Sample-Driven Schema Mapping*

Li Qian
Univ. of Michigan, Ann Arbor
eql@umich.edu

Michael J. Cafarella
Univ. of Michigan, Ann Arbor
michjc@umich.edu

H. V. Jagadish
Univ. of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

End-users increasingly find the need to perform light-weight, customized schema mapping. State-of-the-art tools provide powerful functions to generate schema mappings, but they usually require an in-depth understanding of the semantics of multiple schemas and their correspondences, and are thus not suitable for users who are technically unsophisticated or when a large number of mappings must be performed.

We propose a system for **sample-driven** schema mapping. It automatically constructs schema mappings, in real time, from user-input sample target instances. Because the user does not have to provide any explicit attribute-level match information, she is isolated from the possibly complex structure and semantics of both the source schemas and the mappings. In addition, the user never has to master any operations specific to schema mappings: she simply types data values into a spreadsheet-style interface. As a result, the user can construct mappings with a much lower cognitive burden.

In this paper we present MWEAVER, a prototype sample-driven schema mapping system. It employs novel algorithms that enable the system to obtain desired mapping results while meeting interactive response performance requirements. We show the results of a user study that compares MWEAVER with two state-of-the-art mapping tools across several mapping tasks, both real and synthetic. These suggest that the MWEAVER system enables users to perform practical mapping tasks in about *1/5th* the time needed by the state-of-the-art tools.

Categories and Subject Descriptors

H.2.5 [**Database Management**]: Heterogeneous Databases; D.2.12 [**Software Engineering**]: Interoperability—*Data mapping*

Keywords

Schema Mapping, Sample-Driven, Data Integration, Usability

1. INTRODUCTION

A schema mapping transforms a source database instance into an instance that obeys a target schema. It has long been one of the most important, yet difficult, problems in the areas of data exchange and data integration [9, 21, 22]. Traditional database applications in E-business, data warehousing and semantic query processing have required good schema mappings among heterogeneous schemas. Moreover, as the amount of structured Web-based information explodes (*e.g.*, Wikipedia, Freebase, Google BigTable, etc.), users are directly exposed to the task of combining, structuring and repurposing information [12]. Doing so inevitably requires schema mapping to be democratic: non-technical users should be able to cook their data with their own flavor, even if they cannot master the "professional kitchenware" designed for database experts.

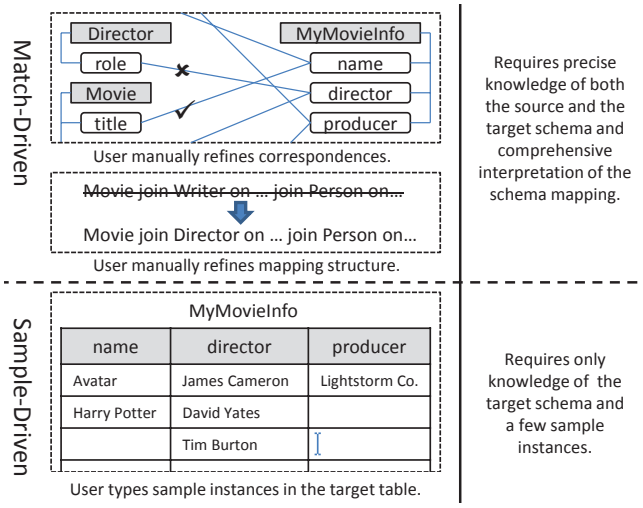


Figure 1: A Comparison between the Match-Driven Approach and the Sample-Driven Approach

Due to the importance of the schema mapping problem, a handful of mapping design systems have been developed. These systems include InfoSphere Data Architect (from Clio [27]), BizTalk Mapper [2], Altova MapForce [1], and Stylus Studio [3]. All of these systems are based on the same general methodology that was first proposed in Clio [27]. The methodology consists of two phases. In the first *matching* phase, a set of correspondences between source and target schema elements is solicited from the user, with possible aid from automated techniques that find similar attribute pairs [25, 23, 20, 14, 24, 13, 16, 26, 15]. During the second *mapping* phase, the set of matches yields an executable transformation from the source schema to the target schema.

Unfortunately, this traditional *match-driven* model is unsuitable for many modern schema mapping tasks. The user must either build attribute-level matches from scratch, or else painstakingly double-check an automatically-generated set of matches. An implicit assumption made by these systems is that the user has detailed knowl-

*This work was supported in part by NSF grant IIS 1017296.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

edge of both the source and target schemas. For traditional schema mapping tasks that involve a sophisticated administrator and a single high-value target database, this assumption makes sense. But modern mapping scenarios feature relatively unsophisticated users and a multiplicity of tasks: a DBA may only map two HR databases together when a corporate acquisition takes place, whereas a Web advertising analyst may need to combine schemas of different datasets multiple times a day. For these less-technical users who perform a large number of mappings, the laborious *match-driven* process can be a heavy burden.

A Sample-Driven Approach In this paper, we propose a *sample-driven* approach that enables relatively unsophisticated end-users, not DBAs, to easily construct their own data. The key idea behind our approach is to allow the user to implicitly specify schema mappings by providing sample data of the target database. Behind the scenes, the system automatically elicits the mappings that transform the source database into this partially-described target. After the user has provided enough information, the system can determine a single best mapping. The process is iterative. As the user types more information from the target database, the system provides increasingly better estimates of the correct mapping. Figure 1 depicts a high-level comparison between the traditional match-driven approach and the sample-driven approach.

Our *sample-driven* approach reduces the user's cognitive burden in two ways. First, the user no longer needs to explicitly understand the source database schema or the mapping. She simply types in sample instances until the system converges to a single proposal. Second, the operations that the user performs are common and require no special training: she simply types in data as in a spreadsheet. In contrast, current tools are applications designed for trained DBAs, and require users to decide whether individual attribute-level matches are correct. The following example provides some intuition about how the *sample-driven* approach works.

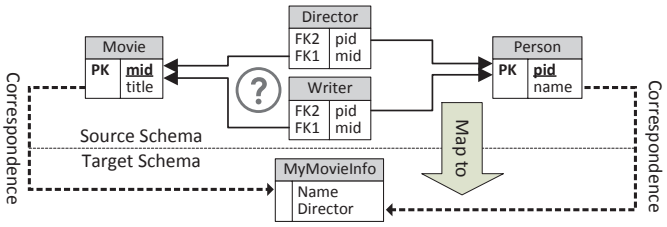


Figure 2: An Example Schema Mapping with The Question Mark Indicating a Join Path Ambiguity.

EXAMPLE 1. *A user is exploring the Yahoo Movies database, and wishes to store the movie title as* **Name** *and the director name as* **Director** *in a target* **MyMovieInfo**, *as shown*[1] *in Figure 2.*

A typical match-driven system proposes attribute correspondences to the users, as shown in Figure 3. The user needs to either pick out each correct correspondence from multiple candidates, or scan the source schema for the correct correspondence if it is not proposed; both situations require a comprehensive knowledge of the schemas.

In a sample-driven approach, the user freely provides sample instances for the target **Director** *field. For each director name she provides, the system automatically searches the source database to find all attributes that contain the name. For example, if a user enters* `Ed Wood`, *the system may find the value in both* **Person.name** *and* **Movie.title**. *As the user enters more names, the set of attributes eventually converges to a single attribute* **Person.name**, *indicating that the user has implicitly specified the correspondence from* **MyMovieInfo.Director** *to* **Person.name**.

[1]We only show a subset of the source schema due to space. The solid arrows represent foreign key constraints.

Even if a match-driven system perfectly generates all the attribute-level correspondences, it may has to deal with multiple possible mappings. In this example, imagine the system matches **MyMovieInfo.Name** *to* **Movie.title**, *and* **MyMovieInfo.Director** *to* **Person.name**. *There are still two possible ways to construct the mapping*[2]*: one by joining* **Movie** *and* **Person** *via* **Director**, *the other by joining via* **Writer**. *(See Figure 2.) Current match-driven systems usually pick only one mapping, which may not be the desired one [7]. Even if the system presents both candidate mappings to the user, she still has to manually select the desired join via* **Director**.

In contrast, the sample-driven approach also considers data-level information to help find the correct mapping. For example, if the user enters (`Harry Potter`, `David Yates`*) in the target, the system will know that the join must NOT be via* **Writer**, *as the source indicates that the writer of* `Harry Potter` *is* `J. K. Rowling`.

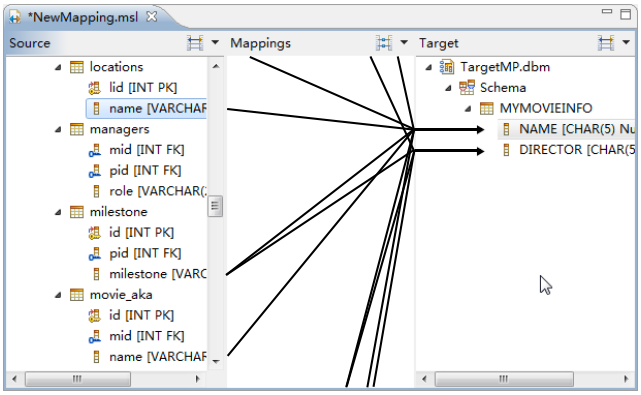


Figure 3: A Screenshot of IBM InfoSphere Data Architect

Of course, the user must be familiar with the target schema in order to provide samples. While traditional database schemas can be quite complex, expert DBAs are likely to know the data well enough to give useful samples. Non-traditional users may not be familiar with database schemas in general. But as the trend for lightweight, Web-based information integration increases, our computationally unsophisticated users are likely to be well-informed about the target database they want to build. In either case, it is reasonable to believe that the sample-driven approach will be suitable for a large group of mapping scenarios.

In this paper, we design and prototype a sample-driven mapping system, MWEAVER, which facilitates schema mapping tasks for end-users. We also conduct a detailed user study that compares users' behavior with MWEAVER and with state-of-the-art mapping tools. We show that by reducing the cognitive burden for the user, and providing a familiar spreadsheet-style interface, MWEAVER allows the average user to perform a practical mapping task in about *1/5th* the time needed for the traditional match-driven approach.

Technical Challenges MWEAVER renders schema mapping easier from the user's perspective, but actually building the runtime system presents two substantial technical challenges. First, it must obtain the desired mapping using just the user-provided samples. Doing so can entail locating each piece of sample throughout the source database, and then deriving all possible mappings that those pieces together suggest. Second, these mappings must be computed quickly enough that the user can obtain "interactive-speed" feedback, allowing her to review the current system status before continuing to provide more samples or stopping if the system has generated the desired mapping.

[2]We only consider joins via foreign key constraints.

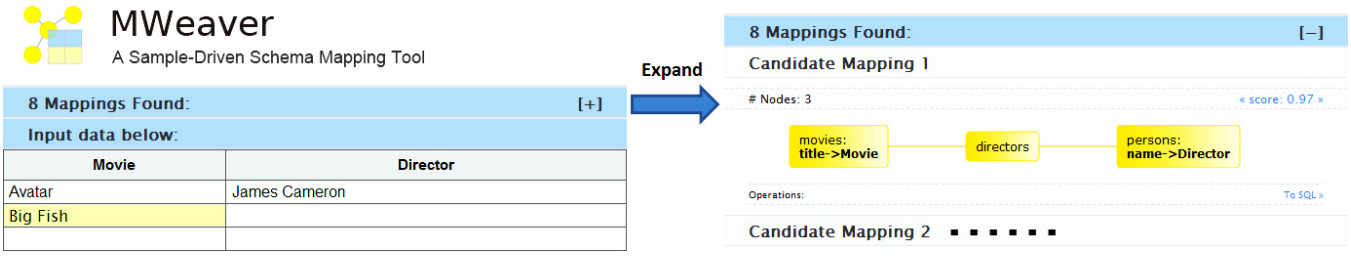


Figure 4: A Screenshot of MWEAVER. Left: The Input Spreadsheet. Right: The Expanded List of Candidate Mappings.

Our Contributions Our work makes the following contributions:

- We propose a sample-driven approach to facilitate schema mapping tasks for end-users, and present a prototype system, MWEAVER.
- We develop an efficient sample search algorithm and show that it can obtain provably correct results at interactive speeds.
- We present the results of a detailed user study that demonstrates, among other things, that a typical MWEAVER user can obtain schema mappings in 1/5th of the time required by state-of-the-art mapping tools.

Our paper is organized as follows. We cover related work in Section 2 and provide an overview of MWEAVER in Section 3. The sample search algorithm is described in Section 4. Section 5 describes how we iterative prune the candidate mappings. In Section 6, we present user studies that demonstrate the usability of our system, as well as performance experiments that demonstrate the efficiency of our algorithm. Finally, we conclude in Section 7.

2. RELATED WORK

Research into schema matching and mapping make up an enormous body of work, as described in a recent text [10]. State-of-the-art schema matching approaches can be roughly classified into three categories. *Schema-based* techniques perform matching by examining metadata, such as in Clio [27] and Similarity Flooding [25]. *Instance-based* approaches determine the similarity between schema elements from the similarity of the characteristics of their instances [23, 20]. Many systems utilize a combination of these two techniques, such as LSD [14], Cupid [24], COMA [13]. *Usage-based* methods improve matching quality by exploiting usage information, such as query logs [16] and search clicklogs [26].

Although we are not aware of systems that use contributed sample instances to directly construct the mapping, data examples have been an important part of the mapping literature. Alexe, et al. recently developed Eirene, a system for interactive design and refinement of schema mappings using data examples, by GLAV fitting generation [8]. Eirene offers abundant flexibility in that it derives the mappings as long as the source and the target schema, as well as a few paired examples under both schemas are provided. However, consequently, the user has to understand both schemas in order to fill in valid data examples and explicitly specify join paths by linking related tables using data with the same value. This may result in some user burden, especially in the presence of a complex source schema and long join paths in the mapping. In contrast, MWEAVER assumes the existence of a complete source database instance, to which the user-input samples belong. As a result, the user does not need to know the source schema or to specify the join paths, because the system can use the source instance as a knowledge base to automatically derive the mappings.

Yan, et al. attempted to choose tuples that best exemplify a mapping [30]. Alexe, et al. systematically investigated the capabilities and limitations of data examples in explaining and understanding schema mappings, especially in using universal examples to characterize mappings defined by s-t tgds [7]. However, these are done in an "explanatory" phase after the mapping has been generated.

SPIDER [6] and MUSE [5] are designed to refine a partially-correct mapping generated by a more traditional tool. They first generate candidate mappings using a match-driven mapping tool, and then ask the user to debug them by examining user-proposed examples. In contrast, MWEAVER asks users to simply enter data items, and to trust that the system will find the correct mapping.

Drumm et al. designed QuickMig for automatic schema matching for data migration [15]. It asks a user to manually create target instances in the source and then apples standard instance-based matching algorithms on these sample instances to determine the matching. However, these sample instances only aim to generate schema element correspondences rather than mapping structure.

Recently, there has been a trend toward leveraging user feedback to improve the quality of an information integration task. Talukdar et al. [29, 28] developed system Q to assist the user in creating integration queries. In system Q, integration is defined as a union of queries weighted by relevance. The system shows the query result to the user, who in turn provides feedback to the system by judging whether a result tuple is relevant. In MWEAVER, the system notifies the user about the current mapping generation status, and the user provides feedback in the form of additional sample instances.

The system [31] is a well-known work that employs example data to assist in query generation. The user constructs a query with QBE by providing example tuples under both the database schema and the result view. Examples with same value suggest how the relations are joined and which attributes are projected. While the user can supply fake data in QBE, the input to MWEAVER must be samples from the database instance. As a result, the user has to manually specify join paths by simulated IDs in QBE, while MWEAVER is able to automatically derive the join paths from sample values.

MWEAVER has a strong relationship to database keyword search techniques, which have been extensively studied in the literature [11, 4, 17]. However, database keyword search focuses on querying tuples that may be related to the keywords; in contrast, MWEAVER focuses on determining the exact mapping that produces a target database containing the samples.

3. SYSTEM OVERVIEW

In this paper, we propose MWEAVER, a sample-driven schema mapping tool that constructs schema mappings based on user-input sample instances. By assuming the user-input samples are approximately present[3] in a source database instance that we have access to, the major advantage of MWEAVER is that it isolates the user from the possible complexity of such a source database and its schema. MWEAVER takes as input the source instance and a partially filled spreadsheet, and produces as output the schema mappings that map the source to a target containing that spreadsheet. We assume the schema mappings are Project-Join queries over relational database. While selection, aggregation and user-defined functions would largely strengthen the expressive power of the mappings, we do not study them in this work because they may produce information loss that is non-recoverable on the target side.

[3]We will detail this notion of "approximation" in Section 4.1

Since we do not expect end-users to be able to specify foreign key constraints[18], we assume in this paper that the target schema comprises one or more table "views", each of which has joined all the information the user wants to see at one time. Since these views are independent, they can be constructed one at a time. Without loss of generality, we can assume the target schema is a single table.

User Interface: The primary UI component of MWEAVER is a spreadsheet that conforms to the target schema. We call it the ***Input Spreadsheet***. On the left of Fig 4 shows an input spreadsheet in which the user is filling data. The user may adjust the input spreadsheet by adding/dropping/renaming columns to meet her mental model of the target. The bar directly under the logo provides information about the current mapping generation status. By default it only displays the number of mappings currently found. If the user wishes to know more about the mappings, she may expand the information bar by clicking the "plus" on the right. This will trigger a ***Mapping List***, which visualizes each mapping with details.

In the mapping list, each mapping is visualized as an undirected tree. Each node in the tree is labeled with the source relation involved in the mapping, together with the correspondences between the target columns and the source attributes. Each edge in the tree represents how these source relations are joined in the mapping.

User Input: After the structure of the input spreadsheet is fixed, the user can input data in any cell in the spreadsheet. Formally, a user input is $Input(i, j, c)$, which updates the content of the cell on the i-th row and the j-th column of the input spreadsheet to c. We call the content in each non-empty cell of the input spreadsheet a ***sample***. We do not consider empty cells.

Interaction Model: The system interacts with the user by maintaining a set of mappings upon each user input. We call such mappings ***candidate mappings***. The interaction starts with the user filling out the first row of the input spreadsheet. We require the first row to be fully populated in order to establish a general impression of the complete desired mapping. After this, MWEAVER constructs the initial set of candidate mappings from the samples in the first row. Formally, we name this process ***Sample Search***.

Afterwards, the user may continue to provide sample instances in any cell below the first row. Whenever one cell is updated, MWEAVER uses all the samples (i.e., non-empty cells) from that row to prune the set of candidate mappings. We call this process ***Sample Pruning***. Finally, the interaction terminates when there is only one mapping left. As long as the user input correctly reflects her knowledge and her knowledge is consistent with the source database, the remaining mapping must be the desired mapping. In other words, as the number of candidate mappings decreases, the average mapping quality increases w.r.t. the number of user-input sample. This finally produces a single best mapping which meets the user requirement. We will elaborate on the sample search and pruning techniques in Section 4 and Section 5, respectively.

4. SAMPLE SEARCH

4.1 Problem Formalization

We consider a ***source database*** D_S with ***schema*** S_S that has n ***relations*** $R_1, ..., R_n$ and a target database with ***target schema*** S_T comprising a single ***target relation*** R. A ***schema mapping*** M is a project-join query that maps S_S to S_T. For each R_i, $i \in [n]$[4], we denote its schema by $S(R_i)$ and its instance by $I(R_i)$. $S(R_i)$ is the set of all the attributes in R_i. Similarly, R has a schema $S(R) = \{A_1, ..., A_m\}$, where m is the ***size*** of the target and A_j ($j \in [m]$) represents the j-th attribute in R. $t[A]$ stands for the projection of tuple t on attribute A.

[4]Throughout the paper, we denote $\{1..n\}$ by $[n]$.

The user types in samples in the input spreadsheet under the target schema. Each sample E is a string. We denote the first row of samples by $t_E = (E_1, ..., E_m)$, and call it a ***sample tuple***. Our goal of sample search is to find all the schema mappings that transform the source database to a target "containing" the sample tuple.

Because the user-input may not have an exact match in the source, as we use them to generate the schema mappings, we forgive inaccurate samples by allowing them to be "noisily contained" by some database instance. Formally, we define this "noisily contain" relationship by a binary operator $\succeq$, which returns a boolean value based on the desired error model. Having this operator, we say $t[A]$ contains sample E iff $t[A] \succeq E$. Similarly, we say t contains E iff $\exists A$ s.t. $t[A] \succeq E$. Furthermore, given $t_E = (E_1, ..., E_m)$, we say t contains t_E, iff $\forall i \in [m], t[A_i] \succeq E_i$. Finally, we say a target database D_T contains t_E iff $\exists t \in D_T$ s.t. t contains t_E. Having this concept of containment, we define sample search as follows.

DEFINITION 1 (SAMPLE SEARCH). *Given a source database D_S and a sample tuple $t_E = (E_1, ..., E_m)$, sample search finds all* schema mappings[5] *M such that $M(D_S)$ contains t_E. Each such mapping is called a* valid schema mapping.

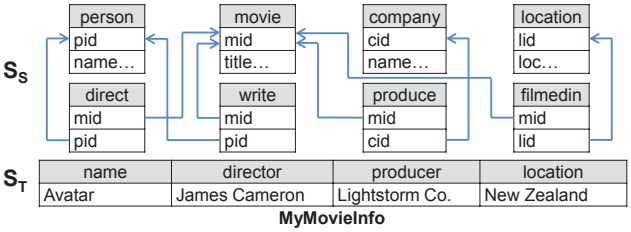


Figure 5: The Source Schema and the Target Relation with Samples

EXAMPLE 2 (THE RUNNING EXAMPLE). *Let D_S be part of the Yahoo Movie Database with its schema S_S partially shown in Figure 5. Let S_T has $R =$ **MyMovieInfo** with $S(R) =$ {**name**, **director**, **producer**, **location**}, where* **name** *indicates the movie title,* **director** *represents the name of the movie director,* **producer** *identifies the company which produces the movie and* **location** *specifies the filming location. Suppose the user enters a sample tuple* (`Avatar`, `James Cameron`, `Lightstorm Co.`, `New Zealand`). *The sample search aims to find all the schema mappings that produce a target database that contains the sample tuple.*

We will use this as a running example throughout this section.

4.2 The Challenge and The Opportunity

Intuitively, one way to solve the sample search problem is to model the whole source database as a graph with each tuple represented by a vertex and each foreign key reference by an edge. Then the problem is equivalent to finding all the subgraphs such that each user-input sample is contained by a vertex of the subgraph. However, the fan-out of vertices in such a graph can be very large (e.g., a director may have directed dozens of movies, and one movie genre may even contain thousands of movies). As a result, searching such a graph may be very inefficient [11].

An alternative approach is to first generate a group of mappings that will *possibly* yield a target database that contains the sample tuple, and then execute each of the mappings on the source database to see if it is actually a valid mapping. Such "lucky" mappings could be constructed by joining sample-containing relations in the source database via various foreign key relationships. Unfortunately, the number of such "lucky" mappings grows exponentially with respect to both the number of joins allowed and the size of the target. Because a "lucky" mapping has to be executed before one

[5]We restrict the search space with certain constraints that we will detail in Section 4.4

knows whether its result contains the sample tuple, this approach may require exponential rounds of database accesses. This can be verified in similar database keyword search scenarios [17, 4].

In fact, the sample search problem is NP-hard, because it essentially deals with the problem of searching a graph for all sub-graphs that satisfy certain properties [19]. However, the definition of sample search implies that, if a mapping is invalid, then any mapping that structurally contains it must be invalid. This inspires us to construct valid mappings from smaller ones to larger ones. Since generating smaller valid mappings is relatively cheap, as long as we can efficiently build larger valid mappings on top of the smaller ones, we can potentially meet practical interactive requirement.

4.3 Our Solution

In this paper, we propose a **tuple path weaving** algorithm TPW to solve the sample search problem. We first create schema mappings for each pair of samples (*pairwise mappings*), and then check their validity within the limit of acceptable noise. The execution returns a set of *pairwise tuple paths*, which are essentially instance-level support for the mappings. A mapping is valid iff such supporting set is non-empty. After that, we "weave" these pairwise tuple paths purely in memory to generate larger and larger paths, which finally cover all the samples. Lastly, we extract the valid mappings from these "complete" tuple paths and return them with ranking.

Informally, TPW functions in the following five major steps.

1. **Find sample occurrences** in the source database.
2. For each pair of sample occurrences, **generate pairwise mapping paths** by searching their possible connections in the source schema.
3. For each pairwise mapping, **create a set of pairwise tuple paths**. Do so by translating the mapping into an approximate search query[6], then executing it in the source database. This will produce all pairwise tuple paths that support the mapping. Mappings with no support will be pruned.
4. From these pairwise paths, build all **complete tuple paths** in a bottom-up manner.
5. **Rank** mappings extracted from the complete tuple paths.

By pruning invalid mappings in an earlier stage, TPW avoids exponential rounds of database accesses. Moreover, since instance-level exploration is done in step 2, there is no overhead from traversing the database with a potentially large tuple fan-out. Finally, it is reasonable to expect "weaving" to be efficient, because as tuple paths grow larger, their number decreases dramatically, which we have verified by experiments and will describe in Section 6.3.

4.4 Definitions

Before we dive into a detailed description of TPW, we need some definitions. Hereafter, for any graph or tree structure g, we use $V(g)$ to denote all its vertices, $E(g)$ to denote all its edges, and $T(g)$ to denote all its terminal vertices (vertices of degree one).

Schema Graph To create pairwise mappings, we need to search for possible join paths between the sample-containing relations. This requires modeling the source schema as a graph. Because a "null" value can not contain any samples, we only consider inner join here. Since inner join is symmetric, we omit the direction of the foreign key to primary key relationship hereafter.

DEFINITION 2 (SCHEMA GRAPH). *The* schema graph G *is an undirected graph that defines relation joinability according to the foreign key to primary key relationships in* S_S. *It has a vertex* R_i *for each relation* $R_i \in S_S$ *and an edge* (R_i, R_j) *for each foreign key to primary key relationship from* R_i *to* R_j *in* S_S.

[6]We use standard full-text search techniques for such approximate search in our implementation.

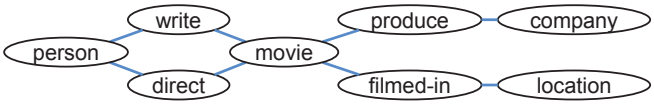


Figure 6: The Schema Graph of the Running Example

Figure 6 shows the schema graph of Example 2.

Relation Path Each possible join path among the sample-containing relations specifies a unique mapping structure. We extract this structure by a path of relations.

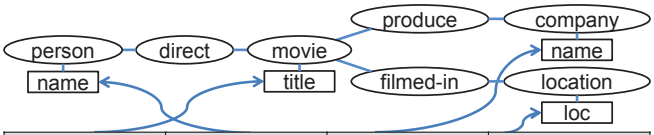
DEFINITION 3 (RELATION PATH). *A* relation path p *is an undirected tree such that: (1)* $\forall u \in V(p)$, u *has a corresponding relation* $R^u \in V(G)$ *and (2)* $\forall (u, v) \in E(p)$, $(R^u, R^v) \in E(G)$.

Note that the same relation can appear multiple times in a relation path, as long as the second condition is satisfied. This potentially means the size of the relation tree is not upper bounded by the size of the schema graph.

Mapping Path Having the join structure defined, a mapping also needs to specify which attributes in which relations are projected to the target relation. We capture this information by a *projection map*, which maps a subset of the target attributes to attributes belonging to vertices on the relation path. Intuitively, each terminal vertex must have at least one projection, or it is redundant. Recall that the target relation has size m, we have the following definition.

DEFINITION 4 (MAPPING PATH). *Given a relation path* p, *let* N *be a subset of* $[m]$ *and* $A(p) = \bigcup_{u \in V(p)} S(R^u)$. *A* mapping path *is* p *augmented with a projection map* $pm : N \to A(p)$, *such that* $\forall v \in T(p)$, $\exists i \in N$ *and* $a \in S(R^v)$, *s.t.* $pm(i) = a$.

The definition says that a mapping path is essentially a relation path whose terminal vertices have some attributes projected to the target. Attributes in non-terminal vertices may also be projected.



1	2	3	4
Avatar	James Cameron	Lightstorm Co.	New Zealand

Figure 7: One Desired Mapping Path for the Running Example

A mapping path is equivalent to a schema mapping in that it can be translated to a SQL query that maps the source database to the target relation. The projection can be fully determined from the projection map while the joining of relations is implied by the structure of the relation path. Hereafter, we use mapping path and schema mapping interchangeably.

Informally, we say a mapping path is ***valid*** iff the corresponding schema mapping is valid. A mapping path can be ***partial***, when N is a proper subset of $[m]$. We define the ***size*** of the mapping path to be the size of N. Specifically, we call a mapping path with size two a ***pairwise mapping path*** and a mapping path with size m a ***complete mapping path***. Our goal mapping paths must be complete. Figure 7 exhibits a complete mapping path that is one of the possible answers to the running example.[7]

Since the size of a relation path is unbounded by the schema, the corresponding schema mapping size is also unbounded. This may lead to an infinite number of mappings, most of which make little practical sense. We will restrict the family of mappings we explore by a join number constraint, which is detailed in Section 4.5.2.

Tuple Path In general, a mapping path may or may not be valid. Because a mapping path is merely a *schema-level object*, and does not guarantee the samples can be connected in the source database

[7]We also show the sample tuple for clarity.

instance following its path. Indeed, a mapping path is valid iff there is a corresponding *instance-level support*. Such a support should comprise source database tuples that connect samples via the mapping path. The formal definition of such a support is given below.

DEFINITION 5 (TUPLE PATH). *A tuple path r is an instantiated mapping path such that: (1) for each vertex $u \in V(r)$, there is an associated tuple $t^u \in I(R^u)$ and (2) for each $(u, v) \in E(r)$, t^u and t^v are directly connected in the source database by the foreign key to primary key relationship between R^u and R^v.*

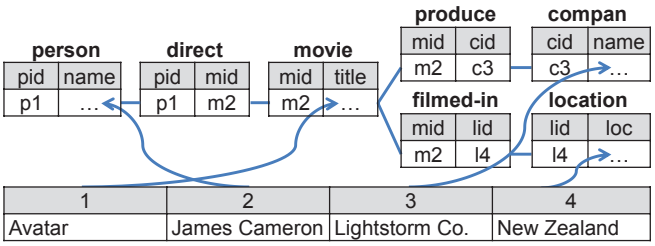


Figure 8: One Tuple Path Supporting the Desired Mapping

We define the ***size*** of a tuple path in the same way we did for a mapping path. We call a tuple path with size two a ***pairwise tuple path*** and one with size m a ***complete tuple path***. Figure 8 depicts a complete tuple path that instantiates the mapping path in Figure 7.[8]

One mapping path may be instantiated to any number of tuple paths. Formally, we say a mapping path is ***valid*** iff it can be instantiated to at least one tuple path. And our goal of sample search is to find those valid complete mapping paths.

4.5 TPW Algorithm

In this section, we elaborate the five steps (see Section 4.3) of the TPW algorithm. Details can be found in Appendix A.

4.5.1 Find Sample Occurrences

A mapping path can be arbitrary. However, if a source attribute does not contain any samples, the mapping path that projects that attribute can not be valid. Therefore, we first narrow our search space by focusing only on the source attributes that contain at least one sample. Formally, we construct a *location map* L, where $L(i)$ ($i \in [m]$) is the set of all the source attributes containing E_i.

EXAMPLE 3. *In the running example, we first search for sample* `Avatar`. *We have* $L(1) = \{\textbf{movie.title}, \textbf{movie.logline}\}$, *because these are all the attributes that contain* `Avatar`. *Similarly,* $L(2) = \{\textbf{person.name}, \textbf{family.family}\}$ *since they contain* `James Cameron`. *A more complete L is shown in Figure 9.*

4.5.2 Pairwise Mapping Path Generation

Next, we generate all the pairwise mapping paths from schema graph G and location map L. The size of such pairwise mapping paths can be arbitrarily large, because the number of joins in these mapping paths is unbounded by the source schema [17]. Fortunately, we realize in practice, the mappings that project two attributes from two relations that are joined via many intermediate relations who have no attribute projected are very rare. Therefore, we set up a threshold value $PMNJ$ (*Pairwise Maximal Number of Joins*) to restrict the family of mapping paths we explore. Specifically, we say a mapping path satisfies the $PMNJ$ constraint iff the largest number of joins between each pair of projected attributes on the path is no larger than $PMNJ$. And we only aim to generate the mapping paths that satisfy the $PMNJ$ constraints. In this running example, we will set $PMNJ = 2$, since it is enough to generate the complete mappings of interest.

The pairwise mapping paths are generated as follows. First, for each $j \in [m]$ and each attribute $A^j \in L(j)$, we issue a breadth-first search from the vertex R^j that contains A^j in the schema graph G with depth limited by $PMNJ$. Whenever we encounter a vertex R^k that contains $A^k \in L(k)$ with $k > j$, we construct a relation path backward from R^k to R^j and augment it with a projection map built from A^j and A^k. The created pairwise mapping paths are stored in a pairwise mapping path map $PMPM$ with key (j, k).f

[8]We present only the primary keys, foreign keys and the projected attributes due to space.

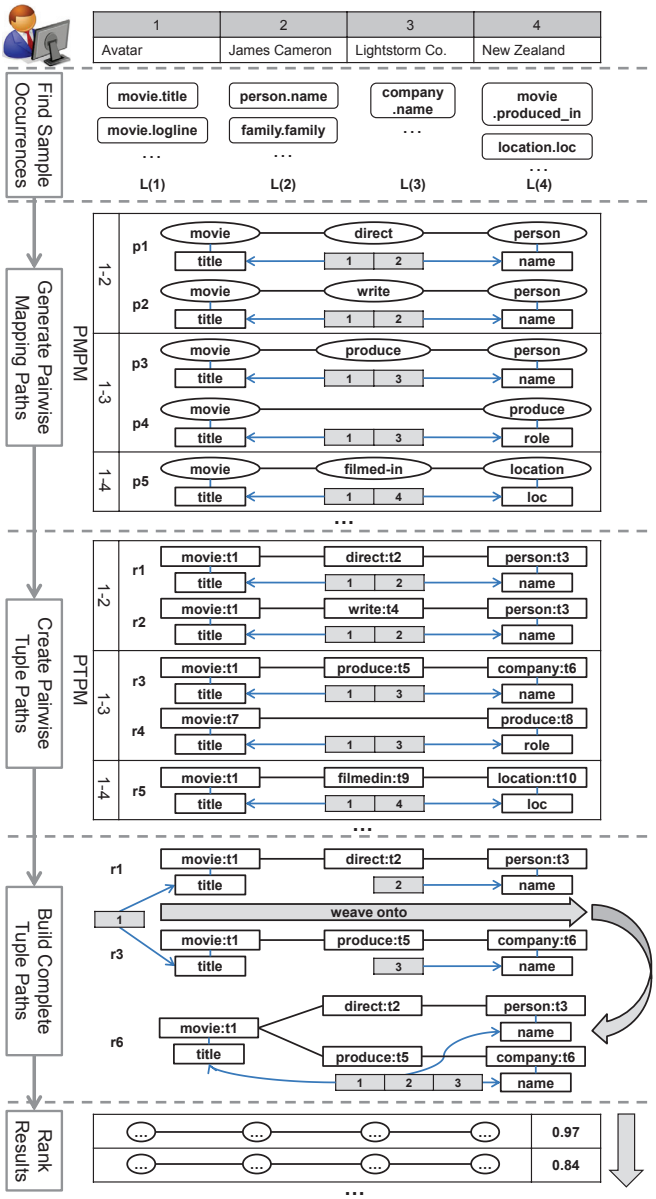


Figure 9: The Tuple Path Weaving Algorithm TPW

EXAMPLE 4. *In the running example, we first generate the pairwise mapping path linking* `Avatar` *and* `James Cameron`. *We start by picking* **movie.title** *from* $L(1)$, **person.name** *from* $L(2)$ *and searching for the relation paths between them. By a breadth-first search from* **movie** *in G, we reach* **Person** *via two paths:* **movie-direct-person** *and* **movie-write-person**. *We augment them with projection information from the two attributes and respectively obtain the mapping paths* ***p1*** *and* ***p2***, *and store them in* $PMPM$ *with key* $(1, 2)$, *as shown in Figure 9.*

4.5.3 Pairwise Tuple Path Creation

Once all the pairwise mapping paths are generated, we produce tuple paths that instantiate them. To do this, we retrieve each pairwise mapping from $PMPM$, translate it into an approximate search

query with the keywords constraints derived from the samples and execute it. If the result set is empty, we prune the corresponding pairwise mapping because any larger mapping containing it must be invalid. Otherwise, we construct a pairwise tuple path from each returned tuple, associate it with the corresponding mapping and store it in a pairwise tuple path map $PTPM$. Since foreign-key connection is normally sparse in real-life datasets, we expect these tuple paths to fit in memory. Subsequent operations can also be completed in memory, because the number of the tuple paths decreases dramatically w.r.t. its size, as we will see in Section 6.3.

EXAMPLE 5. *In Figure 9, suppose the source tuple* **t1** *contains* `Avatar`*,* **t3** *contains* `James Cameron` *and* **t2** *links* **t1** *and* **t3** *by foreign keys. Because these tuples successfully support mapping path* ***p1****, we create a tuple path* ***r1*** *from them, associate it with* ***p1*** *and store it in* $PTPM$ *with key* $(1, 2)$*.* ***r2*** *-* ***r5*** *are created similarly.*

4.5.4 Complete Tuple Path Construction

Since all the necessary information for constructing the complete mapping paths is stored in the pairwise tuple paths, we can then perform the "weaving" in memory. "Weaving" essentially merges a pairwise tuple path onto an existing ***base tuple path*** to produce a new larger tuple path. We use the word "weave" to describe the process of traversing the pairwise tuple path from one end to the other and gradually merging its vertices onto the base tuple path. The "weaving" terminates once a vertex fails to merge.

If all vertices on the pairwise tuple path can be successfully merged onto the base tuple path, the latter will preserve its structure. Otherwise, the "unwoven" part of the the pairwise tuple path will enrich the structure of the base tuple path by adding a "tail" to it. Either way, the cost of "weaving" is upper-bounded by the size of the pairwise tuple path, which is in turn bounded by $PMNJ$.

Since various larger tuple paths may share smaller sub-paths, we "weave" the tuple paths in a bottom-up manner. Specifically, we organize tuple paths by ***levels***. Level n contains all the tuple paths of size n, $n \in \{2..m\}$. For each n from 2 to $m-1$, for each base tuple path in level n, we "weave" a pairwise tuple path onto it to create a tuple path in level $n+1$. This terminates when all the complete tuple paths are "woven" and stored in level m.

The following example illustrates how the "weaving" works. A detailed description can be found in Algorithm 6 in Appendix A.

EXAMPLE 6. *We start by constructing tuple paths of size three from a base tuple path of size two, say,* ***r1*** *in Figure 9. Then, we enumerate all the possible pairwise tuple paths that may be woven onto* ***r1****. Those are pairwise tuple paths whose projection map has exactly one key in common with that of* ***r1****. For instance, it is possible to weave* ***r3*** *or* ***r4*** *onto* ***r1*** *since their keys intersect on key* 1*.*

Next, we attempt to merge the two paths by fusing the two vertices pointed by the common key in both paths. If the tuples associated with the two vertices are identical, the two vertices are fused and the two paths are merged successfully. Otherwise, the merge fails and returns no new path. Here, ***r1*** *and* ***r3*** *successfully merge by fusing the first vertex (since they agree on tuple* **t1***). However,* ***r1*** *and* ***r4*** *fail to merge (because* ***r4*** *has tuple* **t7***).*

Finally, we issue a synchronized search from the fused vertex along the pairwise tuple path to be woven. For each following vertex v on the path, we search in the base tuple path for the next adjacent vertex u such that $t^u = t^v$. If such a vertex does not exist, we stop and return the current tuple path. Otherwise we fuse u and v and proceed to the next vertex. This continues until the whole pairwise tuple path is traversed. In this example, there is no vertex which is adjacent to **movie:t1** *in* ***r1*** *and has a tuple* **t5** *as in* ***r3****. So the process terminates and the new tuple path* ***r6*** *is returned.*

After all tuple paths of size three are generated, we begin to construct tuple paths of size four (the complete tuple paths) from those ones with size three. It is straightforward that ***r5*** *can be woven onto* ***r6****, producing the complete tuple path shown in Figure 8. Another complete tuple path is constructed by weaving* ***r3*** *and* ***r5*** *onto* ***r2****.*

4.5.5 Ranking

We extract all the complete mapping paths from the complete tuple paths and rank them before returning them to the user. For each complete tuple path, we define its score to be a weighted sum of two scores: a matching score which indicates how well the samples match the actual data in the tuple, and a complexity score which is the number of joins in the tuple path. The score of a complete mapping path is the average score of all its corresponding tuple paths.

4.6 Soundness and Completeness

The TPW algorithm is sound and complete in that every complete tuple path generated corresponds to a valid schema mapping, and every valid schema mapping satisfying the PMNJ constraint is discovered by the algorithm. The soundness is straightforward. The completeness is because, for any valid schema mapping that generates a tuple t containing the sample tuple, we can always construct the corresponding complete tuple path from the source tuples that contribute to t, which are completely recorded when generating the pairwise tuple paths. The proof can be found in Appendix B.

5. SAMPLE PRUNING

After the initial set of valid candidate mappings is generated, the user may continue to enter additional samples to prune the candidate set. We call this process sample pruning. Given that our sample search returns a limited number of candidate mappings, the pruning can be processed in a straightforward manner as follows.

Pruning by Attribute Suppose the user enters a new sample E_i in column i on another row in the input spreadsheet. We record all the source attributes that contain E_i. Any mapping path that does not map i to any of these attributes is then discarded.

Pruning by Mapping Structure After the user enters a new sample E_i in column i, whenever there is more than one sample on the same row, we execute an approximate search query for each candidate mapping with the keywords constraints derived from these samples. We discard a mapping if the search returns zero result.

EXAMPLE 7. *In the running example, suppose the user continues to enter* `Big Fish` *on the first column of the second row, and we find that the attribute* **Movie.logline** *does not contain* `Big Fish`*. As a result, any mapping that maps the first column to* **Movie.logline** *will be discarded. Similarly, if the user continues to enter* `Tim Burton` *on the second column of the second row, we will issue a search query with* `Big Fish` *and* `Tim Burton` *on each candidate mapping. Any mapping that joins* **Movie** *and* **Person** *via* **Writer** *will be discarded, because the writer of* `Big Fish` *is not* `Tim Burton` *and, as a result, the query will return empty set.*

6. EVALUATION

The critical test for our sample-driven mapping tool is whether it truly renders schema mapping tasks feasible for end-users. In this section we demonstrate a positive result in two ways. First, we show that our sample-driven tool reduces user effort for completing the mapping task when compared to both a standard match-driven system and a state-of-the-art QBE-like mapping tool. Specifically, we present a user study that compares the usability of the three tools and show that our sample-driven tool enables an average user to complete a schema mapping task in just *1/5-1/4th* the overall

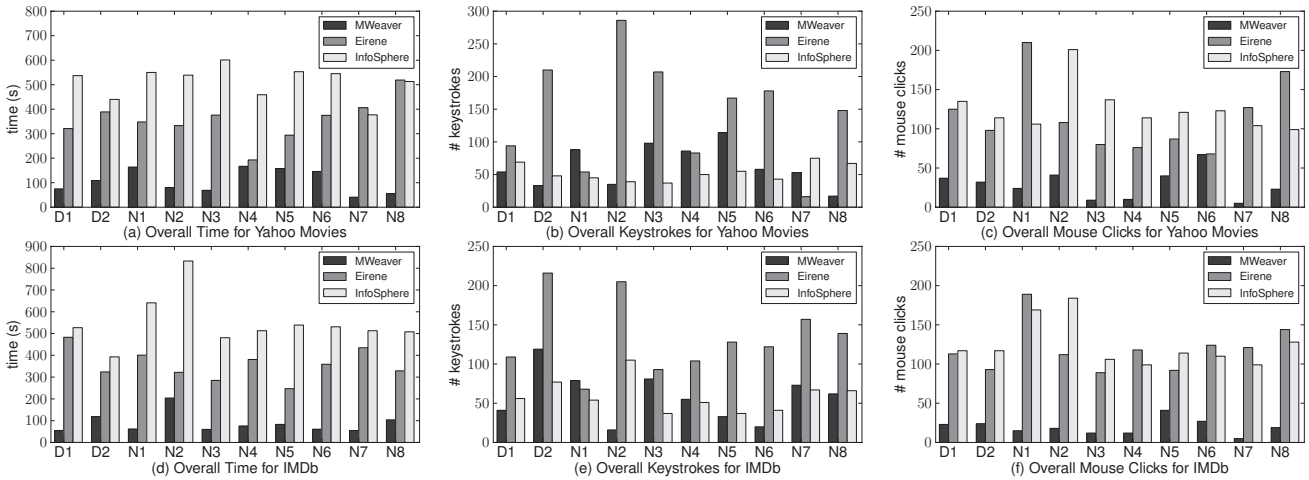


Figure 10: The overall time, keystrokes and mouse clicks for completing the mapping task on Yahoo Movies and IMDb. D1 and D2 are database experts. N1-N8 are non-technical users.

time required by the other tools. Second, we show, with a synthetic mapping task workload, that the sample-driven approach is able to find the goal mapping with just about *two rows* of samples.

The sample search described in Section 4.1 is intuitively expensive since the search space is the whole source database instance. However, it relies on fast cooperation between the user and the system, so any serious computational delay could render the tool unusable. We conduct a performance experiment and show that the *Tuple Path Weaving* algorithm, TPW, is efficient enough to underpin our sample-driven tool. Indeed, we show that TPW is able to find the candidate mappings in seconds, in a 500MB sized database. In contrast, a naive approach constructed in a traditional keyword search manner yields runtime up to hundreds of seconds, far more than a user of a sample-driven tool is likely to tolerate.

6.1 Implementation and Environment

Our implementation of a sample-driven tool, MWEAVER, has a UI written in HTML and javascript that communicates via AJAX to a backend engine written in Java Servlet. We use two datasets in our experiments: Yahoo Movies and IMDb. The Yahoo Movies dataset is 500MB in size and contains 43 relations and 131 attributes. The IMDb dataset is 2GB in size and contains 19 relations and 57 attributes. Both datasets are stored in MYSQL 5.

We use the full-text search engine in MySQL to implement the approximate search query. We set PMNJ to two, which is sufficient for our goal mappings. All the experiments were run on a desktop machine with an Intel Core i7 860 @ 2.80GHz and 8GB RAM.

6.2 Usability

We compare the usability of MWEAVER against two tools. The first is IBM InfoSphere Data Architect (Version: 7.5.3.0), which is a commercialized version of the Clio project, and serves as a typical representative of the state-of-the-art match-driven tools. The second tool, Eirene [8], is a schema mapping tool recently developed by Alexe, et al. to help users design schema mappings through a QBE-like interface. We design the following user study scenario to be simple so that it can be carried out with non-technical subjects.

Suppose a user is browsing her favorite movies on the Yahoo Movies/IMDb website, and finds two pages of particular interests: the movie information page, which lists various properties of a movie, and the personal information page, which displays the biography of the contributors (e.g., the writers and the actors). However, the user finds these pages overwhelming since they contain every piece of related information spanning from a one-hundred-line movie description to an actor's achievements in forty years. In fact, all she wants is just a small subset of all these messy attributes.

Assume the user is only interested in the release date, the production company and the director of the movie, we formalize the mapping task as follows. The source schema S_S includes the complete Yahoo Movies/IMDb Database schema, and the target schema S_T contains only one relation comprising the following attributes: **Movie**: the title of the movie, **ReleaseDate**: the release date of the movie, **ProductionCompany**: the company which produces the movie and **Director**: the director of the movie. The user is asked to develop a schema mapping that transforms the data under S_S to the new database with schema S_T, for both Yahoo Movies and IMDb. The goal mappings are depicted by mapping paths in Figure 11.

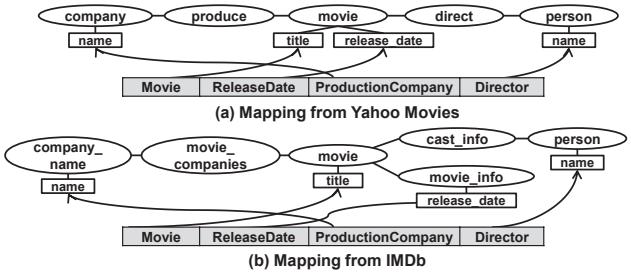


Figure 11: Task Schema Mappings: (a) Yahoo Movies, (b) IMDb.

We recruited eight non-technical subjects and two database experts for comparison. All of them were asked to complete the schema mapping task using MWEAVER, IBM InfoSphere Data Architect and Eirene[9]. We recorded the overall time, keystrokes and mouse clicks to complete the task for each user with each tool, on both datasets. Because the latter two tools require substantial knowledge about the source schema, we provided complete technical support when the users were using these two tools. The results are shown in Figure 10. There was no substantial performance difference between database experts and end-users, or Yahoo Movies and IMDb datasets.

The results demonstrate that, on average, creating the mapping with MWEAVER only needs 1/5 the overall time that required by IBM InfoSphere Data Architect, and 1/4 the overall time required by Eirene[10]. This difference is partly explained by the reduction in number of keystrokes and mouse clicks. The rest is attributed to the (not directly measurable) cognitive burden on the user in reasoning with unfamiliar source schema in the other tools.

While Eirene also asks the users to enter examples as in traditional QBE systems to design the mapping, MWEAVER saves

[9]We randomized the order to counterbalance the learning effect.

[10]All the above differences are statistically significant with p-values < 0.0002 according to the Mann-Whitney test.

around half of the keystrokes required by Eirene. This is because MWEAVER requires only target sample entry aided by auto-completion, while Eirene requires the user to fully specify the examples under both related source and target schema. Finally, MWEAVER only needs 1/5 mouse clicks required by the other two tools, since the UI of MWEAVER is simply a spreadsheet, which the users may naturally navigate through using traditional spreadsheet hot keys.

At the end of the user study, we asked each user how much she was satisfied with each tool and recorded a satisfaction score scaled from one (very dissatisfied) to five (very satisfied). MWEAVER has an average score of 4.7, InfoSphere 2.7 and Eirene 3.45.

We also found that MWEAVER only requires a few user-input samples to derive the goal mapping. However, the scale of the user study prevented us from collecting statistically significant data. Therefore, we focused on the Yahoo Movies dataset and constructed a synthetic mapping task workload containing tasks similar to the one used in the user study. Specifically, we defined three sets of task mappings. All the mappings in the same task set share the same relation path. The relation path has two, three and four joins for the three task sets. Each set contains four mappings, which vary in the target schema size from three to six.

For each mapping in each task set, we simulated user-input by repeatedly randomly sampling instances from a synthetic target database and fed them into MWEAVER until the mapping is discovered. Each task was repeated for one hundred times and the average number of samples required is shown in Table 1. Recall that one row in the target has m samples, the results show that it only takes the user approximately two rows of samples to obtain the goal mapping.

Size of S_T (m)	3	4	5	6
Task Set 1	7.24	9.35	10.80	14.98
Task Set 2	5.08	8.50	11.55	16.18
Task Set 3	6.97	9.27	11.71	13.67

Table 1: The Average Number of Samples to Generate the Goal Mapping.

Finally, for each of the three task sets, we examined the number of candidate mappings (valid complete mapping paths) as the number of user-input samples increased. From the results shown in Figure 12, we observe that the number of candidate mappings drops dramatically as the number of user-input samples increases. Although in the worst case, the system may need about $8m$ samples to discover the goal mapping, the average is only about $2m$, where m is the target schema size.

6.3 Performance

We conducted performance experiments with the same set-up introduced above to demonstrate that TPW is efficient enough to meet interactive requirements.

We first measured the average response time for both searching and pruning to provide an overall sense of our system performance. The results shown in Table 2 demonstrates that MWEAVER is able to respond to a user-input sample within 1s for searching and 50ms for pruning. In practice, the computation for the previous input is largely paralleled with the next user data entry so that the absolute waiting time observed by the user is very small.

Task Set	Size of S_T	3	4	5	6
1	Searching (ms)	534.35	655.03	639.49	577.25
	Pruning (ms)	34.27	24.46	35.13	58.54
2	Searching (ms)	177.98	363.32	407.69	450.91
	Pruning (ms)	27.23	40.63	58.24	62.20
3	Searching (ms)	305.89	442.78	761.69	817.38
	Pruning (ms)	32.53	24.46	40.24	51.58

Table 2: The Average Response Time for Searching and Pruning.

Next, we demonstrate that TPW is much more efficient compared to the naive approach derived directly from the schema-based keyword search techniques [17, 4]. To do so, we developed a naive algorithm which enumerated all the complete mapping paths (no matter valid or not) in the same way as the equivalent "candidate networks" are generated in [17], and validated them by executing an approximate search query translated from each of them. We performed both algorithms on the same workload described above. Table 3 shows the average overall time to complete the sample search for both algorithms. While TPW completed the search within 5 seconds on average, the time required by the naive algorithm grew dramatically. The naive algorithm failed beyond size 5 because the enumerated mapping paths exhausted the memory.

Task Set	Size of S_T	3	4	5	6
1	TPW (ms)	3735.48	3775.22	3008.52	3695.28
	Naive (ms)	35891.43	734319.25	–	–
2	TPW (ms)	578.47	1354.05	2043.77	2804.33
	Naive (ms)	1273.62	41976.94	–	–
3	TPW (ms)	1044.49	1674.66	3885.44	4727.86
	Naive (ms)	11644.93	388723.31	–	–

Table 3: The Average Search Time for TPW and the Naive Algorithm.

From this experiment, we also see that, even if the sample search problem is NP-hard, TPW is typically able to solve it in near-linear time. The intuition behind this huge improvement is that, invalid mapping paths are pruned away much earlier when smaller tuple paths are being processed in MWEAVER. To offer a clearer insight into the performance experiment, we measured the total number of tuple paths (with various size) processed in TPW and the number of potentially valid complete mapping paths generated by the naive algorithm. Both numbers are compared with the exact number of valid mapping paths given a sample tuple. The result is shown in Table 4. Although the number of tuple path processed in our approach increased near-exponentially, it was still many fewer than the number of complete mapping paths that were generated and needed to be validated by the naive algorithm. In addition, the tuple paths were quickly processed in memory in TPW. In contrast, in the naive algorithm, the complete mapping paths had to be validated via expensive database accesses. This explains the performance difference observed in Table 3.

Task Set	Size of S_T	3	4	5	6
1	# Valid MP	2.67	5.05	4.52	6.00
	# TP Woven	15.46	207.40	719.67	3403.20
	# Naive MP	964.38	163634.45	–	–
2	# Valid MP	2.69	2.55	6.61	6.16
	# TP Woven	5.66	39.6	530.16	2008.39
	# Naive MP	35.31	967.25	–	–
3	# Valid MP	2.19	3.45	4.53	6.85
	# TP Woven	4.38	72.69	640.49	4149.37
	# Naive MP	318.36	10582.93	–	–

Table 4: Comparison between TPW and the Naive Algorithm. (MP=Mapping Path, TP=Tuple Path)

Our final experiment examined the average total number of tuple paths generated at each level in the TPW algorithm. The results are shown in Figure 13. We observed that the number of valid tuple paths decreases dramatically as the algorithm approached the full size of the target schema. This is because in a real-life dataset, the distributions of samples in different source attributes are relatively independent, making specific combinations unlikely as the size of the combination increases.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a sample-driven schema mapping approach to facilitate the data integration tasks for end-users. While it is hard for end-users to understand the precise semantics of schemas and mappings, providing sample instances is much easier for them; we exploit this to design and prototype MWEAVER,

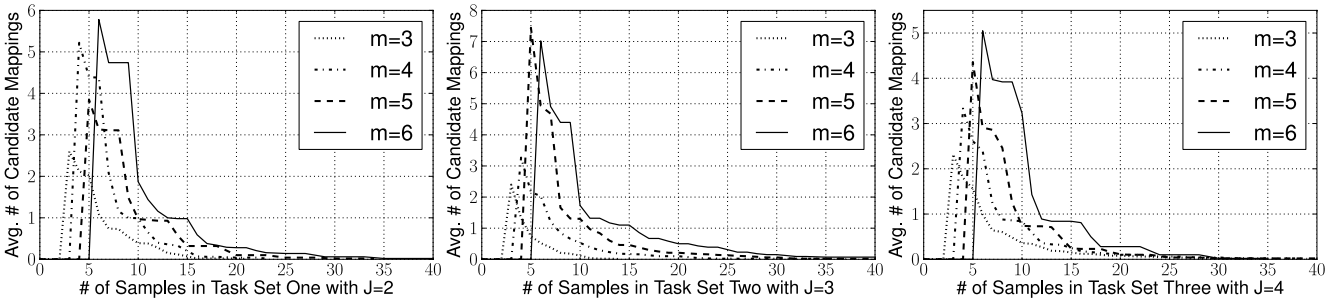


Figure 12: Average Number of Candidate Mappings w.r.t. the Number of Simulated Samples. J: number of joins in each mapping. m: the target schema size.

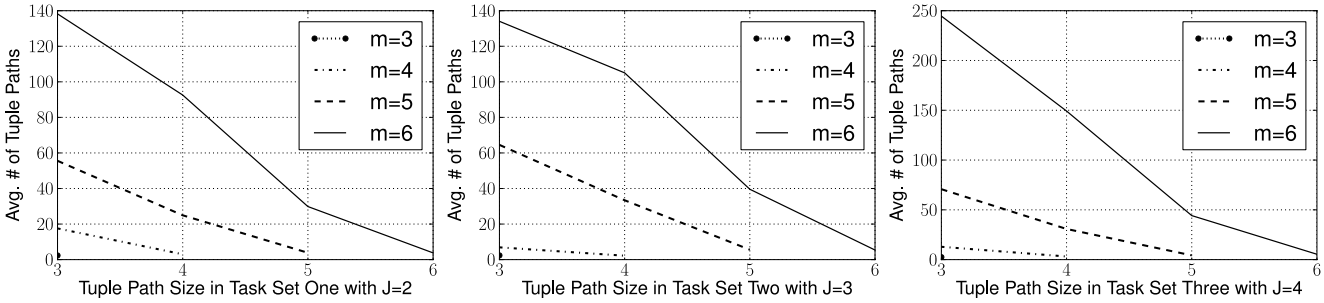


Figure 13: Average Number of Tuple Paths Generated at Each Level in TPW. J: number of joins in each mapping. m: the target schema size.

our sample-driven schema mapping tool, which renders the schema mapping tasks for end-users much more feasible.

We have studied the sample search problem in such a sample-driven approach, and proposed an algorithm to efficiently generate candidate mappings from user-input samples. Through user studies and simulated experiments on real-world datasets, we have demonstrated that MWEAVER is more usable than existing schema mapping tools and our solution to the sample search problem is efficient enough to meet interactive requirements.

Our approach relies on the user-input to be roughly present in the source instance. In case the user-input is totally irrelevant to the source, it will invalidate previously generated correct mappings. We are studying on how to provide features that will automatically suggest relevant data and warn the user about irrelevant one. In this paper, we primarily dealt with samples of string values. If the source contains many numerical attributes, a numerical sample may be contained by multiple source attributes, which will in turn degrade system performance. Also, since the number of tuple paths may grow rapidly w.r.t. the source database size, it is desirable to provide some insights into the scalability of our approach. Finally, we currently only support mappings in the form of project-join queries, which is a subset of GAV mappings. It would be interesting to study how to extend our approach to LAV and GLAV mappings. We plan to address these issues in the future work.

8. REFERENCES

[1] Altova mapforce. http://www.altova.com/mapforce.html.

[2] Microsoft biztalk server. http://www.microsoft.com/biztalk/en/us/.

[3] Stylus studio. http://www.stylusstudio.com/.

[4] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, page 5, 2002.

[5] B. Alexe, L. Chiticariu, R. Miller, and W. Tan. Muse: Mapping understanding and design by example. In *ICDE*, pages 10–19, 2008.

[6] B. Alexe, L. Chiticariu, and W. Tan. SPIDER: a schema mapPIng DEbuggeR. In *VLDB*, pages 1179–1182, 2006.

[7] B. Alexe, P. Kolaitis, and W. Tan. Characterizing schema mappings via data examples. In *SIGMOD*, pages 261–272, 2010.

[8] B. Alexe, B. ten Cate, P. Kolaitis, and W. Tan. Designing and refining schema mappings via data examples. In *SIGMOD*, page 133, 2011.

[9] P. Barceló. Logical foundations of relational data exchange. *SIGMOD Rec.*, 38:49–58, June 2009.

[10] Z. Bellahense, A. Bonifati, and E. Rahm, editors. *Schema Matching and Mapping*. Springer, 2011.

[11] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, S. Sudarshan, and I. Bombay. Keyword searching and browsing in databases using BANKS. In *ICDE*, page 431, 2002.

[12] M. Cafarella, A. Halevy, and N. Khoussainova. Data integration for the relational web. *VLDB*, 2(1):1090–1101, 2009.

[13] H. Do and E. Rahm. COMA: a system for flexible combination of schema matching approaches. In *VLDB*, pages 610–621, 2002.

[14] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, pages 509–520, 2001.

[15] C. Drumm, M. Schmitt, H. Do, and E. Rahm. Quickmig: automatic schema matching for data migration projects. In *CIKM*, pages 107–116, 2007.

[16] H. Elmeleegy, M. Ouzzani, and A. Elmagarmid. Usage-based schema matching. In *ICDE*, pages 20–29, 2008.

[17] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, page 681, 2002.

[18] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. *SIGMOD*, pages 13–24, 2007.

[19] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[20] J. Kang and J. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, pages 205–216, 2003.

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

[22] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[23] J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *ICDE*, pages 57–68, 2005.

[24] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.

[25] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[26] A. Nandi and P. Bernstein. HAMSTER: using search clicklogs for schema and taxonomy matching. *VLDB*, 2(1):181–192, 2009.

[27] L. Popa, Y. Velegrakis, M. Hernández, R. Miller, and R. Fagin. Translating web data. In *VLDB*, pages 598–609, 2002.

[28] P. Talukdar, Z. Ives, and F. Pereira. Automatically incorporating new sources in keyword search-based data integration. In *SIGMOD*, pages 387–398, 2010.

[29] P. Talukdar, M. Jacob, M. Mehmood, K. Crammer, Z. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *VLDB*, 1(1):785–796, 2008.

[30] L. Yan, R. Miller, L. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD*, page 485, 2001.

[31] M. Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438, 1975.

APPENDIX

A. TPW ALGORITHMS

A.1 Find Sample Occurrences

Given a sample tuple and a source database, Algorithm 1 locates in the source all the occurrences of each sample in that tuple by scanning all the source attributes (loops at line 3 and 5). Whenever a source attribute contains a sample (line 6), it is registered to the corresponding location map (line 7 and 11). The check on line 6 is done by a standard full-text search on an individual column which has a pre-computed inverted-index. Line 7 and 11 nest the location map by relation so we can easily generate relation path hereafter.

Algorithm 1 LocateSample

Input: A sample tuple $(E_1, ..., E_m)$, source database D_S with schema S_S
Output: A location map $L = \{L_1, ..., L_m\}$

1: Initialize L
2: **for all** $i \in \{1..m\}$ **do**
3: **for all** relation $R^i \in D_S$ **do**
4: $L_{R^i} \leftarrow \emptyset$
5: **for all** attribute $A \in R^i$ **do**
6: **if** $\exists t \in R^i$ s.t. $t[A] \succeq E_i$ **then**
7: $L_{R^i} \leftarrow L_{R^i} \cup \{A\}$
8: **end if**
9: **end for**
10: **if** $L_{R^i} \neq \emptyset$ **then**
11: $L_i(R^i) \leftarrow L_{R^i}$
12: **end if**
13: **end for**
14: **end for**

A.2 Pairwise Mapping Path Generation

Algorithm 2 constructs all the pairwise mapping paths that satisfy the $PMNJ$ constraint by issuing a breadth-first search on the schema graph G. To perform the breadth-first search, we define a structure *relation node*, which has three properties: $relation$ which points to the corresponding relation in the source database, $dist$ which records the number of joins from this node to the origin and $parent$ which stores the previous node during the search. We also define *Path Vertex*, which composes a relation path. It has a property $relation$ which stores the corresponding relation, and $neighbor$ which stores all its neighbors. A relation path maintains a $lookup$ table to manage vertex membership. Also, we compute a map in advance to prepare for mapping path creation.

We first construct relation paths from the schema graph G and the location map L, as described in Algorithm 2. For each sample E_i (line 2) and each relation R^i that contains it (line 3), we issue a breadth-first search from the relation by calling $Grow()$ (See Algorithm 3). $Grow()$ returns every relation path whose length is no larger than $PMNJ$ and connects R^i with another relation R^j containing E_j ($j > i$). Finally we construct all the pairwise mapping paths by appending registered attributes to each returned relation path by invoking $Create()$ (See Algorithm 4).

A.3 Pairwise Tuple path Creation

For each pairwise mapping path in $PMPM(i, j)$, we then search for all the pairwise tuple paths that instantiate it, and store them in $PTPM(i, j)$. This is done by the following steps. (1) We translate the mapping path into a SQL query, with the join conditions defined by the path structure. (2) For each relation on the mapping path, we project all of its primary keys. (3) We expand the query by full-text search constraints derived from the sample tuple. (4) We execute the full-text search query. (5) For each tuple in the result set: (i) for each relation on the mapping path, we generate a universal tuple id for the provenance tuple in that relation from the schema of that relation and the values of all the projected primary keys. (ii) We collect such tuple ids for all the relations on the mapping path, align them in the same structure as in the mapping path and store them in $PTPM$.

Algorithm 2 GeneratePairwiseMapingPath

Input: The location map L, the schema graph G
Output: $PMPM$, a map from (i,j) to a set of pairwise mapping paths, where $i, j \in \{1..m\} \wedge i < j$

1: Initialize $PMPM$
2: **for all** $i \in \{1..m\}$ **do**
3: **for all** relation $R^i \in Keys(L_i)$ **do**
4: Initialize relation node r
5: $r.relation \leftarrow R^i$
6: $r.dist \leftarrow 0$
7: $r.parent \leftarrow null$
8: $rs =$ Grow($L, r, i, G, PMNJ$)
9: Create($rs, PMPM$)
10: **end for**
11: **end for**

Algorithm 3 Grow

Input: The location map L, a relation node r containing E_i, the index i, the schema graph G, the maximal number of joins $PMNJ$
Output: All the relation paths from r to s within $PMNJ$ joins, where s is a relation node containing E_j and $j > i$

1: Initialize a relation path set rs
2: Initialize a queue of relation node Q
3: $Q.push(r)$
4: **while** Q is not empty **do**
5: $c \leftarrow Q.pop()$
6: **for all** $j \in \{i + 1..m\}$ **do**
7: **if** $c \in Keys(L_j)$ **then**
8: Create relation path p backward from j to i.
9: $rs.add(p)$
10: **end if**
11: **end for**
12: **if** $c.dist < PMNJ$ **then**
13: **for all** R' that is adjacent to $c.relation$ in G **do**
14: Initialize relation node n
15: $n.relation \leftarrow R'$
16: $n.dist \leftarrow c.dist + 1$
17: $n.parent \leftarrow c$
18: $Q.push(n)$
19: **end for**
20: **end if**
21: **end while**

A.4 Complete Tuple Path Construction

After all the pairwise tuple paths are generated, we no longer need to access the database since the tuple paths contain complete information to derive the candidate mappings. Specifically, we compute the complete valid mapping paths by "weaving" the tuple paths in a bottom-up manner, as described in Algorithm 5. At each level (line 3), we retrieve all the tuple paths generated from the previous level (line 4) and try to weave each pairwise tuple path onto these base tuple paths (line 5-18). The weaving is described in Algorithm 6, which simultaneously traversed two tuple paths (line 11, 12), compare corresponding vertices by checking the identities of the tuples associated with them (line 13) and update path structure on successful merges (line 14-16).

B. PROOFS

Here we prove that the tuple path weaving TPW algorithm that solves the sample search problem is sound and complete.

Algorithm 4 Create

Input: A relation path set rs, the pairwise mapping path map $PMPM$
Output: $PMPM$

for all $r \in rs$ **do**
  Let $i, j \in DOM(r.pm)$ and $i < j$
  for all $a_i \in R^{r.pm(i)}$ **do**
    for all $a_j \in R^{r.pm(j)}$ **do**
      Initialize a mapping path p from r
      Append a_i to $p.pm(i)$
      Append a_j to $p.pm(j)$
      $PMPM(i, j) \leftarrow p$
    end for
  end for
end for

Algorithm 5 GenerateCompleteTuplePath

Input: The pairwise tuple path map $PTPM$, the size of the target schema m
Output: The complete tuple path map $CTPM$

Initialize tuple path set pl, npl
$pl \leftarrow PTPM$
for all $i \in \{2..m\}$ **do**
  for all $tp \in pl$ **do**
    for all $ptp \in PTPM$ **do**
      Initialize a set ck
      $ck \leftarrow DOM(tp.pm) \cap DOM(ptp.pm)$
      if $ck.size = 1$ **then**
        Let k be the only element in ck
        Initialize new tuple path ntp, $nptp$
        $ntp \leftarrow tp$
        $nptp \leftarrow ptp$
        $ntp \leftarrow Weave(ntp, nptp, k)$
        if $ntp \neq null$ **then**
          $npl.add(ntp)$
        end if
      end if
    end for
  end for
  $pl \leftarrow npl$
end for
$CTPM \leftarrow pl$

Algorithm 6 Weave

Input: A tuple path tp of size n, A pairwise tuple path ptp
Output: A tuple path ntp of size $n + 1$

Initialize a set $visited$ for visited path vertex
$u \leftarrow tp.map(k)$
$v \leftarrow ptp.map(k)$
if $u.tuple\_id \neq v.tuple\_id$ **then**
  $ntp \leftarrow null$
  return
end if
$visited.add(u)$
$ptp.lookup.remove(v)$
while $ptp.lookup$ is not empty **do**
  Let v' be the next vertex of v in ptp
  for all $u' \in u.neighbor \wedge u' \notin visited$ **do**
    if $u'.tuple\_id \neq v'.tuple\_id$ **then**
      $u.neighbor.add(v')$ {Update path structure}
      $v'.neighbor.remove(v)$
      $v.neighbor.add(u)$
      return
    end if
    $u \leftarrow u'$
    $v \leftarrow v'$
    $visited.add(u)$
    $ptp.lookup.rmove(v)$
  end for
end while

B.1 Soundness

THEOREM 1 (SOUNDNESS). *Any valid complete mapping path generated by* TPW *corresponds to a valid schema mapping.*

We first give some definitions.

DEFINITION 6 (VALID SCHEMA MAPPING). *Given a sample tuple $(E_1, ..., E_m)$, we say a schema mapping M is valid on $N \subseteq [m]$ iff $\exists t \in M(D_S)$ s.t. $\forall i \in N, t[A_i] \succeq E_i$.*

DEFINITION 7 (TUPLE PATH PROJECTION). *Given a tuple path p with projection map pm, its projection t_p is a tuple such that $\forall i \in DOM(pm), t_p[i] = t^{u_a}[a]$, where $a = pm(i)$ and u_a is the vertex containing a in p.*

DEFINITION 8 (VALID TUPLE PATH). *Given a sample tuple $(E_1, ..., E_m)$ and a tuple path p with projection map pm, we say p is valid iff $\forall i \in DOM(pm), t_p[i] \succeq E_i$.*

Hereafter, given a mapping path or tuple path p, we call the schema mapping translated from it its *corresponding schema mapping* and denote it by M_p. Securely, if a tuple path p is valid, then M_p must be valid on $DOM(p.pm)$ since t_p exists in $M_p(D_S)$ and $\forall i \in DOM(p.pm), t_p[A_i] \succeq E_i$.

Then, we have the following lemma.

LEMMA 1. *Let p be a tuple path of size n, q be a pairwise tuple path and $DOM(p.pm) \cap DOM(q.pm) = \{k\}$. Let $r = Weave(p, q, k)$ and assume $r \neq null$. If p is valid and q is valid, then r is also valid.*

The proof to Lemma 1 is straightforward. According to Algorithm 6, since $r \neq null$, p and q must have been successfully merged. Thus $t_r = t_p \cup t_q$ and $\forall i \in DOM(r.pm), t_r[A_i] \succeq E_i$.

Finally, we give the proof to Theorem 1.

PROOF SOUNDNESS. According to the definition, a mapping path is valid iff there is at least one tuple path that instantiates it. So it is equivalent to prove that any valid complete tuple path generated by TPW corresponds to a valid schema mapping. Since we generate pairwise tuple paths by executing full-text search queries in the source database, every pairwise tuple path p must be valid. Also, according to Lemma 1, if each tuple path or size n is valid, then each tuple path of size $n + 1$ must also be valid. According to mathematical induction, every complete tuple path must be valid. For each complete tuple path p, M_p defines the corresponding valid schema mapping. □

B.2 Completeness

THEOREM 2 (COMPLETENESS). *Any valid schema mapping whose corresponding mapping path satisfies the PMNJ constraint must be discovered by* TPW.

Hereafter, we assume every mapping path satisfies the PMNJ constraint for simplicity. We first have the following lemma.

LEMMA 2. *If all the valid tuple paths of size n are discovered by* TPW*, then all the valid tuple paths of size $n + 1$ must also be discovered.*

PROOF. Suppose we have a valid tuple path r of size $n + 1$. We decompose it into two parts as the following. First we choose any of its terminal vertex, say u. According to Definition 4, there must be a map index associated with it. We denote that index by i. Then we traverse r from u until we meet the first vertex v which has an attribute projected by another map index j. We split r on v together with its map split on j into two tuple paths: a pairwise tuple path p connecting u and v, and a tuple path q of size n containing the rest of r including v. Because r is valid, it is straightforward that p and q are both valid. According to the procedure we generate the pairwise tuple paths, each valid pairwise tuple path must be discovered by TPW. So p must be discovered. According to the hypothesis, q must also be discovered. Let $r' = Weave(q, p, j)$. Since TPW is deterministic, we must have $r' = r$. In other words, r must also be discovered. □

This leads to the proof to Theorem 2.

PROOF COMPLETENESS. Because a schema mapping is valid iff there exists a corresponding valid tuple path, the proof is equivalent to: TPW is able to discover any valid tuple path. According to the procedure we generate the pairwise tuple paths, all the valid pairwise tuple paths must be discovered. According to Lemma 2 and mathematical induction, all the complete valid tuple paths must also be discovered. □