The Dissertation Committee for Yuepeng Wang
certifies that this is the approved version of the following dissertation:

Formal Methods for Database Schema Refactoring

Committee:

Isil Dillig, Supervisor

William Cook

Vijaychidambaram Velayudhan Pillai

Armando Solar-Lezama

Christopher Jermaine

Formal Methods for Database Schema Refactoring

by

Yuepeng Wang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2020

Dedicated to my parents.

Acknowledgments

Pursuing PhD is a journey full of adventures and challenges. I am really grateful that I have received lots of help since I started this journey.

First of all, I want to express my deep gratitude to my advisor, Isil Dillig, who brought me to the research world. Isil's passion and enthusiasm always encouraged me to study challenging research problems, and her knowledge and skills constantly inspired me to refine the way I tackle problems and helped me become a better researcher in programming languages. Besides research, thanks for making my PhD journey a pleasant, joyful, and rewarding experience; all those fantastic events, such as group hiking, potluck, pool party, and game night, are unforgettable to me.

I also want to express my deep appreciation to the rest of my committee: William Cook, Vijay Chidambaram, Armando Solar-Lezama, and Christopher Jermaine, for their generous guidance when I prepare and write up this dissertation. Thanks for helping me improve the dissertation with insightful comments and feedback.

I would like to thank Shuvendu Lahiri for mentoring my internship at Microsoft Research and providing valuable suggestions on my project. Thanks for sharing the experience of industrial labs and allowing me to explore the wonderful Seattle area.

I would also like to thank my labmates in the UToPiA group: Yu Feng, Xinyu Wang, Ruben Martins, Valentin Wuestholz, Navid Yaghmazadeh, Kostas Ferles, Jia Chen, Jiayi Wei, Greg Anderson, Jon Stephens, Jocelyn Chen, Shankara Pailoor, Ben Mariano, Jame Dong, Rushi Shah, and Maruth Goyal. We have been working in the same group, collaborating on papers, and providing feedback for each other's projects. We have also been exploring food, scene, music, and culture in "weird" Austin. Thank you all for leaving me good memories along my PhD journey.

Outside the UToPiA group, I have made many considerate and relaxing friends in the Computer Science department: Wenguang Mao, Jian He, Zhiting Zhu, Hangchen Yu, Ye Zhang, Xueyu Mao, Wenzhi Cui, and Jianyu Huang. Thanks for sharing funny stories and entertaining experiences, which added lots of fun to my daily life in the GDC building.

Last but not least, I want to thank my parents for their continuous encouragement and emotional support during the tough times of my journey. The encouragement and support kept me moving forward without giving up and finally made me a better myself.

Formal Methods for Database Schema Refactoring

Yuepeng Wang, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Isil Dillig

Database applications typically undergo several schema refactorings during their life cycle due to performance or maintainability reasons. Such refactorings not only require migrating the underlying data to a new schema but also re-implementing large chunks of the code that query and update the database. The code and data migration tasks implied by schema refactoring are notoriously challenging to developers, as they are time-consuming and error-prone.

Motivated by these challenges, this dissertation presents formal method techniques to help developers correctly and easily evolve database applications during schema refactoring. Specifically, it first describes how to verify equivalence between database applications that operate over different schemas, such as those that arise before and after schema refactoring. Next, it presents a novel technique that can automatically synthesize an equivalent version of the database program that operates over the new schema. Finally, it describes a synthesis technique that helps developers migrate data between schemas using input-output examples.

We also implement research prototypes based on the proposed techniques and perform experiments to evaluate their effectiveness and efficiency. The experimental results demonstrate that our techniques are effective for code and data migration during schema refactoring, and they are more efficient than several baselines and state-of-the-art approaches.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction [1]

Database applications have been, and continue to be, enormously popular in software development. For example, most contemporary websites are built using database applications in order to generate web page content dynamically. As people rely on various database applications in modern software systems, database applications have formed the backbone of many aspects of human society, including healthcare, finance, e-commerce, and industries.

Generally speaking, database applications continuously evolve due to performance or maintainability reasons. One common theme of the evolution is that database applications usually undergo *schema refactoring* several times during their life cycle [9, 54]. A schema refactoring typically involves some changes to the database schema, intending to improve the design and/or performance of the application *without* changing its semantics. For instance, example changes during schema refactoring include splitting a table into multiple tables, merging many tables into a single one, and moving columns from one table to another.

[1] This chapter is adapted from the author's previous publications [130, 131, 133], where the author led the technical discussion, tool development, and experimental evaluation.

Despite the frequent need to perform schema refactoring, this task is known to be non-trivial [8, 138]. In order to retain the semantics of database applications during schema refactoring, developers need to solve two problems after changing the schema: *code migration* and *data migration*. Specifically, code migration requires re-implementing large chunks of code that query and update the database, and data migration requires migrating the underlying data to the new schema. Both of these problems are demanding and challenging for developers.

On one hand, code migration is time-consuming and error-prone. Database applications may have hundreds of SQL functions for data access and manipulation, and many of these functions need to be rewritten after the schema is changed [39]. While developers may choose to migrate affected SQL functions manually, such migration requires lots of effort and thus introduces significant burdens. Even worse, developers need to be very careful about manual code migration, because it is easy to introduce bugs in the migration process [8, 9].

On the other hand, data migration is inefficient and tedious. While developers may write scripts to perform data migration, these scripts are rarely executed after the schema refactoring is finished. Furthermore, migration scripts usually require a long time to write and distract developers from other creative programming tasks. An alternative approach is providing a schema mapping that describes the relationship of two database schemas and generating a script based on the schema mapping to perform data migration [86].

However, the desired schema mappings for realistic database schemas is increasingly impractical to be specified by developers, because large database schemas could result in complicated and daunting schema mappings [6].

Motivated by these challenges, this dissertation presents formal method techniques for schema refactoring. In particular, it consists of *verification* and *synthesis* techniques that can help developers correctly and easily perform code and data migration. Here, verification is an effective means to ensure that developers do not introduce bugs in the refactoring process. Synthesis facilitates a high degree of automation and significantly reduces the manual effort from developers during schema refactoring. Overall, this dissertation aims to present verification and synthesis techniques that can provide developers with high confidence in the correctness of migration, as well as improve the developer productivity for schema refactoring.

First of all, we describe how to verify equivalence between database programs operating over different schemas, such as those that arise before and after schema refactoring [130]. While there has been significant progress in the area of equivalence checking [33, 34, 94, 97, 105, 146], existing techniques cannot be used to verify equivalence between *database programs*. The problem of equivalence checking for database programs introduces several challenges that are not addressed by previous work, ranging from equivalence definition to proof methodology. In this dissertation, we formalize the equivalence verification problem for database programs and propose a verification algorithm based on bisimulation invariants for proving equivalence. Specifically, given

two programs $\mathcal{P}$ and $\mathcal{P}'$ over schemas $\mathcal{S}$ and $\mathcal{S}'$ respectively, our algorithm can automatically verify whether $\mathcal{P}$ is equivalent to $\mathcal{P}'$ or not. This technique is useful for developers to prove the correctness of a manual code migration during schema refactoring.

Next, building on top of the verification procedure, we present a novel synthesis technique that can automatically migrate the database program to a new schema [131]. Specifically, given the original database program $\mathcal{P}$ over schema $\mathcal{S}$ and a new database schema $\mathcal{S}'$, the goal of synthesis is to automatically generate a new program $\mathcal{P}'$ over $\mathcal{S}'$ such that $\mathcal{P}'$ is provably equivalent to $\mathcal{P}$. Since programmers typically spend significant time and effort in rewriting the program after a schema change, such automated synthesis techniques have the potential to improve programmer productivity dramatically. In order to automate code migration in real-world database applications, we must overcome the difficulty of scaling to a large number of SQL functions and minimizing the number of calls issued to the verifier. To address the scalability issues, we present a synthesis algorithm that decomposes the problem into more tractable sub-problems. In particular, we first generate a database program sketch, which includes many candidate instantiations of the target program, by inferring a value correspondence between the source and target schemas. Then we develop an efficient sketch completion algorithm that utilizes counterexamples to rule out many syntactically valid, but semantically incorrect, instantiations of the generated sketch.

Finally, we describe an automatic data migration technique based on

the programming-by-example paradigm to help developers migrate data during schema refactoring [133]. Specifically, given the original database and a target schema, our technique can perform the desired data transformation using only a number of demonstrating examples. Unlike previous programming-by-example efforts [6, 102, 140] that typically focus on data transformations between particular types of schemas, our technique provides a general solution between many types of schemas, including relational, document, and graph schemas. The key idea is to use a Datalog program to relate the source and target schemas and reduce the automatic data migration problem to the problem of synthesizing a Datalog program from examples. However, synthesizing Datalog programs from examples for data transformation is challenging, because real-world database schemas usually have a large number of entities and attributes, and the search space over all possible Datalog programs is enormous. To address this challenge, we leverage the properties of Datalog, analyze the root cause of the discrepancy between the expected and actual output yielded by an incorrect program, and then utilize the root cause information to prune many incorrect programs at a time.

To evaluate the proposed techniques, we have implemented three tools:

1. Mediator for verifying equivalence between database programs that operate over different schemas

2. Migrator for automatically migrating code to a new schema

3. DYNAMITE for automated data migration between different schemas using demonstrating input-output examples

The experimental results demonstrate that our techniques are effective for code and data migration during schema refactoring. Furthermore, these tools are more efficient than several baselines and state-of-the-art systems.

In summary, this dissertation makes the following key contributions:

- We introduce and formalize the equivalence verification problem for database programs over different schemas.
- We present a sound and relatively complete proof methodology for verifying equivalence between database programs.
- We show how to enable automated reasoning over relational algebra with updates and how to automatically infer suitable invariants for verifying equivalence.
- We propose a new synthesis technique for automatically migrating database programs to a new schema.
- We describe a new sketch completion algorithm for database programs that is based on symbolic search and conflict-driven learning from minimum failing inputs.
- We present a formulation of the automated data migration problem in terms of Datalog program synthesis.

- We describe a new algorithm that leverages Datalog properties and root cause analysis to efficiently synthesize Datalog programs from input-output examples.

The rest of this dissertation is organized as follows. Chapter 2 describes the verification technique for proving equivalence between database programs over different schemas. Chapter 3 presents the synthesis technique for automatically migrating database programs from one schema to another. Chapter 4 shows the synthesis technique for automated data migration between different schemas using demonstrating examples. Chapter 5 surveys the related work and Chapter 6 concludes the dissertation.

Chapter 2

Verifying Equivalence of Database Programs [1]

While there has been significant progress in equivalence checking [33, 34, 94, 97, 105, 146], existing techniques cannot be used to verify the equivalence of database programs. To see why verifying equivalence is important in this context, consider the scenario in which a web application interacts with a relational database to dynamically render a web page, and suppose that the database schema needs to be changed either for performance or maintainability reasons. In this case, the developer will need to migrate the database to the new schema and also re-implement the code that interacts with the database *without* changing the observable behavior of the application. While this task of database refactoring arises very frequently during the life cycle of web applications, it is also known to be quite hard and error-prone [8, 138], and several textbooks have been published on this topic [9, 54]. As pointed out by Faroult and L'Hermite, *"database applications are a difficult ground for refactoring"* because *"small changes are not always what they appear to be"* and *"testing the validity of a change may be difficult"* [54].

Motivated by the prevalence of database applications in the real world

[1] This chapter is adapted from the author's previous publication [130], where the author led the technical discussion, tool development, and experimental evaluation.

and the frequent need to perform schema refactoring, we propose a new technique for verifying equivalence of database programs. Given a pair of programs $\mathcal{P}$, $\mathcal{P}'$ that interact with two different databases, we would like to automatically construct a proof that $\mathcal{P}$ and $\mathcal{P}'$ are semantically equivalent. Unfortunately, the problem of equivalence checking for database programs introduces several challenges that are not addressed by previous work: First, it is unclear how to define equivalence in this context, particularly when the two database schemas are different. Second, database applications typically use declarative query languages, such as SQL, but, to the best of our knowledge, there are no automated reasoning tools for a rich enough fragment of SQL that captures realistic use cases.

In this chapter, we formalize the equivalence checking problem for database applications and propose a verification algorithm for proving equivalence. Suppose that we are given two programs $\mathcal{P}$, $\mathcal{P}'$ that interact with databases D, D'. Let us also assume that each program comprises a set of SQL functions (query or update) such that every function f in $\mathcal{P}$ has a corresponding function f' in $\mathcal{P}'$. Our goal is to prove that $\mathcal{P}$ and $\mathcal{P}'$ yield the same result on a pair of corresponding queries Q, Q' whenever we execute the same sequence of update operations on D and D'.

To prove equivalence between a pair of database applications, our approach infers a so-called *bisimulation invariant* that relates states of the two programs. In this context, program states correspond to database instances, so the bisimulation invariants relate a pair of database instances. As shown

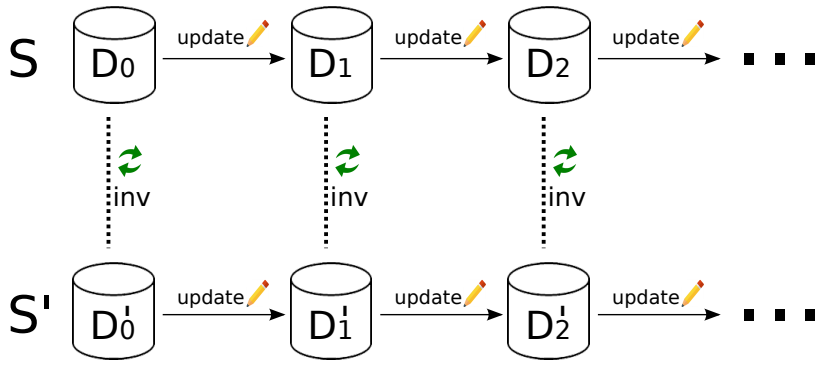


Figure 2.1: Bisimulation invariant between two database programs.

in Figure 2.1, our bisimulation invariants are preserved after each database transaction, and, in addition, they are strong enough to establish that any corresponding pair of queries must yield the same result.

In the context of software verification, program invariants are typically expressed in some first-order theory supported by modern SMT solvers. Unfortunately, since the bisimulation invariants that we require in this context relate a pair of databases, they typically involve non-trivial relational algebra operators, such as join and selection. To solve this difficulty, we consider the *theory of relational algebra with updates*, $\mathcal{T}_{RA}$, and present an SMT-friendly encoding of $\mathcal{T}_{RA}$ into the theory of lists, which is supported by many SMT solvers.

Once we automate reasoning in relational algebra with updates, a remaining challenge is to automatically infer suitable bisimulation invariants. In this chapter, we use the framework of monomial predicate abstraction to automatically synthesize conjunctive quantifier-free invariants that relate two

database states [16, 41, 76]. Specifically, we identify a family F of predicates that are useful for proving equivalence and generate the strongest conjunctive bisimulation invariant over this universe. Towards this goal, we define a strongest post-condition semantics of database transactions and automatically generate verification conditions whose validity establishes the correctness of a candidate bisimulation invariant.

We have implemented the proposed approach in a tool called MEDIATOR for verifying equivalence between applications written in our intermediate representation (IR), which abstracts database applications as a fixed set of queries and updates to the database. To evaluate our methodology, we consider 21 database applications translated into our IR and show that MEDIATOR can successfully verify equivalence between benchmarks extracted from real-world web applications with up to hundreds of functions. We also show that MEDIATOR can handle challenging textbook examples that illustrate a wide spectrum of structural changes to the database schema. Overall, our experiments show that the proposed method is useful and practical: MEDIATOR can successfully verify the desired property for 10 out of 11 real-world benchmarks in under 50 seconds on average.

2.1 Motivating Example

Consider a database-driven Connected Diagnostics Platform (`cdx`) for notifying patients about the results of their medical tests and alerting community health workers about (anonymous) positive test results in their area.

This application interacts with a database that stores information about subscribers, laboratories, institutions and so on. In an earlier version of the application, the database contains a *Subscriber* relation with three attributes, namely *sid*, which corresponds to the subscriber id, *sname*, which is the name of the subscriber, and *filter*, which is used to filter out irrelevant subscriptions. In the updated version, the developers decide to refactor this information into two separate relations:

Subscriber(sid, sname, fid_fk) *Filter(fid, fname, params)*

The new database schema now contains an additional *Filter* relation, and the *fid_fk* attribute in *Subscriber* is now a foreign key referring to the corresponding filter in the *Filter* relation.

After refactoring the database schema in this manner, the developers also re-implement the relevant parts of the code that interact with this database. In particular, Figure 2.2 shows the relevant functionality before and after the database refactoring, where *UUID_x* is a unique *fid*. Both versions of the code contain three functions for updating the database, namely *createSub*, *deleteSub*, and *updateSub*, and two functions (*getSubName* and *getSubFilter*) for querying the database. However, the underlying implementation of these functions has changed due to the migration of the schema to a new format. Nonetheless, we would like to be able to show that the application returns the same query results before and after the schema migration. This verification task is non-trivial because the database transactions in the two implementations are often structurally different and operate over different relations.

```
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber VALUES (id, name, fltr);

void deleteSub(int id)
    DELETE FROM Subscriber WHERE sid=id;

void updateSub(int id, String name, String fltr)
    UPDATE Subscriber SET filter=fltr WHERE sid=id;
    UPDATE Subscriber SET sname=name WHERE sid=id;

List<Tuple> getSubName(int id)
    SELECT sname FROM Subscriber WHERE sid=id;

List<Tuple> getSubFilter(int id)
    SELECT filter FROM Subscriber WHERE sid=id;
```

(a) Before Refactoring

```
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber VALUES (id, name, UUID_x);
    INSERT INTO Filter VALUES (UUID_x,
              "Filter for " + name + " subscriber", fltr);

void deleteSub(int id)
    DELETE FROM Filter WHERE fid IN
              ( SELECT fid_fk FROM Subscriber WHERE sid=id );
    DELETE FROM Subscriber WHERE sid=id;

void updateSub(int id, String name, String fltr)
    UPDATE Filter SET params=fltr WHERE fid IN
              ( SELECT fid_fk FROM Subscriber WHERE sid=id );
    UPDATE Subscriber SET sname=name WHERE sid=id;

List<Tuple> getSubName(int id)
    SELECT sname FROM Subscriber WHERE sid=id;

List<Tuple> getSubFilter(int id)
    SELECT params FROM Filter JOIN Subscriber ON fid=fid_fk WHERE sid=id;
```

(b) After Refactoring

Figure 2.2: Sample database programs.

Let us now see how to verify equivalence of the two versions of the `cdx` application before and after refactoring. As mentioned earlier, our method infers a *bisimulation invariant* that relates two versions of the database. In the remainder of this discussion, let us use primed variables to refer to database relations and attributes in the refactored database. For instance, $Subscriber'$ refers to the version of the original *Subscriber* relation in the refactored database.

To come up with a suitable bisimulation invariant, we first generate a finite universe of atomic predicates that *could* be used to relate the two versions of the database. For example, one possible predicate is $\Pi_{sid,sname}(Subscriber) = \Pi_{sid',sname'}(Subscriber')$, which states that the *sid* and *sname* attributes in relation *Subscriber* correspond to sid' and $sname'$ in $Subscriber'$, respectively. Given such predicates, we then try to find a conjunctive formula that is provably a valid bisimulation invariant.

As an example, let us consider the following candidate formula Φ:

$$\Pi_{sid,sname}(Subscriber) = \Pi_{sid',sname'}(Subscriber') \wedge$$
$$\Pi_{sid,sname,filter}(Subscriber) = \Pi_{sid',sname',params'}(Subscriber' \bowtie Filter')$$

Essentially, this formula states that the *sid* and *sname* attributes of *Subscriber* are unchanged, and the *Subscriber* relation in the original database can be obtained by taking the natural join of *Subscriber* and *Filter* relations in the refactored database and then projecting the relevant attributes.

Using our verification methodology, we can automatically prove that the candidate formula Φ corresponds to a valid bisimulation invariant. In par-

ticular, assuming that Φ holds before each pair of update operations U, U' from the original and revised implementations, we can show that Φ still continues to hold after executing U and U'. In other words, this means that Φ is an *inductive* bisimulation invariant. To prove the inductiveness of Φ, we use a strongest postcondition semantics for database update operations as well as an automated theorem prover for the theory of relational algebra with updates.

After finding an inductive bisimulation invariant Φ, we still need to ensure that Φ is strong enough to prove equivalence. For this purpose, we consider every pair of queries Q, Q' from the old and revised versions of the application and try to prove that Q and Q' yield the same results. Using the axioms of theory of relational algebra with updates, it can be shown that $\Phi \Rightarrow Q = Q'$ is logically valid for both of the queries *getSubName* and *getSubFilter* in this application. Hence, we are able to prove equivalence between these two programs even though they use databases that operate over different schemas.

2.2 Problem Statement

This section formalizes the syntax and semantics of database programs and precisely defines the equivalence and refinement checking problems in this context.

2.2.1 Language Syntax

A database program P is a tuple $(\mathcal{S}, \vec{T_U}, \vec{T_Q})$, where $\mathcal{S}$ is the schema of the underlying database, $\vec{T_U}$ is a vector of database update functions, and

$$
\begin{array}{lrcl}
\textit{Program} & \mathcal{P} & := & (\mathcal{S}, \vec{T}_U, \vec{T}_Q) \\
\textit{Schema} & \mathcal{S} & := & R \rightarrow \{a_1 : \tau_1; \ldots; a_n : \tau_n\} \\
\textit{Function} & T & := & \lambda \vec{v}.\ U \mid \lambda \vec{v}.\ Q \\
\textit{Update} & U & := & \texttt{ins}(R, \{a_1 : v_1, \ldots, a_n : v_n\}) \\
 & & \mid & \texttt{del}(R, \phi) \mid \texttt{upd}(R, \phi, a, v) \mid U; U \\
\textit{Query} & Q & := & R \mid \Pi_\psi(Q) \mid \sigma_\phi(Q) \mid Q \bowtie_\phi Q \mid Q \cup Q \mid Q - Q \\
\textit{Attribute list} & \psi & := & a \mid \psi, \psi \\
\textit{Predicate} & \phi & := & p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\textit{Atomic predicate} & p & := & a \odot a \mid a \odot v \mid a \in Q \\
\textit{Operator} & \odot & := & \leq \mid < \mid = \mid \neq \mid > \mid \geq
\end{array}
$$

$$
\begin{array}{lcllcl}
\tau & \in & \{Int, String, \ldots\} & R & \in & \textit{Relation} \\
a & \in & \textit{Attribute} & v & \in & \textit{Variable} \cup \textit{Constant}
\end{array}
$$

Figure 2.3: Syntax of database programs.

$\vec{T}_Q$ is a vector of database queries (see Figure 2.3). We collectively refer to any update or query in $\vec{T}_U \cup \vec{T}_Q$ as a *database function* and denote the i-th function in $\vec{T}_U$ (resp. in $\vec{T}_Q$) as U_i (resp. Q_i). Let us now take a closer look at the syntax in Figure 2.3.

Database schema. The database schema $\mathcal{S}$ provides a logical view of how the database organizes its data. In particular, the schema describes all relations (i.e., tables) stored in the database as well as the typed attributes for each relation. More precisely, we represent the schema $\mathcal{S}$ as a mapping from table names R to their corresponding record types $\{a_1 : \tau_1; \ldots; a_n : \tau_n\}$, which indicates that attribute a_i of table R has type τ_i. In the rest of this chapter, we use the notation $dom(\mathcal{S})$ to denote the set of tables stored in the database.

Update functions. An update function $\lambda \vec{v}.U \in \vec{T}_U$ contains a sequence of

database update operations, including insertion, deletion, and modification. Specifically, the language construct $\texttt{ins}(R, t)$ models the insertion of tuple t into relation R, where $R \in dom(\mathcal{S})$ and tuple t is represented as a mapping from attributes to symbols (variable or constant). Similarly, the statement $\texttt{del}(R, \phi)$ removes all tuples satisfying predicate ϕ from R, and $\texttt{upd}(R, \phi, a, v)$ assigns value v to the a attribute of all tuples satisfying predicate ϕ in R. We assume that each database function occurs atomically (i.e., either all or none of the updates are committed).

Query functions. In our language, query functions $\lambda\vec{v}.Q \in \vec{T}_Q$ are expressed as relational algebra expressions involving projection (Π), selection (σ), join ($\bowtie$), union ($\cup$), and difference ($-$) operators. While our language allows general theta joins of the form $R_1 \bowtie_\phi R_2$, we abbreviate natural joins using the notation $R_1 \bowtie R_2$.

2.2.2 Language Semantics

To define the formal semantics of database programs, we first need to define what we mean by an *input* to the programs defined in Figure 2.3. Since we consider a model in which the user interacts with the application by performing a sequence of updates and queries to the database, we consider a program input to be an *invocation sequence* ω of the form:

$$\omega = (i_1, \sigma_1); \ldots; (i_{n-1}, \sigma_{n-1}); (i_n, \sigma_n)$$

where each i_j specifies an update function $\lambda\vec{v}.U_{i_j}$ for $j \in [1, n)$ and σ_j is its corresponding valuation, mapping values of formal parameters $\vec{v}$ to their concrete values. The last element (i_n, σ_n) in the invocation sequence always corresponds to a *query function* with corresponding valuation σ_n.

Example 2.2.1. *Consider the motivating example from Figure 2.2. This program can be expressed in our intermediate language as shown in Figure 2.4. Now, consider the following invocation sequence, assuming that the five functions are indexed* $1 - 5$ *from top to bottom:*

$$\omega = (1, [\text{id} \mapsto 100, \text{name} \mapsto \mathit{Alice}, \text{fltr} \mapsto \text{Filter1}]); (4, [\text{id} \mapsto 100])$$

This sequence indicates that the user first invokes the update function createSub(100, Alice, Filter1)*, followed by the query function* getSubName(100)*.*

Figure 2.5 defines the denotational semantics for the language presented in Figure 2.3. Our semantics are defined in terms of the standard list combinators *map*, *append*, *filter*, *foldl*, and *contains*. Given a map x, we write $\mathit{vals}(x)$ to denote the list of values stored in the map, and we think of a map as a list of (key, value) pairs. The function $\mathit{delete}(y, ys)$ removes the first occurence of y in list ys. Given two maps y, z with disjoint keys, $\mathit{merge}(y, z)$ generates a new map that contains all key-value pairs in y and z. We use the notation $[\![\mathcal{P}]\!]_\omega$ to represent the result of the last query in ω after performing all updates on an *empty database*. Since any reachable database state can be modeled using a suitable sequence of insertions to an empty database, this assumption does not result in a loss of generality.

`void` *createSub(int id, String name, String fltr)*
  `ins`*(Subscriber, (id, name, fltr))*

`void` *deleteSub(int id)*
  `del`*(Subscriber, sid=id)*

`void` *updateSub(int id, String name, String fltr)*
  `upd`*(Subscriber, sid=id, filter, fltr)*
  `upd`*(Subscriber, sid=id, sname, name)*

`List<Tuple>` *getSubName(int id)*
  $\Pi_{sname}(\sigma_{sid=id}(Subscriber))$

`List<Tuple>` *getSubFilter(int id)*
  $\Pi_{filter}(\sigma_{sid=id}(Subscriber))$

(a) Before Refactoring

`void` *createSub(int id, String name, String fltr)*
  `ins`*(Filter′, (UUID_x, name, fltr))*
  `ins`*(Subscriber′, (id, name, UUID_x))*

`void` *deleteSub(int id)*
  `del`*(Filter′, fid′*$\in \Pi_{fid\_fk'}(\sigma_{sid'=id}(Subscriber'))$*)*
  `del`*(Subscriber′, sid′=id)*

`void` *updateSub(int id, String name, String fltr)*
  `upd`*(Filter′, fid′*$\in \Pi_{fid\_fk'}(\sigma_{sid'=id}(Subscriber'))$*, params′, fltr)*
  `upd`*(Subscriber′, sid′=id, sname′, name)*

`List<Tuple>` *getSubName(int id)*
  $\Pi_{sname'}(\sigma_{sid'=id}(Subscriber'))$

`List<Tuple>` *getSubFilter(int id)*
  $\Pi_{params'}(\sigma_{sid'=id}(Filter' \bowtie Subscriber'))$

(b) After Refactoring

Figure 2.4: Database programs in intermediate language.

$\boxed{[\![\mathcal{P}]\!] :: \textit{Invocation Sequence } \omega \to \textit{ Instance } \Delta \to \textit{List}}$

$$
\begin{array}{lcl}
[\![(\mathcal{S}, \vec{T_U}, \vec{T_Q})]\!]_{(n,\sigma),\Delta} & = & \mathit{map}([\![Q_n]\!]_{\sigma,\Delta}, \lambda x.\mathit{vals}(x)) \\
[\![(\mathcal{S}, \vec{T_U}, \vec{T_Q})]\!]_{(n,\sigma);\omega,\Delta} & = & [\![(\mathcal{S}, \vec{T_U}, \vec{T_Q})]\!]_{\omega,\Delta'} \text{ where } \Delta' = [\![U_n]\!]_{\sigma,\Delta}
\end{array}
$$

$\boxed{[\![U]\!] :: \textit{Valuation } \sigma \to \textit{Instance } \Delta \to \textit{Instance}}$

$$
\begin{array}{lcl}
[\![U_1; U_2]\!]_{\sigma,\Delta} & = & [\![U_2]\!]_{\sigma,\Delta'} \text{ where } \Delta' = [\![U_1]\!]_{\sigma,\Delta} \\
[\![\texttt{ins}(R,t)]\!]_{\sigma,\Delta} & = & \Delta[R \leftarrow \mathit{append}(\Delta(R), t[\sigma])] \\
[\![\texttt{del}(R,\phi)]\!]_{\sigma,\Delta} & = & \Delta[R \leftarrow \mathit{filter}(\Delta(R), \lambda x.\neg[\![\phi]\!]_{\sigma,\Delta,x})] \\
[\![\texttt{upd}(R,\phi,a,v)]\!]_{\sigma,\Delta} & = & \Delta\Big[R \leftarrow \mathit{append}\begin{pmatrix} \mathit{filter}(\Delta(R), \lambda x.\neg[\![\phi]\!]_{\sigma,\Delta,x}), \\ \mathit{map}(\mathit{filter}(\Delta(R), \lambda x.[\![\phi]\!]_{\sigma,\Delta,x}), \lambda x.x[a \leftarrow v[\sigma]]) \end{pmatrix}\Big]
\end{array}
$$

$\boxed{[\![Q]\!] :: \textit{Valuation } \sigma \to \textit{Instance } \Delta \to \textit{Relation}}$

$$
\begin{array}{lcl}
[\![R]\!]_{\sigma,\Delta} & = & \Delta(R) \\
[\![\Pi_\psi(Q)]\!]_{\sigma,\Delta} & = & \mathit{map}([\![Q]\!]_{\sigma,\Delta}, \lambda x.\mathit{filter}(x, \lambda y.\mathit{contains}(\mathit{first}(y), \psi))) \\
[\![\sigma_\phi(Q)]\!]_{\sigma,\Delta} & = & \mathit{filter}([\![Q]\!]_{\sigma,\Delta}, \lambda x.[\![\phi]\!]_{\sigma,\Delta,x}) \\
[\![Q_1 \times Q_2]\!]_{\sigma,\Delta} & = & \mathit{foldl}(\lambda ys.\lambda y.\mathit{append}(ys, \mathit{map}([\![Q_2]\!]_{\sigma,\Delta}, \lambda z.\mathit{merge}(y,z))), [\,], [\![Q_1]\!]_{\sigma,\Delta}) \\
[\![Q_1 \bowtie_\phi Q_2]\!]_{\sigma,\Delta} & = & [\![\sigma_\phi(Q_1 \times Q_2)]\!]_{\sigma,\Delta} \\
[\![Q_1 \cup Q_2]\!]_{\sigma,\Delta} & = & \mathit{append}([\![Q_1]\!]_{\sigma,\Delta}, [\![Q_2]\!]_{\sigma,\Delta}) \\
[\![Q_1 - Q_2]\!]_{\sigma,\Delta} & = & \mathit{foldl}\ (\lambda ys.\lambda y.\mathit{delete}(y, ys), [\![Q_1]\!]_{\sigma,\Delta}, [\![Q_2]\!]_{\sigma,\Delta})
\end{array}
$$

$\boxed{[\![\phi]\!] :: \textit{Valuation } \sigma \to \textit{Instance } \Delta \to \textit{Tuple } x \to \textit{Predicate}}$

$$
\begin{array}{lcl}
[\![a_1 \odot a_2]\!]_{\sigma,\Delta,x} & = & \mathit{lookup}(x, a_1) \odot \mathit{lookup}(x, a_2) \\
[\![a \odot v]\!]_{\sigma,\Delta,x} & = & \mathit{lookup}(x, a) \odot v[\sigma] \\
[\![a \in Q]\!]_{\sigma,\Delta,x} & = & \mathit{contains}(\mathit{lookup}(x, a),\ \mathit{map}([\![Q]\!]_{\sigma,\Delta}, \lambda y.\mathit{head}(\mathit{vals}(y)))) \\
[\![\phi_1 \wedge \phi_2]\!]_{\sigma,\Delta,x} & = & [\![\phi_1]\!]_{\sigma,\Delta,x} \wedge [\![\phi_2]\!]_{\sigma,\Delta,x} \\
[\![\phi_1 \vee \phi_2]\!]_{\sigma,\Delta,x} & = & [\![\phi_1]\!]_{\sigma,\Delta,x} \vee [\![\phi_2]\!]_{\sigma,\Delta,x} \\
[\![\neg\phi]\!]_{\sigma,\Delta,x} & = & \neg[\![\phi]\!]_{\sigma,\Delta,x}
\end{array}
$$

Figure 2.5: Denotational semantics of database programs.

In our semantics, we model *database instances* Δ as a mapping from relation names to a list of tuples.[2] Similarly, we model tuples as a mapping from attribute names to their corresponding values. Given a program $\mathcal{P}$, an invocation sequence $\omega = \omega'; (n, \sigma)$, and a database instance Δ, we first obtain a new instance Δ' by running ω' on Δ and then evaluate the query Q_n on database instance Δ' and valuation σ. Observe that the result of a program is represented as a list of lists rather than as relations (list of maps). That is, our semantics disregards the names of attributes to enable meaningful comparison between programs over different schemas.

The semantics for update functions in Figure 2.5 are described using the familiar list combinators, such as *append*, *filter*, *map*, and *fold*. In particular, $[\![U]\!]_{\sigma,\Delta}$ yields the database instance after executing update operation U with input σ on database Δ. For example, consider the semantics for $\texttt{ins}(R, t)$: To obtain the new database instance, we first evaluate t under valuation σ, where the notation $t[\sigma]$ denotes applying substitution σ to term t. The entry for relation R in the new database instance is obtained by appending the tuple $t[\sigma]$ to table $\Delta(R)$, which is represented as a list of tuples. Similarly, $\texttt{del}(R, \phi)$ filters from list R the set of all tuples that do not satisfy predicate ϕ. Finally, $\texttt{upd}(R, \phi, a, v)$ first obtains a new relation R_1 that contains all tuples in R that do not satisfy ϕ. It then also filters out all tuples of R that satisfy predicate ϕ, and updates the a attribute of each such tuple to a new value $v[\sigma]$. The

[2]We model relations as lists rather than bags because many libraries provide database interfaces based on ordered data structures.

new entry for R is then obtained by concatenating these two lists.

Let us now turn our attention to the semantics of query functions (third part of Figure 2.5). The semantics are defined inductively, with the first rule for R being the base case. Since R corresponds to the name of a database table, we obtain the query result by simply looking up R in Δ. As another example, consider the semantics of the selection (σ) operator. Given a query of the form $\sigma_\phi(Q)$, we first recursively evaluate Q and obtain the query result $T = [\![Q]\!]_{\sigma,\Delta}$. We then evaluate the predicate ϕ under σ and obtain a *symbolic* predicate $p = [\![\phi]\!]_{\sigma,\Delta,x}$. In particular, predicate p is symbolic in the sense that it refers to a variable x, which ranges over tuples in T. We obtain the final query result by filtering out those rows of T that do not satisfy predicate $\lambda x.p$.

The final part of Figure 2.5 describes predicate semantics inductively. For instance, consider a predicate of the form $a \odot v$, where a is an attribute, $\odot$ is a (logical) binary operator, and v is a symbol (variable or constant). Since the predicate takes as input a tuple x, we evaluate attribute a by looking up a in x, which is represented as a mapping from attributes to values. Thus, the evaluation of the predicate is given by $\mathit{lookup}(x, a) \odot v[\sigma]$, where the notation $v[\sigma]$ applies substitution σ to symbol v.

2.2.3 Equivalence and Refinement

Having defined the semantics of database programs, we are now ready to precisely state the notion of semantic equivalence in this context:

Definition 2.2.1. (Program equivalence) A database program $\mathcal{P}'$ is said

to be semantically equivalent to another program $\mathcal{P}$, denoted $\mathcal{P}' \simeq \mathcal{P}$, if and only if executing ω on $\mathcal{P}'$ yields the same result as executing ω on $\mathcal{P}$ for any invocation sequence ω, i.e.,

$$\mathcal{P}' \simeq \mathcal{P} \triangleq \forall \omega.\ [\![\mathcal{P}']\!]_\omega = [\![\mathcal{P}]\!]_\omega$$

In the above definition, we assume that $\mathcal{P}$ and $\mathcal{P}'$ have the same number of query and update functions. If this condition does not hold, we can immediately conclude that $\mathcal{P}$ and $\mathcal{P}'$ are not equivalent because some inputs that are valid for $\mathcal{P}$ are not valid for $\mathcal{P}'$ or vice versa. We also assume that $\mathcal{P}$ and $\mathcal{P}'$ have functions that are supposed to be functionally equivalent at the same index; otherwise, the functions can be syntactically re-arranged. Under these assumptions, our definition effectively states that programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent whenever their corresponding queries yield the same result under the same sequence of update functions to the database.

While there are many real-world scenarios in which we would like to prove equivalence, there are also some cases where one program *refines* the other. For instance, consider a situation in which a web application developer changes the database schema for performance reasons, but also decides to add some new piece of information to the underlying database such that query results also include this new information. In this scenario, the updated version of the application will *not* be semantically equivalent to its prior version, but we would still like to verify that adding new features does not break existing functionality. Towards this goal, we also formally define what it means for a

database program $\mathcal{P}'$ to *refine* another program $\mathcal{P}$.

Definition 2.2.2. (Valuation refinement) Consider two valuations σ and σ'. We say that σ' is a refinement of σ, denoted $\sigma' \preceq \sigma$, if and only if σ' maps the variables that occur in σ to the same values as in σ. In other words,

$$\sigma' \preceq \sigma \triangleq \forall x \in dom(\sigma).\ \sigma'(x) = \sigma(x)$$

Definition 2.2.3. (Input refinement) Given invocation sequences $\omega = (i_1, \sigma_1)(i_2, \sigma_2)\dots(i_n, \sigma_n)$ and $\omega' = (i'_1, \sigma'_1)(i'_2, \sigma'_2)\dots(i'_n, \sigma'_n)$, we say that ω' refines ω, denoted $\omega' \preceq \omega$, if and only if ω' has the same index sequence as ω and the valuations in ω' refine the corresponding valuations in ω. That is,

$$\omega' \preceq \omega \triangleq \forall k \in [1, n].\ i_k = i'_k \wedge \sigma'_k \preceq \sigma_k$$

Using these definitions, we can now also state what it means for a program to refine another one:

Definition 2.2.4. (Program refinement) Program $\mathcal{P}'$ refines another program $\mathcal{P}$, denoted $\mathcal{P}' \preceq \mathcal{P}$, if and only if, for any invocation sequences ω', ω satisfying $\omega' \preceq \omega$, executing ω on $\mathcal{P}$ yields a relation that is a projection of executing ω' on $\mathcal{P}'$, i.e.,

$$\mathcal{P}' \preceq \mathcal{P} \triangleq \forall \omega, \omega'.\ \omega' \preceq \omega \rightarrow \exists L.\ [\![\Pi_L(\mathcal{P}')]\!]_{\omega'} = [\![\mathcal{P}]\!]_{\omega}$$

where $\Pi_L((\mathcal{S}, \vec{T}_U, \vec{T}_Q)) = (\mathcal{S}, \vec{T}_U, \vec{T}'_Q)$ such that $Q'_i = \Pi_L(Q_i)$ for all Q_i in $\vec{T}_Q$.

As in Definition 2.2.1, we require that ω' and ω are valid inputs for $\mathcal{P}'$ and $\mathcal{P}$ respectively. However, unlike in Definition 2.2.1, we do not assume

that $\mathcal{P}'$ contains the same number of update and query functions in $\mathcal{P}$. Our definition simply disregards the new functions that are added by $\mathcal{P}'$ and only considers invocation sequences that are valid for both. Thus, intuitively, if an application $\mathcal{P}'$ refines $\mathcal{P}$, the query results of $\mathcal{P}$ can be obtained by applying a projection to the corresponding query results in $\mathcal{P}'$.

2.3 Proof Methodology

Having defined the semantic equivalence and refinement properties for database programs, let us now turn our attention to the proof methodology for showing these properties.

2.3.1 Proving Equivalence

A standard methodology for proving equivalence between any two systems A, B is to find a *bisimulation relation* that relates states in A with those in B [35]. In our case, these systems are database-driven applications, and the states that we need to relate are database instances. Our approach does not directly infer an explicit mapping between database instances, but instead finds a *bisimulation invariant* that (a) is satisfied by pairs of database instances from the two systems, and (b) is strong enough to prove equivalence.

In this chapter, we prove that a bisimulation invariant Φ is valid by showing that it is inductive. That is, Φ must hold initially, and assuming that it holds for a pair of databases Δ, Δ', it must continue to hold after executing any pair of corresponding update operations $\lambda\vec{x}.U$ and $\lambda\vec{y}.U'$.

Definition 2.3.1. (Inductive bisimulation invariant) Consider programs $\mathcal{P} = (\mathcal{S}, \vec{T}_U, \vec{T}_Q)$ and $\mathcal{P}' = (\mathcal{S}', \vec{T}'_U, \vec{T}'_Q)$ and suppose that $\mathcal{P}, \mathcal{P}'$ contain a disjoint set of variables (which can be enforced using α-renaming if necessary). A bisimulation invariant Φ is said to be *inductive* with respect to programs $\mathcal{P}$ and $\mathcal{P}'$ if (a) Φ is satisfied by a pair of empty databases, and (b) the following Hoare triple is valid for all $\lambda\vec{x}.U_i \in \vec{T}_U$ and $\lambda\vec{y}.U'_i \in \vec{T}'_U$:

$$\{\Phi \wedge \vec{x} = \vec{y}\}\ U_i \mid\mid U'_i\ \{\Phi\}$$

In the above definition, the notation $U||U'$ denotes the parallel execution of updates $\lambda\vec{x}.U$ and $\lambda\vec{y}.U'$. However, since programs $\mathcal{P}, \mathcal{P}'$ contain a disjoint set of variables, $U||U'$ is semantically equivalent to the sequential composition $U; U'$. Thus, to prove inductiveness, we need to show the validity of the Hoare triple

$$\{\Phi \wedge \vec{x} = \vec{y}\}\ U_i\ ;\ U'_i\ \{\Phi\}$$

for every pair of updates $\lambda\vec{x}.U_i$ and $\lambda\vec{y}.U'_i$ in $\mathcal{P}$ and $\mathcal{P}'$. Also, observe that Φ must hold for a pair of empty databases in the base case because we assume that the databases are initially empty (recall Section 2.2).[3]

While there are many possible inductive bisimulation invariants (including *true*, for example), we need a bisimulation invariant that is strong enough to prove equivalence. According to Definition 2.2.1, two programs are

[3]This assumption is realistic in situations where database migration is performed by calling the new update functions in the application. Otherwise, the base case needs to establish that Φ holds for the initial databases.

equivalent if they yield the same result for every pair of corresponding queries $\lambda\vec{x}.Q$ and $\lambda\vec{y}.Q'$ given the same input. Thus, we can define what it means for a bisimulation invariant to be sufficient in the following way:

Definition 2.3.2. (Sufficiency) A formula Φ is said to be *sufficient* with respect to programs $\mathcal{P} = (\mathcal{S}, \vec{T}_U, \vec{T}_Q)$ and $\mathcal{P}' = (\mathcal{S}', \vec{T}'_U, \vec{T}'_Q)$ if, for all $\lambda\vec{x}.Q_i \in \vec{T}_Q$ and $\lambda\vec{y}.Q'_i \in \vec{T}'_Q$, we have:

$$(\Phi \wedge \vec{x} = \vec{y}) \models Q_i = Q'_i$$

Our general proof methodology for proving equivalence is to find a *sufficient, inductive bisimulation invariant* between the given pair of programs. If we can find such an invariant Φ, we know that Φ holds after executing any invocation sequence ω on $\mathcal{P}, \mathcal{P}'$, so Φ must also hold before issuing any database query. Furthermore, since Φ is a sufficient bisimulation invariant, it implies that any pair of queries yield the same result. Thus, the existence of such a bisimulation invariant Φ implies that $\mathcal{P}, \mathcal{P}'$ are semantically equivalent.

Theorem 2.3.1. ***(Soundness)*** *Given database programs $\mathcal{P}, \mathcal{P}'$, the existence of a sufficient, inductive bisimulation invariant Φ implies $\mathcal{P} \simeq \mathcal{P}'$.*

Proof. See [129]. □

Theorem 2.3.2. ***(Relative Completeness)*** *Suppose we have an oracle for proving any valid Hoare triple and logical entailment. If $\mathcal{P} \simeq \mathcal{P}'$, then there always exists a sufficient, inductive bisimulation invariant Φ for programs $\mathcal{P}, \mathcal{P}'$.*

Proof. See [129]. □

2.3.2 Proving Refinement

Since our notion of refinement is a generalization of equivalence (recall Definition 2.2.4), our proof methodology for showing program refinement closely follows that for verifying equivalence. In particular, rather than finding a one-to-one mapping between database states as in the case of equivalence, it suffices to find a one-to-many mapping for showing refinement. Hence, our proof methodology relies on finding a *simulation invariant* rather than a stronger bisimulation invariant:

Definition 2.3.3. (Inductive simulation invariant) Consider programs $\mathcal{P} = (\mathcal{S}, \vec{T}_U, \vec{T}_Q)$ and $\mathcal{P}' = (\mathcal{S}', \vec{T}'_U, \vec{T}'_Q)$ and suppose that $\mathcal{P}, \mathcal{P}'$ contain a disjoint set of variables. A *simulation invariant* Φ is said to be *inductive* with respect to programs $\mathcal{P}$ and $\mathcal{P}'$ if (a) Φ is satisfied by a pair of empty databases, and (b) the following Hoare triple is valid for all $\lambda\vec{x}.U_i \in \vec{T}_U$ and $\lambda\vec{y}.U'_i \in \vec{T}'_U$ where $i \in [1, |\vec{T}_U|]$:

$$\Big\{\Phi \wedge \bigwedge_{x_j \in \vec{x}} x_j = y_j\Big\} \;\; U_i \;;\; U'_i \;\; \Big\{\Phi\Big\}$$

Recall from Definition 2.2.4 that we allow the second program $\mathcal{P}'$ to contain more functions than $\mathcal{P}$, but the notion of refinement only talks about invocation sequences that use shared functions from $\mathcal{P}$ and $\mathcal{P}'$. Therefore, in Definition 2.3.3, we only require Φ to be preserved by pairs of update functions that are both present in $\mathcal{P}$ and $\mathcal{P}'$. Furthermore, since functions in $\mathcal{P}'$ can take additional arguments not present in their counterparts in $\mathcal{P}$, our precondition states that the arguments are pairwise equal for only the "shared" variables.

As in the equivalence scenario, finding an inductive simulation invariant Φ between $\mathcal{P}$ and $\mathcal{P}'$ is *not* sufficient for proving that $\mathcal{P}'$ refines $\mathcal{P}$, as Φ may not be strong enough to show refinement. Hence, we also need to define what it means for an inductive simulation invariant to be *sufficient* for showing refinement. However, since the notion of refinement is weaker than equivalence, we also weaken our corresponding notion of sufficiency as follows:

Definition 2.3.4. (Projective sufficiency) A formula Φ is said to be *projectively sufficient* with respect to programs $\mathcal{P} = (\mathcal{S}, \vec{T}_U, \vec{T}_Q)$ and $\mathcal{P}' = (\mathcal{S}', \vec{T}'_U, \vec{T}'_Q)$ if for all $\lambda\vec{x}.Q_i \in \vec{T}_Q$ and $\lambda\vec{y}.Q'_i \in \vec{T}'_Q$ where $i \in [1, |\vec{T}_Q|]$, we have:

$$\Big(\Phi \wedge \bigwedge_{x_j \in \vec{x}} x_j = y_j\Big) \models \exists L.\ Q_i = \Pi_L(Q'_i)$$

Observe that the notion of projective sufficiency is weaker than Definition 2.3.2, as we do not require Q_i and Q'_i to yield exactly the same relation and allow the result of Q'_i to contain attributes not present in the result of Q_i. Our general proof methodology for proving refinement then relies on finding a simulation invariant that is both inductive and projectively sufficient.

Theorem 2.3.3. ***(Soundness)*** *Given database applications $\mathcal{P}, \mathcal{P}'$, the existence of a projectively sufficient and inductive simulation invariant Φ implies $P' \preceq P$.*

Proof. See [129]. □

Theorem 2.3.4. ***(Relative Completeness)*** *Suppose we have an oracle for proving any valid Hoare triple and logical entailment. If $P' \preceq P$, then there*

$$
\begin{array}{rrcl}
\textit{Formula} & F & := & \textit{true} \mid \textit{false} \mid t = t \mid F \wedge F \mid F \vee F \\
& & \mid & \neg F \mid F \rightarrow F \mid \exists x.F \mid \forall x.F \\
\textit{Term} & t & := & x \mid T \mid \Pi_{a_1,\ldots,a_n}(t) \mid \sigma_{\phi}(t) \mid t \times t \mid t \cup t \mid t - t \mid t\langle a_i \triangleleft v \rangle \\
\textit{Predicate} & \phi & := & a_i \odot a_j \mid a_i \odot v \mid a_i \in t \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\textit{BinOp} & \odot & := & \leq \mid < \mid = \mid \neq \mid > \mid \geq
\end{array}
$$

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
T & \in & \textit{Table} & a_i & \in & \textit{Attribute} \\
v & \in & \textit{Constant} \cup \textit{Variable} & x & \in & \textit{Variable}
\end{array}
$$

Figure 2.6: Formula in Theory of Relational Algebra with Updates.

always exists a projectively sufficient and inductive simulation invariant Φ *for programs* $\mathcal{P}, \mathcal{P}'$.

Proof. See [129]. □

2.4 SMT Encoding of Relational Algebra with Updates

In the previous section, we defined what it means for simulation and bisimulation invariants to be inductive, but we have not fixed a logical theory over which we express these invariants. In this section, we discuss the theory of relational algebra with updates, $\mathcal{T}_{RA}$, and show how to enable reasoning in $\mathcal{T}_{RA}$ using existing SMT solvers.

Figure 2.6 gives the syntax of the theory of relational algebra with updates $\mathcal{T}_{RA}$, which we use to express simulation and bisimulation invariants. The notation a_i represents the i'th attribute in a relation. Atomic formulas in $\mathcal{T}_{RA}$ are of the form $t_1 = t_2$ where t_1 and t_2 are terms representing relations. Basic terms include variables x and concrete tables T, and more complex

terms can be formed using the relational algebra operators Π (projection), σ (selection), $\times$ (Cartesian product), $\cup$ (union), and $-$ (difference). In addition to these standard relational algebra operators, $\mathcal{T}_{RA}$ also includes an *update* operator, denoted as $t\langle a_i \triangleleft v\rangle$, which represents the new relation after changing the i-th attribute of all tuples in t to v. Observe that the theta join operator $\bowtie_\phi$ is expressible in this logic as $\sigma_\phi(t_1 \times t_2)$. We also write $t_1 \bowtie t_2$ as syntactic sugar for $\sigma_\phi(t_1 \times t_2)$ where ϕ is a predicate stating that the shared attributes of t_1 and t_2 are equal.

Since we view tables as lists of tuples, we axiomatize $\mathcal{T}_{RA}$ using the theory of lists [18]. Our axiomatization is presented in Figure 2.7 in the form of inference rules, where we view tuples as *lists* of values and relations as *lists* of tuples. [ ] represents an empty list `nil`. The binary operator :: denotes the list constructor `cons`. $\times'$ and `cat` are auxiliary functions for axiomatizing Cartesian product. Π', $-'$, and `upd` are auxiliary functions for axiomatizing projection, minus, and update, respectively. An attribute a_i of a tuple is simply an index into the list representing that tuple. For example, consider the axioms for *projection* $\Pi_l(t)$, which projects term t given attribute list l. We first define an auxiliary function $\Pi'_l(h)$ that projects a single tuple h given l. In particular, if the attribute list l is empty, then $\Pi'_l(h)$ yields [ ]. Otherwise, if l consists of head a_i and tail l_1, $\Pi'_l(h)$ composes the i-th value of h (i.e., $\texttt{get}(i, h)$) and the projection of tail $\Pi'_{l_1}(h)$. Similarly, $\Pi_l(t)$ is also recursively defined. If $t = [\,]$, then $\Pi_l(t) = [\,]$. Otherwise if $t = h :: t_1$, then $\Pi_l(t)$ composes the projection $\Pi'_l(h)$ of head h and the projection $\Pi_l(t_1)$ of tail t_1. Also, observe that the

get $\mathtt{get}(i,l)$

$$\frac{l = h :: t \quad i = 0}{\mathtt{get}(i,l) = h} \qquad \frac{l = h :: t \quad i \neq 0}{\mathtt{get}(i,l) = \mathtt{get}(i-1,t)}$$

projection $\Pi_l(t)$

$$\frac{t = [\,]}{\Pi_l(t) = [\,]} \qquad \frac{t = h :: t_1}{\Pi_l(t) = \Pi'_l(h) :: \Pi_l(t_1)}$$

$$\frac{l = [\,]}{\Pi'_l(h) = [\,]} \qquad \frac{l = a_i :: l_1}{\Pi'_l(h) = \mathtt{get}(i,h) :: \Pi'_{l_1}(h)}$$

union $t_1 \cup t_2$

$$\frac{t_1 = [\,]}{t_1 \cup t_2 = t_2} \qquad \frac{t_1 = h :: t}{t_1 \cup t_2 = h :: (t \cup t_2)}$$

minus $t_1 - t_2$

$$\frac{t_2 = [\,]}{t_1 - t_2 = t_1} \qquad \frac{t_2 = h_2 :: t}{t_1 - t_2 = (t_1 -' h_2) - t}$$

$$\frac{t_1 = [\,]}{t_1 -' h_2 = [\,]} \qquad \frac{t_1 = h_1 :: t_3 \quad h_1 = h_2}{t_1 -' h_2 = t_3}$$

$$\frac{t_1 = h_1 :: t_3 \quad h_1 \neq h_2}{t_1 -' h_2 = h_1 :: (t_3 -' h_2)}$$

selection $\sigma_\phi(t)$

$$\frac{t = [\,]}{\sigma_\phi(t) = [\,]} \qquad \frac{\phi(h) \quad t = h :: t_1}{\sigma_\phi(t) = h :: \sigma_\phi(t_1)}$$

$$\frac{\neg\phi(h) \quad t = h :: t_1}{\sigma_\phi(t) = \sigma_\phi(t_1)}$$

Cartesian product $t_1 \times t_2$

$$\frac{t_1 = [\,]}{t_1 \times t_2 = [\,]} \qquad \frac{t_1 = h_1 :: t}{t_1 \times t_2 = (h_1 \times' t_2) \cup (t \times t_2)}$$

$$\frac{t_2 = [\,]}{h_1 \times' t_2 = [\,]} \qquad \frac{t_2 = h_2 :: t_3}{h_1 \times' t_2 = \mathtt{cat}(h_1,h_2) :: h_1 \times' t_3}$$

$$\frac{h_1 = [\,]}{\mathtt{cat}(h_1,h_2) = h_2} \qquad \frac{h_1 = c :: h}{\mathtt{cat}(h_1,h_2) = c :: \mathtt{cat}(h,h_2)}$$

update $t\langle a_i \triangleleft v\rangle$

$$\frac{t = [\,]}{t\langle a_i \triangleleft v\rangle = [\,]} \qquad \frac{t = h :: t_1}{t\langle a_i \triangleleft v\rangle = \mathtt{upd}(h,i,v) :: t_1\langle a_i \triangleleft v\rangle}$$

$$\frac{h = [\,]}{\mathtt{upd}(h,i,v) = [\,]} \qquad \frac{h = c :: h_1 \quad i = 0}{\mathtt{upd}(h,i,v) = v :: h_1}$$

$$\frac{h = c :: h_1 \quad i \neq 0}{\mathtt{upd}(h,i,v) = c :: \mathtt{upd}(h_1, i-1, v)}$$

Figure 2.7: Axioms in Theory of Relation Algebra with Updates.

$$\begin{aligned}
(a_i \odot a_j)(h) &= \mathtt{get}(i,h) \odot \mathtt{get}(j,h) \\
(a_i \odot v)(h) &= \mathtt{get}(i,h) \odot v \\
(a_i \in t)(h) &= \exists j.\ \mathtt{get}(0, \mathtt{get}(j,t)) = \mathtt{get}(i,h) \\
(\phi_1 \wedge \phi_2)(h) &= \phi_1(h) \wedge \phi_2(h) \\
(\phi_1 \vee \phi_2)(h) &= \phi_1(h) \vee \phi_2(h) \\
(\neg\phi)(h) &= \neg\phi(h)
\end{aligned}$$

Figure 2.8: Auxiliary functions for selection axiom schema $\phi(h)$.

rules for selection in Figure 2.7 actually correspond to *axiom schemata* rather than axioms: Because the selection operator is parameterized over a predicate ϕ, this schema needs to be instantiated for each predicate that occurs in the formula.

Example 2.4.1. *Consider the formula $\sigma_{a_1 \geq 2}(x) = \sigma_{a_2 > 1}(y)$ in the theory of relational algebra with updates. We generate the following axioms using the axiom schemata for selection:*

$$(1a) \quad \forall x.\ (x = [\,]) \rightarrow \sigma_{a_1 \geq 2}(x) = [\,]$$

$$(1b) \quad \forall x, h, t.\ \Big(x = h :: t \rightarrow \begin{pmatrix} (\mathrm{get}(1,h) \geq 2 \rightarrow \sigma_{a_1 \geq 2}(x) = h :: \sigma_{a_1 \geq 2}(t)) \wedge \\ (\neg(\mathrm{get}(1,h) \geq 2) \rightarrow \sigma_{a_1 \geq 2}(x) = \sigma_{a_1 \geq 2}(t)) \end{pmatrix}\Big)$$

$$(2a) \quad \forall y.\ (y = [\,]) \rightarrow \sigma_{a_2 > 1}(y) = [\,]$$

$$(2b) \quad \forall y, h, t.\ \Big(y = h :: t \rightarrow \begin{pmatrix} (\mathrm{get}(2,h) > 1 \rightarrow \sigma_{a_2 > 1}(y) = h :: \sigma_{a_2 > 1}(t)) \wedge \\ (\neg(\mathrm{get}(2,h) > 1) \rightarrow \sigma_{a_2 > 1}(y) = \sigma_{a_2 > 1}(t)) \end{pmatrix}\Big)$$

Remark 2.4.1. Since the problem of checking equivalence between a pair of relational algebra expressions is known to be undecidable [124], our theory of relational algebra with updates is also undecidable. However, with the aid of some optimizations that we discuss in Section 2.6, we are able to determine the validity of most $\mathcal{T}_{RA}$ formulas that we encounter in practice.

2.5 Automated Verification

So far, we have explained our general proof methodology and introduced a first-order theory in which we will express our bisimulation invariants. However, we have not yet explained how to *automatically* prove equivalence between programs. In this section, we discuss our strategy for proof automation. Specifically, we first discuss how to automatically prove equivalence

assuming that an oracle provides bisimulation invariants (Section 2.5.1), and then explain how we infer them automatically (Section 2.5.2). Because the automation of refinement checking is very similar, this section only addresses equivalence.

2.5.1 Automation for Bisimulation Invariant Inductiveness

Consider two database programs $\mathcal{P} = (\mathcal{S}, \vec{T}_U, \vec{T}_Q)$ and $\mathcal{P}' = (\mathcal{S}', \vec{T}'_U, \vec{T}'_Q)$, and suppose that an oracle provides a bisimulation invariant Φ between $\mathcal{P}$ and $\mathcal{P}'$. Based on the proof methodology we outlined in Section 2.3, we can prove that $\mathcal{P}, \mathcal{P}'$ are equivalent by showing that Φ satisfies the following conditions for any pair of updates $\lambda\vec{x}.U_i$, $\lambda\vec{y}.U'_i$ and any pair of queries $\lambda\vec{x}.Q_i$, $\lambda\vec{y}.Q'_i$:

$$\begin{array}{ll} (1) & \Phi \wedge \vec{x} = \vec{y} \models Q_i = Q'_i \quad \textit{(Sufficiency)} \\ (2) & \{\Phi \wedge \vec{x} = \vec{y}\}\ U_i; U'_i\ \{\Phi\} \quad \textit{(Inductiveness)} \end{array}$$

The first condition (sufficiency) is easy to prove since we have already defined a logical theory that allows us to write terms of the form $Q_i = Q'_i$. The only small technical hiccup is that $\mathcal{T}_{RA}$ uses the syntax a_i to denote the i-th attribute in a relation, whereas attributes in the queries Q_i, Q'_i are *names* of attributes. To solve this difficulty, we assume a function ς which replaces attribute names s in constructs from Figure 2.3 with a_i, where i is the index of s. Thus, we can check whether formula Φ satisfies condition (1) by querying whether the following formula is valid modulo $\mathcal{T}_{RA}$:

$$(\Phi \wedge \vec{x} = \vec{y}) \rightarrow \varsigma(Q_i) = \varsigma(Q'_i)$$

$$
\begin{array}{rcl}
sp(\Phi, \texttt{ins}(R, \{a_1 : v_1, \ldots, a_n : v_n\})) & = & \exists x.\ (R = x \cup [r]) \wedge \Phi[x/R] \ \textit{where}\ r = [v_1, \ldots, v_n] \\
sp(\Phi, \texttt{del}(R, \phi)) & = & \exists x.\ (R = \sigma_{\varsigma(\neg\phi)}(x)) \wedge \Phi[x/R] \\
sp(\Phi, \texttt{upd}(R, \phi, a, v)) & = & \exists x.\ (R = \sigma_{\varsigma(\neg\phi)}(x) \cup \sigma_{\varsigma(\phi)}(x)\langle\varsigma(a) \triangleleft v\rangle) \wedge \Phi[x/R] \\
sp(\Phi, U_1; U_2) & = & sp(sp(\Phi, U_1), U_2)
\end{array}
$$

Figure 2.9: Strongest postcondition for update functions.

However, to prove the second condition (i.e., inductiveness), we need a way to prove Hoare triples for update statements U from Figure 2.3. Towards this goal, we define a *strongest post-condition* semantics for update statements. Given a formula Φ over $\mathcal{T}_{RA}$ and an update statement U, Figure 2.9 describes the computation of $sp(\Phi, U)$, which represents the strongest post-condition of Φ with respect to statement U.

To compute the strongest postcondition of Φ with respect to $\texttt{ins}(R, \{a_1 : v_1, \ldots, a_n : v_n\})$, we think of the insertion as the assignment $R := \mathit{append}(R, [r])$ where r is the tuple (list) $[v_1, \ldots, v_n]$. Since the union operator $\cup$ in $\mathcal{T}_{RA}$ corresponds to list concatenation, the new value of R is given by $x \cup [r]$, where the existentially quantified variable x represents the old value of R.

To understand the strongest postcondition semantics of deletion, recall that $\texttt{del}(R, \phi)$ removes all rows in R that satisfy ϕ. Hence, we can model this statement using the assignment $R := \sigma_{\neg\phi}(R)$. Thus, when we compute the strongest postcondition of ϕ with respect to $\texttt{del}(R, \phi)$, the new value of R is given by $\sigma_{\varsigma(\neg\phi)}(x)$ where the existentially quantified variable x again represents the old value of R and $\varsigma(\phi)$ replaces attribute names in ϕ with their corresponding indices.

Finally, let us consider the strongest postcondition for update statements of the form $\texttt{upd}(R, \phi, a, v)$. Recall that this statement assigns value v to the a attribute of all tuples in R that satisfy ϕ. Specifically, according to the denotational semantics from Figure 2.5, we can model $\texttt{upd}(R, \phi, a, v)$ using the assignment statement:

$$R \;:=\; (\sigma_{\neg\phi}(R)) \cup (\sigma_{\phi}(R))\langle a \triangleleft v \rangle$$

Hence, we obtain the strongest postcondition of Φ with respect to $\texttt{upd}(R, \phi, a, v)$ by computing the strongest postcondition of the above assignment, where the right-hand side is a term in $\mathcal{T}_{RA}$ (modulo changing attribute names to indices).

Definition 2.5.1. (Agreement) Consider database instance Δ and valuation σ, and let $\varsigma(\Delta)$ denote the representation of Δ where each tuple $\{a_1 : v_1, \ldots, a_n : v_n\}$ is represented as the list $[v_1, \ldots, v_n]$. We say that (Δ, σ) agrees with $\mathcal{T}_{RA}$-formula Φ, written $(\Delta, \sigma) \sim \Phi$, iff $\varsigma(\Delta) \uplus \sigma \models \Phi$.

Theorem 2.5.1. (Soundness of sp) *Suppose that* $[\![U]\!]_{\sigma,\Delta} = \Delta'$*, and let* Φ *be a* $\mathcal{T}_{\mathrm{RA}}$ *formula. If* $(\Delta, \sigma) \sim \Phi$*, then we also have* $(\Delta', \sigma) \sim \mathrm{sp}(\Phi, U)$*.*

Proof. See [129]. □

Now that we have defined a strongest post-condition semantics for update statements in our language, it is easy to check the correctness of the Hoare triple $\{\Phi \wedge \vec{x} = \vec{y}\}\ U_i; U_i'\ \{\Phi\}$ by simply querying the validity of the following formula modulo $\mathcal{T}_{RA}$:

$$sp(\Phi \wedge \vec{x} = \vec{y},\ U_i; U_i') \rightarrow \Phi$$

Algorithm 1 Verification Algorithm for Database Program Equivalence

1: **procedure** VERIFY($\mathcal{P}$, $\mathcal{P}'$)
2: **Input:** program $\mathcal{P} = (\mathcal{S}, \vec{T}_U, \vec{T}_Q)$ and $\mathcal{P}' = (\mathcal{S}', \vec{T}'_U, \vec{T}'_Q)$
3: **Output:** Invariant Φ to establish equivalence or $\bot$ to indicate failure
4: $\mathcal{U} :=$ GetAllPredicates($\mathcal{S}$, $\mathcal{S}'$);
5: $\Phi := \bigwedge_{\varphi \in \mathcal{U}} \varphi$;
6: **while** CHECKSUFFICIENCY(Φ, $\vec{T}_Q$, $\vec{T}'_Q$) **do**
7: ind := *true*;
8: **for each** $\varphi \in \mathcal{U}$ **do**
9: **if** $\neg$CHECKINDUCTIVENESS(Φ, $\vec{T}_U$, $\vec{T}'_U$, φ) **then**
10: ind := *false*; $\mathcal{U} := \mathcal{U} \setminus \{\varphi\}$; $\Phi := \bigwedge_{\varphi \in \mathcal{U}} \varphi$;
11: **break**;
12: **if** ind **then return** Φ;
13: **return** $\bot$;

14: **procedure** CHECKSUFFICIENCY(Φ, $\vec{T}_Q$, $\vec{T}'_Q$)
15: **for each** $\lambda\vec{x}.Q_i \in \vec{T}_Q$ and $\lambda\vec{y}.Q'_i \in \vec{T}'_Q$ **do**
16: **if** $\mathcal{T}_{RA} \not\models (\Phi \wedge \vec{x} = \vec{y} \rightarrow \varsigma(Q_i) = \varsigma(Q'_i))$ **then return** *false*;
17: **return** *true*;

18: **procedure** CHECKINDUCTIVENESS(Φ, $\vec{T}_U$, $\vec{T}'_U$, φ)
19: **for each** $\lambda\vec{x}.U_i \in \vec{T}_U$ and $\lambda\vec{y}.U'_i \in \vec{T}'_U$ **do**
20: **if** $\mathcal{T}_{RA} \not\models (sp(\Phi \wedge \vec{x} = \vec{y},\ U_i; U'_i) \rightarrow \varphi)$ **then return** *false*;
21: **return** *true*;

2.5.2 Bisimulation Invariant Synthesis

So far, we have discussed how to automate the proof that Φ is an inductive and sufficient bisimulation invariant. However, since we do not want users to manually provide such bisimulation invariants, our verification algorithm automatically infers them using *monomial predicate abstraction* [16, 76].

Our technique for inferring suitable bisimulation invariants is shown in Algorithm 1. Given two programs $\mathcal{P}, \mathcal{P}'$ with corresponding schemas $\mathcal{S}, \mathcal{S}'$, the VERIFY procedure first generates the universe of all predicates that may be used in the bisimulation invariant (line 4). We generate all such predicates by instantiating the following database of predefined templates:

1. $\Pi_L(\ R\ ) = \Pi_{L'}(\ R'\ )$
2. $\Pi_L(\ R_1 \bowtie R_2\ ) = \Pi_{L'}(\ R'_1\ )$
3. $\Pi_L(\ R\ ) = \Pi_{L'}(\ R'_1 \bowtie R'_2\ )$
4. $\Pi_L(\ R_1 \bowtie R_2\ ) = \Pi_{L'}(\ R'_1 \bowtie R'_2\ )$

In these templates, L and R represent an attribute list and a relation under schema $\mathcal{S}$, while L' and R' represent the attribute list and relation under schema $\mathcal{S}'$. Please note that we only consider templates with at most one join operator on each side. Any predicate containing a longer join chain can be decomposed into several predicates of these forms.

Once we generate the universe $\mathcal{U}$ of all predicates that may be used in the invariant, we perform a fixed point computation in which we iteratively *weaken* the candidate bisimulation invariant. Initially, the candidate bisimulation invariant Φ starts out as the conjunction of all predicates in our universe. During the fixed point computation (lines 6–12 in Algorithm 1), the candidate invariant Φ is always stronger than the actual bisimulation invariant. Hence, if we get to a point where Φ is not strong enough to show equivalence, we

conclude that the program cannot be verified using conjunctive formulas over our templates (line 13). On the other hand, assuming that Φ is strong enough to prove equivalence, we then proceed to check whether Φ is inductive (lines 7–12). If it is, the strongest postcondition of $\Phi \wedge \vec{x} = \vec{y}$ must logically imply φ for every predicate φ used in Φ. If some predicate φ is not preserved by a pair of updates (i.e., call to CheckInductiveness returns false), we then remove φ from both Φ and our universe of predicates $\mathcal{U}$. We continue this process of weakening the invariant until it becomes an inductive bisimulation invariant, or we prove that no such invariant exists over our universe of predicates.

2.6 Implementation

We have implemented the proposed verification technique in a new tool called Mediator. Mediator utilizes the Z3 SMT solver [42] to automate reasoning over the theory of relational algebra with updates. In particular, we decide the validity of a $\mathcal{T}_{RA}$ formula ϕ by asking Z3 whether the $\mathcal{T}_{RA}$ axioms logically imply ϕ. All queries to the solver are configured to have a time budget of 2 seconds, and we assume that the answer to any query exceeding the time budget is "invalid". In the remainder of this section, we describe several important optimizations that we found necessary for making Mediator practical.

Redundant axioms. During the development of Mediator, we have found that many validity queries cannot be resolved due to Z3's limited capabilities

1.	$A \cup [\,] = A$	($\cup$ nil)
2.	$A \bowtie [\,] = [\,]$	($\bowtie$ nil)
3.	$A - A = [\,]$	($-$ nil)
4.	$\Pi_L(A \cup B) = \Pi_L(A) \cup \Pi_L(B)$	($\Pi \cup$ distributivity)
5.	$\sigma_\phi(A \cup B) = \sigma_\phi(A) \cup \sigma_\phi(B)$	($\sigma \cup$ distributivity)
6.	$(A \cup B) \bowtie C = (A \bowtie C) \cup (B \bowtie C)$	($\bowtie \cup$ distributivity)
7.	$\sigma_\phi(A) \bowtie B = \sigma_\phi(A \bowtie B)$	($\sigma \bowtie$ associativity1)
8.	$A \bowtie \sigma_\phi(B) = \sigma_\phi(A \bowtie B)$	($\sigma \bowtie$ associativity2)
9.	$\sigma_\phi(\sigma_\phi(A)) = \sigma_\phi(A)$	(σ idempotence)
10.	$(\Pi_L(A) = \Pi_{L'}(B) \wedge \phi \leftrightarrow \phi'[L/L']) \rightarrow \Pi_L(\sigma_\phi(A)) = \Pi_{L'}(\sigma_{\phi'}(B))$	($\Pi\sigma$ introduction)
11.	$\Pi_L(A) = \Pi_{L'}(B) \rightarrow \Pi_L(A\langle L_i \triangleleft v\rangle) = \Pi_{L'}(B\langle L'_i \triangleleft v\rangle)$	($\Pi\langle \triangleleft \rangle$ introduction)

Figure 2.10: List of additional axioms used for proving validity.

for performing inductive reasoning. In particular, some of the $\mathcal{T}_{RA}$ theorems needed for proving equivalence require performing structural induction over lists; but Z3 times out in most of these cases. In our implementation, we address this issue by providing a redundant set of axioms, which are logically implied by the $\mathcal{T}_{RA}$ axioms. Figure 2.10 shows a representative subset of the additional theorems that we use when issuing validity queries to Z3. Because these axioms alleviate the need for performing induction, many queries that would otherwise time out can now be successfully proven using Z3.

Conjunctive queries. While the full theory of relational algebra with updates is undecidable, we have identified a class of formulas for which we can come up with an optimization to check validity. Let us call a query *conjunctive* if it uses only projection, selection, and equi-join and all predicates are conjunctions of equalities. As pointed out in prior work, two conjunctive queries are equivalent under bag semantics if (and only if) they are syntactically iso-

morphic [36, 60]. Inspired by their work, we optimize the validity checking of formulas of the form $\Phi \rightarrow Q_i = Q_i'$, which arise when checking sufficiency of a candidate bisimulation invariant Φ. If Q_i, Q_i' are conjunctive queries, we use the schema mapping induced by Φ to rewrite the query Q_i to another query Q_i'' such that Q_i', Q_i'' refer to the same schema elements. If they are syntactically the same modulo reordering of equalities, we can conclude the original formula is valid; otherwise, we check its $\mathcal{T}_{RA}$-validity using an SMT solver.

Invariant synthesis. Recall from Algorithm 1 that our verification procedure looks for conjunctive invariants over a universe of predicates $\mathcal{U}$. While these predicates are constructed from a small set of pre-defined templates, the number of possible instantiations of these templates grows quickly in the number of attributes and relations in the database schema. To prevent a blow-up in the size of the universe $\mathcal{U}$, we use the implementation of insertion transactions to rule out infeasible predicates. For example, we only generate a predicate $\Pi_{[a_1,a_2,\ldots,a_n]}(A) = \Pi_{[b_1,b_2,\ldots,b_n]}(B)$ if there are two corresponding insertion transactions U, U' such that U inserts its argument x_i to attribute a_i of relation A, whereas U' inserts x_i into attribute b_i of relation B. Similarly, we only generate a predicate of the form $\Pi_{[a_1,a_2,\ldots,a_n]}(A) = \Pi_{[b_1,b_2,\ldots,b_n]}(B \bowtie C)$ if B and C can be joined and there are two corresponding transactions U, U' such that U inserts its argument x_i to attribute a_i of relation A, whereas U' inserts the same argument into attribute b_i of relation B or C. We have found these heuristics work quite well in that they do not lead to a loss of completeness in

practice but significantly reduce the number of predicates considered by the invariant synthesis algorithm.

Proving refinement. Recall that proving refinement requires showing that the inferred simulation invariant Φ is projectively sufficient, i.e.,

$$\left(\Phi \wedge \bigwedge_{x_j \in \vec{x}} x_j = y_j\right) \models \exists L.\ Q_i = \Pi_L(Q_i')$$

In our implementation, we determine the $\mathcal{T}_{RA}$-validity of this formula by instantiating the existentially quantified variables L with attributes in the database schema and check validity for each possible instantiation. In particular, suppose that Q_i, Q_i' can contain attributes A, A' respectively. Each instantiation of L essentially corresponds to a mapping M such that $M(A) \subseteq A'$. Our implementation rank-orders candidate mappings based on similarity metrics between attribute names and tries more likely instantiations first.

2.7 Evaluation

In this section, we evaluate the practicality and usefulness of the Mediator tool. Specifically, we use Mediator to verify equivalence and refinement between different versions of 21 database programs containing over 1000 functions in total.

Benchmarks. To perform this evaluation, we collect benchmarks from two different sources, namely challenging refactoring examples from textbooks [9] and tutorials [95] and different versions of web applications collected from

Table 2.1: Benchmark source and description for MEDIATOR.

	ID	Source	Description
textbook bench	1	Oracle tutorial	Merge relations
	2	Oracle tutorial	Split relations
	3	Textbook	Split relations
	4	Textbook	Merge relations
	5	Textbook	Move attributes
	6	Textbook	Rename attributes
	7	Textbook	Introduce associative relations
	8	Textbook	Replace the surrogate key with natural key
	9	Textbook	Introduce new attributes
	10	Textbook	Denormalization
real-world bench	11	cdx	Rename attributes and split relations
	12	coachup	Split relations
	13	2030Club	Split relations
	14	rails-ecomm	Split relations and introduce new attributes
	15	royk	Introduce and move attributes
	16	MathHotSpot	Rename relations and move attributes
	17	gallery	Split relations
	18	DeeJBase	Rename attributes and split relations
	19	visible-closet-1	Split relations
	20	visible-closet-2	Move attributes to a polymorphic relation
	21	probable-engine	Merge relations

Github. The textbook examples are useful for evaluating MEDIATOR, as they illustrate challenging database refactoring tasks that require non-trivial changes to the application code. The remaining half of the benchmarks used in the experiments are taken from real-world web applications on Github. Specifically, we evaluate MEDIATOR on two different versions A, B of the application such that (a) A, B are consecutive versions in the commit history, (b) B is obtained from A by performing a structural schema change that requires rewriting parts of the application code, and (c) B is meant to be equivalent to (or a refinement of) A. Table 2.1 describes each benchmark and changes to the schema between the two versions. Since our current implementation re-

Table 2.2: Experimental results of MEDIATOR.

	ID	Type	Funcs	Source Schema		Target Schema		Status	Time (s)	Iters	Queries
				Rels	Attrs	Rels	Attrs				
textbook bench	1	$\simeq$	4	2	8	1	6	✓	2.4	2	10
	2	$\simeq$	19	3	17	7	25	✓	11.5	5	188
	3	$\simeq$	10	1	6	2	7	✓	2.5	2	32
	4	$\simeq$	10	2	7	1	6	✓	0.3	1	16
	5	$\simeq$	7	2	5	2	5	✓	34.1	7	147
	6	$\simeq$	5	1	2	1	2	✓	0.2	1	5
	7	$\simeq$	8	2	5	3	6	✓	3.1	2	61
	8	$\simeq$	10	2	9	2	8	✓	0.4	1	18
	9	$\preceq$	8	2	7	2	8	✓	0.3	1	14
	10	$\simeq$	14	3	10	3	13	✓	57.7	23	374
real-world bench	11	$\simeq$	138	16	125	17	131	✓	90.8	13	4840
	12	$\simeq$	45	4	51	5	55	✓	23.2	7	489
	13	$\simeq$	125	15	155	16	159	✓	42.6	8	2403
	14	$\preceq$	65	8	69	9	75	✓	23.4	7	1059
	15	$\preceq$	151	19	152	19	155	✓	19.1	1	1307
	16	$\simeq$	54	7	38	8	42	✓	20.9	6	701
	17	$\simeq$	58	7	52	8	57	✓	54.5	13	1512
	18	$\simeq$	70	10	92	11	97	✓	28.7	6	1228
	19	$\simeq$	263	26	248	27	252	✓	150.6	12	9072
	20	$\simeq$	267	28	262	29	261	×	-	-	-
	21	$\simeq$	85	12	83	11	78	✓	13.7	3	823

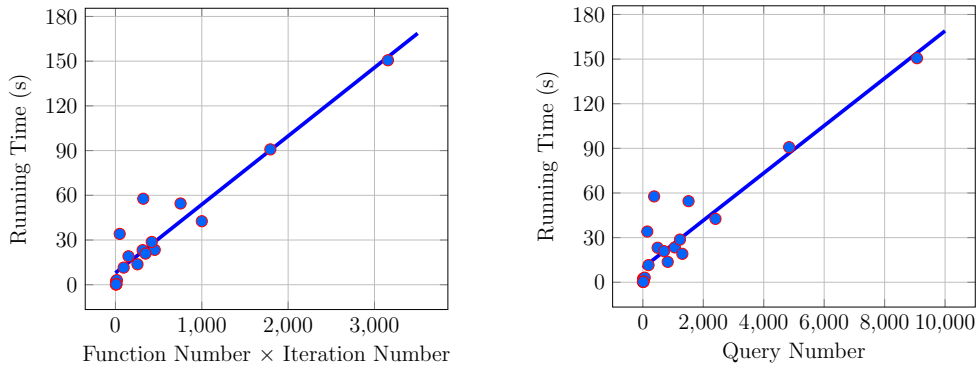
quires manually translating the web application to our IR representation, we only used the first 10 real-world applications that meet the afore-mentioned criteria.

Experimental Setup. All experiments are performed on a computer with Intel Xeon(R) E5-1620 v3 CPU and 32GB of memory, running Ubuntu 14.04 operating system.

Results. Table 2.2 summarizes the results of our evaluation of MEDIATOR on these benchmarks. For each benchmark, the column labeled *Type* shows

whether we used MEDIATOR to check refinement ($\preceq$) or equivalence ($\simeq$), and *Funcs* shows the number of functions in each program. The next two columns provide information about the number of relations and total number of attributes in the source and target schema, respectively. The last four columns provide information about MEDIATOR results: *Status* shows whether MEDIATOR was able to verify the desired property (i.e., equivalence or refinement), and *Time* provides total running time in seconds. The column labeled *Iters* shows the number of iterations that MEDIATOR takes to find an inductive simulation (or bisimulation) invariant. Finally, the last column labeled *Queries* shows the number of $\mathcal{T}_{RA}$-validity checks issued by MEDIATOR.

As we can see from Table 2.2, MEDIATOR is able to successfully verify the desired property for 20 out of 21 benchmarks. The running time of the tool ranges between 0.2 seconds for small textbook examples with a few functions to 150 seconds for large, real-world benchmarks with hundreds of functions. As expected, the running time of MEDIATOR on real-world benchmarks is typically much longer (46.8 seconds on average) than on textbook examples (11.3 seconds on average). However, some textbook examples (namely benchmarks 5 and 10) take longer than some of the real-world examples because many iterations are required to find an inductive bisimulation invariant. As shown in Figure 2.11(b), the running time of MEDIATOR is roughly linear in the number of validity queries to the SMT solver. Because the number of validity checks depends on the number of functions in the program as well as the number of iterations required for finding an inductive bisimulation invariant,



(a) Running time with respect to number of functions and iterations

(b) Running time with respect to number of $\mathcal{T}_{RA}$ validity queries

Figure 2.11: Running time analysis for MEDIATOR.

Figure 2.11(a) also shows that MEDIATOR's running time is roughly linear with respect to $\#functions \times \#iterations$.

Cause of false positives. As shown in Table 2.2, MEDIATOR fails to verify equivalence for benchmark 20, where the schema change involves moving the shared attributes of two relations into a new polymorphic relation. Upon further inspection, we determined this warning to be a false positive that is caused by a shortcoming of our inference engine for synthesizing bisimulation invariants. In particular, proving equivalence of this benchmark requires a bisimulation invariant of the form $\Pi_L(R) = \Pi_{L'}(\sigma_\phi(R'))$, which is currently not supported in our implementation (recall Section 2.5.2). While it is possible to extend our templates to include predicates of this form, this modification would significantly increase the search space, as the selection predicate ϕ can be instantiated in *many* different ways.

2.8 Limitations

As a research prototype, our current implementation of MEDIATOR has a number of limitations: First, MEDIATOR analyzes programs that are written in the language given in Figure 2.3. Hence, the applicability of MEDIATOR relies on translating the original database application to our IR, which abstracts programs as a fixed set of queries and updates to the database. Thus, programs that use dynamically generated SQL functions or control-flow constructs cannot be translated into our IR. Second, MEDIATOR synthesizes simulation and bisimulation invariants by finding the strongest conjunctive formula over a given class of predicates. However, as exemplified by the false positive from our evaluation, MEDIATOR may not be able to prove equivalence if the bisimulation requires additional predicates (or boolean connectives) beyond the ones we consider. Third, MEDIATOR axiomatizes $\mathcal{T}_{RA}$ using the theory of lists, which is also undecidable. Therefore, the SMT solver may time-out when checking validity queries over the theory of lists. Fourth, MEDIATOR can only be used to prove equivalence but not *disequivalence*. In particular, MEDIATOR cannot provide witnesses to prove that two applications are indeed not equivalent. Finally, our verification technique proves equivalence under *list semantics*. Thus, if a web application uses set/bag semantics to represent the results of database queries, MEDIATOR may end up reporting false positives. However, despite these limitations, our evaluation shows that MEDIATOR is still practical and that it can verify equivalence between different versions of many real-world database applications.

Chapter 3

Synthesizing Equivalent Database Programs [1]

While we have addressed the problem of *verifying* equivalence between two database programs before and after schema refactoring in Chapter 2, generating a new version of the program after a schema change still remains an arduous and manual task. Motivated by this problem, this chapter takes a step towards simplifying the evolution of programs that interact with a database. Specifically, we consider *database programs* that consist of a set of functions written in SQL. Given an existing database program $\mathcal{P}$ that operates over source schema $\mathcal{S}$ and a new target schema $\mathcal{S}'$ that $\mathcal{P}$ should be migrated to, our method automatically synthesizes a new database program $\mathcal{P}'$ over the new schema $\mathcal{S}'$ such that $\mathcal{P}$ and $\mathcal{P}'$ are *semantically equivalent*. Thus, our technique automates the code migration process for these kinds of database programs while ensuring that no desirable behaviors are lost and no unwanted behaviors are introduced in the process.

Our methodology for automatically migrating database programs to a new schema is illustrated schematically in Figure 3.1. Rather than synthesizing the new version of the program in one go, our algorithm decomposes the

[1] This chapter is adapted from the author's previous publication [131], where the author led the technical discussion, tool development, and experimental evaluation.

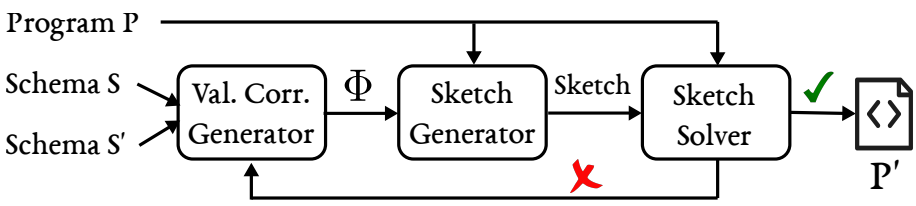


Figure 3.1: Methodology for migrating database programs.

problem into three simpler sub-tasks, each of which leverages the results of the previous task in the pipeline. Specifically, given the source and the target schemas $\mathcal{S}, \mathcal{S}'$, our algorithm starts by guessing a candidate *value correspondence* relating $\mathcal{S}$ and $\mathcal{S}'$. At a high level, a value correspondence Φ specifies how attributes in $\mathcal{S}'$ can be obtained using the attributes in $\mathcal{S}$ [86]. Intuitively, learning a value correspondence is useful because (a) it is relatively easy to guess the correct correspondence based on attribute names in the schema, and (b) having a value correspondence dramatically constrains the space of programs that may be equivalent to the original program $\mathcal{P}$.

While the value correspondence holds important clues as to what the transformation should look like, it nonetheless does not uniquely determine the target program $\mathcal{P}'$. Thus, given a candidate value correspondence Φ, our synthesis algorithm generates a *program sketch* Ω that represents the space of all programs that *may* be equivalent to the original program $\mathcal{P}$ according to Φ. In this context, a program sketch is a database program where some of the tables, attributes, or boolean constants are unknown. Furthermore, assuming the correctness of the candidate value correspondence Φ, the sketch Ω is guaranteed to have a completion that is equivalent to $\mathcal{P}$ (if one exists).

The third, and final, step in our synthesis pipeline "solves" the sketch Ω by finding an instantiation $\mathcal{P}'$ of Ω that is equivalent to $\mathcal{P}$. However, unlike existing sketch solvers that use the *counterexample-guided inductive synthesis (CEGIS)* methodology, we use a different approach that does not require symbolically encoding the semantics of database programs into an SMT formula. Specifically, since database query languages like SQL are not easily amenable to symbolic reasoning using established first-order theories supported by SMT solvers, our approach instead performs enumerative search over the space of all possible completions of the sketch. However, because this search space is typically very large, a naïve search algorithm is difficult to scale to realistic database programs. Our approach deals with this difficulty by using a novel algorithm that leverages *minimum failing inputs (MFIs)* to dramatically prune the search space.

Overall, our synthesis algorithm for automatically migrating database programs to a new schema has several useful properties: First, it is completely push-button and does not require the user to provide anything other than the original program and the source and target schemas. Second, our approach is sound in that the synthesized program is provably equivalent to the original program and does not introduce any new, unwanted behaviors. Finally, since our method performs backtracking search over all possible value correspondences, it is guaranteed to find an equivalent program over the new schema if one exists.

We have implemented our proposed approach in a prototype tool called

update *addInstructor(int id, String name, Binary pic)*
INSERT INTO *Instructor* VALUES *(id, name, pic);*

update *deleteInstructor(int id)*
DELETE FROM *Instructor* WHERE *InstId = id;*

query *getInstructorInfo(int id)*
SELECT *IName, IPic* FROM *Instructor* WHERE *InstId = id;*

update *addTA(int id, String name, Binary pic)*
INSERT INTO *TA* VALUES *(id, name, pic);*

update *deleteTA(int id)*
DELETE FROM *TA* WHERE *TaId = id;*

query *getTAInfo(int id)*
SELECT *TName, TPic* FROM *TA* WHERE *TaId = id;*

Figure 3.2: An example database program.

MIGRATOR for automatically migrating database programs to a new schema. We evaluate MIGRATOR on 20 benchmarks and show that it can successfully synthesize the new versions for *all* twenty database programs with an average synthesis time of 69.4 seconds per benchmark. Thus, we believe these experiment results provide preliminary, but firm, evidence that the proposed synthesis technique can be useful to database program developers during the schema refactoring process.

3.1 Overview

In this section, we give an overview of our technique using a simple motivating example. Consider the database program shown in Figure 3.2 for

managing and querying a course-related database with the following schema:

Class(ClassId, InstId, TaId)
Instructor(InstId, IName, IPic)
TA(TaId, TName, TPic)

This database has three tables that store information about courses, instructors, and TAs respectively. Here, the *Instructor* and *TA* tables store profile information about the course staff, including a picture. Since accessing a table containing large images may be potentially inefficient, the programmer decides to refactor the schema by introducing a new table for images. In particular, the desired new schema is as follows:

Class(ClassId, InstId, TaId)
Instructor(InstId, IName, PicId)
TA(TaId, TName, PicId)
Picture(PicId, Pic)

As a result of this schema change, the program from Figure 3.2 needs to be reimplemented to conform to the new schema. We now explain how MIGRATOR automatically synthesizes the new version of the program.

Value correspondence generation. As mentioned earlier, MIGRATOR lazily enumerates possible value correspondences (VCs) between the source and target schemas. For this example, the first VC Φ generated by MIGRATOR contains the following mappings:

$$\begin{aligned} \mathit{Instructor.IPic} &\rightarrow \mathit{Picture.Pic} \\ \mathit{TA.TPic} &\rightarrow \mathit{Picture.Pic} \end{aligned}$$

In addition, all other attributes $T.a$ in the source schema are mapped to the same $T.a$ in the target schema.

update *addInstructor(int id, String name, Binary pic)*
INSERT INTO $??_1\{$ *Picture* $\bowtie$ *Instructor, Picture* $\bowtie$ *TA* $\bowtie$ *Instructor,*
Picture $\bowtie$ *TA* $\bowtie$ *Class* $\bowtie$ *Instructor* $\}$ VALUES *(id, name, pic);*

update *deleteInstructor(int id)*
DELETE $??_2\{$ *[Picture], ..., [Picture, Instructor, TA, Class]* $\}$
FROM $??_3\{$ *Picture* $\bowtie$ *Instructor, Picture* $\bowtie$ *TA* $\bowtie$ *Instructor,*
Picture $\bowtie$ *TA* $\bowtie$ *Class* $\bowtie$ *Instructor* $\}$ WHERE *InstId = id;*

query *getInstructorInfo(int id)*
SELECT *IName, Pic* FROM $??_4\{$
Picture $\bowtie$ *Instructor, Picture* $\bowtie$ *TA* $\bowtie$ *Instructor,*
Picture $\bowtie$ *TA* $\bowtie$ *Class* $\bowtie$ *Instructor* $\}$ WHERE *InstId = id;*

update *addTA(int id, String name, Binary pic)*
INSERT INTO $??_5\{$ *Picture* $\bowtie$ *TA, Picture* $\bowtie$ *Instructor* $\bowtie$ *TA,*
Picture $\bowtie$ *Instructor* $\bowtie$ *Class* $\bowtie$ *TA* $\}$ VALUES *(id, name, pic);*

update *deleteTA(int id)*
DELETE $??_6\{$ *[Picture], ..., [Picture, Instructor, TA, Class]* $\}$
FROM $??_7\{$ *Picture* $\bowtie$ *TA, Picture* $\bowtie$ *Instructor* $\bowtie$ *TA,*
Picture $\bowtie$ *Instructor* $\bowtie$ *Class* $\bowtie$ *TA* $\}$ WHERE *TaId = id;*

query *getTAInfo(int id)*
SELECT *TName, Pic* FROM $??_8\{$
Picture $\bowtie$ *TA, Picture* $\bowtie$ *Instructor* $\bowtie$ *TA,*
Picture $\bowtie$ *Instructor* $\bowtie$ *Class* $\bowtie$ *TA* $\}$ WHERE *TaId = id;*

Figure 3.3: Generated sketch over the new database schema.

Sketch generation. Next, MIGRATOR uses the candidate VC Φ to generate a program sketch that encodes the space of all programs that are consistent with Φ. The corresponding sketch for this example is shown in Figure 3.3. Here, each hole, denoted $??\{c_1, \ldots, c_n\}$, corresponds to an unknown constant drawn from the set $\{c_1, \ldots, c_n\}$. As will be discussed later in Section 3.2, we

use the statement:

$$\texttt{INSERT INTO } T_1 \bowtie T_2 \texttt{ VALUES } \cdots$$

as short-hand for:

$$\begin{array}{l}\texttt{INSERT INTO } T_1 \texttt{ VALUES } \cdots \\ \texttt{INSERT INTO } T_2 \texttt{ VALUES } \cdots\end{array}$$

Thus, the first function in the sketch corresponds to the following three possible implementations of *addInstructor*:

```
INSERT INTO Instructor VALUES (id, name, v0);
INSERT INTO Picture VALUES (v0, pic);
or
INSERT INTO Instructor VALUES (id, name, v1);
INSERT INTO TA VALUES (v2, v3, v1);
INSERT INTO Picture VALUES (v1, pic);
or
INSERT INTO Instructor VALUES (id, name, v4);
INSERT INTO Class VALUES (v5, id, v6);
INSERT INTO TA VALUES (v6, v7, v4);
INSERT INTO Picture VALUES (v4, pic);
```

where $v_0, v_1, \ldots, v_7$ are unique values.

Observe that the program sketch shown in Figure 3.3 has an enormous number of possible completions — in particular, it corresponds to a search space of $164,025$ possible re-implementations of the original program.

Sketch completion. Given a sketch Ω and the original program $\mathcal{P}$, the goal of sketch completion is to find an instantiation $\mathcal{P}'$ of Ω such that $\mathcal{P}'$ is equivalent to $\mathcal{P}$, if such a $\mathcal{P}'$ exists. Unfortunately, it is difficult to solve this sketch using existing solvers (e.g., [117, 119]) because the symbolic encoding of the program is quite complex due to the non-trivial semantics of SQL. We

deal with this difficulty by (a) encoding the space of *all* possible programs represented by the sketch using a SAT formula Ψ, and (b) using minimum failing inputs to dramatically prune the search space represented by Ψ.

Going back to our sketch Ω from Figure 3.3, MIGRATOR generates the following SAT formula that encodes all possible instantiations of Ω:

$$\oplus(b_1^1, b_1^2, b_1^3) \wedge \oplus(b_2^1, \ldots, b_2^{15}) \wedge \oplus(b_3^1, b_3^2, b_3^3) \wedge \oplus(b_4^1, b_4^2, b_4^3) \wedge \\ \oplus(b_5^1, b_5^2, b_5^3) \wedge \oplus(b_6^1, \ldots, b_6^{15}) \wedge \oplus(b_7^1, b_7^2, b_7^3) \wedge \oplus(b_8^1, b_8^2, b_8^3)$$

Here, $\oplus$ denotes n-ary xor, and b_i^j is a boolean variable that is assigned to true iff hole $??_i$ in the sketch is instantiated with the j-th constant in $??_i$'s domain.

Given this formula Ψ, MIGRATOR queries the SAT solver for a model. For the purpose of this example, suppose the SAT solver returns the following model for Ψ:

$$b_1^3 \wedge b_2^2 \wedge b_3^3 \wedge b_4^3 \wedge b_5^1 \wedge b_6^4 \wedge b_7^3 \wedge b_8^3 \tag{3.1}$$

which corresponds to the following assignment of the holes:

$$\begin{array}{ll} & ??_1 = ??_3 = ??_4 = \mathit{Picture} \bowtie \mathit{TA} \bowtie \mathit{Class} \bowtie \mathit{Instructor} \\ \wedge & ??_2 = [\mathit{Instructor}] \ \wedge \ ??_5 = \mathit{Picture} \bowtie \mathit{TA} \ \wedge \ ??_6 = [\mathit{TA}] \\ \wedge & ??_7 = ??_8 = \mathit{Picture} \bowtie \mathit{Instructor} \bowtie \mathit{Class} \bowtie \mathit{TA} \end{array} \tag{3.2}$$

However, instantiating the sketch with this assignment results in a program $\mathcal{P}'$ that is *not* equivalent to $\mathcal{P}$. Now, we *could* block this program $\mathcal{P}'$ by conjoining the negation of Equation 3.1 with Ψ and asking the SAT solver for another model. While this strategy would give us a different instantiation of sketch Ω, it would preclude *only one* of the $164,025$ possible instantiations of Ω. Our key idea is to learn from this failure and block many other programs that are incorrect for the same reason as $\mathcal{P}'$.

Towards this goal, our approach computes a *minimum failing input*, which is a shortest sequence of function invocations such that the result of $\mathcal{P}$ differs from that of $\mathcal{P}'$. For this example, such a minimum failing input is the following invocation sequence ω:

$$addTA(ta1, name1, pic1); getTAInfo(ta1) \tag{3.3}$$

This input establishes that $\mathcal{P}'$ is *not* equivalent to $\mathcal{P}$ because the query result for $\mathcal{P}$ is *(name1, pic1)* whereas the query result for $\mathcal{P}'$ is empty.

Our idea is to utilize such a minimum failing input ω to prune incorrect programs other than just $\mathcal{P}'$. Specifically, let $\mathcal{F}$ denote the functions that appear in the invocation sequence ω, and let $\mathcal{H}$ be the holes that appear in the sketch for functions in $\mathcal{F}$. Our key intuition is that the assignments to holes in $\mathcal{H}$ are *sufficient* for obtaining a spurious program, as ω is a witness to the disequivalence between $\mathcal{P}$ and $\mathcal{P}'$. Thus, rather than blocking the whole model, we can extract the assignment to the holes in $\mathcal{H}$ and use this partial assignment to obtain a much stronger blocking clause. For our example, this yields the clause $\neg(b_5^1 \wedge b_8^3)$ because only the fifth and eighth holes appear in the sketches for *addTA* and *getTAInfo*. Using this blocking clause, we can eliminate a total of $18,225$ incorrect programs rather than just $\mathcal{P}'$.

Continuing in this manner, MIGRATOR finally obtains the following model for Equation 3.2:

$$b_1^1 \wedge b_2^2 \wedge b_3^1 \wedge b_4^1 \wedge b_5^1 \wedge b_6^4 \wedge b_7^1 \wedge b_8^1$$

```
update addInstructor(int id, String name, Binary pic)
    INSERT INTO Instructor VALUES (id, name, UID0);
    INSERT INTO Picture VALUES (UID0, pic);

update deleteInstructor(int id)
    DELETE Instructor FROM Picture JOIN Instructor
      ON Picture.PicId = Instructor.PicId WHERE InstId = id;

query getInstructorInfo(int id)
    SELECT IName, Pic FROM Picture JOIN Instructor
      ON Picture.PicId = Instructor.PicId WHERE InstId = id;

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, UID1);
    INSERT INTO Picture VALUES (UID1, pic);

update deleteTA(int id)
    DELETE TA FROM Picture JOIN TA
    ON Picture.PicId = TA.PicId WHERE TaId = id;

query getTAInfo(int id)
    SELECT TName, Pic FROM Picture JOIN TA
      ON Picture.PicId = TA.PicId WHERE TaId = id;
```

Figure 3.4: The synthesized database program.

This model corresponds to the program $\mathcal{P}'$ shown in Figure 3.4, which is indeed equivalent to the original program from Figure 3.2. Thus, MIGRATOR returns $\mathcal{P}'$ as the synthesis result.

3.2 Preliminaries

In this section, we introduce the syntax and semantics of database programs for the synthesis algorithm. For the purpose of this chapter, a database program consists of a set of functions, where each function is either a *query* or

$$
\begin{array}{rcl}
\mathit{Prog} & := & \mathit{Func}+ \\
\mathit{Func} & := & \textbf{update}\ \mathit{Name}(\mathit{Param}+)\ U \\
 & | & \textbf{query}\ \mathit{Name}(\mathit{Param}+)\ Q \\
\mathit{Update}\ U & := & \mathit{InsStmt} \mid \mathit{DelStmt} \mid \mathit{UpdStmt} \mid U; U \\
\mathit{Query}\ Q & := & \Pi_{a+}(Q) \mid \sigma_{\phi}(Q) \mid J \\
\mathit{Join}\ J & := & T \mid J_a \bowtie_a J \\
\mathit{Pred}\ \phi & := & a\ \mathit{op}\ a \mid a\ \mathit{op}\ v \mid a \in Q \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\mathit{InsStmt} & := & \texttt{ins}(J, \{(a:v)+\}) \\
\mathit{DelStmt} & := & \texttt{del}([T+], J, \phi) \\
\mathit{UpdStmt} & := & \texttt{upd}(J, \phi, a, v)
\end{array}
$$

$$
\mathit{Param} \in \textbf{Variable} \quad \mathit{Name} \in \textbf{String}
$$
$$
T \in \textbf{Table} \quad a \in \textbf{Attribute} \quad v \in \textbf{Value} \cup \textbf{Variable}
$$

Figure 3.5: Syntax of database programs for synthesis.

update to the database. As shown in Figure 3.5, every function consists of a name, a list of parameters, and a function body.

The body of a query function is a relational algebra expression involving projection (Π), selection (σ), and join ($\bowtie$). As is standard, $\Pi_{a_1,\ldots,a_n}(Q)$ recursively evaluates sub-query Q to obtain a table T and then constructs a table T' that is the same as T but containing only the columns $a_1, \ldots, a_n$. The filter operation $\sigma_{\phi}(Q)$ recursively evaluates Q to obtain a table T and then filters out all rows in T that do not satisfy predicate ϕ. A join expression $J_{1a_1}\bowtie_{a_2} J_2$ corresponds to the equi-join of J_1 and J_2 based on predicate $a_1 = a_2$, where a_1 is an attribute in J_1 and a_2 is an attribute in J_2. In the rest of this chapter, we use the terminology *join* or *join chain* to refer to both database tables as well as (possibly nested) join expressions of the form $J_{1a_1} \bowtie_{a_2} J_2$. Furthermore, since natural join is a special case of equi-join, we also use the

standard notation $J_1 \bowtie J_2$ to denote natural joins where the equality check is implicit on identically named columns.

In contrast to query functions that do not change the state of the database, update functions can add or remove tuples to database tables. Specifically, an insert statement $\texttt{ins}(T, \{a_1 : v_1, \ldots, a_n : v_n\})$ inserts the tuple $\{a_1 : v_1, \ldots, a_n : v_n\}$ into relation T. To simplify presentation in the rest of the chapter, we use the syntax

$$\texttt{ins}(T_{1fk_1} \bowtie_{pk_2} T_2, \{a_1 : v_1, \ldots, a_n : v_n, a'_1 : v'_1, \ldots, a'_m : v'_m\})$$

as short-hand for the following sequence of insertions:

$$\begin{array}{l} \texttt{ins}(T_1,\ \{pk_1 : u_0,\ a_1 : v_1, \ldots, a_n : v_n,\ fk_1 : u_1\}); \\ \texttt{ins}(T_2,\ \{pk_2 : u_1,\ a'_1 : v'_1, \ldots, a'_m : v'_m\}) \end{array}$$

where u_0, u_1 are unique values, and the schema for T_1, T_2 are $T_1(pk_1, a_1, \ldots, a_n, fk)$ and $T_2(pk_2, a'_1, \ldots, a'_m)$ respectively.

A delete statement $\texttt{del}([T_1, \ldots, T_n], J, \phi)$ removes from tables $T_1, \ldots, T_n$ exactly those tuples that satisfy predicate ϕ in join chain J. As an example, consider the delete statement $\texttt{del}([T_1],\ T_{1a_1} \bowtie_{a_2} T_2,\ \phi)$. Here, we first compute $T_{1a_1} \bowtie_{a_2} T_2$ to obtain a virtual table T where each tuple in T is the union of a source tuple in T_1 and a source tuple in T_2. We then obtain another virtual table T' that filters out predicates satisfying ϕ. Finally, we delete from T_1 all tuples that occur as (a prefix of) a tuple in T'. In contrast, if the statement is $\texttt{del}([T_1, T_2],\ T_{1a_1} \bowtie_{a_2} T_2,\ \phi)$, the deletion is performed on both T_1 and T_2.

We refer the reader to [91] for a more detailed discussion of the semantics of delete statements.[2]

An update statement $\texttt{upd}(J, \phi, a, v)$ modifies the value of attribute a to v for all tuples satisfying predicate ϕ in join chain J [92]. For instance, consider the update statement $\texttt{upd}(T_{1a_1} \bowtie_{a_2} T_2,\ \phi,\ T_1.a_3,\ v)$. Like delete statements, we first compute $T_{1a_1} \bowtie_{a_2} T_2$ and get a virtual table T where each tuple in T is the union of a source tuple in T_1 and a source tuple in T_2. Then we filter out tuples satisfying predicate ϕ in T and get another virtual table T'. Finally, we update attribute a_3 in T_1 to value v for all T_1 tuples that appear in T'.

Example 3.2.1. *Consider a simple database with two tables:*

Car

cid	***model***	***year***
1	*M1*	*2016*
2	*M2*	*2018*

Part

name	***amount***	***cid***
tire	*10*	*1*
brake	*20*	*1*
tire	*20*	*2*
brake	*30*	*2*

The delete statement

$$\texttt{del}([Car,\ Part],\ Car \bowtie Part,\ model = M1)$$

would delete tuple $(1, M1, 2016)$ *from the Car table and tuples* $(tire, 10, 1)$, $(brake, 20, 1)$ *from the Part table. On the other hand, the update statement*

$$\texttt{upd}(Car \bowtie Part,\ model = M2 \wedge name = tire,\ amount,\ 30)$$

would modify the third record of Part to $(tire, 30, 2)$.

[2]We consider this form of delete statement rather than the more standard $\texttt{del}(T, \phi)$ as it dramatically simplifies presentation in the rest of the chapter.

3.3 Synthesis Algorithm

In this section, we present our algorithm for automatically migrating database programs to a new schema. We start with an overview of the top-level algorithm and then discuss value correspondence enumeration, sketch generation, and sketch completion in more detail.

3.3.1 Algorithm Overview

Our top-level synthesis algorithm is summarized as pseudo-code in Algorithm 2. Given the original program $\mathcal{P}$ over schema $\mathcal{S}$ and the target schema $\mathcal{S}'$, Synthesize either returns a program $\mathcal{P}'$ such that $\mathcal{P} \simeq \mathcal{P}'$ or $\perp$ to indicate that no equivalent program exists.

In a nutshell, the Synthesize procedure is a while loop (lines 2 - 7) that lazily enumerates all possible *value correspondences* between the source and target schemas. Formally, a value correspondence Φ from source schema $\mathcal{S}$ to target schema $\mathcal{S}'$ is a mapping from each attribute in $\mathcal{S}$ to a *set* of attributes in $\mathcal{S}'$ [86]. Specifically, if $T'.b \in \Phi(T.a)$, this indicates that the entries in column a in the source table T are the same as the entries in column b of table T' in the target schema. Observe that, if Φ maps some attribute $T.a$ in $\mathcal{S}$ to $\emptyset$, this indicates that attribute a of table T has been deleted from the database. Similarly, if $|\Phi(T.a)| > 1$, this indicates that attribute $T.a$ has been duplicated in the target schema. Our value correspondence is a slightly simplified version of the definition given by Miller et al. [86]. For example, their definition also allows attributes in the target schema to be obtained by

Algorithm 2 Synthesizing database programs

1: **procedure** SYNTHESIZE($\mathcal{P}, \mathcal{S}, \mathcal{S}'$)
   Input: Program $\mathcal{P}$ over source schema $\mathcal{S}$, target schema $\mathcal{S}'$
   Output: Program $\mathcal{P}'$ or $\perp$ to indicate failure
2: **while** *true* **do**
3: $\Phi \leftarrow$ NEXTVALUECORR($\mathcal{S}, \mathcal{S}'$);
4: **if** $\Phi = \perp$ **then return** $\perp$;
5: $\Omega \leftarrow$ GENSKETCH($\Phi, \mathcal{P}$);
6: $\mathcal{P}' \leftarrow$ COMPLETESKETCH($\Omega, \mathcal{P}$);
7: **if** $\mathcal{P}' \neq \perp$ **then return** $\mathcal{P}'$;

applying a function to attributes in the source schema. Our technique can be extended to handle this scenario, albeit at the cost of increasing the size of the search space.

Now, given a candidate value correspondence Φ, the GENSKETCH procedure at line 5 generates a sketch Ω that represents all programs that *may* be equivalent to $\mathcal{P}$ under the assumption that Φ is correct. Finally, the COMPLETESKETCH procedure (line 6) tries to find an instantiation $\mathcal{P}'$ of Ω such that $\mathcal{P}' \simeq \mathcal{P}$. If such a $\mathcal{P}'$ exists, then the algorithm terminates and returns $\mathcal{P}'$ as the transformed program. On the other hand, if there is no completion of the sketch that is equivalent to $\mathcal{P}$, this indicates that the conjectured value correspondence is incorrect. In this case, the algorithm moves on to the next value correspondence Φ' and re-attempts the synthesis task using Φ'.

As formalized in more detail in [132], our synthesis algorithm is both sound and relatively complete. That is, if SYNTHESIZE returns $\mathcal{P}'$ as a solution, then $\mathcal{P}'$ is indeed equivalent to $\mathcal{P}$. Furthermore, SYNTHESIZE is relatively

complete, meaning that it can always find an equivalent program $\mathcal{P}'$ under the assumption that (a) we have access to a sound and complete oracle for verifying equivalence of database programs, (b) $\mathcal{P}'$ is related to $\mathcal{P}$ according to a value correspondence that conforms to our definition, and (c) $\mathcal{P}'$ has the same general structure as $\mathcal{P}$.

In the following subsections, we explain the subroutines used in the SYNTHESIZE algorithm in more detail.

3.3.2 Lazy Enumeration of Value Correspondence

In order to guarantee the completeness of our synthesis algorithm, we need a way to enumerate *all* possible value correspondences between the source and target schemas. However, it is infeasible to generate all such value correspondences *eagerly*, as there are exponentially many possibilities. In this section, we describe how to lazily enumerate value correspondences in decreasing order of likelihood using a partial weighted MaxSAT encoding.

Background on MaxSAT. MaxSAT is a generalization of the traditional boolean satisfiability problem and aims to determine the maximum number of clauses that can be satisfied. Specifically, a MaxSAT problem is defined as a triple $(\mathcal{H}, \mathcal{S}, \mathcal{W})$, where $\mathcal{H}$ is a set of *hard clauses (constraints)*, $\mathcal{S}$ is a set of *soft clauses*, and $\mathcal{W}$ is a mapping from each soft clause $c \in \mathcal{S}$ to a weight, which is an integer indicating the relative importance of satisfying clause c. Then, the goal of MaxSAT is to find an interpretation I such that:

1. I satisfies all the hard clauses (i.e., $I \models \bigwedge_{c_i \in \mathcal{H}} c_i$)

2. I maximizes the weight of the satisfied soft clauses

Variables. To describe our MaxSAT encoding, suppose that the source (resp. target) schema contains attributes $a_1, \ldots, a_n$ (resp. $a'_1, \ldots, a'_m$). In our encoding, we introduce a boolean variable x_{ij} to indicate that attribute a_i in the source schema is mapped by the value correspondence Φ to attribute a'_j in the target schema, i.e.,

$$x_{ij} \Leftrightarrow a'_j \in \Phi(a_i)$$

Hard constraints. Hard constraints in our MaxSAT encoding rule out infeasible value correspondences:

- *Type-compatibility:* Since $a'_j \in \Phi(a_i)$ indicates that the entries stored in a_i and a'_j are the same, x_{ij} must be false if a_i and a'_j have different types. Thus, we add the following hard constraint for type compatibility:

$$\bigwedge_{i,j} \neg x_{ij} \text{ where } \mathit{type}(a_i) \neq \mathit{type}(a'_j)$$

- *Necessary condition for equivalence:* If the source program $\mathcal{P}$ queries some attribute a_i of the database, then there must be a corresponding attribute a'_j that a_i is mapped to; otherwise, the source and target programs would not be equivalent (recall Section 2.2.3). Thus, we introduce

the following hard constraint:

$$\bigvee_{1 \leq j \leq m} x_{ij} \text{ where } a_i \text{ is queried in } \mathcal{P}$$

which ensures that every attribute that is queried in the original program is mapped to at least one attribute in the target schema.

Soft constraints. The soft constraints in our encoding serve two purposes: First, since most attributes in the source schema typically have a unique corresponding attribute in the target schema, our soft constraints prioritize one-to-one mappings over one-to-many ones. Second, since attributes with similar names are more likely to be mapped to each other, they prioritize value correspondences that relate similarly named attributes.

To encode the latter constraint, we introduce a soft clause x_{ij} with weight $sim(a_i, a'_j)$ for every variable x_{ij}. Here, sim is a heuristic metric that measures similarity between the names of attributes a_i and a'_j.[3] To encode the former constraint, we add a soft clause $x_{ij} \rightarrow \neg x_{ik}$ (with fixed weight α) for every $i \in [1, n], j \in [1, m]$ and $k \in (j, m]$. Essentially, such clauses tell the solver to de-prioritize mappings where the cardinality of $\Phi(a_i)$ is large.

Blocking clauses. While our initial MaxSAT encoding consists of exactly the hard and soft constraints discussed above, we need to add additional constraints to block previously rejected value correspondences. Specifically, let A

[3]In our implementation, we implement sim as $\alpha - Levenshtein(a_i, a'_j)$ where α is a fixed constant and *Levenshtein* is the standard Levenshtein distance.

$$
\begin{array}{rcl}
\mathit{Prog} & := & \mathit{Func}+ \\
\mathit{Func} & := & \textbf{update}\ \mathit{Name}(\mathit{Param}+)\ U \\
 & | & \textbf{query}\ \mathit{Name}(\mathit{Param}+)\ Q \\
\mathit{Update}\ U & := & \mathit{InsStmt} \mid \mathit{DelStmt} \mid \mathit{UpdStmt} \mid U; U \mid U \circledcirc U \\
\mathit{Query}\ Q & := & \Pi_{(??\{a+\})+}(Q) \mid \sigma_{\phi}(Q) \mid J \mid Q \circledcirc Q \\
\mathit{Join}\ J & := & T \mid J_a \bowtie_a J \\
\mathit{Pred}\ \phi & := & ??\{a+\}\ \mathit{op}\ ??\{a+\} \mid ??\{a+\}\ \mathit{op}\ v \\
 & | & \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\mathit{InsStmt} & := & \texttt{ins}(J,\ \{(??\{a+\} : v)+\}) \\
\mathit{DelStmt} & := & \texttt{del}(??\{L+\},\ J,\ \phi) \\
\mathit{UpdStmt} & := & \texttt{upd}(J,\ \phi,\ ??\{a+\},\ v) \\
\mathit{TabList}\ L & := & [T+]
\end{array}
$$

$$
\mathit{Param} \in \textbf{Variable} \quad \mathit{Name} \in \textbf{String}
$$
$$
T \in \textbf{Table} \quad a \in \textbf{Attribute} \quad v \in \textbf{Value} \cup \textbf{Variable}
$$

Figure 3.6: Sketch language used in the synthesis algorithm.

be an assignment (with corresponding value correspondence Φ_A) returned by the MaxSAT solver, and suppose that there is no program $\mathcal{P}'$ that is equivalent to $\mathcal{P}$ under Φ_A. In this case, our algorithm adds $\neg A$ as a hard constraint to prevent exploring the same value correspondence multiple times.

3.3.3 Sketch Generation

In this section, we explain the GenSketch procedure for generating a sketch that represents all programs that may be equivalent to $\mathcal{P}$ under a given value correspondence Φ. We first describe our sketch language and then explain how to use the value correspondence to generate a suitable sketch.

Sketch language. Our sketch language for database programs is presented in Figure 3.6, where **??** represents a hole in the sketch and the subsequent set indicates the domain of that hole. $\textcircled{?}$ is a choice operator and $s_1 \,\textcircled{?}\, s_2$ denotes the statement could either be s_1 or s_2. $E+$ indicates a list of elements of type E. The language differs from the source language in Figure 3.5 in the following ways: First, programs in the sketch language can contain a construct of the form $\mathbf{??}\{e_1, \ldots, e_n\}$, where the question mark is referred to as a *hole* and the set of elements $\{e_1, \ldots, e_n\}$ is the *domain* of that hole — i.e., the question mark must be filled with some element drawn from the set $\{e_1, \ldots, e_n\}$. In addition, programs in the sketch language also contain a *choice* construct $s_1 \,\textcircled{?}\, s_2$, which is short-hand for the conditional statement:

$$\textbf{if } \mathbf{??}\{\top, \bot\} \textbf{ then } s_1 \textbf{ else } s_2$$

where $\top, \bot$ represent the boolean constants true and false, respectively. Thus, program sketches in this context represent multiple (but finitely many) programs written in the syntax of Figure 3.5.

Join correspondence. In order to generate a sketch from a program $\mathcal{P}$ and value correspondence Φ, our approach first maps each join chain used in $\mathcal{P}$ to a set of possible join chains over the target schema. We refer to such a mapping as a *join correspondence* and say that a join correspondence (J, J') is *valid* with respect to Φ if Φ can map all attributes used in J to attributes in J'.

$$\frac{A \subseteq Attrs(J) \quad \forall a \in A.\ \exists a' \in \Phi(a).\ a' \in Attrs(J')}{\Phi \vdash_A J \sim J'} \quad \text{(Attrs)}$$

$$\frac{A = Attrs(J) \quad \Phi \vdash_A J \sim J'}{\Phi \vdash J \sim J'} \quad \text{(JoinChain)}$$

Figure 3.7: Inference rules for checking join correspondence (J, J') under value correspondence Φ.

Figure 3.7 presents inference rules for checking whether a join correspondence (J, J') is valid under Φ. Specifically, the judgment $\Phi \vdash_A J \sim J'$ indicates that every attribute $a \in A$ of join chain J can be mapped to some attribute of join chain J' under Φ. Similarly, the judgment $\Phi \vdash J \sim J'$ means that *every* attribute in the join chain J can be mapped to an attribute of J' using Φ. Observe that, if $\Phi \vdash J \sim J_1$ *and* $\Phi \vdash J \sim J_2$, it means that join chain J in the source program could map to *either* J_1 or J_2 in the target program.

Sketching approach. Our sketch generation technique uses the inferred join correspondences to produce a sketch that encodes all possible programs that may be equivalent to the source program. However, since a join chain J might correspond to any one of the join chains $J_1, \ldots, J_n$ in the target program, our sketch generation method proceeds in two phases: In the first phase, we non-deterministically pick any one of the join chains J_i that J could map to. Then, in the second phase, we combine the sketches obtained using $J_1, \ldots, J_n$ to obtain a more general sketch that accounts for every possibility.

$$\frac{\Phi \vdash J \sim J'}{\Phi \vdash J \rightsquigarrow J'} \text{ (Join)} \quad \frac{\Phi(a) = \{a'_1, \ldots, a'_n\}}{\Phi \vdash a \rightsquigarrow \texttt{??}\{a'_1, \ldots, a'_n\}} \text{ (Attr)}$$

$$\frac{a_i \in \mathit{Attrs}(\phi) \quad \Phi \vdash a_i \rightsquigarrow h_i \quad i = 1, \ldots, n}{\Phi \vdash \phi \rightsquigarrow \phi[h_1/a_1, \ldots, h_n/a_n]} \text{ (Pred)}$$

$$\frac{\Phi \vdash Q \rightsquigarrow \Omega \quad \Phi \vdash \phi \rightsquigarrow \phi'}{\Phi \vdash \sigma_\phi(Q) \rightsquigarrow \sigma_{\phi'}(\Omega)} \text{ (Filter)}$$

$$\frac{\begin{array}{c}\Phi \vdash Q(J) \rightsquigarrow \Omega(h) \quad \Phi \vdash a_j \rightsquigarrow h_j \quad j = 1, \ldots, m \\ A = \{a_1, \ldots, a_m\} \cup \mathit{Attrs}(Q) \quad \Phi \vdash_A J \sim J'\end{array}}{\Phi \vdash \Pi_{a_1,\ldots,a_m}(Q(J)) \rightsquigarrow \Pi_{h_1,\ldots,h_m}(\Omega(J'))} \text{ (Proj)}$$

$$\frac{\begin{array}{c}A = \mathit{Attrs}(L) \cup \mathit{Attrs}(\phi) \quad \Phi \vdash \phi \rightsquigarrow \phi' \\ \Phi \vdash_A J \sim J' \quad \mathit{TabLists}(J') = \{L_1, \ldots, L_n\}\end{array}}{\Phi \vdash \texttt{del}(L, J, \phi) \rightsquigarrow \texttt{del}(\texttt{??}\{L_1, \ldots, L_n\}, J', \phi')} \text{ (Delete)}$$

$$\frac{\begin{array}{c}\Phi \vdash \phi \rightsquigarrow \phi' \quad \Phi \vdash a \rightsquigarrow h \\ A = \mathit{Attrs}(\phi) \cup \{a\} \quad \Phi \vdash_A J \sim J'\end{array}}{\Phi \vdash \texttt{upd}(J, \phi, a, v) \rightsquigarrow \texttt{upd}(J', \phi', h, v)} \text{ (Update)}$$

$$\frac{\Phi \vdash J \sim J' \quad \Phi \vdash a_i \rightsquigarrow h_i \quad i = 1, \ldots, n}{\begin{array}{l}\Phi \vdash \quad \texttt{ins}(J, \{a_1 : v_1, \ldots, a_m : v_m\}) \rightsquigarrow \\ \qquad \texttt{ins}(J', \{h_1 : v_1, \ldots, h_m : v_m\})\end{array}} \text{ (Insert)}$$

Figure 3.8: Rewrite rules for generating sketch from value correspondence.

Sketch generation, phase I. The first phase of our sketch generation procedure is summarized in Figure 3.8 and assumes that every join chain J in the source program maps to a unique join chain J' in the target program. Here, all holes **??** are annotated with an index to ensure they are globally unique. The function *TabLists* returns all non-empty subset of tables in a join, i.e. $\mathit{TabLists}(T_1 \bowtie \ldots \bowtie T_n) = \mathit{PowerSet}(\{T_1, \ldots, T_n\}) \setminus \emptyset$. The rules in Figure 3.8

derive judgments of the form $\Phi \vdash s \rightsquigarrow \Omega$, meaning that statement s in the original program can be rewritten into sketch Ω under the assumption that (a) Φ is correct and (b) every join chain in the source program corresponds to a unique join chain in the target program. We now explain each of these rules in more detail.

The Attr (resp. Join) rule corresponds to a base case of our inductive rewrite system and generates the sketch directly using the value (resp. join) correspondence. The Pred rule first generates holes $h_1, \ldots, h_n$ for each attribute a_i in ϕ and then generates a predicate sketch by replacing each a_i with its corresponding sketch. The Filter and Proj rules are similar and generate the sketch by recursively rewriting the nested query, predicate, and attributes.

The last three rules in Figure 3.8 generate sketches for update statements. Here, the Update and Insert rules are straightforward and generate the sketch by recursively rewriting the nested attributes and predicates. For the Delete rule, recall that deletion statements are of the form $\texttt{del}(\mathit{Tbls}, J, \phi)$, where *Tbls* can refer to any non-empty subset of the tables used in J. Thus, the sketch for deletion statements contains a hole for *Tbls*, with the domain of the hole being the power-set of the tables used in J'.

Sketch generation, phase II. Recall that a join chain in the source program may correspond to multiple join chains in the target schema — the target join chain is not *uniquely* determined by a given value correspondence. Thus, the second phase of our algorithm combines the sketches generated during the

$$\frac{\Phi \vdash Q \twoheadrightarrow \Omega \quad \Phi \vdash Q \rightsquigarrow \Omega' \quad \Omega = \Omega_1 \oplus \ldots \oplus \Omega_n \quad \Omega' \neq \Omega_i \quad i = 1, \ldots, n}{\Phi \vdash Q \twoheadrightarrow \Omega \oplus \Omega'} \text{ (Query)}$$

$$\frac{\Phi \vdash U \twoheadrightarrow \Omega \quad \Phi \vdash U \rightsquigarrow \Omega' \quad \Omega = \Omega_1 \oplus \ldots \oplus \Omega_n \quad \Omega' \neq \Omega_i \quad i = 1, \ldots, n}{\Phi \vdash U \twoheadrightarrow \Omega \oplus \Omega' \oplus (\Omega \bullet \Omega')} \text{ (Update)}$$

$$\frac{\Phi \vdash s \rightsquigarrow \Omega}{\Phi \vdash s \twoheadrightarrow \Omega} \text{ (Lift)} \qquad \frac{\Phi \vdash U_1 \twoheadrightarrow \Omega_1 \quad \Phi \vdash U_2 \twoheadrightarrow \Omega_2}{\Phi \vdash U_1; U_2 \twoheadrightarrow \Omega_1; \Omega_2} \text{ (Seq)}$$

Figure 3.9: Inference rules for composing multiple sketches.

$$\begin{array}{rcl} U_1 \bullet U_2 & = & U_1; U_2 \quad (U_1 = \texttt{ins} \textit{ or } \texttt{del} \textit{ or } \texttt{upd}) \\ (U_1; U_2) \bullet U_3 & = & U_1; U_2; U_3 \\ (U_1 \oplus U_2) \bullet U_3 & = & (U_1 \bullet U_3) \oplus (U_2 \bullet U_3) \end{array}$$

Figure 3.10: Definition of the composition operator.

first phase to synthesize a more general sketch that accounts for this ambiguity.

Figure 3.9 describes the second phase of sketch generation using judgments of the form $\Phi \vdash s \twoheadrightarrow \Omega$, and the composition operator $\bullet$ is defined in Figure 3.10. At a high level, the rules in Figure 3.9 compose the sketches obtained during the first phase to obtain a more general sketch. To start with, the Lift rule corresponds to a base case and states that the $\twoheadrightarrow$ relation is initially obtained using the $\rightsquigarrow$ relation. The Query rule composes multiple sketches $\Omega_1, \ldots, \Omega_n$ for a query statement Q as $\Omega_1 \oplus \ldots \oplus \Omega_n$ — i.e., the composed sketch is a union of the individual sketches.

The Update rule is similar to Query, but it is slightly more involved. In particular, suppose that we have two different sketches Ω_1, Ω_2 for an update

statement U. Now, we need to account for the possibility that either one or *both* of the updates may happen. Thus, the corresponding sketch for update statements is $\Omega_1 \oplus \Omega_2 \oplus (\Omega_1; \Omega_2)$ rather than the simpler sketch $\Omega_1 \oplus \Omega_2$ for query statements. The Update rule generalizes this discussion to arbitrarily many sketches by using a binary operator $\bullet$ (defined in Figure 3.10) that distributes sequential composition (;) over the choice ($\oplus$) construct. Finally, the Seq rule allows generating a sketch for $U_1; U_2$ using the sketch Ω_i for each U_i.

Given a statement s in the source program, its corresponding sketch Ω is obtained by applying the rewrite rules from Figure 3.9 to a fixed-point. Specifically, let $\Omega_1, \ldots, \Omega_n$ be the set of sketches such that $\Phi \vdash s \twoheadrightarrow \Omega_1, \ \ldots, \ \Phi \vdash s \twoheadrightarrow \Omega_n$, and let us say that a sketch Ω is more general than Ω', written $\Omega \succeq \Omega'$, if Ω represents more programs than Ω'. Then, the resulting sketch for s is the most general sketch Ω_i such that $\forall j \in [1, n].\ \Omega_i \succeq \Omega_j$.

3.3.4 Sketch Completion

In this section, we explain our algorithm for solving the database program sketches from Section 3.3.3. As mentioned earlier, we do not encode the precise semantics of the sketch using an SMT formula because relational algebra operators are difficult to express using standard first-order theories supported by SMT solvers. Instead, we perform symbolic search (using SAT) over the space of programs encoded by the sketch and then subsequently check equivalence. If the two programs are not equivalent, we employ *minimum fail-*

Algorithm 3 Sketch Completion

```
1:  procedure CompleteSketch(Ω, P)
    Input: Sketch Ω, Source program P
    Output: Target program P′ or ⊥ to indicate failure
2:      Ψ ← Encode(Ω);
3:      while SAT(Ψ) do
4:          M ← GetModel(Ψ);
5:          P′ ← Instantiate(Ω, M);
6:          done ← Verify(P, P′);
7:          if done then return P′;
8:          E ← MinCex(P, P′);
9:          Ψ ← Ψ ∧ Block(M, E);
10:     return ⊥;
```

ing inputs to further prune the search space by identifying programs that share the same root cause of failure as a previously encountered program.

Overview. Our sketch completion procedure is summarized in Algorithm 3 and takes as input a program sketch Ω together with the source program $\mathcal{P}$. The output of CompleteSketch is either a completion $\mathcal{P}'$ of Ω such that $\mathcal{P} \simeq \mathcal{P}'$ or $\perp$ to indicate no such program exists.

At a high level, the CompleteSketch procedure first generates a boolean formula Ψ that represents *all* possible completions of the sketch Ω (line 2). While any model of Ψ corresponds to a concrete program $\mathcal{P}'$ that is an instantiation of Ω, such a program $\mathcal{P}'$ may or may not be equivalent to the input program $\mathcal{P}$. Thus, the sketch solving algorithm enters a loop (lines 3–9) that lazily explores different instantiations of Ω, checks equivalence, and adds

useful blocking clauses to the SAT encoding Ψ as needed. In what follows, we explain the algorithm (and its subroutines) in more detail.

Initial SAT encoding. The goal of the ENCODE procedure at line 2 is to generate a SAT formula that encodes all possible completions of Ω. Specifically, for each hole $??_i\{e_1, \ldots, e_n\}$ in the sketch, we introduce n boolean variables $b_i^1, \ldots, b_i^n$ such that $b_i^j = \mathit{true}$ if and only if hole $??_i$ is instantiated with expression e_j. [4] Since any valid completion of sketch Ω must assign every hole $??_i$ to some expression e_j in its domain, our initial SAT encoding is obtained as follows:

$$\Psi = \bigwedge_{??_i \in Holes(\Omega)} \oplus(b_i^1, \ldots, b_i^{i_n})$$

where the domain of $??_i$ consists of expressions $e_1, \ldots e_{i_n}$, and $\oplus$ denotes the n-ary xor operator. Observe that every model $\mathcal{M}$ of formula Ψ corresponds to one particular instantiation of Ω; thus, the procedure Instantiate produces program $\mathcal{P}'$ by assigning hole $??_i$ to expression e_j if and only if $\mathcal{M}$ assigns variable b_i^j to true.

Verification and blocking clauses. As is apparent from the discussion above, our symbolic encoding Ψ of the sketch intentionally does not enforce equivalence between source and target programs. Thus, whenever we obtain a

[4] Since the choice construct $s_1 \,?\, s_2$ is just syntactic sugar for **if** $??\{\top, \bot\}$ **then** s_1 **else** s_2, we assume it has been de-sugared before this SAT encoding.

completion $\mathcal{P}'$ of the sketch, we must check whether $\mathcal{P}, \mathcal{P}'$ are actually equivalent using the Verify subroutine at line 6 of Algorithm 3. If the two programs are indeed equivalent, the algorithm terminates with $\mathcal{P}'$ as a solution. Otherwise, in the next iteration, we ask the SAT solver for a different model, which corresponds to a different instantiation of the input sketch. However, in practice, there are an enormous number (e.g., up to 10^{39}) of completions of the sketch; thus, a synthesis algorithm that tests equivalence for every possible sketch completion is unlikely to scale. Our sketch completion algorithm addresses this issue by using minimum failing inputs to block *many* programs at the same time.

Specifically, a *minimum failing input* for a pair of programs $\mathcal{P}, \mathcal{P}'$ is an invocation sequence ω (recall Section 2.2.3) satisfying the following criteria:

1. We have $[\![\mathcal{P}]\!]_\omega \neq [\![\mathcal{P}']\!]_\omega$. That is, ω is a witness to the disequivalence of $\mathcal{P}$ and $\mathcal{P}'$

2. There does not exist another invocation sequence ω' such that $|\omega'| < |\omega|$ and $[\![\mathcal{P}]\!]_{\omega'} \neq [\![\mathcal{P}']\!]_{\omega'}$

Intuitively, minimum failing inputs are useful in this context because they provide feedback about which assignments to which holes cause program $\mathcal{P}'$ to *not* be equivalent to $\mathcal{P}$. Specifically, let $\mathcal{H}$ (resp. $\overline{\mathcal{H}}$) be the holes used in functions that appear (resp. do *not* appear) in ω, and let $A_{\mathcal{H}}$ denote the assignments to holes $\mathcal{H}$. Then, any program that instantiates Ω by assigning

$A_{\mathcal{H}}$ to $\mathcal{H}$ will also be incorrect, regardless of the assignments to holes $\overline{\mathcal{H}}$. Our sketch completion algorithm uses this observation to rule out many programs beyond $\mathcal{P}'$. Specifically, let $\mathcal{H} = \{??_1, \ldots, ??_n\}$ and suppose that $A_{\mathcal{H}}$ assigns expression e_{k_i} to each $??_i$. Then, the Block procedure (line 9 of Algorithm 3) generates the following blocking clause:

$$\varphi = \neg(b_1^{k_1} \wedge \ldots \wedge b_n^{k_n})$$

Intuitively, this blocking clause φ rules out all completions of Ω that agree with $\mathcal{P}'$ on the assignment to holes in $\mathcal{H}$. Since minimum failing inputs typically involve a small subset of the methods in the program, this technique allows us to rule out *many* programs in one iteration. Furthermore, as we discuss in Section 3.4, minimum failing inputs are inexpensive to obtain using testing.

3.4 Implementation

We have implemented the proposed synthesis technique in a new tool called Migrator, which is implemented in Java. Migrator uses the Sat4J solver [77] for answering all SAT and MaxSAT queries and the Mediator tool [130] for verifying equivalence between a pair of database programs. In the remainder of this section, we discuss two important design choices about our implementation.

Sketch generation. Recall from Section 3.3.3 that our sketch generation algorithm produces a sketch using a so-called *join correspondence*, which in turn

is synthesized from a candidate value correspondence. While our presentation in Section 3.3.3 presents "type-checking" rules that determine whether a join correspondence is valid with respect to some value correspondence, it can be inefficient to consider all possible join chains in the target schema and then check whether they are feasible. Thus, rather than taking an enumerate-then-check approach, our implementation *algorithmically* produces join correspondences that are feasible with respect to a given value correspondence.

To see how we infer all target join chains that may correspond to a source join chain J, suppose we are given a value correspondence Φ and let A be the set of attributes that occur in J. Our goal is to find all join chains $J_1, \ldots, J_n$ over the target schema such that for every attribute $a \in A$, there is a corresponding attribute $a' \in \mathit{Attrs}(J_i)$. We reduce the problem of finding all such possible join chains to the problem of finding all possible Steiner trees [68] over a graph data structure where nodes represent tables and edges represent join-ability relations.

In more detail, let A' be a set of attributes over the target schema such that for every $a \in A$, there exists some $a' \in A'$ where $a' \in \Phi(a)$, and let $\mathcal{T}'$ denote the set of tables containing all attributes in A'. Since the source join chain refers to all attributes in A, we need to find exactly those join chains over the target schema that "cover" the relations in which A' appears. Towards this goal, we construct a graph data structure $G = (V, E)$ as follows: The nodes V are tables in the target schema, and there is an edge (T, T') if tables T and T' can be joined with each other. Now, recall that, given a graph $G = (V, E)$

and a set of vertices $V' \subseteq V$, a Steiner tree is a connected subgraph that spans all vertices V'. Since our goal is to "cover" exactly the tables $\mathcal{T}'$ in the target schema, we compute all possible Steiner trees spanning $\mathcal{T}'$ and convert them to join chains in the expected way.

Generating minimum failing inputs. Recall from Section 3.3.4 that our sketch completion algorithm uses minimum failing inputs to prune the search space. In our implementation, we generate such inputs using a bounded testing procedure. Specifically, we generate a fixed set of constants for each type (e.g., $\{0, 1\}$ for integers) as the seed set to be used for arguments. Then, given such a seed set C of constants, our testing engine generates all possible invocation sequences containing only constants from C in increasing order of length. For each invocation sequence ω, we execute both $\mathcal{P}$ and $\mathcal{P}'$ on ω and check if the outputs are different. If so, we return ω as a minimum failing input, and otherwise, we test equivalence using the next invocation sequence.

Verification. Our sketch completion algorithm from Section 3.3.4 invokes a Verify procedure to check if two programs are equivalent. However, since full-fledged verification using the MEDIATOR tool [130] can be quite expensive, we first perform exhaustive testing up to some bound and invoke MEDIATOR only when no failing inputs are found. In principle, it is possible that the testing procedure fails to find a failing input while the verifier cannot establish equivalence. We have not encountered this kind of scenario in practice, but it could nonetheless happen in theory.

Table 3.1: Main experimental results of MIGRATOR.

	Benchmark	Funcs	Source Schema		Target Schema		Value Corr	Iters	Synth Time(s)	Total Time(s)
			Tables	Attrs	Tables	Attrs				
textbook bench	Oracle-1	4	2	8	1	6	1	1	0.3	2.7
	Oracle-2	19	3	17	7	25	1	5	0.5	11.3
	Ambler-1	10	1	6	2	7	1	2	0.3	2.9
	Ambler-2	10	2	7	1	6	1	1	0.3	0.6
	Ambler-3	7	2	5	2	5	2	5	0.4	30.6
	Ambler-4	5	1	2	1	2	1	1	0.3	0.5
	Ambler-5	8	2	5	3	6	5	7	0.3	3.1
	Ambler-6	10	2	9	2	8	1	1	0.3	0.7
	Ambler-7	8	2	7	2	8	1	1	0.3	0.6
	Ambler-8	14	3	10	3	13	1	7	0.5	3.1
real-world bench	cdx	138	16	125	17	131	1	7	11.9	38.9
	coachup	45	4	51	5	55	1	10	1.8	6.7
	2030Club	125	15	155	16	159	1	2	5.2	24.8
	rails-ecomm	65	8	69	9	75	1	6	2.5	10.3
	royk	151	19	152	19	155	1	17	46.1	60.1
	MathHotSpot	54	7	38	8	42	6	11	1.2	5.8
	gallery	58	7	52	8	57	1	11	2.5	9.4
	DeeJBase	70	10	92	11	97	1	8	3.5	9.3
	visible-closet	263	26	248	27	252	1	108	1304.7	1370.8
	probable-engine	85	12	83	11	78	1	9	4.6	17.5
	Average	**57.5**	**7.2**	**57.1**	**7.8**	**59.4**	**1.5**	**11.0**	**69.4**	**80.5**

3.5 Evaluation

To evaluate the proposed idea, we use MIGRATOR to automatically migrate 20 database programs to a new schema.

Benchmarks. All 20 programs in our benchmark set are taken from Section 2.7 for verifying equivalence between database programs.[5] Specifically, half of these benchmarks are adapted from textbooks and online tutorials, and

[5] While we consider 21 benchmarks in Section 2.7, one of these benchmarks cannot be verified by MEDIATOR. Since we use MEDIATOR as our verifier, we exclude that one benchmark from this evaluation.

the remaining half are manually extracted from real-world web applications on Github. However, because the input language of MIGRATOR is slightly different from that of MEDIATOR, we write a translator to convert the database programs to MIGRATOR's input language.

Experimental Setup. All of our experiments are conducted on a machine with Intel Xeon(R) E5-1620 v3 quad-core CPU and 32GB of physical memory, running the Ubuntu 14.04 operating system. For each synthesis benchmark, we set a time limit of 24 hours.

3.5.1 Main Results

Our main experimental results are summarized in Table 3.1. Here, the first ten rows correspond to benchmarks taken from database schema refactoring textbooks, and the latter ten rows correspond to real-world Ruby-on-Rails applications collected from Github. The "Funcs" column shows the number of functions that need to be synthesized. The next two columns under "Source Schema" (resp. "Target Schema") describe the number of tables and attributes in the source (resp. target) schema. The last four columns report the results obtained by running MIGRATOR on each benchmark. Specifically, the column "Value Corr" shows the number of value correspondences considered by MIGRATOR, and "Iters" shows the number of programs explored before an equivalent one is found. Finally, the "Synth Time" column shows synthesis time in seconds (excluding verification), and "Total Time" shows total time,

including both synthesis and verification.

The key takeaway message from this experiment is that MIGRATOR can successfully synthesize equivalent versions of all 20 benchmarks, including the database programs in real-world Ruby-on-Rails web applications with up to 263 functions. Furthermore, synthesis time (excluding verification) ranges from 0.3 seconds to 1304.7 seconds, with the average time being 69.4 seconds in total or 1.2 seconds per function. We believe these results provide strong evidence that our proposed technique can be quite useful for automating the code migration process for schema refactoring.

3.5.2 Comparison with Baselines

Given that there are other existing techniques for solving program sketches, we also evaluate our sketch completion algorithm by comparing our method against two baselines. In particular, our first baseline is the SKETCH tool [117], and the second one is a variant of our own sketch completion algorithm that does not use minimum failing inputs (MFIs).

Comparison with Sketch. To compare our approach with the SKETCH tool [117], we first implemented the semantics of SQL in SKETCH by encoding each SQL statement as a C function. Specifically, our SKETCH encoding models each database table as an array of arrays, with the nested array representing a tuple, and we model each SQL operation as a function that reads and updates the array as appropriate.

Table 3.2: Comparing MIGRATOR with SKETCH.

	Benchmark	Sketch Synth Time(s)	Speedup of Migrator
textbook bench	Oracle-1	88.2	294.0x
	Oracle-2	>86400.0	>172800.0x
	Ambler-1	3136.5	10455.0x
	Ambler-2	71.5	238.3x
	Ambler-3	74.7	186.8.5x
	Ambler-4	1.6	5.3x
	Ambler-5	494.4	1648.0x
	Ambler-6	226.2	754.0x
	Ambler-7	814.8	2716.0x
	Ambler-8	>86400.0	>172800.0x
real-world bench	cdx	>86400.0	>7260.5x
	coachup	>86400.0	>48000.0x
	2030Club	>86400.0	>16615.4x
	rails-ecomm	>86400.0	>34560.0x
	royk	>86400.0	>1874.2x
	MathHotSpot	>86400.0	>72000.0x
	gallery	>86400.0	>34560.0x
	DeeJBase	>86400.0	>24685.7x
	visible-closet	>86400.0	>66.2x
	probable-engine	>86400.0	>18782.6x
	Average	**>52085.4**	**>750.5x**

The results of this experiment are summarized in Table 3.2. The main observation is that SKETCH times out on all real-world benchmarks from Github as well as two textbook examples, namely Oracle-2 and Ambler-8. For all other benchmarks, MIGRATOR is significantly faster than SKETCH, with speed-ups ranging between 5.3x to 10455.0x in terms of synthesis time. Since SKETCH only performs bounded model checking rather than full-fledged verification, we only report speedup in terms of synthesis time rather than total time including verification. The speedup in terms of total time (including verification) ranges from 2.4x to 1358.0x. We believe this experiment demon-

Table 3.3: Comparing MIGRATOR with symbolic enumerative search.

	Benchmark	Symbolic Enum		Speedup of Migrator
		Iters	Synth Time(s)	
textbook bench	Oracle-1	1	0.3	1.0x
	Oracle-2	5	0.5	1.0x
	Ambler-1	2	0.3	1.0x
	Ambler-2	1	0.3	1.0x
	Ambler-3	6	0.4	1.0x
	Ambler-4	1	0.3	1.0x
	Ambler-5	11	0.4	1.3x
	Ambler-6	1	0.3	1.0x
	Ambler-7	1	0.3	1.0x
	Ambler-8	67996	54367.6	108735.2x
real-world bench	cdx	5595	6169.4	518.4x
	coachup	1303	76.2	42.3x
	2030Club	2	5.2	1.0x
	rails-ecomm	2779	602.5	241.0x
	royk	>31249	>86400.0	>1874.2x
	MathHotSpot	115	5.3	4.4x
	gallery	21483	32266.2	12906.5x
	DeeJBase	605	142.8	40.8x
	visible-closet	>9512	>86400.0	>66.2x
	probable-engine	1661	540.3	117.5x
	Average	**>7116.5**	**>13348.9**	**>192.3x**

strates the advantage of our proposed sketch completion algorithm compared to the standard CEGIS approach implemented in SKETCH.

Comparison with enumerative search. Since the key novelty of our sketch completion algorithm is the use of minimum failing inputs to prune the search space, we also compare our approach against a baseline that does not use MFIs. In particular, this baseline uses the same SAT encoding of the search space but blocks only a single program at a time. More concretely, given a model $\mathcal{M}$ of the SAT encoding Ψ, this baseline updates Ψ by con-

joining $\neg\mathcal{M}$ whenever verification fails. Effectively, this baseline performs enumerative search but does so in a symbolic way using a SAT solver.

The results of this experiment are summarized in Table 3.3. As we can see from this table, the impact of MFIs is particularly pronounced for the Ambler-8 textbook example and almost all real-world benchmarks. In particular, MIGRATOR is 192.3x faster than enumerative search on average. Moreover, without using MFIs to prune the search space, two of the benchmarks do not terminate within a time-limit of 24 hours. Hence, the results demonstrate that our MFI-based sketch completion is very important for practical synthesis.

3.6 Limitations

In this section, we will explain and discuss some limitations of the MIGRATOR tool.

First, MIGRATOR cannot handle schema changes that are not expressible using our notion of value correspondence. For example, one can merge two columns "first name" and "last name" into a single column "name" and use string operations to extract first or last names in a query. These types of schema refactorings cannot be expressed using our definition of value correspondence. While it is relatively straightforward to expand our technique to a richer scope of value correspondences (e.g., by enriching the sketch language to include a set of predefined functions like `concat`, `split`, etc), this change would require a more sophisticated verifier that can reason about the semantics of built-in functions.

Second, MIGRATOR does not synthesize database programs with control-flow constructs such as if statements and loops, because the underlying equivalence verifier [130] does not support database programs with those constructs.

Third, the notion of equivalence considered in this chapter characterizes *behavioral equivalence* between database programs, which ensures that two corresponding sequences of transactions yield the same result. However, it does not enforce that the underlying data stored in the database is inserted or manipulated in particular ways. For example, MIGRATOR may choose to delete from one or multiple tables when performing deletion as long as the new program satisfies the behavioral equivalence requirement. In some contexts, it may be desirable to adopt a stronger definition of equivalence than the one we consider in this chapter.

Chapter 4

Data Migration via Datalog Synthesis [1]

In previous chapters, we have discussed formal method techniques for verifying the correctness of code migration and synthesizing a new database program automatically. Recall that schema refactoring not only requires migrating code but also migrating the underlying data to a new schema. To facilitate data migration for developers, in this chapter, we present a new programming-by-example technique for automatically migrating data from one schema to another.

Specifically, 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, 102, 140], 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.

One of the key ideas underlying our method is to reduce the automated

[1] This chapter is adapted from the author's previous publication [133], where the author led the technical discussion, tool development, and experimental evaluation.

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, 52] and tuple-generating dependencies [101], 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 4.1, our proposed synthesis algorithm consists of three steps:

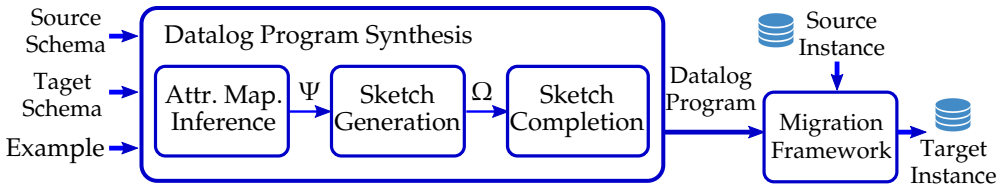


Figure 4.1: Schematic workflow of DYNAMITE.

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.

4.1 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 4.2(a) shows a small subset of the data in the source schema, and 4.2(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:

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

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*

```
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 4.2: Example database instances.

in the source document, and if (id_2, num) is a nested attribute of id_1, then there should be an *Admission* entry $(grad, ug, num)$ 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 4.2.

Attribute Mapping. 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 4.2, DYNAMITE infers the following attribute mapping Ψ:

$$\begin{array}{rcl@{\qquad}rcl} id & \to & \{uid\} & name & \to & \{grad, ug\} \\ uid & \to & \{id\} & count & \to & \{num\} \end{array}$$

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:

$$\begin{aligned} &\mathit{Admission}(\mathit{grad}, \mathit{ug}, \mathit{num}) \text{ :-} \\ &\quad \mathit{Univ}(??_1, ??_2, v_1), \mathit{Admit}(v_1, ??_3, ??_4), \\ &\quad \mathit{Univ}(??_5, ??_6, \_), \mathit{Univ}(??_7, ??_8, \_). \end{aligned} \tag{4.1}$$

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

Here, $??_i$ represents a *hole* (i.e., unknown) in the sketch, and its domain is indicated as $??_i \in \{e_1, \ldots, e_n\}$, meaning that hole $??_i$ can be instantiated with an element drawn 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 $\mathit{Univ}(\mathbf{??}_1, \mathbf{??}_2, v_1)$, $\mathit{Admit}(v_1, \mathbf{??}_3, \mathbf{??}_4)$ 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 4.1, our SMT encoding is the following formula Φ:

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

Here, for each hole $??_i$ 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.[2] 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:

$$(x_2 = grad \vee x_6 = grad) \wedge (x_2 = ug \vee x_6 = ug) \wedge (x_4 = num)$$

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

$$\begin{aligned}&x_1 = id_1 \wedge x_2 = grad \wedge x_3 = id_1 \wedge x_4 = num\\ \wedge\ &x_5 = id_1 \wedge x_6 = ug \wedge x_7 = id_2 \wedge x_8 = name_1\end{aligned} \tag{4.2}$$

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 4.3).

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

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

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 expected and actual outputs $\mathcal{O}$ and $\mathcal{O}'$ shown in Figure 4.3. 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:

$$\begin{array}{c}(x_4 = num \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4 \wedge x_1 = x_5 \\ \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8)\end{array} \tag{4.3}$$

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:

$$\begin{array}{c}x_1 = id_1 \wedge x_2 = grad \wedge x_3 = id_2 \wedge x_4 = num \\ \wedge x_5 = id_2 \wedge x_6 = ug \wedge x_7 = id_3 \wedge x_8 = name_1\end{array}$$

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

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

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

4.2 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.

4.2.1 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:

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

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 4.2.1. *Consider the JSON document schema from our motivating example in Section 4.1:*

```
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 4.2.2. *Consider the following relational schema:*

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

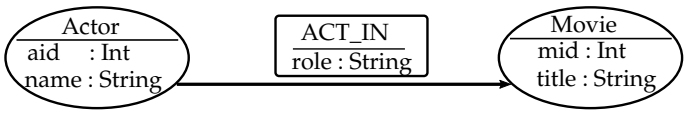
In our representation, this schema is represented as follows:

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

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

Figure 4.4: Syntax of Datalog programs.

Example 4.2.3. *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:*

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

4.2.2 Datalog

As shown in Figure 4.4, 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 chapter, 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 [29]. 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 $[\![\mathcal{P}]\!] \wedge \mathcal{I}$.

4.2.3 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 $\mathcal{R}'$ 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 $Id(r)$ 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:

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

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 4.2.4. *For the JSON document from Figure 4.2(a), our method generates the following Datalog facts*

$$\begin{array}{lll} Univ(1, \text{“}\texttt{U1}\text{”}, id_1) & Univ(2, \text{“}\texttt{U2}\text{”}, id_2) & Admit(id_1, 1, 10) \\ Admit(id_2, 2, 20) & Admit(id_1, 2, 50) & Admit(id_2, 1, 40) \end{array}$$

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 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 $BuildRecord(R_N, N) = \{a_1 : v_1, \ldots, a_n : v_n\}$ where

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

Algorithm 4 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 $\perp$ 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(Ω);
5: **while** SAT(Φ) **do**
6: $\sigma \leftarrow$ GetModel(Φ);
7: $\mathcal{P} \leftarrow$ Instantiate(Ω, σ);
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** $\perp$;

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

4.3 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.

4.3.1 Algorithm Overview

The top-level algorithm for synthesizing Datalog programs is summarized in Algorithm 4. 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 $\perp$ to indicate that the desired data migration task cannot be represented as a Datalog program.

As shown in Algorithm 4, the SYNTHESIZE procedure first invokes the INFERATTRMAPPING procedure (line 2) to infer an attribute mapping Ψ. Specifically, Ψ is a mapping from each $a \in \mathit{PrimAttrbs}(\mathcal{S})$ to a set of attributes $\{a_1, \ldots, a_n\}$ where $a_i \in \mathit{PrimAttrbs}(\mathcal{S}) \cup \mathit{PrimAttrbs}(\mathcal{S}')$ 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 4.1, 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.

4.3.2 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 4.4, except that it allows holes (denoted by **??**) as special constructs indicating unknown expressions. As summarized in Algorithm 5, 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. [3] 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 inten-

[3] The property of the generated sketch is characterized and proved in [134].

Algorithm 5 Generating Datalog program sketches

1: **procedure** SketchGen($\Psi, \mathcal{S}, \mathcal{S}'$)
   Input: Attribute mapping Ψ, source schema $\mathcal{S}$, target schema $\mathcal{S}'$
   Output: Program sketch Ω
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** Ω;
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: $N \leftarrow \mathsf{RecName}(a)$;
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: $n \leftarrow \mathsf{CopyNum}(B, \mathsf{RecName}(a'))$;
17: $V \leftarrow V \cup \bigcup_{i=1}^{n} \{v^i_{a'}\}$;
18: $B \leftarrow B[??_a \mapsto ??_a \in V]$;
19: **return** H :- B.;

sional 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.

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

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

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

Figure 4.5: 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 4.5 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 InRecNested) 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 5). Given a record type N in the source schema and its corresponding predicate(s) R_N, the loop in lines 9–12 of Algorithm 5 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 5 invokes a procedure called GenExtensionalPreds, described in Figure 4.6, 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 4.6 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 4.3.1. *Suppose we want to generate the body sketch for the rule associated with record type* $T : \{a' : \texttt{Int},\ \texttt{b}' : \texttt{Int}\}$ *in the target schema. Also, suppose we are given the attribute mapping* Ψ *where* $\Psi(a) = a'$ *and* $\Psi(b) =$

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

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

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

Figure 4.6: Inference rules for GenExtensionalPreds.

b' and source attributes a, b belong to the following record type in the source schema: $C : \{a : \texttt{Int},\ \texttt{D} : \{\texttt{b} : \texttt{Int}\}\}$. 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 4.6, we generate predicate $R_C(??_a, v_D^1)$, where $??_a$ 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 $R_C(??_a, v_D^2)$ and $R_D(v_D^2, ??_b)$. Putting them together, we obtain the following sketch body:

$$R_C(??_a, v_D^1), R_C(??_a, v_D^2), R_D(v_D^2, ??_b)$$

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 5) 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 $??_x$ associated with attribute x, the domain of $??_x$ consists of all the variables associated with attribute x or one of its aliases.

Example 4.3.2. *Consider the same schemas and attribute mapping from Example 4.3.1 and the following body sketch:*

$$R_C(??_a, v_D^1), R_C(??_a, v_D^2), R_D(v_D^2, ??_b)$$

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

4.3.3 Sketch Completion

While the sketch generated by Algorithm 5 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 4.3.1, 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 $??_i$ 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 $??_i \in \{v_1, \ldots, v_n\}$, 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:

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

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 4) invokes a function called Analyze to find useful blocking clauses to add to the

SMT encoding. This procedure is summarized in Algorithm 6 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 4.3.3. *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 [134]. □

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 4.3.4. *Consider again the sketch from Section 4.1:*

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

$$\begin{gathered}??_1, ??_3, ??_5, ??_7 \in \{\mathit{id}_1, \mathit{id}_2, \mathit{id}_3, \mathit{uid}_1\} \; ??_4 \in \{\mathit{num}, \mathit{count}_1\} \\ ??_2, ??_6, ??_8 \in \{\mathit{grad}, \mathit{ug}, \mathit{name}_1, \mathit{name}_2, \mathit{name}_3\}\end{gathered}$$

Suppose the variable for $??_i$ is x_i and the assignment σ is:

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

Since grad, ug, and num occur in the head, Generalize(σ) yields the following formula:

$$\begin{gathered} x_2 = \mathit{grad} \wedge x_4 = \mathit{num} \wedge x_6 = \mathit{ug} \\ \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4 \wedge x_1 = x_5 \\ \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8 \end{gathered} \tag{4.4}$$

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 4.3.1. (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.

Example 4.3.5. *Consider the same sketch and assignment σ from Example 4.3.4, but now suppose we are given an MDP $\varphi = \{num\}$. Then the function Generalize(σ, φ) yields the following more general formula:*

$$\begin{array}{c} x_4 = num \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4 \wedge x_1 = x_5 \\ \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8 \end{array} \tag{4.5}$$

Note that (4.5) is more general (i.e., weaker) than (4.4) because it drops the constraints $x_2 = grad$ and $x_6 = ug$. Therefore, the negation of (4.5) is a better

Algorithm 6 Analyzing outputs to prune search space

```
1:  procedure ANALYZE(σ, O', O)
    Input: Model σ, actual output O', expected output O
    Output: Blocking clause φ
2:      φ ← true;
3:      Δ ← MDPSET(O', O);
4:      for each φ ∈ Δ do
5:          ψ ← true;
6:          for each (x_i, x_j) ∈ dom(σ) × dom(σ) do
7:              if σ(x_i) = σ(x_j) then ψ ← ψ ∧ x_i = x_j;
8:              else ψ ← ψ ∧ x_i ≠ x_j;
9:          for each x_i ∈ dom(σ) do
10:             if σ(x_i) ∈ φ then ψ ← ψ ∧ x_i = σ(x_i);
11:         φ ← φ ∧ ¬ψ;
12:     return φ;
```

blocking clause than the negation of (4.4), since it rules out more programs in one step.

Based on this discussion, we now explain the full ANALYZE procedure in Algorithm 6. 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 7 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

Algorithm 7 Computing a set of MDPs

```
1:  procedure MDPSet(O', O)
    Input: Actual output O', expected output O
    Output: A set of minimal distinguishing projections Δ
2:      Δ ← ∅;  V ← ∅;
3:      W ← EmptyQueue();
4:      for each a ∈ Attributes(O') do
5:          W.Enqueue({a}); V ← V ∪ {{a}};
6:      while ¬W.IsEmpty() do
7:          L ← W.Dequeue();
8:          if Π_L(O') = Π_L(O) then
9:              for each a' ∈ Attributes(O') \ L do
10:                 L' ← L ∪ {a'};
11:                 if L' ∉ V then
12:                     W.Enqueue(L');
13:                     V ← V ∪ {L'};
14:         else if ∄L'' ∈ Δ. L'' ⊆ L then Δ ← Δ ∪ {L};
15:     return Δ;
```

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 4.3.6. *Let us continue with Example 4.3.5 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 Generalize*(σ, φ_1) *(see formula (4.5) of Example 4.3.5), we also compute*

Generalize(σ, φ_2) *as:*

$$x_2 = grad \wedge x_6 = ug \wedge x_1 \neq x_2 \wedge x_1 = x_3 \wedge x_1 \neq x_4$$
$$\wedge x_1 = x_5 \wedge x_1 \neq x_6 \wedge x_1 \neq x_7 \wedge x_1 \neq x_8 \wedge \cdots \wedge x_7 \neq x_8$$

By adding both blocking clauses $\neg$*Generalize*(σ, φ_1) *as well as* $\neg$*Generalize*(σ, φ_2)*, we can prune even more incorrect Datalog programs.*

Theorem 4.3.7. *Let* ϕ *be a blocking clause returned by the call to* Analyze *at line 10 of Algorithm 4. If* σ *is a model of* $\neg\phi$*, then* σ *corresponds to an incorrect Datalog program.*

Proof. See [134]. □

4.4 Implementation

We have implemented the proposed technique as a new tool called Dynamite. Internally, Dynamite uses the Z3 solver [42] for answering SMT queries and leverages the Souffle framework [70] for evaluating Datalog programs. In the remainder of this section, we discuss some extensions over the synthesis algorithm described in Section 4.3.

Interactive mode. In Section 4.3, 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 $\mathcal{I}'$.

Example 4.4.1. *Suppose the source database contains two relations*

$$Employee(name, deptId) \quad Department(id, deptName)$$

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.*

Filtering operation. While the synthesis algorithm described in Section 4.3 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 4.2.3. However, Dynamite performs one optimization to

make large-scale data migration practical: We leverage MongoDB [88] 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.

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

RQ4 How does the proposed synthesis technique compare against prior techniques?

Benchmarks. To answer these research questions, we collected 12 real-world database instances (see Table 4.1 for details) and created 28 benchmarks in total. Specifically, four of these datasets (namely Yelp, IMDB, Mondial, and DBLP) are taken from prior work [140], and the remaining eight are taken from open dataset websites such as Kaggle [71]. For the document-to-relational transformations, we used exactly the same benchmarks as prior work [140]. For

Table 4.1: Datasets used in the evaluation of DYNAMITE.

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

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 4.2, where "R" stands for relational, "D" stands for document, and "G" stands for graph, our 28 benchmarks collectively cover a broad range of migration scenarios between different types of databases.[4]

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.

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

Table 4.2: Statistics of benchmarks for DYNAMITE.

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**

4.5.1 Main Synthesis Results

In this section, we evaluate RQ1 by using Dynamite to migrate the datasets from Table 4.1 for the source and target schemas from Table 4.2. 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 4.3, 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 Table 4.1.[5] We now highlight the key take-away lessons from this experiment whose results are summarized in Table 4.3. Note that the average search space size is calculated by geometric mean; all other averages are arithmetic mean.

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 4.3, 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).

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

Table 4.3: Main experimental results of DYNAMITE.

Bench	Avg # Examples		Search Space	Synth Time(s)	#Rules	#Preds per Rule	#Optim Rules	Dist to Optim	Mig 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**

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 4.3, 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 "Mig 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.

4.5.2 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 this experiment, we first fix the number r of

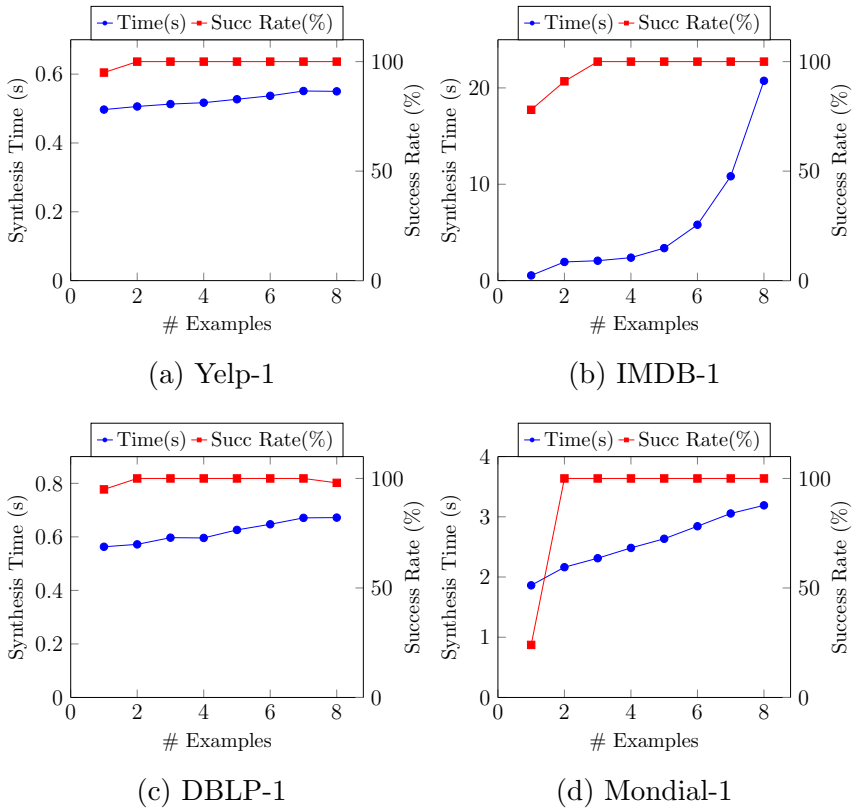


Figure 4.7: Sensitivity analysis of DYNAMITE.

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.

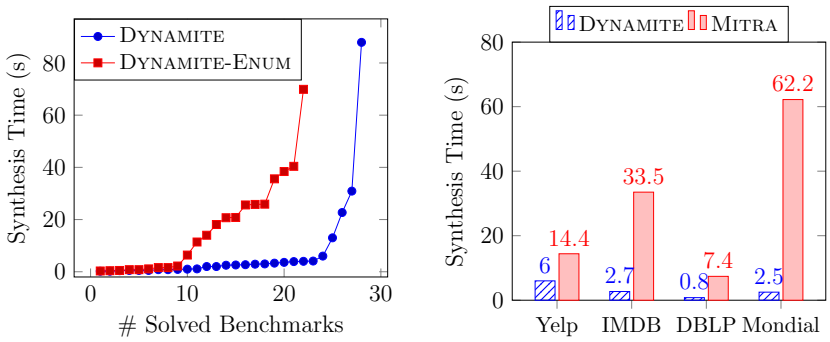
The results of this experiment are summarized in Figure 4.7 for four representative benchmarks (the remaining 24 are provided in [134]). Here,

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.

4.5.3 Comparison with Synthesis Baseline

To answer RQ3, we compare DYNAMITE against a baseline called DYNAMITE-ENUM that uses enumerative search instead of the sketch completion



(a) Comparison with enum (b) Comparison with Mitra

Figure 4.8: Comparing Dynamite with baseline and Mitra.

technique described in Section 4.3.3. 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.

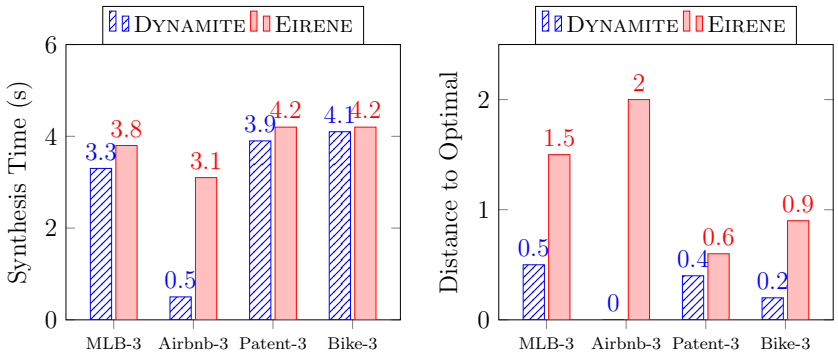
Figure 4.8(a) shows the results of the comparison when using the manually-provided input-output examples from Section 4.5.1. In particular, we plot the time in seconds that each version takes to solve the first n benchmarks. As shown in Figure 4.8(a), Dynamite can successfully solve all 28 benchmarks 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 advan-

tages of our proposed sketch completion algorithm compared to a simpler enumerative-search baseline.

4.5.4 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 specialized data migration scenarios. Specifically, Mitra [140] 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 [140] that involve conversion from a document schema to a relational schema. The results of this comparison are summarized in Figure 4.8(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,



(a) Synthesis Time (b) Mapping Quality

Figure 4.9: Comparing Dynamite against Eirene.

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 4.5.1 using the same input-output examples. As shown in Figure 4.9(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 4.5.1.[6] As

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

shown in Figure 4.9(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.

4.6 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 [48, 84].

Chapter 5

Related Work [1]

The research problems addressed in this dissertation are related to a long line of work about relational verification, database application analysis, program synthesis, and data migration. In this chapter, we survey papers that are most relevant and explain how they differ from our techniques.

5.1 Relational Verification

Formal verification of software programs typically refers to the act of proving a system satisfies a specification of the system's behavior. Among different domains of software verification, relational verification refers to the problem of proving two programs or two runs of the same program satisfy a relational specification, such as equivalence, inversion, etc.

Relational program logics. Formally, the relational specification is described as relational Hoare triples of the form $\{P\}\ S_1 \sim S_2\ \{Q\}$. Here, P is a relational pre-condition that relates inputs to programs S_1, S_2, and Q is a relational post-condition that relates their outputs. In the context of equivalence

[1] This chapter is adapted from the author's previous publications [130, 131, 133], where the author led the technical discussion, tool development, and experimental evaluation.

checking, the pre-condition simply assumes equality between program inputs, and the post-condition asserts equality between their outputs. Prior work has presented program logics, such as Relational Hoare Logic and Cartesian Hoare logic, for showing such relational correctness properties [23, 120, 144]. Another common technique for proving relational correctness properties is to construct a product program $S_1 \times S_2$, which is semantically equivalent to $S_1; S_2$ but that is somehow easier to verify [19, 20]. In contrast to existing relational verification techniques that work on imperative programs, MEDIATOR addresses database programs that work over different schemas. Furthermore, while the main focus of MEDIATOR is to verify equivalence/refinement between programs, we believe our technique can be easily extended for proving other relational correctness properties.

Translation validation. One of the most well-known applications of relational verification is *translation validation*, where the goal is to prove that the compiled version of the code is equivalent to the original one [94, 97, 105, 121, 145, 146]. More recent work extends translation validation to *parameterized equivalence checking* (PEC), which aims to prove equivalence between templatized programs representing many concrete programs [75]. Most of the work in this area focuses on imperative programs and proves equivalence by inferring some form of bisimulation relation. As mentioned earlier, another common technique for proving equivalence is to generate a product program [19, 145] and reduce the equivalence checking problem to the safety verification of a sin-

gle program. Rather than validating the correctness of compiler optimizations, the goal of MEDIATOR is to show equivalence between database programs before and after changes to the database schema. Our bisimulation invariants relate database states rather than program variables and are expressed in the theory of relational algebra with updates instead of standard first-order theories directly supported by SMT solvers.

Contextual equivalence. There has also been a significant body of work on verifying *contextual equivalence*, where the goal is to determine whether two expressions are equivalent under any program context. One important application of contextual equivalence is to identify compiler optimization opportunities, particularly in the context of functional programming languages. For example, Sumii and Pierce define an untyped call-by-value lambda calculus with sealing and present a bisimulation-based approach to prove contextual equivalence [122]. They later present another sound and complete proof methodology based on bisimulation relations, but apply it to a lambda calculus with full universal, existential, and recursive types [123]. Koutavas and Wand extend this line of work to prove contextual equivalence in an untyped lambda calculus with an explicit store by introducing a new notion of bisimulation. Their method enables constructive proofs in the presence of higher-order functions [74]. They also extend the same proof technique to the imperative untyped object calculus [73]. Sangiorgi et al. step further and develop a notion of *environmental bisimulation* for higher-order languages. Their

technique does not require induction on evaluation derivations and is applicable to different calculi, ranging from pure lambda calculus to higher-order π-calculus [109, 110]. Existing techniques for proving contextual equivalence offer a limited degree of automation and do not address database programs.

Query equivalence. Query equivalence has been a long-standing relational verification problem in the database community. For example, Chandra and Merlin study the equivalence of conjunctive queries under set semantics and show that every conjunctive query has a unique representation up to isomorphism [30]. Aho et al. use tableaus to represent conjunctive queries and present a polynomial time algorithm for deciding equivalence between certain kinds of conjunctive queries [1]. Sagiv and Yannanakis generalize this tableau approach to union and difference operators, but place limitations on the use of difference [108]. In more recent work, Green studies equivalence between conjunctive queries and gives an algorithm for deciding their equivalence [60]. As mentioned in Section 2.6, our MEDIATOR implementation leverages insights from this work when checking validity of certain classes of $\mathcal{T}_{RA}$ formulas.

In the past few years, there has been significant activity on proving query equivalence using interactive theorem provers and constraint solvers. Specifically, Chu et al. define a new semantics for SQL based on K-Relations and homotopy type theory and develop a Coq library to interactively prove equivalence between SQL queries [34]. Another recent work by Chu et al. provides a greater degree of automation by incorporating constraint solvers [33].

Specifically, their tool, COSETTE, first translates each SQL query into a corresponding logical formula and uses an SMT solver to find a counterexample that shows the queries are *not* equivalent. If COSETTE fails to find a counterexample, it then tries to prove equivalence using the Coq theorem prover augmented with a library of domain-specific tactics. While tools like COSETTE could be useful for deciding validity of some of our $\mathcal{T}_{RA}$ formulas, existing tools do not support reasoning about updates to the database.

Schema equivalence. There has also been some work on proving schema equivalence under various different definitions of equivalence [21, 87, 107]. Under one definition, two schemas are considered equivalent if there is a bijection from the set of database instances from one schema to that of another [87, 107]. According to another definition, two schemas S_1, S_2 are equivalent if there is a query mapping from S_1 to S_2 such that its inverse is also a valid mapping from S_2 to S_1[3, 14, 67]. While many of these papers provide algorithms for deciding schema equivalence according to these definitions, they do not address the problem of verifying equivalence between applications that operate over databases with different schemas.

5.2 Database Application Analysis

Over the past decade, there has been significant interest in analyzing, verifying, and testing database applications [13, 22, 31, 45, 46, 49, 58, 61, 69, 80, 93, 103, 125, 136, 137]. Some of these techniques aim to verify integrity

constraints, find shallow bugs, or identify security vulnerabilities, while others attempt to uncover violations of functional correctness properties. For example, Benedikt et al. statically verify that database transactions preserve integrity constraints by means of computing weakest preconditions [22]. Itzhaky et al. [69] propose a technique to verify pre- and post-conditions of methods with embedded SQL statements. As another example, the Agenda framework generates test cases by randomly populating the database with tuples that satisfy schema constraints [31, 44], and several papers use concolic testing to find crashes or SQL injection vulnerabilities [12, 137]. Gligoric and Majumdar describe an explicit state model checking technique for database-driven applications and use this technique to find concurrency bugs [58].

There have also been proposals for checking functional correctness of database applications. For example, Near and Jackson present a bounded verifier that uses symbolic execution to check functional correctness properties specified using an extension of the RSpec specification language [93]. As another example, the WAVE project allows users to specify functional correctness properties using LTL formulas and model checks a given database application against this specification [45, 46, 125]. However, we are not aware of any existing work on verifying relational correctness properties of database applications.

5.3 Program Synthesis

Program synthesis refers to the task of generating a program in some language such that the program satisfies a given specification. It aims to free developers from tedious programming tasks or make programming accessible to end-users. Program synthesis has demonstrated its capability towards this end in various domains, including data wrangling [62], compiler superoptimization [111], and so on.

This dissertation is related to a long line of recent work on program synthesis [2, 7, 17, 26, 55, 62, 65, 66, 78, 82, 83, 98, 99, 111, 114, 117, 118, 119, 128, 135]. While the goal of program synthesis is always to produce a program that satisfies the given specification, different synthesizers use different forms of specifications, including input-output examples [17, 55, 62, 99, 128], logical constraints [117, 118, 119], refinement types [98], or a reference implementation [65, 82, 111]. In this work, MIGRATOR uses reference implementation and DYNAMITE uses input-output examples as the specification.

Synthesizing database programs. In recent years, there have been several papers that apply program synthesis to SQL queries or database programs [32, 43, 56, 79, 127, 141]. For instance, SQLIZER [141] synthesizes SQL queries from natural language, whereas SCYTHE [127] and MORPHEUS [56] generate queries from examples. The QBS system uses program synthesis to repair performance bugs in database applications [32]. In addition, FIAT [43] performs deductive synthesis to generate SQL-like operations from declara-

tive specifications. However, none of these techniques consider the problem of automatically migrating database programs for schema refactoring.

Schema evolution and program synthesis. In the database community, *schema evolution* refers to the problem of evolving database schemas to adapt to requirement changes. The code migration problem in schema evolution can be viewed as a program synthesis problem that uses reference implementation as the specification. There is a body of literature on schema evolution, including rewriting SQL queries and updates [27, 37, 38, 47, 100, 126]. Among these works, the most related one is the PRISM project and its successor PRISM++ [37, 38]. In addition to the original program and the source and target schemas, the PRISM approach requires the user to provide so-called *Schema Modification Operators (SMOs)* that describe how tables in the source schema are modified to tables in the target schema. The basic idea is to leverage these user-provided SMOs to rewrite SQL queries using the well-known *chase* and *backchase* algorithms [47, 100]. To deal with updates, they additionally require the user to provide *Integrity Constraint Modification Operators (ICMOs)* and "translate" updates into queries. In contrast to the PRISM approach, MIGRATOR does not require users to provide modification operators expressed in a domain-specific language. Although it is possible to explore the search space of SMOs and ICMOs to automate the generation of new database programs, we decided not to pursue this approach for two reasons: first, the rewriting technique in PRISM requires these modification operators to be in-

vertible, and, second, the search space of operator sequences is also potentially very large.

Inductive logic programming. As a particular type of program synthesis, inductive logic programming (ILP) aims to synthesize a logic program consistent with a set of examples [50, 63, 81, 89, 90, 106, 112, 143]. Among ILP techniques, DYNAMITE is most similar to recent work on Datalog program synthesis [2, 113]. In particular, Zaatar [2] 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 [113] also performs Datalog program synthesis from examples but additionally requires meta-rule templates. In contrast, DYNAMITE 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.

Conflict-driven learning. The synthesis techniques used in MIGRATOR and DYNAMITE bear similarities to recent work on *conflict-driven learning* in program synthesis [55, 85, 131] where the goal is to learn useful information from failed synthesis attempts. Among existing techniques, MIGRATOR is particularly related to the NEO tool [55], which uses conflict-driven learning to infer useful lemmas from failed synthesis attempts. MIGRATOR's sketch

solving algorithm from Section 3.3.4 can also be viewed as performing some form of conflict driven learning in that it uses minimum failing inputs to rule out many programs that share the same root cause of failure as the currently explored one. However, our technique is much more lightweight compared to Neo because it does not compute a minimum unsatisfiable core of the logical specification for the failing program. Instead, our technique exploits the observation that only a subset of the methods in a database program are necessary for proving disequivalence. Dynamite's sketch completion approach is based on a similar insight, but it uses Datalog-specific techniques to perform inference. In contrast to Migrator, Dynamite 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.

5.4 Schema Mapping and Data Migration

Data migration is the process of moving data from one schema to another. To express how to transform the data in this process, developers typically use schema mappings to describe the relationship between the source and target schemas.

Schema mapping formalisms. There are several different formalisms for expressing schema mappings, including visual representations [102, 104], schema modification operators [38, 40], and declarative constraints [4, 5, 6, 11, 28, 52,

72]. Some techniques require the user to express the schema mapping visually by drawing arrows between attributes in the source and target schemas [101, 104]. In contrast, schema modification operators express the schema mapping in a domain-specific language [38, 40]. Another common approach is to express the schema mapping using declarative specifications, such as Global-Local-As-View (GLAV) constraints [4, 5, 6, 52]. Similar to this third category, DYNAMITE expresses the schema mapping using a declarative, albeit executable, formalism.

Schema mapping inference. To help people easily create schema mappings for data migration, researchers have developed a body of techniques on *automatically inferring* schema mappings [5, 6, 10, 51, 57, 59, 86, 101, 102, 142]. For example, CLIO [57, 101] infers schema mappings for relational and XML schemas given a value correspondence between atomic schema elements. Another line of work [15, 96] uses model management operators such as ModelGen [24] 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 [25]. In contrast to DYNAMITE, [25] only focuses on relational-to-relational mappings [5]. Finally, MWEAVER [102] 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.

Data transformation using program synthesis. There has been significant work on automating data transformations using program synthesis [56, 64, 85, 115, 116, 127, 139, 140]. Many techniques focus only on table or string transformations [56, 64, 85, 115, 127], whereas HADES [139] (resp. MITRA [140]) focuses on document-to-document (resp. document-to-table) transformations. DYNAMITE generalizes prior work by automating transformations between many different types of database schemas. Furthermore, as we demonstrate in Section 4.5.4, this generality does not come at the cost of practicality, and, in fact, performs faster synthesis.

Universal and core solutions for data exchange. The data migration problem solved by DYNAMITE is related to the data exchange problem [52], 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 [52, 53]. 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 [29]. 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.

Chapter 6

Conclusion

This dissertation presents formal method techniques for database applications to help developers correctly and easily perform code and data migration during schema refactoring. First, we introduce the equivalence verification problem between database programs over different schemas and describe an automated verification algorithm for proving equivalence. The key insight is inferring a bisimulation invariant to relate two database states and show two programs always yield identical results given any input. Second, we present a synthesis technique for automated code migration during schema refactoring. To scale to real-world database applications, we develop an efficient sketch completion algorithm that utilizes counterexamples to rule out many incorrect programs at a time. Third, we describe an automatic data migration technique based on the programming-by-example paradigm. To enable efficient synthesis, our key idea is leveraging Datalog properties to analyze the root cause of the discrepancy between expected and actual outputs, and then prune many incorrect programs accordingly. We have implemented these techniques in three tools: Mediator, Migrator, and Dynamite, and our evaluation shows that these tools are effective for ensuring correctness and improving developer productivity during schema refactoring.

Bibliography

[1] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM J. Comput.*, 8(2):218–246, 1979.

[2] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In *Proceedings of CP*, pages 689–706, 2017.

[3] Joseph Albert, Yannis E. Ioannidis, and Raghu Ramakrishnan. Equivalence of keyed relational schemas by conjunctive queries. *J. Comput. Syst. Sci.*, 58(3):512–534, 1999.

[4] Bogdan Alexe, Phokion G. Kolaitis, and Wang-Chiew Tan. Characterizing schema mappings via data examples. In *Proceedings of PODS*, pages 261–272, 2010.

[5] Bogdan Alexe, Balder ten Cate, Phokion G. Kolaitis, and Wang Chiew Tan. Designing and refining schema mappings via data examples. In *Proceedings of SIGMOD*, pages 133–144, 2011.

[6] Bogdan Alexe, Balder Ten Cate, Phokion G Kolaitis, and Wang-Chiew Tan. Eirene: Interactive design and refinement of schema mappings via data examples. *PVLDB*, 4(12):1414–1417, 2011.

[7] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proceedings of FMCAD*, pages 1–8, 2013.

[8] Scott W Ambler. Test-driven development of relational databases. *IEEE Software*, 24(3), 2007.

[9] Scott W Ambler and Pramod J Sadalage. *Refactoring databases: Evolutionary database design*. Pearson Education, 2006.

[10] Yuan An, Alexander Borgida, Renée J. Miller, and John Mylopoulos. A semantic approach to discovering schema mapping expressions. In *Proceedings of ICDE*, pages 206–215, 2007.

[11] Patricia C. Arocena, Boris Glavic, and Renee J. Miller. Value invention in data exchange. In *Proceedings of SIGMOD*, pages 157–168, 2013.

[12] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *Proceedings of ISSTA*, pages 261–272, 2008.

[13] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010.

[14] Paolo Atzeni, Giorgio Ausiello, Carlo Batini, and Marina Moscarini. Inclusion and equivalence between relational database schemata. *Theor. Comput. Sci.*, 19:267–285, 1982.

[15] Paolo Atzeni, Paolo Cappellari, Riccardo Torlone, Philip A. Bernstein, and Giorgio Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.

[16] Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. *TOPLAS*, 27(2):314–343, 2005.

[17] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *Proceedings of ICLR*, 2017.

[18] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). www. *SMT-LIB. org*, 15:18–52, 2010.

[19] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proceedings of FM*, pages 200–214, 2011.

[20] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Proceedings of LFCS*, pages 29–43, 2013.

[21] Catriel Beeri, Alberto O. Mendelzon, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence of relational database schemes. In *Proceedings of STOC*, pages 319–329, 1979.

[22] Michael Benedikt, Timothy Griffin, and Leonid Libkin. Verifiable properties of database transactions. *Inf. Comput.*, 147(1):57–88, 1998.

[23] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of POPL*, pages 14–25, 2004.

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

[25] Angela Bonifati, Ugo Comignani, Emmanuel Coquery, and Romuald Thion. Interactive mapping specification with exemplar tuples. In *Proceedings of SIGMOD*, pages 667–682, 2017.

[26] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of PLDI*, pages 467–481, 2017.

[27] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. Synchronization of queries and views upon schema evolutions: A survey. *TODS*, 41(2):9:1–9:41, 2016.

[28] Balder Ten Cate, Phokion G Kolaitis, Kun Qian, and Wang-Chiew Tan. Approximation algorithms for schema-mapping discovery from data examples. *ACM Transactions on Database Systems*, 42(2):12, 2017.

[29] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE TKDE*, 1(1):146–166, 1989.

[30] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of STOC*, pages 77–90, 1977.

[31] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. A framework for testing database applications. In *Proceedings of ISSTA*, pages 147–157, 2000.

[32] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of PLDI*, pages 3–14, 2013.

[33] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *Proceedings of CIDR*, 2017.

[34] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: Proving query rewrites with univalent sql semantics. In *Proceedings of PLDI*, pages 510–524, 2017.

[35] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.

[36] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of PODS*, pages 155–166, 1999.

[37] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. *PVLDB*, 4(2):117–128, 2010.

[38] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.

[39] Carlo Curino, Hyun Jin Moon, Letizia Tanca, and Carlo Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In *Proceedings of ICEIS*, pages 323–332, 2008.

[40] Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Graceful database schema evolution: the PRISM workbench. *PVLDB*, 1(1):761–772, 2008.

[41] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron A. Peled, editors, *Proceedings of CAV*, pages 160–171, 1999.

[42] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, pages 337–340, 2008.

[43] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of POPL*, pages 689–700, 2015.

[44] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *Proceedings of ICSE*, pages 78–87, 2005.

[45] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. A verifier for interactive, data-driven web applications. In *Proceedings of SIGMOD*, pages 539–550, 2005.

[46] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[47] Alin Deutsch and Val Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *Proceedings of VLDB*, pages 201–212, 2003.

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

[49] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of ISSTA*, pages 151–162, 2007.

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

[51] Ronald Fagin, Laura M. Haas, Mauricio A. Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. Clio: Schema mapping creation and data exchange. In *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, pages 198–236, 2009.

[52] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[53] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 30(1):174–210, 2005.

[54] Stéphane Faroult and Pascal L'Hermite. *Refactoring SQL applications*. O'Reilly Media, 2008.

[55] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proceedings of PLDI*, pages 420–435, 2018.

[56] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of PLDI*, pages 422–436, 2017.

[57] Ariel Fuxman, Mauricio A. Hernández, C. T. Howard Ho, Renée J. Miller, Paolo Papotti, and Lucian Popa. Nested mappings: Schema mapping reloaded. In *Proceedings of VLDB*, pages 67–78, 2006.

[58] Milos Gligoric and Rupak Majumdar. Model checking database applications. In *Proceedings of TACAS*, pages 549–564, 2013.

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

[60] Todd J. Green. Containment of conjunctive queries on annotated relations. In *Proceedings of ICDT*, pages 296–309, 2009.

[61] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. Verifying equivalence of spark programs. In *Proceedings of CAV*, pages 282–300, 2017.

[62] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of POPL*, pages 317–330, 2011.

[63] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin G. Zorn. Inductive programming meets the real world. *Communication of the ACM*, 58(11):90–99, 2015.

[64] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of PLDI*, pages 317–328, 2011.

[65] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of PLDI*, pages 38–49, 2011.

[66] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Proceedings of PLDI*, pages 417–428, 2012.

[67] Richard Hull. Relative information capacity of simple relational database schemata. *SIAM J. Comput.*, 15(3):856–886, 1986.

[68] Frank K Hwang, Dana S Richards, and Pawel Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992.

[69] Shachar Itzhaky, Tomer Kotek, Noam Rinetzky, Mooly Sagiv, Orr Tamir, Helmut Veith, and Florian Zuleger. On the automated verification of web applications with embedded SQL. In *Proceedings of ICDT*, pages 16:1–16:18, 2017.

[70] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In *Proceedings of CAV*, pages 422–430, 2016.

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

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

[73] Vasileios Koutavas and Mitchell Wand. Bisimulations for untyped imperative objects. In *Proceedings of ESOP*, pages 146–161, 2006.

[74] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of POPL*, pages 141–152, 2006.

[75] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of PLDI*, pages 327–337, 2009.

[76] Shuvendu K. Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *Proceedings of CADE*, pages 214–229, 2009.

[77] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[78] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of PLDI*, pages 436–449, 2018.

[79] Fei Li and Hosagrahar Visvesvaraya Jagadish. Nalir: an interactive natural language interface for querying relational databases. In *Proceedings of SIGMOD*, pages 709–712, 2014.

[80] Yuliang Li, Alin Deutsch, and Victor Vianu. VERIFAS: A practical verifier for artifact systems. *PVLDB*, 11(3):283–296, 2017.

[81] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *Proceedings of ECAI*, pages 525–530, 2014.

[82] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proceedings of PLDI*, pages 355–368, 2016.

[83] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of ESEC/FSE*, pages 166–178, 2015.

[84] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *Proceedings of VLDB*, pages 49–58, 2001.

[85] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. *PVLDB*, 12(12):1914–1917, 2019.

[86] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *Proceedings of VLDB*, pages 77–88, 2000.

[87] Renée J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. The use of information capacity in schema integration and translation. In *Proceedings of VLDB*, pages 120–133, 1993.

[88] MongoDB. MongoDB website. https://www.mongodb.com, 2019.

[89] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.

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

[91] MySQL Tutorial. Delete from Join. http://www.mysqltutorial.org/mysql-delete-join, 2018.

[92] MySQL Tutorial. Update from Join. http://www.mysqltutorial.org/mysql-update-join, 2018.

[93] Joseph P. Near and Daniel Jackson. Rubicon: bounded verification of web applications. In *Proceedings of FSE*, page 60, 2012.

[94] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of PLDI*, pages 83–94, 2000.

[95] Oracle. Oracle schema optimization guide. https://docs.oracle.com/cd/B14099_19/web.1012/b15901/tuning007.htm, 2005.

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

[97] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of TACAS*, pages 151–166, 1998.

[98] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of PLDI*, pages 522–538, 2016.

[99] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of OOPSLA*, pages 107–126, 2015.

[100] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *Proceedings of SIGMOD*, pages 273–284, 2000.

[101] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *PVLDB*, pages 598–609, 2002.

[102] Li Qian, Michael J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *Proceedings of SIGMOD*, pages 73–84, 2012.

[103] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of ESEC/FSE*, pages 125–135, 2013.

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

[105] Martin Rinard. Credible compilation. In *MIT TechReport*, pages MIT–LCS–TR–776, 1999.

[106] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Proceedings of NIPS*, pages 3788–3800, 2017.

[107] Arnon Rosenthal and David S. Reiner. Tools and transformations - rigorous and otherwise - for practical database design. *ACM Trans. Database Syst.*, 19(2):167–211, 1994.

[108] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[109] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *Proceedings of LICS*, pages 293–302, 2007.

[110] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.

[111] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of ASPLOS*, pages 305–316, 2013.

[112] Xujie Si, Mukund Raghothanman Lee, Kihong Heo, and Mayur Naik. Synthesizing datalog programs using numerical relaxation. In *Proceedings of IJCAI*, 2019.

[113] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of ESEC/SIGSOFT FSE*, pages 515–527, 2018.

[114] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paris Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of FSE*, pages 515–527, 2018.

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

[116] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[117] Armando Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008.

[118] Armando Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of APLAS*, pages 4–13, 2009.

[119] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of ASPLOS*, pages 404–415, 2006.

[120] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of PLDI*, pages 57–69, 2016.

[121] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Proceedings of CAV*, pages 737–742, 2011.

[122] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *Proceedings of POPL*, pages 161–172, 2004.

[123] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *Proceedings of POPL*, pages 63–74, 2005.

[124] Boris A Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70:569–572, 1950.

[125] Victor Vianu. Automatic verification of database-driven systems: a new frontier. In *Proceedings of ICDT*, pages 1–13, 2009.

[126] Joost Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electronic Notes in Theoretical Computer Science*, 200(3):3–23, 2008.

[127] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of PLDI*, pages 452–466, 2017.

[128] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.

[129] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. `http://arxiv.org/abs/1710.07660`, 2017.

[130] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL):56:1–56:29, 2018.

[131] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of PLDI*, pages 286–300, 2019.

[132] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. `http://arxiv.org/abs/1904.05498`, 2019.

[133] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. Data migration using datalog program synthesis. *PVLDB*, 13(7):1006–1019, 2020.

[134] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. Data migration using datalog program synthesis. `http://arxiv.org/abs/2003.01331`, 2020.

[135] Yuepeng Wang, Xinyu Wang, and Isil Dillig. Relational program synthesis. *PACMPL*, 2(OOPSLA):155:1–155:27, 2018.

[136] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar T. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering Methodology*, 16(4):14, 2007.

[137] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of ISSTA*, pages 249–260, 2008.

[138] Wikimedia. Schema changes. `https://wikitech.wikimedia.org/wiki/Schema_changes`, 2017.

[139] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of PLDI*, pages 508–521, 2016.

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

[141] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[142] Ling-Ling Yan, Renée J. Miller, Laura M. Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *Proceedings of SIGMOD*, pages 485–496, 2001.

[143] Fan Yang, Zhilin Yang, and William W. Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Proceedings of NIPS*, pages 2319–2328, 2017.

[144] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[145] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Proceedings of FM*, pages 35–51, 2008.

[146] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003.