Data-Driven Understanding and Refinement of Schema Mappings

Ling Ling Yan, IBM Almaden
Renée J. Miller*, Univ. Toronto
Laura M. Haas, IBM Almaden
Ronald Fagin, IBM Almaden
ling.yan@acta.com, miller@cs.toronto.edu, {laura,fagin}@almaden.ibm.com

ABSTRACT

At the heart of many data-intensive applications is the problem of quickly and accurately transforming data into a new form. Database researchers have long advocated the use of declarative queries for this process. Yet tools for creating, managing and understanding the complex queries necessary for data transformation are still too primitive to permit widespread adoption of this approach. We present a new framework that uses data examples as the basis for understanding and refining declarative schema mappings. We identify a small set of intuitive operators for manipulating examples. These operators permit a user to follow and refine an example by walking through a data source. We show that our operators are powerful enough both to identify a large class of schema mappings and to distinguish effectively between alternative schema mappings. These operators permit a user to quickly and intuitively build and refine complex data transformation queries that map one data source into another.

1. INTRODUCTION

The volume of data available online is increasing daily, but our ability to understand it and transform it for new purposes has not kept pace. E-commerce and other data-intensive applications rely on being able to re-use and integrate data from multiple, often legacy sources. To accomplish this task, mappings must be created between the data source (or a set of heterogeneous data sources) and a target or integrated schema [12, 14]. While we have made important advances in our ability to reason about and manage these mappings, a number of important issues remain.

First, the number of possible, even reasonable, mappings between two data sources can be enormous. Users are not able to conceive of all the possible alternatives, and hence may have difficulty finding the correct mapping for a specific application. To address this issue, we are developing a schema mapping tool, Clio, which systematically considers and manages alternative mappings [6, 9, 10]. Clio uses reasoning about queries (similar to the reasoning used in query answering and optimization [2]) to create, manage and rank alternative mappings. However, the final choice of mappings must necessarily be made by a user who understands the semantics of the target application.

Second, schema mappings are typically complex queries. Subtle changes to the mapping, for example, changing a join from an inner join to an outer join, may dramatically change the target data that results. In other cases, the same change may have no effect due to constraints that hold on the source schema. Again, a tool is better able to embody the complex query and constraint reasoning needed to understand these subtleties. However, a means of effectively communicating these subtleties to a user is needed.

Third, the user performing the mapping may not understand the source data or schema fully. Hence, it is important to provide facilities for both schema and data exploration. In addition, it is important to leverage the parts of the data and schema that the user does understand and use these to maximum advantage in forming the schema mapping.

Fourth, given the complexity of the mappings and the often subtle differences between alternative mappings, even an expert user will need help. Specifically, to select a mapping, the user will need to understand the mapping and understand how a particular mapping differs from other mappings. Rather than exposing tangled SQL (or even complex QBE queries) to the user, we believe the best way to achieve such understanding is to use source data to illustrate the results of each mapping. If done in an unprincipled way, such an approach could easily overwhelm the user with data. Hence, we propose to use carefully selected examples that both illuminate a specific mapping (helping the user to understand the mapping) and also illustrate any differences from alternative mappings (helping the user to differentiate mappings).

Finally, data merging queries require the use of complex, non-associative operators [4, 11]. Reasoning about such operators can be extremely difficult, not only for humans, but for query management tools as well [1, 5]. Because the operators may not be associative, even managing compositions of queries can be a difficult task. However, to be scalable to large schemas, mapping tools must necessarily permit users to incrementally create, evolve and compose such complex queries. To address these challenges, we have developed a mapping representation and a set of mapping operators that permit the incremental creation and management of large, complex mappings.

*Supported by an IBM University Partnership Grant and NSERC

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

Our previous work on Clio has addressed the first issue (the generation and management of alternative mappings) by providing a mapping management and reasoning tool [9]. To address the remaining issues, we build on this foundation to present a new data-driven framework for understanding and choosing mappings based on examples. To the best of our knowledge, ours is the first work that supports the understanding and verification of the correctness of complex data transformation queries. Our main contributions are the following.

- We define a powerful mapping representation that facilitates the incremental creation and management of complex mappings over large data sources.
- We introduce the concept of a *mapping example* to ease the tasks of understanding, selecting, and refining schema mappings.
- We identify a set of operators on mapping examples and mappings, and provide a formal semantics for each. We show that our operators provide an appropriate, easy-to-use abstraction for tackling the schema mapping problem.

2. AN ILLUSTRATION

The ultimate goal of schema mapping is not to build the correct query, but to extract the correct data from the source to populate the target schema. Current data transformation (ETL - Extract, Transform and Load) tools and query formation tools focus on building queries or transformation programs, but provide limited support in verifying that the derived data set is correct. If a query or transformation *is* incomplete or incorrect, there is typically no support for refining and correcting it. The user is expected to have a thorough understanding of the data source and to debug complicated SQL queries or procedural transformation programs by hand. In Clio, we permit a user to focus on populating the target with the correct data, rather than on constructing a correct query. By examining and manipulating carefully chosen data examples, the user decides what source data should be combined and transformed, and where in the target this data should be placed.[1] In this section, we introduce our approach by means of a user scenario.

Assume a user wishes to map the data source shown in Figure 1 to the target relation *Kids*, shown in Figure 2(c).[2] From schematic information, the user may indicate a correspondence between attributes or schema constructs (perhaps using value correspondences [9] or schema assertions [12]). In our example, a user has indicated that *Children.ID* corresponds to *Kids.ID*, and that *Children.name* corresponds to *Kids.name*. These correspondences are shown by edges $v1$, $v2$ in Figure 2 (a). Clio shows a sample of the data from the relevant source table(s) (here *Children*) as shown in Figure 2 (b) along with the result of the current mapping (Figure 2 (c)). This allows the user to verify that *Children.ID* not only looks like *Kids.ID* at the schema level, but that the values of *Children.ID* belong in the *Kids.ID* attribute in the target.

[1] Note that we are using data examples to help users understand the extracted, transformed data. This is in contrast to QBE-style approaches that use examples as an abstraction of the query itself [17].

[2] To explain our framework, we will use this very simple source schema which contains only a few tuples. However, our techniques have been designed specifically to handle the complexity and intricacies of both large schemas and large data sets [17].

Children (C)

ID	name	age	mid	fid	docid
001	Kyle	2	201	202	701
002	Maya	4	203	204	702
003	Eric	5	205	206	703
009	Ben	6	401	402	
004	Carmen	10	205	206	703

Parents (P)

ID	name	affiliation	salary
201	Anne	Safeway	50,000
202	Paul	IBM	60,000
203	Jill	Xerox	65,000
204	Mike	NASA	45,000
205	Sue	MGM	56,000
206	Joe	CISCO	67,000
601	Jacky	AMCO	

XmasBox (X)

give	receive
001	003
002	004
003	002
004	006
006	001

PhoneDir (Ph)

ID	type	number
201	Home	201-0001
202	Work	202-0001
202	Cell	202-0002
203	Home	203-0001
203	Cell	203-0002
204	Work	202-0001
205	Work	205-0001
206	Work	206-0001
601	Home	601-0001
701	Cell	701-0001

SBPS (S)

ID	location	time
002	364 Spod...	MW
003	112 Lean...	MF
005	255 Bail...	TWF
009	216 Main...	TWF
004	112 Lean...	MWF

Figure 1: Source Database for Examples

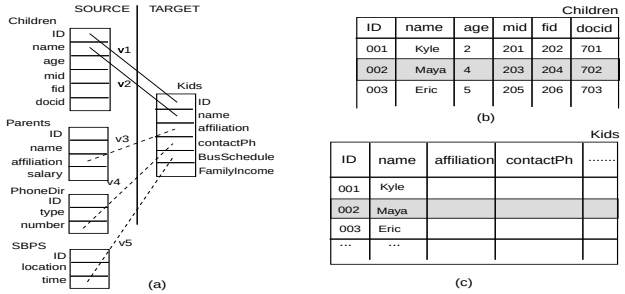

Figure 2: A Schema Mapping Example

Next, the user may indicate that *Parents.affiliation* should map to *Kids.affiliation* (edge $v3$ of Figure 2 (a)). Since *Kids.affiliation* corresponds to a different relation than the rest of the *Kids* data that has been mapped, it is not clear which affiliation value should go with which kid tuple. Assume that Clio is aware of two foreign keys, *mid* and *fid*, both referencing *Parents.ID*. Clio can show the user, using data, these two ways of associating children with affiliations in the source. These two options are illustrated with data examples (Figure 3). If the user has selected the *Children*'s tuple for *Maya* as an example, then the highlighted rows of Figure 3 can be used to illustrate the differences in these two scenarios. Because the user is familiar with this specific example, she quickly realizes that *mid* and *fid* are *mother ID* and *father ID*, respectively. She selects the scenario that has the desired target semantics. For our example, she identifies Scenario 1, where children are associated with their fathers' affiliations.

Next, the user decides to populate *Kids.contactPh* with source data. The user notices that phone numbers in the source all appear in the *PhoneDir* relation, but is unsure how to associate phone numbers with kids. She therefore asks Clio to perform a **data walk** to find associations between children and phone numbers. In response, Clio produces several scenarios, of which two are shown in Figure 4. Scenario 1 associates children with their fathers' phone numbers, and Scenario 2, with mothers' phone numbers. Again,

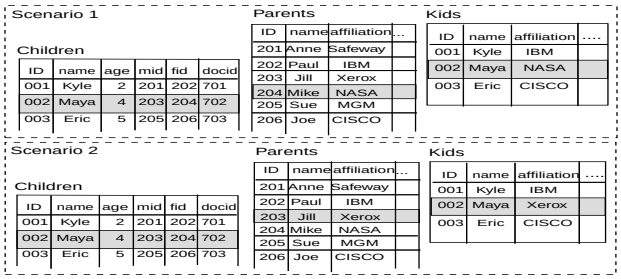

Figure 3: Associating parents and children.

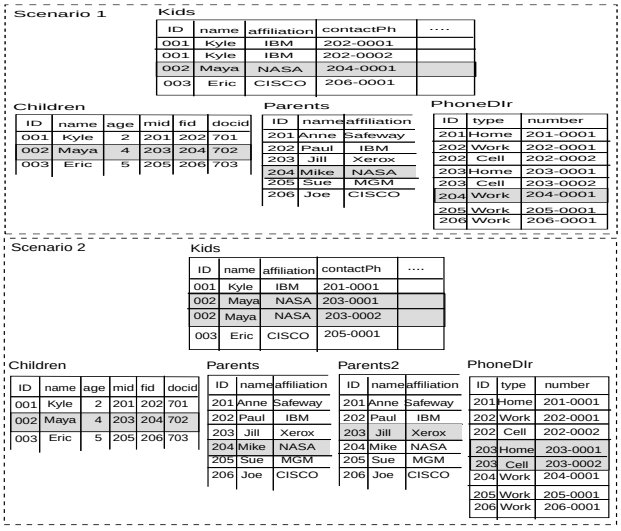

Figure 4: Associating children and phone numbers

both scenarios are illustrated with data. To illustrate the second alternative, a second copy of the *Parents* relation is introduced to indicate that children are being associated with values in two parent tuples. The user is able to view and manipulate the illustrations, perhaps asking for different example tuples, in order to gain sufficient understanding of the alternatives to be able to select the one that reflects her desired semantics. For this example, we assume the user chooses the second scenario and adds a correspondence from *PhoneDir.number* to *Kids.contactPh* (v4 in Figure 2 (a)).

Now assume that the user wants to populate *Kids.BusSchedule*. From the source schema, it is not obvious where to find this data, unless the user knows that the cryptic name *SBPS* stands for "School Bus Pickup Schedule". Hence, our user may not have enough knowledge of the source to request a data walk to a specific relation. However, she guesses that any tuple that records the school bus schedule of a child might carry the ID of the child. So she chooses an ID value, say Maya's ID, 002, and asks Clio to **chase** this value. Clio locates all occurrences of this values in other relations. Clio finds that 002 appears in one attribute of *SBPS* and in two attributes of *XmasBox*. This result is demonstrated using the Maya example, as shown in Figure 5. The user, perhaps after looking at the relevant data associated with the value 002 in these new relations, chooses the first scenario as the right way of associating bus schedules with children. To complete the mapping, she adds a correspondence v5 from *SBPS.time* to *Kids.BusSchedule* (Figure 2 (a)).

During this process, Clio has maintained and manipulated a mapping between source and target, and used it to generate examples. Clio uses input from the user, along with sophisticated query and constraint reasoning to derive this mapping. Clio also uses target constraints (provided as part of the schema or input by the user) as part of mapping creation. For example, a target constraint may indicate that every Kid tuple must have an ID value. From this constraint, Clio would know not to include SBPS or Parent values in the target if they are not associated with a Children tuple. Using this constraint, and the user choices made in this example, Clio would derive the following mapping.

create view Kids **as**
select C.ID **as** ID, C.name **as** name, P.affiliation **as** affiliation, D.number **as** contactPh, S.time **as** BusSchedule
from Children C
left join Parents P **on** C.fid = P.ID
left join Parents P2 **on** C.mid = P2.ID
left join PhoneDir D **on** C.ID = D.ID
left join SBPS S **on** C.ID = S.ID
where C.ID **is not null**;

Notice that we use left outer joins extensively to make sure that even kids without affiliation, contactPh, or BusSchedule are extracted from the source. By showing the result of the mapping as well as sample source data, Clio allows the user to fine-tune this mapping. For example, if the user is interested only in children who have a bus schedule, she can, upon seeing a null in the *BusSchedule* column, indicate that *BusSchedule* is really a required field. Clio would then change this left outer join to an inner join.

This simple example illustrates the power of data in supporting the construction of schema mappings, especially when the source schema is unfamiliar or contains cryptic names. A simple data browsing facility could easily make the user feel lost in a jungle of data. To prevent this, we have developed a framework for showing carefully selected examples to the user, examples that change as needed in response to changes in the mapping. The examples are chosen to permit a user both to understand a mapping and to understand how it differs from alternative mappings. Using examples, the user can "walk around" inside the database, play with data, understand how data is organized, and see how to combine pieces of data meaningfully in the context of the target schema. The user can walk along paths known to Clio (a data walk), or actively discover new ways of connecting data (a data chase). By following the "tracks" of the user, Clio gains enough semantic knowledge to compose the often complex mapping queries. The remainder of the paper focuses on formalizing this framework. We describe the major building blocks (mappings and examples) in the next two sections.

3. MAPPINGS

Tuples in a target relation can often be computed in several ways. For instance, tuples in a target relation "Kids" may be computed differently for children in public schools and children who are home schooled. Generally, portions of a target relation are computed by separate queries. The results of these queries are then combined to form the final content of the target relation. We define a mapping as a query on the source schema that produces a subset of a target relation. Thus, a mapping defines one (out of possibly many) ways of forming target tuples.

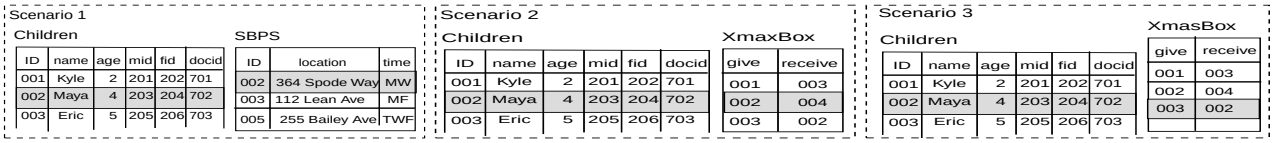

Figure 5: Chasing Data Values

Mapping construction consists of three activities. The first activity involves determining where and how source data should appear in the target. We call this activity *determining correspondences*. For example, in the target, we may wish to populate the *FamilyIncome* field with the sum of a kid's parents' salaries (from the source *Parents.Salary* field). During this activity, we determine how source values should be combined and transformed to make up target values. The second activity involves determining how source tuples from different relations should be combined to form a target tuple. This activity, *data linking*, determines the conditions used to join source relations. For example, to associate children with affiliations, we need to determine first which parent tuples belong with a child. The third activity involves determining which of the joined source tuples should be used in forming a target tuple. We call this activity *data trimming*. For example, a user may not want all children to appear in the target, but only those under the age of seven. Similarly, a user may not want to see information about a person's income in the target, unless that income is associated with a child.

To support mapping creation, we define in this section a formal notion of mapping that represents the decisions made in each of the three mapping activities.

Preliminaries Let $\mathcal{A}$ be a set of attributes, where for each $A \in \mathcal{A}$ there is an associated domain $dom(A)$ of values. A *scheme* S is a finite set of attributes. A *tuple* t on S is an assignment of values to the attributes of S. For an attribute $A \in S$, $t[A] \in dom(A)$ denotes the value of t on A. We may denote a tuple using a set of attribute-value pairs: $t = \{A_1 : v_1, A_2 : v_2, ...\}$. A *relation* on scheme S is a named, finite set of tuples on S. When confusion will not arise, we may use the same symbol for both a relation and its name. A database is a set of relations over mutually disjoint schemes where the database schema is the corresponding set of relation names. We make the common assumption that the relations in the source database do not contain any tuples that are null on all attributes, as the way such a tuple should be reflected in a schema mapping is unclear. A *predicate* P over a scheme S maps tuples on S to true or false. A predicate is *strong* if it evaluates to false on every tuple that is null for all attributes in S [4]. A *join predicate* is a strong predicate over attributes in two relations. Note that join predicates in SQL are strong [4]. A *selection predicate* is a predicate over attributes in one relation. We do not require selection predicates to be strong. Mappings may require multiple copies of a relation. Without loss of generality, and with notational simplicity in mind, we assume that if multiple copies of a relation are required, each copy (and its attributes) have been given unique names which in turn may be used unambiguously in predicates.

3.1 Correspondence to the Target

To begin constructing a mapping, we must know where data should appear in the target, that is, in what attribute and also how it should appear. To indicate this we use *value correspondences* [9]. A value correspondence is a function defining how a value (or set of values) from a source database can be used to form a value in a target relation. Our previous work has shown that value correspondences are a natural, intuitive abstraction for users to enter semantic information about where data should be mapped [9]. Here, we assume that users (or an automated tool [7]) are able to provide value correspondences on which to base the mapping construction.

DEFINITION 3.1. *A* **value correspondence** *v is a function over the values of a set of source attributes $A^1, ..., A^k$ that computes a value for target attribute B. So, $v : dom(A^1) \times ... \times dom(A^k) \rightarrow dom(B) \cup \{null\}$.*

EXAMPLE 3.2. *In the example of Section 2, we showed several value correspondences, including a simple (identity) function, $v_{id} : Children.ID \rightarrow Kids.ID$, to map values in Children.ID to values in the target attribute Kids.ID. Other correspondences might use several source relations or even several copies of the same source relation. To populate the target attribute Kids.FamilyIncome, we could use the value correspondence v_{sal}: Parents.Salary + Parents2.Salary $\rightarrow$ Kids.FamilyIncome.*

Note that value correspondences are simply functions on attribute values. They do not indicate which values will be included in the mapping nor how different values will be associated. For instance, v_{sal} does not indicate which tuples from *Parents* and *Parents2* are to be combined.

3.2 Data Linking

We use query graphs to represent the linkage among source tuples, that is, how source tuples can be combined correctly in the context of a target relation [4].

DEFINITION 3.3. *A* **query graph** *$G = (N, E)$ over a database schema S is an undirected, connected graph where N is a subset of the relation names in S. Each edge $e = (n_1, n_2) \in E$ is labeled by a conjunction of join predicates on attributes in the union of the schemes of n_1 and n_2.*

Intuitively, a query graph defines a way of associating tuples from different source relations. The nodes of a query graph are source relation names. The edges represent join predicates between pairs of source relations.

EXAMPLE 3.4. *Three example query graphs are shown in Figure 6. Each represents one way of linking tuples for the source database of Figure 1. For graph G, PhoneDir and Parents tuples are associated if they have common ID values. Similarly, Parents and Children tuples are associated if Parents.ID = Children.mid.*

Before defining the scope of a mapping, that is, which source tuples are to be included in the mapping, we must

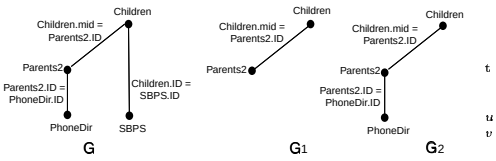

Figure 6: Query (Sub)Graphs

	Children						Parents2						
	ID	name	age	mid	fid	docid	ID	name	aff'n	sal			
t	002	Maya	4	203	204	702	203	Jill	Xerox	65K			
	Children						Parents2				PhoneDir		
	ID	name	age	mid	fid	docid	ID	name	aff'n	sal	ID	type	number
u	002	Maya	4	203	204	702	203	Jill	Xerox	65K	null	null	null
v	002	Maya	4	203	204	702	203	Jill	Xerox	65K	203	Home	203-0001

Figure 7: Examples of data associations.

consider possible interpretations of a query graph. Clearly, one interpretation is as a join query. However, to support the data merging semantics of mappings, we may also want to interpret a query graph as an outer join query or as a combination of joins and outer joins. Informally, we will refer to the set of all possible tuple associations that conform to a query graph as its data associations. To define this notion formally, we begin by defining the set of **full data associations**.

DEFINITION 3.5. *Let $G(N, E)$ be a query graph. Let R_1, ..., R_n be all the nodes of G, that is, $N = \{R_1, ..., R_n\}$. The set of* **full data associations** *of G is defined as $F(G) = \sigma_P(R_1 \times ... \times R_n)$ where P is the conjunction of all edge predicates in G.*

Given a query graph G, the full data associations of G can be computed by an inner join query, based on G. Note that the join need not be lossless, so there may be tuples from the source relations that do not contribute to any full data association. Intuitively, a full data association is "full" because it involves tuples from all nodes of G. In defining mappings, it may also be useful to consider non-full associations.

DEFINITION 3.6. *Let $G = (N, E)$ be a query graph. For each induced, connected subgraph $J = (N_J, E_J)$ of G, if d is a full data association of J, then d padded with nulls on all attributes in $N - N_J$ is a* **possible data association** *of G. The* **coverage** *of d is N_J, denoted as coverage$(d) = N_J$.*

EXAMPLE 3.7. *Consider Query Graph G in Figure 6. The node set {Children, Parents2} induces graph G_1, a connected subgraph of G. Referencing Figure 1, we see that the tuple t in Figure 7 is a full data association of G_1.*
The node set {Children, Parents2, PhoneDir} also induces a connected subgraph, G_2, shown in Figure 6. We can pad t with nulls to form a possible data association u of G_2. The new association u is not a full data association of G_2 since it does not involve a tuple of PhoneDir. By referencing the data of Figure 1, we see that the tuple v is a full data association on G_2.

In the above example, v provides all the information provided by u and more. Hence, u is made redundant in some sense by v. Generally, non-full data associations may be important to include in a mapping if there is no full data association that contains more information. This idea has been formalized using the notion of subsumption [16].

DEFINITION 3.8. *A tuple t_1* **subsumes** *a tuple t_2 if t_1 and t_2 have the same scheme and $t_1[A] = t_2[A]$ for all attributes A where $t_2[A] \neq null$. Moreover, t_1* **strictly subsumes** *t_2 if $t_1 \neq t_2$.*

Intuitively, a strictly subsumed tuple is "redundant" since it repeats information that is already represented by another tuple. In our example, v strictly subsumes u. The minimum union operator (defined below) removes such redundancies.

DEFINITION 3.9. *The* **minimum union** *of two relations R_1 and R_2, denoted $R_1 \oplus R_2$, is the outer union[3] of R_1 and R_2 with strictly subsumed tuples removed.*

EXAMPLE 3.10. *Let R_1 be the set of full data associations of G_1 and let R_2 be the set of full data associations of G_2. So $R_1 = Children \bowtie_{mid=ID} Parents$ and $R_2 = Children \bowtie_{mid=ID} Parents \bowtie_{ID=ID} PhoneDir$. Then $R_1 \oplus R_2 = R_2$. It is easy to verify that all tuples in R_1, after being padded with attributes of PhoneDir, are strictly subsumed by tuples in R_2. This would not be true if some parents had no phone numbers in the data source of Figure 1.*

DEFINITION 3.11. *Let G be a query graph. Let $S(G)$ be all the possible data associations of G. Let $U(G)$ be the set of tuples in $S(G)$ that are strictly subsumed by another tuple in $S(G)$. We define $D(G)$, the set of all data associations of G, as $D(G) = S(G) - U(G)$. A* **data association** *is then a tuple in $D(G)$.*

Referring to Definition 3.6, we see that $D(G)$ can be computed by combining full data associations over all induced and connected subgraphs of G, using minimum union. That is, if $J_1, ..., J_w$ are all the induced and connected subgraphs of G, then $D(G) = F(J_1) \oplus ... \oplus F(J_w)$. Galindo-Legaria calls $D(G)$ the *full disjunction* of query graph G [4]. The full disjunction has been recognized as providing a natural semantics for data merging queries [5, 11].

EXAMPLE 3.12. *The set of all induced, connected subgraphs of G from Figure 6 is the set of subgraphs induced by the following sets of nodes:*[4] *{C}, {P},{Ph},{S},{C, P},{C, S}, {P, Ph}, {C, P, Ph}, {C, P, S}, {C, P, Ph, S}. The set of data associations of G, $D(G)$, can be computed as follows (where p_1, p_2, and p_3 are the predicates "C.mid = P.ID", "C.ID = S.ID", and "P.ID = Ph.ID", respectively).* $D(G) = C \oplus P \oplus Ph \oplus S \oplus (C \bowtie_{p_1} P) \oplus (C \bowtie_{p_2} S) \oplus (P \bowtie_{p_3} Ph) \oplus (C \bowtie_{p_1} P \bowtie_{p_3} Ph) \oplus (C \bowtie_{p_1} P \bowtie_{p_2} S) \oplus (C \bowtie_{p_1} P \bowtie_{p_2} S \bowtie_{p_3} Ph)$. *In Figure 8, we show the tuples in $D(G)$. Each data association is tagged with its coverage. Due to space constraints, we have removed the Children.docid and SBPS.location attributes since these are not used in the mapping (however, these attributes are part of the data association).*

3.3 Data Trimming

Once a query graph G is established, we preserve all possible linkages among source tuples by computing $D(G)$. However, not all tuples of $D(G)$ may be semantically meaningful

[3] The **outer union** is the union of R_1 (padded with nulls on attributes that are in R_2 but not in R_1) and R_2 (padded with nulls on all attributes that are in R_1 but not R_2).

[4] We use the short hand for the relations indicated in Figure 1, with P standing for *Parents2* here.

		Children(C)					Parents(P)				PhoneDir(Ph)			SBPS(S)	
	cover	ID	name	age	mid	fid	ID	name	affil	salary	ID	type	number	ID	time
1	CPPhS	002	Maya	4	203	204	203	Jill	Xerox	65000	203	Home	203-0001	002	MW
2	CPPhS	002	Maya	4	203	204	203	Jill	Xerox	65000	203	Cell	203-0002	002	MW
3	CPPhS	003	Eric	5	205	206	205	Sue	MGM	56000	205	Work	205-0001	003	MF
4	CPPhS	004	Carmen	10	205	206	205	Sue	MGM	56000	205	Work	205-0001	004	MWF
5	CPPh	001	Kyle	2	201	202	201	Anne	Safeway	50000	201	Home	201-0001		
6	CS	009	Ben	6	401	402								009	TWF
7	PPh						601	Jacky	AMCO		601	Home	601-0001		
8	Ph										701	Cell	701-0001		
9	S													005	TWF

Figure 8: The data associations for graph G of Figure 6.

in the context of the target relation. For instance, Row 7 of Figure 8 contains information that is not related to any children, and hence may not be useful for our target relation *Kids*. The goal of data trimming is to specify which data associations match the semantics of target relation.

Let $J = (N_J, E_J)$ be an induced connected subgraph of G, and let $D(G, J)$ be the set of data associations in $D(G)$ whose coverage is N_J. Assuming that no source tuples are null on all attributes, then for any induced connected subgraphs J_1, J_2 of G, we have $D(G, J_1) \cap D(G, J_2) = \emptyset$ if $J_1 \neq J_2$. That is, if $J_1, ..., J_w$ are all the induced connected subgraphs of G, then we can *partition* $D(G)$ into w subsets, $D(G, J_1), ..., D(G, J_w)$. Each $D(G, J_i)$ is called a *category* of $D(G)$. Data associations in some of these categories may all be too incomplete to include in the mapping. For instance, in Figure 8, data associations with coverage PPh (Row 7) are meaningless in the context of the target relation Kids, since they are not related to any children. Generally, a user may determine that some categories $D(G)$ (that is, specific $D(G, J_i)$) must be excluded from the mapping because they have **incomplete coverage**.

Alternatively, a user may wish to exclude data associations that satisfy some selection predicate or other criteria. For example, a user may wish to exclude children who are seven or older from the mapping, or even exclude children whose age is not indicated (that is, whose age is null). Generally, the user may determine that some data associations must be excluded from the mapping because they have **invalid values**, that is, the associations fail to satisfy certain conditions on the values they contain.

In our mapping definition, data associations with incomplete coverage or invalid values can both be excluded using filters as shown by the following example.

EXAMPLE 3.13. *In creating a mapping for the Kids target relation, a user may wish to exclude all target tuples that have a null Kids.ID. The user may indicate this by specifying a not-null constraint in the target schema. In the mapping, we represent this using a target predicate over the target relation Kids: Kids.ID <> null. Alternatively, a user may indicate that unless a data association involves Children, it is not of interest. This choice can be represented by the following source predicate:* $\neg$ *(C.ID = null* $\wedge$ *C.name = null* $\wedge$ *C.age = null* $\wedge$ *C.mid = null* $\wedge$ *C.fid = null* $\wedge$ *C.docid = null). A simpler predicate may be used if one or more attributes of Children are constrained to be non-null. Note that these two predicates on Kids and Children are not necessarily equivalent. Finally, a user may also specify constraints on source or target values. The following examples constrain the FamilyIncome attribute of the target to be under $100,000 and the source attribute Age to be under 7.*

Kids.FamilyIncome < $100,000 *Children.Age < 7*

3.4 Putting it all together: Mapping Definition

We outlined how decisions made during each of the three activities of creating correspondences, data linking and data trimming can be represented using value correspondences, query graphs and selection predicates respectively. We combine these three components to build a representation of a mapping.

DEFINITION 3.14. *Let $N = \{R_1, ..., R_n\}$ be a set of source relation names and $T(B_1, ..., B_m)$ be a target relation name. A* **mapping** *from N to T is a four-tuple $< G, V, C_S, C_T >$ where:*

- *G is a connected query graph with node set N;*
- *$V = \{v_1, ..., v_m\}$ is a set of* value correspondences *where each $v_i : dom(A_1^i) \times ... \times dom(A_{k_i}^i) \rightarrow dom(B_i) \cup \{null\}$;*
- *$C_S = \{p_1^s, ..., p_x^s\}$ is a set of selection predicates over source relations in N; and*
- *$C_T = \{p_1^t, ..., p_y^t\}$ is a set of selection predicates over the target relation T.*

The **mapping query** *defined by M is the following.*

select *
from (**select** $v_1(A_1^1, ..., A_{k_1}^1)$ **as** B_1, ..., $v_m(A_1^m, ..., A_{k_m}^m)$ **as** B_m
from $D(G)$
where p_1^s **and** ... **and** p_k^s)
where p_1^t **and** ... **and** p_k^t

Intuitively, a mapping defines the relationship between a target relation and a set of source relations. This relationship is defined using three components. The first component is the query graph G, which defines how data in the source relations are to be linked, or "pieced" together, to produce all data associations. The second component is the set V of value correspondences, which defines how these data associations can be translated into tuples in the target relation. The final component includes two sets of filters, C_S and C_T, which define conditions that source and target tuples, respectively, must satisfy. These definitions are illustrated by the following example.

EXAMPLE 3.15. *Consider the query graph G pf Figure 6. Let $V = \{v_1, v_2, v_3, v_4, v_5\}$ be value correspondences for Kids.ID, Kids.name, Kids.affiliation, Kids.contactPh, and Kids.BusSchedule, respectively, of the target (Figure 2). Let v_1, v_2, v_3, v_5 be identity functions defined on attributes C.ID, C.name, P.affiliation, and S.time, respectively. Let v_4 be the function* `concat`*, defined on attributes Ph.type and Ph.number, with the following signature:* concat: String $\times$ String $\rightarrow$ String. *The function* `concat` *produces a new string s by concatenating its first parameter, ":", and its second parameter.*

Let C_S = {"C.age < 7"}. Let C_T = {"Kids.ID ≠ null" }. Then (G, V, C_S, C_T) is a mapping. The query defined by this mapping is the following.

```
select *
from ( select C.ID as ID, C.name as name,
         P.affiliation as affiliation,
         concat(Ph.type, Ph.number) as contactPh,
         S.time as BusSchedule
       from D(G)
       where C.age < 7 )
where ID ≠ null
```

We show in the full version of this paper that this mapping representation can be used to represent arbitrary combinations of join and outer join queries [17]. The impact of this result is that we can use this definition to represent and manipulate a powerful class of mapping queries.

4. MAPPING EXAMPLES

We have designed a mapping representation that supports the incremental development of mapping queries. We now consider how to use source data to assist users in constructing these mappings and in verifying that they are semantically correct. The centerpiece of this technique is the concept of mapping *examples*.

4.1 Definition of Example

For a mapping $M = (G, V, C_S, C_T)$, let Q_M be the mapping query defined by M, as in Definition 3.14. Query Q_M is a query over all data associations of G. For a specific data association $d \in D(G)$, we define $Q_M(d)$ as the result of the mapping query applied only to $\{d\}$. The mapping $\phi(M) = (G, V, \emptyset, \emptyset)$ (with mapping query $Q_{\phi(M)}$) is the mapping defined by M without any source or target filters.

DEFINITION 4.1. *An* **example** *e of a mapping $M = (G, V, C_S, C_T)$ is a pair $e = (d, t)$, where $d \in D(G)$, and $t = Q_{\phi(M)}(d)$. Example e is a* **positive example** *if $t = Q_M(d)$ (that is, if d satisfies all the conditions in C_S, and t satisfies all conditions in C_T). Otherwise, e is a* **negative example**.

A positive example demonstrates how a set of source tuples are combined together to contribute towards the target relation. It provides insight into the target tuples a mapping will produce. A negative example demonstrates a case where source tuples are combined correctly (using the valid join conditions) but fail to make it into the target. A negative example provides insight into what has been removed from the target by filter predicates. An illustration is then any set of examples for a mapping. In Clio, we are interested only in illustrations that provide a satisfactory showcase of the behavior of a mapping.

An illustration includes a set of data associations. Data associations for the mapping described in Example 3.15 are shown in Figure 9. We use this as a running example for the rest of this section. Each data association is tagged with a field that encodes its coverage and that indicates whether the example it induces is positive or negative.

4.2 Sufficient Illustrations

A *sufficient illustration* is one that demonstrates all aspects of a mapping. We formally define this by considering how to illustrate each component of the mapping: the query graph, the filters, and the value correspondences.

Sufficient Illustration of a Query Graph

A query graph defines the data associations of a mapping. As discussed in Section 3.3, the set of all data associations defined by query graph G can be partitioned according to coverage of the data associations. Each connected subgraph J defines one component of the partition $D(G, J)$ called a category. It is possible some of these categories are empty (Section 3.3). To sufficiently illustrate a query graph, an illustration must include at least one example induced by a data association from each category of $D(G)$ that is not empty.

DEFINITION 4.2. *Let I be a set of examples of a mapping $M = (G, V, C_S, C_T)$. Then I is a* **sufficient illustration of the query graph** *$G = (N, E)$ if it satisfies the following property. For each induced, connected subgraph $J = (N_J, E_J)$ of G, if there exists a data association in $D(G)$ whose coverage is N_J then I contains an example $(d, t,) \in I$ whose coverage is N_J.*

EXAMPLE 4.3. *The illustration shown in Figure 9 is a sufficient illustration of query graph shown in Figure 6. Notice that if we remove one of the examples with coverage CPPhS, it remains sufficient. However, if we remove the example with coverage PPh, this illustration is no longer sufficient in regard to the query graph shown in Figure 6, since it does not illustrate data associations with coverage PPh. Also notice that there are no parents in the database who have children and no phone. Hence, there exists no data association with coverage CP. For similar reasons, there is no example with coverage C or CPS.*

Sufficient Illustration of Filters

We refine the above definition to develop a sufficient illustration of filters.

DEFINITION 4.4. *Let I be a set of examples of mapping $M = (G, V, C_S, C_T)$. Then I is a* **sufficient illustration of the data trimming predicates** *C_S and C_T if, for each induced, connected subgraph $J = (N_J, E_J)$ of G, the following conditions hold:*
- *if there exists a positive example (d, t) where coverage$(d) = N_J$, then I contains such a positive example; and*
- *if there exists a negative example (d, t) where coverage$(d) = N_J$, then I contains such a negative example.*

Intuitively, we illustrate the filters of a mapping by illustrating the effect of the filters from two perspectives. First, we illustrate the data associations with incomplete coverage that are eliminated by the filters (Section 3.3); these data associations do not carry enough semantics to be meaningful in the context of the target. Second, we illustrate data associations that have enough coverage but fail to satisfy the filters for other reasons. In this sense, the illustration provides some insight into the effect of filters in removing data associations with invalid values (Section 3.3).

Sufficient Illustration of Value Correspondences

We focus on three salient properties of value correspondences to help users understand the correspondence. First, we want the user to understand how data associations are being transformed by a correspondence. To illustrate this,

	Children(C)				Parents(P)		PhoneDir(Ph)		SBPS(S)	
coverage	ID	name	age	mid	ID	affiliation	ID	n umber	ID	time
C,P,Ph(+)	001	Kyle	2	201	201	Safeway	201	201-0001		
C, S(+)	009	Ben	6	401					009	TWF
C,P,Ph,S(+)	002	Maya	4	203	203	Xerox	203	203-0001	002	MW
C,P,Ph,S(+)	002	Maya	4	203	203	Xerox	203	203-0002	002	MW
C,P,Ph,S(-)	004	Carmen	10	205	205	MGM	205	205-0001	004	MWF
P,Ph(-)					601	AMCO	601	601-0001		
Ph(-)							701	701-001		
S(-)									00 5	TWF

Figure 9: Data Associations for a Sufficient Illustration

for each target attribute B we ensure there is an example that creates a tuple with a non-null value on B (if such an example exists). Second, we want the user to understand how complete the mapping is, that is, whether all the target tuples created will have a non-null values for a particular attribute. Third, we want the user to understand the properties of the source columns, which have an impact on the behavior of the value correspondence. For instance, we want the user to understand how a value correspondence behaves when one or more of its source columns are null.

DEFINITION 4.5. *Let I be a set of examples of mapping $M = (G, V, C_S, C_T)$. Then I is a* **sufficient illustration of the value correspondences** *V if for each induced connected subgraph $J = (N_J, E_J)$ of G and for each target attribute B:*
• *if there exists a positive example (d, t), where $coverage(d) = N_J$ and where $t[B] \neq null$, then I contains such an example;*
• *if there exists a positive example (d, t), where $coverage(d) = N_J$ and where $t[B] = null$, then I contains such an example.*

Sufficient Illustration of Mapping
Combining these definitions, we obtain a sufficient illustration for a mapping.

DEFINITION 4.6. *Let I be a set of examples of mapping $M = (G, V, C_S, C_T)$. Then I is a* **sufficient illustration** *of M if it is a sufficient illustration of G, V and the filters C_S and C_T.*

4.3 Illustrations with Focus

Sufficiency is a way of ensuring all aspects of a mapping are illustrated. In addition, we permit a user to select values they understand and use the examples induced by these values. The intuition is that a user may be familiar with specific data values (for example, a user may know a specific child Maya and know how data related to Maya should appear in the target). We want to take maximal advantage of such values in our illustrations.

DEFINITION 4.7. *Let I be a set of examples of mapping $M = (G, V, C_S, S_T)$, where $G = (N, E)$. Let $F \in N$ be a distinguished relation called the* **focus relation** *with scheme S_F. Let $f \subseteq F$ be a distinguished set of tuples of F called the* **focus tuples**. *Then I is* **focussed** *on f if, for every data association $d \in D(G)$ where $\Pi_{S_F}(d) \in f$, the resulting example (d, t) is in I.*

Intuitively, an illustration is focussed on f if all data associations involving tuples of f are included in the illustration.

EXAMPLE 4.8. *The illustration shown in Figure 9 is focussed on Children, with Children tuples with ID values 001, 002, 004, and 009, as the focus tuples. All data associations from Figure 8 that involve these children are included in the illustration. This illustration is not focussed on the Parents tuple identified by ID 205, since the data association shown in Row 3 of Figure 8, which involves Parents "205", is not included in this illustration. This illustration does not provide a complete picture of the behavior of Parents 205 under the mapping. However, this illustration allows the user to learn everything about the children mentioned.*

Given a mapping, Clio is able to build a sufficient illustration that provides an entry point into the data source. We make use of evaluation and optimization techniques for the minimal union operator to efficiently compute D(G) [5] and to efficiently select a minimal sufficient illustration. From this starting point, a user may select subsets of the illustration or subsets of the original source relations to focus the illustration on specific data of interest.

5. MAPPINGS OPERATORS

Illustrations are designed to help a user understand mappings, understand differences between different mappings, and identify shortcomings or flaws in a mapping. The next step is to permit a user to act on the knowledge gained through illustrations to extend or refine a mapping. We do this by providing a suite of operators for manipulating mappings. Due to space limits, we are not able to describe all the operators here. Instead, we describe a few of our operators concentrating on how the operators permit users to easily and effectively make use of the sophisticated query reasoning and query management knowledge encapsulated within our tool.

After examining an illustration of a mapping, a user may invoke a mapping modification operator. The result of the operator is a new mapping or a set of new alternative mappings. Each new mapping is illustrated by a set of examples that are derived naturally from the current examples. Given the incremental nature of this process and given that our goal is to support the creation of complex mappings, it is important that the invocation of these operators be painless for the user and that she be able to quickly discern their results.

The operators can be grouped by their effect on the mapping. First, correspondence operators permit a user to change the value correspondences. In Section 2, we gave an example of a user adding a new value correspondence to a mapping (the correspondence $v3$ of Figure 2). In response to this operator, Clio determined a set of alternative mappings (represented using possibly different query graphs) and helped the user decide among them using illustrations. Second, data trimming operators modify the source and target fil-

ters of a mapping. Data trimming operators do not change the query graph of a mapping, but rather change the set of source and target tuples included in the mapping. Trimming operators are illustrated using positive and negative examples so a user can see the effect of the different filters. Finally, we provide a set of data linking operators, which directly change the query graph of the mapping. We focus on two specific data linking operators to explain our approach. Additional operators are describe elsewhere [17].

Data linking operators allow the users to extend a query graph. Using these operators, a user can incrementally build potentially complex mappings. However, the user does not need to undertake the daunting task of specifying the structure of the new query graph or how the current graph should be changed. Rather, the user may invoke these operators **using data** by indicating what source data is missing from the current illustration. We describe here two data linking operators, *data walk* and *data chase*. In a data walk, the user knows where the missing data resides in the source or more specifically what source relation(s) contain this data. Using this information, Clio infers possible ways of augmenting the query graph to include the new data and illustrates each new mapping alternative. In a data chase, the user does not know where the missing data resides. The chase permits her to explore the source data incrementally to locate the desired data. This is done by exploiting data values included in the data associations in the current illustration. These values are "chased" through the source database to discover new ways of linking the illustration data with other data values until the desired data is found. For both operators, Clio provides the complex query management required to create the new mapping and to illustrate it.

For both of these operators, there may be many ways to extend the query graph. However, the majority of these can quickly be dismissed by a user as semantically invalid, leaving a few viable alternatives that the user can explore further using other mapping or example operators.

5.1 Data Walk

A *data walk* makes use of Clio's knowledge of the source schema (which is gathered from schema and constraint definitions and from mining the source data, views, stored queries and metadata). Using this knowledge, Clio deduces a set of possible ways of joining relations each specified by a query graph [9]. The specific techniques used are not relevant to the current discussion. In general, however, Clio has knowledge of a (possibly empty) set of potential query graphs for joining any two source relations.

To define the walk operator, we first define a walk. Given a query graph $G(N, E)$, a walk from node Q of G, to a relation $R \notin N$, is a path from Q to R, as shown in Figure 10. There maybe many such paths. Formally, let *walks(G (N, E), Q, R)* be the set of all query graphs $G'(N', E')$ conceivable by Clio's inference engine that satisfy the following conditions.

- G' is a path between Q and R.
- If $e \in E'$ is an edge between two nodes in N, then $e \in E$ and the label on this edge is the same in both G and G'.

Within the context of our running example, let $G1$ be a query graph as shown in Figure 11. Then *walks(G1, Children, PhoneDir)* may include the graphs shown in Figure 11 (a). Note that if a potential path would violate the second condition above, Clio will introduce a new copy of a relation to create a valid extension. For instance, $G2'$ in Figure 11 is created in such a way.

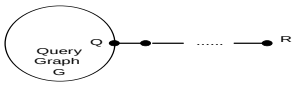

Figure 10: Extensions of a Query Graph

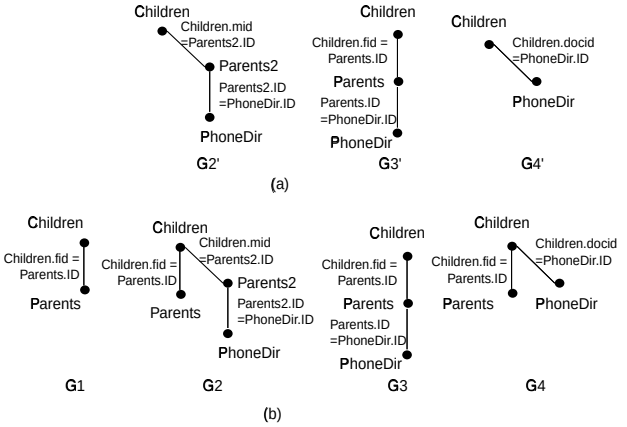

Figure 11: Data Walks

The data walk operator extends a mapping with data walks. Let $M =< G(N, E), V, C_S, C_T >$ be a mapping. Let $Q \in N$ be the *start* relation (which may be chosen by a user or chosen by Clio). Let $R \notin N$ be the *end relation*. The result of the operator $DataWalk(M, Q, R)$ is a set of new mappings, one for each extension $G' \in walks(G, Q, R)$. Each new mapping is defined as $M_e =< G_e, V, C_S, C_T >$ where $G_e = G \cup G'$ (the union of a graph is defined in the normal way as the union of the nodes and union of the edges). Notice that G is an induced, connected subgraph of G_e.

EXAMPLE 5.1. *Suppose we begin with a mapping whose query graph is G1 (Figure 11). Now the user wants to introduce relation PhoneDir into the mapping, since that is where she can find phone numbers. The user may not know how PhoneDir can be incorporated into G1. Hence, she requests a data walk operation, DataWalk(G1, Children, PhoneDir). Clio then produces a set of alternative query graphs, depicted as G2-G4 in Figure 11. Notice that G2 is obtained by merging G1 with G2', shown in the same figure. Similarly, G3 and G4 are obtained by merging G1 with G3' and G4', respectively. Each represents a different way of associating children with phone numbers.*

5.2 Data Chase

The data chase operator also allows the user to extend the query graph of the mapping. However, the chase is designed for cases in which the user may not know which relation(s) she wishes to include in the extended query graph. In a chase, the user selects a source attribute value in the current illustration and asks to be shown how this specific value can be used to extend the mapping. For the chase, Clio identifies

all occurrences of the value within the data source. For each occurrence, an extended mapping is formed and illustrated to the user.

Let $M =< G, V, C_S, C_T >$ be a mapping with illustration I. Let v be a value of attribute $Q[A]$ where Q is one of the relations referenced by a node in G and v is in I. For each relation R that is not referenced by a node in M, where $v \in R[B]$, the mapping M is extended to a new mapping $chase(M) =< G', V, C_S, C_T >$. The new query graph is $G' = (N', E')$ where $N' = N \cup \{R\}$, $e = (Q, R)$ with label $Q[A] = R[B]$ and $E' = E \cup \{e\}$.

A data chase provides the user with a set of alternative scenarios for extending the current mapping with one outer equijoin using a selected value. It is up to the user to decide whether each extension is meaningful in the context of the current mapping. Note that chase is not targeted, that is, the user is not asking for suggestions about how to best extend the mapping to cover a particular relation. Rather, Clio helps the user in experimenting with new data connections. Usually, the data chase operator is used in combination with data walks to combine the user's understanding of the data with that of Clio's.

EXAMPLE 5.2. *Assume that we start with a mapping whose query graph G1 is shown in Figure 12, and we chase value "002" of Children.ID. The chase may produce several options based on where in the database the value of 002 is found. The user reviews these options and selects one. In our example from Section 2, the user chose G4.*

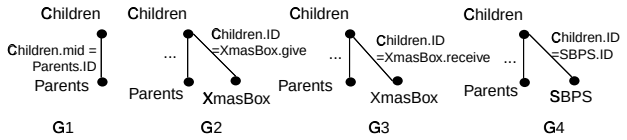

Figure 12: Data Chase

5.3 Continuous Evolution of Illustrations

As a mapping evolves, its illustration must also evolve. We evolve the illustrations in such a way that the user is not required to learn a new set of data in order to understand the evolution. The data in the old illustration, which is familiar to the user, should be retained as much as possible in the new illustration. We refer to this requirement as the *continuity requirement* of illustration evolution.

Intuitively, instead of selecting a completely new set of examples, a continuous evolution extends each example in the current illustration. If the new illustration is not sufficient, the user may request new examples be added to preserve sufficiency. But the role of these new examples in understanding the mapping is made clear and the user does not lose her place when existing examples disappear or mutate beyond recognition. The formal definition of continuous evolution, and a proof that our mapping representation and operators support continuous evolution are included elsewhere [17].

6. USING CLIO FOR LARGE MAPPINGS

In the preceding sections, we detailed our proposal for a mapping creation tool. We supported our description with examples based on a simple source and target schema, with a fairly simple mapping between them. Simple examples make it easy to understand the concepts described. To be useful, however, Clio must be capable of handling real problems of much greater scale and complexity. In this section, we show how Clio helps users with these more realistic scenarios.

There are three types of complexity that we consider. As should be clear by now, during the mapping process we must manage and manipulate multiple (possible) mappings while the user explores the data, creates new correspondences and extends the query graph. We describe the transformation framework that Clio provides to support the mapping process in Section 6.1. The more complex the relationship between source and target, the more (possible) mappings we must handle. In Section 6.2, we illustrate how a complex transformation is created, and show how Clio can reuse portions of mappings to greatly ease the user's task as the number of mappings needed for a particular target mapping becomes more numerous and complex. Large schemas are a second source of complexity. Finally, users often contend with large volumes of data which need to be transformed. If a user is unfamiliar with the data source, the amount of data itself may be an obstacle to understanding how to map it. In a companion paper, we discuss how Clio helps users to deal with both large source and large target schemas [17]. We also discuss how the example mechanism described above helps manage the complexity of large data volumes.

6.1 Clio's Mapping Framework

Clio provides users with a target viewer, a source schema viewer, and a set of workspaces, each associated with a single mapping alternative. At any point in time, one workspace is active. The target viewer always shows the contents of the target as they would be under the mapping associated with the current active workspace (we will refer to this mapping as the *active mapping* to be concise). In other words, the target viewer provides a "What You See Is What You Get" flavor to the mapping process [13].

The schema viewer serves two purposes. It serves as a palette from which users can choose the relations with which they want to work or explicitly select an edge to follow (one way to request a data walk). It also provides a visualization of the query graph being constructed (super-imposed on the schema graph). This visualization depends, like the contents of the target viewer, on the current active mapping.

Each workspace displays the set of examples, E, that illustrates the associated mapping. As the user works with these examples, modifying the query graph, the examples displayed change as described in Section 5 above. The associated mapping also changes, of course, and these changes are reflected both in the query graph visualization, and in the target viewer. When a data walk or data chase results in several alternative mappings, new workspaces are created to represent those alternatives (one of which is chosen as the new active workspace), and the old workspaces are discarded. Alternatively, the old workspaces could be "remembered" to make backing out changes more efficient. When multiple mappings are possible, Clio tries to order them from most likely to least likely, using simple heuristics related to path length, least perturbation to the current active mapping, *etc.* A user can rotate through workspaces or explicitly select a workspace as active in order to try out the effects of different mappings. If the user wishes to eliminate an alternative, she can delete the associated workspace; or she can

	Children(C)				Parents(P)		PhoneDir(Ph)		SBPS(S)		XmasBox(X)	
coverage	ID	name	age	mid	ID	affiliation	ID	number	ID	time	give	receive
C,P,Ph,X(+)	001	Kyle	2	201	201	Safeway	201	201-0001			001	003
C,S(+)	009	Ben	6	401					009	TWF		
C,P,Ph,S,X(+)	002	Maya	4	203	203	Xerox	203	203-0001	002	MW	00 2	004
C,P,Ph,S,X(+)	002	Maya	4	203	203	Xerox	203	203-0002	002	MW	002	004
C,P,Ph,S,X(-)	004	Carmen	10	205	205	MGM	205	205-0001	004	MWF	00 4	006
P,Ph(-)					601	AMCO	601	601-0001				
Ph(-)							701	701-001				
S(-)									00 5	TWF		
X(-)											006	001

Figure 13: Continuous Illustration: Extension

confirm an alternative as the correct mapping (so far), and all alternative workspaces will be deleted.

Clio ensures that the active mapping, the query graph visualization, and the mapping examples in the active workspace are all synchronized: changes to any one of them are automatically reflected in the others. In addition, the data displayed in the target viewer is always that which would be produced by the active mapping.[5] In this way, we empower users to explore the source data and its linkage through both the data and the schema at the same time, seamlessly. Further, and perhaps more importantly, Clio helps the user understand the results of the mappings being formed and allows the user to verify that the transformations that result are what she intended.

6.2 Complex Mappings

Since each mapping produces a subset of the tuples of a single target mapping, many mappings may need to be created to map an entire target schema. Often, these mappings will have a great deal of overlap, differring only in a few correspondences, or a small portion of the query graph. Re-creating the bulk of each mapping from scratch would be tedious, to say the least. Fortunately, the decisions made in creating one mapping can be stored and made available to the user for use in creating additional mappings. This greatly reduces the burden and overhead on the user.

EXAMPLE 6.1. *We return to the user trying to "fill in" Kids.contactPh. The data walk of Section 5 generated the possible query graphs of Figure 11. Each, of course, is the basis for a mapping and an illustration of that mapping. Suppose the user likes the mapping associated with query graph G2, but notices that in cases where mid is null (child has no mother), there is no contactPh in the target. She might accept that mapping, adding the filter "mid not null". However, the result is that motherless children disappear from the target. Realizing that when there is no mother, the father's phone should be used, she also accepts the mapping associated with query graph G3, adding the filter "mid is null".*

In the example, the user's task is made easier because Clio automatically computes both possible mappings, and the user can accept one or several, adding filters as needed. It is worth noting that both mappings inherit all the correspondences and filters of the mapping that existed before the data walk. Such automatic creation of mappings also occurs when the user adds a new correspondence that forces Clio to generate an additional mapping to help complete a target, for example, when the user specifies a second correspondence for the same field in the target. Clio always tries to reuse as much of an existing mapping as possible.

EXAMPLE 6.2. *Suppose that Kids has a column ArrivalTime that tells when the child arrives home. The value for this field comes from the bus schedule table, B, if the child takes a bus, else it is computed from the class schedules table, CS (as a function of the time the child's last class ends). The user might first create a correspondence from B to ArrivalTime, causing Clio to find a join condition (on ID). When the user then creates a correspondence from CS to ArrivalTime, Clio detects that another mapping is needed (because this is a different way to compute ArrivalTime). In creating the new mapping, Clio* copies *the correspondences and filters for already mapped fields (other than ArrivalTime, i.e., ID, name, affiliation, BusSchedule and FamilyIncome), as well as the query graph as it was prior to the addition of the first correspondence for ArrivalTime. Thus the user does not have to re-enter all this information for the new mapping.*

Clio's rich framework supports the user in specifying complex target mappings. Our flexible, powerful mapping representation lends itself to this form of incremental modification. The user can focus on a small portion of a mapping at a time, building each mapping incrementally. Further, when multiple mappings are necessary to create the full target mapping, Clio can transfer correspondences and query graphs from earlier mappings as appropriate, allowing the user to focus on what is new, and alleviating the task of repetitive specification of common transformations.

7. RELATED WORK

***Ad Hoc* Query Tools and QBE Paradigms** *Ad hoc* query tools, including Esperant[6] and Impromptu[7], focus on helping users to access data using natural language or through a point-and-click GUI. These requests are processed by a meta-data layer that translates them into SQL queries. In these systems, the user does not have to know SQL, understand the schema, or know how attributes are decomposed among relations. The translation process is hard-coded using (often procedural) transformation programs. These programs are provided by an (expert) data administrator with complete knowledge of the data. Most of these tools are tightly integrated with a report generating facility so they can readily display the query result. However,

[5] The user can perform some simple transformations on the target as well, such as applying a function to a target attribute. However, to avoid the ambiguity of general target transformations (the well-known view update problem), we restrict most transformations to the source data and resolve ambiguities using source examples.

[6] www.visionyze.com/products/technology/esperant

[7] www.cognos.com/impromptu

the tools do not allow the users to verify or rectify queries by working with the displayed data. Visual query builders, such as QBE Query Builder[8], are also relevant. These tools focus on helping users compose SQL queries faster and with fewer errors. Clio differs from these tools in that we focus on understanding the data source using data, and we allow users to refine their queries using data examples.

Universal Relation Assumption In Clio, we begin with a set of value correspondences that mention attributes and how they should appear in the result of a query. From this, we are trying to deduce a full query, including the join conditions required to connect the relations involved in the query. This statement of our goals appears very similar to the goals of Universal Relation systems [3, 8, 16]. In a Universal Relation interface, the user mentions only attributes, and the system, using reasoning about dependencies, translates this query over the "universal relation" into a query on the actual logical database structure. Indeed, the goal of such systems is to provide logical data independence so the user does not have to understand the table structure of the database. However, the different context of these two problems, schema mapping between heterogeneous structures and querying of a (homogeneous) database, necessitate different solutions. In Universal Relation systems, the goal is to both provide a translation from the universal relation and to characterize when such a translation is well-behaved, that is, when this translation has a well-defined, meaningful semantics. In schema mapping, the goal is to provide possible translations regardless of the qualities (or idiosyncrasies) of the underlying schema. We must be able to handle all schemas and this requirement fundamentally changes the scope of the problem. However, much of the work on universal relations can be used to suggest possible mappings and hence to provide a starting point for our mapping creation algorithms [9]. In this paper, we extended our mapping creation algorithms to present alternatives to the user, something that is not considered in Universal Relation Systems such as System/U [8] or PIQUE [15].

8. CONCLUSIONS

We have presented a new framework that uses examples drawn from source data to illustrate complex schema mappings. We have provided formal definitions of mappings, mapping examples, and mapping operators, and have shown how they can be used to help a user understand the data and develop a mapping. Our approach allows the user to discriminate subtle differences between mappings, while requiring no special data management expertise or even any deep knowledge of the source data. Since Clio understands the complexities of mappings, and enumerates and manages multiple alternatives, the user is able to consider and quickly choose among a broad space of possible mappings to find the one that is best for her application. We believe that our work represents a paradigm shift for schema mapping, from query-centric to data-driven.

9. REFERENCES

[1] G. Bhargava, P. Goel, and B. R. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *ACM SIGMOD Int'l Conf. on the Management of Data*, pp. 304–315, 1995.

[2] A. Deutsch, L. Popa, and V. Tannen. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 459–470, 1999.

[3] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A Simplified Universal Relation Assumption and Its Properties. *ACM Trans. on Database Sys. (TODS)*, 7(3):343–360, Sept. 1982.

[4] C. A. Galindo-Legaria. Outer-joins as Disjunctions. In *ACM SIGMOD Int'l Conf. on the Management of Data*, pp. 348–358, 1994.

[5] C. A. Galindo-Legaria and A. Rosenthal. Outer-join Simplification and Reordering for Query Optimization. *ACM Trans. on Database Sys. (TODS)*, 22(1):43–73, 1997.

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

[7] C.-T. Ho, F. Naumann, X. Tian, L. Haas, and N. Megiddo. Automatic classification of attributes using feature analysis. Submitted, 2001.

[8] H. F. Korth, G. M. Kuper, J. Feigenbaum, A. V. Gelder, and J. D. Ullman. System/U: A Database System Based on the Universal Relation Assumption. *TODS*, 9(3):331–347, 1984.

[9] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 77–88, Cairo, Egypt, Sept. 2000.

[10] R. J. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 30(1), Mar. 2001.

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

[12] S. Ram and V. Ramesh. Schema Integration: Past, Current and Future. In A. Elmagarmid, *et al*, eds, *Management of Heterogeneous & Autonomous Database Systems*, pp. 119–155. Morgan Kaufmann Publishers, 1999.

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

[14] E. A. Rundensteiner, ed. Special issue on data transformations. *IEEE Data Eng. Bull.*, 22(1), 1999.

[15] J. Stein and D. Maier. Relaxing the universal relation scheme assumption. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 25-27, 1985, Portland, Oregon*, pp. 76–84. ACM, 1985.

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

[17] L. Yan, R. J. Miller, L. Haas, and R. Fagin. Data-Driven Schema Mapping. Technical Report CSRG-423, Univ. of Toronto, 2001.

[8] `www.objectplanet.com/sysdeco/QBEQuery`