Concise and Expressive Mappings with +Spicy

Giansalvatore Mecca[1] Paolo Papotti[2] Salvatore Raunich[1] Marcello Buoncristiano[1]

[1] Dipartimento di Matematica e Informatica – Università della Basilicata – Potenza, Italy
[2] Dipartimento di Informatica e Automazione – Università Roma Tre – Roma, Italy

ABSTRACT

We introduce the +Spicy mapping system. The system is based on a number of novel algorithms that contribute to increase the quality and expressiveness of mappings. +Spicy integrates the computation of core solutions in the mapping generation process in a highly efficient way, based on a natural rewriting of the given mappings. This allows for an efficient implementation of core computations using common runtime languages like SQL or XQuery and guarantees very good performances, orders of magnitude better than those of previous algorithms. The rewriting algorithm can be applied both to mappings generated by the system, or to pre-defined mappings provided as part of the input. To do this, the system was enriched with a set of expressive primitives, so that +Spicy is the first mapping system that brings together a sophisticate and expressive mapping generation algorithm with an efficient strategy to compute core solutions.

1. INTRODUCTION

The ability of modern information systems to exchange, transform and integrate data is nowadays considered a crucial requirement. A fundamental requirement for such data integration applications is that of manipulating *mappings* among data sources. Mappings, also called *schema mappings*, are executable transformations – say, SQL queries for relational data or XQuery scripts for XML – that specify how an instance of the source repository should be translated into an instance of the target repository.

Inspired by the seminal papers about the Clio system [8], in the last years a rich body of research has studied algorithms and tools for schema mapping generation. These works have focused on the development of mapping systems that, given a visual specification of the correspondences between the source and target schemas, generate the mappings and then the executable scripts needed to perform the translation. However, despite several years of both system and theory studies, the adoption of mapping systems in real-life integration applications, such as ETL workflows or EII (Enterprise Information Integration), is quite slow. This is due to several factors.

One key factor is the quality of solutions produced by mapping systems. It is well known that a mapping scenario may have many different solutions. These solutions may differ significantly in size, i.e., they may contain a variable amount of redundant tuples. As shown in [5], for large source instances the amount of redundancy in the target may be very large, thus impairing the efficiency of the exchange and the query answering process. A key contribution of data exchange research was the formalization of the notion of *core* [4], which was identified as an "optimal" solution. Informally speaking, the core is irredundant, since it is the smallest among the solutions that preserve the semantics of the exchange, and provides a "good" semantics for answering queries over the target database. Therefore, it can be considered a crucial requirement for a schema mapping system to generate executable scripts that materialize core solutions for a mapping scenario.

Another factor is the expressibility of the mapping system. A benchmark for mapping systems called STBenchmark [1] has been recently proposed to evaluate research mapping systems and commercial ones. None of the systems was able to express all the mappings in the benchmark. It is also known that previous mapping generation algorithms [8] cannot express several natural mappings, like the ones discussed in [2].

The +Spicy system [7] is an attempt at overcoming these limitations. It is the first mapping system that brings together a set of expressive mapping generation primitives and a mapping generation algorithm that generates core solutions. In light of this, we believe +Spicy may contribute towards the goal of integrating schema mapping concepts into practical data integration tasks.

2. OVERVIEW

It is well known that translating data from a given source database may bring to a certain amount of redundancy into the target database. To see this, consider the mapping scenario in Figure 1. A source instance is shown in Figure 2.

In this example, the source database contains tables about books coming from three different data sources, namely the *Internet Book Database (IBD)*, the *Library of Congress* database *(LOC)*, and the *Internet Book List (IBL)*. Based on the correspondences, a constraint-driven mapping system as Clio would generate for this scenario several mappings, under the form of *tuple-generating dependencies (tgds)*, like the

VLDB '09, August 24-28, 2009, Lyon, France

ones below.

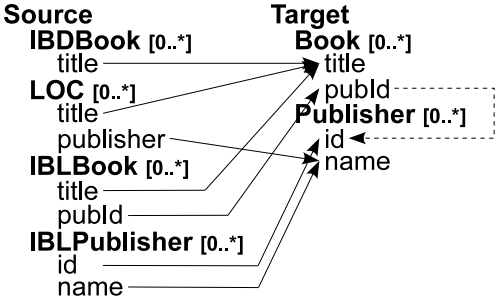

Figure 1: Mapping Bibliographic References

m_1. $\forall t$: $IBDBook(t) \rightarrow \exists N$: $Book(t, N)$
m_2. $\forall t, p$: $LOC(t, p) \rightarrow \exists I$: $Book(t, I) \wedge Publisher(I, p)$
m_3. $\forall t, id$: $IBLBook(t, id) \rightarrow Book(t, id)$
m_4. $\forall id, p$: $IBLPublisher(id, p) \rightarrow Publisher(id, p)$

It can be seen how each source has a slightly different organization wrt the others. In particular, the *IDB* source contains data about book titles only; mapping m_1 copies titles to the *Book* table in the target. The *LOC* source contains book titles and publisher names in a single table; these are copied to the target tables by mapping m_2, which also "invents" a value to correlate the key and the foreign key. Finally, the *IBL* source contains data about books and their publishers in separate tables; these data are copied to the target by mappings m_3, m_4; note that in this case we don't need to invent any values. These expressions materialize the target

IBDBook

title
The Hobbit
The Da Vinci Code
The Lord of the Rings

LOC

title	publisher
The Lord of the Rings	Houghton
The Catcher in the Rye	Lb Books

IBLBook

title	pubId
The Hobbit	245
The Catcher in the Rye	776

IBLPublisher

id	name
245	Ballantine
776	Lb Books

Target: Book

title	pubId
The Hobbit	NULL
The Da Vinci Code	NULL
The Lord of the Rings	NULL
The Lord of the Rings	I1
The Catcher in the Rye	I2
The Hobbit	245
The Catcher in the Rye	776

Target: Publisher

id	name
245	Ballantine
776	Lb Books
I1	Houghton
I2	Lb Books

Figure 2: Instances for the References Scenario

instance in Figure 2, called a *canonical universal instance*. While this instance satisfies the tgds, still it contains many redundant tuples, those with a gray background. Consider for example the tuple t_1 = *(The Hobbit, null)*; it can be seen that the tuple is redundant since the target contains another tuple t_2 = *(The Hobbit, 245)* for the same book, which in addition to the title also gives information about the publisher. The fact that t_1 is redundant with respect to t_2 can be formalized by saying that there is an *homomorphism* from t_1 to t_2. A *homomorphism*, in this context, is a mapping of values that transforms t_1 into t_2. A similar argument holds for the tuple *(The Lord of the Rings, null)*, and for tuples *(The Catcher in the Rye, I2)* and *(I2, LB Books)*, where *I2* is the value invented by executing tgd m_2. The presence of such homomorphisms means that the solution in Figure 2 has an *endomorphism*, i.e., a homomorphism into a sub-instance – the one obtained by removing all redundant tuples.

The fact that tgds produced by a schema mapping algorithm may generate redundancy in the target is well known and has motivated several practical proposals (e.g. [5]) towards the goal of removing such redundant data. Unfortunately, these proposals are applicable only in some cases and do not represent a general solution to the problem. In [4] the notion of *core* solutions has been introduced as a "more desirable" solution than the one in Figure 2. The *core* is the smallest among the solutions for a given source instance that has homomorphisms into all other solutions. The core of the solution in Figure 2 is in fact the portion of the target tables with a white background.

A possible approach to the generation of the core for a relational data exchange problem is to generate the canonical solution, and then to apply a post-processing algorithm for core identification. Several polynomial algorithms have been identified to this end [4, 6]. These algorithms provide a very general solution to the problem of computing core solutions for a data exchange setting. Also, an implementation of the core-computation algorithm in [6] has been developed [9], thus making a significant step towards the goal of integrating core computations in schema mapping systems. However, experience with these algorithms shows that, although polynomial, they require very high computing times since they look for all possible endomorphisms among tuples in the canonical solution. As a consequence, they hardly scale to large mapping scenarios. Our goal is to introduce a core computation algorithm that lends itself to a more efficient implementation as an executable script and that scales well to large databases. To this end, in the following sections we introduce two key ideas: the notion of *homomorphism among formulas* and the use of *negation* to rewrite tgds.

Formula Homomorphisms and Rewriting. The first intuition is that it is possible to analyze the set of formulas in order to recognize when two tgds may generate redundant tuples in the target. This happens when it is possible to find a homomorphism between the right-hand sides of the two tgds. Consider tgds m_1 and m_3 above; it can be seen that the conclusion $Book(t, N)$ of m_1 can be mapped into the conclusion $Book(t, id)$ of m_3 by the following mapping of variables: $t \rightarrow t$, $N \rightarrow id$; in this case, we say that m_3 *subsumes* m_1. This gives us a nice necessary condition to intercept possible redundancy (i.e., possible endomorphisms among tuples in the canonical solution). Note that the condition is merely a necessary one, since the actual generation of endomorphisms among facts depends on values coming from the source. Note also that we are checking for the presence of homomorphisms among formulas, i.e., conclusions of tgds, and not among instance tuples; since the number of tgds is typically much smaller than the size of an instance, this task can be carried out very quickly.

Based on these ideas, in our example we find all possible homomorphisms among tgd conclusions; more specifically, we look for variable mappings that transform atoms in the conclusion of one tgd into atoms belonging to the conclusions of other tgds, with the constraint that universal variables are mapped to universal variables. There are three homomorphisms of this form: (*i*) from the right hand side of m_1 to the rhs of m_3, as discussed above; (*ii*) from the rhs of m_1 to the rhs of m_2 by the following mapping: $t \rightarrow t$,

$N \rightarrow I$, i.e., also m_2 subsumes m_1; (iii) from the rhs of m_2 to the union of the conclusions of m_3, m_4, by the following mapping: $t \rightarrow t$, $I \rightarrow id$, $p \rightarrow p$; in this case we say that m_3, m_4 *cover* m_2.

A second important intuition is that, whenever we identify two tgds m, m' such that m subsumes m', we may prevent the generation of redundant tuples in the target instance by executing them according to the following strategy: (a) generate target tuples for m, the "more informative" mapping; (b) for m', generate only those tuples that actually add some new content to the target. In our example, we rewrite the original tgds as follows (universally quantified variables are omitted):

$$m_3.\ IBLBook(t, id) \rightarrow Book(t, id)$$
$$m_4.\ IBLPublisher(id, p) \rightarrow Publisher(id, p)$$
$$m'_2.\ LOC(t, p) \wedge \neg(IBLBook(t, id) \wedge IBLPublisher(id, p)) \rightarrow \exists I\colon Book(t, I) \wedge Publisher(I, p)$$
$$m'_1.\ IBDBook(t) \wedge \neg(IBLBook(t, id)) \wedge \neg(LOC(t, p)) \rightarrow \exists N\colon Book(t, N)$$

Once we have rewritten the original tgds in this form, we can easily generate an executable transformation under the form of relational algebra expressions. Here, negations become difference operators. The algebraic expressions can be easily implemented in an executable script, say in SQL or XQuery, to be run in any database engine. As a consequence, there is a noticeable gain in efficiency with respect to the algorithms for core computation proposed in [4, 6, 9].[1]

Expressive Power. It can be seen that the rewriting algorithm can be applied to any set of tgds, not necessarily generated by the mapping system. To do this, one of our goals was to extend the expressive power of the mapping system with respect to previous ones.

Suppose we are given the following set of pre-defined tgds that refer to a variant of the self-join example in STBenchmark [1]. The target schema contains a single relation *Gene* with attributes *name*, *type* and *protein*, which holds together primary genes and secondary genes, called "synonyms". A primary gene and its synonyms share the same protein. In the source, we have genes organized in separate tables *PrimaryGene* and *Synonym*, connected through a key-foreign key constraint. In addition, we have a *Protein* table, from which we want to copy only tuples about genes coming from the EMBL database. A key feature of this example is the self-join of table *Gene* in the target on the *protein* attribute.

$$m_1.\ Protein(p, g, \text{'EBML'}) \rightarrow Gene(g, p, \text{'primary'})$$
$$m_2.\ PrimaryGene(i, n, p) \rightarrow Gene(n, p, \text{'primary'})$$
$$m_3.\ Synonym(n, i) \wedge PrimaryGene(i, n', p) \rightarrow Gene(n, p, \text{'synonym'}), Gene(n', p, \text{'primary'})$$

Our goal is to generate a mapping scenario for these tgds, and then rewrite them in order to generate core solutions. In this case, +Spicy proposes to the user the scenario in Figure 3. To handle arbitrary tgds of this form, we had to enrich the set of primitives that can be used to specify a mapping scenario. We extend these inputs in several ways: (i) we introduce the possibility of duplicating sets in the source and in the target; to handle tgd m_3 above, we duplicate the *Gene* table in the target; each duplication of a set

[1]We have recently learned that a similar approach has been independently undertaken in [10].

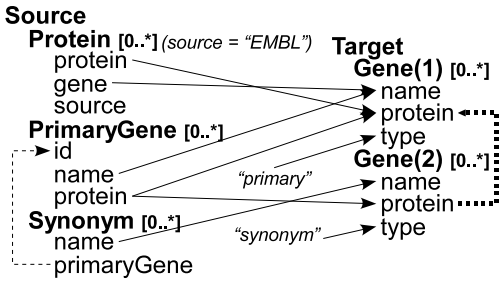

Figure 3: Inverse of Self Joins

R corresponds to adding to the data source a new set named R^k, for some k, that is an exact *copy* of R; (ii) we give users full control over joins in the two data sources, in addition to those corresponding to foreign key constraints; using this feature, users can specify arbitrary join paths, like the self-join on the *protein* attribute in m_3; (iii) finally, we allow users to express selection conditions on sets, like *source = 'EMBL'* on the protein table in m_1.

This richer set of primitives poses some challenges with respect to the rewriting algorithm. In fact, duplications in the target correspond to different ways of contributing tuples to the same set. This makes the search for homomorphisms more delicate, since there exist tgds that write more than one tuple at a time in the same target table, and therefore redundancy can be generated not only across different tgds, but also by firing a single tgd. Our solution to this problem is to adopt a two-step process. First, we rewrite tgds that populate the target with duplications. Then, we construct a second exchange, in order to merge the content of all duplications. We apply the rewriting to this exchange as well in order to remove redundant tuples. The process is sketched in Figure 4. Complex scenarios with self-joins will

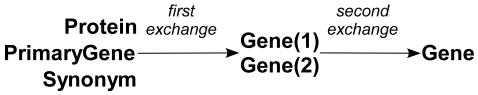

Figure 4: The Double Exchange

be discussed during the demonstration.

3. SYSTEM DESCRIPTION

The +Spicy system is an evolution of the original Spicy system [3]. It has been developed in Java using the NetBeans Platform as a basis for the graphical user interface. A snapshot is shown in Figure 5. The system architecture is shown in Figure 6. The system supports various usage scenarios,

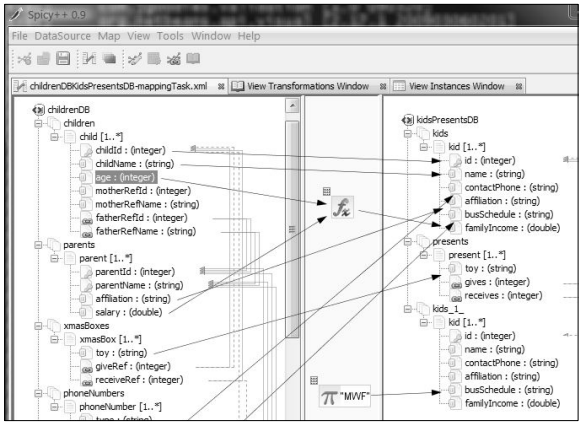

Figure 5: A snapshot of the system

that will be shown during the demonstration. The typical one is that in which a user provides to the system a mapping specification using the GUI; in doing this, besides specifying the source and target schema, users can rely on the primitives offered by the system, namely: (*i*) a rich set of correspondences that include traditional 1:1 correspondences but also n:1 value correspondences with complex transformation functions, constant correspondences, and correspondences with filters; (*ii*) the possibility of duplicating sets in the two schemas; (*iii*) the possibility to define arbitrary join-conditions in the sources; (*iv*) the possibility of specifying selection conditions on sets in the source. The mapping specification is handled by the mapping generation module, which generates the tgds. As an alternative, a simple parser is available to load a set of pre-defined tgds. The parser will generate a scenario from the tgds, and show it to the user so that s/he can visually inspect and possibly modify it. At this point, the user has a set of tgds, either generated

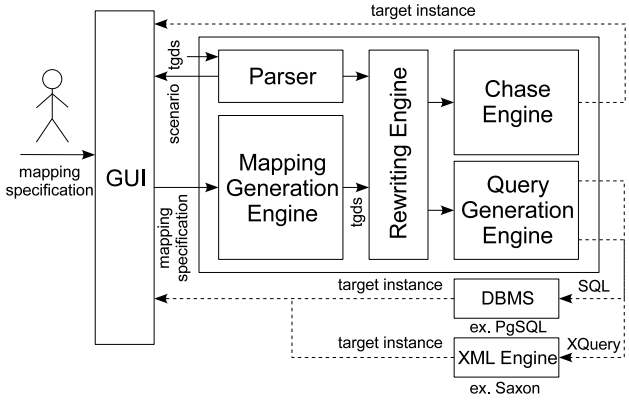

Figure 6: Architecture of Spicy

internally or pre-defined and loaded by the parser. Before moving to the actual query generation phase, the tgds are rewritten by the rewriting engine in order to ensure that core solutions are generated.

Based on these rewritten tgds, an executable query either in SQL or in XQuery can be generated. The system integrates interfaces to various popular SQL and XQuery engines (like PostgreSQL and Saxon), so that the final query can be executed against one or more source instances and results can be inspected using the GUI. To simplify the debugging of the mapping scenario and to reduce dependencies wrt external systems, +Spicy also incorporates an internal chase engine to execute the tgds and generate solutions internally. In our experience, this is more immediate than sending a query to an external engine, and greatly helps users during their work sessions.

The demonstration will be centered around the discussion of various mapping scenarios, with the goal of showing the expressiveness and the quality of solutions produced by the system. We will demonstrate practical scenarios, scenarios from the literature, and synthetic scenarios of large size.

In terms of expressiveness, we will show how +Spicy can handle all of the mapping scenarios proposed in [1]. We will also discuss how the system handles scenarios of the kind discussed in [2] by allowing users to explicitly manipulate join conditions. All scenarios will be run both using the internal engine and by generating SQL or XQuery scripts for external engines, to show how the internal engine guarantees more immediacy but external engines are needed in order to scale to large instances.

With respect to the quality of solutions, we will discuss how tgds are rewritten in order to generate the core. To better evaluate the quality of the core solutions generated by the tgds after the rewriting, we will compare them to the canonical universal instance generated by the original tgds.

Note that all algorithms discussed in the previous sections are applicable to both flat and nested data. However, data exchange research has so far concentrated on relational data and there is still no formal definition of a data exchange setting for nested data. Still, we compare the solutions produced by the system for nested scenarios with the ones generated by the basic [8] and the nested [5] mapping generation algorithms, that we have reimplemented in our prototype. We show that the rewriting algorithm invariably produces smaller solutions, without losing informative content.

One final crucial issue is related to performance. In fact, computing cores may be a challenging task. The polynomial-time algorithm defined in [6] and implemented in [9] usually requires several hours, even for instances of a few thousand tuples. On the contrary, our scripts scale well, as shown in [7]. To show this, we have prepared source databases of varying sizes for the selected scenarios, from 100K to 1M tuples. We will show how in practical cases the computation of the core solution is very efficient and scales well to such large databases. Finally, synthetic scenarios with a large number of tables and tgds will be used to show that the rewriting algorithm performs well when the size of the scenario increases.

4. REFERENCES

[1] B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. *Proc. of the VLDB Endowment*, 1(2):1468–1471, 2008.

[2] Y. An, A. Borgida, R. Miller, and J. Mylopoulos. A Semantic Approach to Discovering Schema Mapping Expressions. In *Proc. of ICDE*, pages 206–215, 2007.

[3] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema Mapping Verification: The Spicy Way. In *Proc. of EDBT*, pages 85 – 96, 2008.

[4] R. Fagin, P. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *ACM TODS*, 30(1):174–210, 2005.

[5] A. Fuxman, M. A. Hernández, C. T. Howard, R. J. Miller, P. Papotti, and L. Popa. Nested Mappings: Schema Mapping Reloaded. In *Proc. of VLDB*, pages 67–78, 2006.

[6] G. Gottlob and A. Nash. Efficient Core Computation in Data Exchange. *J. of the ACM*, 55(2):1–49, 2008.

[7] G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings. In *Proc. of ACM SIGMOD*, 2009.

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

[9] V. Savenkov and R. Pichler. Towards practical feasibility of core computation in data exchange. In *Proc. of LPAR*, pages 62–78, 2008.

[10] B. ten Cate, L. Chiticariu, P. Kolaitis, and W. C. Tan. Laconic Schema Mappings: Computing Core Universal Solutions by Means of SQL Queries. In *Proc. of VLDB*, 2009.