Data Migration using Datalog Program Synthesis

Yuepeng Wang
University of Texas at Austin
ypwang@cs.utexas.edu

Rushi Shah
University of Texas at Austin
rshah@cs.utexas.edu

Abby Criswell
University of Texas at Austin
abby@cs.utexas.edu

Rong Pan
University of Texas at Austin
rpan@cs.utexas.edu

Isil Dillig
University of Texas at Austin
isil@cs.utexas.edu

ABSTRACT

This paper presents a new technique for migrating data between different schemas. Our method expresses the schema mapping as a Datalog program and automatically synthesizes a Datalog program from simple input-output examples to perform data migration. This approach can transform data between different types of schemas (e.g., relational-to-graph, document-to-relational) and performs synthesis efficiently by leveraging the semantics of Datalog. We implement the proposed technique as a tool called DYNAMITE and show its effectiveness by evaluating DYNAMITE on 28 realistic data migration scenarios.

PVLDB Reference Format:
Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, Isil Dillig. Data Migration using Datalog Program Synthesis. *PVLDB*, 13(7): 1006-1019, 2020.
DOI: https://doi.org/10.14778/3384345.3384350

1. INTRODUCTION

A prevalent task in today's "big data era" is the need to transform data stored in a source schema to a different target schema. For example, this task arises frequently when parties need to exchange or integrate data that are stored in different formats. In addition, as the needs of businesses evolve over time, it may become necessary to change the schema of the underlying database or move to a different type of database altogether. For instance, there are several real-world scenarios that necessitate shifting from a relational database to a non-SQL database or vice versa.

In this paper, we present a new programming-by-example technique for automatically migrating data from one schema to another. Given a small input-output example illustrating the source and target data, our method automatically synthesizes a program that transforms data in the source format to its corresponding target format. Furthermore, unlike prior programming-by-example efforts in this space [5, 38, 49], our method can transform data between several types of database schemas, such as from a graph database to a relational one or from a SQL database to a JSON document.

Proceedings of the VLDB Endowment, Vol. 13, No. 7
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3384345.3384350

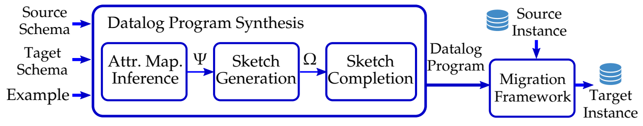


Figure 1: Schematic workflow of DYNAMITE.

One of the key ideas underlying our method is to reduce the automated data migration problem to that of synthesizing a Datalog program from examples. Inspired by the similarity between Datalog rules and popular schema mapping formalisms, such as GLAV [5, 20] and tuple-generating dependencies [37], our method expresses the correspondence between the source and target schemas as a Datalog program in which extensional relations define the source schema and intensional relations represent the target. Then, given an input-output example $(\mathcal{I}, \mathcal{O})$, finding a suitable schema mapping boils down to inferring a Datalog program $\mathcal{P}$ such that $(\mathcal{I}, \mathcal{O})$ is a model of $\mathcal{P}$. Furthermore, because a Datalog program is executable, we can automate the data migration task by simply executing the synthesized Datalog program $\mathcal{P}$ on the source instance.

While we have found Datalog programs to be a natural fit for expressing data migration tasks that arise in practice, *automating* Datalog program synthesis turns out to be a challenging task for several reasons: First, without some a-priori knowledge about the underlying schema mapping, it is unclear what the structure of the Datalog program would look like. Second, even if we "fix" the general structure of the Datalog rules, the search space over all possible Datalog programs is still very large. Our method deals with these challenges by employing a practical algorithm that leverages both the semantics of Datalog programs as well as our target application domain. As shown schematically in Figure 1, our proposed synthesis algorithm consists of three steps:

Attribute mapping inference. The first step of our approach is to infer an *attribute mapping* Ψ which maps each attribute in the source schema to a *set* of attributes that it may correspond to. While this attribute mapping does not uniquely define how to transform the source database to the target one, it substantially constrains the space of possible Datalog programs that we need to consider.

Sketch generation. In the next step, our method leverages the inferred attribute mapping Ψ to express the search space of all possible schema mappings as a *Datalog program sketch* where some of the arguments of the extensional relations are unknown. While such a sketch represents a *finite* search space, this space is exponentially large, making it infeasible to naively enumerate all programs defined by the sketch.

Sketch completion. The final and most crucial ingredient of our method is the *sketch completion* step that performs Datalog-specific deductive reasoning to dramatically prune the search space. Specifically, given a Datalog program that does not satisfy the input-output examples, our method performs logical inference to rule out *many other* Datalog programs from the search space. In particular, our method leverages a semantics-preserving transformation as well as a new concept called *minimal distinguishing projection* (MDP) to generalize from one incorrect Datalog program to many others.

Results. We have implemented our proposed technique in a prototype called Dynamite and evaluate it on 28 data migration tasks between real-world data-sets. These tasks involve transformations between different types of source and target schemas, including relational, document, and graph databases. Our experimental results show that Dynamite can successfully automate all of these tasks using small input-output examples that consist of just a few records. Furthermore, our method performs synthesis quite fast (with an average of 7.3 seconds per benchmark) and can be used to migrate real-world database instances to the target schema in an average of 12.7 minutes per database.

Contributions. The contributions of this paper include:

- a formulation of the automated data migration problem in terms of Datalog program synthesis;
- a new algorithm for synthesizing Datalog programs;
- an implementation of our technique in a tool called Dynamite and experimental evaluation on 28 data exchange tasks between real-world data-sets.

2. OVERVIEW

In this section, we give a high-level overview of our method using a simple motivating example. Specifically, consider a document database with the following schema:

```
Univ: [{ id: Int, name: String,
         Admit: [{uid: Int, count: Int}] }]
```

This database stores a list of universities, where each university has its own id, name, and graduate school admission information. Specifically, the admission information consists of a university identifier and the number of undergraduate students admitted from that university.

Now, suppose that we need to transform this data to the following alternative schema:

```
Admission:[{grad: String, ug: String, num: Int}]
```

This new schema stores admission information as tuples consisting of a graduate school `grad`, an undergraduate school `ug`, and an integer `num` that indicates the number of undergraduates from `ug` that went to graduate school at `grad`. As an example, Figure 2(a) shows a small subset of the data in the source schema, and 2(b) shows its corresponding representation in the target schema.

For this example, the desired transformation from the source to the target schema can be represented using the following simple Datalog program:

$$\begin{aligned}&\mathit{Admission}(\mathit{grad}, \mathit{ug}, \mathit{num}) \texttt{ :-}\\ &\quad \mathit{Univ}(\mathit{id}_1, \mathit{grad}, v_1), \mathit{Admit}(v_1, \mathit{id}_2, \mathit{num}), \mathit{Univ}(\mathit{id}_2, \mathit{ug}, \_).\end{aligned}$$

```
Univ: [
 {id:1, name:"U1",
  Admit: [
   {uid:1, count:10},
   {uid:2, count:50}]},
 {id:2, name:"U2",
  Admit: [
   {uid:2, count:20},
   {uid:1, count:40}]}
]
```

(a) Input Documents

```
Admission: [
 {grad:"U1",
  ug:"U1", num:10},
 {grad:"U1",
  ug:"U2", num:50},
 {grad:"U2",
  ug:"U2", num:20},
 {grad:"U2",
  ug:"U1", num:40}
]
```

(b) Output Documents

Figure 2: Example database instances.

Here, the relation *Univ* corresponds to a university entity in the source schema, and the relation *Admit* denotes its nested *Admit* attribute. In the body of the Datalog rule, the third argument of the first *Univ* occurrence has the same first argument as *Admit*; this indicates that $(\mathit{id}_2, \mathit{num})$ is nested inside the university entry $(\mathit{id}_1, \mathit{grad})$. Essentially, this Datalog rule says the following: "If there exists a pair of universities with identifiers $\mathit{id}_1, \mathit{id}_2$ and names *grad*, *ug* in the source document, and if $(\mathit{id}_2, \mathit{num})$ is a nested attribute of id_1, then there should be an *Admission* entry $(\mathit{grad}, \mathit{ug}, \mathit{num})$ in the target database."

In what follows, we explain how Dynamite synthesizes the above Datalog program given just the source and target schemas and the input-output example from Figure 2.

Attribute Mapping. As mentioned in Section 1, our approach starts by inferring an *attribute mapping* Ψ, which specifies which attribute in the source schema *may* correspond to which other attributes (either in the source or target). For instance, based on the example provided in Figure 2, Dynamite infers the following attribute mapping Ψ:

$$\begin{array}{lcl@{\qquad}lcl} \mathit{id} & \rightarrow & \{\mathit{uid}\} & \mathit{name} & \rightarrow & \{\mathit{grad}, \mathit{ug}\} \\ \mathit{uid} & \rightarrow & \{\mathit{id}\} & \mathit{count} & \rightarrow & \{\mathit{num}\} \end{array}$$

Since the values stored in the *name* attribute of *Univ* in the source schema are the same as the values stored in the *grad* and *ug* attributes of the target schema, Ψ maps source attribute *name* to *both* target attributes *grad* and *ug*. Observe that our inferred attribute mapping can also map source attributes to other source attributes. For example, since the values in the *id* field of *Univ* are the same as the values stored in the nested *uid* attribute, Ψ also maps *id* to *uid* and vice versa.

Sketch Generation. In the next step, Dynamite uses the inferred attribute mapping Ψ to generate a program sketch Ω that defines the search space over all possible Datalog programs that we need to consider. Towards this goal, we introduce an extensional (resp. intensional) relation for each document in the source (resp. target) schema, including relations for nested documents. In this case, there is a single intensional relation *Admission* for the target schema; thus, we introduce the following single Datalog rule sketch with the *Admission* relation as its head:

$$\begin{aligned}&\mathit{Admission}(\mathit{grad}, \mathit{ug}, \mathit{num}) \texttt{ :-}\\ &\quad \mathit{Univ}(\mathbf{??}_1, \mathbf{??}_2, v_1), \mathit{Admit}(v_1, \mathbf{??}_3, \mathbf{??}_4),\\ &\quad \mathit{Univ}(\mathbf{??}_5, \mathbf{??}_6, \_), \mathit{Univ}(\mathbf{??}_7, \mathbf{??}_8, \_).\end{aligned} \tag{1}$$

$$\mathbf{??}_1, \mathbf{??}_3, \mathbf{??}_5, \mathbf{??}_7 \in \{\mathit{id}_1, \mathit{id}_2, \mathit{id}_3, \mathit{uid}_1\} \quad \mathbf{??}_4 \in \{\mathit{num}, \mathit{count}_1\}$$
$$\mathbf{??}_2, \mathbf{??}_6, \mathbf{??}_8 \in \{\mathit{grad}, \mathit{ug}, \mathit{name}_1, \mathit{name}_2, \mathit{name}_3\}$$

Here, $\mathbf{??}_i$ represents a *hole* (i.e., unknown) in the sketch, and its domain is indicated as $\mathbf{??}_i \in \{e_1, \ldots, e_n\}$, meaning that hole $\mathbf{??}_i$ can be instantiated with an element drawn

grad	ug	num
U1	U1	10
U2	U2	20

(a) Actual Result

grad	ug	num
U1	U1	10
U1	U2	50
U2	U2	20
U2	U1	40

(b) Expected Result

Figure 3: Actual and expected results of program $\mathcal{P}$.

from $\{e_1, \ldots, e_n\}$. To see where this sketch is coming from, we make the following observations:

- According to Ψ, the *grad* attribute in the target schema comes from the *name* attribute of *Univ*; thus, we must have an occurrence of *Univ* in the rule body.
- Similarly, the *ug* attribute in the target schema comes from the *name* attribute of *Univ* in the source; thus, we may need another occurrence of *Univ* in the body.
- Since the *num* attribute comes from the *count* attribute in the nested *Admit* document, the body of the Datalog rule contains $Univ(\mathbf{??}_1, \mathbf{??}_2, v_1), Admit(v_1, \mathbf{??}_3, \mathbf{??}_4)$ denoting an *Admit* document stored inside *some Univ* entity (the nesting relation is indicated through variable v_1).
- The domain of each hole is determined by Ψ and the number of occurrences of each relation in the Datalog sketch. For example, since there are three occurrences of *Univ*, we have three variables id_1, id_2, id_3 associated with the *id* attribute of *Univ*. The domain of hole $\mathbf{??}_1$ is given by $\{id_1, id_2, id_3, uid_1\}$ because it refers to the *id* attribute of *Univ*, and *id* may be an "alias" of *uid* according to Ψ.

Sketch Completion. While the Datalog program sketch Ω given above looks quite simple, it actually has $64,000$ possible completions; thus, a brute-force enumeration strategy is intractable. To solve this problem, Dynamite utilizes a novel sketch completion algorithm that aims to learn from failed synthesis attempts. Towards this goal, we encode all possible completions of sketch Ω as a satisfiability-modulo-theory (SMT) constraint Φ where each model of Φ corresponds to a possible completion of Ω. For the sketch from Equation 1, our SMT encoding is the following formula Φ:

$$\begin{aligned}&(x_1 = id_1 \lor x_1 = id_2 \lor x_1 = id_3 \lor x_1 = uid_1)\\ \land\ &(x_2 = grad \lor x_2 = ug \lor x_2 = name_1 \lor \ldots \lor x_2 = name_3)\\ \land\ &\ldots \quad \land (x_8 = grad \lor x_8 = ug \lor \ldots \lor x_8 = name_3)\end{aligned}$$

Here, for each hole $\mathbf{??}_i$ in the sketch, we introduce a variable x_i and stipulate that x_i must be instantiated with exactly one of the elements in its domain.[1] Furthermore, since Datalog requires all variables in the head to occur in the rule body, we also conjoin the following constraint with Φ to enforce this requirement:

$$(x_2 = grad \lor x_6 = grad) \land (x_2 = ug \lor x_6 = ug) \land (x_4 = num)$$

Next, we query the SMT solver for a model of this formula. In this case, one possible model σ of Φ is:

$$\begin{aligned}&x_1 = id_1 \land x_2 = grad \land x_3 = id_1 \land x_4 = num\\ \land\quad &x_5 = id_1 \land x_6 = ug \land x_7 = id_2 \land x_8 = name_1\end{aligned} \tag{2}$$

which corresponds to the following Datalog program $\mathcal{P}$:

$$\begin{aligned}Admission(grad, ug, num)\ \texttt{:-}\quad &Univ(id_1, grad, v_1),\\ Admit(v_1, id_1, num),\quad &Univ(id_1, ug, \_), Univ(id_2, name_1, \_).\end{aligned}$$

However, this program does not satisfy the user-provided example because evaluating it on the input yields a result that is different from the expected one (see Figure 3).

[1]In the SMT encoding, one should think of id_1, id_2 etc. as constants rather than variables.

Now, in the next iteration, we want the SMT solver to return a model that corresponds to a different Datalog program. Towards this goal, one possibility would be to conjoin the negation of σ with our SMT encoding, but this would rule out just a *single* program in our search space. To make synthesis more tractable, we instead analyze the root cause of failure and try to infer other Datalog programs that also do not satisfy the examples.

To achieve this goal, our sketch completion algorithm leverages two key insights: First, given a Datalog program $\mathcal{P}$, we can obtain a *set* of semantically equivalent Datalog programs by renaming variables in an equality-preserving way. Second, since our goal is to rule out incorrect (rather than just semantically equivalent) programs, we can further enlarge the set of rejected Datalog programs by performing root-cause analysis. Specifically, we express the root cause of incorrectness as a *minimal distinguishing projection (MDP)*, which is a minimal set of attributes that distinguishes the expected output from the actual output. For instance, consider the actual and expected outputs $\mathcal{O}$ and $\mathcal{O}'$ shown in Figure 3. An MDP for this example is the singleton *num* because taking the projection of $\mathcal{O}$ and $\mathcal{O}'$ on *num* yields different results.

Using these two key insights, our sketch completion algorithm infers 720 other Datalog programs that are guaranteed *not* to satisfy the input-output example and represents them using the following SMT formula:

$$\begin{aligned}(&x_4 = num \land x_1 \neq x_2 \land x_1 = x_3 \land x_1 \neq x_4 \land x_1 = x_5\\ &\land x_1 \neq x_6 \land x_1 \neq x_7 \land x_1 \neq x_8 \land \cdots \land x_7 \neq x_8)\end{aligned} \tag{3}$$

We can use the negation of this formula as a "blocking clause" by conjoining it with the SMT encoding and rule out many infeasible solutions at the same time.

After repeatedly sampling models of the sketch encoding and adding blocking clauses as discussed above, Dynamite finally obtains the following model:

$$\begin{aligned}&x_1 = id_1 \land x_2 = grad \land x_3 = id_2 \land x_4 = num\\ &\land x_5 = id_2 \land x_6 = ug \land x_7 = id_3 \land x_8 = name_1\end{aligned}$$

which corresponds to the following Datalog program (after some basic simplification):

$$\begin{aligned}&Admission(grad, ug, num)\ \texttt{:-}\\ &\quad Univ(id_1, grad, v_1), Admit(v_1, id_2, num), Univ(id_2, ug, \_).\end{aligned}$$

This program is consistent with the provided examples and can automate the desired data migration task.

3. PRELIMINARIES

In this section, we review some preliminary information on Datalog and our schema representation; then, we explain how to represent data migration programs in Datalog.

3.1 Schema Representation

We represent database schemas using non-recursive record types, which are general enough to express a wide variety of database schemas, including XML and JSON documents and graph databases. Specifically, a schema $\mathcal{S}$ is a mapping from type names N to their definition:

$$\begin{aligned}\textit{Schema } \mathcal{S} &::= N \to T\\ \textit{Type } T &::= \tau \mid \{N_1, \ldots, N_n\}\end{aligned}$$

A type definition is either a primitive type τ or a set of named attributes $\{N_1, \ldots N_k\}$, and the type of attribute N_i is given by the schema $\mathcal{S}$. An attribute N is a *primitive attribute* if $\mathcal{S}(N) = \tau$ for some primitive type τ. Given a schema $\mathcal{S}$, we write $\mathit{PrimAttrbs}(\mathcal{S})$ to denote all primitive attributes in $\mathcal{S}$, and we write $\mathit{parent}(N) = N'$ if $N \in \mathcal{S}(N')$.

EXAMPLE 1. *Consider the JSON document schema from our motivating example in Section 2:*

```
Univ: [{ id: Int, name: String,
         Admit: [{uid: Int, count: Int}] }]
```

In our representation, this schema is represented as follows:

$$\mathcal{S}(\text{Univ}) = \{\texttt{id}, \texttt{name}, \texttt{Admit}\} \quad \mathcal{S}(\text{Admit}) = \{\texttt{uid}, \texttt{count}\}$$
$$\mathcal{S}(\text{id}) = \mathcal{S}(\text{uid}) = \mathcal{S}(\text{count}) = \texttt{Int} \quad \mathcal{S}(\text{name}) = \texttt{String}$$

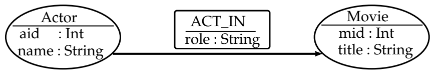
EXAMPLE 2. *Consider the following relational schema:*

$$\texttt{User}(\texttt{id} : \texttt{Int},\ \texttt{name} : \texttt{String},\ \texttt{address} : \texttt{String})$$

In our representation, this schema is represented as follows:

$$\mathcal{S}(\text{User}) = \{\texttt{id}, \texttt{name}, \texttt{address}\}$$
$$\mathcal{S}(\text{id}) = \texttt{Int} \quad \mathcal{S}(\text{name}) = \mathcal{S}(\text{address}) = \texttt{String}$$

EXAMPLE 3. *Consider the following graph schema:*



To convert this schema to our representation, we first introduce two attributes `source` *and* `target` *to denote the source and target nodes of the edge. Then, the graph schema corresponds the following mapping in our representation:*

$$\mathcal{S}(\texttt{Movie}) = \{\texttt{mid}, \texttt{title}\} \quad \mathcal{S}(\texttt{Actor}) = \{\texttt{aid}, \texttt{name}\}$$
$$\mathcal{S}(\texttt{ACT\_IN}) = \{\texttt{source}, \texttt{target}, \texttt{role}\}$$
$$\mathcal{S}(\texttt{mid}) = \mathcal{S}(\texttt{aid}) = \mathcal{S}(\texttt{source}) = \mathcal{S}(\texttt{target}) = \texttt{Int}$$
$$\mathcal{S}(\texttt{title}) = \mathcal{S}(\texttt{name}) = \mathcal{S}(\texttt{role}) = \texttt{String}$$

3.2 Datalog

As shown in Figure 4, a Datalog program consists of a list of rules, where each rule is of the form H :- B. Here, H is referred as the *head* of the rule and B is the *body*. The head H is a single relation of the form $R(v_1, \ldots, v_n)$, and the body B is a collection of predicates $B_1, B_2, \ldots, B_n$. In the remainder of this paper, we sometimes also write

$$H_1, \ldots, H_m \texttt{ :- } B_1, B_2, \ldots, B_n.$$

as short-hand for m Datalog rules with the same body. Predicates that appear only in the body are known as extensional relations and correspond to known facts. Predicates that appear in the head are called intensional relations and correspond to the output of the Datalog program.

Semantics. The semantics of Datalog programs are typically given using Herbrand models of first-order logic formulas [13]. In particular, each Datalog rule $\mathcal{R}$ of the form $H(\vec{x})$:- $B(\vec{x}, \vec{y})$ corresponds to a first-order formula $[\![\mathcal{R}]\!] = \forall \vec{x}, \vec{y}.\ B(\vec{x}, \vec{y}) \rightarrow H(\vec{x})$, and the semantics of the Datalog program can be expressed as the conjunction of each rule-level formula. Then, given a Datalog program $\mathcal{P}$ and an input $\mathcal{I}$ (i.e., a set of ground formulas), the output corresponds to the least Herbrand model of $[\![\mathcal{P}]\!] \wedge \mathcal{I}$.

3.3 Data Migration using Datalog

We now discuss how to perform data migration using Datalog. The basic idea is as follows: First, given a source database instance $\mathcal{D}$ over schema $\mathcal{S}$, we express $\mathcal{D}$ as a collection of Datalog facts over extensional relations $\mathcal{R}$. Then, we express the target schema $\mathcal{S}'$ using intensional relations

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{Program} & ::= & \mathit{Rule}^+ & \mathit{Rule} & ::= & \mathit{Head} \texttt{ :- } \mathit{Body}. \\
\mathit{Head} & ::= & \mathit{Pred} & \mathit{Body} & ::= & \mathit{Pred}^+ \\
\mathit{Pred} & ::= & R(v^+) & v \in \mathit{Variable} & & R \in \mathit{Relation}
\end{array}$$

Figure 4: Syntax of Datalog programs.

$\mathcal{R}'$ and construct a set of (non-recursive) Datalog rules, one for each intensional relation in $\mathcal{R}'$. Finally, we run this Datalog program and translate the resulting facts into the target database instance. Since programs can be evaluated using an off-the-shelf Datalog solver, we only explain how to translate between database instances and Datalog facts.

From instances to facts. Given a database instance $\mathcal{D}$ over schema $\mathcal{S}$, we introduce an extensional relation symbol R_N for each record type with name N in $\mathcal{S}$ and assign a unique identifier $\mathit{Id}(r)$ to every record r in the database instance. Then, for each instance $r = \{a_1 : v_1, \ldots, a_n : v_n\}$ of record type N, we generate a fact $R_N(c_0, c_1, \ldots, c_n)$ where:

$$c_i = \begin{cases} \mathit{Id}(\mathit{parent}(r)), & \text{if } i = 0 \text{ and r is a nested record} \\ v_i, & \text{if } \mathcal{S}(a_i) \text{ is a primitive type} \\ \mathit{Id}(r), & \text{if } \mathcal{S}(a_i) \text{ is a record type} \end{cases}$$

Intuitively, relation R_N has an extra argument that keeps track of its parent record in the database instance if N is nested in another record type. In this case, the first argument of R_N denotes the unique identifier for the record in which it is nested.

EXAMPLE 4. *For the JSON document from Figure 2(a), our method generates the following Datalog facts*

$$\mathit{Univ}(1, \text{“}\texttt{U1}\text{”}, \mathit{id}_1) \quad \mathit{Univ}(2, \text{“}\texttt{U2}\text{”}, \mathit{id}_2) \quad \mathit{Admit}(\mathit{id}_1, 1, 10)$$
$$\mathit{Admit}(\mathit{id}_2, 2, 20) \quad \mathit{Admit}(\mathit{id}_1, 2, 50) \quad \mathit{Admit}(\mathit{id}_2, 1, 40)$$

where id_1 and id_2 are unique identifiers.

From facts to instances. We convert Datalog facts to the target database instance using the inverse procedure. Specifically, given a Datalog fact $R_N(c_1, \ldots, c_n)$ for record type $N : \{a_1, \ldots, a_n\}$, we create a record instance using a function $\mathit{BuildRecord}(R_N, N) = \{a_1 : v_1, \ldots, a_n : v_n\}$ where

$$v_i = \begin{cases} c_i, & \text{if } \mathcal{S}(a_i) \text{ is a primitive type} \\ \mathit{BuildRecord}(R_{a_i}, a_i), & \text{if } \mathcal{S}(a_i) \text{ is a record type and} \\ & \text{the first argument of } R_{a_i} \text{is } c_i \end{cases}$$

Observe that the *BuildRecord* procedure builds the record recursively by chasing parent identifiers into other relations.

4. DATALOG PROGRAM SYNTHESIS

In this section, we describe our algorithm for automatically synthesizing Datalog programs from an input-output example $\mathcal{E} = (\mathcal{I}, \mathcal{O})$. Here, $\mathcal{I}$ corresponds to an example of the database instance in the source schema, and $\mathcal{O}$ demonstrates the desired target instance. We start by giving a high-level overview of the synthesis algorithm and then explain each of the key ingredients in more detail.

4.1 Algorithm Overview

The top-level algorithm for synthesizing Datalog programs is summarized in Algorithm 1. The SYNTHESIZE procedure takes as input a source schema $\mathcal{S}$, a target schema $\mathcal{S}'$, and an input-output example $\mathcal{E} = (\mathcal{I}, \mathcal{O})$. The return value is either a Datalog program $\mathcal{P}$ such that evaluating $\mathcal{P}$ on $\mathcal{I}$ yields $\mathcal{O}$ (i.e. $[\![\mathcal{P}]\!]_{\mathcal{I}} = \mathcal{O}$) or $\bot$ to indicate that the desired data migration task cannot be represented as a Datalog program.

Algorithm 1 Synthesizing Datalog programs

```
1: procedure SYNTHESIZE($\mathcal{S}, \mathcal{S}', \mathcal{E}$)
   Input: Source schema $\mathcal{S}$, target schema $\mathcal{S}'$,
          example $\mathcal{E} = (\mathcal{I}, \mathcal{O})$
   Output: Datalog program $\mathcal{P}$ or $\bot$ to indicate failure
2:     $\Psi \leftarrow$ INFERATTRMAPPING($\mathcal{S}, \mathcal{S}', \mathcal{E}$);
3:     $\Omega \leftarrow$ SKETCHGEN($\Psi, \mathcal{S}, \mathcal{S}'$);
4:     $\Phi \leftarrow$ ENCODE($\Omega$);
5:     while SAT($\Phi$) do
6:         $\sigma \leftarrow$ GetModel($\Phi$);
7:         $\mathcal{P} \leftarrow$ Instantiate($\Omega, \sigma$);
8:         $\mathcal{O}' \leftarrow [\![\mathcal{P}]\!]_{\mathcal{I}}$;
9:         if $\mathcal{O}' = \mathcal{O}$ then return $\mathcal{P}$;
10:        $\Phi \leftarrow \Phi \wedge$ ANALYZE($\sigma, \mathcal{O}', \mathcal{O}$);
11:    return $\bot$;
```

As shown in Algorithm 1, the SYNTHESIZE procedure first invokes the INFERATTRMAPPING procedure (line 2) to infer an attribute mapping Ψ. Specifically, Ψ is a mapping from each $a \in PrimAttrbs(\mathcal{S})$ to a set of attributes $\{a_1, \ldots, a_n\}$ where $a_i \in PrimAttrbs(\mathcal{S}) \cup PrimAttrbs(\mathcal{S}')$ such that:

$$a' \in \Psi(a) \;\Leftrightarrow\; \Pi_{a'}(\mathcal{D}) \subseteq \Pi_a(\mathcal{I})$$

where $\mathcal{D}$ stands for either $\mathcal{I}$ or $\mathcal{O}$. Thus, INFERATTRMAPPING is conservative and maps a source attribute a to another attribute a' if the values contained in a' are a subset of those contained in a.

Next, the algorithm invokes SKETCHGEN (line 3) to generate a Datalog program sketch Ω based on Ψ. As mentioned in Section 2, a sketch Ω is a Datalog program with unknown arguments in the rule body, and the sketch also determines the domain for each unknown. Thus, if the sketch contains n unknowns, each with k elements in its domain, then the sketch encodes a search space of k^n possible programs.

Lines 4-10 of the SYNTHESIZE algorithm perform lazy enumeration over possible sketch completions. Given a sketch Ω, we first generate an SMT formula Φ whose models correspond to all possible completions of Ω (line 4). Then, the loop in lines 5-10 repeatedly queries a model of Φ (line 6), tests if the corresponding Datalog program is consistent with the example (lines 7-9), and adds a blocking clause to Φ if it is not (line 10). The blocking clause is obtained via the call to the ANALYZE procedure, which performs Datalog-specific deductive reasoning to infer a whole set of programs that are *guaranteed not to* satisfy the examples.

In the remainder of this section, we explain the sketch generation and completion procedures in more detail.

4.2 Sketch Generation

Given an attribute mapping Ψ, the goal of sketch generation is to construct the skeleton of the target Datalog program. Our sketch language is similar to the Datalog syntax in Figure 4, except that it allows holes (denoted by **??**) as special constructs indicating unknown expressions. As summarized in Algorithm 2, the SKETCHGEN procedure iterates over each top-level record in the target schema and, for each record type, it generates a Datalog rule sketch using the helper procedure GENRULESKETCH. [2] Conceptually, GENRULESKETCH performs the following tasks: First, it generates a set of intensional predicates for each top-level record in the target schema (line 8). The intensional predicates do not contain any unknowns and only appear in the head of the Datalog rules. Next, the loop (lines 9–12) constructs the skeleton of each Datalog rule body by generating extensional predicates for the relevant source record types. The extensional predicates do contain unknowns, and there can be multiple occurrences of a relation symbol in the body. Finally, the loop in lines 13–18 generates the domain for each unknown used in the rule body.

[2]The property of the generated sketch is characterized and proved in Appendix A of the extend version [47].

Algorithm 2 Generating Datalog program sketches

```
1: procedure SKETCHGEN($\Psi, \mathcal{S}, \mathcal{S}'$)
   Input: Attribute mapping $\Psi$, source schema $\mathcal{S}$,
          target schema $\mathcal{S}'$
   Output: Program sketch $\Omega$
2:     $\Omega \leftarrow \emptyset$;
3:     for each top-level record type $N \in \mathcal{S}'$ do
4:         $R \leftarrow$ GENRULESKETCH($\Psi, \mathcal{S}, \mathcal{S}', N$);
5:         $\Omega \leftarrow \Omega \cup \{R\}$;
6:     return $\Omega$;

7: procedure GENRULESKETCH($\Psi, \mathcal{S}, \mathcal{S}', N$)
8:     $H \leftarrow$ GENINTENSIONALPREDS($\mathcal{S}', N$); $B \leftarrow \emptyset$;
9:     for each $a \in dom(\Psi)$ do
10:        repeat $|\{a' \mid a' \in PrimAttrbs(N) \wedge a' \in \Psi(a)\}|$ times
11:            $N \leftarrow$ RecName($a$);
12:            $B \leftarrow B \cup$ GENEXTENSIONALPREDS($\mathcal{S}, N$);
13:    for each $??_a \in Holes(B)$ do
14:        $V \leftarrow \{v_{a'} \mid a' \in PrimAttrbs(N) \wedge a' \in \Psi(a)\}$;
15:        for each $a' \in \Psi(a) \cup \{a\}$ and $a' \in PrimAttrbs(\mathcal{S})$ do
16:            $n \leftarrow$ CopyNum($B$, RecName($a'$));
17:            $V \leftarrow V \cup \bigcup_{i=1}^{n} \{v_{a'}^i\}$;
18:        $B \leftarrow B[??_a \mapsto ??_a \in V]$;
19:    return $H$ :- $B$.;
```

$$\frac{\mathcal{S}'(N) \in PrimType}{\mathcal{S}' \vdash N \rightsquigarrow (v_N,\ \emptyset)}\ \text{(InPrim)}$$

$$\frac{\mathcal{S}'(N) = \{a_1, \ldots, a_n\} \quad isNested(N) \qquad \mathcal{S}' \vdash a_i \rightsquigarrow (v_i, H_i) \quad i = 1, \ldots, n}{\mathcal{S}' \vdash N \rightsquigarrow (v_N,\ \{R_N(v_N, v_1, \ldots, v_n)\} \cup \bigcup_{i=1}^{n} H_i)}\ \text{(InRecNested)}$$

$$\frac{\mathcal{S}'(N) = \{a_1, \ldots, a_n\} \quad \neg isNested(N) \qquad \mathcal{S}' \vdash a_i \rightsquigarrow (v_i, H_i) \quad i = 1, \ldots, n}{\mathcal{S}' \vdash N \rightsquigarrow (\_,\ \{R_N(v_1, \ldots, v_n)\} \cup \bigcup_{i=1}^{n} H_i)}\ \text{(InRec)}$$

Figure 5: Inference rules describing GENINTENSIONALPREDS

Head generation. Given a top-level record type N in the target schema, the procedure GENINTENSIONALPREDS generates the head of the corresponding Datalog rule for N. If N does not contain any nested records, then the head consists of a single predicate, but, in general, the head contains as many predicates as are (transitively) nested in N.

In more detail, Figure 5 presents the GENINTENSIONALPREDS procedure as inference rules that derive judgments of the form $\mathcal{S}' \vdash N \rightsquigarrow (v, H)$ where H corresponds to the head of the Datalog rule for record type N. As expected, these rules are recursive and build the predicate set H for N from those of its nested records. Specifically, given a top-level record N with attributes $a_1, \ldots, a_n$, the rule InRec first generates predicates H_i for each attribute a_i and then introduces an additional relation $R_N(v_1, \ldots, v_n)$ for N itself. Predicate generation for nested relations (rule InRec-

$$\frac{\mathcal{S}(N) \in \mathit{PrimType}}{\mathcal{S} \vdash N \hookrightarrow (\mathbf{??}_N,\ \_)}\ \text{(ExPrim)}$$

$$\frac{\begin{array}{c}\mathcal{S}(N) = \{a_1, \ldots, a_n\} \quad \text{fresh } v_N \\ \mathcal{S} \vdash a_i \hookrightarrow (h_i, \_) \quad i = 1, \ldots, n \\ N' = \mathit{parent}(N) \quad \mathcal{S} \vdash N' \hookrightarrow (\_, B')\end{array}}{\mathcal{S} \vdash N \hookrightarrow (v_N,\ \{R_N(v_N, h_1, \ldots, h_n)\} \cup B')}\ \text{(ExRecNested)}$$

$$\frac{\begin{array}{c}\mathcal{S}(N) = \{a_1, \ldots, a_n\} \quad \neg \mathit{isNested}(N) \\ \mathcal{S} \vdash a_i \hookrightarrow (h_i, \_) \quad i = 1, \ldots, n\end{array}}{\mathcal{S} \vdash N \hookrightarrow (\_,\ \{R_N(h_1, \ldots, h_n)\})}\ \text{(ExRec)}$$

Figure 6: Inference rules for GenExtensionalPreds

Nested) is similar, but we introduce a new variable v_N that is used for connecting N to its parent relation. The InPrim rule corresponds to the base case of GenIntensionalPreds and generates variables for attributes of primitive type.

Body sketch generation. We now consider sketch generation for the body of each Datalog rule (lines 9–12 in Algorithm 2). Given a record type N in the source schema and its corresponding predicate(s) R_N, the loop in lines 9–12 of Algorithm 2 generates as many copies of R_N in the rule body as there are head attributes that "come from" R_N according to Ψ. Specifically, Algorithm 2 invokes a procedure called GenExtensionalPreds, described in Figure 6, to generate each copy of the extensional predicate symbol.

Given a record type N in the source schema, GenExtensionalPreds generates predicates up until the top-level record that contains N. The rules in Figure 6 are of the form $\mathcal{S} \vdash N \hookrightarrow (h, B)$, where B is the sketch body for record type N. The ExPrim rule is the base case to generate sketch holes for primitive attributes. Given a record N with attributes $a_1, \ldots, a_n$ and its parent N', the rule ExRecNested recursively generates the body predicates B' for the parent record N' and adds an additional predicate $R_N(v_N, h_1, \ldots, h_n)$ for N itself. Here v_N is a variable for connecting N and its parent N', and h_i is the hole or variable for attribute a_i. In the case where N is a top level record, the ExRec rule generates a singleton predicate $R_N(h_1, \ldots, h_n)$.

Example 5. *Suppose we want to generate the body sketch for the rule associated with record type T : $\{a'$: `Int`, `b'` : `Int`$\}$ in the target schema. Also, suppose we are given the attribute mapping Ψ where $\Psi(a) = a'$ and $\Psi(b) = b'$ and source attributes a, b belong to the following record type in the source schema: C : $\{a$: `Int`, `D` : $\{$`b` : `Int`$\}\}$. According to Ψ, a' comes from attribute a of record type C in the source schema, so we have a copy of R_C in the sketch body. Based on the rules of Figure 6, we generate predicate $R_C(\mathbf{??}_a, v_D^1)$, where $\mathbf{??}_a$ is the hole for attribute a and v_D^1 is a fresh variable. Similarly, since b' comes from attribute b of record type D, we generate predicates $R_C(\mathbf{??}_a, v_D^2)$ and $R_D(v_D^2, \mathbf{??}_b)$. Putting them together, we obtain the following sketch body:*

$$R_C(\mathbf{??}_a, v_D^1), R_C(\mathbf{??}_a, v_D^2), R_D(v_D^2, \mathbf{??}_b)$$

Domain generation. Having constructed the skeleton of the Datalog program, we still need to determine the set of variables that each hole in the sketch can be instantiated with. Towards this goal, the last part of GenRuleSketch (lines 13–18 in Algorithm 2) constructs the domain V for each hole as follows: First, for each attribute a of source relation R_N, we introduce as many variables $v_a^1, \ldots, v_a^k$ as there are copies of R_N. Next, for the purposes of this discussion, let us say that attributes a and b "alias" each other if $b \in \Psi(a)$ or vice versa. Then, given a hole $\mathbf{??}_x$ associated with attribute x, the domain of $\mathbf{??}_x$ consists of all the variables associated with attribute x or one of its aliases.

Example 6. *Consider the same schemas and attribute mapping from Example 5 and the following body sketch:*

$$R_C(\mathbf{??}_a, v_D^1), R_C(\mathbf{??}_a, v_D^2), R_D(v_D^2, \mathbf{??}_b)$$

Here, we have $\mathbf{??}_a \in \{v_{a'}, v_a^1, v_a^2\}$ and $\mathbf{??}_b \in \{v_{b'}, v_b^1\}$.

4.3 Sketch Completion

While the sketch generated by Algorithm 2 defines a finite search space of Datalog programs, this search space is still exponentially large. Thus, rather than performing naive brute-force enumeration, our sketch completion algorithm combines enumerative search with Datalog-specific deductive reasoning to learn from failed synthesis attempts. As explained in Section 4.1, the basic idea is to generate an SMT encoding of all possible sketch completions and then iteratively add blocking clauses to rule out incorrect Datalog programs. In the remainder of this section, we discuss how to generate the initial SMT encoding as well as the Analyze procedure for generating useful blocking clauses.

Sketch encoding. Given a Datalog program sketch Ω, our initial SMT encoding is constructed as follows: First, for each hole $\mathbf{??}_i$ in the sketch, we introduce an integer variable x_i, and for every variable v_j in the domain of some hole, we introduce a unique integer constant denoted as $\mathit{Const}(v_j)$. Then, our SMT encoding stipulates the following constraints to enforce that the sketch completion is well-formed:

- ***Every hole must be instantiated:*** For each hole of the form $\mathbf{??}_i \in \{v_1, \ldots, v_n\}$, we add a constraint

$$\bigvee_{j=1}^{n} x_i = \mathit{Const}(v_j)$$

- ***Head variables must appear in the body.*** In a well-formed Datalog program, every head variable must appear in the body. Thus, for each head variable v, we add:

$$\bigvee_i x_i = \mathit{Const}(v) \text{ where } v \text{ is in the domain of } \mathbf{??}_i$$

Since there is a one-to-one mapping between integer constants in the SMT encoding and sketch variables, each model of the SMT formula corresponds to a Datalog program.

Adding blocking clauses. Given a Datalog program $\mathcal{P}$ that does not satisfy the examples $(\mathcal{I}, \mathcal{O})$, our top-level synthesis procedure (Algorithm 1) invokes a function called Analyze to find useful blocking clauses to add to the SMT encoding. This procedure is summarized in Algorithm 3 and is built on two key insights. The first key insight is that the semantics of a Datalog program is unchanged under an equality-preserving renaming of variables:

Theorem 1. *Let $\mathcal{P}$ be a Datalog program over variables X and let $\hat{\sigma}$ be an injective substitution from X to another set of variables Y. Then, we have $\mathcal{P} \simeq \mathcal{P}\hat{\sigma}$.*

Proof. See Appendix A of the extended version [47]. □

To see how this theorem is useful, let σ be a model of our SMT encoding. In other words, σ is a mapping from holes in the Datalog sketch to variables V. Now, let $\hat{\sigma}$ be an injective renaming of variables in V. Then, using the above theorem, we know that any other assignment $\sigma' = \sigma\hat{\sigma}$ is also guaranteed to result in an incorrect Datalog program.

Based on this insight, we can generalize from the specific assignment σ to a more general class of incorrect assignments as follows: If a hole is not assigned to a head variable, then it can be assigned to any variable in its domain as long as it respects the equalities and disequalities in σ. Concretely, given assignment σ, we generalize it as follows:

$$\mathit{Generalize}(\sigma) = \bigwedge_{x_i \in dom(\sigma)} \alpha(x_i, \sigma), \text{ where}$$
$$\alpha(x, \sigma) = \begin{cases} x = \sigma(x) & \text{if } \sigma(x) \text{ is a head variable} \\ \bigwedge_{x_j \in dom(\sigma)} x \star x_j & \text{otherwise} \end{cases}$$

Here the binary operator $\star$ is defined to be equality if σ assigns both x and x_j to the same value, and disequality otherwise. Thus, rather than ruling out just the current assignment σ, we can instead use $\neg\mathit{Generalize}(\sigma)$ as a much more general blocking clause that rules out several equivalent Datalog programs at the same time.

Example 7. *Consider again the sketch from Section 2:*

$$\text{Admission}(grad, ug, num) \;\text{:-}\; \text{Univ}(\mathbf{??}_1, \mathbf{??}_2, v_1),$$
$$\text{Admit}(v_1, \mathbf{??}_3, \mathbf{??}_4), \quad \text{Univ}(\mathbf{??}_5, \mathbf{??}_6, \_), \quad \text{Univ}(\mathbf{??}_7, \mathbf{??}_8, \_).$$

$$\mathbf{??}_1, \mathbf{??}_3, \mathbf{??}_5, \mathbf{??}_7 \in \{id_1, id_2, id_3, uid_1\} \;\; \mathbf{??}_4 \in \{num, count_1\}$$
$$\mathbf{??}_2, \mathbf{??}_6, \mathbf{??}_8 \in \{grad, ug, name_1, name_2, name_3\}$$

Suppose the variable for $\mathbf{??}_i$ *is* x_i *and the assignment* σ *is:*

$$x_1 = \text{id}_1 \wedge x_2 = \text{grad} \wedge x_3 = \text{id}_1 \wedge x_4 = \text{num}$$
$$\wedge \quad x_5 = \text{id}_1 \wedge x_6 = \text{ug} \wedge x_7 = \text{id}_2 \wedge x_8 = \text{name}_1$$

Since grad, ug, and num occur in the head, $\mathit{Generalize}(\sigma)$ *yields the following formula:*

$$\begin{gathered} x_2 = grad \wedge x_4 = num \wedge x_6 = ug \\ \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4 \wedge x_1 = x_5 \\ \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8 \end{gathered} \tag{4}$$

The other key insight underlying our sketch completion algorithm is that we can achieve even greater generalization power using the concept of *minimal distinguishing projections (MDP)*, defined as follows:

Definition 1. **(MDP)** *We say that a set of attributes* A *is a minimal distinguishing projection for Datalog program* $\mathcal{P}$ *and input-output example* $(\mathcal{I}, \mathcal{O})$ *if (1)* $\Pi_A(\mathcal{O}) \neq \Pi_A(\mathcal{P}(\mathcal{I}))$, *and (2) for any* $A' \subset A$, *we have* $\Pi_{A'}(\mathcal{O}) = \Pi_{A'}(\mathcal{P}(\mathcal{I}))$.

In other words, the first condition ensures that, by just looking at attributes A, we can tell that program $\mathcal{P}$ does not satisfy the examples. On the other hand, the second condition ensures that A is minimal.

To see why minimal distinguishing projections are useful for pruning a larger set of programs, recall that our *Generalize*(σ) function from earlier retains a variable assignment $x \mapsto v$ if v corresponds to a head variable. However, if v does not correspond to an attribute in the MDP, then we will still obtain an incorrect program even if we rename x to something else; thus *Generalize* can drop the assignments to head variables that are not in the MDP. Thus, given an MDP φ, we can obtain an improved generalization procedure *Generalize*(σ, φ) by using the following $\alpha(x, \sigma, \varphi)$ function instead of $\alpha(x, \sigma)$ from earlier:

$$\alpha(x, \sigma, \varphi) = \begin{cases} x = \sigma(x) & \text{if } \sigma(x) \in \varphi \\ \bigwedge_{x_j \in dom(\sigma)} x \star x_j & \text{otherwise} \end{cases}$$

Because not all head variables correspond to an MDP attribute, performing generalization this way allows us to obtain a better blocking clause that rules out many more Datalog programs in one iteration.

Algorithm 3 Analyzing outputs to prune search space

```
1:  procedure Analyze($\sigma, \mathcal{O}', \mathcal{O}$)
      Input: Model $\sigma$, actual output $\mathcal{O}'$, expected output $\mathcal{O}$
      Output: Blocking clause $\phi$
2:      $\phi \leftarrow true$;
3:      $\Delta \leftarrow$ MDPSet($\mathcal{O}', \mathcal{O}$);
4:      for each $\varphi \in \Delta$ do
5:          $\psi \leftarrow true$;
6:          for each $(x_i, x_j) \in dom(\sigma) \times dom(\sigma)$ do
7:              if $\sigma(x_i) = \sigma(x_j)$ then $\psi \leftarrow \psi \wedge x_i = x_j$;
8:              else $\psi \leftarrow \psi \wedge x_i \neq x_j$;
9:          for each $x_i \in dom(\sigma)$ do
10:             if $\sigma(x_i) \in \varphi$ then $\psi \leftarrow \psi \wedge x_i = \sigma(x_i)$;
11:         $\phi \leftarrow \phi \wedge \neg\psi$;
12:     return $\phi$;
```

Example 8. *Consider the same sketch and assignment* σ *from Example 7, but now suppose we are given an MDP* $\varphi = \{num\}$. *Then the function* $\mathit{Generalize}(\sigma, \varphi)$ *yields the following more general formula:*

$$\begin{gathered} x_4 = num \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4 \wedge x_1 = x_5 \\ \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8 \end{gathered} \tag{5}$$

Note that (5) is more general (i.e., weaker) than (4) because it drops the constraints $x_2 = grad$ *and* $x_6 = ug$. *Therefore, the negation of (5) is a better blocking clause than the negation of (4), since it rules out more programs in one step.*

Based on this discussion, we now explain the full Analyze procedure in Algorithm 3. This procedure takes as input a model σ of the SMT encoding and the actual and expected outputs $\mathcal{O}', \mathcal{O}$. Then, at line 3, it invokes the MDPSet procedure to obtain a set Δ of minimal distinguishing projections and uses each MDP $\varphi \in \Delta$ to generate a blocking clause as discussed above (lines 6–10).

The MDPSet procedure is shown in Algorithm 4 and uses a breadth-first search algorithm to compute the set of all minimal distinguishing projections. Specifically, it initializes a queue $\mathcal{W}$ with singleton projections $\{a\}$ for each attribute a in the output (lines 2 – 5). Then, it repeatedly dequeues a projection L from $\mathcal{W}$ and checks if L is an MDP (lines 6 – 14). In particular, if L can distinguish outputs $\mathcal{O}'$ and $\mathcal{O}$ (line 14) and there is no existing projection L'' in the current MDP set Δ such that $L'' \subseteq L$, then L is an MDP. If L cannot distinguish outputs $\mathcal{O}'$ and $\mathcal{O}$ (line 8), we enqueue all of its extensions L' with one more attribute than L and move on to the next projection in queue $\mathcal{W}$.

Example 9. *Let us continue with Example 8 to illustrate how to prune incorrect Datalog programs using multiple MDPs. Suppose we obtain the MDP set* $\Delta = \{\varphi_1, \varphi_2\}$, *where* $\varphi_1 = \{num\}$ *and* $\varphi_2 = \{grad, ug\}$. *In addition to* $\mathit{Generalize}(\sigma, \varphi_1)$ *(see formula (5) of Example 8), we also compute* $\mathit{Generalize}(\sigma, \varphi_2)$ *as:*

$$\begin{gathered} x_2 = grad \wedge x_6 = ug \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4 \\ \wedge x_1 = x_5 \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8 \end{gathered}$$

By adding both blocking clauses $\neg\mathit{Generalize}(\sigma, \varphi_1)$ *as well as* $\neg\mathit{Generalize}(\sigma, \varphi_2)$, *we can prune even more incorrect Datalog programs.*

Theorem 2. *Let* ϕ *be a blocking clause returned by the call to* Analyze *at line 10 of Algorithm 1. If* σ *is a model of* $\neg\phi$, *then* σ *corresponds to an incorrect Datalog program.*

Proof. See Appendix A of the extended version [47]. □

Algorithm 4 Computing a set of MDPs

```
 1: procedure MDPSet($\mathcal{O}'$, $\mathcal{O}$)
    Input: Actual output $\mathcal{O}'$, expected output $\mathcal{O}$
    Output: A set of minimal distinguishing projections $\Delta$
 2:     $\Delta \leftarrow \emptyset$;  $\mathcal{V} \leftarrow \emptyset$;
 3:     $\mathcal{W} \leftarrow$ EmptyQueue();
 4:     for each $a \in$ Attributes($\mathcal{O}'$) do
 5:         $\mathcal{W}$.Enqueue($\{a\}$); $\mathcal{V} \leftarrow \mathcal{V} \cup \{\{a\}\}$;
 6:     while $\neg\mathcal{W}$.IsEmpty() do
 7:         $L \leftarrow \mathcal{W}$.Dequeue();
 8:         if $\Pi_L(\mathcal{O}') = \Pi_L(\mathcal{O})$ then
 9:             for each $a' \in$ Attributes($\mathcal{O}'$) $\setminus L$ do
10:                 $L' \leftarrow L \cup \{a'\}$;
11:                 if $L' \notin \mathcal{V}$ then
12:                     $\mathcal{W}$.Enqueue($L'$);
13:                     $\mathcal{V} \leftarrow \mathcal{V} \cup \{L'\}$;
14:         else if $\nexists L'' \in \Delta.\ L'' \subseteq L$ then $\Delta \leftarrow \Delta \cup \{L\}$;
15:     return $\Delta$;
```

5. IMPLEMENTATION

We have implemented the proposed technique as a new tool called Dynamite. Internally, Dynamite uses the Z3 solver [16] for answering SMT queries and leverages the Souffle framework [28] for evaluating Datalog programs. In the remainder of this section, we discuss some extensions over the synthesis algorithm described in Section 4.

Interactive mode. In Section 4, we presented our technique as returning a *single* program that is consistent with an input-output example. However, in this *non-interactive mode*, Dynamite does not guarantee the uniqueness of the program consistent with the given example. To address this potential usability issue, Dynamite can also be used in a so-called *interactive mode* where Dynamite iteratively queries the user for more examples in order to resolve ambiguities. Specifically, when used in this interactive mode, Dynamite first checks if there are multiple programs $\mathcal{P}, \mathcal{P}'$ that are consistent with the provided examples $(\mathcal{I}, \mathcal{O})$, and, if so, Dynamite identifies a small *differentiating input* $\mathcal{I}'$ such that $\mathcal{P}$ and $\mathcal{P}'$ yield different outputs on $\mathcal{I}'$. Then, Dynamite asks the user to provide the corresponding output for $\mathcal{I}'$. More details on how we implement this interactive mode can be found in Appendix B of the extended version [47].

Example 10. *Suppose the source database contains two relations Employee(name, deptId) and Department(id, deptName), and we want to obtain the relation WorksIn(name, deptName) by joining Employee and Department on deptId=id and then applying projection. Suppose the user only provides the input example Employee(Alice, 11), Department(11, CS) and the output WorksIn(Alice, CS). Now* Dynamite *may return one of the following results:*

(1) WorksIn(x, y) :- Employee(x, z), Department(z, y).
(2) WorksIn(x, y) :- Employee(x, z), Department(w, y).

Note that both Datalog programs are consistent with the given input-output example, but only program (1) is the transformation the user wants. Since the program returned by Dynamite *depends on the model sampled by the SMT solver, it is possible that* Dynamite *returns the incorrect solution (2) instead of the desired program (1). Using* Dynamite *in the interactive mode solves this problem. In this mode,* Dynamite *searches for an input that distinguishes the two programs shown above. In this case, such a distinguishing input is Employee(Alice, 11), Employee(Bob, 12), Department(11, CS), Department(12, EE), and* Dynamite *asks the user to provide the corresponding output. Now, if the user provides the output WorksIn(Alice, CS), WorksIn(Bob, EE), only program (1) will be consistent and* Dynamite *successfully eliminates the initial ambiguity.*

Table 1: Datasets used in the evaluation.

Name	Size	Description
Yelp	4.7GB	Business and reviews from Yelp
IMDB	6.3GB	Movie and crew info from IMDB
Mondial	3.7MB	Geography information
DBLP	2.0GB	Publication records from DBLP
MLB	0.9GB	Pitch data of Major League Baseball
Airbnb	0.4GB	Berlin Airbnb data
Patent	1.7GB	Patent Litigation Data 1963-2015
Bike	2.7GB	Bike trip data in Bay Area
Tencent	1.0GB	User followers in Tencent Weibo
Retina	0.1GB	Biological info of mouse retina
Movie	0.1GB	Movie ratings from MovieLens
Soccer	0.2GB	Transfer info of soccer players

Filtering operation. While the synthesis algorithm described in Section 4 does not support data filtering during migration, Dynamite allows the target database instance to contain a subset of the data in the source instance. However, the filtering operations supported by Dynamite are restricted to predicates that can be expressed as a conjunction of equalities. To see how Dynamite supports such filtering operations, observe that if an extensional relation R uses a constant c as the argument of attribute a_i, this is the same as filtering out tuples where the corresponding value is not c. Based on this observation, Dynamite allows program sketches where the domain of a hole can include constants in addition to variables. These constants are drawn from values in the output example, and the sketch completion algorithm performs enumerative search over these constants.

Database instance construction. Dynamite builds the target database instance from the output facts of the synthesized Datalog program as described in Section 3.3. However, Dynamite performs one optimization to make large-scale data migration practical: We leverage MongoDB [2] to build indices on attributes that connect records to their parents. This strategy allows Dynamite to quickly look up the children of a given record and makes the construction of the target database more efficient.

6. EVALUATION

To evaluate Dynamite, we perform experiments that are designed to answer the following research questions:

RQ1 Can Dynamite successfully migrate real-world data sets given a representative set of records, and how good are the synthesized programs?

RQ2 How sensitive is the synthesizer to the number and quality of examples?

RQ3 How helpful is Dynamite for users in practice, and how do users choose between multiple correct answers?

RQ4 Is the proposed sketch completion algorithm significantly more efficient than a simpler baseline?

RQ5 How does the proposed synthesis technique compare against prior techniques?

Table 2: Statistics of benchmarks. "R" stands for relational, "D" stands for document, and "G" stands for graph.

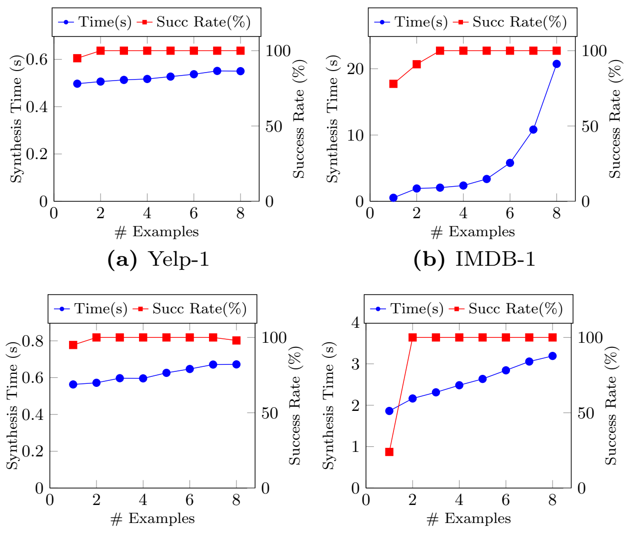
Benchmark	Source Schema			Target Schema		
	Type	#Recs	#Attrs	Type	#Recs	#Attrs
Yelp-1	D	11	58	R	8	32
IMDB-1	D	12	21	R	9	26
DBLP-1	D	37	42	R	9	35
Mondial-1	D	37	113	R	25	110
MLB-1	R	5	83	D	7	85
Airbnb-1	R	4	30	D	6	24
Patent-1	R	5	49	D	7	50
Bike-1	R	4	48	D	7	47
Tencent-1	G	2	8	R	1	3
Retina-1	G	2	17	R	2	13
Movie-1	G	5	18	R	5	21
Soccer-1	G	10	30	R	7	21
Tencent-2	G	2	8	D	1	3
Retina-2	G	2	17	D	2	15
Movie-2	G	5	18	D	4	14
Soccer-2	G	10	30	D	7	23
Yelp-2	D	11	58	G	4	31
IMDB-2	D	12	21	G	11	19
DBLP-2	D	37	42	G	17	28
Mondial-2	D	37	113	G	27	78
MLB-2	R	5	83	G	12	90
Airbnb-2	R	4	30	G	7	32
Patent-2	R	5	49	G	8	49
Bike-2	R	4	48	G	6	52
MLB-3	R	5	83	R	4	75
Airbnb-3	R	4	30	R	7	33
Patent-3	R	5	49	R	8	52
Bike-3	R	4	48	R	5	52
Average	-	**10.2**	**44.4**	-	**8.0**	**39.8**

Benchmarks. To answer these research questions, we collected 12 real-world database instances (see Table 1 for details) and created 28 benchmarks in total. Specifically, four of these datasets (namely Yelp, IMDB, Mondial, and DBLP) are taken from prior work [49], and the remaining eight are taken from open dataset websites such as Kaggle [1]. For the document-to-relational transformations, we used exactly the same benchmarks as prior work [49]. For the remaining cases (e.g., document-to-graph or graph-to-relational), we used the source schemas in the original dataset but created a suitable target schema ourselves. As summarized in Table 2, our 28 benchmarks collectively cover a broad range of migration scenarios between different types of databases.[3]

Experimental setup. All experiments are conducted on a machine with Intel Xeon(R) E5-1620 v3 quad-core CPU and 32GB of physical memory, running the Ubuntu 18.04 OS.

6.1 Synthesis Results in Non-Interactive Mode

In this section, we evaluate RQ1 by using DYNAMITE to migrate the datasets from Table 1 for the source and target schemas from Table 2. To perform this experiment, we first constructed a representative set of input-output examples for each record in the source and target schemas. As shown in Table 3, across all benchmarks, the average number of records in the input (resp. output) example is 2.6 (resp. 2.2). Given these examples, we then used DYNAMITE to synthesize a migration script consistent with the given examples and ran it on the real-world datasets from



(a) Yelp-1 **(b)** IMDB-1 **(c)** DBLP-1 **(d)** Mondial-1

Figure 7: Sensitivity analysis.

Table 1.[4] We now highlight the key take-away lessons from this experiment whose results are summarized in Table 3.

Synthesis time. Even though the search space of possible Datalog programs is very large (5.1×10^{39} on average), DYNAMITE can find a Datalog program consistent with the examples in an average of 7.3 seconds, with maximum synthesis time being 87.9 seconds.

Statistics about synthesized programs. As shown in Table 3, the average number of rules in the synthesized Datalog program is 8.0, and each rule contains an average of 2.5 predicates in the rule body (after simplification).

Quality of synthesized programs. To evaluate the quality of the synthesized programs, we compared the synthesized Datalog programs against manually written ones (which we believe to be optimal). As shown in the column labeled "# Optim Rules" in Table 3, on average, 5.8 out of the 8 Datalog rules (72.5%) are *syntactically identical* to the manually-written ones. In cases where the synthesized rule differs from the manually-written one, we observed that the synthesized program contains redundant body predicates. In particular, if we quantify the distance between the two programs in terms of additional predicates, we found that the synthesized rules contain an average of 0.79 extra predicates (shown in column labeled "Dist to Optim"). However, note that, even in cases where the synthesized rule differs syntactically from the manually-written rule, we confirmed that the synthesized and manual rules produce *the exact same output* for the given input relations in *all cases*.

Migration time and results. For all 28 benchmarks, we confirmed that DYNAMITE is able to produce the intended target database instance. As reported in the column labeled "Migration time", the average time taken by DYNAMITE to convert the source instance to the target one is 12.7 minutes for database instances containing 1.7 GB of data on average.

6.2 Sensitivity to Examples

To answer RQ2, we perform an experiment that measures the sensitivity of DYNAMITE to the number and quality of records in the provided input-output examples. To perform

[3]Schemas for all benchmarks are available at `https://bit.ly/schemas-dynamite`.

[4]All input-output examples and synthesized programs are available at `https://bit.ly/benchmarks-dynamite`.

Table 3: Main results. Average search space size is calculated by geometric mean; all other averages are arithmetic mean.

Benchmark	Avg # Examples		Search Space	Synthesis Time (s)	# Rules	# Preds per Rule	# Optim Rules	Dist to Optim	Migration Time (s)
	Source	Target							
Yelp-1	4.7	3.9	4.8×10^{120}	6.0	8	1.8	7	0.38	328
IMDB-1	6.0	2.7	1.5×10^{20}	2.7	9	3.6	5	1.22	1153
DBLP-1	1.5	2.6	1.1×10^{14}	0.8	9	6.4	0	2.44	1060
Mondial-1	1.2	2.8	2.2×10^{88}	2.5	25	3.3	17	1.40	5
MLB-1	2.0	1.4	9.1×10^{81}	13.0	7	3.9	2	1.71	1020
Airbnb-1	4.0	2.5	1.7×10^{38}	2.0	6	2.7	4	1.33	286
Patent-1	2.6	2.3	1.4×10^{49}	3.0	7	2.4	5	1.14	553
Bike-1	2.3	2.0	3.1×10^{47}	2.0	7	2.0	5	0.71	2601
Tencent-1	1.5	1.0	1.3×10^{12}	0.2	1	4.0	0	3.00	65
Retina-1	1.5	1.5	3.1×10^{19}	0.8	2	2.0	2	0.00	9
Movie-1	3.6	2.2	5.2×10^{11}	2.9	5	2.8	3	1.00	1062
Soccer-1	1.9	2.0	2.9×10^{11}	0.5	7	1.0	7	0.00	15
Tencent-2	1.5	1.0	1.3×10^{12}	0.2	1	4.0	0	3.00	160
Retina-2	2.0	2.0	3.3×10^{19}	4.0	2	2.5	1	0.50	22
Movie-2	2.4	2.3	1.0×10^{18}	22.7	4	7.0	0	4.00	40
Soccer-2	2.5	2.1	6.9×10^{22}	87.9	7	4.4	4	1.71	311
Yelp-2	4.5	1.8	2.9×10^{73}	0.5	4	1.0	4	0.00	1160
IMDB-2	2.4	2.5	2.3×10^{11}	1.1	11	3.1	5	1.27	3409
DBLP-2	2.1	2.1	1.2×10^{4}	3.6	17	1.8	16	0.06	1585
Mondial-2	1.0	2.1	8.2×10^{24}	30.8	27	1.9	26	0.04	7
MLB-2	2.2	1.9	3.3×10^{84}	2.6	12	1.3	10	0.25	785
Airbnb-2	2.8	2.7	1.4×10^{28}	0.9	7	1.3	7	0.00	664
Patent-2	2.0	2.1	3.9×10^{51}	1.0	8	1.4	6	0.38	786
Bike-2	2.3	2.5	7.3×10^{47}	0.4	6	1.8	4	0.83	3346
MLB-3	2.2	1.3	9.1×10^{81}	3.3	4	2.3	3	0.50	145
Airbnb-3	2.5	2.6	3.3×10^{28}	0.5	7	1.1	7	0.00	57
Patent-3	2.8	2.3	1.3×10^{40}	3.9	8	1.6	7	0.38	122
Bike-3	4.3	2.2	7.3×10^{47}	4.1	5	1.8	4	0.20	519
Average	**2.6**	**2.2**	$\mathbf{5.1 \times 10^{39}}$	**7.3**	**8.0**	**2.5**	**5.8**	**0.79**	**760**

this experiment, we first fix the number r of records in the input example. Then, we randomly generate 100 examples of size r and obtain the output example by running the "golden" program (written manually) on the randomly generated input example. Then, for each size $r \in [1, 8]$, we measure average running time across all 100 examples as well as the percentage of examples for which DYNAMITE synthesizes the correct program within 10 minutes.

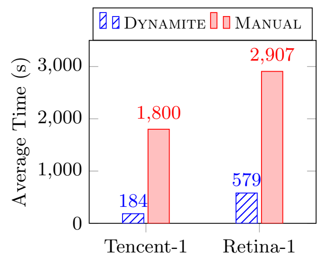
The results of this experiment are summarized in Figure 7 for four representative benchmarks (the remaining 24 are provided in Appendix C of the extend version [47]). Here, the x-axis shows the number of records r in the input example, and the y-axis shows both (a) the average running time in seconds for each r (the blue line with circles) and (b) the % of correctly synthesized programs given r randomly-generated records (the red line with squares).

Overall, this experiment shows that DYNAMITE is not particularly sensitive to the number and quality of examples for 26 out of 28 benchmarks: it can synthesize the correct Datalog program in over 90% of the cases using $2-3$ *randomly-generated* examples. Furthermore, synthesis time grows roughly linearly for 24 out of 28 benchmarks. For 2 of the remaining benchmarks (namely, IMDB-1 and Movie-2), synthesis time seems to grow exponentially in example size; however, since DYNAMITE can already achieve a success rate over 90% with just 2-3 examples, this growth in running time is not a major concern. Finally, for the last 2 benchmarks (namely, Retina-2 and Soccer-2), DYNAMITE does not seem to scale beyond example size of 3. For these benchmarks, DYNAMITE seems to generate complicated intermediate programs with complex join structure, which causes the resulting output to be very large and causes MDP analysis to become very time-consuming. However, this behavior (which is triggered by randomly generated inputs) can be prevented by choosing more representative examples that allow DYNAMITE to generate better sketches.

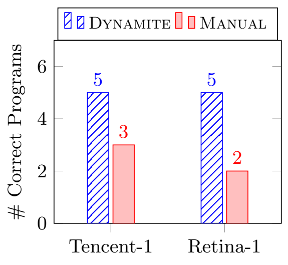
6.3 User Study

To answer question RQ3, we conduct a small-scale user study to evaluate whether DYNAMITE is helpful to users in practice. To conduct this user study, we recruited 10 participants (all of them graduate computer science students with advanced programming skills) and asked them to solve two of our benchmarks from Table 2, namely Tencent-1 and Retina-1, with and without using DYNAMITE. In order to avoid any potential bias, we randomly assigned users to solve each benchmark either using DYNAMITE or without. For the setting where users were not allowed to use DYNAMITE, they were, however, permitted to use any programming language and library of their choice, and they were also allowed to consult search engines and on-line forums. In the setting where the participants did use DYNAMITE, we instructed them to use the tool in interactive mode (recall Section 5). Overall, exactly 5 randomly chosen participants solved Tencent-1 and Retina-1 using DYNAMITE, and 5 users solved each benchmark without DYNAMITE.

The results of this study are provided in Figure 8. Specifically, Figure 8(a) shows the average time to solve the two benchmarks with and without using DYNAMITE. As we can see from this Figure, users are significantly more productive, namely by a factor of 6.2x on average, when migrating data with the aid of DYNAMITE. Furthermore, as shown in Figure 8(b), participants always generate the correct target database instance when using DYNAMITE; however, they fail

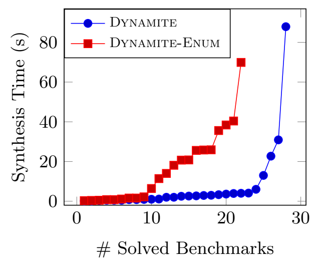


(a) Completion Time

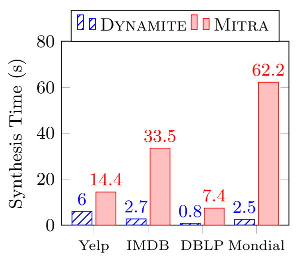


(b) Program Correctness

Figure 8: Results of user study.



(a) Comparison with enum



(b) Comparison with MITRA

Figure 9: Comparing DYNAMITE to baseline and MITRA.

to do so in 50% of the cases when they write the program manually. Upon further inspection, we found that the manually written programs contain subtle bugs, such as failing to introduce newlines or quotation marks. We believe these results demonstrate that DYNAMITE can aid users, including seasoned programmers, successfully and more efficiently complete real-world data migration tasks.

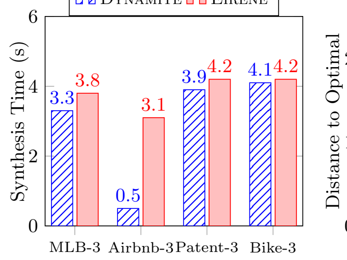
6.4 Comparison with Synthesis Baseline

To answer RQ4, we compare DYNAMITE against a baseline called DYNAMITE-ENUM that uses enumerative search instead of the sketch completion technique described in Section 4.3. In particular, DYNAMITE-ENUM uses the lazy enumeration algorithm based on SMT, but it does not use the ANALYZE procedure for learning from failed synthesis attempts. Specifically, whenever the SMT solver returns an incorrect assignment σ, DYNAMITE-ENUM just uses $\neg\sigma$ as a blocking clause. Thus, DYNAMITE-ENUM essentially enumerates all possible sketch completions until it finds a Datalog program that satisfies the input-output example.

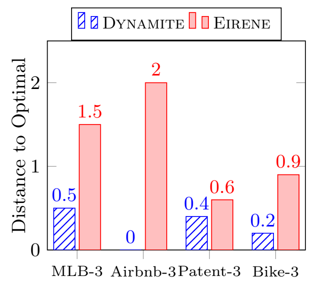
Figure 9(a) shows the results of the comparison when using the manually-provided input-output examples from Section 6.1. In particular, we plot the time in seconds that each version takes to solve the first n benchmarks. As shown in Figure 9(a), DYNAMITE can successfully solve all 28 benchmarks whereas DYNAMITE-ENUM can only solve 22 (78.6%) within the one hour time limit. Furthermore, for the first 22 benchmarks that can be solved by both versions, DYNAMITE is 9.2x faster compared to DYNAMITE-ENUM (1.8 vs 16.5 seconds). Hence, this experiment demonstrates the practical advantages of our proposed sketch completion algorithm compared to a simpler enumerative-search baseline.

6.5 Comparison with Other Tools

While there is no existing programming-by-example (PBE) tool that supports the full diversity of source/target schemas handled by DYNAMITE, we compare our approach against two other tools, namely MITRA and EIRENE, in two more



(a) Synthesis Time



(b) Mapping Quality

Figure 10: Comparing DYNAMITE against EIRENE.

specialized data migration scenarios. Specifically, MITRA [49] is a PBE tool that automates document-to-relational transformations, whereas EIRENE [6] infers relational-to-relational schema mappings from input-output examples.

Comparison with Mitra. Since MITRA uses a domain-specific language that is customized for transforming tree-structured data into a tabular representation, we compare DYNAMITE against MITRA on the four data migration benchmarks from [49] that involve conversion from a document schema to a relational schema. The results of this comparison are summarized in Figure 9(b), which shows synthesis time for each tool for all four benchmarks. In terms of synthesis time, DYNAMITE outperforms MITRA by roughly an order of magnitude: in particular, DYNAMITE takes an average of 3 seconds to solve these benchmarks, whereas MITRA needs 29.4 seconds. Furthermore, MITRA synthesizes 559 and 780 lines of JavaScript for Yelp and IMDB, and synthesizes 134 and 432 lines of XSLT for DBLP and Mondial. In contrast, DYNAMITE synthesizes 13 Datalog rules on average. These statistics suggest that the programs synthesized by DYNAMITE are more easily readable compared to the JavaScript and XSLT programs synthesized by MITRA. Finally, if we compare DYNAMITE and MITRA in terms of efficiency of the synthesized programs, we observe that DYNAMITE-generated programs are 1.1x faster.

Comparison with Eirene. Since EIRENE specializes in inferring relational-to-relational schema mappings, we compare DYNAMITE against EIRENE on the four relational-to-relational benchmarks from Section 6.1 using the same input-output examples. As shown in Figure 10(a), DYNAMITE is, on average, 1.3x faster than EIRENE in terms of synthesis time. We also compare DYNAMITE with EIRENE in terms of the quality of inferred mappings using the same "distance from optimal schema mapping metric" defined in Section 6.1.[5] As shown in Figure 10(b), the schema mappings synthesized by DYNAMITE are closer to the optimal mappings than those synthesized by EIRENE. In particular, EIRENE-synthesized rules have 4.5x more redundant body predicates than the DYNAMITE-synthesized rules.

7. RELATED WORK

Schema mapping formalisms. There are several different formalisms for expressing schema mappings, including visual representations [39, 38], schema modification operators [15, 14], and declarative constraints [5, 4, 6, 20, 29, 12, 8]. Some techniques require the user to express the schema mapping visually by drawing arrows between attributes in the source

[5]To conduct this measurement, we manually wrote optimal schema mappings in the formalism used by EIRENE.

and target schemas [39, 37]. In contrast, schema modification operators express the schema mapping in a domain-specific language [15, 14]. Another common approach is to express the schema mapping using declarative specifications, such as Global-Local-As-View (GLAV) constraints [5, 4, 6, 20]. Similar to this third category, we express the schema mapping using a declarative, albeit executable, formalism.

Schema mapping inference. There is also a body of prior work on *automatically inferring* schema mappings [33, 50, 7, 19, 37, 24, 25, 5, 6, 38]. Clio [37, 24] infers schema mappings for relational and XML schemas given a value correspondence between atomic schema elements. Another line of work [9, 36] uses model management operators such as ModelGen [10] to translate schemas from one model to another. In contrast, Dynamite takes examples as input, which are potentially easier to construct for non-experts. There are also several other schema mapping techniques that use examples. For instance, Eirene [5, 6] interactively solicits examples to generate a GLAV specification. Eirene is restricted to relational-to-relational mappings and does not support data filtering, but their language can also express mappings that are not expressible in the Datalog fragment used in this work. Similarly, Bonifati et al. use example tuples to infer possible schema mappings and interact with the user via binary questions to refine the inferred mappings [11]. In contrast to Dynamite, [11] only focuses on relational-to-relational mappings [5]. Finally, MWeaver [38] provides a GUI to help users to interactively generate attribute correspondences based on examples. MWeaver is also restricted to relational-to-relational mappings and disallows numerical attributes in the source database for performance reasons. Furthermore, it requires the entire source instance to perform mapping inference.

Program synthesis for data transformation. There has been significant work on automating data transformations using program synthesis [49, 23, 32, 27, 48, 45, 43, 44]. Many techniques focus only on table or string transformations [23, 27, 45, 32, 43], whereas Hades [48] (resp. Mitra [49]) focuses on document-to-document (resp. document-to-table) transformations. Our technique generalizes prior work by automating transformations between many different types of database schemas. Furthermore, as we demonstrate in Section 6.5, this generality does not come at the cost of practicality, and, in fact, performs faster synthesis.

Inductive logic programming. Our work is related to inductive logic programming (ILP) where the goal is to synthesize a logic program consistent with a set of examples [34, 35, 40, 26, 30, 51, 18, 41]. Among ILP techniques, our work is most similar to recent work on Datalog program synthesis [3, 42]. In particular, Zaatar [3] encodes an under-approximation of Datalog semantics using the theory of arrays and reduces synthesis to SMT solving. However, this technique imposes an upper bound on the number of clauses and atoms in the Datalog program. The Alps tool [42] also performs Datalog program synthesis from examples but additionally requires meta-rule templates. In contrast, our technique focuses on a recursion-free subset of Datalog, but it does not require additional user input beyond examples and learns from failed synthesis attempts by using the concept of minimal distinguishing projections.

Learning from conflicts in synthesis. Our method bears similarities to recent work on conflict-driven learning in program synthesis [22, 32, 46] where the goal is to learn useful information from failed synthesis attempts. For example, Neo [22] and Trinity [32] use component specifications to perform root cause analysis and identify other programs that also cannot satisfy the specification. Dynamite's sketch completion approach is based on a similar insight, but it uses Datalog-specific techniques to perform inference. Another related work is Migrator [46], which automatically synthesizes a new version of a SQL program given its old version and a new relational schema. In contrast to Migrator, our method addresses the problem of migrating *data* rather than *code* and is not limited to relational schemas. In addition, while Migrator also aims to learn from failed synthesis attempts, it does so using testing as opposed to MDP analysis for Datalog.

Universal and core solutions for data exchange. Our work is related to the data exchange problem [20], where the goal is to construct a target instance J given a source instance I and a schema mapping Σ such that $(I, J) \models \Sigma$. Since such a solution J is not unique, researchers have developed the concept of universal and core solutions to characterize generality and compactness [20, 21]. In contrast, during its data migration phase, Dynamite obtains a unique target instance by executing the synthesized Datalog program on the source instance. The target instance generated by Dynamite is the least Herbrand model of the Datalog rules and the source instance [13]. While the least Herbrand model also characterizes generality and compactness of the target instance, the relationship between the least Herbrand model and the universal/core solution for data exchange requires further theoretical investigation.

8. LIMITATIONS

Our approach has three limitations that we plan to address in future work. First, our synthesis technique does not provide any guarantees about the optimality of the synthesized Datalog programs, either in terms of performance or size. Second, we assume that the examples provided by the user are always correct; thus, our method does not handle any noise in the specification. Third, we assume that we can compare string values for equality when inferring the attribute mapping and obtain the proper matching using set containment. If the values are slightly changed, or if there is a different matching heuristic between attributes, our technique would not be able to synthesize the desired program. However, this shortcoming can be overcome by more sophisticated schema matching techniques [17, 31].

9. CONCLUSION

We have proposed a new PBE technique that can synthesize Datalog programs to automate data migration tasks. We evaluated our tool, Dynamite, on 28 benchmarks that involve migrating data between different types of database schemas and showed that Dynamite can successfully automate the desired data migration task from small input-output examples.

10. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd for their useful comments and suggestions. This work is supported by NSF Awards #1712067 and #1762299.

11. REFERENCES

[1] Kaggle. `https://www.kaggle.com`, 2019.

[2] MongoDB. `https://www.mongodb.com`, 2019.

[3] A. Albarghouthi, P. Koutris, M. Naik, and C. Smith. Constraint-based synthesis of datalog programs. In *Proceedings of CP*, pages 689–706, 2017.

[4] B. Alexe, P. G. Kolaitis, and W.-C. Tan. Characterizing schema mappings via data examples. In *Proceedings of PODS*, pages 261–272, 2010.

[5] B. Alexe, B. ten Cate, P. G. Kolaitis, and W. C. Tan. Designing and refining schema mappings via data examples. In *Proceedings of SIGMOD*, pages 133–144, 2011.

[6] B. Alexe, B. Ten Cate, P. G. Kolaitis, and W.-C. Tan. Eirene: Interactive design and refinement of schema mappings via data examples. *PVLDB*, 4(12):1414–1417, 2011.

[7] Y. An, A. Borgida, R. J. Miller, and J. Mylopoulos. A semantic approach to discovering schema mapping expressions. In *Proceedings of ICDE*, pages 206–215, 2007.

[8] P. C. Arocena, B. Glavic, and R. J. Miller. Value invention in data exchange. In *Proceedings of SIGMOD*, pages 157–168, 2013.

[9] P. Atzeni, P. Cappellari, R. Torlone, P. A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.

[10] P. A. Bernstein. Applying model management to classical meta data problems. In *Proceedings of CIDR*, 2003.

[11] A. Bonifati, U. Comignani, E. Coquery, and R. Thion. Interactive mapping specification with exemplar tuples. In *Proceedings of SIGMOD*, pages 667–682, 2017.

[12] B. T. Cate, P. G. Kolaitis, K. Qian, and W.-C. Tan. Approximation algorithms for schema-mapping discovery from data examples. *ACM Transactions on Database Systems*, 42(2):12, 2017.

[13] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE TKDE*, 1(1):146–166, 1989.

[14] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.

[15] C. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *PVLDB*, 1(1):761–772, 2008.

[16] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, pages 337–340, 2008.

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

[18] R. Evans and E. Grefenstette. Learning explanatory rules from noisy data (extended abstract). In *Proceedings of IJCAI*, pages 5598–5602, 2018.

[19] R. Fagin, L. M. Haas, M. A. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, pages 198–236, 2009.

[20] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[21] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[22] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *Proceedings of PLDI*, pages 420–435, 2018.

[23] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of PLDI*, pages 422–436, 2017.

[24] A. Fuxman, M. A. Hernández, C. T. H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: Schema mapping reloaded. In *Proceedings of VLDB*, pages 67–78, 2006.

[25] G. Gottlob and P. Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2):6:1–6:37, 2010.

[26] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. G. Zorn. Inductive programming meets the real world. *Communication of the ACM*, 58(11):90–99, 2015.

[27] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of PLDI*, pages 317–328, 2011.

[28] H. Jordan, B. Scholz, and P. Subotic. Soufflé: On synthesis of program analyzers. In *Proceedings of CAV*, pages 422–430, 2016.

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

[30] D. Lin, E. Dechter, K. Ellis, J. B. Tenenbaum, and S. Muggleton. Bias reformulation for one-shot function induction. In *Proceedings of ECAI*, pages 525–530, 2014.

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

[32] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An extensible synthesis framework for data science. *PVLDB*, 12(12):1914–1917, 2019.

[33] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *Proceedings of VLDB*, pages 77–88, 2000.

[34] S. H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.

[35] S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[36] P. Papotti and R. Torlone. Heterogeneous data translation through XML conversion. *J. Web Eng.*, 4(3):189–204, 2005.

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

[38] L. Qian, M. J. Cafarella, and H. V. Jagadish.

Sample-driven schema mapping. In *Proceedings of SIGMOD*, pages 73–84, 2012.

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

[40] T. Rocktäschel and S. Riedel. End-to-end differentiable proving. In *Proceedings of NIPS*, pages 3788–3800, 2017.

[41] X. Si, M. R. Lee, K. Heo, and M. Naik. Synthesizing datalog programs using numerical relaxation. In *Proceedings of IJCAI*, 2019.

[42] X. Si, W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of ESEC/SIGSOFT FSE*, pages 515–527, 2018.

[43] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[44] R. Singh, V. V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[45] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of PLDI*, pages 452–466, 2017.

[46] Y. Wang, J. Dong, R. Shah, and I. Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of PLDI*, pages 286–300, 2019.

[47] Y. Wang, R. Shah, A. Criswell, R. Pan, and I. Dillig. Data migration using datalog program synthesis. `http://arxiv.org/abs/2003.01331`, 2020.

[48] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of PLDI*, pages 508–521, 2016.

[49] N. Yaghmazadeh, X. Wang, and I. Dillig. Automated migration of hierarchical data to relational tables using programming-by-example. *PVLDB*, 11(5):580–593, 2018.

[50] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Proceedings of SIGMOD*, pages 485–496, 2001.

[51] F. Yang, Z. Yang, and W. W. Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Proceedings of NIPS*, pages 2319–2328, 2017.