually with different schemas. The query result, in this case, needs to be transformed to conform to the mediated schema before returning it to the user. Other application scenarios include schema evolution (e.g., [30]), peer data management systems (e.g., [15]), and model management [5].

It is evident, however, that the quality, and hence usefulness, of the transformed data hinges on the quality of the mappings used for the transformation. Therefore, it is crucial for the mapping generation process to be as accurate as possible, especially that any following data cleaning operations are generally quite expensive.

To this end, several systems (e.g., [3, 7, 11, 16, 22]) have been proposed in the literature for the generation of schema mappings. Over the last decade, these research efforts, and in particular the Clio project [11], played a great role in making the schema mapping field reach higher levels of maturity. Nevertheless, there are still several issues with the existing schema mapping tools, as their generated mappings do not take into consideration specific cases, which are very common in real-world schemas.

These issues include: (1) the unawareness of those tables representing special case entities, which have an *IS-A* relationship with higher-level entity types; (2) the inability to generate meaningful mappings, which involve joining tables with certain *many-to-many* relationships; and (3) the inability to resolve attribute correspondence conflicts which can arise during mapping generation. All of these issues will be explained in detail later in the paper.

The semantic approach for schema mapping [3] attempts to address some of these issues. However, it is only applicable when the schemas' *conceptual models* are available – a hard requirement to satisfy in many real-world scenarios. Moreover, we will also show that even when the conceptual models are available, there are many common cases, in which the method proposed in [3] will not be able to accurately generate the desired mappings.

In this paper, we introduce the U-MAP system, which mitigates the above problems by leveraging the *usage* information available in the query logs of the data sources being mapped. We show that this new resource, while not exploited before in the context of schema mapping, can be very valuable in generating significantly higher quality mappings. We summarize our contributions as follows.

- We describe a new mechanism for handling tables having *IS-A relationships*. The mechanism takes into account how to detect those tables, how to learn from the query log whether they are overlapping or not, and finally how to *merge* them appropriately in the generated mappings.
- We describe a new version of the chase algorithm [17] (commonly used for mapping generation). The new version, called the *aggressive chase*, enables the generation of mappings spanning all types of *many-to-many relationships*, as long as such mappings are deemed "semantically meaningful" based on the analysis of the query log.
- We describe a novel *conflict resolution* technique, which identifies the most likely mapping to resolve the attribute correspondence conflicts across the *logical relations* of the two schemas. At the core of this technique are two new methods: one for *context-based matching* of attributes between logical relations and queries from the log; and another for *context-based grouping* of attributes to limit the search space of possible mappings.
- We describe an experimental study on a working prototype for U-MAP using databases from the retail and life sciences domains. Our results demonstrate the effectiveness of U-MAP in dealing with the issues we raise in this paper as opposed to previous approaches.

The rest of the paper is organized as follows. Section 2 presents our running example and also some necessary background on the schema

[*]This research was supported by QCRI and by NSF-IIS 0916614.

mapping problem and the existing solutions for it. Section 3 illustrates some key outstanding issues with those existing solutions. The U-MAP approach is explained in detail in Section 4. Our experimental evaluation is given in Section 5. Section 6 describes the related work, and Section 7 finally concludes the paper.

2. RUNNING EXAMPLE AND BACKGROUND

2.1 Running Example

In this subsection, we describe our running example, which we will be referring to throughout the paper. The example shown in Figure 1 is from the retail domain. In particular, it represents the schemas of two bookstores. Each schema covers information on customers, books, authors, and orders – but using two different schema structures. Although not shown in the figure, it is assumed that attribute correspondences are also given. They could have either been generated manually or using an automatic schema matching tool (e.g., [24]). Any two attributes with the same name in both schemas correspond to each other (e.g., X_Customer.c_uname and Y_Customer.c_uname). There are also correspondences between pairs of attributes on both sides which do not have the exact same name (e.g., X_Distributor.d_uname and Y_Customer.c_uname). Finally, only non-key attributes are included in correspondences (e.g., there is no correspondence for c_id).

2.2 The Schema Mapping Problem

Consider the situation where we have two database systems, each with its own schema. It is required to develop declarative rules that specify how data records can be transformed from one schema (the source) to another (the target). The generated rules are generally called *schema mappings*.

We consider the most common formalism for schema mappings, which is *source-to-target tuple-generating dependencies*, or simply *s-t tgds*. Each mapping specifies how one or more records (with each record coming from a different table) from the source schema should be used to generate one or more records in the target schema. The mapping should clearly indicate how the multiple records (if more than one) at each schema are related (i.e., their join predicates), and additionally how the data values should be copied from the source attributes to the target attributes.

We use a "query-like" representation for the mappings, which will be illustrated in the following example.

Example 1: In our bookstores example, the following mapping specifies how a book record along with its corresponding author record in the X-Schema are transformed into three records in the Y-Schema, particularly, in the Y_Book, Y_Book_Author, and Y_Author tables.

m_1
```
for a in X_Author, b in X_Book
where (a.a_id = b.b_a_id)
⇒ exists a' in Y_Author, b' in Y_Book, ba' in Y_Book_Author
where (a'.a_id = ba'.ba_a_id ∧ b'.b_id = ba'.ba_b_id)
with (a'.a_fname = a.a_fname ∧ a'.a_lname = a.a_lname
∧ b'.b_title = b.b_title ∧ b'.b_pub_date = b.b_pub_date)
```

The for clause specifies the tables from which the source records are collected. The following where clause specifies the predicates for joining these records. Moreover, the exists clause indicates the tables of the generated records in the target schema. Similar to the first where clause, the second where clause also provides the join predicates for the target records. However, in addition, it also shows how the target attributes are populated using the values of the source attributes, as shown in the last two lines of m_1. □

More formally, the schema mapping problem can be defined by the tuple (S, T, Σ_S, Σ_T, V, Σ_{ST}), where S, T, Σ_S, Σ_T, and V are inputs, while Σ_{ST} is the desired output. S and T are the source and target schemas respectively. Σ_S and Σ_T are the constraints (typically referential integrity constraints) defined on each of S and T respectively. V is the set of attribute correspondences between S and T, which can be either automatically computed in a preceding schema matching step, or directly provided by the user. Σ_{ST} is the set of s-t tgds representing the desired schema mappings.

2.3 Existing Solutions

The commonly used approach for schema mapping is the one adopted in the Clio system [11], which is based on analyzing the referential integrity constraints in the source and target schemas. We will refer to this approach as the *RIC-based approach*. A more recent approach relies on using additional semantic information about both schemas, which can be found in their associated conceptual models (e.g., ER diagrams, UML). This approach will be referred to as the *semantic approach*. In what follows, we overview the two approaches (with a little more emphasis on the RIC-based approach since our proposed techniques will be building on it).

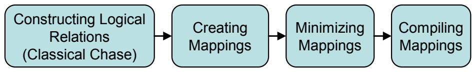


Figure 2: The RIC-based schema mapping process.

2.3.1 *RIC-Based Approach*

This approach proceeds in four main steps as shown in Figure 2.

Step 1 (Constructing Logical Relations): At first, we construct what is referred to as the *logical relations*, separately for each schema. A logical relation is meant to group together all the *related* concepts (or attributes) in a schema, even if these concepts do not reside in the same table in the schema. In other words, a logical relation can be the result of joining multiple tables together.

Since there is typically a large number of possible joins to be done in a schema, where most of them is not semantically meaningful, the goal in this step is to limit the generation of logical relations to the meaningful ones only. For this purpose, the RIC-based technique "expands" each table, A, in the schema into its corresponding logical relation by joining A with all the tables it references to get some intermediate logical relation. Then, the just-generated intermediate logical relation is, in turn, joined with all the tables it references, and so on. The final logical relation corresponding to A is the one which does not contain any more foreign keys.

The process we just described is essentially applying the chase algorithm [17] on each table, where the set of constraints applied during the chase are the referential integrity constraints in each of the schemas (i.e., Σ_S and Σ_T for S and T respectively).

Step 2 (Creating Mappings): Once the logical relations are identified for both schemas, the second step is to find all the pairs of logical relations (one from each schema), which cover one or more attribute correspondences from V. Each such pair will constitute a mapping.

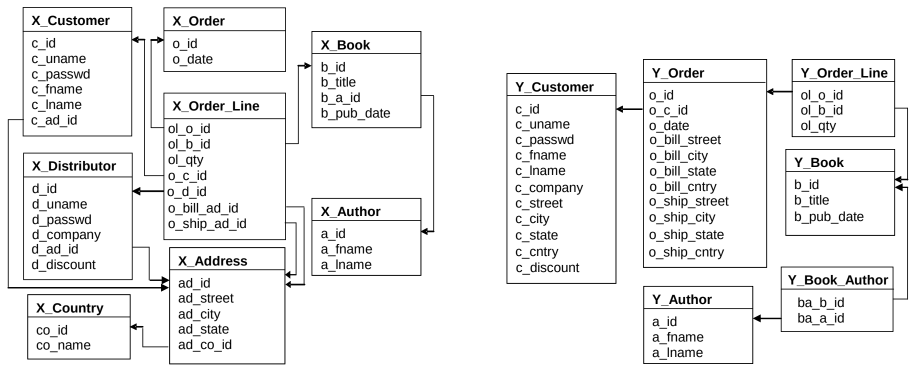
Example 2: Considering the mapping m_1 from Example 1, we find that it *maps* a logical relation from S, specified in the for and first where clauses, to a logical relation from T, specified in the exists and second where clauses. The attribute correspondences between the two logical relations are specified in the with clause, in order to show how values should be copied from the source attributes to the target attributes. □

When a target attribute does not have a corresponding source attribute, we cannot simply have it populated by nulls, because it may be a required attribute, where nulls are not allowed. Moreover, it may not have a default value, which makes it necessary to synthesize new values for its population. These synthetic values, usually referred to as *labeled nulls*, are generated using *Skolem functions*.

Skolem functions ensure that their outputs are different whenever their inputs are different. One of their essential uses is to maintain the referential integrity in the target database. So for example a book record in T will only be linked to the record of its own author, and not any other author records. In this case, the Skolem functions used to generate the book id and author id in the Y_Book and Y_Author records, respectively, should be identical to the ones used to generate the foreign keys in the corresponding Y_Book_Author record.

Step 3 (Minimizing Mappings): The third step is an optimization step, which has been introduced in [13]. In this step, the set of generated mappings can be *minimized* by discarding all the mappings *subsumed* or *implied* by other mappings. The criteria for when a mapping is considered to subsume or imply another mapping are given in [13].

Step 4 (Compiling Mappings): The final step is to compile the minimal set of mappings into an executable script, e.g. SQL queries.



(a) X-Schema (b) Y-Schema

Figure 1: Schemas of two bookstores.

Running this script should actually move the data from the source to the target while making the necessary transformations, which would still respect the constraints of both the source and the target.

2.3.2 *Semantic Approach*

The semantic approach for schema mapping [3] does not only rely on the source and target schemas, as is the case with the typical RIC-based approach. Instead, it takes the *conceptual models* for both the source and the target into account as well. Examples for information that can be found in conceptual models, but not in schemas, include the *cardinalities* of the relationships between entities, and also special types of relationships like the *IS-A* relationship.

In many cases, this additional information helps in distinguishing the more meaningful mappings from the less meaningful ones. If only the RIC-based approach is used, then such mappings will all seem to be equally meaningful to the mapping tool. The details of how the semantic approach exploits this information can be found in [3].

3. ISSUES WITH EXISTING SOLUTIONS

In this section, we describe in more detail the issues involved when relying on the RIC-based approach. At the same time, we point out the situations in which the semantic approach can address some of these issues, and those situations in which some issues remain unresolved. Most clearly, the semantic approach will not be helpful whenever the conceptual models are missing, which needless to say, is a common scenario in the real world. We now explain each of such issues.

3.1 Unawareness of IS-A Relationships

In many schemas, multiple tables can represent different specializations of a higher-level abstract concept or *superclass* (e.g., *student* and *instructor* tables would both store information about *persons*). In other words, they have an *IS-A* relationship with the superclass (which may not itself have have a separate table in the schema). Such tables will thereby have common attributes pertaining to their superclass, in addition to other attributes specific to each of them.

A key question is whether these tables are disjoint or not. In other words, can the same real-world entity have a record in each of these tables? The answer to this question determines how the tables should be correctly *merged*. Merging the tables will be needed if they are to be mapped from the source schema to a single table in the target schema. If they are disjoint, then the merger can be achieved using a simple union. Otherwise, they will need to be *outer-joined* to guarantee that no data duplication occurs in the target database. This can be illustrated by the following example.

Example 3: In our bookstore example, the tables X_Customer and X_Distributor can both be thought of as having an IS-A relationship with some abstract concept, *Buyer*. Hence, if we blindly apply the RIC-based method, we will generate the two mappings m_2 and m_3 given below.

```
m2   for c in X_Customer, ad in X_Address, co in X_Country
     where (c.c_ad_id = ad.ad_id ∧ ad.ad_co_id = co.co_id)
     ⇒ exists c' in Y_Customer
     with (c'.c_uname = c.c_uname ∧ c'.c_passwd = c.c_passwd
     ∧ c'.c_fname = c.c_fname ∧ c'.c_lname = c.c_lname
     ∧ c'.c_street = ad.ad_street ∧ c'.c_city = ad.ad_city
     ∧ c'.c_state = ad.ad_state ∧ c'.c_cntry = co.co_name)
```

```
m3   for d in X_Distributor, ad in X_Address, co in X_Country
     where (d.d_ad_id = ad.ad_id ∧ ad.ad_co_id = co.co_id)
     ⇒ exists c' in Y_Customer
     with (c'.c_uname = d.d_uname ∧ c'.c_passwd = d.d_passwd
     ∧ c'.c_company = d.d_company
     ∧ c'.c_street = ad.ad_street ∧ c'.c_city = ad.ad_city
     ∧ c'.c_state = ad.ad_state ∧ c'.c_cntry = co.co_name)
```

These two mappings separately map the records of each of X_Customer and X_Distributor in the X-Schema to the Y_Customer table in the Y-Schema. However, we may be able to determine that X_Customer and X_Distributor are overlapping; i.e., some distributors also have records in X_Customer (perhaps to maintain the contact person information for the distributor). In this case, the best mapping to populate Y_Customer should look as follows.

```
m4   for (c in X_Customer outerjoin d in X_Distributor),
     ad in X_Address, co in X_Country
     where (ad.ad_id = ifnull(c.c_ad_id,d.d_ad_id)
     ∧ ad.ad_co_id = co.co_id)
     ⇒ exists c' in Y_Customer
     with (c'.c_uname = ifnull(c.c_uname,d.d_uname)
     ∧ c'.c_passwd = ifnull(c.c_passwd,d.d_passwd)
     ∧ c'.c_fname = c.c_fname ∧ c'.c_lname = c.c_lname
     ∧ c'.c_company = d.d_company
     ∧ c'.c_street = ad.ad_street ∧ c'.c_city = ad.ad_city
     ∧ c'.c_state = ad.ad_state ∧ c'.c_cntry = co.co_name)
```

Note that the mapping m_4 first computes the *outer join* of X_Customer and X_Distributor to ensure that records belonging to the same real world entity from both tables are merged first. (The ifnull function is used to merge common attributes in X_Customer and X_Distributor into a single attribute in Y_Customer. It returns the first parameter, or the second in case the first was null.) Then, the records of the resulting outer join are used to populate the Y_Customer table. This approach guarantees that a given real world entity (e.g., a distributor) will only have one record in Y_Customer. If X_Customer and X_Distributor were disjoint, then just using m_2 and m_3 would be sufficient to populate Y_Customer with the union of the two source tables, and hence no special treatment is needed in this case. $\Box$

Of course, beyond our running example, many other examples exist in the real world for tables with overlapping sets of entities. Consider for instance singers and actors, graduate students and instructors, mobile phones and digital cameras – to name just a few.

Using the semantic approach to detect IS-A relationships, as well as the disjointness of the involved tables, can only be possible if the conceptual model for each schema is available and if the disjointness constraints are specified in them. Unfortunately, the RIC-based approach, which is more commonly used does not recognize IS-A relationships or disjointness constraints.

It is worth mentioning that a newly introduced technique [18] can handle a special case of this problem. In particular, [18] presents a post-processing mapping re-writing algorithm, which focuses on maintaining the key constraints (equality generating dependencies, or egds) on the target. This technique works well when the key attributes in the source tables (with an IS-A relationship) are mapped to the key attribute in the target table. The re-written mappings will ensure that records for the same entity in the source tables will be not be mapped to different records in the target table, since they all have the same key in the source, and that key will be mapped to the target as well.

However, it is quite common that keys are just serial numbers and are not included in the correspondences because they do not have any semantic meaning (as in Example 3). Instead, new keys are generated at the target for each newly inserted record. In this scenario, the technique in [18] cannot prevent duplicate records in the target because even if they refer to the same real-world entity, each will be assigned a separate key, and hence no key constraint violations will occur.

3.2 Incomplete Coverage of Logical Relations

As we explained in Section 2.3.1, the RIC-based approach relies on the chase algorithm to generate the logical relations. This algorithm chases foreign keys in the forward direction only; i.e., it follows the edges in the schema going from the foreign key in one table to the primary key in the other table, but it never chases them in the opposite direction. The joins resulting from this chase will always be *lossless*; i.e., at least one of the relations participating in the join can be reconstructed just by applying projection on the join result. The fact that the generated logical relation is a result of a lossless join guarantees that all the records in that relation are semantically meaningful.

In contrast, if the join involved edges going in opposite directions, then the join will be considered a *lossy join*; i.e., it is not guaranteed that any of the participating relations can be reconstructed just by applying projection on the join result. In this case, some of the records in the output relation may not even be meaningful. Moreover, the number of possible lossy joins in a schema can be overwhelming. For these reasons, the RIC-based approach does not consider lossy joins when generating the logical relations.

However, we argue that some lossy joins may in fact be meaningful, and hence it is essential to include them in the finally generated mappings.

The following two examples will illustrate the concepts of lossy and lossless joins, as well as when they result in meaningful records and when they do not.

Example 4: Consider the case when the RIC-based technique generates the target logical relation starting from the Y_Order_Line table. It will chase the foreign keys referencing the Y_Book and Y_Order tables, and then it will chase the foreign key in Y_Order, which references Y_Customer. At this point, no more foreign keys will be left to chase, and the algorithm stops.

Note that this chase does not relate the order line information to the book's author information, although it is quite logical to have them related. The reason is that in the Y-Schema, Y_Book and Y_Author have a many-to-many relationship. Thus, each book is not directly referencing its author(s), but rather they are connected through an intermediate table which references both of them. Since the chase algorithm only goes in one direction, it stops at the Y_Book table.

Ideally, we would like the mapping responsible for populating the Y_Order_Line table to connect each order line record with its corresponding author record as shown in the mapping m_5 below. However, the RIC-based approach will never generate m_5.

```
m5   for ol in X_Order_Line, b in X_Book, a in X_Author, ···
     where (ol.ol_b_id = b.b_id ∧ b.b_a_id = a.a_id ∧ ··· )
     ⇒ exists ol' in Y_Order_Line, b' in Y_Book,
     ba' in Y_Book_Author, a' in Y_Author, ···
     where (ol'.ol_b_id = b'.b_id ∧ b'.b_id = ba'.ba_b_id
     ∧ a'.a_id = ba'.ba_a_id ∧ ··· )
     with (a'.a_fname = a.a_fname ∧ a'.a_lname = a.a_lname
     ∧ b'.b_title = b.b_title ∧ b'.b_pub_date = b.b_pub_date
     ∧ ol'.ol_qty = ol.ol_qty ∧ ··· ) □
```

To make our point even clearer regarding the need for considering lossy joins in certain situations, we present the following example.

Example 5: Consider that we have a third schema, Z-Schema, containing a single table called Z_Reader_Author, which tracks the readers for each author. Hence, Z-Schema will simply look as follows.

Z_Reader_Author(r_fname, r_lname, a_fname, a_lname)

When we generate mappings from the Y-Schema (source) to the Z-Schema (target), there must be a logical relation in the Y-Schema which connects the authors to the customers through the book orders made by the customers. As explained in Example 3, the RIC-based approach will not generate this logical relation. Instead, it will populate the target table using the two logical relations for X_Customer and X_Author. In other words, some records will contain the customer (reader) names only and others will contain the author names only, with no linkage between them – which defeats the whole purpose of building the target table. □

We will now show a scenario where chasing foreign keys in opposite directions can indeed result in semantically meaningless logical relations, and hence the need for a mechanism to distinguish between the situations in which lossy joins should be considered and those in which they should not.

Example 6: In the X-Schema, consider that the table X_Author has an extra attribute a_nationality, which references X_Country. Clearly, there is no meaning in creating a logical relation connecting authors to customers through the X_Country table just because a customer resides in the same country where an author is a citizen(!). □

It is worth noting that the semantic approach attempts to identify some of the situations where a lossy join is meaningful by considering the cardinality constraints for the relationships between entities, as indicated in the conceptual model. If for example two entities in the source have a many-to-many relationship and two corresponding entities in the target also have a many-to-many relationship, then a mapping is generated from the join of the two corresponding source tables to the join of the two corresponding target tables. The mapping is generated even if the joins are lossy, because in this case, the semantic approach considers the compatibility of the cardinality constraints as an indication that the mapping is semantically meaningful.

Even if we assume that the source and target conceptual models are available, this strategy of the semantic approach would still have two main drawbacks. First, a mapping involving lossy joins may be semantically meaningful even if the cardinality constraints for the relationships in both schemas are not compatible. For example, the relationship between order lines and authors is one-to-many in the X-Schema and many-to-many in the Y-Schema, yet the mapping m_5 (from Example 4), which maps the two relationships is semantically meaningful. The semantic approach will not generate m_5.

Second, if relationships between corresponding entities in both the source and the target have compatible cardinality constraints, then still a mapping between the joins of their participating tables can be semantically meaningless. This was shown in Example 6, where the composite relationship between X_Author and X_Customer (going through X_Country) is many-to-many, and similarly the composite relationship between Y_Author and Y_Customer (going through Y_Book_Author, Y_Book, Y_Order_Line, and Y_Order) is also many-to-many. Still, however, generating a mapping between the two relationships is meaningless, primarily because the first relationship (in the X-Schema) is itself not semantically meaningful.

Moreover, some of the recent works [8, 19] proposed that users can provide certain "joins", which are deemed important to consider during the construction of logical relations, to compensate for those missed by the chase algorithm. However, no automatic method for discovering such important joins have been presented.

3.3 Unresolved Correspondences

During the construction of mappings, sometimes it is not clear how to populate the target attributes using the source attribute values. This situation occurs when the correspondences between the attributes of the two logical relations being mapped are conflicting. Such conflicts may exist even if the original set of correspondences, V, is conflict-free. To see why, note that constructing a logical relation typically involves joining tables together. Sometimes the same table is included in the logical relation *more than once*. In this case, the attributes of such a table will appear multiple times in the logical relation (certainly with different semantics, or contexts, depending on the join predicates used each time the table is joined).

Now, if a repeated attribute in a source logical relation corresponds to some target attribute, populating that target attribute becomes unresolved since its values can be obtained from more than one source attribute.

Example 7: In the X-Schema, the X_Order_Line table has two foreign keys referencing X_Address, representing the shipping and billing addresses associated with the order line. Clearly, the logical relation for X_Order_Line will include the address attributes (i.e., street, city, $\cdots$, etc) more than once.

Similarly, the target logical relation for Y_Order_Line also contains more than one set of address attributes (to capture the billing, shipping, and customer addresses). The mapping m_6 given below is the one which correctly resolves all the conflicts in the attribute correspondences between the source and target logical relations.

```
m6 for o in X_Order, ol in X_Order_Line, ad1 in X_Address,
   ad2 in X_Address, co1 in X_Country, co2 in X_Country, · · ·
   where (ol.ol_o_id = o.o_id ∧ ol.ol_bill_ad_id = ad1.ad_id
   ∧ ol.ol_ship_ad_id = ad2.ad_id ∧ ad1.ad_co_id = co1.co_id
   ∧ ad2.ad_co_id = co2.co_id ∧ · · · )
   ⇒ exists o’ in Y_Order, ol’ in Y_Order_Line, · · ·
   where (ol’.ol_o_id = o’.o_id ∧ · · ·
   with ∧ o’.o_date = o.o_date ∧ ol’.ol_qty = ol.ol_qty
   ∧ o’.o_bill_street = ad1.ad_street ∧ o’.o_bill_city = ad1.ad_city
   ∧ o’.o_bill_state = ad1.ad_state ∧ o’.o_bill_cntry = co1.co_name)
   ∧ o’.o_ship_street = ad2.ad_street ∧ o’.o_ship_city = ad2.ad_city
   ∧ o’.o_ship_state = ad2.ad_state ∧ o’.o_ship_cntry = co2.co_name)
   ∧ · · · ) □
```

The existing RIC-based approach cannot independently determine that m_6 from Example 7 is the correct mapping. Hence, it generates a number of mappings equal to the number of all possible ways to resolve the correspondence conflicts. It then becomes the user's duty to select the correct mapping, which is both time-consuming and error-prone. Using the semantic approach will not help in this case either, because the conceptual model (if available) does not contain the necessary information to recognize that m_6 is the correct mapping.

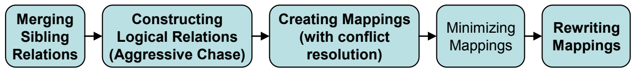
Some recent extensions to the RIC-based approach attempt to reduce the amount of effort needed to resolve the correspondence conflicts. In particular, [1] proposed to assist the user in resolving the ambiguity among mappings using data instances. Given a real data instance from the source for a given mapping, the user is asked to choose the correct values for the target attributes involved in that mapping. The user choices will then determine which of the ambiguous mappings is correct. Also, [20] describes a GUI which allows the user to duplicate certain tables in the schema, and then specify the correspondences for each one of the duplicates separately. Such duplications can help in resolving correspondence conflicts.

However, the above two approaches still require a lot of human time and effort, especially for large and complex schemas. This becomes evident in Section 5.2, where we found that after the construction of logical relations for a moderate-sized schema, more than one thousand attributes in those logical relations were involved in correspondence conflicts.

4. THE U-MAP APPROACH

In this section, we describe our approach in the U-MAP system. We show how we can boost the schema mapping quality when we take query logs into account to address all of the aforementioned issues. In particular, we enrich the schema mapping process as illustrated in Figure 3, where the **boldfaced** steps are either newly introduced or significantly modified. Each one of the introduced/modified steps addresses one of the issues raised in Section 3, mainly by exploiting information present in the query log.

The "merging sibling relations" step is introduced as a preprocessing step to merge groups of tables, which seem to be a specialization of the same parent abstract concept (and overalapping), into a single unified relation. Then, during the "constructing logical relations" step, the classical chase algorithm is replaced by a more aggressive version, which is able to discover additional semantically meaningful logical relations. Moreover, the core "creating mappings" step is extended to enable conflict resolution for attribute correspondences, which may arise in situations similar to those described in Section 3.3. The minimization step remains unchanged in U-MAP. However, a post-processing "re-writing mappings" step is introduced to ensure that the generated mappings only refer to original relations in the input schemas and not to any derived relations that could have been added during the initial merging step. Finally, the "compiling mappings" step is not shown in the figure because U-MAP currently focuses only on the generation process itself for the mappings.



Merging Sibling Relations
Constructing Logical Relations (Aggressive Chase)
Creating Mappings (with conflict resolution)
Minimizing Mappings
Rewriting Mappings

Figure 3: The schema mapping process used in U-MAP. Boldfaced steps are either newly introduced or modified, compared to classical RIC-based schema mapping.

4.1 Managing Sibling Relations

To properly handle the issue of *overlapping sibling relations* (tables having an IS-A relationship with an abstract superclass, and whose records overlap), we need to introduce a pre-processing step and a post-processing step. The goal of the pre-processing step is to first *discover* all the overlapping sibling relations in the schemas, and then to *merge* and replace them with some derived *super relations*. This way, the schema mapping problem will appear to the next steps in the pipeline to be free of any issues related to overlapping sibling relations. In the post-processing step, however, all the output mappings will need to be *re-written* to replace any reference to the super relations with the original sibling relations they represent.

For the pre-processing step, we initially use a simple scheme to discover candidate sibling relations. We consider that two tables are siblings if they have one or more *shared* attributes. Two attributes are considered to be shared across two tables if they both correspond to the same attribute in the other schema. Although simple, this scheme is effective enough to discover sibling relations with a high recall. In our running example, both X_Customer and X_Distributor will be considered siblings because their attributes c_uname and d_uname both correspond to c_uname in Y_Customer for instance.

Since we are only interested in *overlapping* sibling relations, we now need to filter out all the disjoint sibling relations, and the nonsibling relations which might have been erroneously returned by the initial discovery scheme. This is where we rely on the query logs.

The key idea is to determine if there were already specific queries looking for that overlap. In particular, we scan the query log, and for each pair of tables we find to be joined on their primary keys, we check this pair against the discovered pairs of sibling relations. If found, then we mark the pair as overlapping. All pairs which were not found to be joined on their primary keys are filtered out. Note that if the two tables were disjoint, or not siblings in the first place, then their primary keys will be unrelated and it would be pointless to join them using their primary keys.

Each pair of overlapping sibling relations will then be merged into a new super relation, which is the outer join (on the primary keys) of the two sibling relations. The attributes of the super relation will be the union of the attributes of both relations, as shown in the following example.

Example 8: Consider that the two relations X_Customer and X_Distributor in the X-Schema are found to be overlapping sibling relations. Then, their super relation (call it X_Buyer) will be defined using the following query.

```
X_Buyer =    select ifnull(c.c_id,d.d_id) bu_id,
             ifnull(c.c_uname,d.d_uname) bu_uname,
             ifnull(c.c_passwd,d.d_passwd) bu_passwd,
             c.c_fname bu_fname, c.c_lname bu_lname,
             d.d_company bu_company, d.d_discount bu_discount,
             ifnull(c.c_ad_id,d.d_ad_id) bu_ad_id
             from X_Customer c outerjoin X_Distributor d
             on c.c_id=d.d_id
```

Note that each pair of common attributes in X_Customer and X_Distributor are mapped to a single attribute in X_Buyer using the ifnull function. For example, c_uname in X_Customer and d_uname in X_Distributor are mapped to bu_uname in X_Buyer using the function ifnull(c_uname,d_uname). □

To complete the merging process, both the schema and the query log need to be updated such that the newly created super relations properly replace the sibling relations they represent. For the schema, any foreign keys referencing the sibling relations should be updated to reference the new super relation instead. For the query log, it should also be updated to be consistent with the schema. In particular, each reference to a sibling relation in a query is replaced by its corresponding super relation. This query re-writing process is fairly straightforward because each of the sibling relations can be regarded as a projection over the super relation, and hence query re-writing will proceed based on the well-understood view-unfolding mechanism [25].

In case more than one pair of overlapping sibling relations were discovered, then the process described above will be repeated for each pair iteratively, until all the pairs have been merged.

As a result of this merging process, the output mappings will still be referring to the super relations, which were not part of the original input schemas. Therefore, in the post-processing step, all the mappings will be re-written – also using the view-unfolding mechanism. But this time, the views defining the super relations in terms of their underlying sibling relations will be used instead (See Example 8). During rewriting, every reference to an attribute in a super relation representing the merger of two attributes, a and b, is replaced by ifnull(a,b) (with the order being currently arbitrary in U-MAP), leading to mappings similar to m_4 in Example 3. Note that the use of the ifnull function in the final output mappings is comparable to the use of skolemization/concat functions in [26] for instance.

4.2 Aggressive Chase

Now that we are confident that the updated input schemas will not have any overlapping sibling relations, we can proceed with the "constructing logical relations" step. We have seen in Section 3.2 that relying on the classical chase algorithm in this step can result in missing some meaningful logical relations. In this section, we describe how we can exploit the information in the query logs to extend the classical chase algorithm into a more aggressive version, where even when no foreign keys are left to chase in the forward direction, the algorithm may still decide to chase a foreign key in the reverse direction.

We start by formally analyzing the situations where the classical chase will either discover or miss interesting logical relations. Let us first note that a logical relation can essentially be represented by a *directed tree*, with the root being the table where the chase starts. The other nodes represent the other tables visited during the chase, and the edges indicate the direction in which the chase occurred. With classical chase, the directed tree is always an *arborescence* (i.e., all edges point away from the root). If the chase is permitted to occur in the reverse direction (as we will explain shortly), then the directed tree will be a general one, where edges can either point to or away from the root.

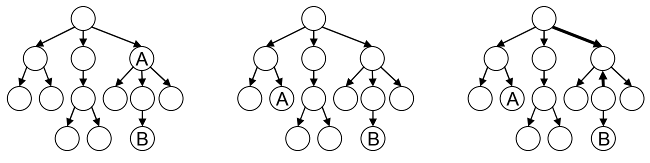
Moreover, we can classify the types of associations between attributes in a logical relation as follows. Given two attributes a and b in the same logical relation, which originally belong to tables A and B respectively, the relationship between A and B can either be 1-1, 1-m, m-1, or m-m. Without loss of generality, we will only focus on the m-1 and m-m relationships (since 1-m is identical to m-1 given that the two tables are switched, and 1-1 is a special case of m-1).

In the directed tree representation of a logical relation, an m-1 relationship occurs between A and B if A is the ancestor of B, as shown in Figure 4(a). For m-m relationships, we can actually further classify them into two types: *common-ancestor many-to-many* and *common-ancestor-free many-to-many* according to the definitions below.

Definition 1: Common-Ancestor Many-to-Many (CA-m-m) Relationships: *Two relations A and B, which are both used in constructing the same logical relation, $\mathbb{A}$, are said to have a common-ancestor many-to-many (CA-m-m) relationship if there exists a third relation C also used in constructing $\mathbb{A}$, such that the relationships between C and A is many-to-one and between C and B is also many-to-one (i.e., C is the common ancestor of both A and B in the directed tree representation of $\mathbb{A}$).*

Definition 2: Common-Ancestor-Free Many-to-Many (CAF-m-m) Relationships: *Two relations A and B, which are both used in constructing the same logical relation, $\mathbb{A}$, are said to have a common-ancestor-free many-to-many (CAF-m-m) relationship if (1) there does NOT exist a third relation C also used in constructing $\mathbb{A}$, such that the relationships between C and A is many-to-one and between C and B is also many-to-one; and (2) the relationship between A and B is neither m-1 nor 1-m (i.e., there is NO common ancestor for both A and B in the directed tree representation of $\mathbb{A}$).*

The two types of relationships, CA-m-m and CAF-m-m, are illustrated in Figures 4(b) and 4(c) respectively. The classical chase can construct logical relations capturing m-1 and CA-m-m relationships, because starting from the common ancestor the chase can proceed in the forward direction to include both A and B. However, it cannot capture CAF-m-m relationships (for the absence of a common ancestor), although such relationships can be interesting as was shown in Examples 4 and 5 in Section 3.2. We show next how we discover interesting CAF-m-m relationships with the help of query logs.



(a) m-1 (b) CA-m-m (c) CAF-m-m

Figure 4: Possible relationships between two tables, A and B, in a logical relation.

Building the FR-Index: In U-MAP, prior to executing the chase algorithm, we first analyze the query log and build an index structure, which we refer to as the *FR-Index* (short for Forward-Reverse Index). Each entry in the FR-Index contains a pair of opposing references; i.e. two different foreign keys referencing the same primary key (see Figure 5(b) for an example). The first reference is considered the forward reference, and the second one is considered the reverse reference. What characterizes the specific pairs maintained in the FR-Index is that joining the referenced table with the two referencing tables using such pairs of foreign keys should result in a semantically meaningful output.

The FR-Index is constructed as follows. Each query encountered in the query log is modeled as a graph, where nodes represent the tables involved in the query, and edges represent the join predicates used to join each pair of tables. If the same table is used more than once in the query, then it will be represented by multiple nodes in the graph.

For a node with n incoming edges, the $\binom{n}{2}$ possible pairs of such edges are stored in the FR-Index. Since each edge in a given pair can either be considered as a forward or a reverse reference (depending on which edge is traversed first), a pair of edges (e,e') is stored twice in the FR-Index: as (e,e') and as (e',e) to account for both cases.

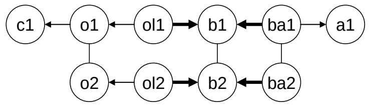
The following example illustrates how a single query is analyzed, and how the FR-Index is populated as a result of this analysis.

Example 9: Consider a query posed on the Y-Schema that requests for customer "John Smith" all the pairs of books written by the same author that John has ordered within less than 30 days. The query will look as follows.

```
Q1   select b1.b_title, b2.b_title, a1.a_fname, a1.a_lname
     from Y_Customer c1, Y_Order o1, Y_Order_Line ol1,
     Y_Book b1, Y_Book_Author ba1, Y_Author a1, Y_Order o2,
     Y_Order_Line ol2, Y_Book b2, Y_Book_Author ba2
     where c1.c_id=o1.o_c_id and o1.o_id=ol1.ol_o_id
     and ol1.ol_b_id=b1.b_id and b1.b_id=ba1.ba_b_id
     and ba1.ba_a_id=a1.a_id and c2.c_id=o2.o_c_id
     and o2.o_id=ol2.ol_o_id and ol2.ol_b_id=b2.b_id
     and b2.b_id=ba2.ba_b_id and ba2.ba_a_id=a2.a_id
     and ba1.ba_a_id=ba2.ba_a_id and b1.b_title<>b2.b_title
     and abs(o1.o_date-o2.o_date)<30 and o1.o_c_id=o2.o_c_id
     and c1.c_fname='John' and c1.c_lname='Smith'
```

The graph constructed for Q_1 is shown in Figure 5(a), and Figure 5(b) shows how the FR-Index is finally populated as a result of analyzing Q_1. The **bold** edges are the ones corresponding to the opposing references, which get stored in the FR-Index. Note that in this case, Y_Order_Line and Y_Book_Author give an example of an interesting CAF-m-m relationship. □

Aggressive Chase Algorithm: Given the FR-Index, the chase algorithm can now be modified as shown in Algorithm 1. Starting from a given table A, the algorithm will execute a series of alternating *rounds of chase*: (forward chase, reverse chase, forward chase, ... etc). The first round is a regular forward chase where A is joined with all the tables it references to create the first intermediate logical relation. The



(a) Query Graph

FR-Index

Forward	Reverse
(Y_Book_Author.ia_b_id, Y_Book.b_id)	(Y_Order_Line.ol_b_id, Y_Book.b_id)
(Y_Order_Line.ol_b_id, Y_Book.b_id)	(Y_Book_Author.ia_b_id, Y_Book.b_id)

(b) FR-Index

Figure 5: Q_1's query graph and the corresponding entries in the FR-Index for Example 9.

second round is a reverse chase, where the algorithm will check each forward reference it used in the previous round against the FR-Index. If one or more lookups for the forward references return corresponding reverse references, then the algorithm will proceed by joining the current intermediate logical relation to all the referencing tables covered by the reverse references returned from the FR-Index. The alternation between forward and reverse chase rounds will then continue, until a forward chase round is reached, where there are no further foreign keys to chase. The aggressive chase algorithm will then stop.

Algorithm 1 AggressiveChase(A: input table, FR-Index)

```
1:  $\mathbb{A}$ = $A$ {initialize logical relation $\mathbb{A}$ with input table $A$}
2:  $F$ = set of forward references outgoing from $\mathbb{A}$
3:  $R$ = set of reverse references incoming to $\mathbb{A}$
4:  while $F$ is not empty do
5:      {forward chase}
6:      for $f$ in $F$ (where $f$ is a forward reference of the form: $\mathbb{A}.a_i \rightarrow B.b_j$) do
7:          $\mathbb{A}$ = output of joining $\mathbb{A}$ with $B$
8:      end for
9:      {reverse chase}
10:     for $f$ in $F$ do
11:         if an entry $(f, r)$ exists in the FR-Index (where $r$ is a reverse reference of the form: $\mathbb{A}.a_i \leftarrow C.c_k$ then
12:             $\mathbb{A}$ = output of joining $\mathbb{A}$ with $C$
13:         end if
14:     end for
15:     {update forward and reverse references}
16:     $F$ = set of forward references outgoing from $\mathbb{A}$
17:     $R$ = set of reverse references incoming to $\mathbb{A}$
18: end while
19: return $\mathbb{A}$
```

Note that with the above algorithm, the path from one table to another in the directed tree of the generated logical relation can contain one or more reverse edges, and hence the two tables may not have a common ancestor. Therefore, the aggressive chase indeed captures the CAF-m-m relationships as long as they are deemed interesting according to the query log.

4.3 Resolving Correspondence Conflicts

Once we have constructed all the logical relations in both schemas, the next step is to construct the mappings between them. However, we explained in Section 3.3 that the correspondences between the attributes of the two logical relations may not be conflict-free. In this section, we explain the process we follow in U-MAP to resolve these conflicts, and hence find the most meaningful mapping in such situations. This is in contrast to simply generating mappings for all the possible resolutions of the correspondence conflicts.

Problem Formulation: We will now formulate the problem of conflict resolution as follows. Given two logical relations $\mathbb{A}$ and $\mathbb{B}$ with attributes $\{a_1, a_2, \ldots, a_I\}$ and $\{b_1, b_2, \ldots, b_J\}$ respectively, and a set of correspondences in the form of $(a_i, b_j), i \in [1, I], j \in [1, J]$, a pair of correspondences are said to be *conflicting* if they were in the form of $\langle(a_i, b_{j_1}), (a_i, b_{j_2})\rangle$ or $\langle(a_{i_1}, b_j), (a_{i_2}, b_j)\rangle$. In these two cases, the attributes b_{j_1}, b_{j_2}, a_{i_1}, and a_{i_2} are said to be *conflicting attributes*.

Moreover, unlike the attributes of regular schema tables, each attribute in a logical relation is associated with what we refer to as a *logical attribute path*, which specifies how this attribute was reached when constructing the logical relation. Hence, the same attribute from the same table may appear multiple times in the logical relation, but each time with a different associated path. It can be formally defined as follows.

Definition 3: Logical Attribute Path ($LAP(a)$): *A logical attribute path is a sequence of K join predicates connecting an initial table A_0 to an attribute a in some table A_K, where $K \geq 0$.*

Consider that a_{i_k} denotes the primary key of a table A_k, and that $a_{i'_k}$ denotes a foreign key in $A_k, k \in [0, K]$. Thus, if all the join predicates in $LAP(a)$ represent forward references, then $LAP(a)$ is given as follows.

$LAP(a) =$ "$(A_0.a_{i'_0} = A_1.a_{i_1}), (A_1.a_{i'_1} = A_2.a_{i_2}), \ldots,$ $(A_{K-1}.a_{i'_{K-1}} = A_K.a_{i_K}), A_K.a$".

Otherwise, if the join predicates in $LAP(a)$ include opposing references, then $LAP(a)$ is given as follows.

$LAP(a) =$ "$(A_0.a_{i'_0} = A_1.a_{i_1}), \ldots, (A_{k-1}.a_{i'_{k-1}} = A_k.a_{i_k}),$ $(A_k.a_{i_k} = A_{k+1}.a_{i'_{k+1}}), \ldots, A_K.a$".

In view of the above discussion, the conflict resolution problem is to search the space of all possible *conflict-free* mappings between the attributes of $\mathbb{A}$ and $\mathbb{B}$, and find the mapping that is most semantically meaningful in terms of how the attributes (with their associated paths) are matched on both sides.

Example 10: Figure 6 shows some conflicting correspondences between the attributes of the X_Order_Line and Y_Order_Line logical relations. In particular, it focuses on those conflicts between the "city" attributes on both sides. Each attribute is shown along with its path in the figure. (Note that the table names in the paths are abbreviated for space limitation.) □

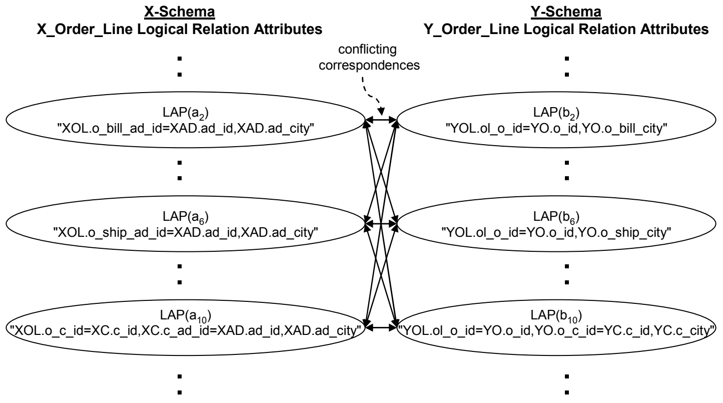

X-Schema
X_Order_Line Logical Relation Attributes
Y-Schema
Y_Order_Line Logical Relation Attributes
conflicting correspondences
LAP(a2)
"XOL.o_bill_ad_id=XAD.ad_id,XAD.ad_city"
LAP(b2)
"YOL.ol_o_id=YO.o_id,YO.o_bill_city"
LAP(a6)
"XOL.o_ship_ad_id=XAD.ad_id,XAD.ad_city"
LAP(b6)
"YOL.ol_o_id=YO.o_id,YO.o_ship_city"
LAP(a10)
"XOL.o_c_id=XC.c_id,XC.c_ad_id=XAD.ad_id,XAD.ad_city"
LAP(b10)
"YOL.ol_o_id=YO.o_id,YO.o_c_id=YC.c_id,YC.c_city"

Figure 6: Correspondence conflicts for all "city" attributes when mapping the logical relations for X_Order_Line and Y_Order_Line.

Solution Overview: Our proposed solution is centered around two ideas, which translate into two consecutive stages.

- *Attribute Grouping:* The purpose of the first stage is to reduce the size of the search space by excluding as many illogical mappings as possible, and hence ensure that they will not be erroneously selected in the following stage. This is achieved by first dividing the attribute sets of each of $\mathbb{A}$ and $\mathbb{B}$ into groups of semantically-related attributes. Then, the only mappings considered are those where no two attributes in the same group on one side are mapped to two attributes in different groups on the other side.

- *Usage-based Conflict Resolution:* The second stage actually selects the most meaningful mapping. Its main idea is to match a pair of attributes from $\mathbb{A}$ and $\mathbb{B}$ only when their *usage patterns* (as reflected in their respective query logs) seem more compatible compared to other alternative matches.

4.3.1 Attribute Grouping

The intuition behind this stage is based on the observation that conflicting attributes in a logical relation result from using the same *concepts* in different *contexts*. For example, as we mentioned in Example

7, the logical relations for X_Order_Line and Y_Order_Line both contain multiple versions of address attributes (i.e., street, city, state, . . . , etc), where each version corresponds to a different context (e.g., billing address, shipping address, customer address, . . . , etc).

Formally speaking, the full context of an attribute a in a logical relation is given by $LAP(a)$. Consequently, an attribute a is said to have a more similar context to attribute b compared to attribute c if $LAP(a)$ and $LAP(b)$ share a longer prefix compared to $LAP(a)$ and $LAP(c)$.

The idea of grouping is to ensure that attributes having similar contexts are grouped together. This way, we can match whole groups of attributes across the source and target logical relations. This approach is in contrast to matching individual attributes, which may lead to matching two attributes in the same context on one side to two attributes in two different contexts on the other side, which would clearly result in a wrong mapping.

Goal: Our goal is to generate attribute groups that satisfy the following properties: (1) All attributes in the same group must be non-conflicting with one another; (2) Given any two attributes a and b in two different groups G_1 and G_2 respectively, the common prefix of $LAP(a)$ and $LAP(b)$ cannot be longer than the common prefix of $LAP(a)$ and $LAP(c)$, for any other attribute c in G_1; and (3) The number of generated groups must be minimal given that they satisfy the above two properties.

The first property ensures that matching attributes from one group on one side to another group on the other side will not involve any further conflict resolution. The second property ensures that attributes within the same group are closer to each other, in terms of their contexts, compared to attributes in other groups. Finally, the third property ensures that groups satisfying the first two properties are not unnecessarily split into smaller subgroups, which in turn ensures that the space of possible mappings to be explored is kept to a minimum.

Attribute Grouping Algorithm: The input to our grouping algorithm is the list of all conflicting attributes in the logical relation, along with their paths. The output is a list of groups of attributes satisfying the properties described above. The algorithm operates as follows. For each conflicting attribute a, $LAP(a)$ is inserted into a *trie*, or a prefix tree. Briefly, a trie is a tree structure for efficiently storing strings (logical attribute paths in our case). Each leaf node stores one of the inserted paths. Internal nodes store the common prefix for all the paths stored in its descendant nodes. Each node also maintains pointers to the set of attributes represented by its descendants (and hence are considered to be represented by that node as well).

The populated trie is then used to partition the attributes into groups as follows. We perform a depth first search starting from the root node. For each visited node, if some attributes in the set it represents conflict with one another, then this set cannot be considered as a group, and its descendant nodes are visited in the standard depth first search order (i.e., the set of attributes must be further split). Conversely, if the set of attributes in the visited node are not conflicting with one another, then the set will be considered as one of the output groups, and none of node's descendants will be visited (i.e., no further splitting is performed). The pseudocode for this algorithm has been omitted for space limitations.

Example 11: Figure 7 shows how the conflicting attributes are grouped in the Y_Order_Line logical relation. The list of such conflicting attributes, along with their paths, is given in Figure 7(a). After all the paths are inserted into a trie, the populated trie will then look as shown in Figure 7(b). The output groups in this case are the sets of attributes associated with each one of the second-level nodes (highlighted nodes). This is because they are all conflict-free, as opposed to the set of attributes associated with the root node (e.g., b_2, b_6, and b_{10} in the root node are all "city" attributes, and hence they conflict with one another).□

Note that the attribute grouping stage is optional, in the sense that conflict resolution (described next) can occur even if the attributes were not grouped. However, we found in our experiments that the grouping constraints introduced in this stage indeed help in protecting the next stage from making wrong decisions, and hence improve the overall quality of the generated mappings.

4.3.2 Usage-based Conflict Resolution

The goal of this stage is to match the groups generated during the attribute grouping stage between the source and target logical relations. (If no grouping was performed, then each individual attribute will be considered as a group.) The straightforward approach would be to match the common path prefixes defining the shared context among the attributes in each group.

For example, the common path prefixes for the attributes in the two groups representing the billing and shipping addresses in the logical relation of X_Order_Line are "(X_Order_Line.ol_bill_ad_id = X_Address.ad_id)," and "(X_Order_Line.ol_ship_ad_id = X_Address.ad_id)," respectively. Similarly, for the logical relation of Y_Order, the common prefixes for the attributes in the two corresponding groups are "Y_Order.o_bill_" and "Y_Order.o_ship_" respectively.

Clearly, in this case, measuring the textual similarity between the prefixes on each side can lead us to the correct matching. For instance, we can break up each prefix into a set of text fragments and then measure the Jaccard similarity coefficient between the resulting sets. However, this approach will not be useful if the attribute naming was not similar in both schemas – for example, if o_bill_city and o_ship_city are instead named o_city1 and o_city2. An even more extreme example is when the two schemas are in two different languages.

For this reason, we do not assume that textual similarity between the prefixes always exists, and instead we rely on a different source of information. In particular, we assume that a group of attributes representing certain concepts in a given context are expected to exhibit different *usage patterns* compared to another group of attributes representing the same concepts but in a different context. For example, the usage of the billing address attributes can be different from the usage of the shipping address attributes as reflected in the query log.

Usage-Based Schema Matching: The idea of relying on the usage information for matching attributes has already been proposed in [10]. However, the focus in [10] was on full-fledged schema matching. In our case, we need to match the attributes of two logical relations rather than two schemas. Moreover, the fact that the same schema attribute may occur multiple times in a logical relation poses new challenges which were not addressed in [10].

The technique proposed in [10] can be summarized as follows. It relies on collecting statistical information from the query logs for each attribute, or each pair of attributes (e.g., their co-occurrence frequency in the select clause, or in the select and where clauses respectively, and so on). Then those collected statistics are compared across the two schemas. The set of attribute correspondences, which result in the highest similarity for the collected statistics in both schemas (based on a scoring function described in [10]), is returned as the schema matching output.

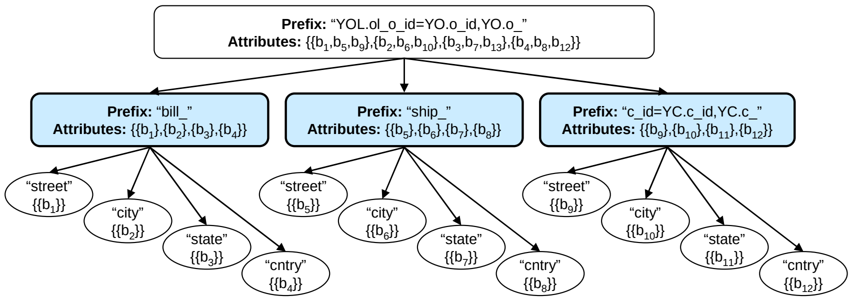
A New Matching Problem: In our problem setting, however, we found that before we can match the attributes of logical relations as described above, a totally separate *matching problem* needs to be solved first – one that requires a new strategy too. In particular, the statistics collected from the query logs for those attributes which appear multiple times in the logical relation must now be split across the different versions of each such attribute. For this purpose, each occurrence of this attribute in the query log must be *matched* to one of the versions in the logical relation. This way we can correctly account for the statistics pertaining to each individual version.

Matching in this scenario will be *context-based* – if an attribute occurs in a query in a *context* similar to that of a given version of the same attribute in the logical relation, then they are matched together. We call this problem *attribute context matching*. To understand how we address this problem, we will explain next how we specify attribute contexts when they occur in logical relations and in queries, and how we actually match them.

Attribute Contexts in Logical Relations: As we mentioned earlier, the context of an attribute a is generally determined based on its path, $LAP(a)$. This path, however, can sometimes be too long, and hence represents a very specific context, which will unlikely match with the contexts found in the query log. In principle, we only need the context

P(b_1):	"YOL.ol_o_id=YO.o_id,YO.o_bill_street"
P(b_2):	"YOL.ol_o_id=YO.o_id,YO.o_bill_city"
P(b_3):	"YOL.ol_o_id=YO.o_id,YO.o_bill_state"
P(b_4):	"YOL.ol_o_id=YO.o_id,YO.o_bill_cntry"
P(b_5):	"YOL.ol_o_id=YO.o_id,YO.o_ship_street"
P(b_6):	"YOL.ol_o_id=YO.o_id,YO.o_ship_city"
P(b_7):	"YOL.ol_o_id=YO.o_id,YO.o_ship_state"
P(b_8):	"YOL.ol_o_id=YO.o_id,YO.o_ship_cntry"
P(b_9):	"YOL.ol_o_id=YO.o_id,YO.o_c_id=YC.c_id,YC.c_street"
P(b_{10}):	"YOL.ol_o_id=YO.o_id,YO.o_c_id=YC.c_id,YC.c_city"
P(b_{11}):	"YOL.ol_o_id=YO.o_id,YO.o_c_id=YC.c_id,YC.c_state"
P(b_{12}):	"YOL.ol_o_id=YO.o_id,YO.o_c_id=YC.c_id,YC.c_cntry"

(a) Paths of conflicting attributes in the X_Order_Line logical relation


Prefix: "YOL.ol_o_id=YO.o_id,YO.o_"
Attributes: {{b_1,b_5,b_9},{b_2,b_6,b_{10}},{b_3,b_7,b_{13}},{b_4,b_8,b_{12}}}
Prefix: "bill_"
Attributes: {{b_1},{b_2},{b_3},{b_4}}
Prefix: "ship_"
Attributes: {{b_5},{b_6},{b_7},{b_8}}
Prefix: "c_id=YC.c_id,YC.c_"
Attributes: {{b_9},{b_{10}},{b_{11}},{b_{12}}}
"street" {{b_1}}
"city" {{b_2}}
"state" {{b_3}}
"cntry" {{b_4}}
"street" {{b_5}}
"city" {{b_6}}
"state" {{b_7}}
"cntry" {{b_8}}
"street" {{b_9}}
"city" {{b_{10}}}
"state" {{b_{11}}}
"cntry" {{b_{12}}}

(b) The trie built for all the paths in (a)

Figure 7: Attribute grouping for the Y_Order_Line logical relation. The output groups of attributes are highlighted.

of an attribute to be specific enough to *distinguish* it from the contexts of the other versions of the same attribute in the logical relation.

Let us consider that attributes $\{a_1, a_2, \ldots, a_n\}$ in a logical relation $\mathbb{A}$ are different versions of the same attribute a in some table A. The paths of all attributes $a_i, i \in [1, n]$, will have the same beginning (the root of the logical relation) and ending (attribute a). Only the internal components in these paths are what distinguish them from one another. Hence, the least specific path for an attribute in a logical relation which can still distinguish it from all other attributes in the same logical relation is given by the following definition.

Definition 4: Minimal Distinguishing Logical Attribute Path ($MDLAP(a)$): *Given an attribute a in a logical relation $\mathbb{A}$, the minimal distinguishing logical attribute path of a, or $MDLAP(a)$, is defined as the minimal* suffix *of $LAP(a)$, such that there does not exist an attribute b in $\mathbb{A}$, where $MDLAP(a)$ is also a suffix of $LAP(b)$.*

In conclusion, for the purpose of attribute context matching across logical relations and queries, the context of an attribute a in a logical relation is given by $MDLAP(a)$.

Attribute Contexts in Queries: In order to identify the context of the attributes appearing in a query, we first need to build its query graph as described in Section 4.2. Then, using this graph, we can compute the path of join predicates leading to the table instance where each attribute belongs. This is achieved by starting form the node in the graph representing the attribute's table instance, and then visiting the chain of its ancestor nodes in the graph. The format of this path for an attribute a will be identical to that of $LAP(a)$ given in Definition 3. However, we will denote the path extracted from a query by $QAP(a)$ – short for *query attribute path*.

The main difference between a query and a logical relation is that the former is represented by a general directed graph, while the latter is represented by a rooted directed tree. As a result, each attribute in a logical relation will only have one path starting from the root. But in theory, an attribute in a query may have multiple paths. (In our running example, for instance, a query may request all addresses which were used both as billing and shipping addresses for the same order. In this case, each address attribute will have two paths corresponding to its billing and shipping roles.) Therefore, for the sake of generality, an attribute in a query is allowed to have multiple contexts depending on how many paths are discovered for it in the query.

Context Matching: To decide whether an attribute a_q appearing in a query is actually an occurrence of an attribute a_l appearing in a logical relation, we need to first match their recognized contexts (or paths). For this purpose, we use the following strategy. For each path $QAP_i(a_q)$ associated with a_q, if $MDLAP(a_l)$ is a *suffix* of $QAP(a_q)$, then a_q is considered to be an occurrence of a_l in the query.

This criterion guarantees that given the context of a_q in the query, a_q will be unambiguously matched to a single attribute, a_l, in the logical relation. This is because the context of a_l is already known to be unique in the logical relation. And since the context for a_q must be *as specific as, or even more specific than*, that of a_l, then a_q cannot be matched to any other attribute in the logical relation given that context. In case a_q has more than one context (as discussed earlier), then it may be matched to more than one attribute in the logical relation – one for each such context.

By solving the context matching problem as described above, we can then directly apply the usage-based matching technique described in [10] to resolve the correspondence conflicts across the two logical relations, and thereby find the most meaningful conflict-free mapping.

5. EVALUATION

To evaluate our approach, we have conducted experiments in two main scenarios: the bookstores scenario, and the life sciences scenario, as will be described next. Unfortunately, research benchmarks such as STBenchmark [2] were not suitable for our experiments because they did not include query log information, which is needed for the operation of U-MAP.

5.1 Bookstores Scenario

In this section, we describe an experimental study whose goal is to measure the effect of each of the new features introduced in the U-MAP system on the quality of the generated mappings, in comparison with the classical RIC-based approach. We also study the performance of the mapping generation process.

Dataset: We have considered schemas from the bookstores domain, based on the industry-standard TPC-W benchmark [27]. TPC-W gives the specifications for building an online bookstore, including the database schema, the data stored in it, and the system workload. In order to simulate the presence of two different bookstore systems, we created two perturbed versions of the specified schema, very similar to those shown in Figure 1. The schemas used in the experiments, however, contain more tables (e.g. related to the shopping cart information), and more attributes (e.g., book subjects and stock levels). In summary, the two schemas used in our experiments contain a total of 18 tables and 82 attributes, covering 30 different correspondences.

In order to generate the query logs, we used the Wisconsin implementation of TPC-W [28], and ran the bookstore system under the different workload mixes specified by the benchmark. In particular, there are three such mixes: a browsing mix, a shopping mix, and an ordering mix, where the read-only Web interactions constitute 95%, 80%, and 50% respectively. By running the system under each mix, we could generate two corresponding query logs for the source and target schemas. The query logs were generated by re-writing the original queries so that they conform to each of the two schemas. To ensure heterogeneity, we assumed that the two schemas were associated with the two most different query logs (browsing for the source, and ordering for the target). A single run of each workload mix was performed by running 30 emulated browsers, which simultaneously submit requests to the bookstore system for about 3 hours. These runs resulted in query logs containing 5,231 and 11,424 queries for the source and target schemas respectively.

It is worth noting that TPC-W is focused on a single use case for the bookstore database, namely that of the bookstore's online customers. Hence, it does not account for other typical use cases of the same database, such as the queries made by the shipping department, the accounting department, etc. In the query logs we used, we tried to account for those use cases as well, while making the smallest possible changes to the original query logs. In particular, we assumed that queries requesting the information of specific orders are not only coming from the online customers. But they might also be coming from the shipping department (which will normally be interested in the order's shipping address), and the accounting department (which, in contrast, will be interested in the billing address). We also assumed that the requests of the accounting department are generally more frequent than those of the shipping department. The rationale is that, unlike the accounting department, the shipping department is only concerned with an order until it gets shipped.

Methodology: We implemented the U-MAP system such that all of its new features can either be enabled or disabled. In particular, we refer to the following four features: merging sibling relations (MSR), aggressive chase (AC), attribute grouping (AG), and usage-based conflict resolution (UCR). To indicate that a given feature is disabled, we prefix it with "!" (e.g., "!AC" implies that aggressive chase is not used). Clearly, if all four features are disabled, then U-MAP will behave similar to a classical RIC-based mapping system.

In the experiments, we run U-MAP using its 16 possible configurations. For each run, we measure the precision and recall by comparing the generated mappings against a set of manually-created ground truth mappings. Both the generated and ground truth mappings represent the set of *minimized* mappings (i.e., they are the outcome of step 3 described in Section 2.3.1). For our experimental setting, we found the total number of ground truth mappings to be 15. Moreover, we also measured the processing time given the different configurations of U-MAP, averaged over 20 independent runs each. The experiments were conducted on a 4-core Windows 7 machine with 2.66 GHz core speed and 6GB of RAM.

Unlike the RIC-based approach, we did not directly compare U-MAP to the semantic approach in our experiments. Beside the fact that no implementation for the semantic approach was available to us, we were also more interested in the general case, where a conceptual model does not necessarily exist. (In fact, no conceptual models were available for the bookstore schemas we used.) Moreover, we could show in Section 3, that even in the presence of the conceptual models, the semantic approach will neither help in discovering some of the meaningful logical relations, nor will it help in resolving correspondence conflicts.

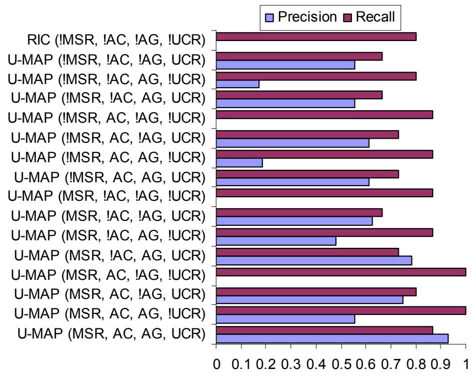


Figure 8: Precision and recall for the different U-MAP configurations, including the classical RIC configuration.

5.1.1 Quality Study

Overview: Figure 8 shows the precision and recall measurements for each of the 16 configurations of U-MAP. Figure 8 shows that the best configuration is where all the new features are enabled. This configuration resulted in a precision and recall of 0.93 and 0.87 respectively. In particular, exactly 14 mappings were generated with one false positive and two false negatives.

The false positive is because of generating an incorrect mapping between the logical relations of X_Buyer (the super relation of X_Customer and X_Distributer) and Y_Order, which incorrectly maps the customer address attributes in X_Buyer to the shipping address attributes in Y_Order.

The reason for this glitch is that the highest-score mapping (based on the scoring function described in [10]) between the two logical relations is discarded during the minimization step despite being the correct mapping. However, we have set our implementation such that if the highest-score mapping is discarded, it gets *replaced* with the second highest-score mapping, and so on. We found that this replacement strategy generally leads to better results, compared to no-replacement. In some cases however (as in this case), it results in generating incorrect mappings. The choice of replacement versus no-replacement is controlled by a configurable parameter in U-MAP.

The two false negatives are because we assumed that there are *two* ground truth mappings between the logical relations of X_Address and Y_Order, and similarly between those of X_Country and Y_Order. These mappings correspond to mapping the address information once to the billing address in Y_Order, and once to the shipping address. In U-MAP, however, we only generate the highest-score mapping between each pair of logical relations, and hence the second ground truth mapping (which scored less) was missed.

Another observation from Figure 8 is that the precision is almost zero whenever both AG and UCR are both disabled (including the RIC configuration). The reason is that in all of these cases, the number of incorrectly generated mappings is exponentially large, because all the possible mappings for all possible conflict resolutions are generated without any grouping constraints.

It is also worth mentioning that in all of our experiments, executing each of MSR, AC, and AG has always been accurate; i.e., overlapping sibling relations were correctly detected and merged whenever MSR was enabled, meaningful reverse references were correctly detected and chased whenever AC was enabled, and the attributes were correctly grouped whenever AG was enabled. In what follows, we study the impact of each individual new feature on the quality of the generated mappings.

Effect of Merging Sibling Relations: To assess the impact of MSR, we will compare the best configuration, where all new features are enabled, to that where all are enabled except for MSR. The precision and recall dropped from 0.93 and 0.87 for the former configuration to 0.56 and 0.67 respectively for the latter configuration. In particular, in the latter configuration, 18 mappings are generated with 8 false positives and 5 false negatives (compared to 1 and 2 respectively for the former).

Clearly, the false positives in this case are mostly because of generating mappings involving X_Customer and X_Distributor, while the false negatives are because of *not* generating the mappings involving X_Buyer.

Effect of Aggressive Chase: The configuration where all new features are enabled except for AC results in a precision and recall of 0.79 and 0.73 respectively. In this case, exactly 14 mappings are generated, similar to the best configuration. However, unlike the best configuration, 3 false positives and 4 false negatives occur. As expected, the additional false positives are due to the generation of mappings involving the chase of Y_Order_Line without including the author information. Also, the additional false negatives are attributed to *not* generating the mappings involving the chase of Y_Order_Line with the author information included.

Effect of Attribute Grouping: When using the configuration where all the new features are enabled except for AG, we find that the precision and recall drop to 0.75 and 0.8 compared to 0.93 and 0.87 respectively, when all features are enabled, including AG. These figures correspond to 16 generated mappings, with 4 false positives and 3 false negatives.

In the absence of grouping constraints, the conflict resolution step will have to search for the best mapping within a large number of possible conflict resolutions. For this reason, it becomes more prone to errors. And indeed, this is what we observed in this experiment. Both the false positives and false negatives occurred, precisely, because some of the generated mappings incorrectly mapped address attributes from the same group in the X-Schema to address attributes in two different groups in the Y-Schema. For example, the mapping between the logical relations of X_Address and Y_Order mapped the street, city, and country attributes in the X-Schema to their corresponding *customer* address attributes in the Y-Schema. Conversely, however, the city attribute in the X-Schema was mapped to the *shipping* city in the Y-Schema.

Effect of Conflict Resolution: Figure 8 shows that the configuration where all the new features are enabled except for UCR, achieves a recall of 1. This is expected because with no conflict resolution, the mappings corresponding to all possible resolutions are generated, including all the desirable ones. The precision however drops from 0.93 to 0.55, compared to the best configuration. Clearly, the reason is that many incorrect mappings are also generated in this case. In particular, 27 mappings are generated for this configuration. This number goes up to 2763 generated mappings for the configuration where both AG and UCR are disabled (i.e., with no grouping constraints).

5.1.2 Performance Study

Although the mapping generation process is typically a one-time offline process, and hence the computational efficiency comes at a lower priority compared to the quality of the generated mappings – we still, however, report here the performance results in our experiments to demonstrate the practicality of U-MAP.

In terms of the processing time, the configuration where all the new features were enabled took about 12.5 seconds to generate all the mappings. The remaining configurations ranged from 0.3 to 16 seconds, except for the four configurations where both MSR and AG were disabled, which took several hours each. These four cases required trying all the possible ways to match 16 address attributes on the source side with 12 address attributes on the target side. Additionally, the two target attributes c_uname and c_passwd had 4 different ways to be matched to the source attributes: c_uname, c_passwd, d_uname, and d_passwd. This striking difference in the processing time underscores the value of the "merging sibling relations" and "attribute grouping" features in U-MAP.

5.2 Life Sciences Scenario

To further assess the validity of our approach, we report in this section our experience with a larger-scale real-world data set from the life sciences domain. In this scenario, we only focus on one system (as opposed to two parallel systems serving as the source and the target). This is primarily because it is the one where information on both its schema and its queries were available to us. While measuring the accuracy and performance of the complete mapping process is not possible in this case, studying this particular system was still quite helpful in showing that: (1) the issues U-MAP is addressing are indeed likely to occur in real-world schemas, and (2) the usage information can help us better understand the semantics of a database schema, and hence perform a better job in mapping it to other schemas.

In particular, we could obtain the schema of a functional genomics database [4] from its authors. The database keeps track of millions of biological measurements collected for over 60,000 tissue samples of the Arabidopsis plant, including both mutant and normal samples. It has been in operation for over four years serving biologists worldwide, who are interested in this particular plant.

The schema, whose layout is shown on the right side of Figure 9, contains 35 tables and 273 attributes. We could not get access to the full-fledged query log for this database. However, we could obtain a list of 73 *query templates* (or parameterized queries), which are embedded in the source code of the application that runs on top of it. Although the frequency information for these queries was missing, they were still an excellent source of information for the usage patterns of the different tables and attributes in the schema. Table 1 summarizes some statistics, which characterize the schema and queries we consider in this scenario. We now discuss how such characteristics can highlight the value of our newly introduced features in U-MAP.

Sibling Relations: While this schema does contain 6 sibling relations (two groups of three relations each), none of them is overlapping. However, the people table (magnified in the left side of Figure 9) in fact represents different types of people: customers requesting the analysis, providers of the samples, specialists who plant and harvest the samples, and analysts who collect and analyze the measurements. Therefore, a very likely alternative design would have a separate table assigned to each type of people. Moreover, the roles of these people often overlap, causing such tables to become overlapping sibling relations. For example, the analyst can also be the person who plants and harvests the sample, and so on. In this case, we can only avoid having duplicate records in the target database, if we appropriately merge those overlapping sibling relations prior to mapping generation.

Table 1: Statistics from the life sciences scenario.

Schema and Queries	
# tables	35
# attributes	273
# query templates	73
Sibling Relations	
# sibling relations	6
# overlapping sibling relations	0
Aggressive Chase	
# pairs of opposite edges	106
# interesting pairs of opposing references	3
# logical relations using opposing references	7
Conflict Resolution	
# logical relations with conflicting attributes	16
# conflicting attributes in all logical relations	1267

Aggressive Chase: We analyzed all the referential integrity constraints in the schema and found that the overall number of pairs of opposing references is 106. Moreover, by analyzing the queries associated with the schema, we discovered that 3 out of those 106 pairs are interesting. An interesting pair is one that is used to join three tables in some query, and hence suggests that their attributes should be associated. When applying the aggressive chase, the discovered interesting pairs are used in constructing 7 different logical relations. To appreciate the significance of this finding on the quality of the generated mappings (between this schema and any other schema), we note that all mappings involving any of the 7 aforementioned logical relations will be missing important associations, and hence considered incorrect, unless the aggressive chase is applied.

Conflict Resolution: The schema in Figure 9 contains numerous cases where one table refers to another table multiple times, and for a different purpose each time (similar to the billing and shipping addresses in the bookstores scenario). For example, the table called trayinfo (shown in the left side of Figure 9, and which covers information about each experiment for planting sample tissues) references the people table three times to track the persons who planted, harvested, and analyzed the results for the tissue sample. It also references a table called typedet (not magnified in the figure, however, it is a general-purpose table used to maintain the possible *types* of different concepts in the schema) twice – once to track the type of the tissue being planted, and another for the growing medium type.

As a result of these and many other similar cases in the schema, 16 of the constructed logical relations were found to contain a total of 1267 conflicting attributes(!). Considering the number of mappings these logical relations will be involved in, and the number of correspondence conflicts they will generate, we can immediately realize how impractical it is to leave the conflict resolution task for the user to perform manually in complex real-world scenarios.

6. RELATED WORK

The problem of mapping generation has received a lot of attention, especially in the last decade(e.g., [3, 6, 9, 11, 13, 21, 23]). In the context of the Clio project, several techniques were proposed for mapping discovery, first for relational data sources [21], and then for XML data sources [23]. Later, the notion of "nested mappings" was introduced in [13], which showed how we can combine multiple mappings together into a single nested mapping.

Besides Clio, other research efforts were made for the discovery of complex mappings beyond the simple attribute correspondences. The semantic approach [3] described in Section 2 is an example. As we already mentioned, a key drawback in this approach is that the conceptual models, which it relies upon, are not often present in real-world scenarios. iMap [9] focuses on finding complex relations between attributes in both schemas such as price=rate*(1+tax). Bohannon et al. [6] introduced *contextual schema matching*, in which a match between a pair of attributes is valid only when certain conditions are met in the data instances.

None of the above techniques, however, relied on query logs. Only

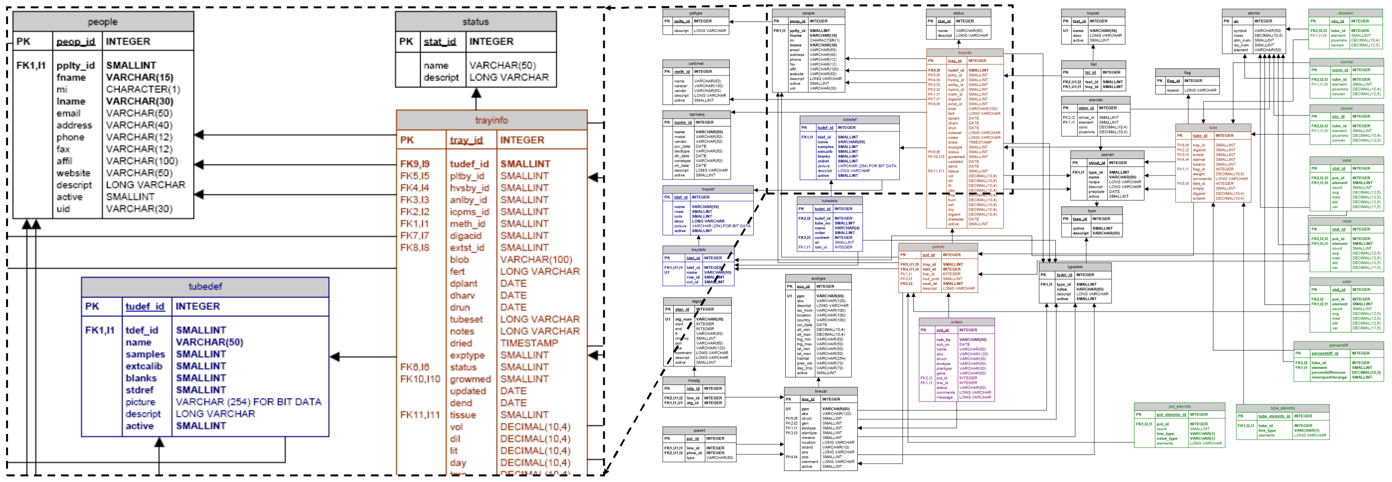


Figure 9: The life sciences schema (right side) with the dashed rectangle containing the people and trayinfo tables magnified (left side).

in a recent overview paper on the Clio project [11], it was briefly mentioned that query logs can be used to find associations between attributes. The paper essentially refers to non-RIC-based associations like having the first two letters in the course ID for instance equal to the department ID. This idea is complementary to the techniques we propose in this paper.

More recently, several works have focused on the post mapping generation phase, where the generated mappings are re-written to obtain a new set of mappings with desirable properties. For example, [26, 19] are two independent approaches for re-writing mappings to generate SQL scripts capable of computing what is known as the *core solution* for a data exchange problem [12]. Also, [14] shows how schema mappings can be normalized in the same spirit in which relational schemas are normalized. The technique proposed in [18] addresses the situation where key constraints and functional dependencies (egds) hold on the target. Given a mapping scenario with s-t tgds along with target egds, a best-effort algorithm is given to re-write this scenario into one with no egds, which can be efficiently executed. However, we explained in Section 3.1 that this technique can handle the problem of merging sibling relations, but not when the key attributes are not mapped across the source and the target. Hence, the above works either do not consider the same problems we address in this paper, or only handle special cases for some of them. It would be useful, however, to have them incorporated into the "minimization step" in Figure 3.

As described in Section 3.3, Muse [1] and +Spicy [20] attempt to solve the problem of unresolved correspondences. However, their solutions involve substantial manual labor and can be tedious for large and complex schemas. Our work is also related to the usage-based schema matching technique presented in [10] in the sense that they both exploit the query logs. However, as we explained in Section 4.3, this work is only focused on the generation of simple attribute correspondences across two schemas.

7. CONCLUSION

We have presented U-MAP, a schema mapping system, which emphasizes the value of the usage information in the query logs to address several unresolved problems in the schema mapping area. We also verified the effectiveness and efficiency of U-MAP by running it on realistic databases from the retail and life sciences domains. Our results suggest that the best quality for the generated mappings is obtained when all the new features in U-MAP are turned on.

8. REFERENCES

[1] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, 2008.

[2] B. Alexe, W. C. Tan, and Y. Velegrakis. Stbenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

[3] Y. An, A. Borgida, R. J. Miller, and J. Mylopoulos. A semantic approach to discovering schema mapping expressions. In *ICDE*, 2007.

[4] I. Baxter et al. PIIMS: An Integrated Functional Genomics Platform. *Plant Physiology*, 143:600–611, 2007.

[5] P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[6] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting context into schema matching. In *VLDB*, 2006.

[7] A. Bonifati et al. Heptox: Marrying xml and hetergeneity in your p2p databases. In *VLDB*, 2005.

[8] L. Cabibbo. On keys, foreign keys and nullable attributes in relational mapping systems. In *EDBT*, pages 263–274, 2009.

[9] R. Dhamankar et al. imap: Discovering complex mappings between database schemas. In *SIGMOD*, 2004.

[10] H. Elmeleegy, M. Ouzzani, and A. Elmagarmid. Usage-based schema matching. In *ICDE*, 2008.

[11] R. Fagin et al. Clio: Schema mapping creation and data exchange. *Conceptual Modeling: Foundations and Applications*, pages 198–236, 2009.

[12] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: Getting to the core. *ACM TODS*, 30(1):174–210, 2005.

[13] A. Fuxman, M. A. Hernandez, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Netsed mappings: Schema mapping reloaded. In *VLDB*, 2006.

[14] G. Gottlob, R. Pichler, and V. Savenkov. Normalization and optimization of schema mappings. *PVLDB*, 2(1):1102–1113, 2009.

[15] A. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *TKDE*, 16(7):787–798, 2004.

[16] Z. Kedad and M. Bouzeghoub. Discovering view expressions from a multi-source information system. In *CoopIS*, 1999.

[17] D. Maier and A. Mendelzon. Testing implications of data dependencies. *ACM TODS*, 4:455–469, 1979.

[18] B. Marnette, G. Mecca, and P. Papotti. Scalable data exchange with functional dependencies. *PVLDB*, 3(1):105–116, 2010.

[19] G. Mecca, P. Papotti, and S. Raunich. Core schema mappings. In *SIGMOD*, 2009.

[20] G. Mecca, P. Papotti, S. Raunich, and M. Buoncristiano. Concise and expressive mappings with +spicy. *PVLDB*, 2(2):1582–1585, 2009.

[21] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, 2000.

[22] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, 1998.

[23] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating web data. In *VLDB*, 2002.

[24] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[25] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *SIGMOD*, 1975.

[26] B. ten Cate et al. Laconic schema mappings: Computing the core with sql queries. *PVLDB*, 2(1):1006–1017, 2009.

[27] The TPC-W benchmark. http://tpc.org/tpcw.

[28] http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[29] C. Yu and L. Popa. Constraint-based xml rewriting for data integration. In *SIGMOD*, 2004.

[30] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, 2005.