MapMerge: Correlating Independent Schema Mappings

Bogdan Alexe
UC Santa Cruz

Mauricio Hernández
IBM Almaden

Lucian Popa
IBM Almaden

Wang-Chiew Tan
IBM Almaden & UC Santa Cruz

ABSTRACT

One of the main steps towards integration or exchange of data is to design the mappings that describe the (often complex) relationships between the source schemas or formats and the desired target schema. In this paper, we introduce a new operator, called MapMerge, that can be used to correlate multiple, independently designed schema mappings of smaller scope into larger schema mappings. This allows a more modular construction of complex mappings from various types of smaller mappings such as schema correspondences produced by a schema matcher or pre-existing mappings that were designed by either a human user or via mapping tools. In particular, the new operator also enables a new "divide-and-merge" paradigm for mapping creation, where the design is divided (on purpose) into smaller components that are easier to create and understand, and where MapMerge is used to automatically generate a meaningful overall mapping. We describe our MapMerge algorithm and demonstrate the feasibility of our implementation on several real and synthetic mapping scenarios. In our experiments, we make use of a novel similarity measure between two database instances with different schemas that quantifies the preservation of data associations. We show experimentally that MapMerge improves the quality of the schema mappings, by significantly increasing the similarity between the input source instance and the generated target instance.

1. Introduction

Schema mappings are essential building blocks for information integration. One of the main steps in the integration or exchange of data is to design the mappings that describe the desired relationships between the various source schemas or source formats and the target schema. Once the mappings are established, they can be used either to support query answering on the (virtual) target schema, a process that is traditionally called data integration [13], or to physically transform the source data into the target format, a process referred to as data exchange [7]. In this paper, we focus on the data exchange aspect although our mapping generation methods will be equally applicable to derive mappings for data integration.

Many commercial data transformation systems such as Altova Mapforce[1] and Stylus Studio[2] to name a few, as well as research prototypes such as Clio [6] or HePToX [3], include mapping design tools that can be used by a human user to derive the data transformation program between a source schema and a target schema. Most of these tools work in two steps. First, a visual interface is used to solicit all known correspondences of elements between the two schemas from the mapping designer. Such correspondences are usually depicted as arrows between the attributes of the source and target schemas. (See, for example, Figure 1(a)). Sometimes, a schema matching module [19] is used to suggest or derive correspondences. Once the correspondences are established, the system interprets them into an executable script, such as an XQuery or SQL query, which can then transform an instance of the source schema into an instance of the target schema. The generated transformation script is usually close to the desired specification. In most cases, however, portions of the script still need to be refined to obtain the desired specification.

We note that for both Clio and HePToX, the correspondences are first compiled into internal *schema mapping assertions* or *schema mappings* in short, which are high-level, declarative, constraint-like statements [12]. These schema mappings are then compiled into the executable script. One advantage of using schema mappings as an intermediate form is that they are more amenable to the formal study of data exchange and data integration [12], as well as to optimization and automatic reasoning. In fact, the main technique that we introduce in this paper represents a form of automatic reasoning on top of schema mappings.

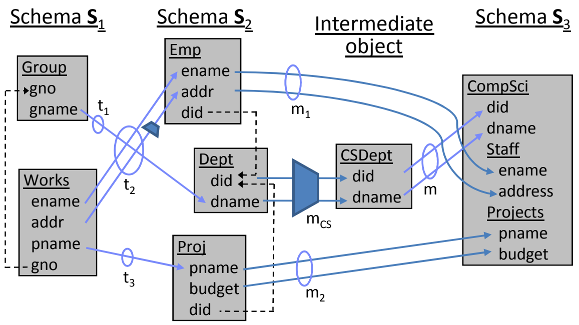
An important drawback of the previously outlined two-step schema mapping design paradigm is that it is hard to retro-fit any pre-existing or user-customized mappings back into the mapping tool, since the mapping tool is based on correspondences. Thus, if the mapping tool is restarted, it will regenerate the same fixed transformation script based on the input correspondences, even though some portions of the transformation task may have already been refined (customized) by the user or may already exist (as part of a previous transformation task, for example).

In this paper, we propose a radically different approach to the design of schema mappings where the mapping tool can take as input arbitrary mapping assertions and not just correspondences. This allows for the modular construction of complex and larger mappings from various types of "smaller" mappings that include schema correspondences but also arbitrary pre-existing or customized mappings. An essential ingredient of this approach is a new operator on schema mappings that we call *MapMerge* and that can be used to automatically correlate the input mappings in a meaningful way.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

[1] www.altova.com

[2] www.stylusstudio.com



(a)

Input mappings from S_1 to S_2:

```
(t1) for g in Group exists d in Dept
     where d.dname = g.gname

(t2) for w in Works, g in Group
       satisfying  w.gno = g.gno, w.addr = "NY"
     exists e in Emp, d in Dept
       where e.did = d.did,
       e.ename = w.ename, e.addr = w.addr,
       d.dname = g.gname

(t3) for w in Works exists p in Proj
     where p.pname = w.pname
```

(b)

Output of MapMerge(S_1, S_2, {t_1, t_2, t_3}):

```
for g in Group exists d in Dept
where d.dname = g.gname, d.did = F[g]

for w in Works, g in Group
  satisfying  w.gno = g.gno, w.addr = "NY"
exists e in Emp
  where e.ename = w.ename, e.addr = w.addr,
        e.did = F[g]

for w in Works, g in Group
  satisfying  w.gno = g.gno
exists p in Proj
  where p.pname = w.pname, p.budget = H1[w],
        p.did = F[g]
```

(c)

Figure 1: (a) A transformation flow from S_1 to S_3. (b) Schema mappings from S_1 to S_2. (c) Output of MapMerge.

1.1 Motivating Example and Overview

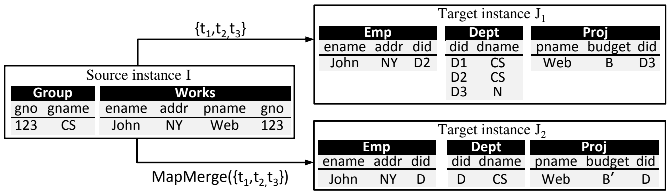
To illustrate the ideas, consider first a mapping scenario between the schemas $\mathbf{S_1}$ and $\mathbf{S_2}$ shown in the left part of Figure 1(a). The goal is data restructuring from two source relations, *Group* and *Works*, to three target relations, *Emp*, *Dept*, and *Proj*. In this example, *Group* (similar to *Dept*) represents groups of scientists sharing a common area (e.g., a database group, a CS group, etc.) The dotted arrows represent foreign key constraints in the schemas.

Independent Mappings. Assume the existence of the following (independent) schema mappings from $\mathbf{S_1}$ to $\mathbf{S_2}$. The first mapping is the constraint t_1 in Figure 1(b), and corresponds to the arrow t_1 in Figure 1(a). This constraint requires every tuple in *Group* to be mapped to a tuple in *Dept* such that the group name (*gname*) becomes department name (*dname*). The second mapping is more complex and corresponds to the group of arrows t_2 in Figure 1(a). This constraint involves a custom filter condition; every pair of joining tuples of *Works* and *Group* for which the *addr* value is "NY" must be mapped into two tuples of *Emp* and *Dept*, sharing the same *did* value, and with corresponding *ename*, *addr* and *dname* values. (Note that *did* is a target-specific field that must exist and plays the role of key / foreign key). Intuitively, t_2 illustrates a pre-existing mapping that a user may have spent time in the past to create. Finally, the third constraint in Figure 1(b) corresponds to the arrow t_3 and maps *pname* from *Works* to *Proj*. This is an example of a correspondence that is introduced by a user after loading t_1 and the pre-existing mapping t_2 into the mapping tool.

The goal of the system is now to (re)generate a "good" overall schema mapping from $\mathbf{S_1}$ to $\mathbf{S_2}$ based on its input mappings. We note first that the input mappings, when considered in isolation, do not generate an ideal target instance.

Indeed, consider the source instance I in Figure 2. The target instance that is obtained by minimally enforcing the constraints $\{t_1, t_2, t_3\}$ is the instance J_1 also shown in the figure. The first *Dept* tuple is obtained by applying t_1 on the *Group* tuple $(123, CS)$. There, $D1$ represents some did value that must be associated with CS in this tuple. Similarly, the *Proj* tuple, with some unspecified value B for $budget$ and a did value of $D3$ is obtained via t_3. The *Emp* tuple together with the second *Dept* tuple are obtained based on t_2. As required by t_2, these tuples are linked via the same did value $D2$. Finally, to obtain a target instance that satisfies all the foreign key constraints, we must also have a third tuple in *Dept* that includes $D3$ together with some unspecified department name N.

Since the three mapping constraints are not correlated, the three did values ($D1$, $D2$, $D3$) are distinct. (There is no requirement that they must be equal.) As a result, the target instance J_1 exhibits the typical problems that arise when uncorrelated mappings are used to transform data: (1) *duplication of data* (e.g., multiple *Dept* tuples for CS with different did values), and (2) *loss of associations* where tuples are not linked correctly to each other (e.g., we have lost the association between project name Web and department name CS that existed in the source).


{t1,t2,t3}
MapMerge({t1,t2,t3})

Source instance I

Group		Works			
gno	gname	ename	addr	pname	gno
123	CS	John	NY	Web	123

Target instance J_1

Emp			Dept		Proj		
ename	addr	did	did	dname	pname	budget	did
John	NY	D2	D1	CS	Web	B	D3
			D2	CS			
			D3	N			

Target instance J_2

Emp			Dept		Proj		
ename	addr	did	did	dname	pname	budget	did
John	NY	D	D	CS	Web	B′	D

Figure 2: An instance of S_1 and two instances of S_2.

Correlated Mappings via MapMerge. Consider now the schema mappings that are shown in Figure 1(c) and that are the result of MapMerge applied on $\{t_1, t_2, t_3\}$. The notable difference from the input mappings is that all mappings consistently use the same expression, namely the Skolem term $F[g]$ where g denotes a distinct *Group* tuple, to give values for the did field. The first mapping is the same as t_1 but makes explicit the fact that did is $F[g]$. This mapping creates a unique *Dept* tuple for each distinct *Group* tuple. The second mapping is (almost) like t_2 with the additional use of the same Skolem term $F[g]$. Moreover, it also drops the existence requirement for *Dept* (since this is now implied by the first mapping). Finally, the third mapping differs from t_3 by incorporating a join with *Group* before it can actually use the Skolem term $F[g]$. As an additional artifact of MapMerge, which we explain later, it also includes a Skolem term $H_1[w]$ that assigns values for $budget$.

The target instance that is obtained by applying the result of MapMerge is the instance J_2 shown in Figure 2. The data associations that exist in the source are now correctly preserved in the target. For example, Web is linked to the CS tuple (via D) and also $John$ is linked to the CS tuple (via the same D). Furthermore, there is no duplication of *Dept* tuples.

Flows of Mappings. Taking the idea of mapping reuse and modularity one step further, an even more compelling use case for MapMerge in conjunction with mapping composition [8, 14, 17], is the *flow-of-mappings* scenario [1]. The key idea here is that to produce a data transformation from the source to the target, one can decompose the process into several simpler stages, where each stage maps from or into some intermediate, possibly simpler schema. Moreover, the simpler mappings and schemas play the role of reusable components that can be applied to build other flows. Such abstraction is directly motivated by the development of real-life, large-scale ETL flows such as those typically developed with IBM Information Server (Datastage), Oracle Warehouse Builder and others.

To illustrate, suppose the goal is to transform data from the schema

$\mathbf{S}_1$ of Figure 1(a) to a new schema $\mathbf{S_3}$, where *Staff* and *Projects* information are grouped under *CompSci*. The mapping or ETL designer may find it easier to first construct the mapping between $\mathbf{S}_1$ and $\mathbf{S}_2$ (it may also be that this mapping may have been derived in a prior design). Furthermore, the schema $\mathbf{S_2}$ is a normalized representation of the data, where *Dept*, *Emp* and *Proj* correspond directly to the main concepts (or types of data) that are being manipulated. Based on this schema, the designer can then produce a mapping m_{CS} from *Dept* to a more specialized object *CSDept*, by applying some customized filter condition (e.g., based on the name of the department). The next step is to create the mapping m from *CSDept* to the target schema. Other independent mappings are similarly defined for *Emp* and *Proj* (see m_1 and m_2).

Once these individual mappings are established, the same problem of correlating the mappings arises. In particular, one has to correlate $m_{\text{CS}} \circ m$, which is the result of applying mapping composition to m_{CS} and m, with the mappings m_1 for *Emp* and m_2 for *Proj*. This correlation will ensure that all employees and projects of computer science departments will be correctly mapped under their correct departments, in the target schema.

In this example, composition itself gives another source of mappings to be correlated by MapMerge. While similar with composition in that it is an operator on schema mappings, MapMerge is fundamentally different in that it correlates mappings that share the same source schema and the same target schema. In contrast, composition takes two sequential mappings where the target of the first mapping is the source of the second mapping. Nevertheless, the two operators are complementary and together they can play a fundamental role in building data flows.

1.2 Contributions and Outline of the Paper

Our main technical contributions are as follows. We give an algorithm for MapMerge, which takes as input arbitrary schema mappings expressed as second-order tgds [8] and generates correlated second-order tgds. As a particular important case, MapMerge can also take as input a set of raw schema correspondences; thus, it constitutes a replacement of existing mapping generation algorithms that are used in Clio [18, 10]. We introduce a novel similarity measure that is used to quantify the preservation of data associations from a source database to a target database. We use this measure to show experimentally that MapMerge improves the quality of schema mappings. In particular, we show that the target data that is produced based on the outcome of MapMerge has better quality, in terms of preservation of source associations, than the target data that is produced based on Clio-generated mappings.

Outline In the next section, we provide some preliminaries on schema mappings and their semantics. In Section 3 we give the main intuition behind MapMerge, while in Section 4 we describe the algorithm. In Section 5, we introduce the similarity measure that quantifies the preservation of associations. We then make use of this measure to evaluate the performance of MapMerge on real-life and synthetic mapping scenarios. We discuss related work in Section 6 and conclude in Section 7.

2. Preliminaries

A schema consists of a set of relation symbols, each with an associated set of attributes. Moreover, each schema can have a set of inclusion dependencies modeling foreign key constraints. While we restrict our presentation to the relational case, all our techniques are applicable and implemented in the more general case of the nested relational data model used in [18], where the schemas and mappings can be either relational or XML.

Schema Mappings. A schema mapping is a triple $(\mathbf{S},\mathbf{T},\Sigma)$ where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, and Σ is a set of *second-order tuple generating dependencies (SO tgds)* [8]. In this paper, we use the notation

$$\underline{\textit{for}}\ \vec{x}\ \underline{\textit{in}}\ \vec{S}\,\underline{\textit{satisfying}}\ B_1(\vec{x})\ \underline{\textit{exists}}\ \vec{y}\ \underline{\textit{in}}\ \vec{T}\,\underline{\textit{where}}\ B_2(\vec{y})\,\underline{\textit{and}}\ C(\vec{x},\vec{y})$$

for expressing SO tgds. Examples of SO tgds in this notation were already given in Figure 1(b) and Figure 1(c). Here, it suffices to say that $\vec{S}$ represents a vector of source relation symbols (possibly repeated), while $\vec{x}$ represents the tuple variables that are bound, correspondingly, to these relations. A similar notation applies for the $\underline{\textit{exists}}$ clause. The conditions $B_1(\vec{x})$ and $B_2(\vec{y})$ are conjunctions of equalities over the source and, respectively, target variables. The condition $C(\vec{x},\vec{y})$ is a conjunction of equalities that equate target expressions (e.g., y.A) with either source expressions (e.g., x.B) or Skolem terms of the form $F[x_1,\ldots,x_i]$, where F is a function symbol and $x_1,\ldots,x_i$ are a subset of the source variables. Skolem terms are used to relate target expressions across different SO tgds. An SO tgd without a Skolem term may also be called, simply, a tuple-generating dependency or tgd [7].

Note that our SO tgds do not allow equalities between or with Skolem terms in the $\underline{\textit{satisfying}}$ clause. While such equalities may be needed for more general purposes [8], they do not play a role for data exchange and can be eliminated, as observed in [24].

Chase-Based Semantics. The semantics that we adopt for a schema mapping $(\mathbf{S},\mathbf{T},\Sigma)$ is the standard data-exchange semantics [7] where, given a source instance I, the result of "executing" the mapping is the target instance J that is obtained by chasing I with the dependencies in Σ. Since the dependencies in Σ are SO tgds, we actually use an extension of the chase as defined in [8].

Intuitively, the chase provides a way of populating the target instance J in a minimal way, by adding the tuples that are *required* by Σ. For every instantiation of the $\underline{\textit{for}}$ clause of a dependency in Σ such that the $\underline{\textit{satisfying}}$ clause is satisfied but the $\underline{\textit{exists}}$ and $\underline{\textit{where}}$ clauses are not, the chase adds corresponding tuples to the target relations. Fresh new values (also called labeled nulls) are used to give values for the target attributes for which the dependency does not provide a source expression. Additionally, Skolem terms are instantiated by nulls in a consistent way: a term $F[x_1,\ldots,x_i]$ is replaced by the same null every time $x_1,\ldots,x_i$ are instantiated with the same source tuples. Finally, to obtain a valid target instance, we must chase (if needed) with the target constraints.

For our earlier example, the target instance J_1 is the result of chasing the source instance I with the tgds in Figure 1(b) and, additionally, with the foreign key constraints. There, the values $D1, D2, D3$ are nulls that are generated to fill did values for which the tgds do not provide a source expression. The target instance J_2 is the result of chasing I with the SO tgds in Figure 1(c). There, D is a null that corresponds to the Skolem term $F[g]$ where g is instantiated with the sole tuple of *Group*.

In practice, mapping tools such as Clio do not necessarily implement the chase with Σ, but generate queries to achieve a similar result [10, 18].

3. Correlating Mappings: Key Ideas

How do we achieve the systematic and, moreover, *correct* construction of correlated mappings? After all, we do not want arbitrary correlations between mappings, but rather only to the extent that the *natural* data associations in the source are preserved and no extra associations are introduced.

There are two key ideas behind MapMerge. The first idea is to exploit the structure and the constraints in the schemas in order to define what natural associations are (for the purpose of the algorithm). Two data elements are considered associated if they are in the same tuple or in two different tuples that are linked via constraints. This idea has been used before in Clio [18], and provides the first (conceptual) step towards MapMerge. For our example, the input mapping t_3 in Figure 1(b) is equivalent, in the presence of the source and target constraints, to the following enriched mapping:

t'_3: *for* w *in* Works, g *in* Group *satisfying* w.gno = g.gno
exists p *in* Proj, d *in* Dept
where p.pname = w.pname *and* p.did = d.did

Intuitively, if we have a w tuple in *Works*, we also have a joining tuple g in *Group*, since gno is a foreign key from *Works* to *Group*. Similarly, a tuple p in *Proj* implies the existence of a joining tuple in *Dept*, since did is a foreign key from *Proj* to *Dept*.

Formally, the above rewriting from t_3 to t'_3 is captured by the well-known chase procedure [2, 15]. The chase is a convenient tool to group together, syntactically, elements of the schema that are associated. The chase by itself, however, does not change the semantics of the mapping. In particular, the above t'_3 does not include any additional mapping behavior from *Group* to *Dept*.

The second key idea behind MapMerge is that of *reusing* or borrowing mapping behavior from a more general mapping to a more specific mapping. This is a heuristic that changes the semantics of the entire schema mapping and produces an arguably better one, with consolidated semantics.

To illustrate, consider the first mapping constraint in Figure 1(c). This constraint (obtained by skolemizing the input t_1) specifies a general mapping behavior from *Group* to *Dept*. In particular, it specifies how to create $dname$ and did from the input record. On the other hand, the above t'_3 can be seen as a more *specific* mapping from a *subset* of *Group* (i.e., those groups that have associated *Works* tuples) to a *subset* of *Dept* (i.e., those departments that have associated *Proj* tuples). At the same time, t'_3 does not specify any concrete mapping for the $dname$ and did fields of *Dept*. We can then borrow the mapping behavior that is already specified by the more general mapping. Thus, t'_3 can be enriched to:

t''_3: *for* w *in* Works, g *in* Group *satisfying* w.gno = g.gno
exists p *in* Proj, d *in* Dept
where p.pname = w.pname *and* p.did = d.did
and d.dname = g.gname *and* d.did = $F[g]$ *and* p.did = $F[g]$

where two of the last three equalities represent the "borrowed" behavior, while the last equality is obtained automatically by transitivity. Finally, we can drop the existence of d in *Dept* with the two conditions for $dname$ and did, since this is repeated behavior that is already captured by the more general mapping from *Group* to *Dept*. The resulting constraint is identical[3] to the third constraint in Figure 1(c), now correlated with the first one via $F[g]$. A similar explanation applies for the second constraint in Figure 1(c).

The actual MapMerge algorithm is more complex than intuitively suggested above, and is described in detail in the next section.

4. The MapMerge Algorithm

MapMerge takes as input a set $\{(\mathbf{S}, \mathbf{T}, \Sigma_1), ..., (\mathbf{S}, \mathbf{T}, \Sigma_n)\}$ of schema mappings over the same source and target schemas, which is equivalent to taking a single schema mapping $(\mathbf{S}, \mathbf{T}, \Sigma_1 \cup ... \cup \Sigma_n)$ as input. The algorithm is divided into four phases and the complete pseudocode is given in the appendix. The first phase decomposes each input mapping assertion into basic components that are, intuitively, easier to merge. In Phase 2, we apply the chase algorithm to compute associations (which we call *tableaux*), from the source and target schemas, as well as from the source and target assertions of the input mappings. By pairing source and target tableaux, we obtain all the possible *skeletons* of mappings. The actual work of constructing correlated mappings takes place in Phase 3, where for each skeleton, we take the union of all the basic components generated in Phase 1 that "match" the skeleton. Phase 4 is a simplification phase that also flags conflicts that may arise and that need to be addressed by the user. These conflicts occur when multiple mappings that map to the same portion of the target schema contribute with different, irreconcilable behaviors.

4.1 Phase 1: Decompose into Basic SO tgds

The first step of the algorithm decomposes each input SO tgd into a set of simpler SO tgds, called *basic SO tgds*, that have the same *for* and *satisfying* clause as the input SO tgd but have exactly one relation in the *exists* clause. Intuitively, we break the input mappings into atomic components that each specify mapping behavior for a single target relation. This decomposition step will subsequently allow us to merge mapping behaviors even when they come from different input SO tgds.

In addition to being single-relation in the target, each basic SO tgd gives a complete specification of all the attributes of the target relation. More precisely, each basic SO tgd has the form

for $\vec{x}$ *in* $\vec{S}$ *satisfying* $B_1(\vec{x})$
exists y *in* T *where* $\bigwedge_{A \in Atts(y)} y.A = e_A(\vec{x})$

where the conjunction in the *where* clause contains one equality constraint for *each* attribute of the record y asserted in the target relation T. The expression $e_A(\vec{x})$ is either a Skolem term or a source expression (e.g., x.B). Part of the role of the decomposition phase is to assign a Skolem term to every target expression y.A for which the initial mapping does not equate it to a source expression.

For our example, the decomposition algorithm (given in the appendix) obtains the following basic SO tgds from the input mappings t_1, t_2, and t_3 of Figure 1(b):

(b_1): *for* g *in* Group *exists* d *in* Dept
where d.did = $F[g]$ *and* d.dname = g.gname

(b_2): *for* w *in* Works, g *in* Group
satisfying w.gno = g.gno *and* w.addr = "NY"
exists e *in* Emp
where e.ename = w.ename *and* e.addr = w.addr *and* e.did = $G[w, g]$

(b'_2): *for* w *in* Works, g *in* Group
satisfying w.gno = g.gno *and* w.addr = "NY"
exists d *in* Dept
where d.did = $G[w, g]$ *and* d.dname = g.gname

(b_3): *for* w *in* Works *exists* p *in* Proj
where p.pname = w.pname *and* p.budget = $H_1[w]$ *and* p.did = $H_2[w]$

The basic SO tgd b_1 is obtained from t_1; the main difference is that d.did, whose value was unspecified by t_1, is now explicitly assigned the Skolem term $F[g]$. The only argument to F is g because g is the only record variable that occurs in the *for* clause of t_1. Similarly, the basic SO tgd b_3 is obtained from t_3, with the difference being that p.budget and p.did are now explicitly assigned the Skolem terms $H_1[w]$ and, respectively, $H_2[w]$.

In the case of t_2, we note that we have two existentially quantified variables, one for *Emp* and one for *Dept*. Hence, the decomposition algorithm generates two basic SO tgds: the first one maps into *Emp* and the second one maps into *Dept*. Observe that b_2 and b'_2 are correlated and share a common Skolem term $G[w, g]$ that is assigned to both e.did and d.did. Thus, the association between

[3] Modulo the absence of $H_1[w]$, which will be explained separately.

e.did and d.did in the original schema mapping t_2 is maintained in the basic SO tgds b_2 and b'_2.

In general, the decomposition process ensures that associations between target facts that are asserted by the original schema mapping are not lost. The process is similar to the Skolemization procedure that transforms first order tgds with existentially quantified variables into second order tgds with Skolem functions (see [8]). After such Skolemization, all the target relations can be separated since they are correlated via Skolem functions. Therefore, the set of basic SO tgds that results after decomposition is equivalent to the input set of mappings.

4.2 Phase 2: Compute Skeletons of Schema Mappings

Next we apply the chase algorithm to compute syntactic associations (which we call *tableaux*), from each of the schemas and from the input mappings. Essentially, a schema *tableau* is constructed by taking each relation symbol in the schema and chasing it with all the referential constraints that apply. The result of such chase is a tableau that incorporates a set of relations that is closed under referential constraints, together with the join conditions that relate those relations. For each relation symbol in the schema, there is one schema tableau. As in [10, 18], in order to guarantee termination, we stop the chase whenever we encounter cycles in the referential constraints. In our example, there are two source schema tableaux and three target schema tableaux, as follows:

T_1 = { $g \in$ Group }
T_2 = { $w \in$ Works, $g \in$ Group; w.gno = g.gno }
T_3 = { $d \in$ Dept }
T_4 = { $e \in$ Emp, $d \in$ Dept; e.did = d.did }
T_5 = { $p \in$ Proj, $d \in$ Dept; p.did = d.did }

Intuitively, schema tableaux represent the categories of data that can exist according to the schema. A *Group* record can exist independently of records in other relations (hence, the tableau T_1). However, the existence of a *Works* record implies that there must exist a corresponding *Group* record with identical *gno* (hence, the tableau T_2).

Since the MapMerge algorithm takes as input arbitrary mapping assertions, we also need to generate user-defined mapping tableaux, which are obtained by chasing the source and target assertions of the input mappings with the referential constraints that are applicable from the schemas (see Appendix A). The notion of user-defined tableaux is similar to the notion of user associations in [23]. In our example, there is only one new tableau based on the source assertions of the input mapping t_2:

T'_2 = { $w \in$ Works, $g \in$ Group; w.gno = g.gno, w.addr = "NY" }

Furthermore, we then pair every source tableau with every target tableau to form a *skeleton*. Each skeleton represents the empty shell of a candidate mapping. For our running example, the set of all skeletons at the end of Phase 2 is: $\{(T_1, T_3)$, (T_1, T_4), (T_1, T_5), (T_2, T_3), (T_2, T_4), (T_2, T_5), (T'_2, T_3), (T'_2, T_4), $(T'_2, T_5)\}$.

4.3 Phase 3: Match and Apply Basic SO tgds on Skeletons

In this phase, for each skeleton, we first find the set of basic SO tgds that "match" the skeleton. Then, for each skeleton, we apply the basic SO tgds that were found matching, and construct a merged SO tgd. The resulting SO tgd is, intuitively, the "conjunction" of all the basic SO tgds that were found matching.

Matching. We say that a basic SO tgd σ matches a skeleton (T, T') if there is a pair (h, g) of homomorphisms that "embed" σ into (T, T'). This translates into two conditions. First, the *for* and *satisfying* clause of σ are embedded into T via the homomorphism h. This means that h maps the variables in the *for* clause of σ to variables of T such that relation symbols are respected and, moreover, the *satisfying* clause of σ (after applying h) is implied by the conditions of T. Additionally, the *exists* clause of σ must be embedded into T' via the homomorphism g. Since σ is a basic SO tgd and there is only one relation in its *exists* clause, the latter condition essentially states that the target relation in σ must occur in T'.

For our running example, it is easy to see that the basic SO tgd b_1 matches the skeleton (T_1, T_3). In fact, b_1 matches every skeleton from Phase 2. On the other hand, the basic SO tgd b_2 matches only the skeleton (T'_2, T_4) under the homomorphisms (h_1, h_2), where $h_1 = \{w \mapsto w, g \mapsto g\}$ and $h_2 = \{e \mapsto e\}$. Altogether, we obtain the following matching of basic SO tgds on skeletons:

$$(T_1, T_3, b_1) \quad (T_1, T_4, b_1) \quad (T_1, T_5, b_1) \quad (T_2, T_3, b_1)$$
$$(T_2, T_4, b_1) \quad (T_2, T_5, b_1 \wedge b_3) \quad (T'_2, T_3, b_1 \wedge b'_2)$$
$$(T'_2, T_4, b_1 \wedge b_2 \wedge b'_2) \quad (T'_2, T_5, b_1 \wedge b'_2 \wedge b_3)$$

Note that the basic SO tgds that match a given skeleton may actually come from different input mappings. For example, each of the basic SO tgds that match (T'_2, T_5) comes from a separate input mapping (from t_1, t_2, and t_3, respectively). In a sense, we aggregate behaviors from multiple input mappings in a given skeleton.

Computing merged SO tgds. For each skeleton along with the matching basic SO tgds, we now construct a "merged" SO tgd. For our example, the following SO tgd s_8 is constructed from the eighth triple $(T'_2, T_4, b_1 \wedge b_2 \wedge b'_2)$ shown earlier.

(s_8) *for* w *in* Works, g *in* Group
satisfying w.gno = g.gno *and* w.addr = "NY"
exists e *in* Emp, d *in* Dept
where e.did = d.did
and d.did = $F[g]$ *and* d.dname=g.gname
and e.ename = w.ename *and* e.addr = w.addr *and* e.did = $G[w, g]$
and d.did = $G[w, g]$

The variable bindings in the source and target tableaux are taken literally and added to the *for* and, respectively, *exists* clause of the new SO tgd. The equalities in T'_2 and T_4 are also taken literally and added to the *satisfying* and, respectively, *where* clause of the SO tgd. More interestingly, for every basic SO tgd σ that matches the skeleton (T'_2, T_4), we take the *where* clause of σ (after applying the respective homomorphisms) and add it to the *where* clause of the new SO tgd. (Note that, by definition of matching, the *satisfying* clause of σ is automatically implied by the conditions in the source tableau.) The last three lines in the above SO tgd incorporate conditions taken from each of the basic SO tgds that match (T'_2, T_4) (i.e., from b_1, b_2, and b'_2, respectively).

The constructed SO tgd consolidates the semantics of b_1, b_2, and b'_2 under one merged mapping. Intuitively, since all three basic SO tgds are applicable whenever the source pattern is given by T'_2 and the target pattern is given by T_4, the resulting SO tgd takes the conjunction of the "behaviors" of the individual basic SO tgds.

Correlations. A crucial point about the above construction is that a target expression may now be assigned multiple expressions. For example, in the above SO tgd, the target expression d.did is equated with two expressions: $F[g]$ via b_1, and $G[w, g]$ via b'_2. In other words, the semantics of the new constraint requires the values of the two Skolem terms to coincide. This is actually what it means to correlate b_1 and b'_2. We can represent such correlation, explicitly, as the following conditional equality (implied by the above SO tgd):

for w *in* Works, g *in* Group *satisfying* w.gno = g.gno *and* w.addr = "NY"
$\Rightarrow F[g] = G[w, g]$

We use the term *residual equality constraint* for such equality constraint where one member in the implied equality is a Skolem term while the other is either a source expression or another Skolem term. Such constraints have to be enforced at runtime when we perform data exchange with the result of MapMerge. In general, Skolem functions are implemented as (independent) lookup tables, where for every different combination of the arguments, the lookup table gives a fresh new null. However, residual constraints will require correlation between the lookup tables. For example, the above constraint requires that the two lookup tables (for F and G) must give the same value whenever w and g are tuples of *Works* and *Group* with the same gno value.

To conclude the presentation of Phase 3, we list the other three merged SO tgds below that result after this phase for our example.

(s_1) from (T_1, T_3, b_1):
for g *in* Group
exists d *in* Dept
where d.did = $F[g]$ *and* d.dname=g.gname

(s_6) from $(T_2, T_5, b_1 \wedge b_3)$:
for w *in* Works, g *in* Group *satisfying* w.gno = g.gno
exists p *in* Proj, d *in* Dept
where p.did = d.did
and d.did = $F[g]$ *and* d.dname=g.gname
and p.pname = w.pname *and* p.budget = $H_1[w]$ *and* p.did = $H_2[w]$

(s_9) from $(T_2', T_5, b_1 \wedge b_2' \wedge b_3)$:
for w *in* Works, g *in* Group
satisfying w.gno = g.gno *and* w.addr = “NY”
exists p *in* Proj, d *in* Dept
where p.did = d.did
and d.did = $F[g]$ *and* d.dname=g.gname
and p.pname = w.pname *and* p.budget = $H_1[w]$ *and* p.did = $H_2[w]$
and d.did = $G[w, g]$

One aspect to note is that not all skeletons generate merged SO tgds. Although we had six earlier skeletons, only three generate mappings that are neither *subsumed* nor *implied*. (See also the appendix.) We use here the technique for pruning subsumed or implied mappings described in [10]. For an example of a subsumed mapping, consider the triple (T_1, T_4, b_1). We do not generate a mapping for this, because its behavior is subsumed by s_1, which includes the same basic component b_1 but maps into a more “general” tableau, namely T_3. Intuitively, we do not want to construct a mapping into T_4, which is a larger (more specific) tableau, without actually using the extra part of T_4. Implied mappings are those that are *logically implied* by other mappings. For example, the mapping that would correspond to (T_2, T_3, b_1) is logically implied by s_6: they both have the same premise (T_2), but s_6 asserts facts about a larger tableau (T_5, which includes T_3) and already covers b_1.

Finally, for our example, we also obtain three more residual equality constraints, arising from s_6, and stating the pairwise equalities of $F[g]$, $H_2[w]$ and $G[w, g]$ (since they are all equal to p.did and d.did, which are also equal to each other).

Since residual equalities cause extra overhead at runtime, it is worthwhile exploring when such constraints can be eliminated without changing the overall semantics. We describe such method next.

4.4 Phase 4: Eliminate Residual Equality Constraints

The fourth and final phase of the MapMerge algorithm attempts to eliminate as many Skolem terms as possible from the generated SO tgds. The key idea is that, for each residual equality constraint, we attempt to substitute, globally, one member of the equality with the other member. If the substitution succeeds then there is one less residual equality constraint to enforce during runtime. Moreover, the resulting SO tgds are syntactically simpler.

Consider our earlier residual constraint stating the equality $F[g] = G[w, g]$ (under the conditions of the *for* and *satisfying* clauses). The two Skolem terms $F[g]$ and $G[w, g]$ occur globally in multiple SO tgds. To avoid the explicit maintenance and correlation of two lookup tables (for both F and G), we attempt the substitution of either $F[g]$ with $G[w, g]$ or $G[w, g]$ with $F[g]$. Care must be taken since such substitution cannot be arbitrarily applied. First, the substitution can only be applied in SO tgds that satisfy the preconditions of the residual equality constraint. For our example, we cannot apply either substitution to the earlier SO tgd s_1, since the precondition requires the existence of *Works* tuple that joins with *Group*. In general, we need to check for the existence of a homomorphism that embeds the preconditions of the residual equality constraint into the *for* and *where* clauses of the SO tgd. The second issue is that the direction of the substitution matters. For example, let us substitute $F[g]$ by $G[w, g]$ in every SO tgd that satisfies the preconditions. There are two such SO tgds: s_8 and s_9. After the substitution, in each of these SO tgds, the equality d.did = $F[g]$ becomes d.did = $G[w, g]$ and can be dropped, since it is already in the *where* clause. Note, however, that the replacement of $F[g]$ by $G[w, g]$ did not succeed globally. The SO tgds s_1 and s_6 still refer to $F[g]$. Hence, we still need to maintain the explicit correlation of the lookup tables for F and G. On the other hand, let us substitute $G[w, g]$ by $F[g]$ in every SO tgd that satisfies the preconditions. Again, there are two such SO tgds: s_8 and s_9. The outcome is different now: $G[w, g]$ disappears from both s_8 and s_9 (in favor of $F[g]$); moreover, it did not appear in s_1 or s_6 to start with. We say that the substitution of $G[w, g]$ by $F[g]$ has globally succeeded. Following this substitution, the constraint s_9 is implied by s_6: they both assert the same target tuples, and the source tableau T_2' for s_9 is a restriction of the source tableau T_2 for s_6. Hence from now on we can discard the constraint s_9.

Similarly, based on the other residual equality constraint we had earlier, we can apply the substitution of $H_2[w]$ by $F[g]$. This affects only s_6 and the outcome is that $H_2[w]$ has been successfully replaced globally. The resulting SO tgds, for our example, are:

(s_1) *for* g *in* Group
exists d *in* Dept
where d.did = $F[g]$ *and* d.dname=g.gname

(s_6') *for* w *in* Works, g *in* Group *satisfying* w.gno = g.gno
exists p *in* Proj, d *in* Dept
where p.did = d.did
and d.did = $F[g]$ *and* d.dname=g.gname
and p.pname = w.pname *and* p.budget = $H_1[w]$ *and* p.did = $F[g]$

(s_8') *for* w *in* Works, g *in* Group
satisfying w.gno = g.gno *and* w.addr = “NY”
exists e *in* Emp, d *in* Dept
where e.did = d.did
and d.did = $F[g]$ *and* d.dname=g.gname
and e.ename = w.ename *and* e.addr = w.addr *and* e.did = $F[g]$

As explained in Section 3, both s_6' and s_8' can be simplified, by removing the assertions about *Dept*, since they are implied by s_1. The result is then identical to the SO tgds shown in Figure 1(c).

Our example covered only residual equalities between Skolem terms. The case of equalities between a Skolem term and a source expression is similar, with the difference that we form only one substitution (to replace the Skolem term by the source expression).

The exact algorithm for eliminating residual constraints, given in the appendix, is an exhaustive algorithm that forms each possible substitution and attempts to apply it on the existing SO tgds. If the replacement is globally successful, the residual equality constraint that generated the substitution can be eliminated. Then, the algorithm goes on to eliminate other residual constraints on the rewritten SO tgds. If the replacement is not globally successful, the algorithm tries the reverse substitution (if applicable). In general, it may be the case that neither substitution succeeds globally. In such case, the corresponding residual constraint is kept as part of the output of MapMerge. Thus, the outcome of MapMerge is, in general, a set of SO tgds *together* with a set of residual equality constraints. (For our example, the latter set is empty.)

Finally, the last issue that arises is the case of conflicts in mapping behavior. Conflicts can also be described via constraints, similar to residual equality constraints but with the main difference that both members of the equality are source expressions (and not Skolem terms). To illustrate, it might be possible that a merged SO tgd asserts that the target expression d.dname is equal to both g.gname (from some input mapping) and with g.code (from some other input mapping, assuming that code is some other source attribute). Then, we obtain conflicting semantics, with two competing source expressions for the same target expression. Our algorithm flags such conflicts, whenever they arise, and returns the mapping to the user to be resolved.

5. Evaluation

To evaluate the quality of the data generated based on MapMerge, we introduce a measure that captures the similarity between a source and target instance by measuring the amount of data associations that are preserved by the transformation from the source to the target instance. We will use this similarity measure in our experiments to show that the mappings derived by MapMerge are better than the input mappings.

5.1 Similarity Measure

The main idea behind our measure is to capture the extent to which the "associations" in a source instance are preserved when transformed into a target instance of a different schema. For each instance, we will compute a single relation that incorporates all the natural associations between data elements that exist in the instance. There are two types of associations we consider. The first type is based on the chase with referential constraints and is naturally captured by tableaux. As seen in Section 4.2, a tableau is a syntactic object that takes the "closure" of each relation under referential constraints. We can then materialize the join query that is encoded in each tableau and select all the attributes that appear in the input relations (without duplicating the foreign key / key attributes). Thus, for each tableau, we obtain a single relation, called *tableau relation*, that conveniently materializes together data associations that span multiple relations. For example, the tableau relations for the source instance I in Figure 1 (for tableaux T_1 and T_2 in Section 4.2) are shown on top of Figure 3(b). We denote the tableau relations of an instance I of schema $\mathbf{S}$ as $\tau_{\mathbf{S}}(I)$, or simply $\tau(I)$. The tableau relations $\tau(J_1)$ and $\tau(J_2)$ for our running example are also shown in Figure 3.

The second type of association that we consider is based on the notion of *full disjunction* [11, 20]. Intuitively, the full disjunction of relations R_1, ..., R_k, denoted as $\mathrm{FD}(R_1, ..., R_k)$, captures in a single relation all the associations (via natural join) that exist among tuples of the input relations. The reason for using full disjunction is that tableau relations by themselves do not capture all the associations. For example, consider the association that exists between *John* and *Web* in the earlier source instance J_2. There, *John* is an employee in *CS*, and *Web* is a project in *CS*. However, since there is no directed path via foreign keys from *John* to *Web*, the two data elements appear in different tableau relations of $\tau(J_2)$ (namely, *DeptEmp* and *DeptProj*). On the other hand, if we take the natural join between *DeptEmp* and *DeptProj*, the association between *John* and *Web* will appear in the result. Thus, to capture all such associations, we apply an additional step which computes the full disjunction $\mathrm{FD}(\tau(I))$ of the tableau relations. This generates a single relation that conveniently captures all the associations in an instance I of schema $\mathbf{S}$. Intuitively, each tuple in this relation corresponds to one association that exists in the data.

Operationally, full disjunction must perform the outer "union" of all the tuples in every input relation, together with all the tuples that arise via all possible natural joins among the input relations. To avoid redundancy, *minimal union* is used instead of union. This means that in the final relation, tuples that are subsumed by other tuples are pruned. A tuple t is *subsumed* by a tuple t' if for all attributes A such that $t.A \neq \mathrm{null}$, it is the case that $t'.A = t.A$. We omit here the details of implementing full disjunction, but we point out that such implementation is part of our experimental evaluation.

For our example, we show $\mathrm{FD}(\tau(J_1))$, $\mathrm{FD}(\tau(I))$, and $\mathrm{FD}(\tau(J_2))$ at the bottom of Figure 3. There, we use the '-' symbol to represent the SQL null value. We note that $\mathrm{FD}(\tau(J_2))$ connects now all three of *John*, *Web* and *CS* in one tuple.

Now that we have all the associations in a single relation, one on each side (source or target), we can compare them. More precisely, given a source instance I and a target instance J, we define the similarity between I and J by defining the similarity between $\mathrm{FD}(\tau(I))$ and $\mathrm{FD}(\tau(J))$. However, when we compare tuples between $\mathrm{FD}(\tau(I))$ and $\mathrm{FD}(\tau(J))$, we should not compare arbitrary pairs of attributes. Intuitively, to avoid capturing "accidental" preservations, we want to compare tuples based only on their *compatible* attributes that arise from the mapping. In the following, we assume that all the mappings that we need to evaluate implement the same set $\mathcal{V}$ of correspondences between attributes of the source schema $\mathbf{S}$ and attributes of the target schema $\mathbf{T}$. This assumption is true for mapping generation algorithms, which start from a set of correspondences and generate a faithful implementation of the correspondences (without introducing new attribute-to-attribute mappings). It is also true for MapMerge and its input, since MapMerge does not introduce any new attribute-to-attribute mappings that are not already specified by the input mappings. Given a set $\mathcal{V}$ of correspondences between $\mathbf{S}$ and $\mathbf{T}$, we say that an attribute A of $\mathbf{S}$ is *compatible* with an attribute B of $\mathbf{T}$ if either there is a direct correspondence between A and B in $\mathcal{V}$, or (2) A is related to an attribute A' via a foreign key constraint of $\mathbf{S}$, B is related to an attribute B' via a foreign key constraint of $\mathbf{T}$, and A' is compatible with B'. For our example, the pairs of compatible attributes (from source to target) are: $(gname, dname)$, $(ename, ename)$, $(addr, addr)$, $(pname, pname)$.

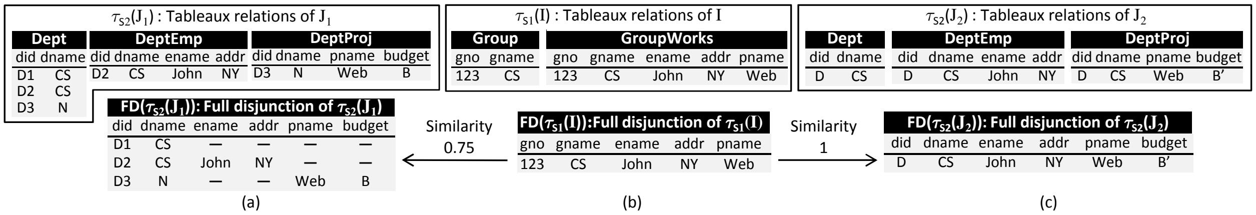
DEFINITION 1 (TUPLE SIMILARITY). Let t_1 and t_2 be two tuples in $\mathrm{FD}(\tau(I))$ and, respectively, $\mathrm{FD}(\tau(J))$. The *similarity of t_1 and t_2*, denoted as $\mathrm{Sim}(t_1, t_2)$, is defined as:

$$\frac{|\{A \in \mathit{Atts}(t_1) \mid \exists B \in \mathit{Atts}(t_2),\ A \text{ and } B \text{ compatible},\ t_1.A = t_2.B \neq \mathrm{null}\}|}{|\{A \in \mathit{Atts}(t_1) \mid \exists B \in \mathit{Atts}(t_2),\ A \text{ and } B \text{ compatible}\}|}$$

Intuitively, $\mathrm{Sim}(t_1, t_2)$ captures the ratio of the number of values that are actually exported from t_1 to t_2 versus the number of values that could be exported from t_1 according to $\mathcal{V}$. For instance, let t_1 be the only tuple in $\mathrm{FD}(\tau(I))$ from Figure 3 and t_2 the only tuple in $\mathrm{FD}(\tau(J_2))$. Then, $\mathrm{Sim}(t_1, t_2)$ is 1.0, since $t_1.A = t_2.B$ for every pair of compatible attributes A and B. Now, let t_2 be the first tuple in $\mathrm{FD}(\tau(J_1))$. Since only t_1.gname = t_2.dname out of four pairs of compatible attributes, we have that $\mathrm{Sim}(t_1, t_2)$ is 0.25.

DEFINITION 2 (INSTANCE SIMILARITY). The similarity between $\mathrm{FD}(\tau(I))$ and $\mathrm{FD}(\tau(J))$ is

$$\mathrm{Sim}(\mathrm{FD}(\tau(I)), \mathrm{FD}(\tau(J))) = \sum_{t_1 \in \mathrm{FD}(\tau(I))} \max_{t_2 \in \mathrm{FD}(\tau(J))} \mathrm{Sim}(t_1, t_2).$$



$\tau_{S2}(J_1)$: Tableaux relations of J_1

Dept	
did	dname
D1	CS
D2	CS
D3	N

DeptEmp			
did	dname	ename	addr
D2	CS	John	NY

DeptProj			
did	dname	pname	budget
D3	N	Web	B

FD($\tau_{S2}(J_1)$): Full disjunction of $\tau_{S2}(J_1)$

did	dname	ename	addr	pname	budget
D1	CS	—	—	—	—
D2	CS	John	NY	—	—
D3	N	—	—	Web	B

(a)

$\tau_{S1}(I)$: Tableaux relations of I

Group	
gno	gname
123	CS

GroupWorks				
gno	gname	ename	addr	pname
123	CS	John	NY	Web

FD($\tau_{S1}(I)$):Full disjunction of $\tau_{S1}(I)$

gno	gname	ename	addr	pname
123	CS	John	NY	Web

(b)

$\tau_{S2}(J_2)$: Tableaux relations of J_2

Dept	
did	dname
D	CS

DeptEmp			
did	dname	ename	addr
D	CS	John	NY

DeptProj			
did	dname	pname	budget
D	CS	Web	B’

FD($\tau_{S2}(J_2)$): Full disjunction of $\tau_{S2}(J_2)$

did	dname	ename	addr	pname	budget
D	CS	John	NY	Web	B’

(c)

Figure 3: Tableau relations of J_1, I, and J_2 of Figure 2 and their full disjunctions.

Figure 3 depicts the similarities $\mathrm{Sim}(\mathrm{FD}(\tau(I)), \mathrm{FD}(\tau(J_1)))$ and $\mathrm{Sim}(\mathrm{FD}(\tau(I)),\mathrm{FD}(\tau(J_2)))$. The former similarity score is obtained by comparing the only tuple in $\mathrm{FD}(\tau(I))$ with the best matching tuple (i.e., the second tuple) in $\mathrm{FD}(\tau(J_1)))$.

5.2 Experiments

To evaluate MapMerge, we conducted a series of experiments on a set of synthetic mapping scenarios as well as on two real-life mapping scenarios from the biological domain. We give next some highlights of our results on the synthetic scenarios and report the rest in the appendix.

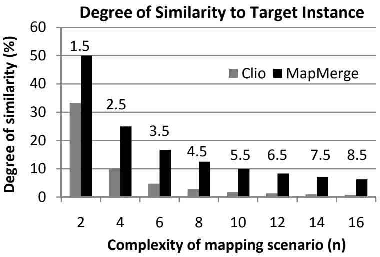


Figure 4: Improvement in quality of generated data.

Our synthetic mapping scenarios transform data from a denormalized source schema with a single relation to a target schema containing a number of hierarchies, with each hierarchy having at its top an “authority” relation, while other relations refer to the authority relation through foreign key constraints. The target schema corresponds roughly to ontological schemas, which often contain top-level concepts that are referred to by many sub-concepts. The synthetic scenarios were also designed to scale so that we can measure both the running time performance of MapMerge and the improvement in target data quality as the schemas increase in size.

Figure 4 shows the improvement in the quality of the generated data that we obtain by using MapMerge versus using Clio-generated mappings [10]. In the experiment, the Clio-generated mappings are used as input to the MapMerge operator. Thus, the experiment shows the benefit of using MapMerge on top of Clio mappings. The parameter n that describes the complexity of the mapping scenario in terms of schemas is shown on the x-axis. On the y-axis, we show the *degree of similarity* of the source instance I to the target instance J that is generated (by using MapMerge or Clio mappings). Here, the degree of similarity of I to J is computed as the ratio of $\mathrm{Sim}(\mathrm{FD}(\tau(I)), \mathrm{FD}(\tau(J)))$ to $\mathrm{Sim}(\mathrm{FD}(\tau(I)), \mathrm{FD}(\tau(I)))$, where the latter represents the ideal case where every tuple in $\mathrm{FD}(\tau(I))$ is preserved by the target instance. (Note that the latter quantity simplifies to the expression $|\mathrm{FD}(\tau(I))|$.)

As the figure shows, the degree of similarity decreases as n increases (for both MapMerge and Clio mappings). The reason is that, as n becomes larger, the source relation is broken into a larger number of uncorrelated top-level target concepts. Thus, the increased loss of associations from the source to the target is inevitable. However, the relative improvement when using MapMerge on top of the Clio mappings (shown as the numbers on top of the bars) increases substantially, as n becomes larger. The reason is that MapMerge is able to correctly map to an entire hierarchy (for each top-level concept) without any loss of associations, while Clio mappings have only a limited ability. Complete explanations of the experiment are given in the appendix, where we also measure the performance of MapMerge in terms of its running time.

6. Related Work

Model management [16] has considered various operators on schema mappings, among which Confluence is closest in spirit to MapMerge. Confluence also operates on mappings with the same source and target schema, and it amounts to taking the conjunction of the constraints in the input mappings. Thus, Confluence does not attempt any correlation of the input mappings. Our work can be seen as a step towards the high-level design and optimization in ETL flows [21, 22]. This can be envisioned by incorporating mappings [4] into such flows, and employing operators such as MapMerge and composition to support modularity and reuse.

The instance similarity measure we used to evaluate MapMerge draws its inspiration from the very general notion of Hausdorff distance between subsets of metric spaces, and from the sum of minimum distances measure. We refer to [5] for a discussion of these measures. Moreover, our notion of tuple similarity is loosely based on the well known Jaccard coefficient. However, the previous measures are symmetric and agnostic to the transformation that produces one database instance from the other. In contrast, our notion is tailored to measure the preservation of data associations from a source database to a target database under a schema mapping.

7. Conclusions

We have presented our MapMerge algorithm and an evaluation of our implementation of MapMerge. Through a similarity measure that computes the amount of data associations that are preserved from one instance to another, our evaluation shows that a given source instance has higher similarity to the target instance obtained through MapMerge when compared to target instances obtained through other mapping tools. As part of our future work, we intend to explore the use of the notion of information loss [9] to compare between mappings generated by MapMerge with those generated by other mapping tools. In addition, we would like to further explore applications of MapMerge to flows of mappings.

Acknowledgements. Hernández and Popa are partially funded by the U.S. Air Force Office for Scientific Research under contracts FA9550-07-1-0223 and FA9550-06-1-0226. Part of the work was done while Alexe was visiting IBM Almaden Research Center. Alexe and Tan are supported by NSF grant IIS-0430994 and NSF grant IIS-0905276. Tan is also supported by a NSF CAREER award IIS-0347065.

8. References

[1] B. Alexe, M. Gubanov, M. A. Hernández, H. Ho, J.-W. Huang, Y. Katsis, L. Popa, B. Saha, and I. Stanoi. Simplifying Information Integration: Object-Based Flow-of-Mappings Framework for Integration. In *BIRTE*, pages 108–121. Springer, 2009.

[2] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *JACM*, 31(4):718–741, 1984.

[3] A. Bonifati, E. Q. Chang, T. Ho, V. S. Lakshmanan, and R. Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. In *VLDB*, pages 1267–1270, 2005.

[4] S. Dessloch, M. A. Hernández, R. Wisnesky, A. Radwan, and J. Zhou. Orchid: Integrating schema mapping and ETL. In *ICDE*, pages 1307–1316, 2008.

[5] T. Eiter and H. Mannila. Distance measures for point sets and their computation. *Acta Inf.*, 34(2):109–133, 1997.

[6] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications*, pages 198–236. Springer, 2009.

[7] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[8] R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *TODS*, 30(4):994–1055, 2005.

[9] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Reverse data exchange: coping with nulls. In *PODS*, pages 23–32, 2009.

[10] A. Fuxman, M. A. Hernández, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *VLDB*, pages 67–78, 2006.

[11] C. A. Galindo-Legaria. Outerjoins as disjunctions. In *SIGMOD Conference*, pages 348–358, 1994.

[12] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[13] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[14] J. Madhavan and A. Y. Halevy. Composing Mappings Among Data Sources. In *VLDB*, pages 572–583, 2003.

[15] D. Maier, A. O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *TODS*, 4(4):455–469, 1979.

[16] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting Executable Mappings in Model Management. In *SIGMOD*, pages 167–178, 2005.

[17] A. Nash, P. A. Bernstein, and S. Melnik. Composition of Mappings given by Embedded Dependencies. In *PODS*, pages 172–183, 2005.

[18] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

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

[20] A. Rajaraman and J. D. Ullman. Integrating information by outerjoins and full disjunctions. In *PODS*, pages 238–248, 1996.

[21] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL Processes in Data Warehouses. In *ICDE*, pages 564–575, 2005.

[22] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual Modeling for ETL Processes. In *DOLAP*, pages 14–21, 2002.

[23] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.

[24] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *VLDB*, pages 1006–1017, 2005.

APPENDIX

A. Pseudocode of MapMerge

The main algorithm for MapMerge is given below. This algorithm makes calls to several subroutines, which are listed separately, in the respective subsections.

```
Algorithm MapMerge($\mathbf{S}$, $\mathbf{T}$, $\Sigma$)
Input: A schema mapping.
Output: $(\mathbf{S}, \mathbf{T}, \Sigma')$ and $F$, where $\Sigma'$ is the correlated schema mapping
and $F$ is a set of failed unifications or "residual constraints".

Phase 1. (Decompose into basic SO tgds)
    Initialize the set of basic SO tgds $B = \emptyset$.
    For each SO tgd $\sigma \in \Sigma$ do
            Add Decompose($\sigma$) to $B$
Phase 2. (Compute skeletons of schema mappings)
    Initialize the set of skeletons $K = \emptyset$
    Initialize the set of source and target tableaux $T_{src} = \emptyset,\ T_{tgt} = \emptyset$
    Generate the schema tableaux:
        For each relation $R \in \mathbf{S}$
            Chase $\{x \in R\}$ with referential constraints in $\mathbf{S}$, add result to $T_{src}$
        For each relation $Q \in \mathbf{T}$
            Chase $\{y \in Q\}$ with referential constraints in $\mathbf{T}$, add result to $T_{tgt}$
    Generate the user-defined tableaux:
        For each SO tgd $\sigma \in \Sigma$ of the form
          for $\vec{x}$ in $\vec{R}$ satisfying $B_1(\vec{x})$ exists $\vec{y}$ in $\vec{T}$ where $B_2(\vec{y}) \wedge C(\vec{x}, \vec{y})$
            Chase $\{\vec{x} \in \vec{R};\ B_1(\vec{x})\}$ with referential constraints in $\mathbf{S}$
              If the result is not implied by $T_{src}$, add it to $T_{src}$
            Chase $\{\vec{y} \in \vec{T};\ B_2(\vec{y})\}$ with referential constraints in $\mathbf{T}$
              If the result is not implied by $T_{tgt}$, add it to $T_{tgt}$
    For each $T \in T_{src}$ and $T' \in T_{tgt}$ do
            Add the skeleton $(T, T')$ to $K$.
Phase 3. (Match and apply basic SO tgds on skeletons)
    Initialize the list of output constraints $\Sigma' = \emptyset$
    For each skeleton $K_i \in K$ do
            Initialize the set $B_i = \emptyset$
            For each $\sigma \in B$ do
                    Let $L_i$ = Match($\sigma, K_i$)
                    If $L_i \neq \emptyset$, then add the pair $\langle \sigma, L_i \rangle$ to $B_i$
            Update $\Sigma'$ to be $\Sigma' \cup$ ConstructSOtgd($K_i, B_i$)
    Remove from $\Sigma'$ every $\sigma'$ such that for some $\sigma'' \in \Sigma'$
            such that $\sigma'' \neq \sigma'$, either $\sigma'' \models \sigma'$ or $\sigma''$ subsumes $\sigma'$
Phase 4. (Eliminate residual equality constraints)
    Initialize the list of failed substitutions $F = \emptyset$
    Repeat
          Let $U$ = FindNextSubstitution($\Sigma'$, $F$)
          If $U$ is a substitution candidate (i.e., not a failure) then
            If $U$ cannot be successfully applied on $\Sigma'$
              (i.e., Substitute($\Sigma', U$) fails) then
              Add the failed substitution $U$ to $F$
    Until no more substitutions can be applied
Return $(\Sigma', F)$ as the output of the algorithm
```

A.1 Pseudocode used by Phase 1

The algorithm that decomposes an input SO tgd into its set of basic SO tgds is listed below.

```
Algorithm Decompose($\sigma$)
Input: $\sigma$ is an input SO tgd
Output: $\Sigma$ is a set of basic SO tgds resulting from the decomposition of $\sigma$
Initialize $\Sigma = \emptyset$
Assume the input SO tgd $\sigma$ is of the form:
        for $\vec{x}$ in $\vec{S}$ satisfying $C(\vec{x})$ exists $\vec{y}$ in $\vec{T}$ where $C'(\vec{x}, \vec{y})$
```

The target condition C' is a conjunction of equalities between source and target expressions, or between target expressions. These equalities partition the source and target expressions in the *where* clause into a set E of equivalence classes. Associate a fresh Skolem term $F_j[\vec{x}]$ to each equivalence class $E_j \in E$.

For each y_i *in* T_i from the *exists* clause of σ

```
    Initialize the basic SO tgd $\sigma'$ to be
            for $\vec{x}$ in $\vec{S}$ satisfying $C(\vec{x})$ exists $y_i$ in $T_i$
    For each attribute $A$ of the record $y_i$ do
            If $y_i.A$ appears in an equivalence class $E_j \in E$ then
              If $E_j$ contains a source expression $s_k.B$ then
                Add $y_i.A = x_k.B$ to the where clause of $\sigma'$
              Else add $y_i.A = F_j[\vec{x}]$ to the where clause of $\sigma'$
            Else add $y_i.A = G[\vec{x}]$ to the where clause of $\sigma'$,
                where $G$ is a fresh Skolem function name
    Add $\sigma'$ to $\Sigma$
Return $\Sigma$ as the output of the algorithm
```

A.2 Pseudocode for Phase 3

The subroutine that determines whether a basic SO tgd σ matches a skeleton (T, T') is presented below. If σ matches (T, T'), then the subroutine Match returns a pair of homomorphisms that "embeds" σ to (T, T'). Otherwise, an empty set is returned.

Algorithm Match(σ, (T, T'))
Input: σ is a basic SO tgd, T and T' are tableaux.
Output: (h, g), where h and g "embed" σ into (T, T').

Recall that the input basic SO tgd σ has the form:

```
for $x_1$ in $S_1, x_2$ in $S_2, \ldots, x_n$ in $S_n$
    satisfying $B(x_1, \ldots, x_n)$
exists $y$ in $Q$
    where $\bigwedge_{A \in Atts(y)} y.A = e_i(x_1, \ldots, x_n)$
```

The *satisfying* clause is a conjunction of equalities of the form $x_i.A_i = x_j.A_j$ or $x_i.A_i = c$, where $A_i \in$ Atts(x_i), $A_j \in$ Atts(x_j), and c is a constant. The set $Atts(y)$ denotes the set of attributes in the record y. The *where* clause contains one equality constraint for each attribute of the y record.

In addition, the tableau T has the form:

$$\{u_1 \in R_1, u_2 \in R_2, \ldots, u_k \in R_k; C_T(u_1, \ldots, u_k)\}$$

```
If there exists a pair of homomorphisms $(h, g)$ such that
        (1) for every $1 \le i \le n$, if $x_i \in S_i$ according to $\sigma$,
            then $h(x_i) \in R_i$ according to $T$,
        (2) $C_T(u_1, \ldots, u_k)$ implies $B(h(x_1), \ldots, h(x_n))$, and
        (3) $g(y) \in Q$ according to $T'$
    Return $(h, g)$
Else
    Return $\emptyset$
```

The algorithm that constructs a merged SO tgd by applying the result of the previous Match algorithm on the skeletons is listed below.

Algorithm ConstructSOtgd($(T, T'), B$)
Input: (T, T') is a skeleton and B is a set of pairs $(\sigma, (h, g))$, where σ is a basic SO tgd, and (h, g) "embeds" σ into (T, T').
Output: A Skolemized SO tgd according to (T, T', B).

Recall that T and T' have the form:

$$\begin{aligned} T &= \{x_1 \in S_1, x_2 \in S_2, \ldots, x_p \in S_p; C(x_1, \ldots, x_p)\} \\ T' &= \{y_1 \in Q_1, y_2 \in Q_2, \ldots, y_k \in Q_k; C'(y_1, \ldots, y_k)\} \end{aligned}$$

```
Initialize the SO tgd $\tau$ to be:
          for $x_1$ in $S_1, x_2$ in $S_2, \ldots, x_p$ in $S_p$
              satisfying $C(x_1, \ldots, x_p)$
          exists $y_1$ in $Q_1, y_2$ in $Q_2, \ldots, y_k$ in $Q_k$
              where $C'(y_1, \ldots, y_k)$
For each $(\sigma, (h, g)) \in B$
    Recall that $\sigma$ is a basic SO tgd of the form:
              for $x_{j_1}$ in $S_{j_1}, x_{j_2}$ in $S_{j_2}, \ldots, x_{j_n}$ in $S_{j_n}$
                  satisfying $B_j(x_{j_1}, \ldots, x_{j_n})$
              exists $y$ in $Q_j$
                  where $\bigwedge_{A \in Atts(y)} y.A = e_A(x_{j_1}, \ldots, x_{j_n})$
    Add $B_j(h(x_{j_1}), \ldots, h(x_{j_n}))$ to the satisfying clause of $\tau$
    Add the following conjunction of equalities:
              $\bigwedge_{A \in Atts(y)} g(y).A = e_A(g(x_{j_1}), \ldots, g(x_{j_n}))$
              to the where clause of $\tau$
Return $\tau$
```

We list below the complete set of SO tgds that are constructed in Phase 3 of MapMerge before the actual step of eliminating the subsumed or implied SO tgds.

```
($s_1$) from $(T_1, T_3, b_1)$:
for $g$ in Group
exists $d$ in Dept
where $d$.did = $F[g]$ and $d$.dname=$g$.gname

($s_2$) from $(T_1, T_4, b_1)$:
for $g$ in Group
exists $e$ in Emp, $d$ in Dept
where $d$.did = $F[g]$ and $d$.dname=$g$.gname

($s_3$) from $(T_1, T_5, b_1)$:
for $g$ in Group
exists $p$ in Proj, $d$ in Dept
where $d$.did = $F[g]$ and $d$.dname=$g$.gname

($s_4$) from $(T_2, T_3, b_1)$:
for $w$ in Works, $g$ in Group satisfying $w$.gno = $g$.gno
exists $d$ in Dept
where $d$.did = $F[g]$ and $d$.dname=$g$.gname

($s_5$) from $(T_2, T_4, b_1)$:
for $w$ in Works, $g$ in Group satisfying $w$.gno = $g$.gno
exists $e$ in Emp, $d$ in Dept
where $d$.did = $F[g]$ and $d$.dname=$g$.gname

($s_6$) from $(T_2, T_5, b_1 \wedge b_3)$:
for $w$ in Works, $g$ in Group satisfying $w$.gno = $g$.gno
exists $p$ in Proj, $d$ in Dept
where $p$.did = $d$.did
  and  $d$.did = $F[g]$ and $d$.dname=$g$.gname
  and  $p$.pname = $w$.pname and $p$.budget = $H_1[w]$ and $p$.did = $H_2[w]$

($s_7$) from $(T_2', T_3, b_1 \wedge b_2')$:
for $w$ in Works, $g$ in Group satisfying $w$.gno = $g$.gno and $w$.addr = "NY"
exists $d$ in Dept
where $d$.did = $F[g]$ and $d$.did = $G[w, g]$ and $d$.dname=$g$.gname

($s_8$) from $(T_2', T_4, b_1 \wedge b_2 \wedge b_2')$:
for $w$ in Works, $g$ in Group satisfying $w$.gno = $g$.gno and $w$.addr = "NY"
exists $e$ in Emp, $d$ in Dept
where $e$.did = $d$.did
  and $d$.did = $F[g]$ and $d$.dname=$g$.gname
  and $e$.ename = $w$.ename and $e$.addr = $w$.addr and $e$.did = $G[w, g]$
  and $d$.did = $G[w, g]$

($s_9$) from $(T_2', T_5, b_1 \wedge b_2' \wedge b_3)$:
for $w$ in Works, $g$ in Group satisfying $w$.gno = $g$.gno and $w$.addr = "NY"
exists $p$ in Proj, $d$ in Dept
where $p$.did = $d$.did
  and  $d$.did = $F[g]$ and $d$.dname=$g$.gname
  and  $p$.pname = $w$.pname and $p$.budget = $H_1[w]$ and $p$.did = $H_2[w]$
  and $d$.did = $G[w, g]$
```

In the above list of constructed SO tgds, s_2 is subsumed by s_1. Similarly, s_3 is subsumed by s_1. Moreover, s_5 is subsumed by s_4, which is in turn logically implied by s_6. Finally, s_7 is logically implied by s_9. The remaining SO tgds are s_1, s_6, s_8, and s_9, and none of them is logically implied or subsumed by another. Hence, these four SO tgds are returned by Phase 3 of the algorithm.

A.3 Pseudocode for Phase 4

The algorithm that forms substitutions to be applied during the elimination of residual equality constraints is listed below. Note that the residual equality constraints are created as needed (in the form of actual substitutions) from the input SO tgds. At the end of MapMerge, all the failed substitutions are returned as the final residual equality constraints.

```
Algorithm FindNextSubstitution($\Sigma$, $F$)
Input: $\Sigma$ is a set of SO tgds, $F$ is a set of substitutions that have failed on previous attempts.
Output: Either (1) $U$: a substitution candidate that has not been applied on $\Sigma$ before or, (2) failure if no substitution candidates can be found.

For each SO tgd $\sigma \in \Sigma$
    Recall that $\sigma$ has the form:
            for $x_1$ in $S_1$, $x_2$ in $S_2, \ldots, x_p$ in $S_p$
                satisfying $B(x_1, \ldots, x_p)$
            exists $y_1$ in $Q_1$, $y_2$ in $Q_2, \ldots, y_k$ in $Q_k$
                where $\bigwedge_{1 \le j \le k} \bigwedge_{A \in Atts(y_j)} y_j.A = e_{jA}(x_1, \ldots, x_p)$
    Let $C_\sigma$ be the source context (i.e., the for and satisfying clause of $\sigma$).
        $\{x_1 \in S_1, x_2 \in S_2, \ldots, x_p \in S_p \ ; \ B(x_1, \ldots, x_p)\}$
    For each target expression $y_j.A$ in the where clause of $\sigma$
        Let $\{E_1, \ldots, E_m\}$ be the list of source expressions equated
            with $y_j.A$ directly or indirectly in the where clause of $\sigma$.
        If $m > 1$ then
            There are three cases to consider depending on the number of
            source expressions in the list $\{E_1, ..., E_m\}$.
            Case 1. There is more than one source expression of the
                form $x_i.A$, where $1 \le i \le p$.
                Return conflicting SO tgd $\sigma$ to the user and exit.
            Case 2. There is exactly one source expression of the
                form $x_i.A$, where $1 \le i \le p$.
                Wlog, let $E_1$ denote the source expression $x_i.A$ in the list.
                Let $U = (C_\sigma, E_i, E_1)$ such that $2 \le i \le m$ and $U \notin F$.
                If such a $U$ can be found, return $U$.
                Otherwise, continue.
            Case 3. There are no source expressions of the
                form $x_i.A$, where $1 \le i \le p$.
                Let $U = (C, E_i, E_j)$ such that $i \ne j$ and
                $1 \le i, j \le m$ and $U \notin F$.
                If such a $U$ can be found, return $U$.
                Otherwise, continue.
Return failure (no substitutions can be found)
```

The algorithm that actually applies a substitution on the SO tgds is presented below.

```
Algorithm Substitute($\Sigma$, $U$)
Input: $\Sigma$ is a set of SO tgds, $U$ is a substitution
Output: Success if $U$ can be applied to $\Sigma$. Otherwise, return failure.

    Recall that $U$ is of the form $(C, E_1, E_2)$, where $E_1$ and $E_2$ are source
expressions, and $C$ has the form:
        $\{x_1 \in S_1, x_2 \in S_2, \ldots, x_p \in S_p \ ; \ B(x_1, \ldots, x_p)\}$

For each constraint $\sigma \in \Sigma$
    Assume $\sigma$ is of the form:
            for $x'_1$ in $S'_1$, $x'_2$ in $S'_2, \ldots, x'_n$ in $S'_n$
                satisfying $B'(x'_1, \ldots, x'_n)$
            exists $y_1$ in $Q_1$, $y_2$ in $Q_2, \ldots, y_k$ in $Q_k$
                where $\bigwedge_{1 \le j \le k} \bigwedge_{A \in Atts(y_j)} y_j.A = e_{jA}(x'_1, \ldots, x'_p)$
    If there is a homomorphism $h : \{x_1, ..., x_p\} \to \{x'_1, ..., x'_n\}$ such that
        $B'(x'_1, ..., x'_n)$ implies $B(h(x_1), ..., h(x_p))$
            Replace $h(E_1)$ with $h(E_2)$ in $\sigma$
    Else
        Revert to the original $\Sigma$ from the start of the Substitute routine
        Return failure
```

B. Experimental Evaluation

We conducted a series of experiments on a set of synthetic and real-life mapping scenarios to evaluate MapMerge. We first report on the synthetic mapping scenarios and, using the similarity measure presented in Section 5.1, demonstrate a clear improvement in the preservation of data associations when using MapMerge. We then present results for two interesting real-life scenarios, whose characteristics match those of our synthetic scenarios. We have also implemented some of our synthetic scenarios on two commercial mapping systems. The comparison between the mappings generated by these systems and by MapMerge produced results similar to the previous experiments.

We implemented MapMerge in Java as a module of Clio [10]. For all our experiments we started by creating the mappings with Clio. These mappings were then used as input to the MapMerge operator. To perform the data exchange, we used the query generation component in Clio to obtain SQL queries that implement the mappings in the input and output of MapMerge. These queries were subsequently run on DB2 Express-C 9.7. All results were obtained on a Dual Intel Xeon 3.4GHz machine with 4GB of RAM.

B.1 Synthetic Mapping Scenarios

Our synthetic mapping scenarios follow the pattern of transforming data from a denormalized source schema to a target schema containing a number of relational hierarchies, with each hierarchy having at its top an "authority" relation, while other relations refer to the authority relation through foreign key constraints. For example, Figure 5 shows a source schema that consists of a single relation S. The target schema consists of 2 hierarchies of relations, rooted at relations T_1 and T_2. Each relation in a hierarchy refers to the root via a foreign key constraint from its F attribute to the K attribute of the root. This type of target schema is typical in ontologies, where a hierarchy of concepts often occurs.

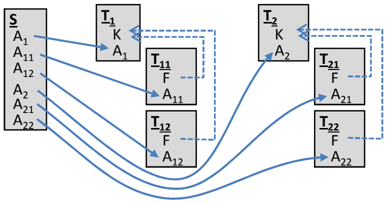


Figure 5: Synthetic Experimental Scenario

The synthetic scenarios are parameterized by the number of hierarchies in the target schema, as well as the number of relations referring to the root in each hierarchy. For our experimental settings we choose these two parameters to be equal, and their common value n defines what we call the complexity of the mapping scenario. In Figure 5 we show an example where $n = 2$. The table in Figure 6 shows the sizes of the experimental scenarios in terms of the number of target relations and the execution times for generating Clio mappings and running the MapMerge operator. We notice that the time needed to execute MapMerge is small (less than 2 minutes in our largest scenario) but dominates the overall execution time as the number of target relations grow.

The graphs in Figure 6 show the results of our experiments on the synthetic mapping scenarios. For each scenario, the source instance contained 100 tuples populated with randomly generated string values. The first graph shows that the target instances generated using MapMerge mappings are consistently (and considerably) smaller than the instances generated using Clio mappings. Here we used the total number of atomic data values on the generated target instances as the size of the target instance (i.e., the product of number

(n) Scenario Complexity	Number of target relations	Mapping generation time (s)	
		Clio	MapMerge
2	6	0.55	0.42
4	20	2.07	0.82
6	42	2.22	2.27
8	72	5.54	6.14
10	110	7.81	10.67
12	156	10.43	22.98
14	210	19.02	54.94
16	272	31.51	107.99

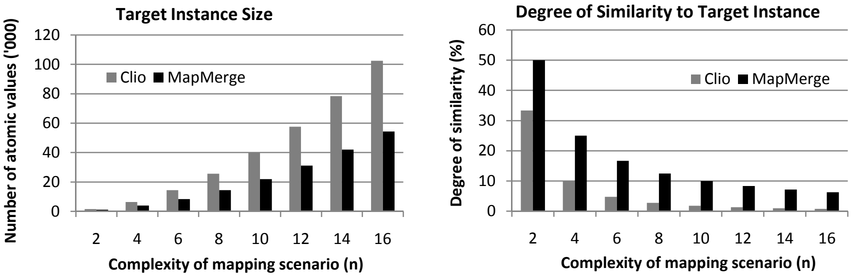


Figure 6: Experiments on synthetic scenarios

of tuples and relation arity, summed across target relations).

The second graph in Figure 6 shows that the source instances have a higher degree of similarity to these smaller target MapMerge instances. The degree of similarity of a source instance I to a target instance J is computed as a ratio of $\mathrm{Sim}(\mathrm{FD}(\tau(I)), \mathrm{FD}(\tau(J))$ to $\mathrm{Sim}(\mathrm{FD}(\tau(I)), \mathrm{FD}(\tau(I)))$, where the latter represents the ideal case where every tuple in $\mathrm{FD}(\tau(I))$ is preserved by the target instance and this quantity simplifies to the expression $|\mathrm{FD}(\tau(I))|$. We notice that the degree of similarity decreases as the complexity of the mapping scenario (n) increases. This is because as n increases, more uncorrelated hierarchies are used in the target schema. In turn, this means that the source relation is broken into more uncorrelated target hierarchies, and hence, they are less similar to the source. The graph shows that Clio mappings, when compared to MapMerge mappings produce target instances that are significantly less similar to the source instance in all cases. Intuitively, this is because most of the Clio mappings will map the source data into each root and one of its child relation. On the other hand, MapMerge factors out the common mappings into the root relation and properly correlates the generated tuples for the child relation with the tuples in the root relation. The effect is that all child relations in the hierarchy are correlated by MapMerge while Clio mappings can only correlate root-child pairs.

B.2 Real-life Scenarios

We consider two related scenarios from the biological domain in this section. In the first scenario, we mapped Gene Ontology into BioWarehouse[4]. In the second, we mapped UniProt to BioWarehouse. The BioWarehouse documentation specifies the semantics of the data transformations needed to load data from various biological databases, including Gene Ontology and UniProt. In the GeneOntology scenario, we extracted 1000 tuples for each relation in the source schema of the mapping, while in the UniProt scenario we extracted the first 100 entries for the human genome and converted them from XML to a relational format to use as a source instance in our experiments. Table 1 shows the number of source and target tables mapped, the number of correspondences used for each mapping scenario, and the number of mapping expressions generated by Clio for each scenario.

Mapping Scenario	Source relations	Target relations	Attribute Correspondences	Clio Mappings
GeneOntology	3	4	5	4
UniProt	13	10	23	14

Table 1: Characteristics of real-life mapping scenarios

Table 2 shows the results of applying MapMerge to the mappings generated by Clio in each scenario. The *generation time* columns show the time needed to generate the Clio mappings and the time needed by MapMerge to process those mappings (i.e., the total execution time is the sum of the two times). The *size of target instance* columns show the total number of atomic data values on the generated target BioWarehouse instance for each scenario. In both cases, the mappings produced with MapMerge reduced the target instance sizes.

Mapping Scenario	Mapping generation time (s)		Size of target instance		Degree of similarity (%)	
	Clio	MapMerge	Clio	MapMerge	Clio	MapMerge
GeneOntology	1.71	0.34	11557	7801	29.7	35.3
UniProt	2.36	2.13	12923	11446	20.8	75.8

Table 2: Results for real-life mapping scenarios

The *degree of similarity* columns present the similarity measure from Section 5.1 for each scenario. This similarity is normalized as a percentage to ease comparison across scenarios and the percentage is with respect to the ideal similarity that a mapping can produce for the scenario. As discussed in Section B.1, this ideal similarity is the number of tuples in the tableau full disjunction of the source, i.e., $|\mathrm{FD}(\tau(I))|$.

On the two real-life settings, MapMerge is able to further correlate the mappings produced by Clio by reusing behavior in mappings that span across different target tableaux and, thus, improving the degree of similarity. This improvement is very significant in the UniProt scenario, where the target schema has a central relation and twelve satellite relations that point back to the central relation (via a key/foreign key). Here, each Clio mapping maps source data to the central and one satellite relation. MapMerge factors out this common part from all Clio mappings and properly correlates all generated tuples to the central relation.

B.3 Commercial Systems

We implemented some of the synthetic scenarios described in Section B.1 in two commercial mapping systems. Provided with only the correspondences from source attributes to target attributes, these systems produced mappings that scored lower than both the Clio and MapMerge mappings with respect to preservation of data associations. For instance, in the synthetic scenario of complexity 2, while the MapMerge mappings had a result of 50% and the Clio mappings 33%, the result for both commercial systems was only 16%. The main reason behind this result is that these systems do not automatically take advantage of any constraints present on the schemas to better correlate generated data and increase the preservation of data associations. The mappings generated by these commercial systems need to be manually refined to fix this lack of correlations.

[4] http://biowarehouse.ai.sri.com/