Synthesizing Bidirectional Programs on Relations

Nguyen Trong Bach

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences

The Graduate University for Advanced Studies, SOKENDAI

March 2024

Acknowledgments

This dissertation is a product of an extended collaboration with Prof. Zhenjiang Hu and Prof. Kanae Tsushima. I gratefully acknowledge their contributions to this dissertation.

I wish to express my deepest gratitude to my main advisor, Prof. Zhenjiang Hu, for his insightful instructions, convincing suggestions, and practical advice. He always motivated me to be more professional and keep pushing forward, even when faced with many difficulties.

I would like to convey my profound gratitude to Prof. Kanae Tsushima for dedicating her time to attend the small personal meetings and providing valuable feedback on my research.

I would also like to extend my thanks to my supervisor, Prof. Hasuo Ichiro, for his facilitation and assistance during my research in Japan.

To the members of the Programming Research Laboratory at National Institute of Informatics, I appreciate the creation of a warm, friendly, and inviting research atmosphere. I would like to thank the professors, Prof. Zhenjiang Hu, Prof. Masato Takeichi, Prof. Hiroyuki Kato, Prof. Kanae Tsushima, and Prof. Taro Sekiyama, for their interesting presentations and thoughtful comments during the lab meetings. I thank Tran Van Dang very much for his help to me both in the laboratory and in the real life. Thanks to the other members, Josh Ko, Zirun Zhu, Yongzhe Zhang, Chunmiao Li, Zhixuan Yang, Liye Guo and Jumpei Tanaka, for their tireless efforts over the years of research and sharing stories together.

To SOKENDAI and National Institute of Informatics, many thanks for the educational and financial support.

To readers, thank you for taking the time to read this dissertation.

To my family far away, thank you for not letting me worry even though many not-so-good things happened.

Finally, I recognize and appreciate my perseverance in dedicating extensive time to study, overcoming numerous challenges that I may not be at ease sharing, and likely will not recall them all in the future.

Somewhere in Tokyo, 2023
Nguyen Trong Bach

Abstract

Bidirectional transformations between different representations of related information appear frequently in many different areas like databases, software engineering, and programming languages. A bidirectional program expressing a bidirectional transformation includes a pair of programs – a forward program *get* that defines a view over a source and a backward program *put* that translates view updates to source updates – and provides strong guarantees about the well-behavedness of the *get* and the *put*. It is known to be challenging to develop bidirectional programs that are both well-behaved and practically useful.

Although many advanced synthesizers can generate unidirectional programs on relations (tables of relational database systems) from user-provided examples, the difficulty of synthesizing bidirectional programs from examples, especially the ones involving tables that have internal functional dependencies, still remains an unsolved issue.

In this thesis, we propose an approach to synthesizing well-behaved and practical bidirectional programs on relations from user-provided examples and data schemas with functional dependencies.

We start by synthesizing a *get* and decomposing it into a set of simple and atomic get_a whose corresponding put_a exists. With user-given functional dependencies and the set of atomic get_a, we forward-propagate functional dependencies from the source through intermediate relations to the view, so that the functional dependencies imposed on all relations can be clearly determined. We also compute atomic examples corresponding to (get_a, put_a). Then, the synthesis of (get, put) could be reduced into sub-synthesis of (get_a, put_a). To solve each sub-synthesis task, we design well-behaved templates for put_a given get_a. These templates encode not only the existing minimal-

effect atomic view update strategies but also the extra constraints and effects of functional dependencies, following the knowledge in the database community. With the set of well-designed templates, we adopt a modern example-and-template-based synthesizer named ProSynth to find put_a, and then combine all results $(get_a, put_a)s$ to form the final bidirectional program (get, put).

We have implemented our approach in two prototypes, SynthBX and SynthBP, which are developed using two different template designs. On a benchmark suite of 56 tasks from three sources, SynthBX successfully synthesizes well-behaved bidirectional programs for 52 tasks, with an average synthesis time of 19 seconds per task and within 3 seconds each for 37 of them, which shows practical usefulness of our approach. SynthBX is limited in supporting functional dependencies. SynthBP is a more advanced prototype with supporting templates for functional dependencies. It can automatically solve 37 out of 38 practical benchmarks. The overheads for handling functional dependencies are not too significant when the number of functional dependencies is reasonable.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This chapter includes three sections. First, we present a general background of this thesis (Section 1.1). Then, we discuss about research objectives (Section 1.2). Finally, we list the thesis organization and contributions (Section 1.3).

1.1 Background

Bidirectional transformations (BX) [1], originated from the view update problem [2, 3] in relational databases, are a mechanism for synchronizing or maintaining consistency between two related information where one is specified as the *source*, and the other as the *view*. If either the source or the view is changed, the other will be changed as well to restore consistency. People apply bidirectional transformations to solve a diversity of synchronization problems such as relational view update, model-driven software development, data exchange, and serializer/deserializer [4].

A typical bidirectional program [1, 5, 6] (Figure 1.1) expressing a bidirectional transformation consists of a pair of transformations: $get :: \mathbb{S} \rightarrow \mathbb{V}$ and $put :: \mathbb{S} \times \mathbb{V} \rightarrow \mathbb{S}$,

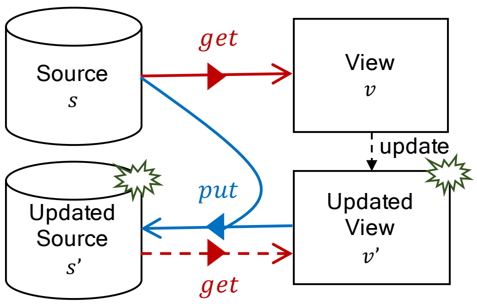


Figure 1.1: A bidirectional program ~ A well-behaved pair of (get, put)

where $\mathbb{S}$ and $\mathbb{V}$ are the source and view domains, respectively. The *get*, which is a forward transformation, produces a view over a source. The *put*, which is a backward transformation, takes the original source and a possibly updated view and produces an updated source.

To ensure consistency of a bidirectional transformation, *get* and *put* need to satisfy *well-behavedness properties* as below:

$$\forall s.\ put(s,\ get(s)) = s \tag{GetPut}$$

$$\forall s, v'.\ get(put(s, v')) = v' \tag{PutGet}$$

The GetPut law ensures that no change in the view is reflected as no change in the source, while the PutGet law ensures that all changes in the view are fully reflected to the source and applying *get* on the updated source produces exactly the updated view [1, 5, 6].

Despite over a decade of active study in developing well-behaved bidirectional programs, achieving both well-behavedness and practical usability across different domains remains a recognized challenge in bidirectional program development [7, 1, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]. A programmer can use a bidirectional language to write both transformations as a single, well-behaved program; however, existing bidirectional languages are designed for only a few specific domains. As argued in [14], such languages can be difficult to program in because they require the programmer to work in complex type systems and know tricky details. Alternatively, a general unidirectional language can be used to write two separate programs, *get* and *put*; ensuring the well-behavedness of the two programs poses a challenging task, however.

1.2 Research Objective

To enable easy construction of bidirectional programs on relational databases, we aim at an automatic synthesis of well-behaved bidirectional programs (get, put) from user-given examples over specified data schemas.

To see the problem concretely, consider a relational view update task where the schemas of the source and the view are adapted from a sample on SQLServerTutorial.net. In this case, the source database consists of two relations `staffs` and `customers` of the schemas

```
staffs(sid, name, city, active)
customers(cid, name, city)
```

and the view consists of one relation `tokyoac` of the schema

```
tokyoac(name).
```

The relation `staffs` stores staff information as records including the staff's identifier, name, city and active status. The relation `customers` stores a list of customers where each customer has an identifier, name and city. The view `tokyoac` stores the names of people whose certain characteristics. For each relation whether of the sources or the view, there is an updated relation that shares the same schema and describes the state of the relation after the updates take place (e.g., `staffs′`, `customers′`, `tokyoac′`).

Now the question is how we can automatically synthesize a well-behaved bidirectional program, *get* and *put*, just from an example that describes the bidirectional transformation behavior with the original source (Table 1.1a of `staffs` and Table 1.1b of `customers`), the original view (Table 1.1c of `tokyoac`, which is supposed to be obtained from *get*), the updated view (Table 1.1c' of `tokyoac′`), and the updated source (Table 1.1a' of `staffs′` and Table 1.1b' of `customers′`, both of which are supposed to be obtained from *put*). In other words, we wish to synthesize from the example a pair of component programs: a *view definition* (*query*) program *get* that describes how `tokyoac` is relationally defined from `staffs` and `customers`, and a *view update* program *put* that describes how `staffs′` and `customers′` are computed from `staffs`, `customers` and `tokyoac′`.

There are three main methods of program synthesis that are most related.

Table 1.1: A user-provided example including original source ((1.1a), (1.1b)) , original view ((1.1c)) , updated source ((1.1a'), (1.1b')) , and updated view ((1.1c'))

(1.1a) staffs

sid	name	city	active
10	Anna	Berlin	1
11	Ken	Tokyo	1
12	Jose	Rio	0
13	Yua	Tokyo	0

(1.1b) customers

cid	name	city
100	Logan	Denver
101	Olsen	Oslo
102	Kai	Tokyo
103	Luis	Lisbon
104	Mori	Tokyo

(1.1c) tokyoac

name
Ken
Kai
Mori

(1.1a') staffs'

sid	name	city	active
10	Anna	Berlin	1
~~11~~	~~Ken~~	~~Tokyo~~	~~1~~
12	Jose	Rio	0
13	Yua	Tokyo	0
14	*Shin*	*Tokyo*	*1*

(1.1b') customers'

cid	name	city
100	Logan	Denver
101	Olsen	Oslo
~~102~~	~~Kai~~	~~Tokyo~~
103	Luis	Lisbon
104	Mori	Tokyo
105	*Yuri*	*Tokyo*

(1.1c') tokyoac'

name
~~Ken~~
~~Kai~~
Mori
Shin
Yuri

original data *inserted data* ~~deleted data~~

First, a series of synthesis algorithms have been developed for obtaining bidirectional programs, as proposed in [14, 13, 15]. While these algorithms can use examples to derive well-behaved programs written in a bidirectional language, they are designed specifically for simple string processing and are limited to regular expressions, which cannot handle relations and query languages. Another algorithm, introduced in [18], can synthesize bidirectional programs written in a tree-oriented bidirectional language by constructing a sketch of *put* based on the given code of *get*, and then fills the sketch in a modular manner based on the properties of bidirectional programs. However, such a tree-oriented language is not well-suited for transformations commonly performed on relations [7].

Second, relational program synthesis [19] allows us to simultaneously generate multiple programs that satisfy a given relational specification (e.g., the GetPut and the PutGet laws) through example-based dual synthesis. It uses hierarchical tree automata to represent a relational version space that encodes all tuples of satisfying programs. As the space of the automata grows dramatically when dealing with complex underlying languages, it becomes challenging to handle practical relational query languages.

Third, many other general synthesis methods (e.g., ProSynth [20]) possibly take examples on relations as input and independently synthesize a *get* and a *put*, but they *cannot* guarantee the well-behavedness of the synthesized pair. ProSynth, a state-of-the-art synthesizer, is able to use a tabular input–output example to synthesize a program written in Datalog. Since a Datalog program is largely a set of rules, ProSynth reduces the synthesis problem to a rule-selection problem by (1) requiring the preparation of a fixed finite set of candidate rules by using *templates* and enumerations, and (2) selecting a subset of the prepared set that satisfies the given example. Preparing a "good" set of templates and candidate rules is very important against ProSynth because it forms a "good" search space where ProSynth can find an expected program. The reason that keeps ProSynth from guaranteeing the well-behavedness between two separately synthesized programs *get* and *put* is that there is no relationship between two sets of templates or candidate rules of the two programs. Naively enumerating all *get*s and *put*s rarely produces a well-behaved bidirectional program.

In fact, the essential limitation of the existing approaches to the synthesis of bidirectional programs is that they cannot directly cope with the complexity of relations and query languages along with the well-behavedness properties of bidirectional programs.

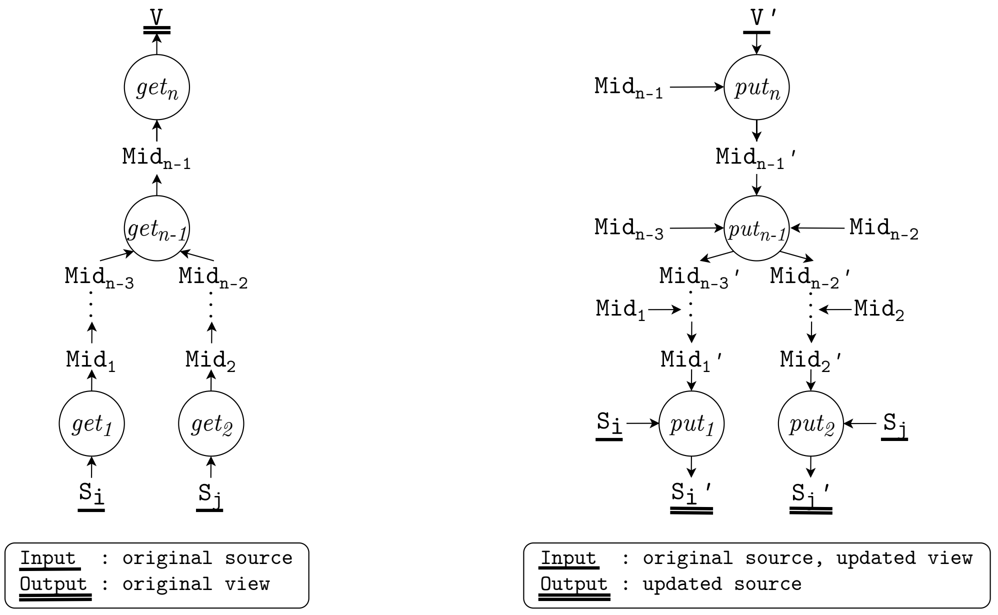
To cope with that complexity and well-behavedness, we propose a novel approach to the synthesis of bidirectional programs on relations, reducing a complex synthesis problem into simpler ones, solving simpler synthesis to obtain small well-behaved programs, and combining the results of simpler synthesis to achieve a complete well-behaved program. Our approach mainly consists of three steps as follows.

First, we synthesize a unidirectional *get* from the given tables of the original source and the original view. Then we can *decompose* this *get* to a combination of atomic queries $C_g(get_1, \dots, get_n)$ (Figure 1.2a). Now, based on the PutGet law, we can compute unknown intermediate tables (examples) for get_a and put_a [1]

$$\begin{gathered}\texttt{Mid}_1, \dots, \texttt{Mid}_{n-1},\\ \texttt{Mid}_1', \dots, \texttt{Mid}_{n-1}'\end{gathered}$$

by applying the decomposed program C_g on the given original and updated source

[1]The subscript a in get_a and put_a indicates "atomic". We employ get_a and put_a to represent abstract atomic programs.



(a) $C_g(get_1, \ldots, get_n)$ (b) $C_p(put_1, \ldots, put_n)$

Figure 1.2: An example of combinations for *get* and *put*

tables, one by one. The synthesis of (get, put) is now reduced to the sub-synthesis of (get_a, put_a).

Second, for each atomic query described by get_a, we synthesize an atomic view update put_a such that (get_a, put_a) forms a well-behaved bidirectional program. We solve this synthesis problem by *templatizing* the existing *minimal well-behaved view update strategies* [21, 22, 23] to generate a set of candidate rules and adapting ProSynth. With the templates, we can prepare the space for efficient synthesis of put_a for get_a while guaranteeing that they are well-behaved.

Last, we construct a *put* program as a combination of atomic view update programs $C_p(put_1, \ldots, put_n)$ (Figure 1.2b) where the evaluation order in C_p is reversed from that in C_g. The final bidirectional program is a pair of (C_g, C_p) that basically is a combination of the atomic well-behaved pairs $(get_1, put_1), \ldots, (get_n, put_n)$, and this final program is guaranteed to be well-behaved due to the well-behavedness properties of a composition of bidirectional programs [1].

A challenge associated with our current approach is that the minimal well-behaved view update strategies introduced in [21, 22, 23] do not cover extra constraints and

Table 1.2: A user-provide example related to functional dependencies including original source (1.2a) , original view (1.2b) , updated source (1.2a') , and updated view (1.2b')

(1.2a) original source

S

A	B	C	D
1	b_1	F	d_1
1	b_1	T	d_1
2	b_2	T	d_2

(1.2b) original view

V

A	B	C
1	b_1	T
2	b_2	T

(1.2a') updated source

S'

A	B	C	D
~~1~~	~~b_1~~	~~F~~	~~d_1~~
~~1~~	~~b_1~~	~~T~~	~~d_1~~
1	b_2	*F*	d_1
1	b_2	*T*	d_1
2	b_2	T	d_2

(1.2b') updated view

V'

A	B	C
~~1~~	~~b_1~~	~~T~~
1	b_2	*T*
2	b_2	T

original data *inserted data* ~~deleted data~~

effects. Things become more complicated and difficult when the examples are defined over tables that have internal dependencies such as *functional dependencies* (FDs).

Let us consider another example in Table 1.2 that contains tables with clearer functional dependencies. This example includes Table 1.2a of the original source S, Table 1.2b of the original view V, Table 1.2a' of the updated source S', and Table 1.2b' of the updated view V', where S and S' share the same schema of (A, B, C, D), while V and V' share the same schema of (A, B, C).

If we look closely in Tables 1.2a and 1.2a', there seem to be FDs on the original and updated sources: $A \longrightarrow B$ and $A \longrightarrow D$. If there are FDs on relation schemas, they impose some specific internal *constraints*, such as that data on a relation r must agree on the FDs of r (e.g., in the updated source S', two tuples with the same A-value of 1 must have the same B-value and D-value).

FDs also cause *effects* when reflecting view updates into source updates: If we run a bidirectional program backward, more changes would occur in the source so that all the data on the source would match the FDs. For instance, when $\langle 1, b_1, T\rangle$ is replaced by $\langle 1, b_2, T\rangle$ in the updated view V', then the updated source S' includes not only $\langle 1, b_2, T, d_1\rangle$ (due to the minimal strategies) but also $\langle 1, b_2, F, d_1\rangle$ even though there is no corresponding tuple $\langle 1, b_2, F\rangle$ in V'.

The existing minimal well-behaved view update strategies are not rich enough to cover the constraints and effects of FDs.

In practice, tables with dependent data frequently arise. Addressing update strategies related to FDs becomes crucial for resolving many practical view update tasks over these tables. Support for synthesis involving FDs is needed to avoid overlooking solutions for real-world scenarios.

We propose an approach to synthesizing bidirectional programs on relations from examples with FDs. To avoid diving into the efficient discovery of essential FDs, which is a well-known difficult challenge in database research [24, 25], we require the user to explicitly provide the possible FDs in a special structure called tree form for relations of the source. This approach improves upon the one mentioned above in the following two points.

First, we forward-propagate the provided FDs from the source through intermediate relations to the view once we have the set of atomic get_a satisfying the given example (i.e., after the decomposition of a synthesized query get). Then, for each atomic part, we obtain the FDs of the corresponding source and view.

Second, besides templates of minimal well-behaved view update strategies, we prepare more templates for the constraints and effects of FDs to enhance the search spaces of the synthesis of put_as. The new templates are designed based on the constraint semantics of FDs and the other non-minimal view update strategies [7] which describe the effects of FDs in tree form when performing updates. The combination of the new templates encoding constraints and effects of FDs, and the templates of minimal well-behaved view update strategies makes the candidate rules more diverse, thereby giving ProSynh more choices when performing synthesis.

1.3 Organization and Contributions

We organize the rest of the thesis as follows. Chapters 3-6 (Figure 1.3) are the main contributions of this thesis, which include our approaches, algorithms and implementations for synthesizing bidirectional programs on relations.

Chapter 2. We present related works on relational databases, non-recursive Datalog (NR-Datalog*), bidirectional programs in NR-Datalog*, constraints and effects of functional dependencies (FDs) when performing updates, the view update problem and view update strategies, program synthesis, and a survey of existing example-based

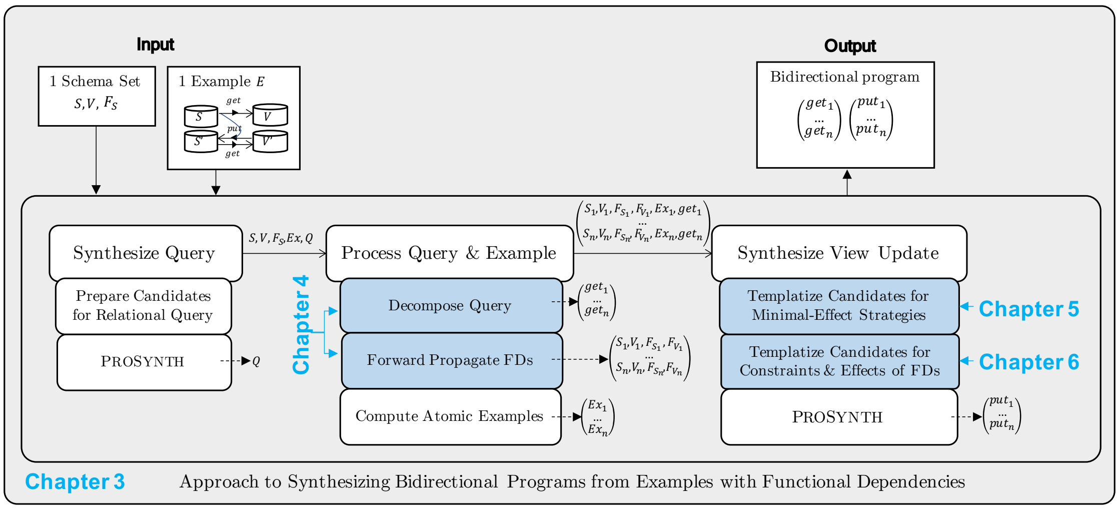

Input
1 Schema Set S, V, F_S
1 Example E
get
put
get
Output
Bidirectional program
$\begin{pmatrix} get_1 \\ \dots \\ get_n \end{pmatrix}$ $\begin{pmatrix} put_1 \\ \dots \\ put_n \end{pmatrix}$
Synthesize Query
Prepare Candidates for Relational Query
PROSYNTH
Q
S, V, F_S, Ex, Q
Chapter 4
Process Query & Example
Decompose Query
Forward Propagate FDs
Compute Atomic Examples
$\begin{pmatrix} get_1 \\ \dots \\ get_n \end{pmatrix}$
$\begin{pmatrix} S_1, V_1, F_{S_1}, F_{V_1} \\ \dots \\ S_n, V_n, F_{S_n}, F_{V_n} \end{pmatrix}$
$\begin{pmatrix} Ex_1 \\ \dots \\ Ex_n \end{pmatrix}$
$\begin{pmatrix} S_1, V_1, F_{S_1}, F_{V_1}, Ex_1, get_1 \\ \dots \\ S_n, V_n, F_{S_n}, F_{V_n}, Ex_n, get_n \end{pmatrix}$
Synthesize View Update
Templatize Candidates for Minimal-Effect Strategies
Templatize Candidates for Constraints & Effects of FDs
PROSYNTH
$\begin{pmatrix} put_1 \\ \dots \\ put_n \end{pmatrix}$
Chapter 5
Chapter 6
Chapter 3
Approach to Synthesizing Bidirectional Programs from Examples with Functional Dependencies

Figure 1.3: Main contributions

synthesizers including ProSynth which we will rely on to develop our synthesizer later.

Chapter 3. We present the high-level of our proposed approach to synthesizing well-behaved bidirectional programs on relations from examples and functional dependencies (Figure 1.3). Our approach reduces the synthesis of a well-behaved pair of (get, put) to the synthesis of atomic well-behaved pairs. The combination of atomic well-behaved pairs would be well-behaved because of the well-behavedness of a composition of bidirectional programs. After we found a query *get* with ProSynth, we decompose the query into a set of atomic queries and forward-propagate information of FDs from the source through intermediate relations to the view. For each atomic query, we synthesize a corresponding atomic view update by templatizing minimal view update strategies and by templatizing the constraints and effects of FDs to generate candidate rules so that ProSynth can be adapted efficiently. The templates are well-designed to encode the well-behavedness of the atomic query and the corresponding atomic view update.

Chapter 4. We present a method for decomposing a query in NR-Datalog* to a set of atomic queries in NR-Datalog*, and an algorithm for forward-propagating FDs in tree form from the source to the view over a set of atomic queries. Having access to information about atomic queries and the FDs of all relations is beneficial when designing templates for the corresponding atomic view updates.

Chapter 5. We present Datalog templates that encode the minimal view update strategies. We implement our synthesis approach without FDs in a tool called SynthBX and demonstrate its power, efficiency, and sensitivity to example sizes through 56 benchmark tasks from three different sources. SynthBX successfully solves 52 tasks, takes an average of 19 seconds to find a solution, and less than 3 seconds each for 37 of them.

Chapter 6. We present algorithms for templatizing the constraints and effects of FDs on relations. They help to strengthen the search space of the ProSynth-based synthesis of well-behaved put_a. We implement our proposed approach with the FDs processing steps in a tool called SynthBP that can automatically solve 37/38 practical synthesis benchmarks. The overheads for handling FDs are not too significant when the number of functional dependencies is reasonable.

Chapter 7. We conclude the thesis by summarizing our contributions and discussing possible future works.

2 Related Work

In this chapter, we briefly overview relational databases [26] (Section 2.1), non-recursive Datalog [27] (Section 2.2), and bidirectional programs [1] (Section 2.3). We then review the concepts of functional dependencies (FDs), the tree structure of FDs, and relation revisions over that structure when updating relational views [7] (Section 2.4), and summarize related work on the view update problem [2] (Section 2.5). Finally, we cover a bit about program synthesis [28, 29] (Section 2.6) and survey many existing example-based synthesis approaches (Section 2.7) including ProSynth [20] which we will rely on to develop our synthesizer later.

2.1 Relational Databases

A database $\mathcal{D}$ is a finite map from relation schemas $\mathbb{R}$ to relations R. A *relation schema* is a finite nonempty set $\mathbb{R}$ of *attributes*. Each attribute A of a relation schema $\mathbb{R}$ is associated with a domain $dom(A)$ (or $\mathbb{A}$), which represents the possible values a that could appear in column A. If X and Y are attribute sets, we may write XY for the union

$X \cup Y$. If $X = \{A_1, A_2, \ldots, A_m\}$, we may write $A_1A_2 \ldots A_m$ for X. The singleton $\{A\}$ may be written as simply A. A *tuple* t (alternatively written as $\langle t_1, \ldots, t_k \rangle$) over $\mathbb{R}$, written $t :: \mathbb{R}$, is a function $t : \mathbb{R} \rightarrow \bigcup_{A \in \mathbb{R}} \mathbb{A}$, where $\forall A \in \mathbb{R}.\ t(A) \in \mathbb{A}$. We write $t[X]$ for the projection of a tuple t over $\mathbb{R}$ to X if $X \subseteq \mathbb{R}$. A *relation* R over $\mathbb{R}$, written $R :: \mathbb{R}$, is a finite set of tuples over $\mathbb{R}$. For $R :: A_1 \ldots A_m$, we sometimes write $R(A_1 : \mathbb{A}_1, \ldots, A_m : \mathbb{A}_m)$ and say that R is of the schema of $(A_1, \ldots, A_m)$.

For example, in Table 1.2, we said that relations S and S' share the same schema of (A, B, C, D), which means that

$$\begin{array}{lcl} S :: ABCD & \text{or} & S(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, D : \mathbb{D}) \\ S' :: ABCD & \text{or} & S'(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, D : \mathbb{D}) \end{array}$$

We can describe relations S and S' as sets of tuples as below:

$$\begin{aligned} S &= \{\langle 1, b_1, F, d_1 \rangle, \langle 1, b_1, T, d_1 \rangle, \langle 2, b_2, T, d_2 \rangle\} \\ S' &= \{\langle 1, b_2, F, d_1 \rangle, \langle 1, b_2, T, d_1 \rangle, \langle 2, b_2, T, d_2 \rangle\}. \end{aligned}$$

A database $\mathcal{D}$ could be represented as a set of $R(t_1, \ldots, t_k)$ corresponding to the tuple $\langle t_1, \ldots, t_k \rangle$ of relation R in $\mathcal{D}$. The source database including only relation S in Table 1.2 could be written as

$$\{S(1, b_1, F, d_1), S(1, b_1, T, d_1), S(2, b_2, T, d_2)\}$$

while the source database with two relations `staffs` and `customers` in Table 1.1 could be written as

```
{  staffs(10,Anna,Berlin,1),staffs(11,Ken,Tokyo,1),
   staffs(12,Jose,Rio,0),staffs(13,Yua,Tokyo,0),
   customers(100,Logan,Denver),customers(101,Olsen,Oslo),
   customers(102,Kai,Tokyo),customers(103,Luis,Lisbon),
   customers(104,Mori,Tokyo)  }
```

$$
\begin{array}{lll}
Program & := & Clause^{+} \\
Clause & := & Fact \mid Rule \\
Fact & := & R(\ Constant\ (,\ Constant)^{*}\ ). \\
Rule & := & Head\ \ :-\ Body. \\
Head & := & Literal \\
Body & := & Disjunction \\
Disjunction & := & Conjunction\ (;\ Conjunction)^{*} \\
Conjunction & := & Item\ (,\ Item)^{*} \\
Item & := & Literal \mid \neg\ Literal \mid Constraint \\
Literal & := & R(\ Term\ (,\ Term)^{*}\ ) \\
Term & := & Variable \mid Constant \\
Constraint & := & Variable\ Op\ Constant \\
\end{array}
$$

$$R \in RelationSymbol \qquad Variable \in \{v_i\}_{i \geq 0} \cup \{\_\}$$

$$Op \in \{=, \neq, >, <, \geq, \leq\}$$

Figure 2.1: Syntax of NR-Datalog* (based on SOUFFLÉ)

2.2 NR-Datalog*

NR-Datalog*[27] is a fragment of Datalog with negation, built-in predicates of comparison and no recursion. We follow the syntax of Datalog given by SOUFFLÉ [1] to express the syntax of NR-Datalog* (Figure 2.1).

An NR-Datalog* program P is a finite nonempty set of *clauses*, where each clause is either a *fact* "$R(c_1, \ldots, c_n)$." or a *rule* "$H\ :-\ B_1, \ldots, B_n$.", where R is a *relation symbol*, the c_is are constants, H is a *head*, and $\{B_1, \ldots, B_n\}$ is a *body*. Let us consider the following program:

$$
\begin{array}{ll}
cl_1 & S(0, 1). \\
cl_2 & V(v_1)\ :-\ S_1(v_1, v_2)\ ,\ \neg\ S_2(v_1). \\
cl_3 & V(v_1)\ :-\ S_1(v_1, v_2)\ ,\ v_2 = 2. \\
\end{array}
$$

This program consists of one fact cl_1 and two rules $\{cl_2, cl_3\}$.

The head H is a *positive literal* (or *atom*) of the form $r(t_1, \ldots, t_n)$ (alternatively written as $r(\vec{t})$) (e.g., $V(v_1)$ in cl_2), where r is a relation symbol and each argument t_i is a *term*, which is generally either a variable or a constant. The body $\{B_1\ , \ldots,\ B_n\}$ is a *conjunction* of B_is which are each either a positive literal (e.g., $S_1(v_1, v_2)$ in cl_2),

[1]https://souffle-lang.github.io/program

a *negative literal* (e.g., $\neg S_2(v_1)$ in cl_2) or a *constraint* (e.g., $v_2 = 2$ in cl_3). A variable occurring only once in a rule (e.g., v_2 in cl_2) could be conventionally replaced by an anonymous variable denoted as "_".

We say a rule is *normal* if it has form "$H\ :-\ B_1\ , \ldots ,\ B_n.$". Normal rules with a common head H could be abbreviated as a single-line rule whose head is H and whose body is a *disjunction* that joins conjunctions, each of which expresses the body of a normal rule, by ";"s. Simply put, a single-line rule of form "$H\ :-\ B_1; \ldots ; B_n.$" is a short expression of n rules "$H\ :-\ B_1.$" , … , "$H\ :-\ B_n.$". For instance, we can rewrite cl_2 and cl_3 as cl_{23} as below:

$$cl_{23} \qquad V(v_1)\ :-\ S_1(v_1, v_2)\ ,\ \neg\ S_2(v_1)\ ;\ S_1(v_1, v_2)\ ,\ v_2 = 2.$$

To ensure that the set of all facts derived from P is finite, P needs to satisfy *safety* conditions [27]. If we convert all rules in P to normal form, for each rule r of the form "$H\ :-\ B$",

1. Each variable appearing in H must also appear in B;
2. Each non-anonymous variable appearing in a negative literal of B must also appear in a positive literal of B;
3. The variable appearing in a constraint of B must be non-anonymous and either appear in a positive literal of B or be bound by equality to a constant.

The semantics of a Datalog program are defined using Herbrand models of first-order logic formulas [27]:

1. Each fact F represents an atomic formula.
2. Each rule R of the form "$H\ :-\ B_1\ , \ldots ,\ B_n$" represents a first-order formula of the form "$\forall v_1, \ldots, v_m.\ B_1 \wedge \ldots \wedge B_n \longrightarrow H$", where the v_is are all variables occurring in R.

For example, cl_1 means that there is a tuple $(0, 1)$ in S, and cl_2 means that if (v_1, v_2) exists in S_1 and there is no (v_1) in S_2, then (v_1) is in V.

2.3 Bidirectional Program

A bidirectional program [1] consists of a pair of transformation programs – a forward $get :: \mathbb{S} \rightarrow \mathbb{V}$, which produces a view over a source, and a backward $put :: \mathbb{S} \times \mathbb{V} \rightarrow \mathbb{S}$, which reflects the changes to a view back to the source – and this pair must satisfy the following *well-behavedness* properties:

$$\begin{array}{lll} \forall S. & put(S,\ get(S)) = S & [\textsc{GetPut}] \\ \forall S, V'. & get(put(S, V')) = V' & [\textsc{PutGet}] \end{array}$$

The GetPut states that if one gets a view $V = get(S)$ from a source S and puts it back again, S will not change; the PutGet states that an updated view V' can be fully restored by applying get over the updated source $S' = put(S, V')$.

We can use NR-Datalog* to construct a bidirectional program in form of a well-behaved pair (get, put). Writing a get that defines a view over a source is easier since it is like writing a relational query. The harder problem is to write a put that describes update propagation from the view back to the source. To describe such propagation, the concept of delta relations [30, 31, 16] is used to represent changes to the source and the view.

A delta relation ΔR over a schema $\mathbb{R}$, denoted as $\Delta R :: \Delta\mathbb{R}$, is a pair $(\Delta R^-, \Delta R^+)$ of disjoint relations of the same schema $\mathbb{R}$. ΔR^- and ΔR^+ are used to capture insertions and deletions against R, respectively. For instance, consider a relation $R = \{\langle 1, 2\rangle, \langle 3, 4\rangle\}$ and a corresponding updated relation $R' = \{\langle 1, 2\rangle, \langle 5, 6\rangle\}$, then $\Delta R^- = \{\langle 3, 4\rangle\}$ and $\Delta R^+ = \{\langle 5, 6\rangle\}$. We call data in R and R' as *state-based data*, and data in ΔR^- and ΔR^+ as *delta-based data*. Since a relation appearing in a bidirectional program could be either a source or a view, we could use different notations to express source delta and view delta relations. From here on, we denote $\Delta R \equiv (\Delta R^-, \Delta R^+) :: \Delta\mathbb{R}$ as a source delta relation corresponding to a source relation $R :: \mathbb{R}$, and $\delta R \equiv (\delta R^-, \delta R^+) :: \delta\mathbb{R}$ as a view delta relation corresponding to a view relation $R :: \mathbb{R}$.

The following two examples describe writing bidirectional programs in NR-Datalog*.

Example 2.1. Given a source $S = \{\langle 1, 2\rangle, \langle 3, 4\rangle\}$ of schema $S(Num, Num)$. We can define a view $V = \{\langle 1\rangle, \langle 3\rangle\}$ of schema $V(Num)$ by writing a program $get = \{r_1^e\}$ where

$$r_1^e \qquad V(v_1) \ :-\ S(v_1, v_2).$$

If the view is changed to $V' = \{\langle 1\rangle, \langle 5\rangle\}$, to propagate the update, we can write a program $put = \{r_i^e\}_{i=1}^{6}$ that takes S and V' as input where

$$
\begin{array}{ll}
r_2^e & \delta V^-(v_1) \ :- \ V(v_1) \ , \ \neg \ V'(v_1). \\
r_3^e & \delta V^+(v_1) \ :- \ V'(v_1) \ , \ \neg \ V(v_1). \\
r_4^e & \Delta S^-(v_1, v_2) \ :- \ V^-(v_1) \ , \ S(v_1, v_2). \\
r_5^e & \Delta S^+(v_1, v_2) \ :- \ V^+(v_1) \ , \ \neg \ S(v_1, v_2) \ , \ v_2 = 0. \\
r_6^e & S'(v_1, v_2) \ :- \ S(v_1, v_2), \ \neg \ \Delta S^-(v_1, v_2) \ ; \ \Delta S^+(v_1, v_2).
\end{array}
$$

The evaluation of the *put* is as follows: deriving $V = \{\langle 1\rangle, \langle 3\rangle\}$ with *get*; computing delta relations against V ($\delta V^- = \{\langle 3\rangle\}, \delta V^+ = \{\langle 5\rangle\}$); computing delta relations against S ($\Delta S^- = \{\langle 3, 4\rangle\}, \delta S^+ = \{\langle 5, 0\rangle\}$); and outputting $S' = \{\langle 1, 2\rangle, \langle 5, 0\rangle\}$.

To verify the well-behavedness of the pair (get, put), we can either manually check if it satisfies two laws GetPut and PutGet or adapt an automated tool named Birds [16]. Birds enables users to write a view update program *put* using delta relations. In certain cases, it can derive a corresponding query *get* and validate the well-behavedness of these two programs through internal logic inferences. If users write not only *put* but also *get*, Birds will expand the logic inference with the rules of *get* and perform the validation of well-behavedness.

The changes against the view may not uniquely determine the changes against the source, and in the worst case, it could be impossible to propagate the changes to the view back to the source [2, 21, 22, 23]. For instance, if we modify rule r_5^e with constraint $v_2 = 0$ to rule r_5^{e*} with constraint $v_2 = 1$, the program $put^* := put - \{r_5^e\} \cup \{r_5^{e*}\}$ returns $S'^* = \{\langle 1, 2\rangle, \langle 5, 1\rangle\}$ and also forms with the given $get = \{r_1^e\}$ a well-behaved bidirectional program. Using appropriate examples ((S, V, S', V') or (S, V, S'^*, V')) can guide a correct construction of a well-behaved pair (get, put). ▲

Example 2.2. Suppose that we have written a program $get = \{r_1^b\}$ to define a view $V = \{\langle 1\rangle, \langle 3\rangle\}$ of schema $V(Num)$ over a source $S = \{\langle 1, 2\rangle, \langle 3, 4\rangle\}$ of schema $S(Num, Num)$.

$$
r_1^b \quad V(v_1) \ :- \ S(v_1, v_2).
$$

If the view is changed to $V' = \{\langle 1\rangle, \langle 5\rangle\}$, to propagate the update, we can write a program *put* = $\{r_2^b, r_3^b, r_4^b\}$ that takes S and V' as input where

$$
\begin{array}{ll}
r_2^b & \Delta S^-(v_1,v_2) \ :- \ S(v_1,v_2)\ ,\ \neg\ V'(v_1). \\
r_3^b & \Delta S^+(v_1,v_2) \ :- \ V'(v_1)\ ,\ \neg\ S(v_1,v_2)\ ,\ v_2 = 0. \\
r_4^b & S'(v_1,v_2) \ :- \ S(v_1,v_2),\ \neg\ \Delta S^-(v_1,v_2)\ ;\ \Delta S^+(v_1,v_2).
\end{array}
$$

Each rule for computing ΔS (e.g. r_2^b and r_3^b) describes a *view update strategy*. A run of *put* evaluates $\Delta S^- = \{\langle 3,4\rangle\}$ and $\Delta S^+ = \{\langle 5,0\rangle\}$, then outputs $S' = \{\langle 1,2\rangle, \langle 5,0\rangle\}$. We can either manually verify the well-behavedness or adapt Birds [16] to perform the verification.

Note in a special point that the *put* here is written in a different way than the corresponding one in Example 2.1. In the current example, *put* computes ΔS from S and V' (e.g. r_2^b, r_3^b) rather than from S and δV (e.g. $r_1^e, \ldots, r_5^e$) where δV is computed from V' and the result V of applying the given *get*. Both ways of writing allow good programs to be written. ▲

2.4 Functional Dependency: Constraints & Effects

A *functional dependency* over $\mathbb{R}$ is written as $X \rightarrow Y :: \mathbb{R}$, where $X \subseteq \mathbb{R}$ and $Y \subseteq \mathbb{R}$ refer to the *left-hand side* (*lhs*) and *right-hand side* (*rhs*) of the FD, respectively. If $X \rightarrow Y :: \mathbb{R}$ and $R :: \mathbb{R}$, we say that R satisfies *constraints* $X \rightarrow Y$, written $R \models X \rightarrow Y$, if $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$ for all $t_1, t_2 \in R$. It is conventional to write FD $A \rightarrow BC$ to mean FD $\{A\} \rightarrow \{B, C\}$. Let $\mathcal{F}$ be a set of FDs over $\mathbb{R}$, written $\mathcal{F} :: \mathbb{R}$. We write $R \models \mathcal{F}$ to mean that $\forall X \rightarrow Y \in \mathcal{F}.\ R \models X \rightarrow Y$. We can normalize $X \rightarrow Y$ as $\{X \rightarrow A_i\}_{A_i \in Y}$. To simplify the presentation later, we sometimes write $R(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, \mathcal{F} = \{A \rightarrow B\})$ for $R :: ABCD$, $\mathcal{F} = \{A \rightarrow B\}$, $\mathcal{F} :: ABCD$ and $R \models F$.

Bohannon et al. [7] introduced a special structure of FDs called a *tree form*, in which they can construct properly bidirectional programs with FDs. Given an FD set $\mathcal{F}$, let $V_{\mathcal{F}} = \{X \mid X \rightarrow Y \in \mathcal{F}\} \cup \{Y \mid X \rightarrow Y \in \mathcal{F}\}$ and $E_{\mathcal{F}} = \{(X, Y) \mid X \rightarrow Y \in \mathcal{F}\}$; then, we say $\mathcal{F}$ is in *tree form* if the directed graph $T_{\mathcal{F}} = (V_{\mathcal{F}}, E_{\mathcal{F}})$ is a forest and no distinct nodes of T_F have common attributes. If $\mathcal{F}$ is in tree form, we write $roots(\mathcal{F})$ and $leaves(\mathcal{F})$ for the sets of root and leaf nodes in $T_{\mathcal{F}}$, respectively. Every $\mathcal{F}$ in tree form has a unique *canonical tree from* $\mathcal{F}^*$ where $size(X) = 1$ for all $X \in leaves(\mathcal{F})$.

To calculate the *effects* of FDs putting changes from a view to a source, Bohannon et al. [7] introduced revision operators. Given an FD set $\mathcal{F} :: \mathbb{R}$ and relations $R_1, R_2 :: \mathbb{R}$ such that $R_1 \models \mathcal{F}$, a relation revision $R_1 \hookleftarrow_{\mathcal{F}} R_2$ [7] computes a new relation $R_1^* :: \mathbb{R}$ similar to R_1 whose tuples do not conflict with those of R_2 on $\mathcal{F}$. For example, given $\mathbb{R} = AB$, $\mathcal{F} = \{A \rightarrow B\}$, $R_1 = \{\langle 1, b_2\rangle, \langle 2, b_1\rangle, \langle 3, b_3\rangle\}$, and $R_2 = \{\langle 2, b_2\rangle, \langle 4, b_4\rangle\}$, we have that $R_1 \hookleftarrow_{\mathcal{F}} R_2$ computes $R_1^* = \{\langle 1, b_2\rangle, \underline{\langle 2, b_2\rangle}, \langle 3, b_3\rangle\}$.

The relation revision operation $R_1 \hookleftarrow_{\mathcal{F}} R_2$ is expressed by a set of tuple revisions $t_1 \hookleftarrow_{\mathcal{F}} R_2$, where $t_1 \in R_1$, $R_1 :: \mathbb{R}$, $R_2 :: \mathbb{R}$, and $\mathcal{F} :: \mathbb{R}$ is an FD set in tree form such that $R_1 \models \mathcal{F}$.

$$R_1 \hookleftarrow_{\mathcal{F}} R_2 = \{t_1 \hookleftarrow_{\mathcal{F}} R_2 \mid t_1 \in R_1\}$$

Tuple revision $t_1 \hookleftarrow_{\mathcal{F}} R_2$ [7] can be defined by recursion over the forest $T_{\mathcal{F}}$ that indicates the update propagation from the roots to the leaves:

$$\begin{aligned}
&t_1 \hookleftarrow_{\emptyset} R_2 = t_1\\
&t_1 \hookleftarrow_{\{X \rightarrow Y\} \cup \mathcal{F}'} R_2 = \begin{cases} (t_1 \leftharpoonup t_2[Y]) \hookleftarrow_{\mathcal{F}'} R_2 & \text{if } \exists\, t_2 \in R_2.\ t_1[X] = t_2[X] \\ t_1 \hookleftarrow_{\mathcal{F}'} R_2 & \text{otherwise} \end{cases}\\
&\text{where } X \in roots(\mathcal{F}) \text{ and } \mathcal{F} = \{X \rightarrow Y\} \cup \mathcal{F}'
\end{aligned}$$

If $\mathcal{F}$ is empty, tuple revision simply returns t_1. Otherwise, there must be at least one FD $X \rightarrow Y$ where $X \in roots(T_{\mathcal{F}})$. If t_1 and some $t_2 \in R_2$ have the same values against X, we return a copy of t_1 whose $t_1[Y]$ has been updated according to $t_2[Y]$ (written as $t_1 \leftharpoonup t_2[Y]$); otherwise, t_1 is returned. The remaining FDs continue to be considered recursively.

From the definitions of revisions, we have:

$$\begin{aligned}
&R_1 \hookleftarrow_{\emptyset} R_2 = R_1\\
&R_1 \hookleftarrow_{\{X \rightarrow Y\} \cup \mathcal{F}'} R_2 = (R_1 \hookleftarrow_{\{X \rightarrow Y\}} R_2) \hookleftarrow_{\mathcal{F}'} R_2
\end{aligned}$$

2.5 View Update

The view update problem [2] (Figure 2.2) could be considered as the starting origin of bidirectional transformations and bidirectional programs. Given a query *get* that

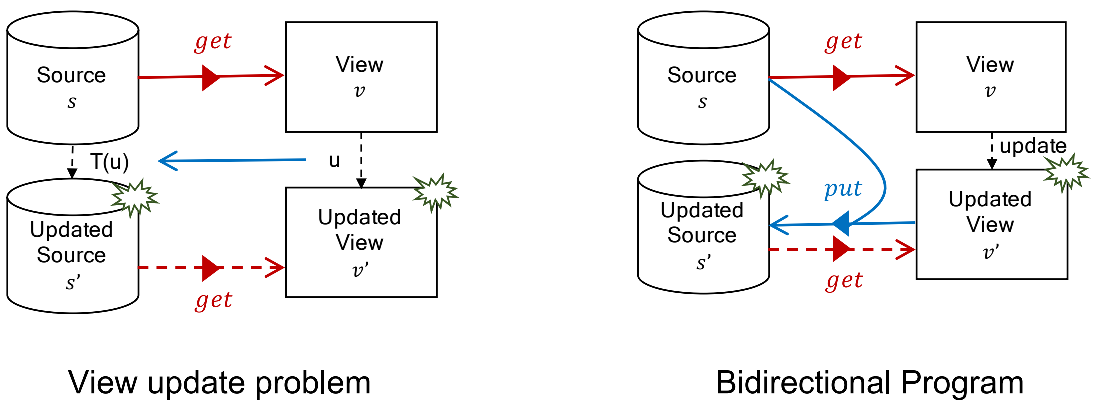


Figure 2.2: View update problem & bidirectional program

defines a view v from a source databases s, the view update problem is to derive an update translator T that maps each update u on v to an update $T(u)$ on s such that $u(v) = get(T(u)(s))$.

Constructing bidirectional programs is a common practical solution for the view update problem, which has been investigated for decades [2, 32, 21, 22, 23, 7, 8, 31, 16]. The view update problem proved to be ambiguous because the update to the view may not uniquely determine the update to the source, and in the worst case, it could be impossible to propagate the update to the view back to the source. Keller [21, 22] presented a complete list of update strategies that satisfy certain criteria for a large class of selection, projection and join views on BCNF relations, with the choice of strategies being made through a dialog at the view definition time. Larson [23] listed update rules for simple and nested views involving set difference, union, intersection, selection, projection and join views by using both syntactic and semantic knowledge at the view definition time and the view update time. Bohannon [7] introduced a *get*-based bidirectional query language (relational lens) targeting selection, projection and join queries with the use of FDs. Horn [31] made relational lenses more practical by performing incrementalization through a series of delta relations. Tran [16] proposed a language-based approach and system called Birds, based on the theory of *put*-based bidirectional transformations, to construct bidirectional programs in a fragment of Datalog as a set of rules of delta relations and verifying well-behavedness.

In our work, the templates of minimal update strategies are designed based on the works of Keller [21, 22] and Larson [23], while the templates encoding the effects of

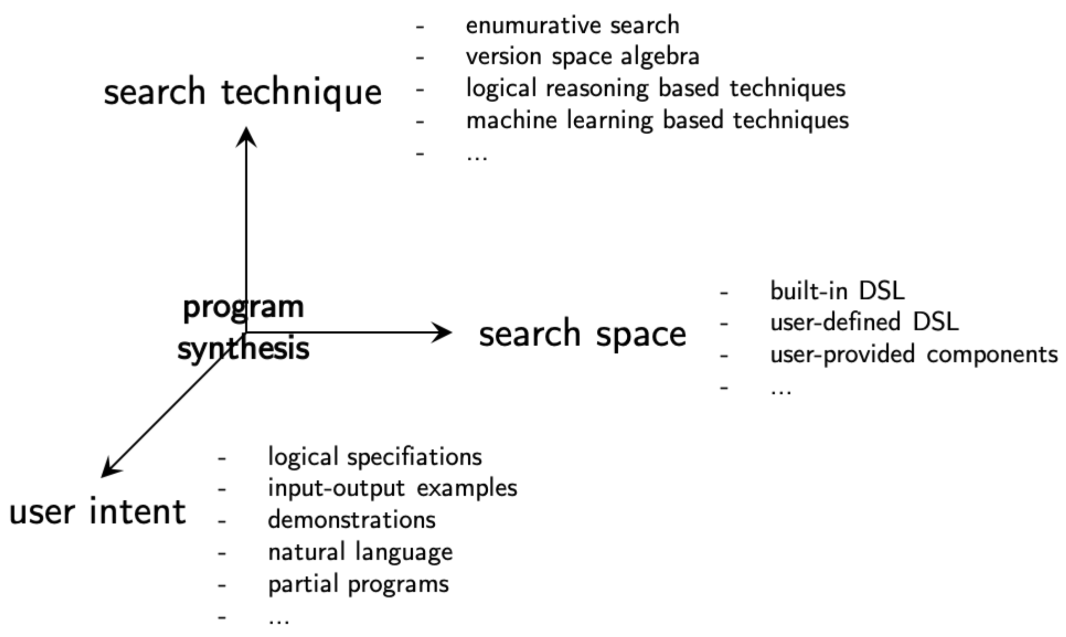


Figure 2.3: Three key dimensions of a typical program synthesis problem

FDs are inspired by the works of Bohannon [7] and Horn [31].

2.6 Program Synthesis

Program synthesis is the task of synthesizing programs in the underlying programming language that satisfy the user intent expressed in the form of some specification or constraints. Unlike compilers which take as input programs written in a high-level code and normally convert them to low-level machine code using syntax-directed translations, synthesizers typically perform some kind of search over the space of programs to generate a program that is consistent with diverse constraints [28]. To this day, program synthesis is applied successfully in various fields, including relational databases, software development, data cleaning, and biology [29].

A synthesizer is mainly characterized by three key dimensions (Figure 2.3): the kind of constraints that express user intent, the space of programs over which it searches, and the search technique it employs [28, 29].

The user intent can be expressed in a variety of forms such as logical specification which represents logical relations between inputs and outputs of a program, natural language descriptions, a set of input-output examples, demonstrations (traces) which describe, step-by-step, how the program should behave on a given input, and higher-order, inefficient or partial programs [28].

The search space can be over imperative or functional programs with possible restrictions on the control structure or the operator set, or over restricted models of computations such as regular or context-free grammar. It needs a good balance between expressiveness and efficiency. On one hand, the program space should be expressive enough to include programs that users care about. On the other hand, the program space should be restrictive enough so that it is amenable to efficient search, and it should be over a domain of programs that are amenable to efficient reasoning. The program space can be restricted to a subset of existing programming language or to a specifically designed domain-specific language [28].

The search technique can be based on enumerative search, version space algebra, logical reasoning based techniques, machine learning based techniques, or some combination of them [28]. The enumerative search techniques enumerate programs in the search space in some order and check if a program satisfies the synthesis constraints. Instead of enumerating programs one-by-one, version space algebra techniques enumerate programs by hypothesis/version spaces which basically group multiple programs satisfying some condition together. Logical reasoning based techniques reduce the program synthesis to the problem of solving an SAT/SMT formula and let an off-the-shelf SAT/SMT solver efficiently explore the search space. Machine learning based techniques learn a probability distribution over the space of programs that are more likely to be consistent with the given specification.

2.7 Existing Example-Based Synthesis Approaches

2.7.1 Synthesis of Bidirectional Programs

There is a body of work on the synthesis of bidirectional programs [14, 13, 15, 18]. These works all target the synthesis of well-behaved bidirectional programs written in a certain bidirectional language. The embedded well-behavedness properties of the underlying bidirectional languages ensure that the output programs are implicitly well-behaved.

Given data format specifications in the form of regular expressions and input-output examples, Optician [14, 13, 15] synthesizes bidirectional regular expressions written in Boomerang [8] using type-directed synthesis algorithms over the space of well-behaved combinators. The given examples of Optician are basically a pair of an updated view and an updated source, so the synthesized programs are relatively close to bijections. Meanwhile, the user-provided examples in our approach require more information about the original data, which helps to guide the synthesis toward finding common bidirectional transformations. In addition, the synthesis algorithms of Optician are limited to regular expressions and cannot be adapted to relations and query languages.

SynBit [18] synthesizes bidirectional programs written in HOBiT [33] from the corresponding unidirectional code of *get* plus a few input-output examples of *put*. It sketches the code of *put* with some holes based on the given code of *get*, then fills the holes by exploiting the properties of bidirectional programs. The base language - HOBiT, however, is a tree-oriented language that is not appropriate for transformations commonly performed on relations [7]. Although sharing the same thoughts of finding *put* when *get* is specified, our work is different from SynBit. SynBit ensures well-behavedness by taking advantage of the properties of the underlying bidirectional language, whereas we employ well-designed templates of well-behaved bidirectional programs on relations, but uses a non-bidirectional language. While SynBit sketches a whole program of *put* with some incomplete parts that are later filled in, we attempt to identify smaller, atomic parts and compose them to create a bigger program.

2.7.2 Relational Program Synthesis

The concept of relational program synthesis was first introduced in [19], aiming to discover one or more programs that collectively meet a relational specification. Relish [19] is a rare approach to solve this problem. It only allows relational specifications with functions $f_1, \ldots, f_n$ of the form $\forall \vec{x}.\ \phi(\vec{x})$ where ϕ is a quantifier-free formula with uninterpreted functions $f_1, \ldots, f_n$. The well-behavedness properties, GetPut and PutGet satisfy that form. It combines the counterexample-guided inductive synthesis (CEGIS) framework with a bottom-up search engine over a relational version space. The key idea is to build a relational version space in the form of a hierarchical tree automaton from a counterexample of a ground formula to encode all tuples of programs that satisfy the given specification. Adapting Relish to our problem is challenging due to the complex structures of relations and tabular examples, which cause the automata space to grow dramatically. Instead of building a relational version space, we use well-behaved templates to keep the relational specification.

2.7.3 ProSynth

ProSynth [20] is a state-of-the-art synthesis tool for synthesizing a Datalog program P, which is consistent with one input–output example E on relations. It supports a synthesis procedure "*ProSynth*$(\mathcal{S}, \mathcal{E}, \mathcal{P}_{all})$" where $\mathcal{E} = (I, O)$ is a tabular example containing input and output tuples, each matching an appropriate schema in $\mathcal{S}$, and $\mathcal{P}_{all} = \{r_1, \ldots, r_n\}$ is a fixed set of candidate rules. ProSynth reduces the synthesis to a rule selection problem, i.e., selecting $\mathcal{P}_s \subseteq \mathcal{P}_{all}$ such that $\mathcal{P}_s(I) \equiv O$.

ProSynth uses counterexample-guided inductive synthesis [34] with an SAT solver to suggest a selection of rules and a Datalog solver named Soufflé [35] to check whether the selection is consistent with the given example $\mathcal{E}$. Each rule r_i in P_{all} is associated with a Boolean variable b_{r_i} that describes whether r_i is selected or not. A synthesis constraint $\varphi = f(b_{r_1}, \ldots, b_{r_n})$ is kept up to date during the synthesis process. While φ is satisfiable, its variable b_{r_i} would be assigned to *True* or *False* by the SAT solver, which corresponds to whether or not r_i is selected in $\mathcal{P}_s$. Soufflé evaluates the selection on the given input tuples I, checks the result with the expected output tuples O. If a match occurs, the current P_s will be returned immediately. Otherwise, φ will be updated with some new constraints. If the SAT solver reports that φ is unsatisfiable, ProSynth returns *None* for no solution.

The original ProSynth only returns a solution if one exists. We later will slightly adapt it into "*ProSynth*$^+(\mathcal{S}, \mathcal{E}, \mathcal{P}_{all}, \varphi)$" that can return a pair of (P_s, φ) for further purposes like searching other programs. We will use ProSynth$^+$ as the name for both the adapted algorithm and the adapted tool.

One of the most difficult challenges in efficiently using ProSynth is the preparation of a "good" $\mathcal{P}_{all}$ for a specific task. In fact, experiments against ProSynth in [20] use two approaches to generating candidate rules: (1) instantiating *meta-rules* or *templates* (e.g., $H_0(x, z)\ :-\ H_1(x, y)\ ,\ H_2(y, z)$. where each H_i is a hole that could be replaced by a relation name) that describe common rules in the task, and (2) enumerating all sequences of length $k \leq 3$ of literals in the body of meta-rules. $\mathcal{P}_{all}$ could be chosen randomly with a fixed size if the number of generated rules is too large. The experiments only prepare candidate rules with no constants from predefined templates along with the names and schemas of relations.

ProSynth cannot trivially and directly synthesize a well-behaved pair of (get, put) on relations. Templates for a practical query *get* are quite familiar and well known, but such things for a practical view update *put* are rather vague. Ideally, by considerable efforts for preparing candidates from the common space of Datalog rules, we can use ProSynth to separately synthesize a *get* and a *put*. However, this approach cannot guarantee the well-behavedness between the *get* and the *put*, since there is no clear relationship between the search spaces (candidate rules) of the two programs.

Example 2.3. To specifically observe the challenge faced by ProSynth, let us consider the problem of synthesizing a well-behaved bidirectional program (get, put) satisfying the example given in Table 1.2. With two examples of *get* (Table 1.2a $\longrightarrow$ Table 1.2b and Table 1.2a' $\longrightarrow$ Table 1.2b') and one example of *put* ((Table 1.2a, Table 1.2b') $\longrightarrow$ Table 1.2a'), ProSynth may be adapted independently to synthesize a $get = get_i = \{r^i_1\}$ and a $put = put_i = \{r^i_2\}$ where

$$\begin{array}{lll} r^i_1 & V(v_0, v_1, v_2) & :-\ S(v_0, v_1, v_2, v_3)\ ,\ v_2 = \text{“}T\text{”}. \\ r^i_2 & S'(v_0, v_1, v_2, v_3) & :-\ S(v_0, \_, v_2, v_3)\ ,\ V'(v_0, v_1, \_). \end{array}$$

Unfortunately, (get_i, put_i) is not well-behaved. Indeed, for instance, if V' contains tuple $\langle 3, b_2, T \rangle$, then $put_i(S, V')$ will evaluate S' excluding any tuples with an A-value of 3. As a consequence, we cannot recover the new V' by applying get_i on the newly evaluated S', which violates one of the well-behavedness properties. ▲

Example 2.4. To demonstrate another failure of ProSynth in finding a well-behaved pair (get, put), we utilize a concrete example shown in Table 2.1, where S_1 and S_2 are sources, V is a view, and all relations share the same schema of (X, Y, Z).

From two examples of *get* (i.e., {Table 2.1a, Table 2.1b} $\longrightarrow$ Table 2.1c and {Table 2.1a', Table 2.1b'} $\longrightarrow$ Table 2.1c'), and one example of *put* (i.e., ({Table 2.1a, Table 2.1b}, Table 2.1c') $\longrightarrow$ {Table 2.1a', Table 2.1b'}), ProSynth can independently find a $get = \{r^g_1\}$ where

$$r^g_1 \quad V(x, y, z)\ :-\ S_1(x, y, \_)\ ,\ S_2(\_, y, z).$$

Table 2.1: An example of (get, put) provided to ProSynth

(2.1a) S_1

X	Y	Z
x1	y1	z1

(2.1b) S_2

X	Y	Z
x2	y1	z1

(2.1c) V

X	Y	Z
x1	y1	z1

(2.1a') S_1'

X	Y	Z
x1	y1	z1
x2	*y2*	*z1*

(2.1b') S_2'

X	Y	Z
x2	y1	z1
x2	*y2*	*z2*

(2.1c') V'

X	Y	Z
x1	y1	z1
x2	*y2*	*z2*

and a $put = \{r_1^p, r_2^p, r_3^p, r_4^p\}$ where

$$\begin{array}{lll}
r_1^p & S_1'(x,y,z) & :- \; V'(x,y,z)\;,\; S_1(x,y,z). \\
r_2^p & S_1'(x,y,z) & :- \; V'(x,y,\_)\;,\; S_2(x,\_,z). \\
r_3^p & S_2'(x,y,z) & :- \; S_2(x,y,z). \\
r_4^p & S_2'(x,y,z) & :- \; V'(x,y,z)\;,\; \neg\, S_1(x,y,z).
\end{array}$$

These two programs are not well-behaved because they violate the PutGet law: If we insert a new tuple $\langle x0, y0, z0\rangle$ to V' and run the put, then $S_1' = \{\langle x1, y1, z1\rangle, \langle x2, y2, z1\rangle\}$ and $S_2' = \{\langle x2, y1, z1\rangle, \langle x2, y2, z2\rangle, \langle x0, y0, z0\rangle\}$. However, a run of the get over the new S_1' and S_2' does not produce any $\langle x0, y0, z0\rangle$ in V'. ▲

2.7.4 Template-Based Synthesis

Templates are commonly used to guide the search in program synthesis [36, 37, 38, 20]. Zaatar [37] encodes templates as SMT formulas whose solutions produce the expected program. ALPS [38] and ProSynth [20] search for the target program as a subset of templates. Template-based synthesizers depend greatly on the quantity and quality of templates. In this thesis, we provide a set of templates for synthesizing atomic view update programs put_as given atomic queries get_as. The provided templates encode the existing minimal-effect view update strategies and the constraints and effects imposed by FDs.

2.7.5 Datalog Synthesis

There has been significant work on automatically synthesizing Datalog programs from examples [38, 39, 20, 40]. ALPS [38] adopts a bidirectional search strategy with top-down and bottom-up refinement operators over the syntax of Datalog programs to traverse the space of possible programs efficiently. DIFFLOG [39] and PROSYNTH [20] perform the synthesis by solving the equivalent rule selection problems from the given candidate rules. While DIFFLOG minimizes the difference between the weighted set of candidate rules and the reference output using numerical optimization, PROSYNTH uses query provenance to scale the CEGIS procedure and employs an SAT solver for constraint solving. GENSYNTH [40] learns Datalog programs from examples without requiring any templates, by introducing an evolutionary search strategy that mutates candidate programs and evaluates their fitness on examples using a Datalog solver. While we aim at the synthesis of well-behaved bidirectional programs (get, put) on relations, those works only target the synthesis of unidirectional programs. They cannot guarantee the well-behavedness of two independently synthesized programs, *get* and *put*. We later would adapt PROSYNTH as the unidirectional synthesizer inside our systems since PROSYNTH proved to be more effective than ALPS and DIFFLOG [20]. We also observed that the well-behaved templates for updating the atomic views are small-sized and could be categorizable, so we did not consider any dedicated template-free algorithms like GENSYNTH.

2.7.6 Query Synthesis

Many studies have been proposed to synthesize relational queries from input–output tables [41, 42, 38, 43, 44, 20, 45, 40]. SQLSYNTHESIZER [41] employed an adaptive decision tree algorithm to build suitable predicates in the `where` clauses of SQL queries. SQLSOL [43] used an off-the-shelf modern SMT solver to construct a whole query by encoding SQL components and tables into logic constraints. SCYTHE [42] enumerated abstract queries in a bottom-up manner and instantiated each query by encoding tables in bit vectors. PATSQL [45] synthesized a SQL query by employing a form of constraint and its top-down propagation mechanism for efficient sketch completion. While SQLSYNTHESIZER [41], SQLSOL [43], SCYTHE [42] and PATSQL [45] specialize in synthesizing SQL queries, ALPS [38], PROSYNTH [20] and GENSYNTH[40] target Datalog query synthesis. Alps and ProSynth are template-guided synthesizers, whereas GenSynth synthesizes programs without templates. These synthesizers cannot directly and efficiently synthesize well-behaved bidirectional programs on relations and cannot work successfully with examples containing internal FDs.

In fact, the primary limitation of current approaches to the synthesis of bidirectional programs on relations is their inability to effectively handle the complexity of relations and query languages, while also maintaining the desired well-behaved properties of bidirectional programs. They also cannot directly cope with examples having internal dependencies like functional dependencies.

In the next chapter, we will present a novel approach to synthesize well-behaved bidirectional programs on relations from examples with functional dependencies.

3 A Proposed Approach to Synthesizing Bidirectional Programs on Relations

Existing synthesizers for bidirectional programs do not work on the complex domains of relations and query languages. Other example-based synthesizers are only unidirectional and cannot guarantee the well-behavedness of two programs *get* and *put*. They also cannot directly cope with tables containing internal functional dependencies.

In this chapter, after we define our target synthesis problem of bidirectional programs on relations from examples with functional dependencies in Section 3.1, we propose a novel approach to solving that problem in Section 3.2, and describe a high-level algorithm corresponding to the proposed approach in Section 3.3.

3.1 Problem Definition

Definition 3.1 (Synthesis problem of bidirectional programs on relations)**.** *Given a specification* $(\mathcal{S}, \mathcal{E})$ *where*

- $\mathcal{S} = (\mathbb{S}, \mathbb{V})$ *where*
  - $\mathbb{S} = \{\mathbb{S}_1, \ldots, \mathbb{S}_n\}$ *includes n schemas of source relations* $S_1, \ldots, S_n$ *correspondingly associated with n sets of functional dependencies* $\mathcal{F}_{S_1}, \ldots, \mathcal{F}_{S_n}$,
  - $\mathbb{V}$ *is a schema of a view V*
- $\mathcal{E} = (T_{source}, T_{view}, T_{source'}, T_{view'})$ *is an example where*
  - T_{source} *and* $T_{source'}$ *include n original source tables and n updated source tables, respectively, each agreeing with an appropriate schema in* $\mathbb{S}$,
  - T_{view} *and* $T_{view'}$ *include one original view table and one updated view table, respectively, each agreeing with schema* $\mathbb{V}$,

the problem is to synthesize a pair (get, put) *satisfying the example* $\mathcal{E}$ *and two well-behavedness laws* GetPut *and* PutGet*, which is denoted as*

$$\textsc{Prob}(schema = \mathcal{S}, example = \mathcal{E})^{1}.$$

$\blacksquare$

Example 3.2. Let us consider

$\mathcal{S}_1 = ($
  $\mathbb{S} = \{$
  `staffs(sid:SID,name:NAME,city:CITY,active:ACTIVE,`$\mathcal{F}_{\texttt{staffs}} = \emptyset$`),`
  `customers(cid:CID,name:NAME,city:CITY,`$\mathcal{F}_{\texttt{customers}} = \emptyset$`)`
  $\},$
  $\mathbb{V} =$`tokyoac(name:NAME),`
$)$

consisting of the schemas of the sources `staffs` and `customers`, as well as the schema of the view `tokyoac`, all of which were mentioned in Section 1.2. There are no functional dependencies associated with the sources `staffs` and `customers`.

[1] Keyword-arguments (e.g., *schema* =, *example* =) are used to clearly identify arguments of a structure. They can be omitted for brevity (e.g., $\textsc{Prob}(\mathcal{S}, \mathcal{E})$).

We sometimes use a relation name as a shorthand for a schema for brevity, e.g., $\mathcal{S}_1 = (\{\texttt{staffs}, \texttt{customers}\}, \texttt{tokyoac})$.

Table 1.1 provides an example

$$\mathcal{E}_1 = (T_{source} = \{1.1a, 1.1b\}, T_{view} = \{1.1c\}, T_{source'} = \{1.1a', 1.1b'\}, T_{view'} = \{1.1c'\})$$

Problem $\textsc{Prob}(\mathcal{S}_1, \mathcal{E}_1)$ is to synthesize a well-behaved pair of NR-Datalog* programs (get, put) consistent with $\mathcal{E}_1$. ▲

Example 3.3. Let us see another problem in which nonempty functional dependencies are given. Suppose that we provide a specification $(\mathcal{S}_2, \mathcal{E}_2)$ where

$$\begin{aligned} \mathcal{S}_2 = (& \\ & \mathbb{S} = \{S(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, D : \mathbb{D}, \mathcal{F}_S = \{A \to B, A \to D\})\}, \\ & \mathbb{V} = V(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}), \\ ) & \end{aligned}$$

$$\mathcal{E}_2 = (T_{source} = \{1.2a\}, T_{view} = \{1.2b\}, T_{source'} = \{1.2a'\}, T_{view'} = \{1.2b'\})$$

In this specification, $\mathcal{S}_2$ consists of a schema of the source S associated with a nonempty set of FDs $\mathcal{F}_S$, and a schema of the view V, while $\mathcal{E}_2$ consists of original and updated tables listed in Table 1.2.

Problem $\textsc{Prob}(\mathcal{S}_2, \mathcal{E}_2)$ is to synthesize a well-behaved pair of NR-Datalog* programs (get, put) consistent with $\mathcal{E}_2$. ▲

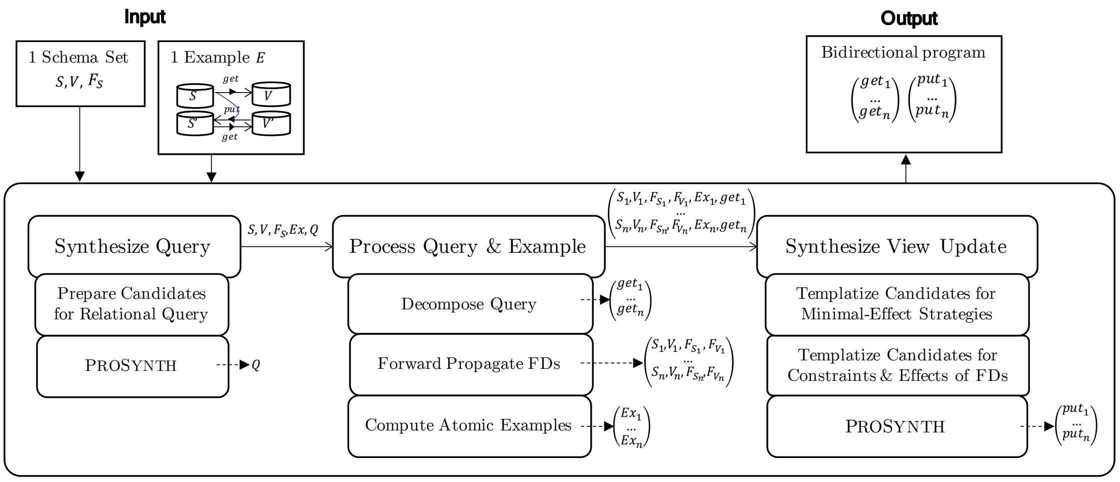
Given a specification $(\mathcal{S}, \mathcal{E})$, from the example $\mathcal{E}$, we have two examples of *get*

$$\begin{aligned} \mathcal{E}_{get} &:= T_{source} \to T_{view} \\ \mathcal{E}'_{get} &:= T_{source'} \to T_{view'} \end{aligned}$$

and one example of *put*

$$\mathcal{E}_{put} := (T_{source}, T_{view'}) \to T_{source'}$$

The example $\mathcal{E}'_{get}$ is a consequence of the $\textsc{PutGet}$ law which says that the updated view could be recovered by applying *get* on the updated source.



Input
1 Schema Set S, V, F_S
1 Example E
get
put
get
Output
Bidirectional program
$\begin{pmatrix} get_1 \\ \dots \\ get_n \end{pmatrix} \begin{pmatrix} put_1 \\ \dots \\ put_n \end{pmatrix}$
Synthesize Query
Prepare Candidates for Relational Query
PROSYNTH
Q
S, V, F_S, Ex, Q
Process Query & Example
Decompose Query
$\begin{pmatrix} get_1 \\ \dots \\ get_n \end{pmatrix}$
Forward Propagate FDs
$\begin{pmatrix} S_1, V_1, F_{S_1}, F_{V_1} \\ \dots \\ S_n, V_n, F_{S_n}, F_{V_n} \end{pmatrix}$
Compute Atomic Examples
$\begin{pmatrix} Ex_1 \\ \dots \\ Ex_n \end{pmatrix}$
$\begin{pmatrix} S_1, V_1, F_{S_1}, F_{V_1}, Ex_1, get_1 \\ \dots \\ S_n, V_n, F_{S_n}, F_{V_n}, Ex_n, get_n \end{pmatrix}$
Synthesize View Update
Templatize Candidates for Minimal-Effect Strategies
Templatize Candidates for Constraints & Effects of FDs
PROSYNTH
$\begin{pmatrix} put_1 \\ \dots \\ put_n \end{pmatrix}$

Figure 3.1: A proposed approach to synthesizing bidirectional programs on relations

Example $\mathcal{E}$ can describe one or many small update strategies, each involving the deletion or insertion of a tuple. If a user provides a specification with many examples, it is possible to reasonably modify values of tuples in these examples, and merge the new examples into one "big" example called $\mathcal{E}$. In this work, we simplify the problem by considering only such a "big" example.

3.2 Approach

Following the introduction in Section 1.2, the main keywords behind our proposed approach to synthesizing a well-behaved bidirectional program (get, put) on relations are *"decomposition"*, *"composition"*, and *"templates"*.

On relations, we know a *get* is a query that is possibly *decomposable*. If we have a set of atomic queries get_a and we synthesize the corresponding atomic view update put_a such that (get_a, put_a) is well-behaved, then we can *compose* all pairs to obtain a bigger program that is also well-behaved. Given get_a, to synthesize put_a such that (get_a, put_a) is well-behaved, we design *templates* that encode both existing minimal-effect view update strategies for atomic queries and the constraints along with effects of functional dependencies. The well-behavedness is embedded in these templates. With the templates, we can generate candidate rules for put_a and adapt the state-of-the-art example-and-template-based synthesizer ProSynth to find put_a.

Figure 3.1 show our proposed approach more concretely. Given a specification $(\mathcal{S}, \mathcal{E})$, we can easily separate out the input-output examples of *get* and *put*. Our approach consists of three steps.

1. We synthesize a query Q satisfying the examples of *get* by adapting ProSynth over the predefined templates for relational queries. These templates are extensively provided in the database community.

2. We process the obtained query and the given example as follows.

   - We decompose the query Q to a set of atomic queries $\{get_1, \ldots, get_n\}$.
   - We forward-propagate FDs from the source through intermediate relations that appear in $\{get_1, \ldots, get_n\}$ to the view, so that we are aware of FDs F_{S_a} and F_{V_a} respectively associated with relations S_a and V_a, both corresponding to get_a.
   - We compute atomic examples Ex_a for (get_a, put_a) by apply $\{get_1, \ldots, get_n\}$ to the original and updated source tables

3. We synthesize a view update satisfying the example of *put* as follows.

   - We design templates that encode minimal-effect view update strategies for atomic queries (based on the works of Keller [21] and Larson [23]), to generate candidate rules of put_a given get_a.
   - We templatize more with the constraints and effects of FDs (based on the work of Bohannon [7]), to enrich the candidate rules of put_a given get_a.
   - We adapt ProSynth to synthesize put_a from the input-output example in Ex_a and the generated rules above. We compose programs put_as by merging their rules.

The well-behavedness of a synthesized bidirectional program is achieved by

1. the well-behavedness of each atomic pair (get_a, put_a) (inherited from well-behaved strategies in the works of Keller [21], Larson [23] and Bohannon [7]).

2. the well-behavedness of the composition of bidirectional programs [1].

Algorithm 1: SynthB: Synthesizing bidirectional programs on relations

```
Input: specification $(\mathcal{S}, \mathcal{E})$
Output: a bidirectional program $(C_g, C_p)$ or $None$
procedure SynthB ($\mathcal{S}, \mathcal{E}$):
    $\mathcal{P}_{get} \leftarrow$ PrepareGetCand($\mathcal{S}, \mathcal{E}$)
    $\mathcal{E}_{get}, \mathcal{E}'_{get} \leftarrow$ examples of $get$ based on $\mathcal{E}$
    $\varphi \leftarrow True$
    while $True$ do
        $g,\ \varphi \leftarrow$ ProSynth$^+(\mathcal{S}, \mathcal{E}_{get}, \mathcal{P}_{get}, \varphi)$
        if $g = None$ then return $None$
        if $g$ is not consistent with $\mathcal{E}'_{get}$ then
            $\varphi \leftarrow$ Strengthen$(\varphi, g, \mathcal{P}_{get})$ ; continue
        $C_g, \mathcal{S}_f \leftarrow$ Decompose$(g, \mathcal{S})$
        $\mathcal{S}_f \leftarrow$ ForwardPropagateFDs$(C_g, \mathcal{S}_f)$
        $\mathcal{E}_f \leftarrow$ Eval$(C_g, \mathcal{E})$
        for $r^a \in C_g$ do
            $\mathcal{P}^{r^a}_{put} \leftarrow$ PreparePutCandMEVUS$(\mathcal{S}_f, \mathcal{E}_f, r^a)$
            $\mathcal{P}^{r^a}_{put} \leftarrow \mathcal{P}^{r^a}_{put} \cup$ PreparePutCandCEFDs$(\mathcal{S}_f, \mathcal{E}_f, r^a)$
        $\mathcal{P}_{put} \leftarrow$ Combine$(\{\mathcal{P}^{r}_{put} \mid r \in C_g\})$
        $C_p,\ \_ \leftarrow$ ProSynth$^+(\mathcal{S}_f, \mathcal{E}_f, \mathcal{P}_{put}, True)$
        if $C_p \neq None$ then return $(C_g, C_p)$
        else $\varphi \leftarrow$ Strengthen$(\varphi, g, \mathcal{P}_{get})$
```

3.3 High-Level Algorithm

Algorithm 1 gives a high-level description of the algorithm solving problem Prob$(\mathcal{S}, \mathcal{E})$.

The procedure SynthB takes as input a specification $(\mathcal{S}, \mathcal{E})$, and returns as output either a well-behaved pair (C_g, C_p) consistent with $\mathcal{E}$ or *None* if no such a pair exists.

SynthB would internally use an adaptation of ProSynth called ProSynth$^+$ to unidirectionally synthesize Datalog programs from input-output examples and a fixed set of candidate rules. While the original "ProSynth$(\mathcal{S}, \mathcal{E}, \mathcal{P}_{all})$" (Section 2.7.3) only returns a solution P_s if one exists, "*ProSynth*$^+(\mathcal{S}, \mathcal{E}, \mathcal{P}_{all}, \varphi)$" can return a pair of (P_s, φ) for searching other programs, where φ is the synthesis constraints.

At the beginning of the synthesis, SynthB prepares a fixed set $\mathcal{P}_{get} = \{r_1, \ldots, r_n\}$ consisting of candidate rules for query *get* by invoking PrepareGetCand (line 2) (or by asking the user to provide such a set).

Next, it extracts two examples of get, $\mathcal{E}_{get}$ and $\mathcal{E}'_{get}$, from $\mathcal{E}$ (line 3), and initializes a synthesis constraint $\varphi = f(b_{r_1}, \ldots, b_{r_n})$ as $True$ (line 4).

Then, the loop in lines 5-19 iteratively:

1. synthesizes a query g ($g \subseteq \mathcal{P}_{get}$) consistent with $\mathcal{E}_{get}$, and stores the state of synthesis constraint φ (ProSynth$^+$ - line 6);

2. immediately returns $None$ if no such a g exists (line 7);

3. if g is not consistent with $\mathcal{E}'_{get}$ (which serves only to check the correctness of g), strengthens φ then goes to a new loop (lines 8-9), otherwise, decomposes g into an equivalent query C_g consisting of only atomic queries that are specified over a schema $\mathcal{S}_f$ (Decompose - line 10);

4. forward-propagates FDs from the source to the view over C_g to compute the FDs of all relations in $\mathcal{S}_f$ (ForwardPropagateFDs - line 11);

5. evaluates C_g one by one on T_{source} and $T_{source'}$ in $\mathcal{E}$ to obtain full examples (Eval - line 12);

6. prepares candidate rules for atomic view update programs corresponding to the atomic queries, based on templates encoding minimal-effect view update strategies (PreparePutCandMEVUS - line 14) and templates encoding the constraints along with effects of FDs (PreparePutCandCEFDs - line 15), then combines them into a set $\mathcal{P}_{put}$ including candidate rules of put (Combine - line 16);

7. synthesizes a put program C_p that, when combined with C_g, can form a well-behaved bidirectional program (ProSynth$^+$ - lines 17-18) (equivalent to simultaneously synthesizing atomic view update programs and automatically combining results);

8. strengthens φ if no put is found (Strengthen - line 19) and goes to a new loop.

Algorithm 1 will terminate if

1. no more query g can be synthesized (line 7);

2. a query g is synthesized and decomposed to C_g and the corresponding C_p can be synthesized (line 18).

We skip the details of PrepareGetCand since the preparation of candidates for relational queries is well-discussed in [38, 20].

Procedure Eval basically calls the Datalog solver Soufflé to evaluate an NR-Datalog* program on specific inputs.

Procedure Strengthen$(\varphi, g, \mathcal{P}_{get})$ boosts the synthesis constraint φ as

$$\varphi \leftarrow \varphi \ \wedge \ \neg \, (\bigwedge_{r_i \in g} b_{r_i} \wedge \bigwedge_{r_j \in \mathcal{P}_{get} - g} \neg b_{r_j})$$

to ignore the found query g ($g \subseteq \mathcal{P}_{get}$) in the next loops in which the SAT solver inside ProSynth$^+(\mathcal{S}, \mathcal{E}_{get}, \mathcal{P}_{get}, \varphi)$ (line 6) is unsatisfiable for the rule selection corresponding to g.

In the next chapters, we will explain more about the remaining procedures. Chapter 4 presents Decompose and ForwardPropagateFDs for decomposing queries, propagating FDs and dividing synthesis into sub-synthesis. Chapter 5 presents PreparePutCandMEVUS/Combine for solving/combining the sub-synthesis with only templates of minimal-effect view update strategies. Chapter 6 presents PreparePutCandCEFDs for solving the sub-synthesis by enriching the templates with the ones encoding the constraints and effects of FDs.

4 Processing Over Synthesized Queries

In this chapter, we focus more on two procedures after a query is synthesized: the query decomposition in NR-Datalog* and the forward propagation of functional dependencies. These procedures are meaningful in reducing the synthesis problem to smaller problems in which the relevant relations may be specifically constrained by FDs.

The following sections are organized as follows. Section 4.1 discusses decomposing queries in NR-Datalog*. Section 4.2 covers forward-propagating FDs from the source to the view over a set of atomic queries. Section 4.3 summarizes this chapter.

4.1 Decomposing Queries

If a query *get* is synthesized, by decomposing it into a set of atomic queries get_as that each defines an atomic view, we can reduce the synthesis of (get, put) to the synthesis of (get_a, put_a)s, which, as you will see later, can be automatically synthesized by our approach. In this section, we focus on describing the query decomposition to obtain atomic queries in NR-Datalog*, and at the end, we will provide an example showing the sub-synthesis problems that are reduced after the decomposition.

While query decomposition and atomic queries are recognized as standard concepts in relational algebra, their discussions in the context of Datalog have been limited. We aim to bridge this gap by exploring these aspects within Datalog. Specifically, we formulate atomic queries in Datalog and demonstrate how a complex Datalog query, satisfying *decomposable conditions* (to be discussed later), can be decomposed into these atomic queries. The choice of Datalog over relation algebra is also driven by our awareness of the framework Birds [29], which utilizes Datalog to write bidirectional programs, motivating our objective to establish a solid foundation to construct both query and view update programs in Datalog.

When considering atomic queries, our focus lies on those for which well-behaved view update strategies exist and have been extensively studied. The theoretical research described in [21, 22, 23] investigates view update strategies for the selection (σ), projection (π), natural join ($\bowtie$), union ($\cup$), intersection ($\cap$) and set difference ($\setminus$) queries. Furthermore, the practical research outlined in [16] enables the construction of view update programs for a rename (ρ) query and a specialized cross-product ($\times$) query. From both theory and practice, we carefully select eight essential atomic queries and formulate them as atomic NR-Datalog* rules in Figure 4.1. We say that an atomic rule r is of type α if r is an *α-rule* where $\alpha \in \{\rho, \cup, \setminus, \cap, \sigma, \pi, \bowtie, \times\}$.

Due to limitations in [21, 22, 23, 16], the atomic rules have restricted forms:

- Each constraint in a *σ-rule* is a comparison between a variable and a constant.
- A *π-rule* defines a proper projection.
- A *$\times$-rule* describes a cross-product between a relation S and a relation $E^{1\times m}$ which can be expressed by a list of equality constraints.

Despite their restrictions, the selected atomic rules remain valuable for composing a wide range of practical queries (e.g. in [16]).

$\underline{\rho\text{-}rule}$	$V(\vec{x})\ :-\ S(p(\vec{x})).$ where p is a permutation function
$\underline{\cup\text{-}rule}$	$V(\vec{x})\ :-\ S_1(\vec{x})\ ;\ S_2(\vec{x}).$
$\underline{\setminus\text{-}rule}$	$V(\vec{x})\ :-\ S_1(\vec{x})\ ,\ \neg\ S_2(\vec{x}).$
$\underline{\cap\text{-}rule}$	$V(\vec{x})\ :-\ S_1(\vec{x})\ ,\ S_2(\vec{x}).$
$\underline{\sigma\text{-}rule}$	$V(\vec{x})\ :-\ S(\vec{x}),\ v_1 \oplus_1 c_1, \ldots,\ v_m \oplus_m c_m.$ where $m > 0$, $\{v_1, \ldots, v_m\} \subseteq set(\vec{x})$
$\underline{\pi\text{-}rule}$	$V(\vec{x})\ :-\ S(\vec{y}).$ where $set(\vec{x}) \subsetneq set(\vec{y})$
$\underline{\bowtie\text{-}rule}$	$V(\vec{x})\ :-\ S_1(\vec{y_1})\ ,\ S_2(\vec{y_2}).$ where $set(\vec{y_1}) \cup set(\vec{y_2}) = set(\vec{x})$ and $set(\vec{y_1}) \cap set(\vec{y_2}) \neq \emptyset$
$\underline{\times\text{-}rule}$	$V(\vec{x})\ :-\ S(\vec{y})\ ,\ v_1 = c_1\ , \ldots,\ v_m = c_m.$ where $m > 0$, $set(\vec{y}) \cup \{v_1, \ldots, v_m\} = set(\vec{x})$ and $set(\vec{y}) \cap \{v_1, \ldots, v_m\} = \emptyset$

Note: $\vec{x}$, $\vec{y}$, $\vec{y_1}$ and $\vec{y_2}$ are tuples, each containing only variables.
$set(\vec{x})$ converts a tuple $\vec{x}$ to a set.
v_i is a variable, c_i is a constant, $\oplus_i \in \{=, \neq, >, <, \leq, \geq\}$.

Figure 4.1: Atomic rules/queries in NR-Datalog*

Example 4.1. The following eight rules are instances of the atomic rules in Figure 4.1:

$\rho - rule$	$X(a,b,c,d)$	$:-$	$Y(b,d,a,c).$
$\cup - rule$	$X(a,b,c,d)$	$:-$	$Y(a,b,c,d)\ ;\ Z(a,b,c,d).$
$\setminus - rule$	$X(a,b,c,d)$	$:-$	$Y(a,b,c,d)\ ,\ \neg\ Z(a,b,c,d).$
$\cap - rule$	$X(a,b,c,d)$	$:-$	$Y(a,b,c,d)\ ,\ Z(a,b,c,d).$
$\sigma - rule$	$X(a,b,c,d)$	$:-$	$Y(a,b,c,d)\ ,\ d \neq \texttt{"0"}\ ,\ d \neq \texttt{"1"}.$
$\pi - rule$	$X(a,b,c,d)$	$:-$	$Y(a,b,e,c,d,f).$
$\bowtie -rule$	$X(a,b,c,d)$	$:-$	$Y(a,d,b)\ ,\ Z(b,c).$
$\times - rule$	$X(a,b,c,d)$	$:-$	$Y(a,d,b)\ ,\ c = \texttt{"0"}.$

where X, Y, Z are relation symbols, a, b, c, d, e, f are variables.

In the $\rho - rule$, X is a view, and Y is a source. In the $\cup - rule$, X is a view, and Y and Z are sources. Similarly, we can specify the source and the view for the remaining rules. ▲

Example 4.2. The following NR-Datalog* rules are also atomic

```
r_1^a   M₁(i,n,c,a) :- staffs(i,n,c,a) , c="Tokyo" , a="1".
r_2^a   M₂(i,n,c) :- customers(i,n,c) , c="Tokyo".
r_3^a   tokyoac₀(n) :- M₁(i,n,c,a).
r_4^a   tokyoac₁(n) :- M₂(i,n,c).
r_5^a   tokyoac(n) :- tokyoac₀(n) ; tokyoac₁(n).
```

where r_1^a and r_2^a are *σ-rules*, r_3^a and r_4^a *π-rules*, and r_5^a a *$\cup$-rule.* ▲

A query g is written in NR-Datalog* as a set of rules. To successfully decompose g into atomic queries, we require that each rule r in g needs to satisfy *decomposable conditions* as follows:

1. r satisfies safety conditions [27];

2. there is at least one positive literal in the body of r;

3. variables in a literal of r are different;

4. all positive literals in the body of r can be joined without cross-product

The first two conditions are essential for query practicality. The third condition is necessary to prevent implicit comparisons between two attributes of a relation, while the last condition is crucial for avoiding non-specialized cross-products.

We next describe the behavior of DECOMPOSE$(g, \mathcal{S})$, which takes a query g defined over a schema set $\mathcal{S}$ as input, performs *rewrite laws* (later denoted by $\mathcal{L}_{...}$) to decompose g, and returns a set C_g of atomic rules over a new schema set $\mathcal{S}_f$. After presenting the flow inside DECOMPOSE, we will briefly explain an example shown in Figure 4.2, which demonstrates the use of rewrite laws.

If g satisfies *decomposable conditions*, the flow to decompose g is a sequence

$$g \xrightarrow{\mathcal{L}_1} T_1 \xrightarrow{\mathcal{L}_2} T_2 \xrightarrow{\mathcal{L}_3} T_3 \xrightarrow{\mathcal{L}_4} C_g \tag{F1}$$

using the following laws to rewrite sets of rules:

$\mathcal{L}_1$ rewriting g to T_1 by converting all rules in g to normal form;

$\mathcal{L}_2$ rewriting T_1 to T_2 by repeatedly decomposing each normal rule in T_1 to a set of atomic rules and then merging these sets;

$\mathcal{L}_3$ rewriting T_2 to T_3 by updating rules with the same head in T_2 and pushing out unions ($\sim$ extracting $\cup$-*rules*);

$\mathcal{L}_4$ rewriting T_3 to C_g by renaming relations whose names occur more than once in the bodies of rules in T_3.

When processing $\mathcal{L}_2$, if a normal rule r is atomic, we can move on to consider another rule. Otherwise, the flow to decompose a normal, non-atomic rule r is a sequence

$$r \xrightarrow[\hookrightarrow^*\sigma]{\mathcal{L}_{2.1}} r_1 \xrightarrow[\hookrightarrow^*\pi]{\mathcal{L}_{2.2}} r_2 \xrightarrow[\hookrightarrow^*\bowtie,\cap]{\mathcal{L}_{2.3}} r_3 \xrightarrow[\hookrightarrow^*\pi]{\mathcal{L}_{2.4}} r_4 \xrightarrow[\hookrightarrow^*\ltimes,\setminus]{\mathcal{L}_{2.5}} r_5 \xrightarrow[\hookrightarrow^*\pi]{\mathcal{L}_{2.6}} r_6 \tag{F2}$$

using the following sub-laws, each of which extracts zero or more ($\hookrightarrow^*$) atomic rules that define possible new intermediate states, updates the remaining form of r with these states to form r_i, and progressively rewrites until the remainder r_i becomes atomic:

$\mathcal{L}_{2.1}$ pushing out selections ($\sim$ extracting σ-*rules*);

$\mathcal{L}_{2.2}$ pushing out projections before joins ($\sim$ extracting π-*rules*)

$\mathcal{L}_{2.3}$ rewriting multi-ary joins as binary joins and replacing binary joins over the same schema with intersections ($\sim$ extracting $\bowtie$- and $\cap$-*rules*);

$\mathcal{L}_{2.4}$ pushing out projections after joins ($\sim$ extracting π-*rules*);

$\mathcal{L}_{2.5}$ handling negative literals with left-anti-semi-joins ($\sim$ extracting $\ltimes$- and $\setminus$-*rules*);

$\mathcal{L}_{2.6}$ pushing out projections after joins ($\sim$ extracting π-*rules*);

In Figure 4.2, Decompose applies $\mathcal{L}_1$ to change one rule with two conjunctions in c2 into two normal rules in d2. Each rule in d2 is decomposed using $\mathcal{L}_2$. In the first use of $\mathcal{L}_2$, the non-atomic rule in c3 is decomposed to two σ-*rules* in d3 and another rule in c4 by $\mathcal{L}_{2.1}$. Decompose similarly performs the next sub-laws until finishing $\mathcal{L}_{2.6}$ in which the remaining rule in c9 is atomic. In the second use of $\mathcal{L}_2$, the rule in

	A	B	C	D
1	Laws		NR-Datalog*Rules	Extracted Rules
2	$\mathcal{L}_1$		$V(a,d) :- S_1(a,b,c,e), S_2(a),$ $S_3(a,c), \neg S_4(b),$ $b = \texttt{"T"}, d = \texttt{"0"}$ $;\ S_2(a), S_5(a,d).$	$V(a,d) :- S_1(a,b,c,e), S_2(a),$ $S_3(a,c), \neg S_4(b),$ $b = \texttt{"T"}, d = \texttt{"0"}.$ $V(a,d) :- S_2(a), S_5(a,d).$
3		$\mathcal{L}_{2.1}$	$V(a,d) :- S_1(a,b,c,e), S_2(a),$ $S_3(a,c), \neg S_4(b),$ $b = \texttt{"T"}, d = \texttt{"0"}.$	$M_0(a,b,c,e) :- S_1(a,b,c,e),$ $b = \texttt{"T"}.$ $M_1(b) :- S_4(b), b = \texttt{"T"}.$
4		$\mathcal{L}_{2.2}$	$V(a,d) :- M_0(a,b,c,e), S_2(a),$ $S_3(a,c), \neg M_1(b),$ $d = \texttt{"0"}.$	$M_2(a,b,c) :- M_0(a,b,c,e).$
5	$\mathcal{L}_2$	$\mathcal{L}_{2.3}$	$V(a,d) :- M_2(a,b,c), S_2(a),$ $S_3(a,c), \neg M_1(b),$ $d = \texttt{"0"}.$	$M_3(a,b,c) :- M_2(a,b,c), S_2(a).$ $M_4(a,b,c) :- M_3(a,b,c), S_3(a,c).$
6		$\mathcal{L}_{2.4}$	$V(a,d) :- M_4(a,b,c), \neg M_1(b),$ $d = \texttt{"0"}.$	$M_5(a,b) :- M_4(a,b,c).$
7		$\mathcal{L}_{2.5}$	$V(a,d) :- M_5(a,b), \neg M_1(b),$ $d = \texttt{"0"}.$	$M_6(a,b) :- M_5(a,b), M_1(b).$ $M_7(a,b) :- M_5(a,b), \neg M_6(a,b).$
8		$\mathcal{L}_{2.6}$	$V(a,d) :- M_7(a,b), d = \texttt{"0"}.$	$M_8(a) :- M_7(a,b).$
9			$V(a,d) :- M_8(a), d = \texttt{"0"}.$	
10	$\mathcal{L}_2$		$V(a,d) :- S_2(a), S_5(a,d).$	
11	$\mathcal{L}_3$		$V(a,d) :- M_8(a), d = \texttt{"0"}.$ $V(a,d) :- S_2(a), S_5(a,d).$	$V_0(a,d) :- M_8(a), d = \texttt{"0"}.$ $V_1(a,d) :- S_2(a), S_5(a,d).$ $V(a,d) :- V_0(a,d); V_1(a,d).$
12	$\mathcal{L}_4$		$M_3(a,b,c) :- M_2(a,b,c), S_2(a).$ $V_1(a,d) :- S_2(a), S_5(a,d).$	$M_3(a,b,c) :- M_2(a,b,c), S_2^0(a).$ $V_1(a,d) :- S_2^1(a), S_5(a,d).$ $S_2^0(a) :- S_2(a).$ $S_2^1(a) :- S_2(a).$

Figure 4.2: An application example of rewrite laws

C10 is already atomic, so no sub-laws need to be applied. A set of the atomic rules decomposed by $\mathcal{L}_2$ is further revised in sequence by $\mathcal{L}_3$ and $\mathcal{L}_4$, where rules in C11 and C12 are rewritten into rules in D11 and D12, respectively.

Example 4.3. Let $\text{HD}(r)$ and $\text{BD}(r)$ be the head and the body of a rule r. A more detailed explanation of Figure 4.2 is as follows.

- $\mathcal{L}_1$

  $$\begin{array}{lll} r & V(a,d) & :- S_1(a,b,c,e), S_2(a), S_3(a,c), \neg S_4(b), b = \texttt{"T"}, d = \texttt{"0"} \\ & & ;\ S_2(a), S_5(a,d). \end{array}$$

  Because $\text{BD}(r)$ has two conjunctions ($[conj_1, conj_2]$), r will be rewritten as two rules like $\text{HD}(r)\ :-\ conj_1$ and $\text{HD}(r)\ :-\ conj_2$. So, we obtain two following rules:

  $$\begin{array}{lll} r_1 & V(a,d) & :- S_1(a,b,c,e), S_2(a), S_3(a,c), \neg S_4(b), b = \texttt{"T"}, d = \texttt{"0"}. \\ r_2 & V(a,d) & :- S_2(a), S_5(a,d). \end{array}$$

- $\mathcal{L}_2$ While r_2 is atomic ($\bowtie -rule$), r_1 is non-atomic. We will perform a series of sub-laws to decompose r_1.

- $\mathcal{L}_{2.1}$ $\text{BD}(r_1)$ has two constraints $b = \texttt{"T"}$ and $d = \texttt{"0"}$. Variable b in constraint $b = \texttt{"T"}$ appears in some literals in $\text{BD}(r_1)$, which show an indication of $\sigma - rule$(s). Variable d in constraint $d = \texttt{"0"}$ doesn't appear in any literals in $\text{BD}(r_1)$ and just appear in $\text{HD}(r_1)$, which show an indication of $\times - rule$(s). For $\mathcal{L}_{2.1}$, we only extract $\sigma - rule$(s) if any. Two literals in $\text{BD}(r_1)$, $S_1(a,b,c,e)$ and $S_4(b)$, contain variable b, so we can obtain two following $\sigma - rules$ with new intermediate relations, M_0 and M_1:

  $$\begin{array}{llll} r_3 & M_0(a,b,c,e) & :- & S_1(a,b,c,e)\ ,\ b = \texttt{"T"}. \\ r_4 & M_1(b) & :- & S_4(b)\ ,\ b = \texttt{"T"}. \end{array}$$

  Then the head of r_3 and r_4 will respectively replace literals $S_1(a,b,c,e)$ and $S_4(b)$ in r_1, and the constraint $b = \texttt{"T"}$ will be removed from r_1. So r_1 is updated to r_5 as follows:

  $$r_5 \quad V(a,d)\ :- M_0(a,b,c,e), S_2(a), S_3(a,c), \neg M_1(b), d = \texttt{"0"}.$$

- $\mathcal{L}_{2.2}$ In $\textsc{Bd}(r_5)$, we found that variable e only appears once in $\textsc{Bd}(r_5)$ (only in literal $M_0(a,b,c,e)$), but doesn't appear in $\textsc{Hd}(r_5)$. So from r_5, we can extract a $\pi - rule$ with new intermediate relation M_2 as follows:

$$r_6 \quad M_2(a,b,c) \quad :- \quad M_0(a,b,c,e).$$

  Then the head of r_6 will replace literals $M_0(a,b,c,e)$ in r_5, so we have an updated remaining rule r_7 as below:

$$r_7 \quad V(a,d) \quad :- \quad M_2(a,b,c), S_2(a), S_3(a,c), \neg\, M_1(b), d = \texttt{"0"}$$

- $\mathcal{L}_{2.3}$ In $\textsc{Bd}(r_7)$, there are three positive literals, $M_2(a,b,c)$, $S_2(a)$ and $S_3(a,c)$, which could be joined together. But we only allow binary joins as atomic rules. To rewrite the multi-join into binary joins, we extract from r_7 two following $\bowtie -rules$ with new intermediate relations, M_3 and M_4:

$$\begin{array}{llll} r_8 & M_3(a,b,c) & :- & M_2(a,b,c)\ ,\ S_2(a). \\ r_9 & M_4(a,b,c) & :- & M_3(a,b,c)\ ,\ S_3(a,c). \end{array}$$

  $M_4(a,b,c)$ is similar to the results of multi-joins, so it will replace three positive literals in $\textsc{Bd}(r_7)$. The updated remaining rule r_{10} is as follows:

$$r_{10} \quad V(a,d) \quad :- \quad M_4(a,b,c), \neg\, M_1(b), d = \texttt{"0"}.$$

- $\mathcal{L}_{2.4}$ After joining, there may be some variables that could be projected, such as variable c in literal $M_4(a,b,c)$ in r_{10}. We perform such a projection by extract from r_{10} a following $\pi - rule$ with new intermediate relation M_5:

$$r_{11} \quad M_5(a,b) \quad :- \quad M_4(a,b,c).$$

  The head of r_{11} will replace literals $M_4(a,b,c)$ in r_{10}, so we have an updated remaining rule r_{12} as below:

$$r_{12} \quad V(a,d) \quad :- \quad M_5(a,b), \neg\, M_1(b), d = \texttt{"0"}.$$

- $\mathcal{L}_{2.5}$ We continue to decompose r_{12} with negation in its body by using left anti semi join to extract the following atomic rules with new intermediate relations, M_6 and M_7:

$$\begin{array}{llll} r_{13} & M_6(a,b) & :- & M_5(a,b)\ ,\ M_1(b). \\ r_{14} & M_7(a,b) & :- & M_5(a,b)\ ,\ \neg\ M_6(a,b). \end{array}$$

  The result of the left anti semi join, $M_7(a,b)$, will replace the literals of M_5 and M_1 in r_{12}, then we obtain an updated rule r_{15} as below:

$$r_{15} \quad V(a,d) \quad :- \quad M_7(a,b), d = \texttt{"0"}.$$

- $\mathcal{L}_{2.6}$ After joining, there may be some variables that could be projected, such as variable b in literal $M_7(a,b)$ in r_{15}. We perform such a projection by extract from r_{15} a following $\pi - rule$ with new intermediate relation M_8:

$$r_{16} \quad M_8(a) \quad :- \quad M_7(a,b).$$

  The updated remaining rule is as below:

$$r_{17} \quad V(a,d) \quad :- \quad M_8(a), d = \texttt{"0"}.$$

  r_{17} is a $\times - rule$.

- $\mathcal{L}_3$ Because r_{17} and r_2 have the same head but different bodies, we rewrite them with the necessary appearance of binary unions. Two heads of r_{17} and r_2 will be replaced with two new relations, e.g., V_0 and V_1, like r_{18} and r_{19} as below:

$$\begin{array}{llll} r_{18} & V_0(a,d) & :- & M_8(a), d = \texttt{"0"}. \\ r_{19} & V_1(a,d) & :- & S_2(a), S_5(a,d). \end{array}$$

  and we can have a union rule r_{20} as follows:

$$r_{20} \quad V(a,d) \quad :- \quad V_0(a,d)\ ;\ V_1(a,d).$$

- $\mathcal{L}_4$ Each of two rules r_8 and r_{19} has multiple sources, and S_2 appears in both $\textsc{Bd}(r_8)$

and $\textsc{Bd}(r_{19})$. We would like to create aliases of S_2, e.g., S_2^0 and S_2^1, to replace such appearances of S_2.

$$\begin{array}{llll} r_{21} & M_3(a,b,c) & :- & M_2(a,b,c), S_2^0(a). \\ r_{22} & V_1(a,d) & :- & S_2^1(a), S_5(a,d). \\ r_{23} & S_2^0(a) & :- & S_2(a). \\ r_{24} & S_2^1(a) & :- & S_2(a). \end{array}$$

Only $\sigma - rules$, $\pi - rules$ and $\rho - rules$ should define a view relation over the same source relation.

▲

Proposition 4.4. *Given g is a nonempty set of NR-Datalog* rules over schemas in $\mathcal{S}$, each of which satisfies the decomposable conditions, $\textsc{Decompose}(g, \mathcal{S})$ returns $(C_g, \mathcal{S}_f)$ where (1) C_g is a nonempty set of only atomic rules over a new schema set $\mathcal{S}_f$; (2) $\mathcal{S}_f \supseteq \mathcal{S}$; (3) if two rules in C_g have a common relation name in their bodies, each of the rules is either a ρ-rule or a σ-rule or a π-rule.* ■

Proof Sketch. (1) We show that the sequence of laws, comprising $\mathcal{L}_1$, $\mathcal{L}_2$ (with $\mathcal{L}_{2.1}$ through $\mathcal{L}_{2.6}$ inside), $\mathcal{L}_3$ and $\mathcal{L}_4$, is sufficient to rewrite a set g to a set C_g of only atomic rules. In the flow (F1), $\mathcal{L}_1$ rewrites g to a set T_1 of only normal rules, each of which cannot directly decompose to any $\cup$-*rules*. $\mathcal{L}_2$ decomposes each normal rule in T_1 to atomic rules without $\cup$-*rules* (as explained a bit later) and then merges the results into T_2, so T_2 contains only atomic rules but has no $\cup$-*rules*. $\mathcal{L}_3$ and $\mathcal{L}_4$ update the atomic rules and introduce possible $\cup$-*rules* and ρ-*rules*. Consequently, all rules in the final set C_g of (F1) are atomic. When handling $\mathcal{L}_2$, to prove a normal rule is decomposed to a set of atomic rules, we can go step-by-step with the flow (F2). If a normal, non-atomic rule r has some constraints in the body, each constraint may be part of either a selection or a cross-product. Based on the constraints, it is possible to extract σ-*rules* first ($\mathcal{L}_{2.1}$) and keep all constraints not belonging to these rules to the end of (F2) (i.e., in r_6). Without constraints, the body of r with positive literals and potential negative literals generally expresses a combination of projections and joins (e.g. a binary joins between two positive literals, a left-anti-semi-join between a positive literal and a negative literal). Before and after extracting $\bowtie$-*rules* ($\mathcal{L}_{2.3}$, $\mathcal{L}_{2.5}$), we may drop unnecessary

Table 4.1: Intermediate tables of middle relations.

(4.1a) M$_1$

sid	name	city	active
11	Ken	Tokyo	1

(4.1b) M$_2$

cid	name	city
102	Kai	Tokyo
104	Mori	Tokyo

(4.1c) tokyoac$_0$

name
Ken

(4.1d) tokyoac$_1$

name
Kai
Mori

(4.1a') M$_1'$

sid	name	city	active
~~11~~	~~Ken~~	~~Tokyo~~	~~1~~
14	*Shin*	*Tokyo*	*1*

(4.1b') M$_2'$

cid	name	city
~~102~~	~~Kai~~	~~Tokyo~~
104	Mori	Tokyo
105	*Yuri*	*Tokyo*

(4.1c') tokyoac$_0'$

name
~~Ken~~
Shin

(4.1d') tokyoac$_1'$

name
~~Kai~~
Mori
Yuri

attributes with π-*rules* ($\mathcal{L}_{2.2}$, $\mathcal{L}_{2.4}$, $\mathcal{L}_{2.6}$). If all middle forms $r_1, \ldots, r_5$ in (F2) are not atomic and r_6 is reached, then r_6 would contain a positive literal and possibly some constraints not in any extracted σ-*rules*. Due to the safety, r_6 will be either a ρ-*rule* or a $\times$-*rule*. (2) The new set $\mathcal{S}_f$ includes not only the existing schemas in $\mathcal{S}$ but also the schemas of intermediate relations introduced in the decomposition. (3) With $\mathcal{L}_4$, we can rename so that only rename, selection, and projection views can be defined over the same source. □

Next, let us follow an example of query decomposition in NR-Datalog* and another example of obtaining sub-synthesis problems after the decomposition.

Example 4.5. Consider problem PROB($\mathcal{S}_1, \mathcal{E}_1$) in Example 3.2. Suppose that we synthesized a *get* program $g = \{r_1, r_2\}$ over schemas in $\mathcal{S}$ = ({staffs, customers}, tokyoac) where where

```
r1   tokyoac(n) :- staffs(i,n,c,a) , c="Tokyo" , a="1".
r2   tokyoac(n) :- customers(i,n,c) , c="Tokyo".
```

DECOMPOSE($g, \mathcal{S}$) outputs ($C_g, \mathcal{S}_f$) where $C_g = \{r_1^a, r_2^a, r_3^a, r_4^a, r_5^a\}$, r_i^as are in Example 4.2, and $\mathcal{S}_f = \mathcal{S} \cup \{M_1$, M$_2$, tokyoac$_0$, tokyoac$_1\}$. ▲

Example 4.6. Let us see how to divide the synthesis after obtaining the decomposed set C_g in Example 4.5. Suppose that we skip the step of forward-propagating FDs. By applying C_g on T_{source} = (1.1*a*, 1.1*b*) and $T_{source'}$ = (1.1*a*′, 1.1*b*′), one by one, we obtain a full example $\mathcal{E}_f$ (Tables 1.1 and 4.1).

Then, we can divide the synthesis in a loop of SYNTHB into five subproblems:

$$\begin{aligned}
\mathrm{PA}_1 &:= \textsc{ProbA}((\{\texttt{staffs}\}, \mathrm{M}_1), \mathcal{E}_1, \{r_1^a\}) \\
\mathrm{PA}_2 &:= \textsc{ProbA}((\{\texttt{customers}\}, \mathrm{M}_2), \mathcal{E}_2, \{r_2^a\}) \\
\mathrm{PA}_3 &:= \textsc{ProbA}((\{\mathrm{M}_1\}, \texttt{tokyoac}_0), \mathcal{E}_3, \{r_3^a\}) \\
\mathrm{PA}_4 &:= \textsc{ProbA}((\{\mathrm{M}_2\}, \texttt{tokyoac}_1), \mathcal{E}_4, \{r_4^a\}) \\
\mathrm{PA}_5 &:= \textsc{ProbA}((\{\texttt{tokyoac}_0, \texttt{tokyoac}_1\}, \texttt{tokyoac}), \mathcal{E}_5, \{r_5^a\})
\end{aligned}$$

where $\textsc{ProbA}(\mathit{schema} = \mathcal{S}_i, \mathit{example} = \mathcal{E}_i, \mathit{atomic\_query} = \{r_i^a\})$, which will be discussed in Chapter 5, denotes the problem to synthesize a well-behaved pair (get_a, put_a) such that the pair is consistent with $\mathcal{E}_i$ and $get_a = \{r_i^a\}$.

Each example $\mathcal{E}_i = (T^i_{source}, T^i_{view}, T^i_{source'}, T^i_{view'})$ includes tables in both forward and backward transformations corresponding to query $\{r_i^a\}$:

$$\begin{aligned}
\mathcal{E}_1 &= (\{1.1a\}, \{4.1a\}, \{1.1a'\}, \{4.1a'\}) \\
\mathcal{E}_2 &= (\{1.1b\}, \{4.1b\}, \{1.1b'\}, \{4.1b'\}) \\
\mathcal{E}_3 &= (\{4.1a\}, \{4.1c\}, \{4.1a'\}, \{4.1c'\}) \\
\mathcal{E}_3 &= (\{4.1b\}, \{4.1d\}, \{4.1b'\}, \{4.1d'\}) \\
\mathcal{E}_5 &= (\{4.1c, 4.1d\}, \{1.1c\}, \{4.1c', 4.1d'\}, \{1.1c'\})
\end{aligned}$$

▲

4.2 Forward-Propagating Functional Dependencies

At the end of the previous section, we saw that the synthesis could be reduced to many sub-synthesis after the query decomposition. However, relations involving a sub-synthesis task are not associated with any functional dependencies. The lack of this information will cause a loss of templates or candidate rules for future synthesis. To address this shortage, we will forward-propagate FDs attached to the source to all other relations through the specified set of atomic queries.

Suppose that we have a set C_g of atomic queries get_as that form a query get satisfying the given example.

Algorithm 2 describes how to forward-propagate FDs from the source to the view. The procedure FORWARDPROPAGATEFDS takes as input a set of schemas $\mathcal{S}$ (of the

Table 4.2: Forward propagation of FDs for atomic rules

Type	Atomic Rule r	Schema & Mapping	FDs: $\mathcal{F}_V$
ρ	$V(x_1,\dots,x_n) :- S(y_1,\dots,y_n).$ where $(x_1,\dots,x_n) = \rho(y_1,\dots,y_n)$ ρ is a permutation function	$\mathbb{S} = A^y_1 A^y_2 \dots A^y_n$ $\mathbb{V} = A_1 A_2 \dots A_n$ $\mathcal{M}_r[A^y_j] = A_i$ if $\exists i.\ y_j = x_i$	$\mathcal{F}_V = \textsc{AdaptAttr}(\mathcal{F}_S, \mathcal{M}_r)$
	Example: $V(v_2, v_1, v_0) :- S(v_0, v_1, v_2).$	$\mathbb{S} = ABC$, $\mathbb{V} = A_1A_2A_3$ $\mathcal{M}_r[A] = A_3, \mathcal{M}_r[B] = A_2, \mathcal{M}_r[C] = A_1$	$\mathcal{F}_S = \{A \rightarrow B\}$ $\mathcal{F}_V = \{A_3 \rightarrow A_2\}$
$\cup$	$V(x_1,\dots,x_n) :- S_1(x_1,\dots,x_n);$ $S_2(x_1,\dots,x_n).$	$\mathbb{S}_1 = \mathbb{S}_2 = \mathbb{V} = A_1A_2\dots A_n$ $\mathcal{M}_r[A_j] = A_j$	$\mathcal{F}_V = \emptyset$
	Example: $V(v_0, v_1, v_2) :- S_1(v_0, v_1, v_2); S_2(v_0, v_1, v_2).$	$\mathbb{S}_1 = \mathbb{S}_2 = ABC$, $\mathbb{V} = A_1A_2A_3$ $\mathcal{M}_r[A] = A_1, \mathcal{M}_r[B] = A_2, \mathcal{M}_r[C] = A_3$	$\mathcal{F}_{S_1} = \{A \rightarrow B\}, \mathcal{F}_{S_2} = \{A \rightarrow B\}$ $\mathcal{F}_V = \emptyset$
$\setminus$	$V(x_1,\dots,x_n) :- S_1(x_1,\dots,x_n),$ $\neg S_2(x_1,\dots,x_n).$	$\mathbb{S}_1 = \mathbb{S}_2 = \mathbb{V} = A_1A_2\dots A_n$ $\mathcal{M}_r[A_j] = A_j$	$\mathcal{F}_V = \textsc{AdaptAttr}(\mathcal{F}_{S_1}, \mathcal{M}_r)$
	Example: $V(v_0, v_1, v_2) :- S_1(v_0, v_1, v_2), \neg S_2(v_0, v_1, v_2).$	$\mathbb{S}_1 = \mathbb{S}_2 = ABC$, $\mathbb{V} = A_1A_2A_3$ $\mathcal{M}_r[A] = A_1, \mathcal{M}_r[B] = A_2, \mathcal{M}_r[C] = A_3$	$\mathcal{F}_{S_1} = \{A \rightarrow B\}, \mathcal{F}_{S_2} = \{B \rightarrow C\}$ $\mathcal{F}_V = \{A_1 \rightarrow A_2\}$
$\cap$	$V(x_1,\dots,x_n) :- S_1(x_1,\dots,x_n),$ $S_2(x_1,\dots,x_n).$	$\mathbb{S}_1 = \mathbb{S}_2 = \mathbb{V} = A_1A_2\dots A_n$ $\mathcal{M}_r[A_j] = A_j$	$\mathcal{F}_V = \textsc{AdaptAttr}(\mathcal{F}_{S_1}, \mathcal{M}_r)$ $\cup\textsc{AdaptAttr}(\mathcal{F}_{S_2}, \mathcal{M}_r)$
	Example: $V(v_0, v_1, v_2) :- S_1(v_0, v_1, v_2), S_2(v_0, v_1, v_2).$	$\mathbb{S}_1 = \mathbb{S}_2 = ABC$, $\mathbb{V} = A_1A_2A_3$ $\mathcal{M}_r[A] = A_1, \mathcal{M}_r[B] = A_2, \mathcal{M}_r[C] = A_3$	$\mathcal{F}_{S_1} = \{A \rightarrow B\}, \mathcal{F}_{S_2} = \{B \rightarrow C\}$ $\mathcal{F}_V = \{A_1 \rightarrow A_2, A_2 \rightarrow A_3\}$
σ	$V(x_1,\dots,x_n) :- S(x_1,\dots,x_n),$ $v_1 \oplus c_1, \dots, v_m \oplus c_m.$ where $m > 0$, $\{v_1,\dots,v_m\} \subseteq \{x_1,\dots,x_n\}$	$\mathbb{S} = \mathbb{V} = A_1A_2\dots A_n$ $\mathcal{M}_r[A_j] = A_j$	$\mathcal{F}_V = \textsc{AdaptAttr}(\mathcal{F}_S, \mathcal{M}_r)$
	Example: $V(v_0, v_1, v_2) :- S(v_0, v_1, v_2), v_1 != \text{“}b_1\text{”}.$	$\mathbb{S} = ABC$, $\mathbb{V} = A_1A_2A_3$ $\mathcal{M}_r[A] = A_1, \mathcal{M}_r[B] = A_2, \mathcal{M}_r[C] = A_3$	$\mathcal{F}_S = \{A \rightarrow B, B \rightarrow C\}$, $\mathcal{F}_V = \{A_1 \rightarrow A_2, A_2 \rightarrow A_3\}$
π	$V(x_1,\dots,x_m) :- S(y_1,\dots,y_n).$ where $\{x_1,\dots,x_m\} \subsetneq \{y_1,\dots,y_n\}$	$\mathbb{S} = A^y_1 A^y_2 \dots A^y_n$ $\mathbb{V} = A_1A_2\dots A_n$ $\mathcal{M}_r[A^y_j] = A_i$ if $\exists i. y_j = x_i$ else $null$	$\mathcal{F}_V = \textsc{AdaptAttr}($ $\textsc{RepairTree}(\mathcal{F}_S, \mathcal{M}_r)$ $)$
	Example: $V(v_1, v_2, v_3) : -S(v_0, v_1, v_2, v_3).$	$\mathbb{S} = ABCD$, $\mathbb{V} = A_1A_2A_3$ $\mathcal{M}_r[A] = null, \mathcal{M}_r[B] = A_1$ $\mathcal{M}_r[C] = A_2, \mathcal{M}_r[D] = A_3$	$\mathcal{F}_S = \{B \rightarrow A, B \rightarrow C, A \rightarrow D\}$ $\mathcal{F}_V = \{A_1 \rightarrow A_2, A_1 \rightarrow A_3\}$
$\bowtie$	$V(x_1,\dots,x_m) :- S_1(y_1,\dots,y_n),$ $S_2(v_1,\dots,v_o).$ where $\{y_1,\dots,y_n\} \cup \{v_1,\dots,v_o\} = \{x_1,\dots,x_m\}$, $\{y_1,\dots,y_n\} \cap \{v_1,\dots,v_o\} \neq \emptyset$	$\mathbb{S}_1 = A^y_1 A^y_2 \dots A^y_n$ $\mathbb{S}_2 = A^v_1 A^v_2 \dots A^v_o$ $\mathbb{V} = A_1A_2\dots A_m$ $\mathcal{M}_r[A^y_j] = A_i$ if $\exists i.\ y_j = x_i$ $\mathcal{M}_r[A^v_j] = A_i$ if $\exists i.\ v_j = x_i$	$\mathcal{F}_V = \textsc{AdaptAttr}(\mathcal{F}_{S_1}, \mathcal{M}_r)$ $\cup\textsc{AdaptAttr}(\mathcal{F}_{S_2}, \mathcal{M}_r)$
	Example: $V(v_0, v_1, v_2, v_3) :- S_1(v_0, v_1, v_2), S_2(v_1, v_3).$	$\mathbb{S}_1 = ABC, \mathbb{S}_2 = BD$, $\mathbb{V} = A_1A_2A_3A_4$ $\mathcal{M}_r[A] = A_1, \mathcal{M}_r[B] = A_2$ $\mathcal{M}_r[C] = A_3, \mathcal{M}_r[D] = A_4$	$\mathcal{F}_{S_1} = \{A \rightarrow B, A \rightarrow C\}, \mathcal{F}_{S_2} = \{B \rightarrow D\}$ $\mathcal{F}_V = \{A_1 \rightarrow A_2, A_1 \rightarrow A_3, A_2 \rightarrow A_4\}$
$\times$	$V(x_1,\dots,x_{m+n}) :- S_1(y_1,\dots,y_n),$ $v_1 = c_1, \dots, v_m = c_m.$ where $\{y_1,\dots,y_n\} \cup \{v_1,\dots,v_m\} = \{x_1,\dots,x_{m+n}\}$, $\{y_1,\dots,y_n\} \cap \{v_1,\dots,v_m\} = \emptyset$	$\mathbb{S}_1 = A^y_1 A^y_2 \dots A^y_n$ $dom(v_1) = A^v_1, \dots, dom(v_m) = A^v_m$ $\mathbb{V} = A_1A_2\dots A_{m+n}$ $\mathcal{M}_r[A^y_j] = A_i$ if $\exists i.\ y_j = x_i$ $\mathcal{M}_r[A^v_j] = A_i$ if $\exists i.\ v_j = x_i$	$\mathcal{F}_V = \textsc{AdaptAttr}(\mathcal{F}_S, \mathcal{M}_r)$
	Example: $V(v_3, v_0, v_1, v_2) : -S(v_0, v_1, v_2), v_3 = 1.$	$\mathbb{S} = ABC, dom(v_3) = D$, $\mathbb{V} = A_1A_2A_3A_4$ $\mathcal{M}_r[A] = A_2, \mathcal{M}_r[B] = A_3$ $\mathcal{M}_r[C] = A_4, \mathcal{M}_r[D] = A_1$	$\mathcal{F}_S = \{A \rightarrow B, B \rightarrow C\}$ $\mathcal{F}_V = \{A_2 \rightarrow A_3, A_2 \rightarrow A_4\}$

Algorithm 2: Forward-Propagating Functional Dependencies

```
Input: $\mathcal{S}$: a set of schemas, $C_g$: a set of atomic rules
Output: $\mathcal{S}_f$: a new set of schemas where each relation is associated with FDs
1  procedure ForwardPropagateFDs ($\mathcal{S}, C_g$):
2      $visited \leftarrow dict\{r : False$ for $r \in C_g\}$
3      $Q \leftarrow \emptyset$
4      $R \leftarrow$ a set of relation names occurring in $C_g$
5      for $r \in \mathcal{S}$ do
6          if $r$ is a source then $Q, R \leftarrow Q \cup \{\textsc{Name}(r)\}, R - \{\textsc{Name}(r)\}$
7      while $R \neq \emptyset$ do
8          $R_f \leftarrow \{r \mid r \in C_g \wedge visited[r] = False\}$
9          for $r \in R_f$ do
10             if $\textsc{Name}(\textsc{Head}(r)) \notin Q \wedge \textsc{Name}(\textsc{BodyLiteral}(r)) \subseteq Q$ then
11                 $\mathcal{S} \leftarrow$ PropagateFDs-A($\mathcal{S}, r$)
12                 $Q, R \leftarrow Q \cup \{\textsc{Name}(\textsc{Head}(r))\}, R - \{\textsc{Name}(\textsc{Head}(r))\}$
13                 $visited[r] = True$
14     return $\mathcal{S}$ as $\mathcal{S}_f$
```

source, the view and intermediate relations) and a set of atomic rules C_g and produces as output the updated set of schemas $\mathcal{S}_f$ with FDs computed for each relation. We say a relation is *processed* (*unprocessed*) if its FDs are (not) computed, and a rule is *visited* (*unvisited*) if (not) all relations appearing in the rule are processed.

ForwardPropagateFDs uses a key-value dictionary *visited* to check if a rule has been visited and two sets R and Q that include the names of unprocessed and processed relations, respectively. Initially, all values of *visited* are *False*, Q is empty, and R contains all relation names in C_g (lines 2-4). Because source relations have FDs given by users, those source names are added to Q and removed from R (lines 5-6). Then, as long as there is an unprocessed relation, a loop of propagating FDs (lines 7-13) will be executed. For each atomic and unvisited rule r in C_g, if all relations appearing in the body of r (*rhs*) are processed and the relation appearing in the head of r (*lhs*) is unprocessed, PropagateFDs-A is invoked to propagate FDs from the *rhs* to the *lhs* of the atomic rule r. Then, the relation in *lhs* is processed, and its name should be added to Q and removed from R. The rule r is also set to be visited.

Table 4.2 denotes the result of invoking the subprocedure PropagateFDs-A for each atomic rule r with view V and source S (or (S_1, S_2)). In this table, in addition to the description of atomic type and atomic rule in the first two columns, the third column expresses the schemas and rule-based attribute mapping, while the last column specifies the calculation of FDs. For each atomic type, we directly provide an example in the lower row of formal representations. We write $\mathcal{F}_X$ as the FD set against a relation X, $\mathcal{M}_r$ as an attribute mapping of a rule r that maps attributes in the rule body to attributes in the rule head, and $\textsc{AdaptAttr}(\mathcal{F}_X, \mathcal{M}_r)$ as a function that revises $\mathcal{F}_X$ following $\mathcal{M}_r$.

Example 4.7. Given a *π-rule r*

$$V(v_1, v_2, v_3) :- S(v_0, v_1, v_2, v_3).$$

where $\mathbb{S} = ABCD$, $\mathcal{F}_S = \{B \rightarrow A, B \rightarrow C, A \rightarrow D\}$, $\mathbb{V} = A_1A_2A_3$, we have:

$\mathcal{M}_r[A] = \textit{null}$ (the A-position of S is v_0 and there is no v_0 in V),

$\mathcal{M}_r[B] = A_1$ (the B-position of S is v_1 and v_1 is at A_1-position of V),

$\mathcal{M}_r[C] = A_2$,

$\mathcal{M}_r[D] = A_3$,

and

$\mathcal{F}_V = \{A_1 \rightarrow A_2, A_1 \rightarrow A_3\}$.

When attribute A is dropped from the source S, before adapting the attributes with respect to the view V, we need to repair the tree form of the FDs (function $\textsc{RepairTree}$ in Table 4.2) by deleting all edges related to A (e.g., $B \rightarrow A$, $A \rightarrow D$) and possibly adding edges from A's parents to A's children (e.g., $B \rightarrow D$) if A is not at a root. ▲

To continue this section, let us look at two examples of forward-propagating FDs over a set of atomic queries in the synthesis against two specifications in Example 3.2 and Example 3.3.

Example 4.8. Consider problem $\textsc{Prob}(\mathcal{S}_1, \mathcal{E}_1)$ in Example 3.2. We have two sources `staffs` and `customers` with the following schemas:

`staffs(sid:SID, name:NAME, city:CITY, active:ACTIVE,` $\mathcal{F}_{\texttt{staffs}} = \emptyset$`)`

`customers(cid:CID, name:NAME, city:CITY,` $\mathcal{F}_{\texttt{customers}} = \emptyset$`)`

After the decomposition in Example 4.5, we have a set of atomic queries C_g =

$\{r^a_1, r^a_2, r^a_3, r^a_4, r^a_5\}$ that are defined over a set of schemas $\mathcal{S}_f$ where

```
r1a   M₁(i,n,c,a) :- staffs(i,n,c,a) , c="Tokyo" , a="1".
r2a   M₂(i,n,c) :- customers(i,n,c) , c="Tokyo".
r3a   tokyoac₀(n) :- M₁(i,n,c,a).
r4a   tokyoac₁(n) :- M₂(i,n,c).
r5a   tokyoac(n) :- tokyoac₀(n) ; tokyoac₁(n).
```

and

$\mathcal{S}_f = \{$
  staffs(sid:SID, name:NAME, city:CITY, active:ACTIVE, $\mathcal{F}_{\texttt{staffs}} = \emptyset$),
  customers(cid:CID, name:NAME, city:CITY, $\mathcal{F}_{\texttt{customers}} = \emptyset$),
  $\texttt{M}_1$(sid:SID, name:NAME, city:CITY, active:ACTIVE),
  $\texttt{M}_2$(cid:CID, name:NAME, city:CITY),
  $\texttt{tokyoac}_0$(name:NAME),
  $\texttt{tokyoac}_1$(name:NAME),
  tokyoac(name:NAME)
$\}$

Besides the sources `stasffs` and `customers`, the remaining relations in $\mathcal{S}_f$ are not associated with any FDs. A run of ForwardPropagateFDs($\mathcal{S}_f, \mathcal{C}_g$) would compute FDs for these remaining relations in the following order:

$$\mathcal{F}_{\texttt{staffs}} \Rightarrow \mathcal{F}_{\texttt{M}_1}$$
$$\mathcal{F}_{\texttt{customers}} \Rightarrow \mathcal{F}_{\texttt{M}_2}$$
$$\mathcal{F}_{\texttt{M}_1} \Rightarrow \mathcal{F}_{\texttt{tokyoac}_0}$$
$$\mathcal{F}_{\texttt{M}_2} \Rightarrow \mathcal{F}_{\texttt{tokyoac}_1}$$
$$\mathcal{F}_{\texttt{tokyoac}_0}, \mathcal{F}_{\texttt{tokyoac}_1} \Rightarrow \mathcal{F}_{\texttt{tokyoac}}$$

Since $\mathcal{F}_{\texttt{staffs}} = \mathcal{F}_{\texttt{customers}} = \emptyset$, we have

$$\mathcal{F}_{\texttt{M}_1} = \mathcal{F}_{\texttt{M}_2} = \mathcal{F}_{\texttt{tokyoac}_0} = \mathcal{F}_{\texttt{tokyoac}_1} = \mathcal{F}_{\texttt{tokyoac}} = \emptyset$$

Hence, the updated $\mathcal{S}_f$ is as below:

$$
\begin{aligned}
&\mathcal{S}_f = \{\\
&\quad \texttt{staffs(sid:SID, name:NAME, city:CITY, active:ACTIVE, } \mathcal{F}_{\texttt{staffs}} = \emptyset\texttt{)},\\
&\quad \texttt{customers(cid:CID, name:NAME, city:CITY, } \mathcal{F}_{\texttt{customers}} = \emptyset\texttt{)},\\
&\quad \texttt{M}_1\texttt{(sid:SID, name:NAME, city:CITY, active:ACTIVE, } \mathcal{F}_{\texttt{M}_1} = \emptyset\texttt{)},\\
&\quad \texttt{M}_2\texttt{(cid:CID, name:NAME, city:CITY, } \mathcal{F}_{\texttt{M}_2} = \emptyset\texttt{)},\\
&\quad \texttt{tokyoac}_0\texttt{(name:NAME, } \mathcal{F}_{\texttt{tokyoac}_0} = \emptyset\texttt{)},\\
&\quad \texttt{tokyoac}_1\texttt{(name:NAME, } \mathcal{F}_{\texttt{tokyoac}_1} = \emptyset\texttt{)},\\
&\quad \texttt{tokyoac(name:NAME, } \mathcal{F}_{\texttt{tokyoac}} = \emptyset\texttt{)}\\
&\}
\end{aligned}
$$

Based on the new $\mathcal{S}_f$, we see that the relations appearing on each subproblem in Example 4.6 are not constrained by any FDs. ▲

Example 4.9. Consider problem $\textsc{Prob}(\mathcal{S}_2, \mathcal{E}_2)$ in Example 3.3. We have a source S of the following schema:

$$S(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, D : \mathbb{D}, \mathcal{F}_S = \{A \rightarrow B, A \rightarrow D\})$$

Suppose that we synthesized a set of atomic queries $C_g = \{r_1^o, r_2^o, r_3^o\}$ that are defined over a set of schemas $\mathcal{S}_f$ where

$$
\begin{array}{llll}
r_1^o & M_1(v_0, v_1, v_2) & :- & S(v_0, v_1, v_2, v_3).\\
r_2^o & M_2(v_0, v_1, v_2) & :- & M_1(v_0, v_1, v_2)\ ,\ v_2 = \text{“}T\text{”}.\\
r_3^o & V(v_0, v_1, v_2) & :- & M_2(v_0, v_1, v_2).
\end{array}
$$

and

$$
\begin{aligned}
&\mathcal{S}_f = \{\\
&\qquad S(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, D : \mathbb{D}, \mathcal{F}_S = \{A \rightarrow B, A \rightarrow D\}),\\
&\qquad M_1(A_{M_1} : \mathbb{A}, B_{M_1} : \mathbb{B}, C_{M_1} : \mathbb{C}),\\
&\qquad M_2(A_{M_2} : \mathbb{A}, B_{M_2} : \mathbb{B}, C_{M_2}),\\
&\qquad V(A_V : \mathbb{A}, B_V : \mathbb{B}, C_V : \mathbb{C}),\\
&\}
\end{aligned}
$$

Here, we use the terms A_{M_1} and B_{M_2} to precisely describe attributes in the intermediate

Table 4.3: Intermediate tables containing internal functional dependencies

(4.3a) M_1

A	B	C
1	b_1	F
1	b_1	T
2	b_2	T

(4.3a) M_2

A	B	C
1	b_1	T
2	b_2	T

(4.3a) M_1'

A	B	C
~~1~~	~~$b_\top$~~	~~F~~
~~1~~	~~$b_\top$~~	~~T~~
1	b_2	*F*
1	b_2	*T*
2	b_2	T

(4.3a) M_2'

A	B	C
~~1~~	~~$b_\top$~~	~~T~~
1	b_2	*T*
2	b_2	T

relations, but they can be simplified to just A and B. By forward-propagating FDs in order $\mathcal{F}_S \Rightarrow \mathcal{F}_{M_1} \Rightarrow \mathcal{F}_{M_2} \Rightarrow \mathcal{F}_V$, we can update $\mathcal{S}_f$ as below:

$$
\begin{aligned}
\mathcal{S}_f = \{ & \\
& S(A:\mathbb{A}, B:\mathbb{B}, C:\mathbb{C}, D:\mathbb{D}, \mathcal{F}_S = \{A \rightarrow B, A \rightarrow D\}), \\
& M_1(A_{M_1}:\mathbb{A}, B_{M_1}:\mathbb{B}, C_{M_1}:\mathbb{C}, \mathcal{F}_{M_1} = \{A_{M_1} \rightarrow B_{M_1}), \\
& M_2(A_{M_2}:\mathbb{A}, B_{M_2}:\mathbb{B}, C_{M_2}, \mathcal{F}_{M_2} = \{A_{M_2} \rightarrow B_{M_2}), \\
& V(A_V:\mathbb{A}, B_V:\mathbb{B}, C_V:\mathbb{C}, \mathcal{F}_V = \{A_V \rightarrow B_V), \\
\} &
\end{aligned}
$$

Then, similarly to Example 4.6, we can compute the full example $\mathcal{E}_f$ (Tables 1.2 and 4.3) and divide the synthesis in a loop of SYNTHB info three subproblems:

$$
\begin{aligned}
\text{PA}_1 &:= \text{PROBA}((\{S\}, M_1), \mathcal{E}_1, \{r_1^o\}) \\
\text{PA}_2 &:= \text{PROBA}((\{M_1\}, M_2), \mathcal{E}_2, \{r_2^o\}) \\
\text{PA}_3 &:= \text{PROBA}((\{M_2\}, V), \mathcal{E}_3, \{r_3^o\})
\end{aligned}
$$

where PROBA($\mathit{schema} = \mathcal{S}_i$, $\mathit{example} = \mathcal{E}_i$, $\mathit{atomic\_query} = \{r_i^a\}$), denotes the problem to synthesize a well-behaved pair (get_a, put_a) such that the pair is consistent with $\mathcal{E}_i$ and $get_a = \{r_i^a\}$. Each example $\mathcal{E}_i$ is of the form $(T^i_{source}, T^i_{view}, T^i_{source'}, T^i_{view'})$ where

$$
\begin{aligned}
\mathcal{E}_1 &= (\{1.2a\}, \{4.3a\}, \{1.2a'\}, \{4.3a'\}) \\
\mathcal{E}_2 &= (\{4.3a\}, \{4.3b\}, \{4.3a'\}, \{4.3b'\}) \\
\mathcal{E}_3 &= (\{4.3b\}, \{1.2b\}, \{4.3b'\}, \{1.2b'\})
\end{aligned}
$$

In the three subproblems, the relations S, M_1, M_2 and V are constrained by the corresponding FDs in the new schema set $\mathcal{S}_f$ above. ▲

Discussion on Significance of Forward-Propagating FDs

In reality, tables may have internal functional dependencies, i.e., data on tables may depend on each other. Functional dependencies are widely used in real-world schemas and play a crucial role in the view update tasks. As mentioned in the introduction, existing approaches cannot synthesize bidirectional programs from examples including internal FDs. To avoid missing solutions for real-world view update problems, supporting FDs is necessary.

Forward propagation of FDs is important because, with information of FDs for relations, we later can prepare some more templates encoding constraints and effects of FDs against the relations, which enrich the search space of the synthesis. Moreover, forward-propagating FDs may help avoid the automatic discovery of FDs from tables - a challenge recognized as difficult in database research [24].

Limitation

If the inferred FDs are overwritten to the empty (e.g., over unions), no templates related to those FDs will be used, i.e., no rules related to those FDs will be generated. Consider the query with four atomic rules:

$$\begin{aligned}
&V(v_0) :- S_1(v_0).\\
&V(v_0) :- S_2(v_0).\\
&S_1(v_0) :- S(v_0).\\
&S_2(v_0) :- S(v_0).
\end{aligned}$$

then $\mathcal{F}_{S_1} = \mathcal{F}_{S_2} = \mathcal{F}_S$ and $\mathcal{F}_V = \emptyset$ (while we expect that $\mathcal{F}_V$ and $\mathcal{F}_S$ should coincide). Since $\mathcal{F}_V$ is computed as empty, no constraint rules are generated later. The synthesized programs (if one exists) would break the PutGet law, if we input the *put* program with an updated view V' including some tuples that violate the "expected" constraints of $\mathcal{F}_{V'}$ (which is the same as $\mathcal{F}_V$).

Our method of forward propagation is currently not perfect since we only focus on FDs as constraints on relations. In the future, if we handle other types of relation constraints (domain constraints, cardinality constraints, inclusion dependencies), the forward propagation could be revised to be more useful, so that we can fully propagate constraints from the source to the view.

4.3 Summary

In this chapter, we introduce how to decompose the query written in NR-Datalog* into a set of atomic NR-Datalog* rules and how to forward-propagate information of functional dependencies from the source through the set of atomic rules to all other relations appearing in the rules. Then we can divide the synthesis problem of (get, put) to many sub-synthesis problems of the form $\textsc{ProbA}(\mathit{schema} = \mathcal{S}_i, \mathit{example} = \mathcal{E}_i, \mathit{atomic\_query} = \{r_i^a\})$. Each relation in the schema set $\mathcal{S}_i$ of a subproblem is clearly associated with a set of FDs, either empty or non-empty. In Chapters 5 and 6, we will solve the subproblem using templates.

5 Synthesizing View Update Programs with Minimal-Effect Templates

After decomposing the synthesized query into a set of atomic queries get_a, it is possible to divide the synthesis of (get, put) into many sub-synthesis of (get_a, put_a). To synthesize the atomic view update put_a given the get_a, we will design well-behaved templates encoding well-behaved view update strategies for atomic queries, use these templates to generate candidate rules of put_a, and adapt ProSynth.

In this chapter, we relax the extra constraints and effects of dependencies, or, in other words, consider the dependencies as empty. We focus on templatizing minimal-effect view update strategies, which are strategies that involve minimal and no redundant changes. We discuss this templatizing in Section 5.1. We implement a prototype SynthBX using the designed templates and evaluate it on a suite of 56 practical benchmarks. Section 5.2 covers the implementation and evaluation. We summarize this chapter in Section 5.3.

5.1 Templatizing Minimal-Effect View Update Strategies

In the previous chapter, we know that after decomposing the synthesized query to be atomic rules, the synthesis of (get, put) can be reduced to sub-synthesis problems, each of form $\mathrm{PA}_i := \textsc{ProbA}((\mathbb{S}_i, \mathbb{V}_i), \mathcal{E}_i, \{r_i^a\})$. In this section, we present preparing templates to adapt ProSynth$^+$ to solve these subproblems and thereby find out a solution for (get, put).

To solve a sub-synthesis problem $\mathrm{PA}_i := \textsc{ProbA}((\mathbb{S}_i, \mathbb{V}_i), \mathcal{E}_i, \{r_i^a\})$, we need to synthesize a well-behaved put_a of type $(\mathbb{S}_i, \mathbb{V}_i) \longrightarrow \mathbb{S}_i$ corresponding to the given atomic $get_a = \{r_i^a\}$ of type $\mathbb{S}_i \longrightarrow \mathbb{V}_i$ such that (get_a, put_a) is consistent with $\mathcal{E}_i$. If a put_a is synthesized from a common space (e.g., the space of queries), the well-behavedness of (get_a, put_a) cannot be guaranteed since no "well-behaved" constraints exist between two search spaces of get_a and put_a. Additionally, it is non-trivial to enumerate all possible put_a that can be synthesized and verify their well-behavedness with the given get_a. This difficulty applies to the synthesis of both the simpler put_a and the more complex put. To address the well-behavedness issue, it is important to prepare a "good" set of candidate rules for view update. We focus on the preparation against put_a because it can be lighter than the preparation against put.

Fortunately, it is known in the database community [21, 22, 23] that there is a complete and finite set of strategies with *minimal effects* for well-behaved put_a if get_a is an atomic query. With this, we can prepare templates encoding the strategies for efficient synthesis of put_a while guaranteeing that they are well-behaved.

However, existing strategies normally describe how to translate a single change (an insertion or a deletion) against a view to a change against a source. In practice, the changes against the view often include many insertions and/or deletions. Moreover, these insertions and deletions reflect delta-based data, while the example $\mathcal{E}_i$ of PA_i contains state-based data. With delta relations (Section 2.3), we can switch between state-based data and delta-based data, and propagate multiple changes backward.

To avoid repeating the index i when presenting templates for sub-synthesis PA_i, we rename the atomic problem as $\textsc{ProbA}((\mathbb{S}, \mathbb{V}), \mathcal{E}_\alpha, \{get_a\})$ where $\mathbb{S}$ and $\mathbb{V}$ are respectively schemas of the source S and the view V against the atomic get_a of type α ($\mathbb{S}$ and $\mathbb{V}$ here are different from the originals in Definition 3.1).

Looking at Examples 2.1 and 2.2 of bidirectional programs, we see that if a get_a is given, the corresponding put_a could be constructed with or without computing delta relations against the view, which corresponds to two Setups A and B as follows.

Setup A. Given a $get_a : \mathbb{S} \longrightarrow \mathbb{V}$, a $put_a : (\mathbb{S}, \mathbb{V}) \longrightarrow \mathbb{S}$, which takes (S, V') as input, could be constructed as a combination of rules of four programs including get_a, $P_{\delta V}$, $P_{\Delta S}$ and $P_{S'}$ where:

- get_a derives view data V from S.
- $P_{\delta V} : \mathbb{V} \times \mathbb{V} \longrightarrow \delta\mathbb{V}$, which changes state-based view data (V, V') into delta-based view data δV, includes two rules as below:

$$\begin{array}{llll} r^A_{0-} & \delta V^-(\vec{x}) & :- & V(\vec{x}),\ \neg\, V'(\vec{x}). \\ r^A_{0+} & \delta V^+(\vec{x}) & :- & V'(\vec{x}),\ \neg\, V(\vec{x}). \end{array}$$

- $P_{\Delta S} : \mathbb{S} \times \delta\mathbb{V} \longrightarrow \Delta\mathbb{S}$, which describes update strategies, computes delta-based source data ΔS from an input of $(S, \delta V)$.
- $P_{S'} : \mathbb{S} \times \Delta\mathbb{S} \longrightarrow \mathbb{S}$, which applies delta-based source data ΔS to the original source S to output updated source data S', includes one rule as below:

$$\begin{array}{llll} r^A_0 & S'(\vec{x}) & :- & S(\vec{x}),\ \neg\, \Delta S^-(\vec{x})\ ;\ \Delta S^+(\vec{x}). \end{array}$$

▲

Setup B. Given a $get_a :: \mathbb{S} \longrightarrow \mathbb{V}$, a $put_a :: (\mathbb{S}, \mathbb{V}) \longrightarrow \mathbb{S}$, which takes (S, V') as input, could be constructed as a combination of rules of two programs including $P_{\Delta S}$ and $P_{S'}$ where

- $P_{\Delta S} : \mathbb{S} \times \mathbb{V}' \longrightarrow \Delta\mathbb{S}$ consists of Datalog clauses that compute ΔS from (S, V')
- $P_{S'} : \mathbb{S} \times \Delta\mathbb{S} \longrightarrow \mathbb{S}$, which applies delta-based source data ΔS to the original source S to output updated source data S', includes one rule as below:

$$\begin{array}{llll} r^B_0 & S'(\vec{x}) & :- & S(\vec{x}),\ \neg\, \Delta S^-(\vec{x})\ ;\ \Delta S^+(\vec{x}). \end{array}$$

▲

The two setups above use delta relations to form the program put_a. They share program $P_{S'}$ but differ in the rest programs. Setup A is closer to the view update problem (Figure 2.2) since it translates the update δV to the update ΔS. Setup B is closer to a general view update program since it directly uses (S, V') rather than $(S, \delta V)$ to compute ΔS.

In both Setups A and B, Datalog clauses of $P_{\Delta S}$ are unknown. Later on, for each setup, we will templatize the minimal-effect view update strategies [21, 22, 23] to generate candidates for $P_{\Delta S}$, and you will see that the templates against the same strategies for Setups A and B have the similarity. Note in particular that during the synthesis, we will use templates for either only Setup A or only Setup B instead of mixing them together. With templates, we can generate candidate rules by substitutions. Template parameters S, V, S', V', ΔS and δV can be realized as relation names derived from the given schemas $\mathbb{S}$ and $\mathbb{V}$. Meanwhile, the argument $\vec{x}$ can be realized in the concrete form of $\langle v_0, \ldots, v_n \rangle$.

Before diving deeper into missing templates of $P_{\Delta S}$, let us try to imagine things further after we have template-based clauses for all PA$_i$s. We could *combine* the clauses into a pool where ProSynth$^+$ can process subproblems in parallel. A *put* program successfully synthesized from the pool is an automatic combination of "$put_n, \ldots, put_1$" and would pair with the combination of "$get_1, \ldots, get_n$" to form a bidirectional program (as shown in Figure 1.2). This bidirectional program is primarily a well-behaved composition of atomic well-behaved bidirectional programs (get_a, put_a)s that each satisfies a PA$_i$. To avoid failure during the combination process, we may need to prepare more templates that impose additional constraints on the sources and the views of PA$_i$s. It is important to keep in mind that template-based rules for these constraints (and for $P_{\delta V}$ and $P_{S'}$) should be fixed in the output programs.

Figures 5.1 and 5.2 show our well-designed view update templates for Setup A. Figures 5.3 and 5.4 show our well-designed view update templates for Setup B. The notations used in templates in these four figures are explained in Figure 5.5.

For an atomic query of type α ($\alpha \in \{\rho, \cup, \setminus, \cap, \sigma, \pi, \bowtie, \times\}$), there are three basic types of templates:

- (SDR)$^\alpha$ includes template rules representing strategies for $P_{\Delta S}$;

$\underline{\rho\text{-}rule}\ V(\vec{x})\ :-\ S(p(\vec{x})).$

$\quad (\text{SDR})^{\rho} : \Delta S^{+}, \Delta S^{-}$

$\qquad \Delta S^{+}(p(\vec{x}))\ :-\ \delta V^{+}(\vec{x}).$

$\qquad \Delta S^{-}(p(\vec{x}))\ :-\ \delta V^{-}(\vec{x}).$

$\underline{\cup\text{-}rule}\ V(\vec{x})\ :-\ S_1(\vec{x})\ ;\ S_2(\vec{x}).$

$\quad (\text{SDR})^{\cup} : \Delta S_i^{+}, \Delta S_i^{-}, i \in \{1,2\}$

$\qquad \Delta S_i^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_i(\vec{x}).$

$\qquad \Delta S_i^{+}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ \neg\ S_i(\vec{x}).$

$\qquad \Delta S_i^{+}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ Au_{\cup}^{*}(\vec{x})\ ,\ \neg\ S_i(\vec{x}).$

$\qquad \Delta S_i^{+}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ \neg\ Au_{\cup}^{*}(\vec{x})\ ,\ \neg\ S_i(\vec{x}).$

$\quad (\text{AR})^{\cup} : Au_{\cup}^{*} \in \{Au_{\cup}^{1}, Au_{\cup}^{2}, Au_{\cup}^{12}\}$

$\qquad Au_{\cup}^{1}\ =\ \{t \text{ for } t \in \mathcal{E}_{\cup}|_{\delta V^{+}} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S_1^{+}} \wedge t \notin \mathcal{E}_{\cup}|_{\Delta S_2^{+}}\}$

$\qquad Au_{\cup}^{2}\ =\ \{t \text{ for } t \in \mathcal{E}_{\cup}|_{\delta V^{+}} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S_2^{+}} \wedge t \notin \mathcal{E}_{\cup}|_{\Delta S_1^{+}}\}$

$\qquad Au_{\cup}^{12}\ =\ \{t \text{ for } t \in \mathcal{E}_{\cup}|_{\delta V^{+}} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S_1^{+}} \wedge t \in \mathcal{E}_{\cup}|_{\Delta S_2^{+}}\}$

$\underline{\setminus\text{-}rule}\ V(\vec{x})\ :-\ S_1(\vec{x})\ ,\ \neg\ S_2(\vec{x}).$

$\quad (\text{SDR})^{\setminus} : \Delta S_i^{+}, \Delta S_i^{-}, i \in \{1,2\}$

$\qquad \Delta S_1^{+}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ \neg\ S_1(\vec{x}).$

$\qquad \Delta S_2^{-}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ S_2(\vec{x}).$

$\qquad \Delta S_1^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_1(\vec{x}).$

$\qquad \Delta S_2^{+}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ \neg\ S_2(\vec{x}).$

$\qquad \Delta S_1^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_1(\vec{x})\ ,\ Au_{\setminus}^{*}(\vec{x}).$

$\qquad \Delta S_1^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_1(\vec{x})\ ,\ \neg\ Au_{\setminus}^{*}(\vec{x}).$

$\qquad \Delta S_2^{+}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ Au_{\setminus}^{*}(\vec{x})\ ,\ \neg\ S_2(\vec{x}).$

$\qquad \Delta S_2^{+}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ \neg\ Au_{\setminus}^{*}(\vec{x})\ ,\ \neg\ S_2(\vec{x}).$

$\quad (\text{AR})^{\setminus} : Au_{\setminus}^{*} \in \{Au_{\setminus}^{1}, Au_{\setminus}^{2}, Au_{\setminus}^{12}\}$

$\qquad Au_{\setminus}^{1}\ =\ \{t \text{ for } t \in \mathcal{E}_{\setminus}|_{\delta V^{-}} \text{ if } t \in \mathcal{E}_{\setminus}|_{\Delta S_1^{-}} \wedge t \notin \mathcal{E}_{\setminus}|_{\Delta S_2^{+}}\}$

$\qquad Au_{\setminus}^{2}\ =\ \{t \text{ for } t \in \mathcal{E}_{\setminus}|_{\delta V^{-}} \text{ if } t \in \mathcal{E}_{\setminus}|_{\Delta S_2^{+}} \wedge t \notin \mathcal{E}_{\setminus}|_{\Delta S_1^{-}}\}$

$\qquad Au_{\setminus}^{12}\ =\ \{t \text{ for } t \in \mathcal{E}_{\setminus}|_{\delta V^{-}} \text{ if } t \in \mathcal{E}_{\setminus}|_{\Delta S_1^{-}} \wedge t \in \mathcal{E}_{\setminus}|_{\Delta S_2^{+}}\}$

$\underline{\cap\text{-}rule}\ V(\vec{x})\ :-\ S_1(\vec{x})\ ,\ S_2(\vec{x}).$

$\quad (\text{SDR})^{\cap} : \Delta S_i^{+}, \Delta S_i^{-}, i \in \{1,2\}$

$\qquad \Delta S_i^{+}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ \neg\ S_i(\vec{x}).$

$\qquad \Delta S_i^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_i(\vec{x}).$

$\qquad \Delta S_i^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_i(\vec{x})\ ,\ Au_{\cap}^{*}(\vec{x}).$

$\qquad \Delta S_i^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S_i(\vec{x})\ ,\ \neg\ Au_{\cap}^{*}(\vec{x}).$

$\quad (\text{AR})^{\cap} : Au_{\cap}^{*} \in \{Au_{\cap}^{1}, Au_{\cap}^{2}, Au_{\cap}^{12}\}$

$\qquad Au_{\cap}^{1}\ =\ \{t \text{ for } t \in \mathcal{E}_{\cap}|_{\delta V^{-}} \text{ if } t \in \mathcal{E}_{\cap}|_{\Delta S_1^{-}} \wedge t \notin \mathcal{E}_{\cap}|_{\Delta S_2^{-}}\}$

$\qquad Au_{\cap}^{2}\ =\ \{t \text{ for } t \in \mathcal{E}_{\cap}|_{\delta V^{-}} \text{ if } t \in \mathcal{E}_{\cap}|_{\Delta S_2^{-}} \wedge t \notin \mathcal{E}_{\cap}|_{\Delta S_1^{-}}\}$

$\qquad Au_{\cap}^{12}\ =\ \{t \text{ for } t \in \mathcal{E}_{\cap}|_{\delta V^{-}} \text{ if } t \in \mathcal{E}_{\cap}|_{\Delta S_1^{-}} \wedge t \in \mathcal{E}_{\cap}|_{\Delta S_2^{-}}\}$

Figure 5.1: View update templates for ρ-,$\cup$-,$\setminus$-,$\cap$-queries (Setup A)

$\underline{\sigma\text{-}rule}\ V(\vec{x})\ :-\ S(\vec{x})\ ,v_1\ \oplus_1\ c_1\ ,\ \ldots\ ,v_m\ \oplus_m\ c_m.$ where $\{v_1\ ,\ \ldots\ ,v_m\} \subseteq set(\vec{x})$

$(\text{SDR})^{\sigma} : \Delta S^{+}, \Delta S^{-}$

$\Delta S^{-}(\vec{x})\ :-\ \delta V^{-}(\vec{x})\ ,\ S(\vec{x}).$

$\Delta S^{+}(\vec{x})\ :-\ \delta V^{+}(\vec{x})\ ,\ \neg\, S(\vec{x}).$

$\Delta S^{+}(\vec{x^r})\ :-\ \delta V^{-}(\vec{x})\ ,\ S(\vec{x})\ ,\ Au^{0}_{\sigma}(\vec{y^r_\sigma})\ ,\ \neg\, S(\vec{x^r}).$

$\Delta S^{+}(\vec{x^r})\ :-\ \delta V^{-}(\vec{x})\ ,\ S(\vec{x})\ ,\ Au^{1}_{\sigma}(\vec{x^r})\ ,\ \neg\, S(\vec{x^r}).$

where $\vec{x^r} = ara(\vec{x}, \vec{y_\sigma}, \vec{y^r_\sigma}), \vec{y_\sigma} = \langle u_1, \ldots, u_k\rangle, \vec{y^r_\sigma} = \langle u^r_1, \ldots, u^r_k\rangle$

$\langle u_1, \ldots, u_k\rangle = tuple(set(\{v_1, \ldots, v_m\}))$

$(\text{AR})^{\sigma} : Au^{*}_{\sigma} \in \{Au^{0}_{\sigma}, Au^{1}_{\sigma}\}$

$Au^{0}_{\sigma}\ =\ \{\Pi_{pos(\vec{y_\sigma},\vec{x})} t \text{ for } t \in D_\sigma\}$

$Au^{1}_{\sigma}\ =\ D_\sigma$

where $D_\sigma = \{t \text{ for } t \in \mathcal{E}_\sigma|_{\Delta S^{+}} \text{ if } \Pi_{pos(ad(\vec{x},\vec{y_\sigma}),\vec{x})} t \in \Pi_{pos(ad(\vec{x},\vec{y_\sigma}),\vec{x})}(\mathcal{E}_\sigma|_{\Delta S^{-}})\}$

$(\text{FR})^{\sigma} : Fr$

$Fr(\text{“\_r\_"})\ :-\ V^{*}(awa(\vec{x}, \{v_i\})), v_i \neg \oplus_i\ c_i.$

where $V^{*} \in \{V, V'\}, i \in \{1, \ldots, m\}$

$\underline{\pi\text{-}rule}\ V(\vec{x})\ :-\ S(\vec{y}).$ where $set(\vec{x}) \subsetneq set(\vec{y})$

$(\text{SDR})^{\pi} : \Delta S^{+}, \Delta S^{-}$

$\Delta S^{-}(\vec{y})\ :-\ \delta V^{-}(\vec{x})\ ,\ S(\vec{y}).$

$\Delta S^{+}(\vec{y})\ :-\ \delta V^{+}(\vec{x})\ ,\ Au^{0}_{\pi}(ad(\vec{y}, \vec{x}))\ ,\ \neg\, S(awa(\vec{y}, \vec{x})).$

$\Delta S^{+}(\vec{y})\ :-\ \delta V^{+}(\vec{x})\ ,\ Au^{1}_{\pi}(\vec{y})\ ,\ \neg\, S(awa(\vec{y}, \vec{x})).$

$\Delta S^{+}(\vec{y})\ :-\ \delta V^{+}(\vec{x})\ ,\ Au^{0}_{\pi}(ad(\vec{y}, \vec{x}))\ ,\ \neg\, Au^{1}_{\pi}(awa(\vec{y}, \vec{x}))\ ,\ \neg\, S(awa(\vec{y}, \vec{x})).$

$(\text{AR})^{\pi} : Au^{*}_{\pi} \in \{Au^{0}_{\pi}, Au^{1}_{\pi}\}$

$Au^{0}_{\pi}\ =\ \{most\_common(\langle \Pi_{pos(ad(\vec{y},\vec{x}),\vec{y})} t \text{ for } t \in D_\pi\rangle)\}$

$Au^{1}_{\pi}\ =\ \{t \text{ for } t \in D_\pi \text{ if } \Pi_{pos(ad(\vec{y},\vec{x}),\vec{y})} t \notin Au^{0}_{\pi}\}$

where $D_\pi = \{t \text{ for } t \in \mathcal{E}_\pi|_{\Delta S^{+}} \text{ if } \Pi_{pos(\vec{x},\vec{y})} t \in \mathcal{E}_\pi|_{\delta V^{+}}\}$

$\underline{\bowtie\text{-}rule}\ V(\vec{x})\ :-\ S_1(\vec{y_1})\ ,\ S_2(\vec{y_2}).$ where $set(\vec{y_1}) \cup set(\vec{y_2}){=}set(\vec{x})$

$(\text{SDR})^{\bowtie} : \Delta S^{+}_{i}, \Delta S^{-}_{i}, i \in \{1, 2\}$ | and $set(\vec{y_1}) \cap set(\vec{y_2}) \neq \emptyset$

$\Delta S^{+}_{i}(\vec{y_i})\ :-\ \delta V^{+}(awa(\vec{x}, \vec{y_i}))\ ,\ \neg\, S_i(\vec{y_i}).$

$\Delta S^{-}_{i}(\vec{y_i})\ :-\ \delta V^{-}(awa(\vec{x}, \vec{y_i}))\ ,\ S_i(\vec{y_i}).$

$\Delta S^{-}_{i}(\vec{y_i})\ :-\ \delta V^{-}(awa(\vec{x}, \vec{y_i}))\ ,\ S_i(\vec{y_i})\ ,\ Au^{*}_{\bowtie}(awa(\vec{x}, \vec{y_i})).$

$\Delta S^{-}_{i}(\vec{y_i})\ :-\ \delta V^{-}(awa(\vec{x}, \vec{y_i}))\ ,\ S_i(\vec{y_i})\ ,\ \neg\, Au^{*}_{\bowtie}(awa(\vec{x}, \vec{y_i})).$

$(\text{AR})^{\bowtie} : Au^{*}_{\bowtie} \in \{Au^{1}_{\bowtie}, Au^{2}_{\bowtie}, Au^{12}_{\bowtie}\}$

$Au^{1}_{\bowtie}\ =\ \{t \text{ for } t \in \mathcal{E}_{\bowtie}|_{\delta V^{-}} \text{ if } \Pi_{pos(\vec{y_1},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{1}} \wedge \Pi_{pos(\vec{y_2},\vec{x})} t \notin \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{2}}\}$

$Au^{2}_{\bowtie}\ =\ \{t \text{ for } t \in \mathcal{E}_{\bowtie}|_{\delta V^{-}} \text{ if } \Pi_{pos(\vec{y_2},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{2}} \wedge \Pi_{pos(\vec{y_1},\vec{x})} t \notin \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{1}}\}$

$Au^{12}_{\bowtie}\ =\ \{t \text{ for } t \in \mathcal{E}_{\bowtie}|_{\delta V^{-}} \text{ if } \Pi_{pos(\vec{y_1},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{1}} \wedge \Pi_{pos(\vec{y_2},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{2}}\}$

$(\text{FR})^{\bowtie} : Fr$

$Fr(\text{"\_r\_"})\ :-\ \Delta S^{+}_{i}(awa(\vec{y_i}, \vec{y_{3-i}}))\ ,\ S_i(awa(\vec{y_i}, \vec{y_{3-i}})).$

$Fr(\text{"\_r\_"})\ :-\ \Delta S^{-}_{i}(awa(\vec{y_i}, \vec{y_{3-i}}))\ ,\ S'_i(awa(\vec{y_i}, \vec{y_{3-i}})).$

$\underline{\times\text{-}rule}\ V(\vec{x})\ :-\ S(\vec{y})\ ,v_1{=}c_1\ ,\ \ldots\ ,v_m{=}c_m.$ where $set(\vec{y}) \cup \{v_1, \ldots, v_m\}{=}set(\vec{x})$

$(\text{SDR})^{\times} : \Delta S^{+}, \Delta S^{-}$ | and $set(\vec{y}) \cap \{v_1, \ldots, v_m\}{=}\emptyset$

$\Delta S^{+}(\vec{y})\ :-\ \delta V^{+}(awa(\vec{x}, \vec{y}))\ ,\ \neg\, S(\vec{y}).$

$\Delta S^{-}(\vec{y})\ :-\ \delta V^{-}(awa(\vec{x}, \vec{y}))\ ,\ S(\vec{y}).$

$(\text{FR})^{\times} : Fr$

$Fr(\text{“\_r\_"})\ :-\ V^{*}(awa(\vec{x}, \{v_i\}))\ ,\ v_i \neq c_i.$

where $V^{*} \in \{V, V'\}, i \in \{1, \ldots, m\}$

Figure 5.2: View update templates for σ-,π-,$\bowtie$-,$\times$-queries (Setup A)

$\underline{\rho\text{-}rule}\; V(\vec{x}) \;:-\; S(p(\vec{x})).$

$(\mathrm{SDR})^{\rho} : \Delta S^{+}, \Delta S^{-}$

$$\begin{aligned}
\Delta S^{+}(p(\vec{x})) &:- V'(\vec{x}) \,,\, \neg\, S(p(\vec{x})).\\
\Delta S^{-}(p(\vec{x})) &:- \neg\, V'(\vec{x}) \,,\, S(p(\vec{x})).
\end{aligned}$$

$\underline{\cup\text{-}rule}\; V(\vec{x}) \;:-\; S_1(\vec{x}) \;;\; S_2(\vec{x}).$

$(\mathrm{SDR})^{\cup} : \Delta S_i^{+}, \Delta S_i^{-}, i \in \{1,2\}$

$$\begin{aligned}
\Delta S_i^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_i(\vec{x}).\\
\Delta S_i^{+}(\vec{x}) &:- V'(\vec{x}) \,,\, \neg\, S_i(\vec{x}) \,,\, \neg\, S_{3-i}(\vec{x}).\\
\Delta S_i^{+}(\vec{x}) &:- V'(\vec{x}) \,,\, \neg\, S_i(\vec{x}) \,,\, \neg\, S_{3-i}(\vec{x}) \,,\, Au_{\cup}^{*}(\vec{x}).\\
\Delta S_i^{+}(\vec{x}) &:- V'(\vec{x}) \,,\, \neg\, S_i(\vec{x}) \,,\, \neg\, S_{3-i}(\vec{x}) \,,\, \neg\, Au_{\cup}^{*}(\vec{x}).
\end{aligned}$$

$(\mathrm{AR})^{\cup} : Au_{\cup}^{*} \in \{Au_{\cup}^{1}, Au_{\cup}^{2}, Au_{\cup}^{12}\}$

$$\begin{aligned}
Au_{\cup}^{1} &= \{t \text{ for } t \in \mathcal{E}_{\cup}|_{V'} - \mathcal{E}_{\cup}|_{V} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S_1^{+}} \wedge t \notin \mathcal{E}_{\cup}|_{\Delta S_2^{+}}\}\\
Au_{\cup}^{2} &= \{t \text{ for } t \in \mathcal{E}_{\cup}|_{V'} - \mathcal{E}_{\cup}|_{V} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S_2^{+}} \wedge t \notin \mathcal{E}_{\cup}|_{\Delta S_1^{+}}\}\\
Au_{\cup}^{12} &= \{t \text{ for } t \in \mathcal{E}_{\cup}|_{V'} - \mathcal{E}_{\cup}|_{V} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S_1^{+}} \wedge t \in \mathcal{E}_{\cup}|_{\Delta S_2^{+}}\}
\end{aligned}$$

$\underline{\setminus\text{-}rule}\; V(\vec{x}) \;:-\; S_1(\vec{x}) \,,\, \neg\, S_2(\vec{x}).$

$(\mathrm{SDR})^{\setminus} : \Delta S_i^{+}, \Delta S_i^{-}, i \in \{1,2\}$

$$\begin{aligned}
\Delta S_1^{+}(\vec{x}) &:- V'(\vec{x}) \,,\, \neg\, S_1(\vec{x}).\\
\Delta S_2^{-}(\vec{x}) &:- V'(\vec{x}) \,,\, S_2(\vec{x}).\\
\Delta S_1^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_1(\vec{x}) \,,\, \neg\, S_2(\vec{x}).\\
\Delta S_2^{+}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, \neg\, S_2(\vec{x}) \,,\, S_1(\vec{x}).\\
\Delta S_1^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_1(\vec{x}) \,,\, \neg\, S_2(\vec{x}) \,,\, Au_{\setminus}^{*}(\vec{x}).\\
\Delta S_1^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_1(\vec{x}) \,,\, \neg\, S_2(\vec{x}) \,,\, \neg\, Au_{\setminus}^{*}(\vec{x}).\\
\Delta S_2^{+}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, \neg\, S_2(\vec{x}) \,,\, S_1(\vec{x}) \,,\, Au_{\setminus}^{*}(\vec{x}).\\
\Delta S_2^{+}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, \neg\, S_2(\vec{x}) \,,\, S_1(\vec{x}) \,,\, \neg\, Au_{\setminus}^{*}(\vec{x}).
\end{aligned}$$

$(\mathrm{AR})^{\setminus} : Au_{\setminus}^{*} \in \{Au_{\setminus}^{1}, Au_{\setminus}^{2}, Au_{\setminus}^{12}\}$

$$\begin{aligned}
Au_{\setminus}^{1} &= \{t \text{ for } t \in \mathcal{E}_{\setminus}|_{V} - \mathcal{E}_{\setminus}|_{V'} \text{ if } t \in \mathcal{E}_{\setminus}|_{\Delta S_1^{-}} \wedge t \notin \mathcal{E}_{\setminus}|_{\Delta S_2^{+}}\}\\
Au_{\setminus}^{2} &= \{t \text{ for } t \in \mathcal{E}_{\setminus}|_{V} - \mathcal{E}_{\setminus}|_{V'} \text{ if } t \in \mathcal{E}_{\setminus}|_{\Delta S_2^{+}} \wedge t \notin \mathcal{E}_{\setminus}|_{\Delta S_1^{-}}\}\\
Au_{\setminus}^{12} &= \{t \text{ for } t \in \mathcal{E}_{\setminus}|_{V} - \mathcal{E}_{\setminus}|_{V'} \text{ if } t \in \mathcal{E}_{\setminus}|_{\Delta S_1^{-}} \wedge t \in \mathcal{E}_{\setminus}|_{\Delta S_2^{+}}\}
\end{aligned}$$

$\underline{\cap\text{-}rule}\; V(\vec{x}) \;:-\; S_1(\vec{x}) \,,\, S_2(\vec{x}).$

$(\mathrm{SDR})^{\cap} : \Delta S_i^{+}, \Delta S_i^{-}, i \in \{1,2\}$

$$\begin{aligned}
\Delta S_i^{+}(\vec{x}) &:- V'(\vec{x}) \,,\, \neg\, S_i(\vec{x}).\\
\Delta S_i^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_i(\vec{x}) \,,\, S_{3-i}(\vec{x}).\\
\Delta S_i^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_i(\vec{x}) \,,\, S_{3-i}(\vec{x}) \,,\, Au_{\cap}^{*}(\vec{x}).\\
\Delta S_i^{-}(\vec{x}) &:- \neg\, V'(\vec{x}) \,,\, S_i(\vec{x}) \,,\, S_{3-i}(\vec{x}) \,,\, \neg\, Au_{\cap}^{*}(\vec{x}).
\end{aligned}$$

$(\mathrm{AR})^{\cap} : Au_{\cap}^{*} \in \{Au_{\cap}^{1}, Au_{\cap}^{2}, Au_{\cap}^{12}\}$

$$\begin{aligned}
Au_{\cap}^{1} &= \{t \text{ for } t \in \mathcal{E}_{\cap}|_{V} - \mathcal{E}_{\cap}|_{V'} \text{ if } t \in \mathcal{E}_{\cap}|_{\Delta S_1^{-}} \wedge t \notin \mathcal{E}_{\cap}|_{\Delta S_2^{-}}\}\\
Au_{\cap}^{2} &= \{t \text{ for } t \in \mathcal{E}_{\cap}|_{V} - \mathcal{E}_{\cap}|_{V'} \text{ if } t \in \mathcal{E}_{\cap}|_{\Delta S_2^{-}} \wedge t \notin \mathcal{E}_{\cap}|_{\Delta S_1^{-}}\}\\
Au_{\cap}^{12} &= \{t \text{ for } t \in \mathcal{E}_{\cap}|_{V} - \mathcal{E}_{\cap}|_{V'} \text{ if } t \in \mathcal{E}_{\cap}|_{\Delta S_1^{-}} \wedge t \in \mathcal{E}_{\cap}|_{\Delta S_2^{-}}\}
\end{aligned}$$

Figure 5.3: View update templates for ρ-,$\cup$-,$\setminus$-,$\cap$-queries (Setup B)

$\underline{\sigma\text{-}rule}\ V(\vec{x})\ :-\ S(\vec{x})\ , v_1\ \oplus_1\ c_1\ ,\ \ldots\ , v_m\ \oplus_m\ c_m.\quad$ where $\{v_1\ ,\ \ldots\ , v_m\} \subseteq set(\vec{x})$

$(\mathrm{SDR})^{\sigma} : \Delta S^{+}, \Delta S^{-}$

$\Delta S^{-}(\vec{x})\ :-\ \neg\ V'(\vec{x})\ ,\ S(\vec{x})\ ,\ v_1\ \oplus_1\ c_1,\ \ldots,\ v_m\ \oplus_m\ c_m.$

$\Delta S^{+}(\vec{x})\ :-\ V'(\vec{x})\ ,\ \neg\ S(\vec{x})\ ,\ v_1\ \oplus_1\ c_1,\ \ldots,\ v_m\ \oplus_m\ c_m.$

$\Delta S^{+}(\vec{x^r})\ :-\ \neg\ V'(\vec{x})\ ,\ S(\vec{x})\ ,\ \neg\ S(\vec{x^r})\ ,\ Au^{0}_{\sigma}(\vec{y^r_\sigma})\ ,\ v_1\ \oplus_1\ c_1,\ \ldots,\ v_m\ \oplus_m\ c_m.$

$\Delta S^{+}(\vec{x^r})\ :-\ \neg\ V'(\vec{x})\ ,\ S(\vec{x})\ ,\ \neg\ S(\vec{x^r})\ ,\ Au^{1}_{\sigma}(\vec{x^r})\ ,\ v_1\ \oplus_1\ c_1,\ \ldots,\ v_m\ \oplus_m\ c_m.$

where $\vec{x^r} = ara(\vec{x}, \vec{y_\sigma}, \vec{y^r_\sigma}), \vec{y_\sigma} = \langle u_1, \ldots, u_k\rangle, \vec{y^r_\sigma} = \langle u^r_1, \ldots, u^r_k\rangle$

$\langle u_1, \ldots, u_k\rangle = tuple(set(\{v_1, \ldots, v_m\}))$

$(\mathrm{AR})^{\sigma} : Au^{*}_{\sigma} \in \{Au^{0}_{\sigma}, Au^{1}_{\sigma}\}$

$Au^{0}_{\sigma}\ =\ \{\Pi_{pos(\vec{y_\sigma},\vec{x})} t \text{ for } t \in D_\sigma\}$

$Au^{1}_{\sigma}\ =\ D_\sigma$

where $D_\sigma = \{t \text{ for } t \in \mathcal{E}_\sigma|_{\Delta S^{+}} \text{ if } \Pi_{pos(ad(\vec{x},\vec{y_\sigma}),\vec{x})} t \in \Pi_{pos(ad(\vec{x},\vec{y_\sigma}),\vec{x})}(\mathcal{E}_\sigma|_{\Delta S^{-}})\}$

$(\mathrm{FR})^{\sigma} : Fr$

$Fr(\text{``\_r\_''})\ :-\ V^{*}(awa(\vec{x}, \{v_i\})), v_i\ \neg\oplus_i\ c_i.$

where $V^{*} \in \{V, V'\}, i \in \{1, \ldots, m\}$

$\underline{\pi\text{-}rule}\ V(\vec{x})\ :-\ S(\vec{y}).\quad$ where $set(\vec{x}) \subsetneq set(\vec{y})$

$(\mathrm{SDR})^{\pi} : \Delta S^{+}, \Delta S^{-}$

$\Delta S^{-}(\vec{y})\ :-\ \neg\ V'(\vec{x})\ ,\ S(\vec{y}).$

$\Delta S^{+}(\vec{y})\ :-\ V'(\vec{x})\ ,\ \neg\ S(awa(\vec{y},\vec{x}))\ ,\ Au^{0}_{\pi}(ad(\vec{y},\vec{x})).$

$\Delta S^{+}(\vec{y})\ :-\ V'(\vec{x})\ ,\ \neg\ S(awa(\vec{y},\vec{x}))\ ,\ Au^{1}_{\pi}(\vec{y}).$

$\Delta S^{+}(\vec{y})\ :-\ V'(\vec{x})\ ,\ \neg\ S(awa(\vec{y},\vec{x}))\ ,\ Au^{0}_{\pi}(ad(\vec{y},\vec{x}))\ ,\ \neg\ Au^{1}_{\pi}(awa(\vec{y},\vec{x})).$

$(\mathrm{AR})^{\pi} : Au^{*}_{\pi} \in \{Au^{0}_{\pi}, Au^{1}_{\pi}\}$

$Au^{0}_{\pi}\ =\ \{most\_common(\langle \Pi_{pos(ad(\vec{y},\vec{x}),\vec{y})} t \text{ for } t \in D_\pi\rangle)\}$

$Au^{1}_{\pi}\ =\ \{t \text{ for } t \in D_\pi \text{ if } \Pi_{pos(ad(\vec{y},\vec{x}),\vec{y})} t \notin Au^{0}_{\pi}\}$

where $D_\pi = \{t \text{ for } t \in \mathcal{E}_\pi|_{\Delta S^{+}} \text{ if } \Pi_{pos(\vec{x},\vec{y})} t \in (\mathcal{E}_\pi|_{V'} - \mathcal{E}_\pi|_{V})\}$

$\underline{\bowtie\text{-}rule}\ V(\vec{x})\ :-\ S_1(\vec{y_1})\ ,\ S_2(\vec{y_2}).\quad$ where $set(\vec{y_1}) \cup set(\vec{y_2}) = set(\vec{x})$

$(\mathrm{SDR})^{\bowtie} : \Delta S^{+}_{i}, \Delta S^{-}_{i}, i \in \{1, 2\}\quad$ $|$ and $set(\vec{y_1}) \cap set(\vec{y_2}) \neq \emptyset$

$\Delta S^{+}_{i}(\vec{y_i})\ :-\ V'(awa(\vec{x},\vec{y_i}))\ ,\ \neg\ S_i(\vec{y_i}).$

$\Delta S^{-}_{i}(\vec{y_i})\ :-\ \neg\ V'(awa(\vec{x},\vec{y_i}))\ ,\ S_i(\vec{y_i})\ ,\ S_{3-i}(\vec{y_{3-i}}).$

$\Delta S^{-}_{i}(\vec{y_i})\ :-\ \neg\ V'(awa(\vec{x},\vec{y_i}))\ ,\ S_i(\vec{y_i})\ ,\ S_{3-i}(\vec{y_{3-i}})\ ,\ Au^{*}_{\bowtie}(awa(\vec{x},\vec{y_i})).$

$\Delta S^{-}_{i}(\vec{y_i})\ :-\ \neg\ V'(awa(\vec{x},\vec{y_i}))\ ,\ S_i(\vec{y_i})\ ,\ S_{3-i}(\vec{y_{3-i}})\ ,\ \neg\ Au^{*}_{\bowtie}(awa(\vec{x},\vec{y_i})).$

$(\mathrm{AR})^{\bowtie} : Au^{*}_{\bowtie} \in \{Au^{1}_{\bowtie}, Au^{2}_{\bowtie}, Au^{12}_{\bowtie}\}$

$Au^{1}_{\bowtie}\ =\ \{t \text{ for } t \in \mathcal{E}_{\bowtie}|_{V} - \mathcal{E}_{\bowtie}|_{V'} \text{ if } \Pi_{pos(\vec{y_1},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{1}} \wedge \Pi_{pos(\vec{y_2},\vec{x})} t \notin \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{2}}\}$

$Au^{2}_{\bowtie}\ =\ \{t \text{ for } t \in \mathcal{E}_{\bowtie}|_{V} - \mathcal{E}_{\bowtie}|_{V'} \text{ if } \Pi_{pos(\vec{y_2},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{2}} \wedge \Pi_{pos(\vec{y_1},\vec{x})} t \notin \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{1}}\}$

$Au^{12}_{\bowtie}\ =\ \{t \text{ for } t \in \mathcal{E}_{\bowtie}|_{V} - \mathcal{E}_{\bowtie}|_{V'} \text{ if } \Pi_{pos(\vec{y_1},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{1}} \wedge \Pi_{pos(\vec{y_2},\vec{x})} t \in \mathcal{E}_{\bowtie}|_{\Delta S^{-}_{2}}\}$

$(\mathrm{FR})^{\bowtie} : Fr$

$Fr(\text{``\_r\_''})\ :-\ \Delta S^{+}_{i}(awa(\vec{y_i}, \vec{y_{3-i}}))\ ,\ S_i(awa(\vec{y_i}, \vec{y_{3-i}})).$

$Fr(\text{``\_r\_''})\ :-\ \Delta S^{-}_{i}(awa(\vec{y_i}, \vec{y_{3-i}}))\ ,\ S'_i(awa(\vec{y_i}, \vec{y_{3-i}})).$

$\underline{\times\text{-}rule}\ V(\vec{x})\ :-\ S(\vec{y})\ , v_1{=}c_1\ ,\ \ldots\ , v_m{=}c_m.\quad$ where $set(\vec{y}) \cup \{v_1, \ldots, v_m\} = set(\vec{x})$

$(\mathrm{SDR})^{\times} : \Delta S^{+}, \Delta S^{-}\quad$ $|$ and $set(\vec{y}) \cap \{v_1, \ldots, v_m\} = \emptyset$

$\Delta S^{+}(\vec{y})\ :-\ V'(\vec{x})\ ,\ \neg\ S(\vec{y})\ ,\ v_1 = c_1\ ,\ \ldots, v_m = c_m.$

$\Delta S^{-}(\vec{y})\ :-\ \neg\ V'(\vec{x})\ ,\ S(\vec{y})\ ,\ v_1 = c_1\ ,\ \ldots, v_m = c_m..$

$(\mathrm{FR})^{\times} : Fr$

$Fr(\text{``\_r\_''})\ :-\ V^{*}(awa(\vec{x}, \{v_i\}))\ ,\ v_i \neq c_i.$

where $V^{*} \in \{V, V'\}, i \in \{1, \ldots, m\}$

Figure 5.4: View update templates for σ-,π-,$\bowtie$-,$\times$-queries (Setup B)

(SDR)$^{\alpha}$, (AR)$^{\alpha}$ and (FR)$^{\alpha}$ are sets of template-based clauses for source delta, auxiliary and flag relations, respectively.
Each Au^{*}_{α} in (AR)$^{\alpha}$ is an auxiliary relation that holds necessary ground tuples/facts.
$\mathcal{E}_{\alpha}$ is an example of form $(T_S, T_{S'}, T_V, T_{V'})$ against an *α-rule*.
$\mathcal{E}_{\alpha}|_X$ contains tuples in X that can be calculated from $\mathcal{E}_{\alpha}$.
Fr is a flag relation that checks constraints imposed on the sources and views. $\langle$"_r_"$\rangle \in Fr$ expresses a rejection.
$\{\ldots\}$ and $\langle\ldots\rangle$ respectively represent a set and a tuple.
$set(X)$ and $tuple(X)$ respectively convert X to a set and a tuple.
If X is a tuple,
  $X[i]$ returns the i-th element,
  $index(v, X)$ returns the index of an element v,
  $most\_common(X)$ returns the most common element, in X.
If X and Y are tuples and $set(Y) \subseteq set(X)$, (*POSition function*) $pos(Y, X) := \langle index(v, X) \text{ for } v \in Y\rangle$.
If t is a tuple, $\Pi_{pos(Y,X)}t$ returns a new tuple by projecting t at $pos(Y, X)$.
If R is a set of tuples, $\Pi_{pos(Y,X)}R := \{\Pi_{pos(Y,X)}t \text{ for } t \in R\}$.
(*Argument function*) $arg(X)$ returns a tuple of variables appearing in literal/constraint/set/tuple X.
(*Argument-Difference function*) $ad(X, Y) := \langle v \text{ for } v \in arg(X) \text{ if } v \notin arg(Y)\rangle$
(*Argument-With-Anonymous function*) $awa(X, Y) := \langle v \text{ if } v \in arg(Y) \text{ else } \_ \text{ for } v \in arg(X)\rangle$
(*Argument-Replacing-Argument function*) $ara(X, V_1, V_2) := \langle v \text{ if } v \notin V_1 \text{ else } V_2[index(v, V_1)] \text{ for } v \in arg(X)\rangle$
if $set(V_1) \subseteq set(arg(X))$ and $|V_1| = |V_2|$
Example:
If $X = \langle v_0, v_1, v_2, v_3\rangle, Y = \langle v_2, v_3, v_0\rangle$, then $pos(Y, X) = \langle 2, 3, 0\rangle, \Pi_{pos(Y,X)}\{\langle \text{"}a\text{"}, \text{"}b\text{"}, \text{"}c\text{"}, \text{"}d\text{"}\rangle\} = \{\langle \text{"}c\text{"}, \text{"}d\text{"}, \text{"}a\text{"}\rangle\}$
If $X = \langle v_0, v_1, v_2, v_3\rangle, Y = \langle v_0, v_4\rangle$,
  then $arg(X) = \langle v_0, v_1, v_2, v_3\rangle, arg(Y) = \langle v_0, v_4\rangle, ad(X, Y) = \langle v_1, v_2, v_3\rangle, awa(X, Y) = \langle v_0, \_, \_, \_\rangle$
If $X = \langle v_0, v_1, v_2, v_3\rangle, V_1 = \langle v_1, v_3\rangle, V_2 = \langle v_4, v_5\rangle$, then $ara(X, V_1, V_2) = \langle v_0, v_4, v_2, v_5\rangle$

Figure 5.5: Explanation of notations used in the templates

- (AR)$^{\alpha}$ includes *possible* template facts of example-based auxiliary relations used in (SDR)$^{\alpha}$;
- (FR)$^{\alpha}$ includes *possible* template rules against a special flag relation Fr for verifying the imposed constraints.

While the clauses generated by templates in (AR)$^{\alpha}$ and (FR)$^{\alpha}$ are all fixed in a possible result, the rules generated by templates in (SDR)$^{\alpha}$ are selectable and only selected in the output by synthesis.

We examine how to design view update templates for a $\cup$-*rule* defining a union view V from two sources S_1 and S_2:

$$V(\vec{x}) \;:-\; S_1(\vec{x}) \;;\; S_2(\vec{x}).$$

Based on the existing work on updating the union view [23], we know a set of view update strategies, informally described as follows:

$\cup$1 If a new tuple $\vec{t}$ is inserted into V, then tuple $\vec{t}$ should (1) appear in S_1, (2) or appear in S_2, (3) or appear in both S_1 and S_2.

∪2 If an available tuple $\vec{t}$ is deleted from V, then tuple $\vec{t}$ should disappear from both S_1 and S_2.

Each template rule in (SDR)$^\cup$ describes a view update strategy. We encode the strategy ∪2 by the first template rule of (SDR)$^\cup$ in Figure 5.1 of Setup A:

$$r_1^A \quad \Delta S_i^-(\vec{x}) \quad :- \quad \delta V^-(\vec{x}) \,,\ S_i(\vec{x}).$$

which means that if $\vec{x} \in \delta V^-$ (i.e., $\vec{x}$ is deleted from V), then $\vec{x} \in \Delta S_1^-$ if $\vec{x} \in S_1$ (i.e., $\vec{x}$ disappears from S_1), and $\vec{x} \in \Delta S_2^-$ if $\vec{x} \in S_2$ (i.e., $\vec{x}$ disappears from S_2).

The strategy ∪2 can be also encoded in the first rules of (SDR)$^\cup$ in Figure 5.3 of Setup B:

$$r_1^B \quad \Delta S_i^-(\vec{x}) \quad :- \quad \neg V'(\vec{x}) \,,\ S_i(\vec{x}).$$

which means that if $\vec{x} \notin V'$ and $\vec{x} \in S_i$, then $\vec{x} \in \Delta S_i^-$ (i.e., $\vec{x}$ disappears from S_i). Since $\vec{x} \in S_i$, according to the definition of the ∪-*rule* (i.e., $V(\vec{x}) : -S_1(\vec{x}); S_2(\vec{x})$), we can infer that $\vec{x} \in V$. But since $\vec{x} \notin V'$, $\vec{x}$ is deleted from V.

We see that both templates r_1^A and r_1^B represent the semantic of the strategy ∪2, but they are written in different ways. In r_1^A, if we rewrite $\delta V^-(\vec{x})$ by $V(\vec{x}), \neg V'(\vec{x})$ and eliminate redundant literals based on the meaning of the ∪ query, then we obtain r_1^B. In r_1^B, if we use the meaning of the ∪ query, we can extend the rule with an occurrence of $V(\vec{x})$, then replace $V(\vec{x}), \neg V'(\vec{x})$ with $\delta V^-(\vec{x})$, and finally we obtain r_1^A. Other pairs of corresponding rules in two Setups A and B also exhibit similarity if we rewrite between $\delta V^-(\vec{x})$ and $V(\vec{x}), \neg V'(\vec{x})$, as well as between $\delta V^+(\vec{x})$ and $V'(\vec{x}), \neg V(\vec{x})$.

If a new tuple $\vec{x}$ is inserted into V (i.e., $\vec{x} \in \delta V^+$), $\vec{x}$ could be either inserted into S_1 (i.e., $\vec{x} \in \Delta S_1^+$) or inserted into S_2 (i.e., $\vec{x} \in \Delta S_2^+$) or inserted into both S_1 and S_2. There are three different update strategies for inserting a tuple $\vec{x}$ into V and the different inserted tuples could follow different strategies. The next three rules of (SDR)$^\cup$ in Figure 5.1 of Setup A encode different cases of combining these strategies for multiple insertions:

$$\begin{array}{llll} r_2^A & \Delta S_i^+(\vec{x}) & :- & \delta V^+(\vec{x}) \,,\ \neg\, S_i(\vec{x}). \\ r_3^A & \Delta S_i^+(\vec{x}) & :- & \delta V^+(\vec{x}) \,,\ Au_\cup^*(\vec{x}) \,,\ \neg\, S_i(\vec{x}). \\ r_4^A & \Delta S_i^+(\vec{x}) & :- & \delta V^+(\vec{x}) \,,\ \neg\, Au_\cup^*(\vec{x}) \,,\ \neg\, S_i(\vec{x}). \end{array}$$

where the first describes that all inserted tuples follow the same update strategy and the others use auxiliary relations $Au^*_{\cup}$ to cover the tuples following the most common strategy and other specific strategies (like an if-then-else statement). The auxiliary relations are defined in $(\mathrm{AR})^{\cup}$ using set comprehensions as:

$$\begin{aligned}
Au^1_{\cup} &= \{t \text{ for } t \in \mathcal{E}_{\cup}|_{\delta V^+} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S^+_1} \land t \notin \mathcal{E}_{\cup}|_{\Delta S^+_2}\} \\
Au^2_{\cup} &= \{t \text{ for } t \in \mathcal{E}_{\cup}|_{\delta V^+} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S^+_2} \land t \notin \mathcal{E}_{\cup}|_{\Delta S^+_1}\} \\
Au^{12}_{\cup} &= \{t \text{ for } t \in \mathcal{E}_{\cup}|_{\delta V^+} \text{ if } t \in \mathcal{E}_{\cup}|_{\Delta S^+_1} \land t \in \mathcal{E}_{\cup}|_{\Delta S^+_2}\}
\end{aligned}$$

where $\mathcal{E}_{\cup}|_X$ contains tuples in X that can be calculated from the given example $\mathcal{E}_{\cup}$. Informally, for tuples that are inserted into V, $Au^1_{\cup}$ holds tuples that are only inserted in S_1, $Au^2_{\cup}$ holds tuples that are only inserted in S_2, and $Au^{12}_{\cup}$ holds tuples that are inserted in both S_1 and S_2, which is close to the three mentioned strategies when inserting a tuple against V.

If in r^A_2, r^A_3, r^A_4, we rewrite $\delta V^+(\vec{x})$ by $V'(\vec{x}), \neg V(\vec{x})$ and eliminate redundant literals, we will obtain the corresponding rules of $(\mathrm{SDR})^{\cup}$ in Figure 5.3 of Setup B:

$$\begin{array}{lllll}
r^B_2 & \Delta S^+_i(\vec{x}) & :- & V'(\vec{x})\ ,\ \neg\ S_i(\vec{x})\ ,\ \neg\ S_{3-i}(\vec{x}). \\
r^B_3 & \Delta S^+_i(\vec{x}) & :- & V'(\vec{x})\ ,\ \neg\ S_i(\vec{x})\ ,\ \neg\ S_{3-i}(\vec{x})\ ,\ Au^*_{\cup}(\vec{x}). \\
r^B_4 & \Delta S^+_i(\vec{x}) & :- & V'(\vec{x})\ ,\ \neg\ S_i(\vec{x})\ ,\ \neg\ S_{3-i}(\vec{x})\ ,\ \neg\ Au^*_{\cup}(\vec{x}).
\end{array}$$

We next examine how to design view update templates against a σ-rule that defines a selection view V from a source S sharing the same schema and a selection condition f represented as a list of comparisons, as follows:

$$V(\vec{x}) \quad :- \quad S(\vec{x})\ ,\ v_1 \oplus_1 c_1\ ,\ \ldots\ ,\ v_m \oplus_m c_m.$$

where $\oplus_1, \ldots, \oplus_m$ are comparison operators, $c_1, \ldots, c_m$ are constants, $v_1, \ldots, v_m$ are variables occurring in $\vec{x}$, and some of the variables $v_1, \ldots, v_m$ may be the same (e.g. $V(v_0, v_1)\ :-\ S(v_0, v_1), v_0 > 0, v_0 < 10.$).

Based on the existing work on updating a selection view with minimal effects [21, 22, 23], we know the following informal minimal-effect view update strategies:

(σ1) If a new tuple $\vec{t}$, which satisfies that $f(\vec{t})$ is true, is inserted into V, then $\vec{t}$ should appear in S'.

(σ2) If an available tuple $\vec{t}$, which satisfies that $f(\vec{t})$ is true, is deleted from V, then we should (1) either delete tuple $\vec{t}$ from S, (2) or modify attributes of tuple $\vec{t}$ that are related to f to obtain $\vec{t^*}$ such that $f(\vec{t^*})$ is false, and replace $\vec{t}$ by $\vec{t^*}$ in S.

The template rules in (SDR)$^\sigma$ of Figure 5.4 of Setup B encode these strategies. The strategy (σ1) is encoded as template as below:

$$r_5^B \quad \Delta S^+(\vec{x}) \quad :- \quad V'(\vec{x})\ ,\ \neg\, S(\vec{x})\ ,\ v_1 \oplus_1 c_1\ ,\ \ldots\ ,\ v_m \oplus_m c_m.$$

which means that if $\vec{x}$ satisfies $f(\vec{x})$ being true, and $\vec{x}$ is in V' but not in S (i.e., $\vec{x} \in V' - V$, or in other words, $\vec{x}$ is inserted into V), then $\vec{x} \in \Delta S^+$ (i.e., $\vec{x}$ is inserted into S).

We encode the first part of the strategy (σ2) as

$$r_6^B \quad \Delta S^-(\vec{x}) \quad :- \quad \neg\, V'(\vec{x})\ ,\ S(\vec{x})\ ,\ v_1 \oplus_1 c_1\ ,\ \ldots\ ,\ v_m \oplus_m c_m.$$

which means that if $\vec{x}$ satisfies $f(\vec{x})$ being true, and $\vec{x}$ is in S but not in V' (i.e., $\vec{x} \in V - V'$ or $\vec{x}$ is deleted from V), then $\vec{x} \in \Delta S^-$ (i.e., $\vec{x}$ is deleted from S).

The last two rules of (SDR)$^\sigma$ are related to encoding the second part of the strategy (σ2):

$$r_7^B \quad \Delta S^+(\vec{x^r}) \quad :- \quad \neg\, V'(\vec{x})\ ,\ S(\vec{x})\ ,\ \neg\, S(\vec{x^r})\ ,\ Au_\sigma^0(\vec{y_\sigma^r})\ ,\ v_1 \ \oplus_1\ c_1,\ \ldots,\ v_m\ \oplus_m\ c_m.$$
$$r_8^B \quad \Delta S^+(\vec{x^r}) \quad :- \quad \neg\, V'(\vec{x})\ ,\ S(\vec{x})\ ,\ \neg\, S(\vec{x^r})\ ,\ Au_\sigma^1(\vec{x^r})\ ,\ v_1 \ \oplus_1\ c_1,\ \ldots,\ v_m\ \oplus_m\ c_m.$$

where
$\vec{x^r} = ara(\vec{x}, \vec{y_\sigma}, \vec{y_\sigma^r})$, $\vec{y_\sigma} = \langle u_1, \ldots, u_k \rangle$, $\vec{y_\sigma^r} = \langle u_1^r, \ldots, u_k^r \rangle$,
$\langle u_1, \ldots, u_k \rangle = tuple(set(\{v_1, \ldots, v_m\}))$.

The meanings of r_7^B and r_8^B are roughly as follows: If $\vec{x}$ satisfies that $f(\vec{x})$ being true, and $\vec{x}$ is in S but not in V' (i.e., $\vec{x}$ is deleted from V), then a new tuple $\vec{x^r}$, not in S and satisfying $f(\vec{x^r})$ being false (related to auxiliary relations Au_σ^0 and Au_σ^1), should be inserted into S.

Tuple x^r should be different from $\vec{x}$ in at least one of the positions participating in condition f, i.e., variables $v_1, \ldots, v_m$. If we denote $\vec{y_\sigma}$ as a tuple of different variables in f, and $\vec{y_\sigma^r}$ as another tuple of the same size as $\vec{y_\sigma}$ and with completely fresh variables, then $\vec{x^r}$ is formed by replacing $\vec{y_\sigma}$ in $\vec{x}$ with $\vec{y_\sigma^r}$ (denoted as $\vec{x^r} = ara(\vec{x}, \vec{y_\sigma}, \vec{y_\sigma^r})$).

We utilize the auxiliary relations Au_σ^0 and Au_σ^1 to retain data that cause $f(\vec{x^r})$ to be false. From the specified example $\mathcal{E}_\sigma$ of the form of $(T_S, T_{S'}, T_V, T_{V'})$, we can compute these auxiliary relations using set comprehensions, as outlined in (AR)$^\sigma$:

$$
\begin{array}{lcl}
Au_\sigma^0 & = & \{\Pi_{pos(\vec{y_\sigma},\vec{x})} t \text{ for } t \in D_\sigma\} \\
Au_\sigma^1 & = D_\sigma = & \{t \text{ for } t \in \mathcal{E}_\sigma|_{\Delta S^+} \\
 & & \quad \text{if } \Pi_{pos(ad(\vec{x},\vec{y_\sigma}),\vec{x})} t \in \Pi_{pos(ad(\vec{x},\vec{y_\sigma}),\vec{x})} (\mathcal{E}_\sigma|_{\Delta S^-})\}
\end{array}
$$

where $\mathcal{E}_\sigma|_X$ contains tuples in X that can be derived from $\mathcal{E}_\sigma$, $pos(\vec{y}, \vec{x})$ returns a tuple of positions of $\vec{y}$ in $\vec{x}$, $ad(\vec{x}, \vec{y})$ returns a tuple of variables that are in $\vec{x}$ but not in $\vec{y}$. Since $\vec{y_\sigma}$ contains different variables in the selection condition f, we have $pos(ad(\vec{x}, \vec{y_\sigma}), \vec{x})$ as a tuple of positions of non-selected variables in $\vec{x}$. Au_σ^1 is computed as a set of tuple $\vec{x^r}$ such that $\vec{x^r}$ is inserted into S and there exists a corresponding $\vec{x}$ that is deleted from S. Au_σ^0 is computed from Au_σ^1 by only retaining the data in selection positions. Depending on the situation in which the deleted tuples $\vec{x}$ against V follow different strategies, Au_σ^0 and Au_σ^1 (r_7^B and r_8^B) might be used reasonably.

Since the σ-rule introduces restrictions on data of the view V (i.e., $v_1 \oplus_1 c_1$, ... , $v_m \oplus_m c_m$), we may need some rules encoding these restrictions. As suggested by [16] which shares the idea of negative constraints introduced in [46], a truth constant *False*, denoted as $\perp$, can be used to express the restrictions. Experimentally, we found that ProSynth does not work well when performing the synthesis over the set of rules that contain zero-term literals like $\perp$. So we replace $\perp$ with a special relation Fr that contains only one attribute. Fr will flag a rejection with the value "*_r_*" if a false truth occurs. Rather than heavily interfering with a Datalog solver in order to immediately reject update propagation, we simply use Fr to monitor the rejection. Data can be restored to the original state if ⟨"_r_"⟩ is in Fr.

(FR)$^\sigma$ covers the rules for Fr against a σ-rule as follows:

$$
\begin{array}{llcl}
r_9^B & Fr(\text{“\_r\_”}) & :- & V^*(awa(\vec{x}, \{v_i\})), v_i \neg \oplus_i \ c_i. \\
(\sim & \perp & :- & V^*(awa(\vec{x}, \{v_i\})), v_i \neg \oplus_i \ c_i.)
\end{array}
$$

where $V^* \in \{V, V'\}$, $i \in \{1, \dots, m\}$, and $awa(\vec{x}, \vec{y})$ replaces variables in $\vec{x}$ that are not in $\vec{y}$ by anonymous variables. Rule r_9^B says that if there is a tuple that violates the imposed restrictions on V and V', Fr will flag a rejection with the value "_r_".

It will not be too difficult to turn the rules r_5^B, r_6^B, r_7^B, r_8^B, r_9^B, into the corresponding rules in Figure 5.2 of Setup A. With the fixed constraints in $(\mathrm{FR})^\sigma$, we may omit the list of comparisons in template rules in $(\mathrm{SDR})^\sigma$ for brevity.

For other atomic queries of type α, $(\mathrm{SDR})^\alpha$ and $(\mathrm{AR})^\alpha$ could be interpreted in similar ways as above. Using tuple comprehensions, we define some functions (e.g., *ad*, *awa*, *ara*) that compute specific tuples of variables to formally describe the body of template rules, which makes the rules safe and correct. $(\mathrm{FR})^\alpha$ has no clauses if no constraints are imposed on the corresponding source and view. For a *σ-rule*/×*-rule* r, the negation of each constraint in the body of r is used as a constraint in a flag rule of $(\mathrm{FR})^\sigma/(\mathrm{FR})^\times$. For a ⋈*-rule*, rejects are flagged if in the join positions, the inserted data are already in the original source or the deleted data are still in the update source. If there is no rejection during the evaluation of atomic programs, their combinations could be made without failure.

Example 5.1. Let us see an example of using templates of Setup A to generate clauses for $\mathrm{PA}_2 := \textsc{ProbA}((\{\texttt{customers}\}, \texttt{M}_2), \mathcal{E}_2, \{r_2^a\})$ in Example 4.6 where $\mathcal{E}_2$ and r_2^a are as follows:

$$\mathcal{E}_2 = (\{1.1b\}, \{4.1b\}, \{1.1b'\}, \{4.1b'\})$$

$$r_2^a \quad M_2(i, n, c) \quad :- \quad \texttt{customers}(i, n, c)\ ,\ c = \text{“}Tokyo\text{”}.$$

From $S = \{\texttt{customers}\}$ and $V = \texttt{M}_2$, we can generate fixed rules of programs switching state/delta-based data, including:

$$\begin{array}{lll}
\delta\texttt{M}_2^-(i, n, c) & :- & \texttt{M}_2(i, n, c)\ ,\ \neg\ \texttt{M}_2'(i, n, c). \\
\delta\texttt{M}_2^+(i, n, c) & :- & \texttt{M}_2'(i, n, c)\ ,\ \neg\ \texttt{M}_2(i, n, c). \\
\texttt{customers}'(i, n, c) & :- & \texttt{customers}(i, n, c)\ ,\ \neg\ \Delta\texttt{customers}^-(i, n, c) \\
 & & ;\ \Delta\texttt{customers}^+(i, n, c).
\end{array}$$

Since the $get_a = \{r_2^a\}$ is a *σ-rule*, we prepare selectable rules based on $(\mathrm{SDR})^\sigma$, including:

$$\begin{array}{lll} \Delta\mathtt{customers}^-(i,n,c) & :- & \delta\mathtt{M}_2^-(i,n,c)\ ,\ \mathtt{customers}(i,n,c). \\ \Delta\mathtt{customers}^+(i,n,c) & :- & \delta\mathtt{M}_2^+(i,n,c)\ ,\ \neg\ \mathtt{customers}(i,n,c). \end{array}$$

These two rules mean that a deletion/insertion of tuple $\langle i,n,c\rangle$ from the view $\mathtt{M}_2$ is reflected to a deletion/insertion of the same tuple to the source `customers`. Another update strategy is that if a tuple $\langle i,n,c\rangle$ is deleted from the view $\mathtt{M}_2$, we could replace tuple $\langle i,n,c\rangle$ in the source `customers` by $\langle i,n,c^r\rangle$ where c^r is a valid value not equal to `"Tokyo"` (i.e., c^r violates constraints of the *σ-rule*). By checking example $\mathcal{E}_2$[1] , there is no such tuple $\langle i,n,c\rangle$. In other words, auxiliary relations in $(\mathrm{AR})^\sigma$ hold no fact. Hence, we skip preparing the rules containing the auxiliary relations in $(\mathrm{SDR})^\sigma$.

Moreover, based on $(\mathrm{FR})^\sigma$, two fixed rules that express constraints imposing on the view $\mathtt{M}_2$ are generated as below:

$$\begin{array}{l} \mathtt{Fr("\_r\_")}\ :-\ \mathtt{M}_2\mathtt{(\_,\_,}c\mathtt{)}\ ,\ c \neq \mathtt{"Tokyo"}. \\ \mathtt{Fr("\_r\_")}\ :-\ \mathtt{M}_2'\mathtt{(\_,\_,}c\mathtt{)}\ ,\ c \neq \mathtt{"Tokyo"}. \end{array}$$

The relation `Fr` will flag a rejection if, in $\mathtt{M}_2$ or $\mathtt{M}_2'$, there is a tuple whose third attribute value is different from `"Tokyo"`.

So when considering PA_2, we generated candidate rules for put_a, consisting of 7 rules above and 1 rule in $get_a = \{r_2^a\}$. ▲

Example 5.2. Let us see an example of using templates of Setup B to generate clauses for $\mathrm{PA}_2 := \textsc{ProbA}((\{M_1\},M_2),\mathcal{E}_2,\{r_2^o\})$ in Example 4.9 where $\mathcal{E}_2$ and r_2^o are as follows:

$$\mathcal{E}_2 = (\{4.3a\},\{4.3b\},\{4.3a'\},\{4.3b'\})$$

$$r_2^o \quad M_2(v_0,v_1,v_2) \quad :- \quad M_1(v_0,v_1,v_2), v_2 = \text{“}T\text{”}.$$

We prepare candidate clauses for minimal-effect view update strategies against the

[1]If there exists such a replacement in $\mathcal{E}_2$, e.g., from `customers(“104","Mori",“Tokyo")` to `customers'(“104","Mori","Berlin")`, we can prepare more, e.g., $\Delta\mathtt{customers}^+(i,n,c^r)\ :-\ \delta\mathtt{M}_2^+(i,n,c), \mathtt{customers}(i,n,c), Au_\sigma^0(c^r), \neg\mathtt{customers}(i,n,c^r)$. where $Au_\sigma^0 \supseteq \{\langle\text{“}\mathtt{Berlin"}\rangle\}$.

σ-rule r_2^o as below:

$$
\begin{array}{ll}
r_4^o & Fr(\text{“\_r\_”}) :- M_2(\_,\_,v_2)\ ,\ v_2 \neq \text{“}T\text{”}. \\
r_5^o & Fr(\text{“\_r\_”}) :- M_2'(\_,\_,v_2)\ ,\ v_2 \neq \text{“}T\text{”}. \\
r_6^o & \Delta M_1^-(v_0,v_1,v_2) :- \neg M_2'(v_0,v_1,v_2)\ ,\ M_1(v_0,v_1,v_2)\ ,\ v_2 = \text{“}T\text{”}. \\
r_7^o & \Delta M_1^+(v_0,v_1,v_2) :- M_2'(v_0,v_1,v_2)\ ,\ \neg M_1(v_0,v_1,v_2)\ ,\ v_2 = \text{“}T\text{”}. \\
r_8^o & M_1'(v_0,v_1,v_2) :- M_1(v_0,v_1,v_2)\ ,\ \neg\Delta M_1^-(v_0,v_1,v_2)\ ;\ \Delta M_1^+(v_0,v_1,v_2).
\end{array}
$$

Rules r_4^o and r_5^o are generated from template r_9^B, while rules r_6^o and r_7^o are generated from templates r_5^B and r_6^B, respectively, and rule r_8^o is generated from template r_0^B. There are no rules generated from templates r_7^B and r_8^B since both Au_σ^0 and Au_σ^1 are empty ($\langle 1, b_1, T\rangle$ is deleted from M_2 but there is no $\langle 1, b_1, c_0\rangle$ with $c_0 \neq T$ in M_1' - see Table 4.3). If $\langle 2, b_2, T\rangle$ is not in M_2', and $\langle 2, b_2, T\rangle$ in M_1' is replaced by $\langle 2, b_2, F\rangle$, we may generate more clauses as follows:

$$
\begin{array}{lll}
\Delta M_1^+(v_0,v_1,v_2^r) :- & \neg\, M_2'(v_0,v_1,v_2)\ ,\ M_1(v_0,v_1,v_2)\ , \\
& \neg\, M_1(v_0,v_1,v_2^r)\ ,\ Au_\sigma^0(v_2^r), v_2 = \text{“}T\text{”}. \\
\Delta M_1^+(v_0,v_1,v_2^r) :- & \neg\, M_2'(v_0,v_1,v_2)\ ,\ M_1(v_0,v_1,v_2)\ , \\
& \neg\, M_1(v_0,v_1,v_2^r)\ ,\ Au_\sigma^1(v_0,v_1,v_2^r), v_2 = \text{“}T\text{”}. \\
Au_\sigma^0(F). & \\
Au_\sigma^1(2, b_2, F). &
\end{array}
$$

▲

5.2 Evaluation

We have implemented a prototype named SYNTHBX for Algorithm 1 in 7K lines of Python, which relaxes procedures related to functional dependencies (i.e., FORWARDPROPAGATEFDs and PREPAREPUTCANDCEFDs). The minimal-effect view update strategies introduced in the previous section are encoded in templates of Setup A. The templates are embedded inside PREPAREPUTCANDMEVUS to automatically generate candidate rules of view update programs. SYNTHBX internally uses PROSYNTH$^+$ as the unidirectional synthesizer and SOUFFLÉ as the underlying Datalog solver. We equipped SYNTHBX with a simple PREPAREGETCAND that can extract useful constants from inserted/deleted data and enumerate literals based on the number of given sources. The PrepareGetCand would be bypassed if the user provides a set of candidate rules of *get*.

Research questions

To evaluate SynthBX, we design experiments to answer the following questions:

Q1 How powerful is SynthBX to solve a variety of synthesis tasks from many different sources?

Q2 How efficient is SynthBX to synthesize only a component program and the whole program of (get, put)?

Q3 How sensitive is SynthBX to different example sizes?

Benchmark suite

We prepare 56 benchmarks from three different sources to perform the evaluation, representing various practical view update problems from textbooks, papers, online sites and real systems [20, 16].

B1 We adjusted 15 relational benchmarks from ProSynth [20] by reusing the given sets of candidate rules of *get*. We eliminated the recursive rules and created additional tables to form well-provided examples.

B2 We adapted 32 relational benchmarks from Birds [16], a framework for manually writing bidirectional programs on relations, by relaxing key constraints and reusing the provided *get*s. For each benchmark containing no example, we executed the corresponding written *put* on specific and/or random inputs to create necessary tables. Such a *put* is only used to create examples.

B3 We handcrafted 9 additional benchmarks derived from real-world view update tasks, where the views are described in Table 5.1. Some schemas and examples were adapted from common sample databases of AirBnB[2], Joomla[3], and Microsoft

[2]http://insideairbnb.com/get-the-data.html
[3]https://downloads.joomla.org/cms/joomla3

Table 5.1: Descriptions for handcrafted benchmarks in B3

Benchmark	Brief description of the view
tokyoac	active staffs or customers in Tokyo
open_port	open ports in a system
logging	a portion of log data
admin_post_cms	the posts published by admins on a Joomla site
blog_2020	the articles written in 2020 on a Joomla site
neighbour_airbnb	residential data of AirBnB
dw_ship	ship data extracted from a data warehouse
dw_survey	survey data extracted from a data warehouse
dw_sales	sales data extracted from a data warehouse

SQL Server[4]. A few other specifications were crafted from working with a Unix system. These benchmarks are other practical scenarios of bidirectional programs, where a manager, rather than directly interacting with a large database, can perform updates on a smaller view without compromising consistency between the database and the view. Of the nine handcrafted benchmarks, some have more input with nonempty sets of human-provided candidate rules for *get*, while others are not given such sets.

Experimental Setup

All experiments were run on a 2.6 GHz Intel Core i7 processor with 16 GB of 2400 MHz DDR4 running macOS Ventura 13.3. We independently performed 32 runs for each benchmark on the same example data and collected statistics of the experiments. The evaluation results are summarized in Table 5.2.

5.2.1 Q1: Power

To answer Q1 about the power of SynthBX, we check the outputs of 56 benchmarks. Table 5.2 shows that SynthBX successfully solves 52/56 ($\approx$ 91%) benchmarks where 14/15, 28/32, and 9/9 benchmarks are from B1, B2, and B3, respectively. Each obtained bidirectional program is consistent with the given example and two properties GetPut

[4]https://github.com/microsoft/sql-server-samples/tree/master/samples/databases

Table 5.2: Main experiment results [SynthBX]

B	#	Benchmark	Atomic Views	#Tuples			#Rules			#Programs		MeanSynthTime (s)		
				In/Out[1]	IS/DS[2]	IV/DV[3]	Sg/Cg[4]	DSg[5]	Sp/Cp[6]	SPair[7]	WNSg[8]	MSTget[9]	MSTput[10]	MSTtotal[11]
	#01	sql-01	π, $2\bowtie$	24/20	0/1	0/1	1/24	3	32/32	1	1	0.81	0.19	1.30
	#02	sql-02	2π, $\bowtie$	6/4	1/0	1/0	1/6	3	21/30	1	1	0.32	0.38	0.99
	#03	sql-03	4π, $2\bowtie$	8/3	1/2	1/1	1/33	6	65/65	1	1	4.97	0.42	5.72
	#04	sql-04	2ρ, 2π, $2\bowtie$, $\cup$	22/11	3/1	2/1	2/5	7	57/80	1	1	0.25	3.13	3.81
	#05	sql-05	ρ, 2π, $2\bowtie$	24/16	5/1	3/1	1/8	5	46/55	1	1	0.50	1.27	2.13
	#06	sql-06	-	27/9	1/1	1/1	2/6	-	-/-	-	-	0.28	-	-
	#07	sql-07	2π, $\bowtie$	16/6	3/2	3/2	1/19	3	38/38	1	1	1.45	0.22	1.97
B1	#08	sql-08	3ρ, 3π, $4\bowtie$	10/7	2/1	1/1	3/32	10	95/95	3	1	54.73	0.66	55.83
	#09	sql-09	ρ, 3π, $4\bowtie$	8/6	1/1	1/1	2/11	8	86/86	1	2	3.70	20.17	24.25
	#10	sql-10	3ρ, π, $2\bowtie$	17/16	6/0	3/0	2/49	6	54/54	1	1	59.23	0.35	59.95
	#11	sql-11	2ρ, 7π $7\bowtie$, $\cap$	35/32	3/1	2/1	4/45	12	127/175	12	1	127.76	2.50	130.85
	#12	sql-12	2ρ, 2π, $4\bowtie$, $\cup$	50/36	3/3	2/2	3/10	9	89/89	1	1	1.44	0.59	2.40
	#13	sql-13	π, $\bowtie$	32/19	4/2	4/3	1/4	2	24/24	1	1	0.26	0.14	0.67
	#14	sql-14	4ρ, 2π, $3\bowtie$, $\cup$	21/9	0/2	0/2	3/7	10	98/98	1	1	0.42	0.73	1.56
	#15	sql-15	2ρ, π, $3\bowtie$	65/55	5/0	1/0	2/87	6	59/59	2	1	300.13	0.50	301.04
	#01	cars_master	π	33/11	1/1	1/1	1/1	1	7/7	1	1	0.10	0.08	0.45
	#02	goodstudents	σ, π	13/5	2/3	2/3	1/1	2	17/17	1	1	0.10	0.12	0.53
	#03	luxuryitems	σ	8/4	2/1	2/1	1/1	1	9/9	1	1	0.10	0.09	0.46
	#04	usa_city	σ, π	672/600	2/2	2/2	1/1	2	17/17	1	1	0.12	0.14	0.59
	#05	ced	$\setminus$	33/21	8/0	5/3	1/1	1	8/8	1	1	0.10	0.08	0.46
	#06	residents1962	σ	14/10	3/2	3/2	1/1	1	11/11	1	1	0.10	0.09	0.48
	#07	employees	π, $\bowtie$	39/26	10/3	5/3	1/1	2	21/21	1	1	0.10	0.14	0.56
	#08	researchers	σ, 2π, $\cap$	20/19	4/2	2/1	1/1	4	35/35	1	1	0.11	0.23	0.70
	#09	retired	2π, $\setminus$	25/17	0/1	1/0	1/1	3	22/22	1	1	0.11	0.15	0.58
	#10	paramountmovies	σ, π	5/3	1/1	1/1	1/1	2	16/16	1	1	0.10	0.13	0.55
	#11	officeinfo	π	21/7	1/1	1/1	1/1	1	7/7	1	1	0.10	0.09	0.47
	#12	vw_brands	2π, $2\times$, $\cup$	35/13	5/3	5/3	2/2	5	50/56	1	1	0.10	2.16	2.63
	#13	residents	$2\times$, $2\cup$	42/14	5/8	5/6	3/3	4	44/56	1	1	0.11	2.35	2.81
	#14	bstudents	σ, 2π, $\bowtie$	15/12	3/1	2/1	1/1	4	36/36	1	1	0.10	0.23	0.66
	#15	all_cars	$\bowtie$	37/15	1/2	1/2	1/1	1	13/13	1	1	0.10	0.11	0.50
	#16	tracks1	$\bowtie$	19/9	4/5	3/4	1/1	1	-/-	-	-	0.11	-	-
B2	#17	tracks2	π	14/4	3/4	3/4	1/1	1	7/7	1	1	0.10	0.08	0.46
	#18	tracks3	σ	10/4	3/4	2/3	1/1	1	-/-	-	-	0.10	-	-
	#19	newpc	σ, π, $\bowtie$	69/46	6/3	3/3	1/1	3	29/29	1	1	0.10	0.19	0.63
	#20	activestudents	σ, π, $\bowtie$	17/8	2/1	1/1	1/1	3	28/28	1	1	0.10	0.18	0.59
	#21	vw_customers	2π, $\bowtie$	1275/637	0/1	0/1	1/1	3	27/27	1	1	0.13	0.26	0.81
	#22	measurement	2σ, $\cup$	80/30	5/0	5/0	2/2	3	31/41	1	1	0.11	1.27	1.71
	#23	ukaz_lok	$\times$	46/14	4/6	4/6	1/1	1	11/11	1	1	0.10	0.10	0.48
	#24	message	$2\times$, $\cup$	104/34	6/7	6/7	2/2	3	32/38	1	1	0.10	1.49	1.94
	#25	phonelist	$3\times$, $2\cup$	98/32	5/6	5/6	3/3	5	52/64	1	1	0.11	3.57	4.09
	#26	purchaseview	π, $\bowtie$	155/55	11/21	11/21	1/1	2	30/30	1	1	0.11	0.16	0.58
	#27	vehicle_view	π, $\bowtie$	36/17	5/4	4/4	1/1	2	22/22	1	1	0.11	0.17	0.61
	#28	outstanding_tasks	ρ, π, $\bowtie$	22/11	4/2	3/2	1/1	2	34/34	1	1	0.11	0.26	0.72
	#29	poi_view	2π, $\bowtie$	2499/1499	200/200	100/100	1/1	3	30/30	1	1	0.12	0.26	0.87
	#30	products	4ρ, π, $\times$, $2\bowtie$, $\setminus$, $\cup$	127/48	3/5	2/5	2/2	10	75/103	1	1	0.11	9.87	10.49
	#31	koncerty	ρ, $2\bowtie$	247/213	9/6	3/6	1/1	3	31/31	1	1	0.11	0.21	0.65
	#32	emp_view	A[0], -	-	-	-	-	-	-	-	-	-	-	-
	#01	tokyoac	2σ, 2π, $\cup$	15/9	2/2	2/2	2/2	5	44/50	1	1	0.10	1.38	1.85
	#02	open_port	σ, 4π, $2\bowtie$, $\cup$	927/913	4/2	4/2	3/152	8	84/84	4	2	64.99	0.27	65.68
	#03	logging	3σ, 2π, $\bowtie$, $2\cup$	240/210	4/8	4/8	4/112	8	81/103	14	2	105.97	21.12	128.03
	#04	admin_post_cms	5σ, 6π, $4\bowtie$, $3\cup$	49/41	6/3	4/3	7/100	18	147/195	10	9	38.21	12.71	51.46
B3	#05	blog_2020	4σ, π, $3\cup$	42/20	3/3	3/3	5/128	8	58/89	1	1	10.74	21.30	32.49
	#06	neighbour_airbnb	3σ, π, $2\cup$	294/250	4/7	4/7	5/87	8	57/84	1	1	9.50	7.06	16.96
	#07	dw_ship	4π, $2\bowtie$, $\cup$	2815/818	114/300	113/300	2/2	7	294/294	1	1	0.12	0.87	1.78
	#08	dw_survey	ρ, σ	154/146	29/4	29/4	2/5	2	17/17	1	1	0.34	0.14	0.75
	#09	dw_sales	4σ, 4π, $4\times$, $3\cup$	249/163	4/4	4/4	4/4	15	128/146	1	1	0.12	19.03	19.69

(0) Unsupported: A stands for aggregation.
(1) In/Out is the number of input/output tuples. (2) IS/DS and (3) IV/DV are the numbers of insertion/deletion tuples against the source and the view, respectively.
(4) Sg/Cg and (5) Sp/Cp are the numbers of rules in a solution/candidate program of *get* and *put*, respectively.
(6) DSg is the number of single-line rules in the decomposed solution program of *get*.
(7) SPair is the number of different well-behaved pairs synthesized in 32 runs.
(8) WNSg is the worst number of solutions of *get*s that are needed to synthesize before finding a well-behaved pair.
(9) MSTget, (10) MSTput and (11) MSTtotal are means of the runtime for successfully synthesizing only *get*, only *put* and the whole (get, put) respectively.

and PUTGET. The failed reasons are as follows. Benchmark `emp_view` (B2#32) contains an aggregation (i.e., use of aggregate functions such as *count*, *sum*, *min*, *max* to group data and summarize information) in the view definition. Benchmarks `sql-6` (B1#06), `tracks1` (B2#16), and `tracks3` (B2#18) include examples where non-minimal changes occur due to internal dependencies (such as functional dependencies) that impose extra constraints on the user-provided tables. The minimal-effect update strategies given in [21, 22, 23] do not cover these cases, resulting in a shortage of essential templates during the synthesis. While designing additional templates to support dependencies is more complex and requires more effort, it is still unknown how to encode view update strategies for aggregations as templates. The ambiguity of well-behaved update strategies against aggregations has not been extensively explored. Additionally, these strategies involve calculations outside the domain of relations (e.g., addition in the case of the aggregate function *sum*), leading to increased challenges in templatizing within Datalog.

All of the bidirectional programs that were successfully synthesized are free of dependencies and aggregations, and use various combinations of atomic views. We randomly picked up a combination in 32 runs to show in Table 5.2. Constraint strengthening to continue the synthesis occurs many times out of the 32 runs of some benchmarks, for instance, `sql-09` (B1#09) and `admin_post_cms` (B3#04). Furthermore, our equipped procedure PREPAREGETCAND has allowed SYNTHBX to synthesize well-behaved pairs for benchmarks in which the set of candidate rules of *get* is provided empty, as in B3.

5.2.2 Q2: Efficiency

To answer Q2 about the efficiency of SYNTHBX, we mostly observe the number of rules, the number of programs, and the means of synthesis times, which are listed in the last eight columns of Table 5.2.

Regardless of the mentioned failures, SYNTHBX is able to synthesize a well-behaved bidirectional program in an average of 19 seconds and less than 3 seconds for 37 benchmarks. Compared to taking a long time to write a well-behaved bidirectional program manually - possibly minutes or hours - a synthesis time of 19 seconds on average is relatively quick, making it feasible for real-world applications where timely results are essential. Additionally, synthesizing a program in less than 3 seconds for 37 benchmarks demonstrates high efficiency. This is valuable in scenarios where quickly creating programs is crucial, improving the overall usability and effectiveness of the approach. Furthermore, in [20], ProSynth proves its effectiveness by making the desired program in about 10 seconds per task, and for 28 of them, it takes less than a second. Despite the difference in environment setup (we use a 6-core Intel machine, whereas they use an 18-core Xeon server), the time metrics provided by both parties are appealing.

SynthBX is shown to be extremely effective to solve tasks in B2, where the programs of *get*s are given at the beginning. The synthesis time against *get* (`MSTget`) is insignificant, while the synthesis time against *put* (`MSTput`) is relatively small. For instance, consider benchmark `vw_brands` (B2#12), SynthBx finds a well-behaved pair of (get, put) in 2.63 seconds, with 0.10 seconds are for synthesizing *get*, 2.16 seconds for synthesizing *put*, and the rest for other I/O costs. SynthBx prepares 56 candidate rules (`Cp`) to make the search space for the synthesis of *put* and selects 50 of them (`Sp`) as the solution for *put*. Our templates keep the distance between the candidate and the solution of *put* not too far. For 33/52 successful benchmarks, SynthBx finds a solution of *put* almost immediately after checking the prepared candidate. For the rest, the search space for synthesizing *put* is not too large. The numbers of rules in the solution and candidate programs of *put* (`Sp` and `Cp`) look big but they actually count both selectable and fixed clauses.

If a *get* is not given, SynthBX would be affected by the performance of ProSynth^{+} which is invoked to synthesize a *get* then a suitable *put* too. For benchmark `sql-09` (B1#09), we prepared 11 candidate rules for *get* instead of its solution. SynthBX needs around 24 seconds to synthesize a well-behaved pair where 3.7 seconds and 20.17 seconds are, respectively, the synthesis time to its *get* and *put*. In 32 runs, SynthBX finds one solution pair (`SPair`) where the *get* contains 2 rules (`Sg`) in 11 candidate rules (`Cg`). In the worst case, SynthBX has to synthesize 2 programs of *get* (`WNSg`) to find a well-behaved pair. The more programs that do not match, the more time it takes. The synthesis times, `MSTget` and `MSTput`, are accumulated during the time that SynthBX synthesizes a *get* and a suitable *put*. In the other 32 runs of benchmark `admin_post_cms` (B3#04), SynthBX once had to synthesize 9 solution programs of *get* before obtaining a suitable *put*. For this benchmark, the number of atomic rules

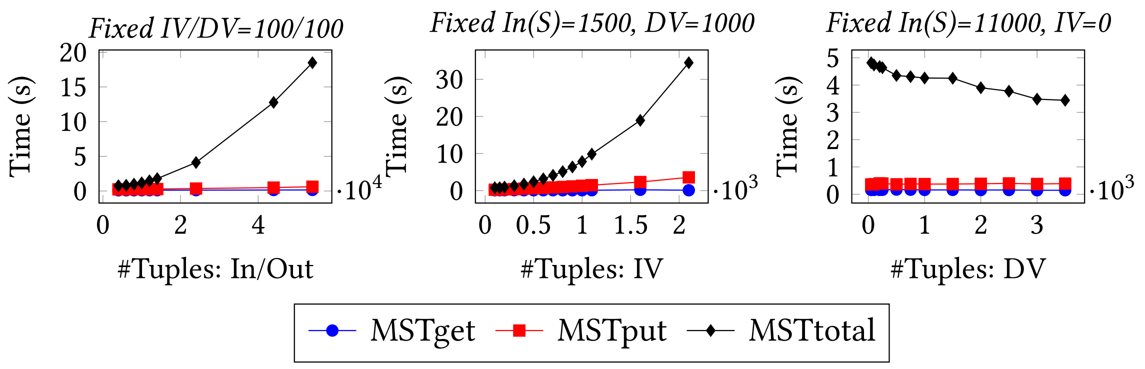


Figure 5.6: Stress tests for benchmark `poi_view` (B2#29)

after decomposing a *get* (DSg) is up to 18, the biggest for all benchmarks, and the average time to find a valid pair is up to 51.46 seconds. While we do not provide any candidates of *get* to B3#04, SynthBX automatically generates 100 candidate rules. Using a large number of candidate rules makes the synthesis space more diverse, but may require lots of time to search. It would be better if users had more intervention in generating and choosing candidates.

5.2.3 Q3: Sensitivity against example sizes

To answer Q3, we examine the sensitivity of successful benchmarks to the number of input/output tuples (`In`/`Out`) and the number of insertion/deletion tuples against the source and the view (IS/DS and IV/DV). The analysis is based on the synthesis times `MSTget` and `MSTput`, as well as the set of candidate rules of *get*.

If a user provides a *get* program or if the number of candidate rules of *get* is relatively small, and the given example is of reasonable size, the performance of SynthBX would not significantly depend on `In`, `Out`, IS, DS, IV and DV. For instance, SynthBX solves benchmarks `sql-02` (B1#02), `usa_city` (B2#04) and `poi_view` (B2#29) in less than 1 second. `sql-02` (B1#02) has `In`, `Out`, IS, DS, IV and DV all under 7, `poi_view` (B2#29) has typically larger corresponding numbers, while `usa_city` (B2#04) has medium `In` and `Out` (around 600 for each) but small IS, DS, IV and DV (all equal to 2).

We set up some stress tests on (B2#29) to check the sensitivity more carefully. Figure 5.6 illustrates the results in three cases in which we either (a) fix IV/DV, and vary `In`/`Out`; (b) fix `In` against the sources, fix DV, and vary IV; and (c) fix `In` against the sources, fix IV, and vary DV. The total synthesis time strongly depends on the number of tuples. This is due to the cost of I/O used to compute auxiliary facts as well as evaluate programs.

If no candidate rule of *get* is provided, SynthBX generates candidate rules using the schemas and constants from the given tables. The more tuples, the more constants, the more candidate rules, the longer the synthesis time, for instance, in benchmarks `open_port` (B3#02) and `logging` (B3#03).

5.2.4 Discussions

For the benchmark suite of 56 tasks, SynthBX successfully synthesizes 52 tasks. The synthesized programs not only satisfy the corresponding given examples but also adhere to well-behavedness. However, there were some failures because of the lack of templates for features such as dependencies and aggregations, which are related to more constraints and extra effects of update propagation. Additionally, other relational constraints such as domain constraints and inclusion constraints were not given adequate attention in our template design. We will continue to further develop templates to accommodate these constraints and effects.

SynthBX takes 19 seconds on average to successfully synthesize a solution and less than 3 seconds each for 37 out of 52 successful tasks. It is affected by the performance of ProSynth$^+$. SynthBX is more efficient for reasonable-sized examples.

At present, we have not addressed the issue of how closely the synthesized result matches the intended one, which is an essential direction for future work.

There is no tool to which we can make an apple-to-apple comparison. Existing synthesis tools *cannot* directly cope with both the well-behavedness of bidirectional programs and the complexity of relations as well as query languages. The base synthesizer inside SynthBX, ProSynth, *cannot* ensure the well-behavedness, while all of the existing synthesizers of bidirectional programs *do not* work on relations. Having bidirectional programs that are well-behaved and work with relations plays an important role in addressing practical view update problems in relational databases. As you can see in the benchmarks, there are many practical view update cases in which we need to maintain consistency across data in various relations. The well-behaved nature of bidirectional programs ensures consistency and any violations of well-behavedness result in a loss of data consistency.

Considering Example 3.2 (referred to as benchmark B3#01 and Table 1.1), we would like to synthesize a query program *get* and a view update program *put* such that they are well-behaved. ProSynth can be adapted to automatically synthesize $get = \{r_1, r_2\}$ and $put = \{r_3, r_4, r_5, r_6, r_7, r_8, f_9, f_{10}\}$ where r_1 and r_2 are in Example 4.5, while $r_3, \ldots, r_8$, f_9 and f_{10} are as follows:

```
r3    staffs′(i,n,c,a)   :−  staffs(i,c,n,a), a=“0".
r4    staffs′(i,n,c,a)   :−  staffs(i,c,n,a), c≠“Tokyo".
r5    staffs′(i,n,c,a)   :−  tokyoac′(n), Au1(i,n,c,a).
r6    customer′(i,n,c)   :−  customers(i,n,c), c≠“Tokyo".
r7    customer′(i,n,c)   :−  tokyoac′(n), customers(i,n,c).
r8    customer′(i,n,c)   :−  tokyoac′(n),  Au2(i,n,c).
f9    Au1(“14",“Shin",“Tokyo",“1").
f10   Au2(“105",“Yuri",“Tokyo").
```

Over `staffs` and `customers`, the query *get* defines a view `tokyoac` that includes the names of individuals living in Tokyo who are either active staff members or customers. The view update *put* reflects the changes against the `tokyoac` into the changes against `staffs` and `customers`.

If we evaluate the above *put* where `staffs` and `customers` are as original as in Table 1.1a and Table 1.1b, respectively, and `tokyoac′` = {⟨“Ken"⟩, ⟨“Mori"⟩}, then `staffs′` and `customers′` will be as follows:

staffs′

sid	name	city	active
10	Anna	Berlin	1
12	Jose	Rio	0
13	Yua	Tokyo	0

customers′

cid	name	city
100	Logan	Denver
101	Olsen	Oslo
103	Luis	Lisbon
104	Mori	Tokyo

Then, if we perform the query *get* over the two above tables, the obtained result will be {⟨“Mori"⟩}, which differs from the given `tokyoac′` = {⟨“Ken"⟩, ⟨“Mori"⟩}. This indicates a violation against the PutGet law, leading to a break in well-behavedness or consistency between (`staffs`, `customers`) and (`tokyoac`).

While ProSynth failed to provide the well-behavedness of bidirectional programs, other existing synthesis approaches of bidirectional programs do not operate on relations, rendering them unsuitable for application to the benchmarks in the experiments.

5.3 Summary

In this chapter, we present how to encode minimal-effect view update strategies in templates. With the templates, we can generate candidate rules expressing minimal-effect strategies for the view update program. We introduce two Setups A and B for designing templates. The corresponding template rules in these two setups have the similarity to express the meaning of a strategy, but are written in different ways.

We have implemented a prototype SynthBX that uses templates of Setup A. SynthBX skips the procedures related to FDs in our high-level algorithm. We evaluated SynthBX on 56 practical scenarios. The results showed that SynthBX is powerful and efficient for reasonable-sized examples.

However, SynthBX failed to synthesize certain tasks involving tables with internal functional dependencies, a scenario encountered in many real-world view update problems. To resolve that problem, we need to handle FDs like in the high-level algorithm, by forward-propagating FDs (as mentioned in Section 4.2), and by designing more templates that encode the constraints and effects of FDs to enrich the synthesis space. We will discuss these new templates in the next chapter.

6 Synthesizing View Update Programs with More Templates against FDs

In practice, tables with internal functional dependencies are quite common. FDs play an important role in practical view update strategies over these tables. FDs impose constraints on the tables, and when performing updates, they may cause extra effects that are non-minimal changes. If we do not handle the FDs carefully, we might miss important rules in the synthesis, causing failures. By forward-propagating FDs from the source, we have specified the FDs associated with all relations in the sub-synthesis problems. With this information of FDs, we can design templates encoding the constraints of FDs. For the effects of FDs in the atomic view update program, we will design other templates encoding that effects over the minimal templates.

The following sections are organized as follows. Section 6.1 and 6.2 present templatizing constraints and effects of FDs, respectively. Section 6.3 covers SynthBP, another prototype of our approach, which supports templates related to FDs, and includes an evaluation of SynthBP on 38 benchmarks. Section 6.4 summarizes this chapter.

6.1 Templatizing the Constraints of FDs

If a relation r has internal dependencies described by the FDs, both the original and updated data on r are *constrained* to agree on the FDs.

Example 6.1. Consider the source relation S where $S :: ABCD$ and S satisfies $\mathcal{F}_S = \{A \longrightarrow B, A \longrightarrow D\}$. For FD $A \longrightarrow B$, we can use the relation Fr to encode the constraints as follows:

$$\begin{array}{lll} r_1^c & Fr(\text{“\_r\_”}) & :- \quad S(v_1, v_2, \_, \_), S(v_1, v_2^x, \_, \_), v_2 \neq v_2^x. \\ r_2^c & Fr(\text{“\_r\_”}) & :- \quad S'(v_1, v_2, \_, \_), S'(v_1, v_2^x, \_, \_), v_2 \neq v_2^x. \end{array}$$

As explained in the Section 5.1, we can use $\perp$, a truth value of $False$, to express the constraints imposed on a relations, for instances,

$$\begin{array}{ll} \perp & :- \quad S(v_1, v_2, \_, \_), S(v_1, v_2^x, \_, \_), v_2 \neq v_2^x. \\ \perp & :- \quad S'(v_1, v_2, \_, \_), S'(v_1, v_2^x, \_, \_), v_2 \neq v_2^x. \end{array}$$

But we replace $\perp$ by $Fr(\text{“\_r\_”})$ because ProSynth does not work well with the zero-term literals like $\perp$.

Rule r_1^c means that a rejection will occur if there are two tuples in S such that the first attribute values are the same but the second attribute values are different. This is similar to the meaning of FD $A \longrightarrow B$ in $\mathcal{F}_S$. Rule r_2^c can be understood similarly, but for S', which shares the same schema as S. If $\mathcal{F}_S$ has many FDs, we need to prepare such pairs of flag rules for each single FD.

We can prepare a similar pair of flag rules for FD $A \longrightarrow D$ as below:

$$Fr(\text{“\_r\_”}) :- S(v_1, \_, \_, v_4), S(v_1, \_, \_, v_4^x), v_4 \neq v_4^x.$$
$$Fr(\text{“\_r\_”}) :- S'(v_1, \_, \_, v_4), S'(v_1, \_, \_, v_4^x), v_4 \neq v_4^x. \qquad \blacktriangle$$

Example 6.2. Consider the intermediate relation $M_1 :: ABC$ in Example 4.9. After forward propagation of FDs from the source, the schema of M_1 associated with FDs is as follows:

$$M_1(A:\mathbb{A}, B:\mathbb{B}, C:\mathbb{C}, \mathcal{F}_{M_1} = \{A \longrightarrow B\})$$

Similarly to Example 6.1, we can encode constraints of $\mathcal{F}_{M_1} = \{A \longrightarrow B\}$ by the following two flag rules:

$$\begin{array}{lll} Fr(\text{“\_}r\text{\_”}) & :- & M_1(v_1, v_2, \_, \_), M_1(v_1, v_2^x, \_, \_), v_2 \neq v_2^x. \\ Fr(\text{“\_}r\text{\_”}) & :- & M_1'(v_1, v_2, \_, \_), M_1'(v_1, v_2^x, \_, \_), v_2 \neq v_2^x. \end{array}$$

▲

Algorithm 3 describes how we can templatize the constraints of the FDs in NR-Datalog. TEMPLATIZECONSTRAINTFDs takes as input a set $\mathcal{S}_f$ of schemas of relations with computed FDs and produces as output a set C of flag rules describing the constraints of the FDs.

The procedure executes nested loops where the relation r iterates over $\mathcal{S}_f$, state s iterates over a set of original and updated states, and FD $X \longrightarrow A_k$ iterates over $\mathcal{F}_r$. Then, we prepare the variables in the body of the flag rule and build a complete rule expressing the constraint of the considered FD. Informally, in such a flag rule against FD f, we need to prepare two positive literals L_1 and L_2 that have the same variables in the positions on the *lhs* of f, different variables in the positions on the *rhs* of f, and anonymous variables for the remaining positions.

6.2 Templatizing the Effects of FDs

The *effects* of FDs could appear in $P_{\Delta S}$ (Setups A and B in Section 5.1), where an updated tuple in ΔS could conflict with an original tuple t of S on an FD. If such conflicts are not handled, the result S' would contain data inconsistencies on the FD. There are two strategies for resolving the conflict: (1) deleting t from S; and (2) revising t based on specified FDs with revision operators (Section 2.4). The former could be simply done by templatizing rules for adding the conflicting tuple t to ΔS^-. The latter is more complicated since revision operators are recursive structures with FDs that have never been defined in NR-Datalog*. Moreover, an NR-Datalog* program has no information on how the FDs are structured. We need to develop an intermediary that converts revision operators into NR-Datalog* clauses.

Algorithm 3: Templatizing Constraints of FDs

```
Input: $\mathcal{S}_f$: a set of schemas, each associated with a set of FDs
Output: $C$: a set of rules encoding the constraints of the associated FDs
1  procedure TemplatizeConstraintFDs($\mathcal{S}_f$):
2      $C \leftarrow \emptyset$
3      for $r :: A_1A_2 \ldots A_n \in \mathcal{S}_f$ do
4          for $s \in \{original, updated\}$ do
5              for $X \longrightarrow A_k \in \mathcal{F}_r$ do
6                  $ars_1, ars_2 \leftarrow [\_] * n, [\_] * n$
7                  for $i \in [1, \ldots, n]$ do
8                      if $A_i \in X$ then $ars_1[i], ars_2[i] \leftarrow v_i, v_i$
9                      if $A_i = A_k$ then $ars_1[i], ars_2[i] \leftarrow v_k, v_k^x$
10                 Add the following rule to $C$:
                     $Fr(\text{“\_r\_”}) :- r^s(ars_1), r^s(ars_2), v_k \neq v_k^x$.
11     return $C$
```

Assume that we have generated Datalog rules for $P_{\Delta S}$ from the base templates with minimal effects. We slightly rename the heads of the rules to ΔS_b^- and ΔS_b^+ as appropriate. We use $\Delta S_b \equiv (\Delta S_b^-, \Delta S_b^+)$ to hold the updated data with minimal effects and use ΔS to hold the updated data after accounting for the effects of the FDs. The templates encoding effects of the FDs will be designed over ΔS_b; i.e., we expect to obtain a put_a program that computes the updated data as usual and then resolves the conflicts.

Example 6.3. Consider templatizing the effects of FDs for the subproblem

$$\text{PA}_2 := \text{ProbA}((\{M_1\}, M_2), \mathcal{E}_2, \{r_2^o\})$$

in Example 4.9 where

$$\begin{aligned}
&M_1(A : \mathbb{A}, B : \mathbb{B}, C : \mathbb{C}, \mathcal{F}_{M_1} = \{A \longrightarrow B\})\\
&M_2(A : \mathbb{A}, B : \mathbb{B}, C, \mathcal{F}_{M_2} = \{A \longrightarrow B\})\\
&\mathcal{E}_2 = (\{4.3a\}, \{4.3b\}, \{4.3a'\}, \{4.3b'\})\\
&r_2^o \quad M_2(v_0, v_1, v_2) \ :-\ M_1(v_0, v_1, v_2), v_2 = \text{“}T\text{”}.
\end{aligned}$$

Let us recall all minimal-effect rules related to ΔM_1 that we generated for PA$_2$ in

Example 5.2:

$$r_6^o \quad \Delta M_1^-(v_0,v_1,v_2) :- \neg M_2'(v_0,v_1,v_2)\ ,\ M_1(v_0,v_1,v_2)\ ,\ v_2 = \text{“}T\text{”}.$$
$$r_7^o \quad \Delta M_1^+(v_0,v_1,v_2) :- M_2'(v_0,v_1,v_2)\ ,\ \neg M_1(v_0,v_1,v_2)\ ,\ v_2 = \text{“}T\text{”}.$$

First, we replace the basic rules (r_6^o, r_7^o), with a head of either ΔM_1^- or ΔM_1^+, with the following new ones, whose heads are adapted correspondingly to ΔM_{1b}^- or ΔM_{1b}^+:

$$r_1^e \quad \Delta M_{1b}^-(v_0,v_1,v_2) :- \neg M_2'(v_0,v_1,v_2)\ ,\ M_1(v_0,v_1,v_2)\ ,\ v_2 = \text{“}T\text{”}.$$
$$r_2^e \quad \Delta M_{1b}^+(v_0,v_1,v_2) :- M_2'(v_0,v_1,v_2)\ ,\ \neg M_1(v_0,v_1,v_2)\ ,\ v_2 = \text{“}T\text{”}.$$

The remaining minimal-effect candidate rules previously prepared are unchanged.

To encode a strategy that resolves the conflict on the FDs of the source ($\mathcal{F}_{M_1} = \{A \longrightarrow B\}$) by deletion, we generate more three rules r_3^e, r_4^e, r_5^e as follows:

$$r_3^e \quad M_{1d}^{\mathcal{F}}(v_0,v_1,v_2) :- \Delta M_{1b}^+(v_0,v_1^x,\_)\ ,\ M_1(v_0,v_1,v_2)\ ,\ v_1 \neq v_1^x.$$
$$r_4^e \quad \Delta M_1^-(v_0,v_1,v_2) :- \Delta M_{1b}^-(v_0,v_1,v_2).$$
$$r_5^e \quad \Delta M_1^-(v_0,v_1,v_2) :- M_{1d}^{\mathcal{F}}(v_0,v_1,v_2).$$

Rule r_3^e computes $M_{1d}^{\mathcal{F}}$ containing all tuples in M_1 that conflict with some basic inserted tuple to M_1 (in ΔM_{1b}^+) on FD $A \longrightarrow B$. There will be more rules like r_3^e if $\mathcal{F}_{M_1}$ has more FDs.

The next two rules say that a tuple in ΔM_1^- (i.e., a tuple that should be deleted from M_1) is in either ΔM_{1b}^- (which keeps the basic deletions) or $M_{1d}^{\mathcal{F}}$ (which keeps the conflicting tuples).

For the strategy that resolves the conflict on $\mathcal{F}_{M_1} = \{A \longrightarrow B\}$ by revision, we will generate more rules as follows:

$$r_6^e \quad M_{1i_0}^{\mathcal{F}}(v_0,v_1,v_2) :- M_1(v_0,v_1,v_2), \neg M_2(v_0,v_1,v_2).$$
$$r_7^e \quad M_{1i_1}^{\mathcal{F}}(v_0,v_1,v_2) :- M_{1i_0}^{\mathcal{F}}(v_0,v_1,v_2), \neg \Delta M_{1b}^+(v_0,\_,\_).$$
$$r_8^e \quad M_{1i_1}^{\mathcal{F}}(v_0,v_1,v_2) :- M_{1i_0}^{\mathcal{F}}(v_0,\_,v_2), \Delta M_{1b}^+(v_0,v_1,\_).$$
$$r_9^e \quad M_{1i_\#}^{\mathcal{F}}(v_0,v_1,v_2) :- M_{1i_1}^{\mathcal{F}}(v_0,v_1,v_2), v_2 = \text{“}T\text{”}, \neg M_2'(v_0,v_1,v_2)$$
$$r_{10}^e \quad \Delta M_1^+(v_0,v_1,v_2) :- \Delta M_{1b}^+(v_0,v_1,v_2).$$
$$r_{11}^e \quad \Delta M_1^+(v_0,v_1,v_2) :- \Delta M_{1i_1}^{\mathcal{F}}(v_0,v_1,v_2), \neg \Delta M_{1i_\#}^{\mathcal{F}}(v_0,v_1,v_2).$$

Rule r_6^e defines $M_{1i_0}^{\mathcal{F}}$ as keeping all tuples in the source M_1 that might need to be revised.

Then, for an FD f whose *lhs* is a root of the tree form of $\mathcal{F}_{M_1}$, we prepare a pair of rules with the same head whose name specifies a new relation, for instance, $M_{1i_1}^{\mathcal{F}}$, to compute the relation revision $M_{1i_0}^{\mathcal{F}} \leftarrow_f \Delta M_{1b}^{+}$ (i.e., the results of this relation revision are kept in $M_{1i_1}^{\mathcal{F}}$). More generally, we translate the recursive definition of the relation revision (at the end of Section 2.4) to a series of pairs of NR-Datalog* rules, where each pair calculates $M_{1i_j}^{\mathcal{F}} \leftarrow_f \Delta M_{1b}^{+}$ and assigns it to a new relation $M_{1i_{j+1}}^{\mathcal{F}}$ (where f is an FD in the rebuilt tree form with a root in the *lhs* of f, and $j = 0, 1, \ldots$).

Rules r_7^e and r_8^e indicate a pair of revisions where ΔM_{1b}^{+} is used to revise $M_{1i_0}^{\mathcal{F}}$ into $M_{1i_1}^{\mathcal{F}}$ on FD $f \equiv A \longrightarrow B$. Informally, a tuple $\langle v_0, v_1, v_2\rangle$ is kept in $M_{1i_1}^{\mathcal{F}}$ if either (rule r_7^e - without conflict) $\langle v_0, v_1, v_2\rangle$ is in $M_{1i_0}^{\mathcal{F}}$ while there is no tuple in ΔM_{1b}^{+} with the first attribute (A-position) value equal to v_0 or (rule r_8^e - with conflict) there are some tuples $\langle v_0, \_, v_2\rangle$ in $M_{1i_0}^{\mathcal{F}}$ and $\langle v_0, v_1, \_\rangle$ in ΔM_{1b}^{+} that have the same A-position value of v_0 (note that v_1 is a variable in the B-position and v_2 represents a variable not in the positions of the *lhs* or *rhs* of f). After finishing an FD f, the corresponding edge in tree form will be removed. We can remove a root of the tree form if that node is not in the *lhs* of any remaining FDs and rebuild the forest.

Rule r_9^e defines $M_{1i_{\#}}^{\mathcal{F}}$ as keeping all violations of PutGet. Supposing the results of the last relation revision in the previous step are in $M_{1i_1}^{\mathcal{F}}$, then $M_{1i_1}^{\mathcal{F}}$ contains tuples that have been revised with $\mathcal{F}_{M_1}$ following the tree form and should be inserted into M_1'; however, some tuples may violate PutGet [7].

The last two rules r_{10}^e and r_{11}^e describe two ways that a tuple $\langle v_0, v_1, v_2\rangle$ would be inserted into M_1. The former requires $\langle v_0, v_1, v_2\rangle$ to be in ΔM_{1b}^{+} as usual. The latter requires $\langle v_0, v_1, v_2\rangle$ to be in the last revised relation $M_{1i_1}^{\mathcal{F}}$ but not in the violation relation $M_{1i_{\#}}^{\mathcal{F}}$. ▲

Algorithm 4: Templatizing Effects of FDs

```
Input: $\mathcal{S}_f$: a set of schemas, each associated with a set of FDs
Input: $C_g$: a set of atomic rules
Output: $E$: a set of rules encoding the effects of the associated FDs
1  procedure TemplatizeEffectFDs($\mathcal{S}_f, C_g$):
2      $E \leftarrow \emptyset$
3      for $r = H :- B. \in C_g$ do
4          $V, Ss \leftarrow$ Relation($H$), Relation($B$)
5          for $S \in Ss$ do
6              if $\mathcal{F}_S = \emptyset$ then continue
7              $\Delta S_b^-, \Delta S_b^+ \leftarrow$ base deletion and insertion rules
8              // strategy: conflict => deletion
9              for $X \rightarrow A_k \in \mathcal{F}_S$ do
10                 $ars_1, ars_2 \leftarrow [v_1, \ldots, v_n], [\_] * n$
11                 for $i \in [1..n]$ do
12                     if $A_i \in X$ then $ars_2[i] \leftarrow v_i$
13                     if $A_i = A_k$ then $ars_2[i] \leftarrow v_k^x$
14                 Add to $E$ the rule: $S_d^{\mathcal{F}}(ars_1) :- \Delta S_b^+(ars_2), S(ars_1), v_k \neq v_k^x$.
15             Add to $E$ the rules: $\Delta S^-(ars_1) :- \Delta S_b^-(ars_1)$. and $\Delta S^-(ars_1) :- S_d^{\mathcal{F}}(ars_1)$.
16             // strategy: conflict => revision
17             $S_{i_0}^{\mathcal{F}} \leftarrow$ FreshName()
18             $ars_1 \leftarrow [v_1, \ldots, v_n]$
19             Add to $E$ the rule with head $S_{i_0}^{\mathcal{F}}(ars_1)$ that calculates tuples which might
                 need to be revised in $S$
20             $T_{\mathcal{F}} \leftarrow$ tree form of $\mathcal{F}_S$
21             while $roots(T_{\mathcal{F}}) \neq \emptyset$ do
22                 $root \leftarrow$ a node in $roots(T_{\mathcal{F}})$
23                 $children \leftarrow$ set of child nodes of $root$
24                 for $child \in children$ do
25                     $il, ir \leftarrow [index(a)$ for $a \in$ Value($root$)], [$index(a)$ for $a \in$
                         Value($right$)]
26                     $S_{i_1}^{\mathcal{F}} \leftarrow$ FreshName()
27                     $ars_2 \leftarrow [v_i$ if $i \in il$ else _ for $i \in [1..n]]$
28                     $ars_3 \leftarrow [\_$ if $i \in ir$ else $v_i$ for $i \in [1..n]]$
29                     $ars_4 \leftarrow [v_i$ if $i \in il \cup ir$ else _ for $i \in [1..n]]$
30                     Add to $E$ the following rules: $S_{i_1}^{\mathcal{F}}(ars_1) :- S_{i_0}^{\mathcal{F}}(ars_1), \neg\Delta S_b^+(ars_2)$.
                         and $S_{i_1}^{\mathcal{F}}(ars_1) :- S_{i_0}^{\mathcal{F}}(args_3), \Delta S_b^+(args_4)$.
31                     $S_{i_0}^{\mathcal{F}} \leftarrow S_{i_1}^{\mathcal{F}}$
32                 Remove $root$ from $T_{\mathcal{F}}$ and rebuild $T_{\mathcal{F}}$ in tree form
33             Add the following rules to $E$:
                 rule with head $S_{i_\#}^{\mathcal{F}}(ars_1)$ that calculates violations of PutGet,
                 $\Delta S^+(ars_1) :- \Delta S_b^+(ars_1)$. and $\Delta S^+(ars_1) :- S_{i_1}^{\mathcal{F}}(ars_1), \neg S_{i_\#}^{\mathcal{F}}(ars_1)$.
34     return $E$
```

Algorithm 4 describes how we can templatize the effects of FDs in NR-Datalog*. TEMPLATIZEEFFECTFDS takes a set $\mathcal{S}_f$ of schemas with computed FDs and a set of atomic queries C_g as input and produces a set E consisting of addition candidate rules related to handling the effects of FDs.

For each atomic rule r in C_g, we can specify the view V and the source S against r as well as their given FDs in $\mathcal{S}_f$. This procedure only makes more rules about the effects of FDs if a considered source has a nonempty set of FDs in tree form. In such a case, the procedure will

1. replace the basic candidate rules whose heads are either ΔS^- or ΔS^+ with new base rules whose heads are, respectively, either ΔS_b^- or ΔS_b^+ (line 7);
2. prepare the rules encoding the strategy that resolves conflicts caused by FDs by deletion (lines 8-15);
3. prepare the rules encoding the strategy that resolves conflicts caused by FDs by revision (lines 16-33);

To prepare rules for deletions, TEMPLATIZEEFFECTFDS uses the two-step pattern:

1. defining a relation that keeps conflicting tuples due to FDs;
2. preparing rules for usual deletions and deletions that occur due to conflicts.

To prepare rules for revisions, TEMPLATIZEEFFECTFDS uses the four-step pattern:

1. defining a relation that keeps tuples in S that might need to be revised;
2. following the tree form of $\mathcal{F}_S$ from roots to leaves and preparing pairs of rules that each define a new relation expressing a recursive step of revision relations with ΔS_b^+;
3. generating another relation to calculate violations of PUTGET;
4. preparing rules for usual insertions and insertions that occur due to conflicts.

Discussion on Correctness, Well-behavedness and Incompleteness

If a pair of (get, put) is synthesized by our approach, the pair will be consistent with the user-provided example because of the correctness of ProSynth.

The well-behavedness of a synthesized pair (get, put) is ideally due to

1. the well-behavedness of all atomic pair (get_a, put_a)s,

2. the well-behavedness of the composition of bidirectional programs [1].

For each atomic pair, the GetPut law holds no matter what rules ProSynth chooses and no matter what rules ProSynth does not choose, while the PutGet law can be fulfilled because of our well-designed templates encoding well-behaved view update strategies.

For the composition of all atomic pairs, i.e., (get, put), the GetPut is always ensured, while the PutGet is not always ensured. Forward-propagating FDs sometimes cannot fully propagate constraints against the source s to constraints against the view v. Then, if we evaluate put over s and an updated view v' including a tuple that violates the missing constraints against the view, to obtain s', and apply get over s', the result would be different from v', which violates the PutGet law.

The necessary conditions for PutGet to be ensured include

1. the original (updated) view could be forward defined from the original (updated) source by a set of atomic rules satisfying the decomposable conditions,

2. all constraints from the source are fully propagated to intermediate relations and the view.

It should be remarked that if we find a pair of (get, put) from a valid user-given example $\mathcal{E} = (T_S, T_V, T_{S'}, T_{V'})$ (i.e., $\mathcal{E}$ satisfies the well-behavedness properties), the PutGet law is ensured against $\mathcal{E}$ because we synthesize the get from $(input = T_S, output = T_V)$ and $(input = T_{S'}, output = T_{V'})$, and synthesize the put from $(input = (T_S, T_{V'}), output = T_{S'})$.

Our approach is not complete. If there exists a well-behaved bidirectional program of (get, put) consistent with example $\mathcal{E}$, our approach may return $None$. It can output such a well-behaved pair (get, put) for the specification in which

1. the example $\mathcal{E}$ is an instance of the backward update propagation with minimal effects plus constraints and effects of FDs (if any) from the view to the source,

2. both of the above necessary conditions to ensure PutGet are satisfied.

6.3 Evaluation

We have fully implemented a prototype named SynthBP for Algorithm 1 in 8K lines of Python, without relaxing any procedures. The minimal-effect view update strategies are encoded in templates of Setup B. These templates are embedded inside PreparePutCandMEVUS to automatically generate basic candidate rules of view update programs. Procedures TemplatizeConstraintFDs (Algorithm 3) and TemplatizeEffectsFDs (Algorithm 4) introduced in this chapter are implemented as steps of PreparePutCandCEFDs in the high-level algorithm (Algorithm 1), to generate more rules that handle FDs.

SynthBP also respectively uses ProSynth$^+$ and Soufflé as the unidirectional synthesizer and the underlying Datalog solver. For flexibility, SynthBP supports users to provide their own candidates for the desired query or even a complete query in the form of NR-Datalog* rules. If a query is given as a set of atomic queries, the step of decomposing the query may be skipped. These flexibilities not only help users be more proactive if they understand the desired query but can also help SynthBP reduce the amount of time it takes to synthesize the query during the entire working time.

Research questions

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

Q1 Can SynthBP successfully synthesize bidirectional programs from examples with and without FDs?

Q2 How efficient is SynthBP?

Table 6.1: Benchmark characteristics

#	Benchmark	Schema				Example	
		#Rels	#Attrs	#FDs	?PK	#InsDel Tuples	#Total Tuples
$\mathcal{B}_1$ (adapted from [16])							
#01	cars_master	2	5	2	Y	4	44
#02	goodstudents	2	7	4	Y	10	18
#03	luxuryitems	2	6	2	Y	6	12
#04	usa_city	2	7	3	Y	8	1272
#05	ced	3	6	-	-	16	54
#06	residents1962	2	6	-	-	10	24
#07	employees	3	8	-	-	21	65
#08	researchers	3	6	-	-	9	39
#09	retired	3	6	-	-	2	42
#10	paramountmovies	2	8	-	-	4	8
#11	officeinfo	2	12	8	Y	4	28
#12	vw_brands	3	6	-	-	16	48
#13	residents	4	10	-	-	24	56
#14	bstudents	3	11	5	Y	7	27
#15	all_cars	3	9	3	Y	6	52
#16	tracks1	3	11	3	N	16	28
#17	tracks2	2	9	3	N	14	18
#18	tracks3	2	8	2	N	12	14
#19	newpc	3	14	-	-	15	115
#20	activestudents	3	12	-	-	5	25
#21	vw_customers	3	19	-	-	2	1912
#22	measurement	3	12	6	Y	10	110
#23	ukaz_lok	2	10	-	-	20	60
#24	message	3	22	12	Y	26	138
#25	phonelist	4	21	-	-	22	130
#26	purchaseview	3	8	3	Y	64	210
#27	vehicle_view	3	8	3	Y	17	53
#28	outstanding_task	2	26	-	-	11	33
#29	poi_view	3	13	6	Y	600	3998
#30	products	3	19	-	-	15	175
#31	koncerty	4	12	2	Y	24	460
#32	emp_view	3	10	-	-	0	28
$\mathcal{B}_2$ (handcrafted using data from the `airportdb` database)							
#01	airplane_br	3	13	6	Y	31	85
#02	bookinginfo	3	16	7	Y	19	261
#03	flightsc_fmosaka	3	23	7	N	14	150
#04	flightsc_wk	2	19	-	-	14	52
#05	newflightdat	3	16	9	Y	18	188
#06	passengerinfo	4	15	9	Y	36	168

Table 6.2: Main experiment results [SynthBP]

#	Benchmark	Synthesis of *get*		Set of Atomic Queries		Synthesis of *put*		MeanSynthTime [s]		
		#SRules	#CRules	Types	#Rels	#SRules	#CRules	MSTget	MSTput	MSTtotal
$\mathcal{B}_1$ (adapted from [16])										
#01	cars_master	1	1	π	2	15	15	0.08	0.07	0.40
#02	goodstudents	1	1	π, σ	3	45	45	0.08	0.20	0.58
#03	luxuryitems	1	1	σ	2	23	23	0.08	0.09	0.44
#04	usa_city	1	1	π, σ	3	40	40	0.09	0.17	0.58
#05	ced	1	1	$\setminus$	3	6	6	0.08	0.06	0.38
#06	residents1962	1	1	σ	2	9	9	0.08	0.06	0.37
#07	employees	1	1	π, $\bowtie$	4	18	18	0.08	0.08	0.43
#08	researchers	1	1	2π, σ, $\cap$	6	26	26	0.08	0.11	0.47
#09	retired	1	1	2π, $\setminus$	5	16	16	0.08	0.08	0.42
#10	paramountmovies	1	1	π, σ	3	12	12	0.08	0.07	0.41
#11	officeinfo	1	1	π	2	27	27	0.08	0.13	0.49
#12	vw_brands	2	2	2π, $2\times$, $\cup$	7	40	46	0.09	0.89	1.28
#13	residents	3	3	$2\times$, $2\cup$	7	36	48	0.08	1.02	1.39
#14	bstudents	1	1	2π, σ, $\bowtie$	6	66	66	0.08	0.31	0.73
#15	all_cars	1	1	$\bowtie$	3	34	34	0.08	0.15	0.48
#16	tracks1	1	1	$\bowtie$	3	34	34	0.08	0.15	0.52
#17	tracks2	1	1	π	2	19	19	0.08	0.09	0.43
#18	tracks3	1	1	σ	2	27	27	0.08	0.12	0.46
#19	newpc	1	1	π, σ, $\bowtie$	5	22	22	0.08	0.10	0.46
#20	activestudents	1	1	π, σ, $\bowtie$	5	22	22	0.09	0.11	0.49
#21	vw_customers	1	1	2π, $\bowtie$	5	21	21	0.10	0.13	0.59
#22	measurement	2	2	2σ, $\cup$	5	59	73	0.08	1.26	1.66
#23	ukaz_lok	1	1	$\times$	2	9	9	0.08	0.06	0.39
#24	message	2	2	$2\times$, $\cup$	5	34	40	0.08	0.93	1.32
#25	phonelist	3	3	$3\times$, $2\cup$	8	42	54	0.08	1.56	1.97
#26	purchaseview	1	1	π, $\bowtie$	4	58	58	0.08	0.22	0.60
#27	vehicle_view	1	1	π, $\bowtie$	4	50	50	0.09	0.23	0.64
#28	outstanding_task	1	1	ρ, π, $\bowtie$	4	28	28	0.08	0.15	0.53
#29	poi_view	1	1	2π, $\bowtie$	5	70	70	0.09	0.38	0.88
#30	products	2	2	4ρ, π, $2\bowtie$, $\times$, $\setminus$, $\cup$	12	55	83	0.08	5.09	5.57
#31	koncerty	1	1	ρ, $2\bowtie$	6	57	57	0.11	0.40	0.96
#32	emp_view	1	1	A, …	-	-	-	0.08	-	-
$\mathcal{B}_2$ (handcrafted using data from the `airportdb` database)										
#01	airplane_br	1	1	2σ, $\times$, $\cap$	6	102	114	0.09	6.40	6.90
#02	bookinginfo	1	1	π, $\bowtie$	4	74	79	0.09	4.26	4.69
#03	flightsc_fmosaka	1	1	2π, 2σ, $\bowtie$, $\times$	8	135	146	0.10	4.72	5.44
#04	flightsc_wk	1	1	π, σ	3	18	19	0.09	0.32	0.72
#05	newflightdat	1	1	π, $\setminus$	4	66	72	0.10	3.07	3.54
#06	passengerinfo	1	1	π, $2\cap$	6	93	105	0.10	6.23	6.73

Benchmark suite

We adopted a suite of practical view update benchmarks listed in [16], totaling 32 benchmarks, out of which 16 have no FDs, 13 incorporate primary keys, and 3 utilize general FDs. Although well-collected from many sources, there are some benchmarks in [16] where certain tables are absent, rendering them insufficient to construct a valid example that aligns with the requirement of SynthBP. By adapting programs, *get* and *put*, associated with these benchmarks, and performing program evaluation on existing and/or randomized data, we can obtain the missing tables. For convenience, we refer to the original benchmark set in [16] as $\mathcal{B}_0$, and the adapted benchmark set as $\mathcal{B}_1$. Leveraging the flexibility of SynthBP, for each benchmark in $\mathcal{B}_1$, we additionally provide to SynthBP the corresponding *get* program in $\mathcal{B}_0$ in the form of NR-Datalog* rules.

We also manually crafted another benchmark set, denoted as $\mathcal{B}_2$, consisting of six benchmarks that demonstrate practical cases of updating views related to the airport. For each benchmark in $\mathcal{B}_2$, the relation schemas and examples have been customized according to the sample database `airportdb` [1]. Utilizing the flexibility of SynthBP, we have also supplemented each benchmark with our queries that are consistent with the tailored examples.

Table 6.1 presents characteristics of the adapted benchmarks, including the number of relations (#Rels), the number of attributes of relations (#Attrs), the number of FDs attached to the source (#FDs), the existence of primary keys (?PK), the number of inserted/deleted tuples (#InsDelTuples) and the total number of tuples (#TotalTuples) in the given examples.

Experimental setup

All experiments are conducted on a machine with a 2.6 GHz Intel Core i7 processor and 16 GB of 2400 MHz DDR4, running macOS Ventura 13.3. Each benchmark is experimented with 32 independent runs.

[1]https://dev.mysql.com/doc/airportdb/en/

6.3.1 Q1: Capability

Table 6.2 shows the main experiment results. For each benchmark, we collect the mean amount of time for successfully synthesizing *get* (MSTget), for successfully synthesizing *put* (MSTput) and for performing the whole synthesis process (MSTtotal). The statistics MSTget and MSTput show us whether SynthBP discovered a well-behaved pair (get, put).

According to Table 6.2, SynthBP is capable of automatically synthesizing a pair of (get, put) for each of the 31 out of 32 benchmarks in $\mathcal{B}_1$ and for each of the 6 out of 6 benchmarks in $\mathcal{B}_2$. A failure happens in benchmark $\mathcal{B}_1$#32 because the query contains an aggregation (denoted as A in Table 6.2) whose view update strategies have not been carefully investigated to be templated in Datalog. As for the remaining successful benchmarks, SynthBP shows the ability to handle diverse combinations of atomic queries. From Table 6.1 and Table 6.2, we also observe that SynthBP can solve the synthesis tasks with FDs (e.g. $\mathcal{B}_1$#01, $\mathcal{B}_1$#11, $\mathcal{B}_1$#18, $\mathcal{B}_2$#02) or without FDs (e.g. $\mathcal{B}_1$#07, $\mathcal{B}_1$#20, $\mathcal{B}_1$#28, $\mathcal{B}_2$#04).

6.3.2 Q2: Efficiency

In addition to the mean synthesis times (MSTget, MSTput, MSTtotal), Table 6.2 also provides other statistics related to the synthesis of *get* and the synthesis of *put* inside SynthBP. #SRules and #CRules are the number of rules in the solution program and the number of candidate rules, respectively.

With a *get* given by users, #SRules is equal to #CRules in the synthesis of *get*. This means that SynthBP only needs to call ProSynth/Soufflé once to check if the *get* matches the user-provided example. The number of tuples for each benchmark is of a reasonable size, and the number of candidate rules for *get* is small. This prevents excessive slowness when evaluating programs and comparing data with Soufflé and ProSynth. As a consequence, MSTget is insignificant, only around 0.1 second.

In the synthesis of *put*, if the example does not reflect complex update strategies, #SRules may be equal to #CRules, which means that all candidates are chosen by ProSynth, and ProSynth only verifies the consistency of these rules with the given example. Many benchmarks in $\mathcal{B}_1$ (e.g., $\mathcal{B}_1$#01 – $\mathcal{B}_1$#11, $\mathcal{B}_1$#16 – $\mathcal{B}_1$#21) reuse programs from $\mathcal{B}_0$ to generate the missing tables. However, those programs employ simple

Table 6.3: Experimental results about impact of FDs

#	Benchmark	Synthesis (without FDs)				Synthesis (with FDs)			
		#SRules/#CRules	MSTget	MSTput	MSTtotal	#SRules/#CRules	MSTget	MSTput	MSTtotal
$\mathcal{B}_1$									
#01	cars_master	5/5	0.10	0.09	0.44	15/15	0.09	0.13	0.61
#02	goodstudents	12/12	0.10	0.12	0.61	45/45	0.09	0.31	0.76
#03	luxuryitems	7/7	0.09	0.10	0.45	23/23	0.10	0.18	0.61
#04	usa_city	12/12	0.09	0.12	0.60	40/40	0.10	0.28	0.72
#11	officeinfo	5/5	0.10	0.08	0.45	27/27	0.11	0.30	0.69
#14	bstudents	27/27	0.09	0.23	0.68	66/66	0.10	0.49	0.97
$\mathcal{B}_2$									
#02	bookinginfo	23/28	0.10	1.34	1.89	74/79	0.09	4.26	4.69
#03	flightsc_fmosaka	38/44	0.10	1.21	1.68	135/146	0.09	4.72	5.44
#05	newflightdat	16/22	0.09	1.32	1.76	66/72	0.10	3.07	3.54
#06	passengerinfo	25/37	0.11	2.56	3.14	93/105	0.10	6.23	6.73

update strategies, resulting in the tables (examples) only capturing the simple strategies. If SynthBP cannot identify any tuples for auxiliary relations *Au* from the considered tables, SynthBP will exclude various more complex templates associated *Au*, similarly to the case of Example 5.2. This is mainly why #SRules closely matches #CRules in many cases of $\mathcal{B}_1$. In such cases, MSTput is small but tends to increase if the number of candidate rules exceeds a certain threshold ($\mathcal{B}_1$#02, $\mathcal{B}_1$#26, $\mathcal{B}_1$#27, $\mathcal{B}_1$#31). This might be due to the fact that Soufflé experiences slower processing when evaluating a program with more rules. For the benchmarks where #SRules is smaller than #CRules ($\mathcal{B}_1$#13, $\mathcal{B}_1$#22, $\mathcal{B}_1$#24, $\mathcal{B}_1$#30, $\mathcal{B}_2$#01 – $\mathcal{B}_2$#06), the provided examples capture more complicated view update strategies, and SynthBP requires more interactions between ProSynth and Soufflé to synthesize *put*. Subsequently, ProSynth must choose proper subsets of the set of candidate rules, and Soufflé needs to evaluate a greater number of candidate programs during the synthesis. In such cases, MSTput may be longer, especially for cases involving many atomic portions ($\mathcal{B}_1$#30) or a high number of candidate rules ($\mathcal{B}_2$#01, $\mathcal{B}_2$#03, $\mathcal{B}_2$#06).

The total amount of time SynthBP takes to complete a synthesis task, MSTtotal, includes not only the time to synthesize *get* and *put* but also the time to process overheads such as decomposing queries, forward-propagating FDs, computing atomic examples. The variance between MSTtotal and (MSTget + MSTput) remains relatively stable, indicating SynthBP's efficiency in handling overheads for benchmarks of reasonable size.

To estimate the impact of FDs on performance of SynthBP more carefully, we conduct additional experiments. We select 10 benchmarks from $\mathcal{B}_1$ and $\mathcal{B}_2$, relax the requirement of FDs in the schema but keep the given examples, and perform the synthesis such that synthesized programs can be output. Table 6.3 presents a summary of the metrics from the additional experiments. For each benchmark, we examine two synthesis scenarios: one without FDs and another with FDs. We collect #SRules and #CRules against the synthesis of *put*, and mean synthesis times (MSTget, MSTput, MSTtotal).

Examining Table 6.3 reveals that, among the considered benchmarks, MSTtotal for synthesis with FDs is always larger than MSTtotal for synthesis without FDs. MSTget(s) for both synthesis are quite close, given that the same *get* was specified. The variance in MSTtotal(s) is primarily attributed by differences in MSTput(s). Notably, the synthesis involving FDs tends to have a higher number of candidate rules (#CRules) for *put* compared to the synthesis without FDs. This is a result of numerous rules expressing constraints and effects of FDs generated during the synthesis process. SynthBP have the same inefficiency problem of ProSynth/Soufflé against the higher candidate rules: They will be slower if #CRules exceeds a threshold (depending on the machine environment) and if ProSynth has to find a proper subset from the set of candidate rules (i.e., #SRules is smaller than #CRules). The more number of FDs, the more candidate rules are generated, the slower the synthesis time will be.

6.3.3 Discussions

It is remarked that we have not yet developed an automatic way to check the closeness between the programs synthesized by SynthBP and those that are manually written or generated by other systems.

Performing manual comparisons of the synthesized programs and the corresponding ones provided in $\mathcal{B}_0$ is currently quite difficult. Based on the workflow of SynthBP, the programs synthesized by SynthBP are pairs each comprising a program *get* and a program *put* where *get* and *put* are essentially two combinations of many atomic portions. Unfortunately, these programs have not been specifically optimized. The synthesized *get* is equivalent to the corresponding *get* in $\mathcal{B}_0$ due to our benchmark adaptation method with flexibility. However, the synthesized *put* is more complex, especially in benchmarks involving many types of atomic queries as well as FDs.

Automatically evaluating the quality of the synthesized programs may be the focus of our future research efforts.

6.4 Summary

Functional dependencies impose constraints on relations and cause effects when updating the relations. In this chapter, we present how to templatize the constraints and effects of FDs to enrich the search space of the synthesis of view update programs.

We have fully implemented a prototype SynthBP which is equipped with the minimal templates of Setup B and the templates encoding the constraints and effects of FDs. On a suite of 38 practical benchmarks with and without FDs, SynthBP successfully and efficiently synthesizes 37 programs. The overheads for handling FDs are not too significant when the number of FDs is reasonable.

7 Conclusions

7.1 Summary

Writing well-behaved bidirectional programs (get, put) helps solve many synchronization problems such as the view problem of relational databases; however, writing such programs yourself would be difficult. In this thesis, we have researched on synthesizing bidirectional programs on relations from examples.

We survey the existing example-based synthesizers and find that: (1) the current synthesizers for bidirectional programs do not work on the domain of relations and query languages; (2) other synthesizers are unidirectional; they may be used to find two programs, a *get* and a *put*, but they cannot guarantee well-behavedness between the *get* and the *put*. A key challenge with the existing approaches to the synthesis of bidirectional programs on relations is their struggle to effectively manage the complexities of relations and query languages while preserving the desired well-behavedness of bidirectional programs. Furthermore, these approaches also face challenges in directly handling examples with internal functional dependencies.

We propose a novel approach to synthesizing bidirectional programs (get, put) on relations from examples with functional dependencies. Three main keywords behind our approach are *"decomposition"*, *"composition"* and *"templates"*. We start by synthesizing a *get* as a query on relations, then *decompose* the query *get* as a set of atomic queries get_a, and forward-propagate FDs associated with the source over this set. Subsequently, we can divide the synthesis of (get, put) to sub-synthesis of (get_a, put_a) in which the relevant relations against (get_a, put_a) are clearly associated with sets of FDs. Given an atomic queries get_a, we may synthesize the corresponding atomic view update put_a by designing well-behaved *templates*, which include both minimal templates and extra templates causing by FDs, and adapting a modern example-and-template-based synthesizer. Finally, we *compose* atomic programs (get_a, put_a)s to obtain the bidirectional program (get, put). The well-behavedness of the final program is due to the well-behavedness of each atomic program and of the composition.

We use NR-Datalog* as the base language of bidirectional programs. Although query decomposition and atomic queries are standard concepts in relational algebra, their exploration within the context of Datalog has been relatively restricted. We formulate atomic queries in NR-Datalog* and illustrate the decomposition of a complex Datalog query, satisfying decomposable conditions, into these atomic queries. Using a set of atomic queries and user-provided sets of FDs attached to the source, we describe the process of forward-propagating FDs from the source through intermediate relations defined in the atomic queries to the view. Thanks to that, we can bypass the hard problem of automatically discovering FDs from tables.

We follow the existing minimal-effect view update strategies for an atomic query get_a to design well-behaved templates for the corresponding atomic view update put_a. We introduce two different setups A and B to make templates with delta relations. The corresponding templates in these two setups have the similarity to express the meaning of the same strategy, but are written in different ways. We have implemented a prototype SynthBX that uses templates of Setup A and relaxes the issues of FDs. We evaluated SynthBX on 56 practical benchmarks, revealing its powerful and efficient performance for reasonable-size examples but noting limitations with FDs.

If a nonempty set of FDs is linked to a relation, the relation becomes constrained by FDs. Consequently, when performing updates on this relation, extra effects or additional changes may occur as a result of the impact of FDs. With FDs associated to relations after forward-propagating FDs, we present how to templatize the constraints and effects of FDs to enrich the synthesis space of the view update program. We have fully implemented a prototype SynthBP, incorporating both the minimal templates from Setup B and templates encoding the constraints and effects of FDs. Assessing SynthBP on 38 benchmarks, we find that it efficiently manages both benchmarks with and without FDs, with negligible overhead.

7.2 Future Work

The findings of this thesis open some future directions, which can be listed as follows.

Exploring and templatizing other view update strategies

In this thesis, we only provide templates encoding minimal-effect strategies and extra-effect strategies resulting from functional dependencies during view updates. Some strategies, particularly those addressing view updates involving aggregations, require further exploration. In addition to functional dependencies, various relational constraints can introduce additional effects on view updates, and they need to be included in templates. The greater the number of practical features and strategies encoded into templates, the more robust and powerful the synthesizer becomes.

Evaluating quality and optimizing programs

At present, our approach only synthesizes a bidirectional program, if any, and we currently lack a technical solution to thoroughly evaluate the solution's quality. Exploring additional programs, comparing rule quantities, or investigating suitable scoring functions for closeness computation might be necessary. Optimization techniques for Datalog programs and/or bidirectional programs may be relevant here.

Synthesizing from multiple examples

We currently consider the user-given example is a big example consisting of several smaller examples. It might to be easier for users to provide the smaller ones. We may develop a complete algorithm of merging the small examples to a big one and then slightly adapt SynthBX/SynthBP. We may also investigate the synthesis step-by-step via examples and improve the correctness of the obtained programs. The first one requires the overhead of processing a large-size example, the latter requires multiple calls to the base synthesizer. Experiments will be needed to assess which solutions are best for which situations.

Improving performance for scalability

Our prototypes do not work well with big-size examples, due to the limitations of the base synthesizer ProSynth and the base Datalog solver Soufflé. A new implementation by replacing ProSynth with a more advanced and efficient synthesis-as-rule-selection tool, such as ASPSynth [47], might to be considered.

Final remarks

The synthesis of bidirectional programs can be studied over more practical domains and for other specific applications, not just over relational databases and for the view update problem like in this thesis. In non-relation domains or domains with recursions, it is difficult to have a procedure similar to the query decomposition to reduce the synthesis to sub-synthesis. We may design a library of well-behaved combinators (with or without recursions) for a specific problem and develop the synthesis algorithms over the domain of these combinators.

Bibliography

[1] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17–es, may 2007.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, dec 1981.

[3] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, sep 1982.

[4] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. A clear picture of lens laws - functional pearl. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 215–223. Springer, 2015.

[6] Zhenjiang Hu and Hsiang-Shang Ko. Principles and practice of bidirectional programming in bigul. In Jeremy Gibbons and Perdita Stevens, editors, *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures*, volume 9715 of *Lecture Notes in Computer Science*, pages 100–150. Springer, 2016.

[7] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updatable views. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. Association for Computing Machinery.

[8] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. *SIGPLAN Not.*, 43(1):407–419, jan 2008.

[9] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. *SIGPLAN Not.*, 45(9):193–204, sep 2010.

[10] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. *SIGPLAN Not.*, 45(9):205–216, sep 2010.

[11] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. *SIGPLAN Not.*, 46(1):371–384, jan 2011.

[12] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. Bigul: A formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 61–72, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing quotient lenses. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

[14] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[15] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[16] Van-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Programmable view update strategies on relations. *Proc. VLDB Endow.*, 13(5):726–739, jan 2020.

[17] Kanae Tsushima, Bach Nguyen Trong, Robert Glück, and Zhenjiang Hu. An efficient composition of bidirectional programs by memoization and lazy update. In *Functional and Logic Programming: 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, pages 159–178, Berlin, Heidelberg, 2020. Springer-Verlag.

[18] Masaomi Yamaguchi, Kazutaka Matsuda, Cristina David, and Meng Wang. Synbit: Synthesizing bidirectional programs using unidirectional sketches. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[19] Yuepeng Wang, Xinyu Wang, and Isil Dillig. Relational program synthesis. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[20] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[21] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '85, pages 154–163, New York, NY, USA, 1985. Association for Computing Machinery.

[22] Arthur M. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 467–474, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[23] James A. Larson and Amit P. Sheth. Updating relational views using knowledge at view definition and view update time. *Inf. Syst.*, 16(2):145–168, mar 1991.

[24] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proc. VLDB Endow.*, 8(10):1082–1093, jun 2015.

[25] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. Discovery of approximate (and exact) denial constraints. *Proc. VLDB Endow.*, 13(3):266–278, nov 2019.

[26] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[27] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, mar 1989.

[28] Sumit Gulwani. Dimensions in program synthesis. pages 13–24, 01 2010.

[29] S. Gulwani, O. Polozov, and R. Singh. *Program Synthesis.* 2017.

[30] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 157–166, New York, NY, USA, 1993. Association for Computing Machinery.

[31] Rudi Horn, Roly Perera, and James Cheney. Incremental relational lenses. *Proc. ACM Program. Lang.*, 2(ICFP), jul 2018.

[32] Arthur M Keller and Jeffrey D Ullman. On complementary and independent mappings on databases. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 143–148, New York, NY, USA, 1984. Association for Computing Machinery.

[33] Kazutaka Matsuda and Meng Wang. Hobit: Programming lenses without using lens combinators. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 31–59, Cham, 2018. Springer International Publishing.

[34] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Not.*, 41(11):404–415, oct 2006.

[35] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In *CAV*, 2016.

[36] Saurabh Srivastava, Sumit Gulwani, and Jeffrey Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer*, 15, 04 2013.

[37] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming*, pages 689–706, Cham, 2017. Springer International Publishing.

[38] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 515–527, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing datalog programs using numerical relaxation. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6117–6124. ijcai.org, 2019.

[40] Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. Gensynth: Synthesizing datalog programs without language bias. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6444–6453, May 2021.

[41] Sai Zhang and Yuyin Sun. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234, 2013.

[42] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not.*, 52(6):452–466, jun 2017.

[43] Lin Cheng. Sqlsol: An accurate sql query synthesizer. In *Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings*, pages 104–120, Berlin, Heidelberg, 2019. Springer-Verlag.

[44] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. Squares: A sql synthesizer using query reverse engineering. *Proc. VLDB Endow.*, 13(12):2853–2856, sep 2020.

[45] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. Patsql: Efficient synthesis of sql queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.*, 14(11):1937–1949, oct 2021.

[46] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics*, 14:57–83, 2012.

[47] Aaron Bembenek, Michael Greenberg, and Stephen Chong. From smt to asp: Solver-based approaches to solving datalog synthesis-as-rule-selection problems. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.