# Synthesizing Database Programs for Schema Refactoring

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

James Dong
University of Texas at Austin
Austin, Texas, USA
jdong@cs.utexas.edu

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

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

## Abstract

Many programs that interact with a database need to undergo *schema refactoring* several times during their life cycle. Since this process typically requires making significant changes to the program's implementation, schema refactoring is often non-trivial and error-prone. Motivated by this problem, we propose a new technique for *automatically synthesizing* a new version of a database program given its original version and the source and target schemas. Our method does not require manual user guidance and ensures that the synthesized program is equivalent to the original one. Furthermore, our method is quite efficient and can synthesize new versions of database programs (containing up to 263 functions) that are extracted from real-world web applications with an average synthesis time of 69.4 seconds.

***CCS Concepts*** • **Software and its engineering** → **Programming by example**; **Automatic programming**; • **Information systems** → **Database utilities and tools**.

***Keywords*** Program Synthesis, Program Sketching, Relational Databases

**ACM Reference Format:**
Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314588

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314588

## 1 Introduction

*Database-driven applications* have been, and continue to be, enormously popular for web development. For example, most contemporary websites are built using database-driven applications in order to generate webpage content dynamically. As a result, database applications form the backbone of many industries, ranging from banking and e-commerce to telecommunications.

A common theme in the evolution of database applications is that they typically undergo *schema refactoring* several times during their life cycle [4, 21]. Schema refactoring involves a change to the database schema, with the goal of improving the design and/or performance of the application *without* changing its semantics. Despite the frequent need to perform schema refactoring, this task is known to be non-trivial and error-prone [3, 59]. In particular, changes to the database schema often require re-implementing parts of the database program to make the program logic consistent with the underlying schema. This task is especially non-trivial in the presence of *structural* schema changes, such as those that involve splitting and merging relations or moving attributes between different tables.

While prior work has addressed the problem of *verifying* equivalence between two database programs before and after schema refactoring [54], generating a new version of the program after a schema change still remains an arduous and manual task. Motivated by this problem, this paper 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 database transactions 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 schema evolution process for these kinds of database programs while ensuring that no desirable behaviors are lost and no unwanted behaviors are introduced in the process.

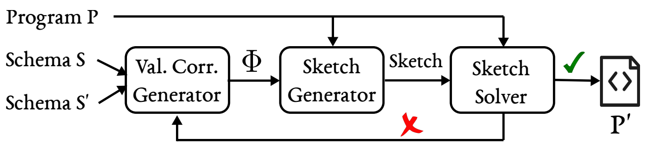

**Figure 1.** Synthesis methodology.

Our methodology for automatically migrating database programs to a new schema is illustrated schematically in Figure 1. Rather than synthesizing the new version of the program in one go, our algorithm decomposes the 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 $\Phi$ specifies how attributes in $\mathcal{S}'$ can be obtained using the attributes in $\mathcal{S}$ [37]. 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 $\Phi$, our synthesis algorithm generates a *program sketch* $\Omega$ that represents the space of all programs that *may* be equivalent to the original program $\mathcal{P}$ according to $\Phi$. 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 $\Phi$, the sketch $\Omega$ 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 $\Omega$ by finding an instantiation $\mathcal{P}'$ of $\Omega$ 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 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 evolution process.

In all, this paper makes the following key contributions:

- We propose a new synthesis technique for automatically migrating database programs to a new schema.
- We describe a MaxSAT-based approach for lazily enumerating possible value correspondences between two schemas.
- We describe a technique for generating program sketches from a given value correspondence.
- We propose a new sketch solver based on symbolic search and conflict-driven learning from minimum failing inputs.
- We evaluate the proposed technique on 20 schema refactoring scenarios and demonstrate that our method can automate the desired migration task in all cases.

```
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 2.** An example database program.

## 2 Overview

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

```
update addInstructor(int id, String name, Binary pic)
  INSERT INTO ??₁ { Picture ⋈ Instructor, Picture ⋈ TA ⋈ Instructor,
    Picture ⋈ TA ⋈ Class ⋈ Instructor } VALUES (id, name, pic);

update deleteInstructor(int id)
  DELETE ??₂ { [Picture], . . ., [Picture, Instructor, TA, Class] }
    FROM ??₃ { Picture ⋈ Instructor, Picture ⋈ TA ⋈ Instructor,
    Picture ⋈ TA ⋈ Class ⋈ Instructor } WHERE InstId = id;

query getInstructorInfo(int id)
  SELECT IName, Pic FROM ??₄ {
    Picture ⋈ Instructor, Picture ⋈ TA ⋈ Instructor,
    Picture ⋈ TA ⋈ Class ⋈ Instructor } WHERE InstId = id;

update addTA(int id, String name, Binary pic)
  INSERT INTO ??₅ { Picture ⋈ TA, Picture ⋈ Instructor ⋈ TA,
    Picture ⋈ Instructor ⋈ Class ⋈ TA } VALUES (id, name, pic);

update deleteTA(int id)
  DELETE ??₆ { [Picture], . . ., [Picture, Instructor, TA, Class] }
    FROM ??₇ { Picture ⋈ TA, Picture ⋈ Instructor ⋈ TA,
    Picture ⋈ Instructor ⋈ Class ⋈ TA } WHERE TaId = id;

query getTAInfo(int id)
  SELECT TName, Pic FROM ??₈ {
    Picture ⋈ TA, Picture ⋈ Instructor ⋈ TA,
    Picture ⋈ Instructor ⋈ Class ⋈ TA } WHERE TaId = id;
```

**Figure 3.** Generated sketch over the new database schema.

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 2 needs to be re-implemented to conform to the new schema. We now explain how Migrator automatically synthesizes the new version of the program.

***Value correspondence generation.*** As mentioned in Section 1, Migrator lazily enumerates possible value correspondences (VCs) between the source and target schemas. For this example, the first VC $\Phi$ generated by Migrator contains the following mappings:

$$\begin{array}{rcl} \textit{Instructor.IPic} & \longrightarrow & \textit{Picture.Pic} \\ \textit{TA.TPic} & \longrightarrow & \textit{Picture.Pic} \end{array}$$

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

***Sketch generation.*** Next, Migrator uses the candidate VC $\Phi$ to generate a program sketch that encodes the space of all programs that are consistent with $\Phi$. The corresponding sketch for this example is shown in Figure 3. Here, each hole, denoted $\textbf{??}\{c_1, \ldots, c_n\}$, corresponds to an unknown constant drawn from the set $\{c_1, \ldots, c_n\}$. As will be discussed later in Section 3, we use the statement:

**INSERT INTO** $T_1 \bowtie T_2$ **VALUES** $\cdots$

as short-hand for:

**INSERT INTO** $T_1$ **VALUES** $\cdots$
**INSERT INTO** $T_2$ **VALUES** $\cdots$

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 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 $\Omega$ and the original program $\mathcal{P}$, the goal of sketch completion is to find an instantiation $\mathcal{P}'$ of $\Omega$ 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., [47, 49]) because the symbolic encoding of the program is quite complex due to the non-trivial semantics of SQL. In this paper, we deal with this difficulty by (a) encoding the space of *all* possible programs represented by the sketch using a SAT formula $\Psi$, and (b) using minimum failing inputs to dramatically prune the search space represented by $\Psi$.

Going back to our sketch $\Omega$ from Figure 3, Migrator generates the following SAT formula that encodes all possible instantiations of $\Omega$:

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

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

Given this formula $\Psi$, Migrator queries the SAT solver for a model. For the purpose of this example, suppose the SAT solver returns the following model for $\Psi$:

$$b_1^3 \land b_2^2 \land b_3^3 \land b_4^3 \land b_5^1 \land b_6^4 \land b_7^3 \land b_8^3 \quad (1)$$

```
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 4.** The synthesized database program.

which corresponds to the following assignment of the holes:

$$\begin{array}{ll} & ??_1 = ??_3 = ??_4 = \mathit{Picture} \bowtie \mathit{TA} \bowtie \mathit{Class} \bowtie \mathit{Instructor} \\ \wedge & ??_2 = [\mathit{Instructor}] \ \wedge \ ??_5 = \mathit{Picture} \bowtie \mathit{TA} \ \wedge \ ??_6 = [\mathit{TA}] \\ \wedge & ??_7 = ??_8 = \mathit{Picture} \bowtie \mathit{Instructor} \bowtie \mathit{Class} \bowtie \mathit{TA} \end{array} \tag{2}$$

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 1 with $\Psi$ and asking the SAT solver for another model. While this strategy would give us a different instantiation of sketch $\Omega$, it would preclude *only one* of the 164, 025 possible instantiations of $\Omega$. 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 $\omega$:

$$\mathit{addTA}(\mathit{ta1}, \mathit{name1}, \mathit{pic1}); \mathit{getTAInfo}(\mathit{ta1}) \tag{3}$$

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 $\omega$ to prune incorrect programs other than just $\mathcal{P}'$. Specifically, let $\mathcal{F}$ denote the functions that appear in the invocation sequence $\omega$, 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 $\omega$ 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 2:

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

This model corresponds to the program $\mathcal{P}'$ shown in Figure 4, which is indeed equivalent to the original program from Figure 2. Thus, Migrator returns $\mathcal{P}'$ as the synthesis result.

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

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

**Figure 5.** Syntax of database programs. + denotes the previous construct appears once or multiple times.

## 3 Preliminaries

In this section, we introduce the syntax and semantics of database programs and review what equivalence means in this context.

### 3.1 Syntax and Semantics of Database Programs

For the purpose of this paper, a database program consists of a set of functions, where each function is either a *query* or *update* to the database. As shown in Figure 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 ($\Pi$), selection ($\sigma$), 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 $\phi$. A join expression $J_1\ {}_{a_1}\bowtie_{a_2} J_2$ 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 paper, we use the terminology *join* or *join chain* to refer to both database tables as well as (possibly nested) join expressions of the form $J_1\,{}_{a_1}\!\bowtie_{a_2} J_2$. 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 paper, we use the syntax

$$\texttt{ins}(T_1\,{}_{fk_1}\!\bowtie_{pk_2} T_2, \{a_1 : v_1, \ldots, a_n : v_n, a'_1 : v'_1, \ldots, a'_m : v'_m\})$$

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 $\phi$ in join chain $J$. As an example, consider the delete statement $\texttt{del}([T_1],\ T_1\,{}_{a_1}\!\bowtie_{a_2} T_2,\ \phi)$. Here, we first compute $T_1\,{}_{a_1}\!\bowtie_{a_2} T_2$ 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 $\phi$. Finally, we delete from $T_1$ all tuples that occur as (a prefix of) a tuple in $T'$. In contrast, if the statement is $\texttt{del}([T_1, T_2],\ T_1\,{}_{a_1}\!\bowtie_{a_2} T_2,\ \phi)$, the deletion is performed on both $T_1$ and $T_2$. We refer the reader to [38] for a more detailed discussion of the semantics of delete statements. [1]

An update statement $\texttt{upd}(J, \phi, a, v)$ modifies the value of attribute $a$ to $v$ for all tuples satisfying predicate $\phi$ in join chain $J$ [39]. For instance, consider the update statement $\texttt{upd}(T_1\,{}_{a_1}\!\bowtie_{a_2} T_2,\ \phi,\ T_1.a_3,\ v)$. Like delete statements, we first compute $T_1\,{}_{a_1}\!\bowtie_{a_2} T_2$ 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 $\phi$ 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 3.1.** 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

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

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

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

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

[1] 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 paper.

### 3.2 Equivalence of Two Database Programs

Since our goal is to synthesize a program $\mathcal{P}'$ that is equivalent to another database program $\mathcal{P}$ with a different schema, we review the definition of equivalence introduced in prior work [54].

Consider a database program $\mathcal{P}$ over schema $\mathcal{S}$ that has a set of update functions $U = (U_1, \ldots, U_n)$ and a set of query functions $Q = (Q_1, \ldots, Q_m)$. First, an *invocation sequence* for $\mathcal{P}$ is of the form

$$\omega = (f_1, \sigma_1); \ldots; (f_{k-1}, \sigma_{k-1}); (f_k, \sigma_k)$$

where $f_k$ is the name of a query function in $Q$, $f_1, \ldots, f_{k-1}$ refer to names of updates functions in $U$, and $\sigma_i$ corresponds to the arguments for function $f_i$. Given a program $\mathcal{P}$, we use the notation $[\![\mathcal{P}]\!]_\omega$ to denote the result of executing $\mathcal{P}$ on $\omega$.

Now, consider two programs $\mathcal{P}, \mathcal{P}'$ over schemas $\mathcal{S}, \mathcal{S}'$. Following [54], we say that $\mathcal{P}$ is equivalent to $\mathcal{P}'$, written $\mathcal{P} \simeq \mathcal{P}'$, if for *any* invocation sequence $\omega$, we have $[\![\mathcal{P}]\!]_\omega = [\![\mathcal{P}']\!]_\omega$ — i.e., executing $\omega$ on $\mathcal{P}$ yields the same query result as executing $\omega$ on $\mathcal{P}'$ starting with an empty database instance. Thus, if two database programs are equivalent, then they yield the same query result after performing the same sequence of update operations on the database.

## 4 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.

### 4.1 Overview

Our top-level synthesis algorithm is summarized as pseudo-code in Algorithm 1. 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 $\bot$ 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 $\Phi$ 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 $\mathcal{S}'$ [37]. 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 $\Phi$ 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

**Algorithm 1** 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 $\bot$ to indicate failure
2:     while true do
3:         $\Phi \leftarrow$ NEXTVALUECORR($\mathcal{S}, \mathcal{S}'$);
4:         if $\Phi = \bot$ then return $\bot$;
5:         $\Omega \leftarrow$ GENSKETCH($\Phi, \mathcal{P}$);
6:         $\mathcal{P}' \leftarrow$ COMPLETESKETCH($\Omega, \mathcal{P}$);
7:         if $\mathcal{P}' \neq \bot$ then return $\mathcal{P}'$;
```

indicates that attribute $T.a$ has been duplicated in the target schema. [2]

Now, given a candidate value correspondence $\Phi$, the GENSKETCH procedure at line 5 generates a sketch $\Omega$ that represents all programs that *may* be equivalent to $\mathcal{P}$ under the assumption that $\Phi$ is correct. Finally, the COMPLETESKETCH procedure (line 6) tries to find an instantiation $\mathcal{P}'$ of $\Omega$ 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 $\Phi'$ and re-attempts the synthesis task using $\Phi'$.

As formalized in more detail in the extended version of this paper [55], 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 $\mathcal{P}$ by the definition from Section 3.2. 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.

## 4.2 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.

[2] Our notion of value correspondence is a slightly simplified version of the definition given by Miller et al. [37]. For example, their definition also allows attributes in the target schema to be obtained by 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.

***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 the MaxSAT problem 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 $a_1, \dots, a_n$ (resp. $a'_1, \dots, a'_m$). 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 $\Phi$ 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:

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

- *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 3.2). Thus, we introduce the following hard constraint:

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

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

$$\begin{array}{rcl}
\textit{Prog} & := & \textit{Func}+ \\
\textit{Func} & := & \textbf{update}\ \textit{Name}(\textit{Param}+)\ U \\
 & \mid & \textbf{query}\ \textit{Name}(\textit{Param}+)\ Q \\
\textit{Update}\ U & := & \textit{InsStmt} \mid \textit{DelStmt} \mid \textit{UpdStmt} \mid U; U \mid U \textcircled{?} U \\
\textit{Query}\ Q & := & \Pi_{(??\{a+\})+}(Q) \mid \sigma_{\phi}(Q) \mid J \mid Q \textcircled{?} Q \\
\textit{Join}\ J & := & T \mid J\ {}_{a}\!\bowtie_{a} J \\
\textit{Pred}\ \phi & := & ??\{a+\}\ \textit{op}\ ??\{a+\} \mid ??\{a+\}\ \textit{op}\ v \\
 & \mid & \phi \land \phi \mid \phi \lor \phi \mid \neg\phi \\
\textit{InsStmt} & := & \texttt{ins}(J,\ \{(??\{a+\} : v)+\}) \\
\textit{DelStmt} & := & \texttt{del}(??\{L+\},\ J,\ \phi) \\
\textit{UpdStmt} & := & \texttt{upd}(J,\ \phi,\ ??\{a+\},\ v) \\
\textit{TabList}\ L & := & [T+]
\end{array}$$

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

**Figure 6.** Sketch Language. **??** 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 $E$.

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

## 4.3 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 $\Phi$. 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 6 and differs from the source language in Figure 5 in the following ways: First, programs in the sketch language can contain a construct of the form $??\{e_1, \ldots, e_n\}$, 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:

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

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 5.

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

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

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

**Figure 7.** Inference rules for checking join correspondence $(J, J')$ under value correspondence $\Phi$.

***Join correspondence.*** In order to generate a sketch from a program $\mathcal{P}$ and value correspondence $\Phi$, 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 $\Phi$ if $\Phi$ can map all attributes used in $J$ to attributes in $J'$.

Figure 7 presents inference rules for checking whether a join correspondence $(J, J')$ is valid under $\Phi$. 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 $\Phi$. 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 $\Phi$. Observe that, if $\Phi \vdash J \sim J_1$ *and* $\Phi \vdash J \sim J_2$, this 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.

***Sketch generation, phase I.*** The first phase of our sketch generation procedure is summarized in Figure 8 and assumes that every join chain $J$ in the source program maps to a unique join chain $J'$ in the target program. Specifically, the rules in Figure 8 derive judgments of the form $\Phi \vdash s \rightsquigarrow \Omega$, meaning that statement $s$ in the original program can be rewritten into sketch $\Omega$ under the assumption that (a) $\Phi$ is correct and (b) every join chain in the source program

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

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

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

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

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

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

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

**Figure 8.** Rewrite rules for generating sketch from value correspondence $\Phi$. 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. $\mathit{TabLists}(T_1 \bowtie \ldots \bowtie T_n) = \mathit{PowerSet}(\{T_1, \ldots, T_n\}) \setminus \emptyset$.

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 $\phi$ 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 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 $\mathtt{del}(\mathit{Tbls}, J, \phi)$, 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

$$\frac{\Phi \vdash s \rightsquigarrow \Omega}{\Phi \vdash s \twoheadrightarrow \Omega}\ (\text{Lift})$$

$$\frac{\begin{array}{c}\Phi \vdash Q \twoheadrightarrow \Omega \quad \Phi \vdash Q \rightsquigarrow \Omega' \\ \Omega = \Omega_1 \textcircled{?} \ldots \textcircled{?} \Omega_n \quad \Omega' \neq \Omega_i \quad i = 1, \ldots, n\end{array}}{\Phi \vdash Q \twoheadrightarrow \Omega \textcircled{?} \Omega'}\ (\text{Query})$$

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

$$\frac{\Phi \vdash U_1 \twoheadrightarrow \Omega_1 \quad \Phi \vdash U_2 \twoheadrightarrow \Omega_2}{\Phi \vdash U_1; U_2 \twoheadrightarrow \Omega_1; \Omega_2}\ (\text{Seq})$$

**Figure 9.** Inference rules for composing multiple sketches. The composition operator $\bullet$ is defined in Figure 10.

$$\begin{array}{rcl} U_1 \bullet U_2 & = & U_1; U_2 \quad (U_1 = \mathtt{ins} \text{ or } \mathtt{del} \text{ or } \mathsf{upd}) \\ (U_1; U_2) \bullet U_3 & = & U_1; U_2; U_3 \\ (U_1 \textcircled{?} U_2) \bullet U_3 & = & (U_1 \bullet U_3) \textcircled{?} (U_2 \bullet U_3) \end{array}$$

**Figure 10.** Definition of the composition operator.

the target schema — i.e., 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 first phase to synthesize a more general sketch that accounts for this ambiguity.

Figure 9 describes the second phase of sketch generation using judgments of the form $\Phi \vdash s \twoheadrightarrow \Omega$. At a high level, the rules in Figure 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 $\Omega_1 \textcircled{?} \ldots \textcircled{?} \Omega_n$ — 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 $\Omega_1, \Omega_2$ 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 $\Omega_1 \textcircled{?} \Omega_2 \textcircled{?} (\Omega_1; \Omega_2)$ rather than the simpler sketch $\Omega_1 \textcircled{?} \Omega_2$ for query statements. The Update rule generalizes this discussion to arbitrarily many sketches by using a binary operator $\bullet$ (defined in Figure 10) that distributes sequential composition (;) over the choice ($\textcircled{?}$) construct. Finally, the Seq rule allows generating a sketch for $U_1; U_2$ using the sketch $\Omega_i$ for each $U_i$.

Given a statement $s$ in the source program, its corresponding sketch $\Omega$ is obtained by applying the rewrite rules from Figure 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

**Algorithm 2** Sketch Completion

```
1: procedure CompleteSketch($\Omega, \mathcal{P}$)
   Input: Sketch $\Omega$, Source program $\mathcal{P}$
   Output: Target program $\mathcal{P}'$ or $\bot$ to indicate failure
2:     $\Psi \leftarrow$ Encode($\Omega$);
3:     while SAT($\Psi$) do
4:         $\mathcal{M} \leftarrow$ GetModel($\Psi$);
5:         $\mathcal{P}' \leftarrow$ Instantiate($\Omega, \mathcal{M}$);
6:         $done \leftarrow$ Verify($\mathcal{P}, \mathcal{P}'$);
7:         if $done$ then return $\mathcal{P}'$;
8:         $\mathcal{E} \leftarrow$ MinCex($\mathcal{P}, \mathcal{P}'$);
9:         $\Psi \leftarrow \Psi \land$ Block($\mathcal{M}, \mathcal{E}$);
10:    return $\bot$;
```

let us say that a sketch $\Omega$ is more general than $\Omega'$, written $\Omega \succeq \Omega'$, if $\Omega$ represents more programs than $\Omega'$. Then, the resulting sketch for $s$ is the most general sketch $\Omega_i$ such that $\forall j \in [1, n].\Omega_i \succeq \Omega_j$.

## 4.4 Sketch Completion

In this section, we explain our algorithm for solving the database program sketches from Section 4.3. 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 failing 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 2 and takes as input a program sketch $\Omega$ together with the source program $\mathcal{P}$. The output of CompleteSketch is either a completion $\mathcal{P}'$ of $\Omega$ such that $\mathcal{P} \simeq \mathcal{P}'$ or $\bot$ to indicate no such program exists.

At a high level, the CompleteSketch procedure first generates a boolean formula $\Psi$ that represents *all* possible completions of the sketch $\Omega$ (line 2). While any model of $\Psi$ corresponds to a concrete program $\mathcal{P}'$ that is an instantiation of $\Omega$, 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 $\Omega$, checks equivalence, and adds useful blocking clauses to the SAT encoding $\Psi$ 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 $\Omega$. Specifically, for each hole $\textbf{??}_i\{e_1, \ldots, e_n\}$ in the sketch, we introduce $n$ boolean variables $b_i^1, \ldots, b_i^n$ such that $b_i^j = true$ if and only if hole $\textbf{??}_i$ is instantiated with expression $e_j$. [4] Since any valid completion of sketch $\Omega$ must assign every hole $\textbf{??}_i$ 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 $\textbf{??}_i$ 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 $\Psi$ corresponds to one particular instantiation of $\Omega$; thus, the procedure Instantiate produces program $\mathcal{P}'$ by assigning hole $\textbf{??}_i$ to expression $e_j$ if and only if $\mathcal{M}$ assigns $b_i^j$ to true.

***Verification and blocking clauses.*** As is apparent from the discussion above, our symbolic encoding $\Psi$ of the sketch intentionally does not enforce equivalence between source and target programs. Thus, whenever we obtain a 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 2. 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 $\omega$ (recall Section 3.2) satisfying the following criteria:

1. We have $[\![\mathcal{P}]\!]_\omega \neq [\![\mathcal{P}']\!]_\omega$. That is, $\omega$ is a witness to the disequivalence of $\mathcal{P}$ and $\mathcal{P}'$
2. There does not exist another invocation sequence $\omega'$ 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 $\omega$, and let $A_\mathcal{H}$ denote the assignments to holes $\mathcal{H}$. Then, any program that instantiates $\Omega$ 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 $\textbf{??}_i$. Then, the Block procedure (line 9 of Algorithm 2) generates the following blocking clause:

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

[4]Since the choice construct $s_1 \circled{?} s_2$ 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.

Intuitively, this blocking clause $\varphi$ rules out all completions of $\Omega$ 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 5, minimum failing inputs are inexpensive to obtain using testing.

## 5 Implementation

We have implemented the proposed synthesis technique in a new tool called Migrator, which is implemented in Java. Migrator uses the Sat4J solver [31] for answering all SAT and MaxSAT queries and the Mediator tool [54] 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 4.3 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 4.3 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 $\Phi$ 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 $a' \in Attrs(J_i)$. We reduce the problem of finding all such possible join chains to the problem of finding all possible Steiner trees [29] 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 4.4 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 $\omega$, we execute both $\mathcal{P}$ and $\mathcal{P}'$ on $\omega$ and check if the outputs are different. If so, we return $\omega$ as a minimum failing input, and otherwise, we test equivalence using the next invocation sequence.

***Verification.*** Our sketch completion algorithm from Section 4.4 invokes a Verify procedure to check if two programs are equivalent. However, since full-fledged verification using the Mediator tool [54] 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.

## 6 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 prior work [54] for verifying equivalence between database programs. [5] Specifically, half of these benchmarks are adapted from textbooks and online tutorials, and 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.

### 6.1 Main Results

Our main experimental results are summarized in Table 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 "Description" column in Table 1 explains how the database schema differs between

[5] While prior work considers 21 benchmarks in the evaluation, one of these benchmarks cannot be verified by Mediator. Since we use Mediator as our verifier, we exclude this one benchmark from our evaluation.

**Table 1.** Main experimental results.

| | **Benchmark** | **Description** | **Funcs** | **Source Schema** | | **Target Schema** | | **Value Corr** | **Iters** | **Synth Time(s)** | **Total Time(s)** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | **Tables** | **Attrs** | **Tables** | **Attrs** | | | | |
| **textbook bench** | Oracle-1 | Merge tables | 4 | 2 | 8 | 1 | 6 | 1 | 1 | 0.3 | 2.7 |
| | Oracle-2 | Split tables | 19 | 3 | 17 | 7 | 25 | 1 | 5 | 0.5 | 11.3 |
| | Ambler-1 | Split tables | 10 | 1 | 6 | 2 | 7 | 1 | 2 | 0.3 | 2.9 |
| | Ambler-2 | Merge tables | 10 | 2 | 7 | 1 | 6 | 1 | 1 | 0.3 | 0.6 |
| | Ambler-3 | Move attrs | 7 | 2 | 5 | 2 | 5 | 2 | 5 | 0.4 | 30.6 |
| | Ambler-4 | Rename attrs | 5 | 1 | 2 | 1 | 2 | 1 | 1 | 0.3 | 0.5 |
| | Ambler-5 | Add associative tables | 8 | 2 | 5 | 3 | 6 | 5 | 7 | 0.3 | 3.1 |
| | Ambler-6 | Replace keys | 10 | 2 | 9 | 2 | 8 | 1 | 1 | 0.3 | 0.7 |
| | Ambler-7 | Add attrs | 8 | 2 | 7 | 2 | 8 | 1 | 1 | 0.3 | 0.6 |
| | Ambler-8 | Denormalization | 14 | 3 | 10 | 3 | 13 | 1 | 7 | 0.5 | 3.1 |
| **real-world bench** | cdx | Rename attrs, split tables | 138 | 16 | 125 | 17 | 131 | 1 | 7 | 11.9 | 38.9 |
| | coachup | Split tables | 45 | 4 | 51 | 5 | 55 | 1 | 10 | 1.8 | 6.7 |
| | 2030Club | Split tables | 125 | 15 | 155 | 16 | 159 | 1 | 2 | 5.2 | 24.8 |
| | rails-ecomm | Split tables, add new attrs | 65 | 8 | 69 | 9 | 75 | 1 | 6 | 2.5 | 10.3 |
| | royk | Add and move attrs | 151 | 19 | 152 | 19 | 155 | 1 | 17 | 46.1 | 60.1 |
| | MathHotSpot | Rename tables, move attrs | 54 | 7 | 38 | 8 | 42 | 6 | 11 | 1.2 | 5.8 |
| | gallery | Split tables | 58 | 7 | 52 | 8 | 57 | 1 | 11 | 2.5 | 9.4 |
| | DeeJBase | Rename attrs, split tables | 70 | 10 | 92 | 11 | 97 | 1 | 8 | 3.5 | 9.3 |
| | visible-closet | Split tables | 263 | 26 | 248 | 27 | 252 | 1 | 108 | 1304.7 | 1370.8 |
| | probable-engine | Merge tables | 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** |

the source and target versions, and “Funcs” 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 schema refactoring process for database programs.

## 6.2 Comparison to 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 [47], 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 [47], 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.

The results of this experiment are summarized in Table 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. [6] We believe this experiment demonstrates 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.

[6] 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.

**Table 2.** Comparison with Sketch.

| | Benchmark | Sketch | |
|---|---|---|---|
| | | **Synth Time(s)** | **Speedup** |
| 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** |

**Table 3.** Comparison with symbolic enumerative search.

| | Benchmark | Symbolic Enum | | |
|---|---|---|---|---|
| | | **Iters** | **Synth Time(s)** | **Speedup** |
| 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** |

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 $\Psi$, this baseline updates $\Psi$ by conjoining $\neg\mathcal{M}$ 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. 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, these results demonstrate that our MFI-based sketch completion is very important for practical synthesis.

## 7 Related Work

In this section, we survey related papers on program synthesis, schema refactoring, and analysis of database applications.

***Schema evolution.*** There is a body of literature on automating the schema refactoring process, including the rewrite of SQL queries and updates [8, 14, 15, 19, 43, 50]. Among these works, the most related one is the Prism project and its successor Prism++ [14, 15]. 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 [19, 43]. 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, our method 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 invertible, and, second, the search space of operator sequences is also potentially very large.

***Analysis of database applications.*** Over the past decade, there has been significant interest in analyzing, verifying, and testing database applications [5, 9, 17, 18, 20, 24, 25, 30, 34, 40, 44, 54, 57, 58]. For instance, the WAVE [17, 18] and VERIFAS [34] projects aim to verify temporal properties of database applications, and recent work by Itzhaky et al. [30] proposes a technique to verify pre- and post-conditions of methods with embedded SQL statements. There has also been some work on model checking database applications [24, 40] as well as automatically generating test cases [5, 9, 20, 58].

Since our goal is to synthesize a new version of the program that is *equivalent* to a previous version, this paper is particularly related to verification techniques for checking equivalence [11–13, 52, 54]. Most of these papers focus on equivalence between individual SQL queries [11–13, 52]. The only work that addresses the problem of verifying equivalence between entire database programs is Mediator, which automatically infers a bisimulation invariant between the two programs [54]. As discussed in Section 3.2, we adopt the definition of equivalence proposed in that paper. However, our synthesis technique does not simply add a CEGIS loop on top of Mediator's SMT theory because such an approach would require solving complex quantified formulas over the theory of lists. Instead, our approach performs enumerative search but uses minimum failing inputs to significantly prune the search space.

***Program synthesis.*** This paper is related to a long line of recent work on program synthesis [1, 2, 6, 7, 22, 26–28, 32, 35, 36, 41, 42, 45–49, 53, 56]. 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 [6, 22, 26, 42, 53], logical constraints [47–49], refinement types [41], or a reference implementation [27, 35, 45]. Our technique belongs in the latter category in that it uses the original implementation as the specification.

Among existing techniques, our synthesis algorithm is particularly related to Neo [22], which uses *conflict-driven learning* to infer useful lemmas from failed synthesis attempts. Our sketch solving algorithm from Section 4.4 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.

***Synthesis for database programs.*** In recent years, there have been several papers that apply program synthesis to SQL queries or database programs [10, 16, 23, 33, 51, 60]. For instance, Sqlizer [60] synthesizes SQL queries from natural language, whereas Scythe [51] and Morpheus [23] generate queries from examples. The QBS system uses program synthesis to repair performance bugs in database applications [10]. Finally, Fiat [16] performs deductive synthesis to generate SQL-like operations from declarative specifications. However, none of these techniques consider the problem of automatically migrating database programs in the presence of schema refactoring.

## 8 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 [54] does not support database programs with those constructs.

Third, the notion of equivalence considered in Section 3.2 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 paper.

## 9 Conclusion

In this paper, we have studied the problem of automatically synthesizing database programs in the presence of schema refactoring. Our technique decomposes the synthesis procedure into three tasks, namely (i) lazy value correspondence enumeration, (ii) sketch generation from a candidate value correspondence, and (iii) sketch completion using conflict-driven learning with minimum failing inputs. We have implemented the proposed technique in a tool called Migrator and evaluated it on 20 schema refactoring benchmarks, including real-world scenarios taken from Github. Our evaluation shows that Migrator can automatically synthesize the new versions of all 20 benchmarks and indicates that the proposed technique would be useful to database application developers during the schema evolution process.

## Acknowledgments

We would like to thank Shuvendu Lahiri for initial discussions on the topic of this paper. We also thank Xinyu Wang, Kostas Ferles, Shankara Pailoor, and other UToPiA group members for their useful feedback on earlier drafts. We would also like to thank our shepherd, Emina Torlak, and the anonymous reviewers for their constructive suggestions for improvement. Finally, we are very grateful to the National Science Foundation for supporting this work under Grants #1762299, #1712067, and #1811865.

## References

[1] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Proc. of CP*. 689–706.

[2] 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. 2013. Syntax-guided synthesis. In *Proc. of FMCAD*. 1–8.

[3] Scott W Ambler. 2007. Test-Driven Development of Relational Databases. *IEEE Software* 24, 3 (2007).

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

[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering* 36, 4 (2010), 474–494.

[6] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proc. of ICLR*.

[7] James Bornholt and Emina Torlak. 2017. Synthesizing memory models from framework sketches and Litmus tests. In *Proc. of PLDI*. 467–481.

[8] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. 2016. Synchronization of Queries and Views Upon Schema Evolutions: A Survey. *TODS* 41, 2 (2016), 9:1–9:41.

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

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

[11] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *PVLDB* 11, 11 (2018), 1482–1495.

[12] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Proc. of CIDR*.

[13] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proc. of PLDI*. 510–524.

[14] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2010. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB* 4, 2 (2010), 117–128.

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

[16] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proc. of POPL*. 689–700.

[17] Alin Deutsch, Richard Hull, and Victor Vianu. 2014. Automatic Verification of Database-Centric Systems. *SIGMOD Record* 43, 3 (2014), 5–17.

[18] Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu, and Dayou Zhou. 2005. A Verifier for Interactive, Data-Driven Web Applications. In *Proc. of SIGMOD*. 539–550.

[19] Alin Deutsch and Val Tannen. 2003. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proc. of VLDB*. 201–212.

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

[21] Stéphane Faroult and Pascal L'Hermite. 2008. *Refactoring SQL Applications*. O'Reilly Media.

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

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

[24] Milos Gligoric and Rupak Majumdar. 2013. Model Checking Database Applications. In *Proc. of TACAS*. 549–564.

[25] Shelly Grossman, Sara Cohen, Shachar Itzhaky, Noam Rinetzky, and Mooly Sagiv. 2017. Verifying Equivalence of Spark Programs. In *Proc. of CAV*. 282–300.

[26] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. of POPL*. 317–330.

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

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

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

[30] Shachar Itzhaky, Tomer Kotek, Noam Rinetzky, Mooly Sagiv, Orr Tamir, Helmut Veith, and Florian Zuleger. 2017. On the Automated Verification of Web Applications with Embedded SQL. In *Proc. of ICDT*. 16:1–16:18.

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

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

[33] Fei Li and Hosagrahar Visvesvaraya Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *Proc. of SIGMOD*. 709–712.

[34] Yuliang Li, Alin Deutsch, and Victor Vianu. 2017. VERIFAS: A Practical Verifier for Artifact Systems. *PVLDB* 11, 3 (2017), 283–296.

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

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

[37] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. 2000. Schema Mapping as Query Discovery. In *Proc. of VLDB*. 77–88.

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

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

[40] Joseph P. Near and Daniel Jackson. 2012. Rubicon: bounded verification of web applications. In *Proc. of FSE*. 60.

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

[42] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proc. of OOPSLA*. 107–126.

[43] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. 2000. A Chase Too Far?. In *Proc. of SIGMOD*. 273–284.

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

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

[46] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paris Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *Proc. of FSE*. 515–527.

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

[48] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proc. of APLAS*. 4–13.

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

[50] Joost Visser. 2008. Coupled Transformation of Schemas, Documents, Queries, and Constraints. *Electronic Notes in Theoretical Computer Science* 200, 3 (2008), 3–23.

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

[52] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018. Speeding up Symbolic Reasoning for Relational Queries. *PACMPL* 2, OOPSLA (2018), 157:1–157:25.

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

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

[55] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. http://arxiv.org/abs/1904.05498. arXiv:1904.05498

[56] Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *PACMPL* 2, OOPSLA (2018), 155:1–155:27.

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

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

[59] Wikimedia. 2018. Schema changes. https://wikitech.wikimedia.org/wiki/Schema_changes.

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