# Programmable Strategies for Co-Existence of Relational Database Schemas

by

**Jumpei Tanaka**

**Dissertation**

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

***Doctor of Philosophy***

The Graduate University for Advanced Studies, SOKENDAI

March 2022

# Committee

|  |  |
|---|---|
| Supervisor | Dr. Ichiro HASUO<br>Associate Professor of National Institute of Informatics/SOKENDAI |
| Advisor | Dr. Zhenjiang HU<br>Professor of Peking University/National Institute of Informatics |
| Advisor | Dr. Hiroyuki KATO<br>Assistant Professor of National Institute of Informatics/SOKENDAI |
| Examiner | Dr. Masato TAKEICHI<br>Professor Emeritus of University of Tokyo |
| Examiner | Dr. Taro SEKIYAMA<br>Assistant Professor of National Institute of Informatics/SOKENDAI |

# Acknowledgments

There are many people without whom I would never have achieved this goal. All of them contributed in their way and proved to be essential to my academic and professional growth, personal development, and mental health. Here, I want to thank them all.

My sincere and deep gratitude goes first and foremost to my (former) supervisor, Professor Zhenjiang Hu. Since we met at the school's guidance five years ago, I have been strongly impressed and encouraged by you. I remember you encouraged me when I was deeply depressed by the struggles of writing a paper. I never forgot your simple comment, never give up, and repeated it again and again. Especially, I am grateful for your careful attention to making a good environment for the laboratory and the team. I have been helped many times by people around you. I would also like to express my deep gratitude to Professor Emeritus Masato Takeichi and Assistant Professor Hiroyuki Kato. While being very busy, you kindly spent time on frequent discussions. I cannot thank you enough. I am grateful to my advisory committee members, Professor Ichiro Hasuo and Assistant Professor Taro Sekiyama. I highly appreciate their insightful and helpful comments.

I would like to offer my special thanks to Associate Professor Akimasa Morihata. Before I started Ph.D. course, you sincerely commented about my research agenda and gave the assignment to study fundamental knowledge. They helped me a lot in my research work. I am grateful to my collaborator, Van-Dang Tran. Your deep insight and a great idea of BIRDS triggered me to research about co-existence of schemas. I highly appreciate your sincere and polite comments when I ask about very detailed items about coding and proofs.

I would like to thank Mr. Akira Nakada at Nomura. You accepted without hesitation when I asked you to join a graduate school besides work. You encouraged me to challenge and allowed me to take a balance between the company's work and research activity at my discretion. I would like to thank Mr. Akio Hori at Nomura. I am proud that I worked with you for fifteen years. The root of my research agenda comes from difficulties we faced in our business. I highly appreciate your unstinting support even after expressing my intention to leave the firm.

Finally, I wish to show my great appreciation to my family. Especially my wife, Yuko, helped me over many years. I could not challenge Ph.D. without your huge support. My children, Hina and Tekkai, helped me become what I am by totally ignoring my odds.

# Abstract

The co-existence of relational database schemas is an important feature of a database. Today, most information systems consist of applications and relational databases. They continuously evolve to meet the ever-changing real world. Applications are well supported by the technology to maintain and operate multiple versions for continuous and iterative development. However, it is a challenging task in databases to make one relational database schema (source schema) evolve to a new relational database schema (target schema) and make them co-exist by sharing data for concurrently running multiple application versions or applications.

The existing work proposes the view-based approach to realize the co-existence of relational database schemas in a database. It gives logically computed view instances of both schemas transformed from a shared database. Update sharing is realized between view instances through updates of the database. A data structure of the database is compatible between a database schema suited for source schema (source-side database schema) and a database schema suited for target schema (target-side database schema). When updates against view instances of target schema increase, the source-side database becomes unsuited for view instances of target schema. The existing work makes data migration into the target-side database based on the target-side database schema available.

However, there exist three problems. First, a co-existence strategy to specify how to evolve a schema and which case data is shared between view instances of schemas is limited to the predefined one, which lacks the flexibility. Second, there is no systematic methodology to realize co-existence strategies based on the view-based approach. This problem causes difficulty in designing user intended strategies and realizing them. Especially it is challenging to design the source-side and target-side database schemas with auxiliary relation names to control update sharing between view instances of schemas. Third, due to the second problem, there is no systematic methodology for data migration between the source-side database and the target-side database for an arbitrary co-existence strategy. This problem makes the co-existence of relational database schemas with flexible data migration difficult for any co-existence strategies.

In order to make the co-existence of relational database schemas more practical, this thesis aims to make co-existence strategies for the co-existence of relational database schemas programmable by solving the problems mentioned above.

To solve the first problem, we propose a Datalog-based DSL (domain-specific language) to specify various co-existence strategies. DSL is designed for the following three items: describing definitions of source schema and target schema, describing schema evolution to specify how to evolve relations of source schema to a relation of target schema, and describing backward update sharing to specify how to share updates against a relation of target schema to relations of source schema. We assume a relationship between source schema and target schema is kept after schema evolution. Thus update sharing from relations of source schema to a relation of target schema follows the relationship specified by schema evolution. Furthermore, we introduce a property, the *consistency* of updates, and its verification method. We show that the proposed DSL can describe co-existence strategies of the existing work and other co-existence strategies.

To solve the second problem, we propose methods to systematically derive two types of bidirectional transformations (BXs for short) from a given co-existence strategy: BX between source-side or target-side database and view instances of source schema, and BX between source-side or target-side database and a view instance of target schema.

We define the source-side database schema to consist of base relation names having the same data structure with relation names of source schema and auxiliary relation names. A key idea of deriving BXs on the source-side database is that BX between the source-side database and the view instance of source schema is basically identity mapping between the base relation and the view instance. BX between the source-side database and the view instances of target schema is derived so that a co-existence strategy is realized between the base relations and the view instance of target schema. Auxiliary relations are systematically specified to compute the updated view instances from the updated base relations and the auxiliary relations without loss or gain.

To have a database suited for target schema, we define the target-side database schema to consist of a base relation name having the same data structure with a relation name of target schema and auxiliary relation names that are different from auxiliary relation names of the source-side database schema. A key idea of deriving BXs on the target-side database is that schema evolution is replaced to BXs between target-side database and view instances of source schema, and backward update sharing is replaced to BX between target-side database and the view instance of target schema. Auxiliary relations are systematically specified to compute the updated view instances from the updated base relations and the auxiliary relations without loss or gain.

We evaluate the usefulness of the proposed methods by showing results of derived BXs and execution time of writing and reading against view instances of source schema and target schema. Co-existence strategies of the existing work and other co-existence strategies are evaluated.

To solve the third problem, we propose a method of data migration based on the derived BXs. The source-side database is migrated to the target-side database by the following two steps: the BXs on the source-side database compute view instances of source schema and target schema, then such view instances are transformed into the target-side database by the BXs on the target-side database. The target-side database is migrated to the source-side database by the opposite operations. We evaluate the usefulness of the proposed method by showing the execution time of data migration. Co-existence strategies of the existing work and other co-existence strategies are evaluated.

# Contents

# List of Figures

# List of Tables

# 1
# Introduction

## 1.1 Motivation

The co-existence of relational database schemas is an important feature of a database. Today, most information systems consist of applications and relational databases [13]. They continuously evolve to meet the ever-changing real world. In an application, technologies to support continuous evolution are widely and deeply studied [39, 4, 27, 77]. For example, strong tools, SVN and GIT, support continuous and iterative development by maintaining multiple versions of an application and deploying several versions to run concurrently. Databases are also expected to accommodate these features. However, it is a challenging task in databases to make one relational database schema (a schema for short) evolve to a new schema and make multiple schemas co-exist by sharing data over them for concurrently running multiple application versions or applications.

### 1.1.1 Strategies for Co-existence of Relational Database Schemas

We give an overview of the co-existence of relational database schemas and what strategies are designed for it. The co-existence of relational database schemas (the co-existence of schemas for short) requires that each schema can operate arbitrary updates against a certain common data set [3]. In this thesis, the co-existence of schemas is a set of schemas that satisfies the following features: a new schema (target schema) is defined by schema evolution based on an original schema (source schema), and updates can be shared among these schemas. Note that update sharing has cases to share and not share an update over one schema with another schema depending on the situation, such as application design, phases of application rollout, or business logic.

Strategies for the co-existence of schemas to meet such situations are designed by specifying schema evolution and update sharing. Suppose that a relation $S_1(X, Y, Z)$ of source schema evolves to a relation $T_1(X, Y)$ of target schema by projection $\pi$ to project away $S_1$'s attribute $Z$. Schema evolution computes the initial data of target schema as follows:

$$\left\{\begin{array}{c|ccc} S_1 & X & Y & Z \\ \hline & x1 & y1 & z1 \end{array}\right\} \xrightarrow{\pi} \left\{\begin{array}{c|cc} T_1 & X & Y \\ \hline & x1 & y1 \end{array}\right\}$$

Once target schema is evolved from source schema and has initial data, updates would occur against data over each schema. Several cases of update sharing between them could exist. The following strategy shares any updates in the forward and backward direction between $S_1$ and $T_1$ except for values in the projected away attribute $Z$ (+/− denotes an update as insertion/deletion of a tuple against a relation):

$$\left\{\begin{array}{c|ccc} S_1 & X & Y & Z \\ \hline - & \cancel{x1} & \cancel{y1} & \cancel{z1} \\ + & x2 & y2 & z2 \end{array}\right\} \rightarrow \left\{\begin{array}{c|cc} T_1 & X & Y \\ \hline - & \cancel{x1} & \cancel{y1} \\ + & x2 & y2 \end{array}\right\}$$

$$\left\{\begin{array}{c|ccc} S_1 & X & Y & Z \\ \hline - & \cancel{x2} & \cancel{y2} & \cancel{z2} \\ + & x3 & y3 & \end{array}\right\} \leftarrow \left\{\begin{array}{c|cc} T_1 & X & Y \\ \hline - & \cancel{x2} & \cancel{y2} \\ + & x3 & y3 \end{array}\right\}$$

As another strategy, if source schema evolves to target schema as a testbed of a new application version, inserted and deleted test data of a target schema must not contaminate data of source schema. It is an extreme but reasonable strategy that updates against $S_1$ of source schema is shared with $T_1$ of a target schema, and updates against $T_1$ is not shared with $S_1$:

$$\left\{\begin{array}{c|ccc} S_1 & X & Y & Z \\ \hline - & \cancel{x1} & \cancel{y1} & \cancel{z1} \\ + & x2 & y2 & z2 \end{array}\right\} \rightarrow \left\{\begin{array}{c|cc} T_1 & X & Y \\ \hline - & \cancel{x1} & \cancel{y1} \\ + & x2 & y2 \end{array}\right\}$$

$$\left\{\begin{array}{c|ccc} S_1 & X & Y & Z \\ \hline & x2 & y2 & z2 \end{array}\right\} \leftarrow \left\{\begin{array}{c|cc} T_1 & X & Y \\ \hline - & \cancel{x2} & \cancel{y2} \\ + & x3 & y3 \end{array}\right\}$$

Designing the co-existence of schemas amounts to designing a strategy consisting of schema evolution and a rule to specify in what case an update is shared or not shared among data over schemas. This strategy, named the co-existence strategy, ought to be programmable to meet a variety of situations in practical use. "Programmable" is to equip a language to describe co-existence strategies and a mechanism to realize written strategies.

### 1.1.2 Problems of Existing Methods

We show existing methodologies for the co-existence of schemas and clarify their problems. Current RDBMSes (Relational Database Management Systems) do not support the co-existence of schemas. Practitioners prepare one database for source schema and another database for target schema. Schema evolution is implemented as data migration from one database to another database. Update sharing is implemented so that an update against one database triggers an update against another database [2]. Since all can be programmed by SQL, this method is good to make schema evolution and rules of update sharing programmable. However, writing a mapping for data migration by SQL is error-prone work and requires considerable development resources [13, 36]. Furthermore, update sharing between schemas causes duplicated data in two databases. It makes update sharing in both directions, from source schema to target schema and vice versa, difficult because a synchronization mechanism for duplicated data is additionally required.

Research about the co-existence of schemas is still in the early stage, even if we look around the database community. As a notable work, Herrmann et al. propose MSVDB (Multi-Schema-Version Database) [34, 35] to overcome the difficulty mentioned above. MSVDB provides schema modification operations (SMOs). Each SMO predefines a co-existence strategy: schema evolution specifying how to evolve relations of source schema to relations of target schema, and one rule of update sharing between these relations. Note that SMOs' expressive power to describe schema evolution is relationally complete.

Figure 1.1 depicts MSVDB's high-level architecture. MSVDB realizes the co-existence of schemas by the view-based approach. Figure 1.1 (a) shows that MSVDB gives instances of both source schema and target schema as union of view instances logically computed from a shared database (source-side database) by views. The source-side database schema consists of base relation names having the same data structure with relation names of source schema and auxiliary relation names for supplemental data to compute view instances of target schema. Updates against a view instance are transformed into the database by an update translator. Updates are shared among view instances through the updated database. When updates against view instances of target schema increase, the source-side database becomes unsuited for

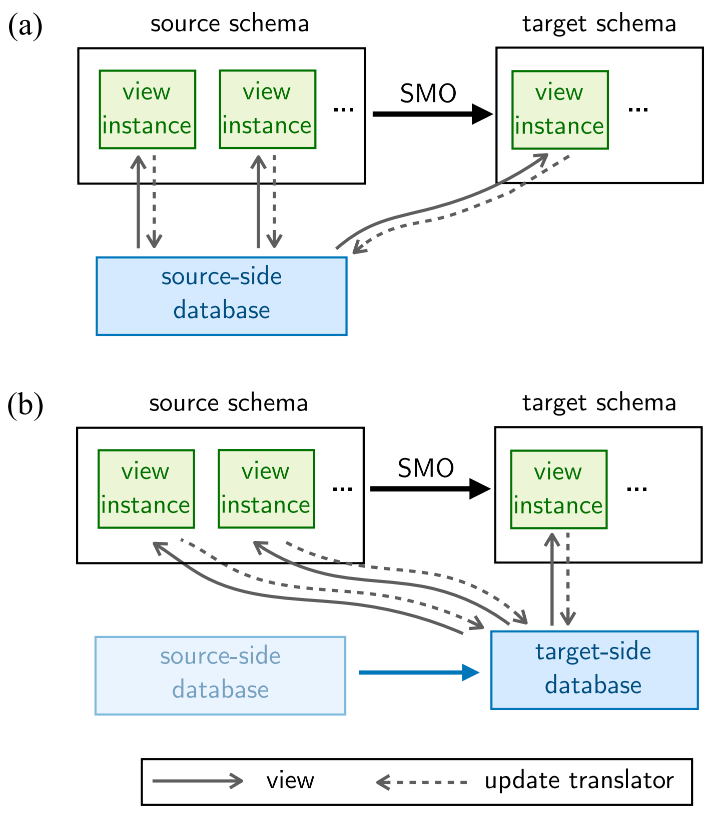


Figure 1.1: The co-existence of schemas by the view-based approach.

view instances of target schema. MSVDB makes data migration into the target-side database based on target schema (Figure 1.1 (b)). The target-side database consists of the base relation names having the same data structure with the relation names of target schema and other auxiliary relation names for supplemental data to compute the view instances of source schema and target schema. MSVDB provides another set of views and update translators on the target-side database.

MSVDB provides a solution to design the co-existence of schemas and realize it easily. However, several problems exist:

- A relationship between schemas is fixed by one predefined strategy of SMO. Since SMO's co-existence strategy consists of one schema evolution with one predefined behavior of update sharing, it lacks the flexibility to arbitrarily design a co-existence strategy by specifying schema evolution and update sharing.

- There is no systematic methodology to realize co-existence strategies by the view-based approach. The architect of MSVDB designs views and update translators on an ad hoc basis for each SMO's predefined strategy. Especially, it is challenging to design source-side and target-side database schemas with auxiliary relation names to control update sharing between view instances by following co-existence strategies. This problem causes difficulty in designing user intended strategies and realizing them. Even if a user of MSVDB will change a strategy or introduce a new strategy, it is challenging to systematically derive views, update translators and auxiliary relation names without troubling a user.

- Due to the second problem, there is no systematic methodology for data migration between a source-side database and a target-side database for an arbitrary co-existence strategy. This problem makes the co-existence of schemas with flexible data migration difficult for any co-existence strategies.

## 1.2 Research Objective

This thesis aims to make the co-existence of schemas practical by making co-existence strategies programable. "Programmable" for a co-existence of relational database schema is to satisfy two features: a co-existence strategy is describable, and a mechanism to realize a described strategy is equipped. A co-existence strategy consists of schema evolution from the source schema to the target schema and a specification of update sharing between data over them. An expressive power for schema evolution must be relationally complete.

## 1.3 Contributions

To achieve the research objective, we must overcome two challenges: how to describe a co-existence strategy and how to realize an arbitrarily written co-existence strategy. For the first challenge, we propose a language to describe a co-existence strategy. For the second challenge, we propose methods to derive bidirectional transformations that realize a co-existence strategy on the source-side database or the target-side database.

The view-based approach of the existing work is based on an updatable view by view and update translator between a view instance and a database. The updatable view has been widely studied in the database community [1, 16] and the programming language community as bidirectional transformation [29]. We treat deriving views and update translators as deriving bidirectional transformations (BX for short). Our contributions of this thesis are summarized as follows:

- For the first challenge, the existing work has a problem that it cannot arbitrarily describe a co-existence strategy. We propose a Datalog-based DSL to describe co-existence strategies. It makes a co-existence strategy between relations of source schema and a relation of target schema programmable. The proposed DSL is relationally complete to describe schema evolution and has the same expressive power with MSVDB's SMOs. We show that the proposed DSL can describe SMOs' strategies and other strategies. Furthermore, we introduce a property, the *consistency* of updates, and its verification methods. We show that the proposed DSL can describe co-existence strategies of the existing work and other co-existence strategies.

- For the second challenge, the existing work has a problem that there is no systematic methodology to realize a co-existence strategy regardless of whether the source-side or the target-side database. We propose a method to systematically derive two types of BXs on the source-side database from a given co-existence strategy: BXs between the source-side database and view instances of source schema and BX between the source-side database and a view instance of target schema. We define the source-side database schema to consist of base relation names having the same data structure with relation names of source schema and auxiliary relation names. A key idea of deriving the BXs is that BX between the source-side database and a view instance of source schema is basically identity mapping between the base relation and the view instance. Then BX between the source-side database and a view instance of target schema is derived so that a co-existence strategy is realized between the base relations and the view instance of target schema. We evaluate the usefulness of the proposed method by showing results of derived BXs and execution time of writing and reading against view instances of source schema and target schema. Co-existence strategies of the existing work and other co-existence strategies are evaluated.

- Following the second challenge, we propose a method to systematically derive two types of BXs on the target-side database from a given co-existence strategy: BXs between the target-side database and view instances of source schema and BX between the target-side database and a view instances of target schema. A key idea of deriving the BXs is that schema evolution is replaced to BXs between the target-side database and view instances of source schema, and backward update sharing is replaced to BX between the target-side database and a view instance of target schema. We evaluate the usefulness of the proposed method deriving BXs on the target-side databases in the same manner as the third contribution mentioned above.

- Due to the second challenge, the existing work has a problem that there is no systematic methodology for data migration between the source-side database and the target-side database for a co-existence strategy. We propose a method of data migration based on the derived BXs. The source-side database is migrated to the target-side database by the following two steps: the BXs on the source-side database compute view instances of source schema and target schema, then such view instances are transformed into the target-side database by the BXs on the target-side database. The target-side database is migrated to the source-side database by the opposite operation. We evaluate the usefulness of the proposed method by showing the execution time of data migration. Co-existence strategies of the existing work and other co-existence strategies are evaluated.

We published parts of first and second contributions in [66, 65, 67].

Figure 1.2 shows an overall procedure to derive BXs for the realization of co-existence strategies. The input is a co-existence strategy written by the proposed DSL. The output is BXs to realize the co-existence strategy on the source-side database or the target-side database. The first step is a verification of the *consistency* of updates whether a given co-existence strategy satisfies it. If satisfied, the second step is a realization of co-existence strategies on the source-side database or the target-side database. This step outputs two types of BXs on the source-side database or the target-side database.

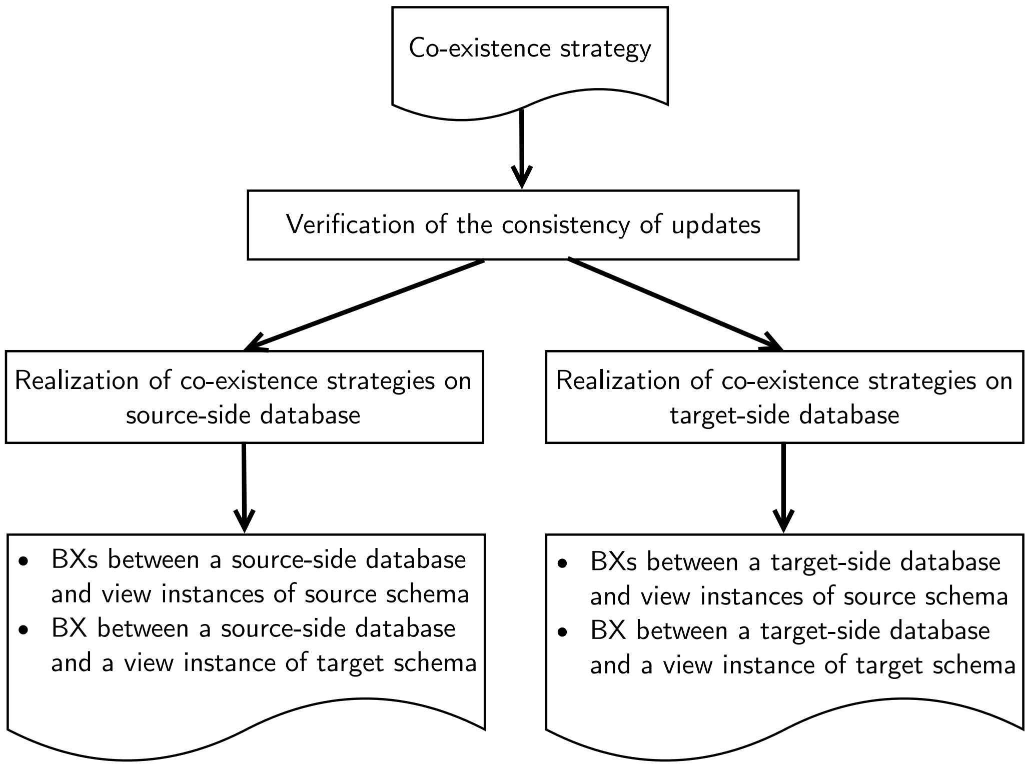

Co-existence strategy
Verification of the consistency of updates
Realization of co-existence strategies on source-side database
Realization of co-existence strategies on target-side database
• BXs between a source-side database and view instances of source schema
• BX between a source-side database and a view instance of target schema
• BXs between a target-side database and view instances of source schema
• BX between a target-side database and a view instance of target schema

Figure 1.2: Overall procedure to derive BXs for realization of co-existence strategies.

The following chapters of this thesis explain each step in order from the top. The next section shows an outline of this thesis.

## 1.4 Outline

We organize the remainder of the thesis as follows. Chapter 3 – Chapter 5 are the main contributions of this thesis. These chapters include ideas to make the co-existence schemas programmable.

**Chapter 2.** We introduce background knowledge for making the co-existence of schemas programmable. It consists of four major objectives: Datalog with a notation that will be used throughout this thesis, schema evolution as a basis of co-existence of schemas to support continuous evolution of database schema, co-existence of schemas to make schemas utilized in parallel by sharing data, and bidirectional transformation as a method to realize co-existence of schemas.

**Chapter 3.** We present a DSL, domain-specific language, to describe a co-existence strategy. Its syntax and semantics are given. A property, *consistency* of updates, and its verification method are explained. We provide examples of strategies written by the proposed DSL.

**Chapter 4**. We present how to realize a given co-existence strategy on the source-side database. We explain how to design auxiliary relations systematically and derive BXs between the source-side database and view instances of source schema and target schema. We show experimental results to evaluate the usefulness of the proposed method for co-existence strategies of SMOs in the related work and other co-existence strategies.

**Chapter 5**. We present how to realize a given co-existence strategy on the target-side database. We explain how to design other auxiliary relations systematically and derive BXs between the target-side database and view instances of source schema and target schema. Further, we explain how to migrate data between the source-side database and the target-side database. We show experimental results to evaluate the usefulness of the proposed method for co-existence strategies of SMOs in the related work and other co-existence strategies.

**Chapter 6**. We conclude the thesis by reviewing our contributions and discussing possible future works.

# 2
# Background Knowledge

This chapter provides background knowledge for making the co-existence of schemas programmable. We first introduce basic knowledge of Datalog with a notation that is used throughout this thesis. We explain schema evolution by presenting its basic concept, then introduce how the co-existence of schemas by the view-based approach works and bidirectional transformation as a technique for realization of a co-existence schemas.

## 2.1 Datalog

We first introduce a notation of relational database and then explain basic knowledge of Datalog. A database schema is a finite sequence of relation names (or predicate symbols) $\langle r_1, \dots, r_n \rangle$. Each relation name $r_i$ has attribute names $X_1, \dots X_m$. By denoting attribute names as a vector $\vec{X_i} = (X_1, \dots X_m)$, a relation name is denoted as $r_i(\vec{X_i})$. A relation $R_i(\vec{X_i})$ (or simply $R_i$ by omitting attributes) is a set of tuples over relation name $r_i(\vec{X_i})$. Database $\mathbf{D}$ is union of relations $R_1(\vec{X_1}), \dots, R_n(\vec{X_n})$.

A Datalog program $P$ is a non-empty finite set of rules expressed by the following form [72, 15]:

$$H :- L_1, \dots, L_i, \dots, L_n. \tag{2.1}$$

$H$ is called the head of the rule, and $L_1, \dots, L_i, \dots, L_n$ is called the body of the rule. $H$ is a predicate. Each $L_i$ in the body is literal. Literal is a non-negated or negated predicate $r_i(\vec{X_i})$ corresponding to a relation $R_i(\vec{X_i})$, a non-negated or negated predicate appearing in heads of other Datalog rules, or a non-negated or negated predicate built-in predicate with an arithmetic comparison operator such as =, <.

## 2.2 Schema Evolution

An objective of schema evolution is to modify an original database schema (source schema) to a new database schema (target schema) without data loss and translate queries and updates issued over the source schema to work on the target schema. In order to specify schema evolution this thesis follows the *instance-level semantics* [11] of schema evolution. It specifies a mapping between a database of the source schema and one of target schema given data structure change between these schemas. If **S** and **T** are databases of the source schema and the target schema, a mapping between them defines a subset of $\mathbf{S} \times \mathbf{T}$. Usually, the mapping is expressed as a set of transformations. A transformation is a formula in some mapping language and defines the subset of $\mathbf{S} \times \mathbf{T}$ that the formula holds. Each transformation might be a mapping based on simple schema modifications, such as the addition, deletion, or renaming of attributes of a relation, and composed schema modifications, such as join of two relations into one relation, decomposition of one relation to two relations, merge of tuples in two relations into one relation, and partition of tuples in one relation into two relations [48].

**Example 1.** Suppose a database of source schema as union of relations $S_1(A, B, C)$, $S_2(X, Y)$ and $S_3(X, Z)$ and another database of target schema as union of relations $T_1(A, B)$ and $T_2(X, Y, Z)$. The relation $T_1(A, B)$ is a result of schema evolution from $S_1(A, B, C)$ by projection to drop the attribute $C$. The relation $T_2(X, Y, Z)$ is a result of schema evolution from $S_2(X, Y)$ and $S_3(X, Z)$ by join of them. These transformations of schema evolution are described by Datalog as follows:

$$t_1(A, B) \quad :- \quad s_1(A, B, C). \tag{2.2}$$

$$t_2(X, Y, Z) \quad :- \quad s_2(X, Y), s_3(X, Z). \tag{2.3}$$

where predicates $s_1(A, B, C)$, $s_2(X, Y)$, $s_3(X, Z)$, $t_1(A, B)$, and $t_2(X, Y, Z)$ correspond to relations $S_1$, $S_2$, $S_3$, $T_1$, and $T_2$ respectively. Rule (2.2) expresses that the relation $T_1$ is transformed from the relation $S_1$ by dropping the attribute $C$. Rule (2.3) expresses that the relation $T_2$ is transformed from the relations $S_2$ and $S_3$ by join when values of attributes $X$ are the same. □

## 2.3 Co-Existence of Schemas

An objective of co-existence of schemas is to realize a co-existence strategy consisting of schema evolution and update sharing between relations of schemas. In this thesis, we follow the view-based approach proposed in [34, 35] to realize co-existence strategies.

Recall the example of co-existence strategy shown in section 1.1 that the relation $S_1$ of source schema is evolved to the relation $T_1$ of target schema by projecting away an attribute $Z$ and any updates are shared in the forward and backward direction between $S_1$ and $T_1$ except for values in the projected away attribute $Z$. Figure 2.1 depicts the view-based approach to realize this strategy in one database. Figure 2.1 (a) shows initial data after schema evolution. Database is designed to have a base relation $B_{S_1}$ that has same attributes with a relation $S_1$. The relation $S_1$ of source schema is turned to a view instance $S_1$ that is logically computed by the view as identity mapping from the base relation $B_{S_1}$. The relation $T_1$ of target schema is turned to a view instance $T_1$ that is logically computed by the view that specifies schema evolution as projection from the base relation $B_{S_1}$. Figure 2.1 (b) shows forward update sharing from $S_1$ to $T_1$. The updated view instance of $S_1$ is transformed to the updated base relation by the update translator as identity mapping. Then the updated base relation is transformed into the updated view instance of $T_1$ by the view as projection. As a result, inserted and deleted tuples against $S_1$ are shared with $T_1$. Figure 2.1 (c) shows backward update sharing from $T_1$ to $S_1$. The updated instance of $T_1$ is transformed to the updated base relation by the update translator. Then the updated base relation is transformed to the updated view instance of $S_1$ by the view as identity mapping. As a result, inserted and deleted tuples against $T_1$ are shared with $S_1$.

However, if a co-existence strategy specifies not to share update against $T_1$ to $S_1$, we cannot update the base relation $B_{S_1}$. If the base relation $B_{S_1}$ is updated, its updated result is reflected to the view instance $S_1$ even though the strategy specifies not to share with $S_1$. To avoid this case, database can be deigned to have auxiliary relations so that supplemental data to compute the view instance $T_1$ is separately kept from the base relation. In this thesis, we propose methods to systematically derive a database schema having auxiliary relation names to realize update sharing specified by an arbitrary co-existence strategy.

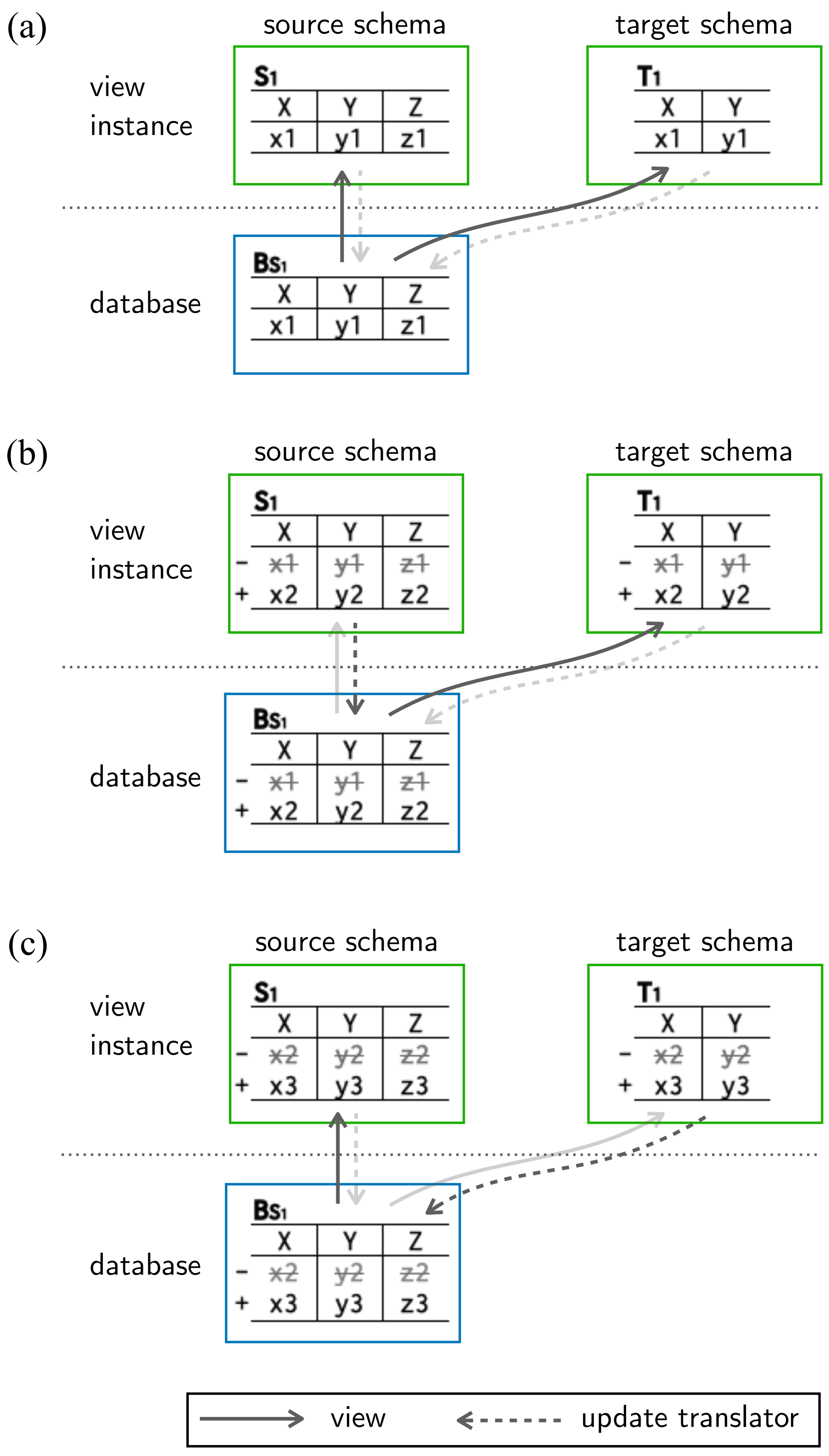


Figure 2.1: Realization of a co-existence strategy by the view-based approach.

## 2.4 Bidirectional Transformation

Bidirectional transformation (BX for short) [29] is a mature technique synchronizing updates between heterogeneous data models. It is a pair of a forward transformation *get* and a backward transformation *put*. Given source data $\mathcal{S}$ and view data $\mathcal{V}$, a transformation $get(\mathcal{S}) = \mathcal{V}$ accepts source data $\mathcal{S}$ and produces view data $\mathcal{V}$. Another transformation $put(\mathcal{S}, \mathcal{V}') = \mathcal{S}'$ accepts the original source data S and updated view data $\mathcal{V}'$, and produces updated source $\mathcal{S}'$.

To ensure the consistency between source data and target data, BX must satisfy the following round-tripping laws, called GetPut and PutGet:

$$\begin{aligned} put(\mathcal{S}, get(\mathcal{S})) &= \mathcal{S} && (\textsc{GetPut}) \\ get(put(\mathcal{S}, \mathcal{V}')) &= \mathcal{V}' && (\textsc{PutGet}) \end{aligned}$$

The GetPut ensures not updated target data corresponds to not updated source data. The PutGet ensures an updated target data is transformed into source data such that the updated target can be computed again by *get* giving the updated source data. BX satisfying both GetPut and PutGet is *well-behaved*.

In this thesis, we realize the co-existence schemas by the view-based approach based on updatable views. A problem of updatable view is how to find an update translator that translates updates of a view instance to updates of a database so that the updated view is recomputed from the updated database without loss or gain [1].

It is known that bidirectional transformation and view update problem essentially handle the same topic [71]. Let source data $\mathcal{S}$ be a database and view data $\mathcal{V}$ be a view instance. A view to compute a view instance from the database corresponds to $get(\mathcal{S}) = \mathcal{V}$. Let $u$ be update of a view instance from $\mathcal{V}$ to $\mathcal{V}'$, i.e. $u(get(\mathcal{S})) = \mathcal{V}'$. If an update translator T transforms the update $u$ so that $T(u)$ updates database $\mathcal{S}$ to $\mathcal{S}'$, i.e. $\mathrm{T}(u)(\mathcal{S}) = \mathcal{S}'$, T is obtained by *get* and *put* of BX as follows:

$$\mathrm{T}(u)(\mathcal{S}) = put(\mathcal{S}, u(get(\mathcal{S})))$$

Thus this thesis treats a realization of a given co-existence strategy by the view-based approach as a challenge to derive BXs from a given co-existence strategy.

# 3 DSL for Description of Co-Existence Strategies

This chapter presents a DSL, domain-specific language, to describe co-existence strategies. In general, DSL is designed for efficiency to describe a domain-specific program and easiness for verification of domain-specific properties [73, 52]. In this thesis, we design DSL to describe co-existence strategies efficiently by specifying items required for it and introducing a verification method of a specific property, the *consistency* of updates. We first give an overview of DSL, then formally explain syntax snd semantics of the DSL, and a property of the *consistency* of updates with its verification method. Examples of co-existence strategies are given. The end of this chapter explains related work to the proposed DSL.

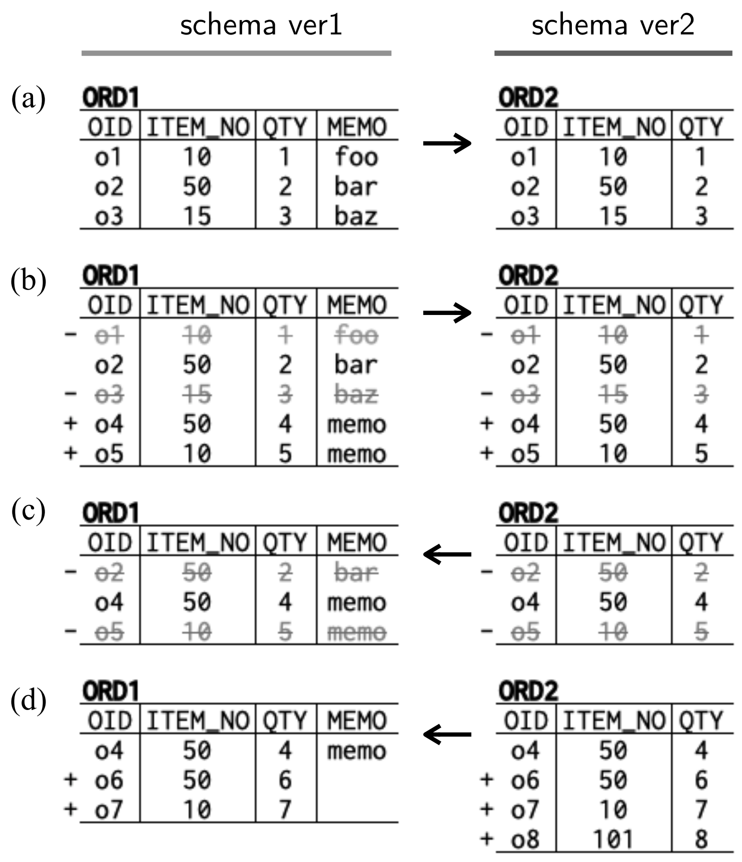
## 3.1 Overview

We go through an overview of the co-existence of schemas: how they behave, how their behaviors are described, and what consistency must be satisfied.

### 3.1.1 Running Example

We introduce a running example of a co-existence of schemas which is used through this thesis. Suppose a simplified OMS (order management system) is used in shops. It is modified to a new version to meet new requirements. The original version and the new versions of OMS run in parallel to provide the service of each version concurrently while gradually migrating users from the original version to the new version. The following example shows that one relation of source schema evolves to one relation in target schema, and data of two schemas co-exist.

**Example 2.** Suppose schema `ver1` is source schema for the OMS. Schema `ver1` consists of a relation name `ord1(OID, ITEM_NO, QTY, MEMO)` in which `OID` is a primary key and `ITEM_NO` must be greater than 1. Also, suppose that the OMS is modified to handle new products in which `ITEM_NO` is greater than 100 and drop `MEMO` because its use has become unclear. For that purpose, schema `ver2` is target schema for the modified OMS consisting of a relation name `ord2(OID, ITEM_NO, QTY)`. `OID` is a primary key, and `ITEM_NO` must be greater than 1. We denote `ORD1` and `ORD2` as relations over relation names `ord1` and `ord2`, respectively.


schema ver1
schema ver2
(a)
(b)
(c)
(d)
→
→
←
←

(a) ORD1 → ORD2

ORD1

| | OID | ITEM_NO | QTY | MEMO |
|---|---|---|---|---|
| | o1 | 10 | 1 | foo |
| | o2 | 50 | 2 | bar |
| | o3 | 15 | 3 | baz |

ORD2

| | OID | ITEM_NO | QTY |
|---|---|---|---|
| | o1 | 10 | 1 |
| | o2 | 50 | 2 |
| | o3 | 15 | 3 |

(b) ORD1 → ORD2

ORD1

| | OID | ITEM_NO | QTY | MEMO |
|---|---|---|---|---|
| - | ~~o1~~ | ~~10~~ | ~~1~~ | ~~foo~~ |
| | o2 | 50 | 2 | bar |
| - | ~~o3~~ | ~~15~~ | ~~3~~ | ~~baz~~ |
| + | o4 | 50 | 4 | memo |
| + | o5 | 10 | 5 | memo |

ORD2

| | OID | ITEM_NO | QTY |
|---|---|---|---|
| - | ~~o1~~ | ~~10~~ | ~~1~~ |
| | o2 | 50 | 2 |
| - | ~~o3~~ | ~~15~~ | ~~3~~ |
| + | o4 | 50 | 4 |
| + | o5 | 10 | 5 |

(c) ORD1 ← ORD2

ORD1

| | OID | ITEM_NO | QTY | MEMO |
|---|---|---|---|---|
| - | ~~o2~~ | ~~50~~ | ~~2~~ | ~~bar~~ |
| | o4 | 50 | 4 | memo |
| - | ~~o5~~ | ~~10~~ | ~~5~~ | ~~memo~~ |

ORD2

| | OID | ITEM_NO | QTY |
|---|---|---|---|
| - | ~~o2~~ | ~~50~~ | ~~2~~ |
| | o4 | 50 | 4 |
| - | ~~o5~~ | ~~10~~ | ~~5~~ |

(d) ORD1 ← ORD2

ORD1

| | OID | ITEM_NO | QTY | MEMO |
|---|---|---|---|---|
| | o4 | 50 | 4 | memo |
| + | o6 | 50 | 6 | |
| + | o7 | 10 | 7 | |

ORD2

| | OID | ITEM_NO | QTY |
|---|---|---|---|
| | o4 | 50 | 4 |
| + | o6 | 50 | 6 |
| + | o7 | 10 | 7 |
| + | o8 | 101 | 8 |

Figure 3.1: An example of a co-existence of schemas when one relation evolves to one relation.

Figure 3.1 (a) shows initial data as a result of schema evolution. The relation `ORD1` of schema `ver1` is transformed into the relation `ORD2` of schema `ver2`. The relation `ORD1` $= \{\langle o1, 10, 1, foo\rangle, \langle o2, 50, 2, bar\rangle, \langle o3, 15, 3, baz\rangle\}$ is transformed to the relation `ORD2` $= \{\langle o1, 10, 1\rangle, \langle o2, 50, 2\rangle, \langle o3, 15, 3\rangle\}$ by projection to drop an attribute `MEMO` from `ORD1`.

In order to run the original OMS and the modified OMS in parallel after schema evolution, updates over schema `ver1` and schema `ver2` are shared with each. Figure 3.1 (b) shows forward update sharing from schema `ver1` to schema `ver2` by following a relationship defined by schema evolution. A set of deleted tuples from `ORD1`, $\{\langle o1, 10, 1, foo\rangle, \langle o3, 15, 1, baz\rangle\}$, is transformed into a set of deleted tuples from `ORD2`, $\{\langle o1, 10, 1\rangle, \langle o3, 15, 3\rangle\}$. A set of inserted tuples to `ORD1`, $\{\langle o4, 50, 4, memo\rangle,$

$\langle o5, 10, 5, memo\rangle\}$ is transformed into a set of inserted tuples to ORD2, $\{\langle o4, 50, 4\rangle,$ $\langle o5, 10, 5\rangle\}$.

Figure 3.1 (c) and (d) show backward update sharing from schema `ver2` to `ver1`. Figure 3.1 (c) shows that a set of deleted tuples from ORD2, $\{\langle o2, 50, 20\rangle, \langle o5, 10, 5\rangle\}$, is shared with ORD1 by transforming to a set of deleted tuples from ORD1, $\{\langle o2, 50, 2, bar\rangle,$ $\langle o5, 10, 5, memo\rangle\}$. Figure 3.1 (d) shows that a set of inserted tuples to ORD2, $\{\langle o6, 50, 6\ \rangle,$ $\langle o7, 10, 7\ \rangle, \langle o8, 101, 8\ \rangle\}$, is shared with ORD1 by transforming to a set of inserted tuples to ORD1, $\{\langle o6, 50, 6,\ \ \rangle, \langle o7, 10, 7,\ \ \rangle\}$. An inserted tuple $\langle o8, 101, 8\rangle$ to the relation ORD2 is not shared because it is an order of new products specified by the value of ITEM_NO as 101 greater than 100. □

### 3.1.2 Co-Existence Strategy

This subsection shows an overview of describing a co-existence strategy and verifying the *consistency* of updates.

#### Description of Co-Existence Strategy

A co-existence strategy consists of specifications of source schema and target schema, schema evolution, and update sharing. Schema evolution is a change of data structure from source schema to target schema and a transformation of an instance as a set of tuples of source schema to one of target schema. We describe it by the *instance-level semantics* introduced in chapter 2. Data sharing is a transformation between a set of updated tuples over source schema and a set of updated tuples over target schema. This thesis follows Keller' s definition of updates [42]. An update over a schema is insertion, deletion, or replacement. Replacement is a sequence of deletion and insertion. Thus, we describe update sharing as transformations between a set of inserted/deleted tuples over source schema and a set of inserted/deleted tuples over target schema.

Taking Example 2, we illustrate how to describe a co-existence strategy by Datalog rules. Schema `ver1` as source schema and schema `ver2` as target schema are defined as follows.

$$\text{source: ver1\#}\mathtt{ord1}(O\text{:string}, I\text{:int}, Q\text{:int}, M\text{:string}). \tag{3.1}$$

$$\text{target: ver2\#}\mathtt{ord2}(O\text{:string}, I\text{:int}, Q\text{:int}). \tag{3.2}$$

$$\text{pk}(\mathtt{ord1}, [{}'O']). \tag{3.3}$$

$$\text{pk}(\mathtt{ord2}, [{}'O']). \tag{3.4}$$

Lowercase characters express a relation name by following the convention of Datalog. Variables are assigned corresponding to an order of attributes. For example, variables $(O, I, Q, M)$ correspond to attributes (`OID`, `ITEM_NO`, `QTY`, `MEMO`). Types of each attribute are specified as `string`, `int`, `int`, and `string`, respectively. A primary key is specified by describing a relation name and its variables of primary key.

We specify schema evolution by describing transformations from relations of source schema to a relation of target schema. Following Example 2, schema evolution is described by a transformation from the relation $ORD1$ of schema `ver1` to the relation $ORD2$ of schema `ver2` based on Datalog as follows:

$$\mathtt{ord2}(O, I, Q) :- \mathtt{ord1}(O, I, Q, M). \tag{3.5}$$

Predicates with relation names, `ord1` and `ord2`, correspond to relations, $ORD1$ and $ORD2$, respectively. Rule (3.5) represents projection from a relation $ORD1$ by omitting the attribute `MEMO`.

Update sharing after schema evolution consists of forward update sharing from updates over source schema to updates over target schema and backward update sharing as the inverse of forward update sharing. We assume a relationship between source schema and target schema defined by schema evolution is kept while they co-exist. Thus forward update sharing follows a relationship defined by schema evolution. We do not specifically describe forward update sharing. We specify backward update sharing by describing transformations from a set of inserted/deleted tuples against a relation of target schema to sets of inserted/deleted tuples against relations of source schema. Following Example 2, its backward update sharing is described as follows:

$$+\texttt{ord1}(O, I, Q, M) \;{:}{-}\; +\texttt{ord2}(O, I, Q), \neg\texttt{ord1}(O, I, Q, \_), I < 100, M = \text{‘ ’}. \tag{3.6}$$

$$-\texttt{ord1}(O, I, Q, M) \;{:}{-}\; -\texttt{ord2}(O, I, Q), \texttt{ord1}(O, I, Q, M), I < 100. \tag{3.7}$$

A predicate with symbols of +/− represents a set of inserted/deleted tuples against a corresponding relation. For example, $+\texttt{ord1}(O, I, Q, M)$ represents a set of inserted tuples into a relation $ORD1$. Rule (3.6) expresses that a set of inserted tuples to the relation $ORD2$ is transformed to a set of inserted tuples to the relation $ORD1$ by filling the value of MEMO as ‘ ’ when such tuples do not exist in the relation $ORD1$ and their value of ITEM_NO is less than 100. Rule (3.7) expresses that a set of deleted tuples from $ORD2$ is transformed to a set of deleted tuples from a relation $ORD1$ when these tuples exist in the relation $ORD1$ and their value of ITEM_NO is less than 100.

In Example 2, values of ITEM_NO in relations $ORD1$ and $ORD2$ must be greater than 1. We specify such constraints as followings by giving a truth constant *false* to rule head:

$$\bot() \;{:}{-}\; \texttt{ord1}(O, I, Q, M), I \leq 0. \tag{3.8}$$

$$\bot() \;{:}{-}\; \texttt{ord2}(O, I, Q), I \leq 0. \tag{3.9}$$

Note that prior researches [34, 35] define one predefined co-existence strategy of SMO for schema evolution by projection. The strategy specifies backward update sharing to transform any updates on a relation in target schema without specifying selection conditions. The example above shows another strategy of backward update sharing by specifying a selection condition.

### Verification of Co-Existence Strategy

We verify the *consistency* of updates between schemas. Intuitively, the *consistency* of updates assures that updates against a relation of schema do not cause additional updates by sharing updates following a co-existence strategy.

Additional updates occur on a relation of target schema if further tuples of insertion or deletion are resulted in schema evolution after backward update sharing based on inserted and deleted tuples against the relation of target schema. In other words, satisfying the *consistency* of updates requires that tuples of insertion and deletion resulted in schema evolution after backward update sharing must be subsets of initially inserted and deleted tuples against the relation of target schema.

We focus on the *consistency* of updates against a relation of target schema because updates against a relation of source schema do not cause additional updates following a co-existence strategy. Updates against the relation of source schema are shared with the relation of target schema following schema evolution. Since this transformation does not directly compute a set of inserted or deleted tuples to the relation of target schema, backward update sharing does not compute a set of inserted or deleted tuples to relations of source schema. Thus additional updates do not occur based on updates against the relation of source schema.

The next example shows that the co-existence strategy of Example 2 satisfies the *consistency* of updates on updates against a relation of target schema.

**Example 3.** Suppose updates shown in Figure 3.1 and the co-existence strategy consisting of schema evolution ruled by (3.6) and backward update sharing ruled by (3.6) and (3.7).

Figure 3.1 (c) shows that a set of tuples $\{\langle o2, 50, 2\rangle, \langle o5, 10, 5\rangle\}$ is initially deleted from the relation $ORD2$. The set is transformed to a set of deleted tuples $\{\langle o2, 50, 2, bar\rangle, \langle o5, 10, 5, memo\rangle\}$ from the relation $ORD1$ by the backward update sharing. By applying the schema evolution, the set of deleted tuples from $ORD1$ is transformed to a set of deleted tuples $\{\langle o2, 50, 2\rangle, \langle o5, 10, 5\rangle\}$ from $ORD2$. The result equals the initial set of deleted tuples from $ORD2$. Additional deletions do not occur based on deletion from $ORD2$ of target schema.

Figure 3.1 (d) shows that a set of tuples $\{\langle o6, 50, 6\rangle, \langle o7, 10, 7\rangle, \langle o8, 101, 8\rangle\}$ is initially inserted to $ORD2$. The set is transformed to a set of inserted tuples $\{\langle o6, 50, 6, \ \rangle, \langle o7, 10, 7, \ \rangle\}$ to the relation $ORD1$ by the backward update sharing. By applying the schema evolution, the set of inserted tuples to $ORD1$ is transformed to a set of inserted tuples $\{\langle o6, 50, 6\rangle, \langle o7, 10, 7\rangle\}$ to $ORD1$. The result is a subset of the initial set of inserted tuples to $ORD1$. Additional insertions do not occur based on insertion to $ORD2$ of target schema.

Thus the co-existence strategy satisfies the *consistency* of updates. □

Note that the definition of the *consistency* of updates is a relaxed version of the consistency of bidirectional transformation [29]. Its PutGet requires that an updated view data must be recomputed from an updated source data without loss or gain by assuming all updates of view data are transformed into the source data. In contrast, backward update sharing of a co-existence strategy may not transform any updates against a relation of target schema into relations of source schema. Therefore, as a relaxed version of bidirectional transformation's consistency, the *consistency* of updates requires that a result of schema evolution after backward update sharing generates subsets of initially inserted and deleted tuples against a relation of target schema, but does not require to generate all of them.

## 3.2 Definitions

We define a co-existence of relational database schemas.

**Definition 3.1** (Co-Existence of Relational Database Schemas). *Let a relational database schema be a set of relation names and an instance of schema be union of relations. A set of relational database schemas is a co-existence of relational database schemas when it satisfies the followings:*

1. *A new schema (target schema) is defined by schema evolution that specifies how to modify data structure and transform an instance of original schema (source schema) to an instance of target schema.*
2. *Once the instance of target schema is defined by schema evolution, updates on relations over source schema and target schema can be shared in both directions, from the instance of source schema to the instance of target schema and vice versa.*

□

We use a co-existence of schemas as an abbreviation of a co-existence of relational database schemas. Note that a relationship between source schema and target schema defined by schema evolution is kept over a period while both schemas are available and co-exist.

Based on Definition 3.1, a co-existence of schema is specified by a definition of source schema and target schema, schema evolution, and rules of update sharing.

Update sharing consists of a forward update sharing from source schema to target schema and a backward update sharing from target schema to source schema. Since a relationship defined by schema evolution is kept between schemas and a forward update sharing follows the relationship, a backward update sharing must be separately specified. Now we design a co-existence of schemas by a co-existence strategy consisting of these items.

**Definition 3.2** (Co-Existence Strategy). *A co-existence strategy consists of follows:*

1. *A definition of source schema and target schema to specify schema name and a set of relation names of each scehma.*
2. *Schema evolution to modify data structure and transform relations of source schema into a relation of target schema.*
3. *Backward update sharing to specify transformations from sets of inserted and deleted tuples against a relation of target schema into sets of inserted and deleted tuples against relations of source schema.*

□

We make a co-existence strategy between relations of source schema and a relation of target schema programmable.

## 3.3 Design Details

### 3.3.1 Syntax

Figure 3.2 shows syntax of a proposed DSL. It follows syntax of Datalog with additional things to specify a co-existence strategy. New things are <schema> to specify a definition of schema, and symbols + and − to specify <predicate> appearing in Datalog rules. A program of a co-existence strategy is a <program> which consists of a set of <statement>. Each <statement> is <schema>, <rule> or <constraint>. <schema> specifies a definition of schemas. <rule> or <constraint> is a Datalog rule described with <predicate> and <literal>. <predicate> can be described with or without a symbol + or −.

```
<program>     ::= <statement>*
<statement>   ::= <schema> | <rule> | <constraint>
<schema>      ::= source:<version>#<relname> ( <varname>:<type>
                    {, <varname>:<type> }* ).
                  | target:<version>#<relname> (<varname>:<type>
                    {, <varname>:<type>}*).
                  | pk(<relname>, [<varname> {, <varname>}* ]).
<rule>        ::= <predeicate> :− <literal> {, <literal>}*.
<constraint>  ::= ⊥() :− <literal> {, <literal> }*.
<literal>     ::= <predeicate> | not <predeicate> | <builtin> | not <builtin>
<predeicate>  ::= [ + | - ] <relname>(<term> {, <term>}*)
<builtin>     ::= <varname> ( = | <> | < | > | <= | >= ) <const>
<term>        ::= <varname> | <annonvar> | <const>
<type>        ::= <int> | <float> | <string>
```

Figure 3.2: Syntax of DSL for a co-existence strategy.

### 3.3.2 Semantics

Semantics of the DSL follows the semantics of Datalog with additional things, <schema> and symbols + and − to specify a predicate.

<schema> expresses a definition of schemas by specifying source schema or target schema, <version> for a schema version, <relname> for a relation name, <varname> for variable name, and <type> for a type of variable. Primary key constraint is described by *pk* with <relname> and <varname> of keys. <rule> expresses a transformation to specify schema evolution and backward update sharing by following the semantics of Datalog. A predicate described by <relname> with symbols + or − expresses a predicate corresponding to a set of inserted or deleted tuples. <constraint> expresses constraints other than primary key by following semantics of Datalog. Its head $\bot()$ expresses a truth constant *false*.

### 3.3.3 Description of Schema Evolution

We explain details of how to describe schema evolution. Restrictions of its description are also introduced so that a written co-existence strategy can be verified whether the *consistency* of updates is satisfied or not.

Let source schema consist of a set of relation names $s_i(\vec{x_i})$ ($i \in [1, n]$) and target schema consist of a relation name $t(\vec{y})$. Vectors $\vec{x}_i$ and $\vec{y}$ are sets of attributes of relation name $s_i$ and $t$ respectively. We denote a relation $S_i(\vec{X}_i)$ (or $S_i$ by omitting attributes $\vec{X}_i$) as a set of tuples over relation name $s_i(\vec{X}_i)$ and an instance of source schema $\mathbf{S}$ as union of relations of source schema. In the same manner, we denote a relation $T(\vec{Y})$ (or $T$) as a set of tuples over relation name $t(\vec{Y})$ and an instance of target schema $\mathbf{T}$ as union of relations of target schema.

We follow the *instance-level semantics* [11] to describe schema evolution $F$ as a mapping from source schema instance $\mathbf{S}$ to target schema instance $\mathbf{T}$ given data structure change between them.

$$\mathbf{T} = F(\mathbf{S}) \tag{3.10}$$

Since $\mathbf{T}$ is a set of all tuples of relations of target schema and a co-existence strategy handles one relation $T$ of target schema, schema evolution $F$ is a transformation $f$ from source schema instance $\mathbf{S}$ to a relation $T$ as follows:

$$T = f(\mathbf{S}) \tag{3.11}$$

We specify a transformation $f$ by a set of Datalog rules. The following format describes each Datalog rule:

$$H :- L_1, \ldots, L_k, \ldots, L_{kn}. \tag{3.12}$$

Each $L_k$ in body is a literal. A literal is a non-negated or negated predicate $s_i(\vec{X}_i)$ corresponding to a relation $S_i(\vec{X}_i)$ ($i \in [1, n]$), a non-negated or negated predicate predicate appearing in heads of other Datalog rules consisting of $f$, or a non-negated or negated built-in predicate with an arithmetic comparison operator such as $=, <$. Head $H$ is a predicate $t(\vec{Y})$ corresponding to a relation $T(\vec{Y})$ of target schema or a predicate which appears in bodies of other Datalog rules consisting of $f$ except for a predicate $s_i(\vec{X}_i)$ and built-in predicates.

Datalog rule of $f$ is restricted to be described by non-recursive GN-Datalog (Guarded Negation Datalog [10, 9, 71]) so that the *consistency* of updates for a written strategy can be verified. To describe as non-recursive Datalog rule, a literal $L_k$ in body of a rule must not be a predicate appearing its rule head. To describe as GN-Datalog rule, body of a rule must have non-negated predicates or built-in predicates with equality (=) that contain all variables occurring in a predicate of head and negated predicates in body.

**Example 4.** Following Datalog rule is not GN-Datalog because a variable $X_3$ of a negated predicate $\neg s_2(X_1, X_2, X_3)$ in its body does not appear in a non-negated predicate in its body.

$$t1(X_1, X_2) :- s_1(X_1, X_2), \neg s_2(X_1, X_2, X_3). \tag{3.13}$$

□

In Datalog program, the satisfiability is an existence of non-empty relations corresponding to predicates appearing in rule body such that these tuples result in a non-empty relation corresponding to a rule head that a set of Datalog rules in a program finally specifies. It is known that the satisfiability is decidable in Datalog program consisting of GN-Datalog rules [10, 9, 71]. Verification for the *consistency* of updates in subsection 3.3.5 exploits this property.

### 3.3.4 Description of Backward Update Sharing

We explain details of how to describe backward update sharing with some restrictions.

Given a relation $R$, let $\Delta^+_R$ be a set of tuples to insert into a relation $R$, $\Delta^-_R$ be a set of tuples to delete from a relation $R$, and $\Delta R$ be a delta relation that is a set of all tuples of $\Delta^+_R$ and $\Delta^-_R$. We denote $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ (or $\Delta^+_{S_i}$ and $\Delta^-_{S_i}$) as sets of inserted and deleted tuples against a relation $S_i(\vec{X_i})$ of source schema and $\Delta S_i(\vec{X_i})$ (or $\Delta S_i$) as a delta relation of $S_i(\vec{X_i})$. In the same manner, we denote $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$ (or $\Delta^+_T$ and $\Delta^-_T$) as sets of inserted and deleted tuples against a relation $T(\vec{Y})$ of target schema and $\Delta T(\vec{Y})$ (or $\Delta T$) as delta relation of $T(\vec{Y})$. Further, we denote $\Delta^+_{\mathrm{S}}$ and $\Delta^-_{\mathrm{S}}$ as sets of inserted and deleted tuples against source schema instance S, and $\Delta\mathrm{S}$ as a set of tuples of all $\Delta^+_{\mathrm{S}}$ and $\Delta^-_{\mathrm{S}}$.

We define backward update sharing $G$ as a transformation from a pair of source schema instance S and a delta relation $\Delta T$ into $\Delta\mathrm{S}$ as a set of all inserted and deleted tuples against S as follows:

$$\Delta \mathrm{S} = G(\mathrm{S}, \Delta T) \tag{3.14}$$

$G$ refers to source schema instance S to compensate information for transformation into ΔS from $\Delta T$ because $\Delta T$ may not have all information of S.

Since ΔS is a set of all inserted and deleted tuples against source schema instance S and S is union of all relations $S_i$ $(i \in [1, n])$, backward update sharing $G$ is a set of transformations $g^+_{s_i}$ and $g^-_{s_i}$ which transform to $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}$ as sets of inserted and deleted tuples against a relation $S_i$ from a pair of source schema instance S and a delta relation $\Delta T$.

$$\Delta^+_{S_i} = g^+_{s_i}(\mathrm{S}, \Delta T) \tag{3.15}$$
$$\Delta^-_{S_i} = g^-_{s_i}(\mathrm{S}, \Delta T) \tag{3.16}$$

We specify transformations $g^+_{s_i}$ and $g^-_{s_i}$ by sets of Datalog rules. Each Datalog rule of $g^+_{s_i}$ and $g^-_{s_i}$ are described by the following format:

$$H^+ :- L^+_1, \ldots, L^+_k, \ldots, L^+_{kn+}. \tag{3.17}$$
$$H^- :- L^-_1, \ldots, L^-_k, \ldots, L^-_{kn-}. \tag{3.18}$$

$L^+_k$ and $L^-_k$ in bodies are literals. A literal is a predicate $s_i(\vec{X_i})$ $(i \in [1, n])$, a predicate $+t(\vec{Y})$, a predicate $-t(\vec{Y})$, a predicate appearing in heads of other Datalog rules consisting of $g^+_{s_i}$ and $g^-_{s_i}$ respectively, or a built-in predicate. All predicates are non-negated or negated. Predicates $+t(\vec{Y})$ and $-t(\vec{Y})$ corresponds to $\Delta^+_T$ and $\Delta^-_T$ against a relation $T$ of target schema. Head $H^+$ is a predicate $+s_i(\vec{X_i})$ corresponding to $\Delta^+_{S_i}$ or a predicate which appears in bodies of other Datalog rules consisting of $g^+_{s_i}$ except for $s_i(\vec{X_i})$ $(i \in [1, n])$, $+t(\vec{y})$, $-t(\vec{Y})$ and built-in predicates. Head $H^-$ is a predicate $-s_i(\vec{X_i})$ corresponding to $\Delta^-_{S_i}$ or a predicate which appears in bodies of other Datalog rules consisting of $g^-_{s_i}$ except for $s_i(\vec{X_i})$ $(i \in [1, n])$, $+t(\vec{Y})$, $-t(\vec{Y})$ and built-in predicates.

A Datalog rule consisting of $g^+_{s_i}$ or $g^-_{s_i}$ is restricted to be described by non-recursive GN-Datalog with *monotonicity* and *linearity*. This thesis defines the *monotonicity* as a transformation from a set of inserted or deleted tuples to a set of inserted or deleted tuples. Description of backward update sharing is restricted so that a non-negated predicate with a symbol of − or + can not appear in body of a rule in which head is a predicate symbol of + or − respectively.

The *linearity* is a restriction that a non-negated predicate with a symbol of + or − can appear in body at most once. If $+t(\vec{Y})$ or $-t(\vec{Y})$ appears in body of the rule more than twice, the rule expresses self-join of inserted or deleted tuples. Self-join generates more tuples than actually inserted or deleted over target schema. Therefore it is prohibited that a rule has more than two non-negated $+t(\vec{Y})$ or $-t(\vec{Y})$ in body.

**Example 5.** The following Datalog rule does not satisfy a restriction of *monotonicity* because a predicate with symbol − appears in rule body while its head is a predicate with symbol +.

$$+s_1(X_1, X_2, X_3) \quad :- \quad -t1(X_1, X_2), s_1(X_1, X_2, X_3).$$

□

**Example 6.** The following Datalog rule does not satisfy a restriction of *linearity* because predicates with symbol + appear twine in rule body, i.e., $+t_1(X_1, X_2)$ and $+t_2(X_1, X_3)$.

$$+s_1(X_1, X_2, X_3) \quad :- \quad +t1(X_1, X_2), +t2(X_1, X_3), \neg s_1(X_1, X_2, X_3).$$

□

### 3.3.5 Consistency of Updates

In this subsection, we formally define the *consistency* of updates and show a method to verify it.

#### Definition of Consistency of Updates

First, we introduce an application of delta relation, then formally define the *consistency* of updates by utilizing the application of delta relation.

The application of delta relation is to compute an updated relation by applying a set of inserted and deleted tuples. Given a relation $R$, $\Delta_R^+$, and $\Delta_R^-$, sets of inserted and deleted against $R$, an updated relation $R'$ is computed by adding tuples of $\Delta_R^+$ and deleting tuples of $\Delta_R^-$ against $R$. Considering set semantics, the application of delta relation $\Delta R$ to $R$ is the following:

$$R' = R \oplus \Delta R = (R \cap \neg\Delta_R^-) \cup \Delta_R^+ \tag{3.19}$$

We define the *consistency* of updates. Recall satisfying the *consistency* of updates requires that tuples of insertion and deletion resulted from schema evolution after backward update sharing must be subsets of initially inserted and deleted tuples against a relation of target schema, respectively. Let us start to define $\Delta_T^{+\prime}$ and $\Delta_T^{-\prime}$ as sets of tuples of insertion and deletion resulted from schema evolution after backward update sharing based on delta relation $\Delta T$ as initially inserted and deleted tuples to a relation of target schema. They are computed as set difference between a result of schema evolution $F$ from an original source schema instance $\mathrm{S}$ and a result of schema evolution $F$ from an updated source schema instance $\mathrm{S}'$ by backward update sharing. Since set difference $A \setminus B = A \cap \neg B$ when $A$ and $B$ are sets, $\Delta_T^{+\prime}$ and $\Delta_T^{-\prime}$ are defined as follows:

$$\Delta_T^{+\prime} = F(\mathrm{S}') \cap \neg F(\mathrm{S}) \tag{3.20}$$

$$\Delta_T^{-\prime} = F(\mathrm{S}) \cap \neg F(\mathrm{S}') \tag{3.21}$$

An updated source schema instance $\mathrm{S}'$ is defined by applying $\Delta\mathrm{S}$ as a set of inserted and deleted tuples against an original source schema instance $\mathrm{S}$ as follows:

$$\mathrm{S}' = \mathrm{S} \oplus \Delta\mathrm{S} \tag{3.22}$$

Since $\Delta\mathrm{S}$ is computed by backward update sharing $G$ of (3.14), equations (3.20) and (3.21) are rewritten as followings:

$$\Delta_T^{+\prime} = F(\mathrm{S} \oplus G(\mathrm{S}, \Delta T)) \cap \neg F(\mathrm{S}) \tag{3.23}$$

$$\Delta_T^{-\prime} = F(\mathrm{S}) \cap \neg F(\mathrm{S} \oplus G(\mathrm{S}, \Delta T)) \tag{3.24}$$

Now satisfying the *consistency* of updates requires that $\Delta_T^{+\prime}$ is a subset of $\Delta_T^+$, a set of initially inserted tuples into a relation $T$ of target schema, and $\Delta_T^{-\prime}$ is a subset of $\Delta_T^-$, a set of initially deleted tuples from $T$.

**Definition 3.3** (Consistency of Updates). *Let* S *be source schema instance, $T$ be a relation of target schema, $\Delta_T^+$ be a set of inserted tuples to $T$, $\Delta_T^-$ be a set of deleted tuples from $T$, and $\Delta T$ be delta relation of $T$. A co-existence strategy satisfies the consistency of updates when the followings are satisfied.*

$$F(\mathrm{S} \oplus G(\mathrm{S}, \Delta T)) \cap \neg F(\mathrm{S}) \subseteq \Delta_T^+ \tag{3.25}$$

$$F(\mathrm{S}) \cap \neg F(\mathrm{S} \oplus G(\mathrm{S}, \Delta T)) \subseteq \Delta_T^- \tag{3.26}$$

□

## Verification for the Consistency of Updates

We show that verification for the *consistency* of updates is treated as a decidable satisfiability problem of Datalog program. Let us take the formula (3.25) of Definition 3.3 and show how to verify it. The formula (3.26) of Definition 3.3 is also verified in the same manner.

Because of $A \subseteq B \Leftrightarrow A \cap \neg B = \emptyset$, the formula (3.25) is transformed as follows:

$$F(\mathrm{S} \oplus G(\mathrm{S}, \Delta T)) \cap \neg F(\mathrm{S}) \cap \neg \Delta_T^+ = \emptyset \tag{3.27}$$

When equation (3.27) is satisfied, the *consistency* of updates is satisfied. Let a result of left-hand side of (3.27) be a relation $D^+$ as follows:

$$D^+ = F(\mathrm{S} \oplus G(\mathrm{S}, \Delta T)) \cap \neg F(\mathrm{S}) \cap \neg \Delta_T^+ \tag{3.28}$$

If we can derive Datalog program specifying a relation $D^+$ from a co-existence strategy, proof of equation (3.27) is equivalent to a proof of unsatisfiability of the Datalog program. The next example shows a derivation of such Datalog program consisting of GN-Datalog rules, and satisfiability of the derived Datalog program is treated as a decidable problem.

**Example 7.** Suppose the co-existence strategy consisting of schema definition specified by (3.1) – (3.4), schema evolution ruled by (3.6), backward update sharing ruled by (3.6) – (3.7), and constraints rules by (3.8) – (3.9). Since source schema consists of one relation name `ord1`, source schema instance is equivalent to a relation $ORD1$ over ord1. The relation $D^+$ of this co-existence strategy is expressed as follows:

$$D^+ = F(ORD1 \oplus G(ORD1, \Delta ORD2)) \cap \neg F(ORD1) \cap \neg \Delta^+_{ORD2} \tag{3.29}$$

where $\Delta ORD2$ is a delta relation for a relation $ORD2$ of target schema and $\Delta^+_{ORD2}$ is a set of inserted tuples to $ORD2$. Datalog rules to express $D^+$ is derived as follows:

$$+\texttt{ord1}(O, I, Q, M) :- +\texttt{ord2}(O, I, Q), \neg\texttt{ord1}(O, I, Q, \_), I < 100, M = `\ '. \tag{3.30}$$

$$-\texttt{ord1}(O, I, Q, M) :- -\texttt{ord2}(O, I, Q), \texttt{ord1}(O, I, Q, M), I < 100. \tag{3.31}$$

$$\texttt{ord1'}(O, I, Q, M) :- \texttt{ord1}(O, I, Q, M), \neg -\texttt{ord1}(O, I, Q, M). \tag{3.32}$$

$$\texttt{ord1'}(O, I, Q, M) :- +\texttt{ord1}(O, I, Q, M). \tag{3.33}$$

$$\texttt{ord2'}(O, I, Q) :- \texttt{ord1'}(O, I, Q, M). \tag{3.34}$$

$$\texttt{ord2}(O, I, Q) :- \texttt{ord1}(O, I, Q, M). \tag{3.35}$$

$$\texttt{d+}(O, I, Q) :- \texttt{ord2'}(O, I, Q), \neg\texttt{ord2}(O, I, Q), \neg +\texttt{ord2}(O, I, Q). \tag{3.36}$$

Rules (3.30) and (3.31) correspond to $\Delta ORD1 = G(ORD1, \Delta ORD2)$ where $\Delta ORD1$ is a set of inserted tuples to $ORD1$. Rules (3.32) and (3.33) correspond to $ORD1' = ORD1 \oplus \Delta ORD1$. A rule (3.34) corresponds to $ORD2' = F(ORD1')$. A rule (3.35) corresponds to $ORD2 = F(ORD1)$. Finally, a rule (3.36) corresponds to $D^+ = ORD2' \cap \neg ORD2 \cap \neg \Delta^+_{ORD2}$.

Rules (3.30), (3.31), (3.34) and (3.35) are GN-Datalog because they are GN-Datalog rules of backward update sharing and schema evolution of the co-existence strategy. Rules (3.32) and (3.33) are also GN-Datalog. When a set of tuples of updated relation is specified by applying sets of inserted and deleted tuples against the relation, all tuples have the same attributes. Thus these rules are GN-Datalog because all attributes in heads appear in bodies. Finally, rule (3.36) is GN-Datalog because all attributes in head and negated predicates in body appear in a non-negated predicate in body. Therefore, the satisfiability of the above Datalog program to transform into $D^+$ is decidable. □

In Example 7, the Datalog program for verification of the satisfiability consists of GN-Datalog rules of a co-existence strategy and GN-Datalog rules to treat sets of inserted and deleted tuples against an original relation. Therefore, its satisfiability is decidable for an arbitrary co-existence strategy.

```
// schemas:
source : ver1#$s_1(X$:int, Y:int).
target : ver2#$t(X$:int, Y:int).

// schema evolution:
$t(X, Y)$ :- $s_1(X, Y), Y = $ ‘A’.

// backward update sharing:
$+s_1(X, Y)$ :- $+t(X, Y), \neg s_1(X, Y), Y = $ ‘A’.
$+s_1(X, Y)$ :- $+t(X, Y), \neg s_1(X, Y), Y = $ ‘B’.
$-s_1(X, Y)$ :- $-t(X, Y), s_1(X, Y), Y = $ ‘A’.
$-s_1(X, Y)$ :- $-t(X, Y), s_1(X, Y), Y = $ ‘B’.
```

Figure 3.3: Co-existence strategy for schema evolution by selection.

## 3.4 Examples of Strategies

This subsection shows examples of co-existence strategies: co-existence of schemas when a schema is evolved by selection, projection, join, union, and set difference.

### Co-Existence Strategy for Schema Evolution by Selection

Figure 3.3 shows a co-existence strategy for schema evolution by selection. A rule of schema evolution transforms a relation $S_1$ of source schema (ver1) to a relation $T$ of target schema (ver2) if tuples of $S_1$ satisfy the condition of $Y =$ ‘A’. Rules of backward update sharing transform inserted and deleted tuples against the relation $T$ into inserted and deleted tuples against the relation $S_1$ if tuples of $T$ satisfy the condition $Y =$ ‘A’ or $Y =$ ‘B’. Note that any conditions can be designed for schema evolution and backward update sharing. No selection condition of backward update sharing is also available.

// schemas:
source : ver1#$s_1(X\text{:int}, Y\text{:int}, Z\text{:string})$.
target : ver2#$t(X\text{:int}, Y\text{:int})$.

// schema evolution:
$t(X, Y) \text{ :- } s_1(X, Y, Z)$.

// backward update sharing:
$+s_1(X, Y, Z) \text{ :- } +t(X, Y), \neg s_1(X, Y, \_), Z = \text{‘w’}$.
$-s_1(X, Y, Z) \text{ :- } -t(X, Y), s_1(X, Y, Z)$

Figure 3.4: Co-existence strategy for schema evolution by projection.

### Co-Existence Strategy for Schema Evolution by Projection

Figure 3.4 shows a co-existence strategy for schema evolution by projection. A rule of schema evolution transforms a relation $S_1$ of source schema (ver1) into a relation $T$ of target schema (ver2) by dropping attribute $Z$ because it appears in body of the rule but does not appear in head of the rule.

Backward update sharing is that inserted and deleted tuples against $T$ are shared with the relation $S_1$. Its rules transform inserted tuples to a relation $T$ to tuples to be inserted into $S_1$ by giving default value w of attribute $Z$, and deleted tuples from a relation $T$ to tuples to be deleted from $S_1$ when tuples' values of attributes $X$ and $Y$ are the same.

### Co-existence Strategy for Schema Evolution by Join

Figure 3.5 shows a co-existence strategy for schema evolution by join. The rule of schema evolution transforms the relations $S_1$ and $S_2$ of source schema (ver1) to the relation $T_1$ of target schema (ver2) by join if tuples of relations $S_1$ and $S_2$ have the same value of the primary key, attribute $X$.

The rules of backward update sharing transform inserted and deleted tuples against $T_1$ to inserted and deleted tuples against $S_1$ and $S_2$. The rule that head is $-s_1(X, Y)$

```
// schemas:
source : ver1#$s_1(X$:int, $Y$:int$)$.
source : ver1#$s_2(X$:int, $Z$:int$)$.
target : ver2#$t(X$:int, $Y$:int, $Z$:int$)$.
pk($s_1$, [‘$X$’]).
pk($s_2$, [‘$X$’]).
pk($t$, [‘$X$’]).

// schema evolution:
$t(X, Y, Z)$ :− $s_1(X, Y), s_2(X, Z)$.

// backward update sharing:
$+s_1(X, Y)$ :− $+t(X, Y, Z), \neg s_1(X, Y), s_2(X, Z)$.
$+s_2(X, Z)$ :− $+t(X, Y, Z), \neg s_2(X, Z), s_1(X, Y)$.
$-s_1(X, Y)$ :− $-t(X, Y, \_), \neg +t(X, Y, \_), s_1(X, Y), s_2(X, \_)$.
$-s_2(X, Z)$ :− $-t(X, \_, Z), \neg +t(X, \_, Z), s_1(X, \_), s_2(X, Z)$.
```

Figure 3.5: Co-existence strategy for schema evolution by join.

expresses a transformation of a set of deleted tuples from $T_1$ into a set of deleted tuples from $S_1$. Its body has $-t_1(X, Y, \_)$ and negated $+t_1(X, Z, \_)$ to handle a replacement by a sequence of deletion and insertion. Suppose that a replacement of a tuple $(1, 1, 1)$ in the relation $T_1$ to a tuple $(1, 1, 2)$ is a sequence of deletion $(1, 1, 1)$ and insertion $(1, 1, 2)$. Value of attribute $Z$ is replaced from 1 to 2. Since only the relation $S_2$ has attribute $Z$ and the relation $S_1$ does not have it, a replacement of the tuple in $S_1$ is not required when values of an attribute $Z$ in a relation $T_1$ is replaced and values of other attributes $X$ and $Y$ are not changed. This transformation is ruled by having predicates $-t_1(X, Y, \_)$ and negated $+t_1(X, Z, \_)$ in body of the rule. This is also equipped in the rule that head is $-s_2(X, Y)$ in the same manner.

```
// schemas:
source : ver1#s1(X:int, Y:int).
source : ver1#s2(X:int, Y:int).
target : ver2#t(X:int, Y:int).
pk(s1, ['X']).
pk(s2, ['X']).
pk(t, ['X']).

// schema evolution:
t(X, Y) :− s1(X, Y).
t(X, Y) :− s2(X, Y), ¬s1(X, _).

// backward update sharing:
 +s1(X, Y) :− +t(X, Y), ¬s1(X, Y), ¬s2(X, Y), Y ≥ 1.
 +s2(X, Y) :− +t(X, Y), ¬s1(X, Y), ¬s2(X, Y), Y = 1.
 −s1(X, Y) :− −t(X, Y), s1(X, Y).
 −s2(X, Y) :− −t(X, Y), s2(X, Y), ¬s1(X, _).
 −s2(X, Y) :− −t(X, Y1), ¬ +t(X, Y), s2(X, Y), s1(X, Y1).
```

Figure 3.6: Co-existence strategy for schema evolution by union.

### Co-existence Strategy for Schema Evolution by Union

Figure 3.6 shows a co-existence strategy for schema evolution by union. The rules of schema evolution transform the relations $S_1$ and $S_2$ of source schema (ver1) to the relation $T$ of target schema (ver2) by giving a priority to $S_1$. If tuples having the same value of the primary key $X$ exist in both $S_1$ and $S_2$, tuples of $S_1$ are transformed into tuples of $T$ of target schema.

The rules of backward update sharing transform inserted and deleted tuples against the relation $T$ to inserted and deleted tuples against $S_1$ and $S_2$. Inserted tuples to $T_1$ are transformed to inserted tuples to $S_1$ and $S_2$ if they are not exist in both $S_1$ and $S_2$ and satisfy the condition $Y \geq 1$ and $Y = 1$ respectively. Deleted tuples from $T$ are

// schemas:
source : ver1#$s_1(X$:int, $Y$:int).
source : ver1#$s_2(X$:int, $Y$:int).
target : ver2#$t(X$:int, $Y$:int).
pk($s_1$, [‘$X$’]).
pk($s_2$, [‘$X$’]).
pk($t$, [‘$X$’]).

// schema evolution:
$t(X, Y) :- s_1(X, Y), \neg s_2(X, Y)$

// backward update sharing:
$+s_1(X, Y) :- +t(X, Y), \neg s_1(X, Y), \neg s_2(X, Y).$
$-s_1(X, Y) :- -t(X, Y), s_1(X, Y), \neg s_2(X, Y).$

Figure 3.7: Co-existence strategy for schema evolution by set difference.

transformed to deleted tuples from $S_1$ and $S_2$ by satisfying a condition to give priority to $S_1$. Furthermore, the last rule expresses a transformation of deleted tuples from $T$ to $S_2$ if tuples of $T$ are deleted, tuples that have the same values of the primary key $X$ are not inserted into $T$, and tuples that have the same values of the primary key $X$ exist in $S_1$ and $S_2$. Recall that the *consistency* of updates requires additional inserted or deleted tuples must not appear as a result of schema evolution after backward updated sharing. Suppose a tuple $(1, 1)$ exist in $S_1$ and a tuple $(1, 2)$ exist in $S_2$. The rule of schema evolution transforms them to a tuple $(1, 1)$ of $T$ because of priority to $S_1$. If a tuple $(1, 1)$ of $T$ is deleted and transformed to deletion of $(1, 1)$ from $S_1$ but is not transformed to deletion of $(1, 2)$ from $S_2$, a tuple $(1, 2)$ newly appears in $T$ by following schema evolution. The last rule is to avoid such violence of the *consistency* of updates.

### Co-existence Strategy for Schema Evolution by Set Difference

Figure 3.7 shows a co-existence strategy for schema evolution by set difference. The rule of schema evolution transforms tuples of the relations $S_1$ of source schema (ver1) to tuples of the relation $T$ of target schema (ver2) if such tuples do not exist in the relation $S_2$ of source schema.

The rules of backward update sharing specify transformations to sets of inserted and deleted tuples against the relation $S_1$. If inserted tuples to $T$ do not exist in $S_1$ and $S_2$, they are transformed into a set of inserted tuples to $S_1$. If deleted tuples from $T$ exist in $S_1$ and do not exist in $S_2$, they are transformed into a set of deleted tuples from $S_1$.

Based on schema evolution by set difference, it could be another transformation that insertions (deletions) against $T$ are transformed into deletions (insertions) against $S_2$. However, such transformations are not specified because Datalog rules of such transformations are not *monotonic*.

### Co-existence Strategy for Schema Evolution by Join and Projection

Figure 3.8 shows a co-existence strategy for schema evolution by join and projection. In comparing with backward update sharing of the co-existence strategy for schema evolution by join, a predicate, $W =$ '`w`', to set a default value `w` to the attribute $W$ is added because the attribute $W$ is projected away by schema evolution.

## 3.5 Related Work

This section shows related work about specifying schema evolution and co-existence of schemas.

### Schema Evolution

Schema evolution has been widely drawn the attention of database researchers to contribute continuous evolution of information systems with changes of databases. Due to a wide research domain, we will not go over thousand of works for schema evolution but introduce relevant works for the co-existence of schemas.

To highlight feature of schema evolution, Skoulis [62] and Vassiliadis [74] report investigation result about open source databases. In contrast to the continuous

```
// schemas:
source : ver1#$s_1$($X$:int, $Y$:int).
source : ver1#$s_2$($X$:int, $Z$:int, $W$:string).
target : ver2#$t$($X$:int, $Y$:int, $Z$:int).
pk($s_1$, [‘$X$’]).
pk($s_2$, [‘$X$’]).
pk($t$, [‘$X$’]).

// schema evolution:
$t(X, Y, Z) :- s_1(X, Y), s_2(X, Z, W).$

// backward update sharing:
$+s_1(X, Y)$       $:- +t(X, Y, Z), \neg s_1(X, Y), s_2(X, Z).$
$+s_2(X, Z, W)$ $:- +t(X, Y, Z), \neg s_2(X, Z, \_), W = ‘w’, s_1(X, Y).$
$-s_1(X, Y)$       $:- -t(X, Y, \_), \neg +t(X, Y, \_), s_1(X, Y), s_2(X, \_, \_).$
$-s_2(X, Z, W)$ $:- -t(X, \_, Z), \neg +t(X, \_, Z), s_1(X, \_), s_2(X, Z, W).$
```

Figure 3.8: Co-existence strategy for schema evolution by join and projection.

evolution of information systems in general, schema evolution of databases shows burst in their whole lifetime. For example, early periods of the database life demonstrate a higher level of evolutionary changes than later ones. The necessity to support such burst changes of database underpins our work to make co-existence strategies programmable.

Roddick[58, 59] provides a bibliography of schema evolution in 1992 and 1995. It highlights two aspects of schema evolution: modification of schema definition from source schema to target schema without data loss and access to data over any schema through an interfaced schema when multiple schemas exist as a result of schema modification. More recently in 2016, Caruccio et al. [14] survey schema evolution from information capacity perspective whether schema evolution lost information or not [38, 54], and categorize methods to specify schema evolution into three approaches: operation-based approach, mapping-based approach, and hybrid approach. In the following, we show the related work of each approach.

Note that schema evolution assumes to modify a schema into a new schema without data loss but does not assume the co-existence of schemas to share independent updates over schemas.

### Operation-Based Approach for Schema Evolution

Roddick [59] insists that the modifications algebraic operations on a schema should express the modifications of a schema, and changes of a schema on a larger scale might be made available through a composition of elementary operations. Curino et al. [20] investigates evolutional steps of MediaWiki as a backend technology of Wikipedia and propose a set of primitive operations, SMOs (Schema Modification Operations), to describe schema evolution. These SMOs are designed so that a composition of SMOs can describe the actual schema evolution of MediaWiki. Herrmann et al. [33] propose CODEL as enhanced SMO. Since practitioners write transformations for schema evolution by SQL, the author redesigns SMO to be relationally complete.

These SMOs are good for describing schema evolution. However, describing update sharing to program co-existence strategy is out of scope. This thesis proposes a DSL to describe co-existence strategies consisting of schema evolution and update sharing.

### Mapping-Based Approach for Schema Evolution

Since a mapping between source schema and target schema specifies schema evolution, the idea of describing a mapping has been universalized to the abstract level of schema evolution. Bernstein and Melnick [11] propose Model Management 2.0 to manage mappings among heterogeneous metamodels. Schema evolution is treated as a mapping between one metamodel as the source schema and another metamodel as the target schema. Domínguez et al. [24] propose MeDEA as a model-driven schema evolution that schema changes are specified in an Entity-Relationship diagram, and it is automatically mapped to the underlying relational database model. Wall and Angryk [75] propose ScaDaVar that gives a schema version control mechanism by specifying mapping between schema versions. The schema version control realizes making branched schema version from an original schema version and merging branched schema versions into one schema version.

Again they are good for describing schema evolution. However, a description of update sharing to program co-existence strategy is out of scope. Since it is challenging to classify a variety of rules for update sharing into limited numbers of basic operations, this thesis proposes mapping-based DSL to describe co-existence strategies.

### Hybrid Approach for Schema Evolution

The hybrid approach is to describe schema evolution by operation-based DSL and automatically generates a mapping between schemas. Schuler and Kessleman [60] propose CHiSEL as domain-specific SMO for a scientific database. It shows the usefulness of the hybrid approach that designs SMO for domain-specific matter and provides a mapping of schemas of generally used databases. Curino et al. [19, 18, 18] propose PRISM/PRISM++ based on the previously proposed SMO from investigation of MediaWiki [20]. In general, query rewriting is problematic when schema evolution discards information, for example, deleting attributes, because lost information cannot be queried. By introducing the idea of inverse SMO and primitive operators of integrity constraints, PRISM/PRISM++ generates transformations from target schema to source schema. The transformations inverse data of target schema so that queries and updates issued over source schema work. Moon et al. [55] propose PRIMA. By describing schema evolution based on SMO, PRIMA translates queries and updates on target schema retrospectively to legacy schemas to access data over them.

PRIMA is the first step toward the co-existence of schemas. Since the co-existence of schemas accommodates update sharing, it is required to transform updates against target schema to updates against source schema. PRIMA provides a feature to generate a mapping from target schema to source schema under schema evolution though it is limited to relationships defined by SMOs. In order to realize the co-existence of schemas, this thesis proposes methods to handle schema evolution and update sharing under circumstances that schema evolution may or may not lose information and arbitrary updates occurs against each schema after schema evolution.

### Co-Existence of Schemas

To realize the co-existence of schemas, Herrmann et al. [34, 35] enhance CoDEL [33] to BiDEL of MSVDB to describe co-existence strategies. As we already mentioned in

Chapter 1, this SMO lacks the flexibility to arbitrarily design a co-existence strategy by specifying schema evolution and backward update sharing. This thesis proposes a DSL to specify schema evolution and backward update sharing based on Datalog.

# 4
# Realization of Co-Existence Strategies on Source-Side Database

This chapter presents deriving BXs between the source-side database and view instances of schemas as a realization of co-existence strategies on the source-side database in the overall procedure shown in Figure 1.2 of Chapter 1. We start with an overview and then explain how to derive BXs and show their evaluation with experimental results. Related work is explained at the end of this chapter.

## 4.1 Overview

Before discussing the details of deriving BXs, we present an overview of what BXs realize co-existence strategies on the source-side database, how to derive them, and their relation with the later sections.

### BX to Realize Co-Existence Strategies

Recall a co-existence strategy specifies a relationship between source schema instance S and a relation $T$ of target schema by schema evolution $F$ and backward updated sharing $G$ (Figure 4.1 (a)). A source schema instance **S** is union of relations $S_i$ ($i \in [1, n]$) of source schema. We realize co-existence strategies by two types of bidirectional transformations following the view-based approach (Figure 4.1 (b)). We make relations $S_i$ ($i \in [1, n]$) and $T$ turned to view instances. $\mathbf{D}_s$ is the source-side database. It is union of base relations corresponding to relations of source schema and auxiliary relations for supplemental information. In this chapter, we shorten the source-side database as the database unless specified otherwise.

For each $i$ ($i \in [1, n]$), $BX_{src.i}$ is a bidirectional transformation between the database $\mathbf{D}_s$ and the view instance $S_i$ of source schema. Its *get* (denoted as $get_{src.i}$) and *put* (denoted as $put_{src.i}$) are as followings:

$$get_{src.i}(\mathbf{D}_s) = S_i \tag{4.1}$$

$$put_{src.i}(\mathbf{D}_s, S_i') = \mathbf{D}_s' \tag{4.2}$$

where $S_i'$ is the updated view instance and $\mathbf{D}_s'$ is the updated database. We derive this BX as identity mapping between the view instance $S_i$ and the corresponding base relation and transformations from $S_i'$ to the updated auxiliary relation.

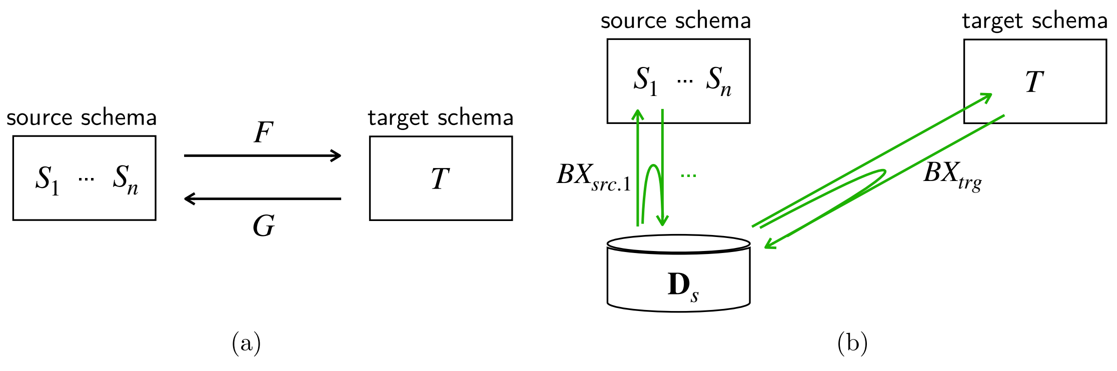


Figure 4.1: Realization of a co-existence strategy by bidirectional transformations on the source-side database.

$BX_{trg}$ is a bidirectional transformation between the database $\mathbf{D}_s$ and the view instance $T$ of target schema. Its *get* (denoted as $get_{trg}$) and *put* (denoted as $put_{trg}$) are as followings:

$$get_{trg}(\mathbf{D}_s) = T \tag{4.3}$$

$$put_{trg}(\mathbf{D}_s, T') = \mathbf{D}_s{}' \tag{4.4}$$

where $T'$ is the updated view instance. We derive this bidirectional transformation so that it realizes a co-existence strategy between the base relations of the database and the view instance $T$ of target schema while unshared tuples of $T$ with view instance $S_i$ of source schema are separately stored in the auxiliary relations of the database.

These two types of BXs make updates against the view instance of source schema (target schema) shared with the view instance of target schema (source schema) through updates of the database. In the forward direction, $put_{src.i}$ transforms the updated $S_i{}'$ of source schema into the updated database $\mathbf{D}_s{}'$. And then, $get_{trg}$ transforms $\mathbf{D}_s{}'$ into the updated $T'$. In the backward direction, $put_{trg}$ then $get_{src.i}$ transforms the updated $T'$ into the updated $S_i{}'$ through the updated database $\mathbf{D}_s{}'$.

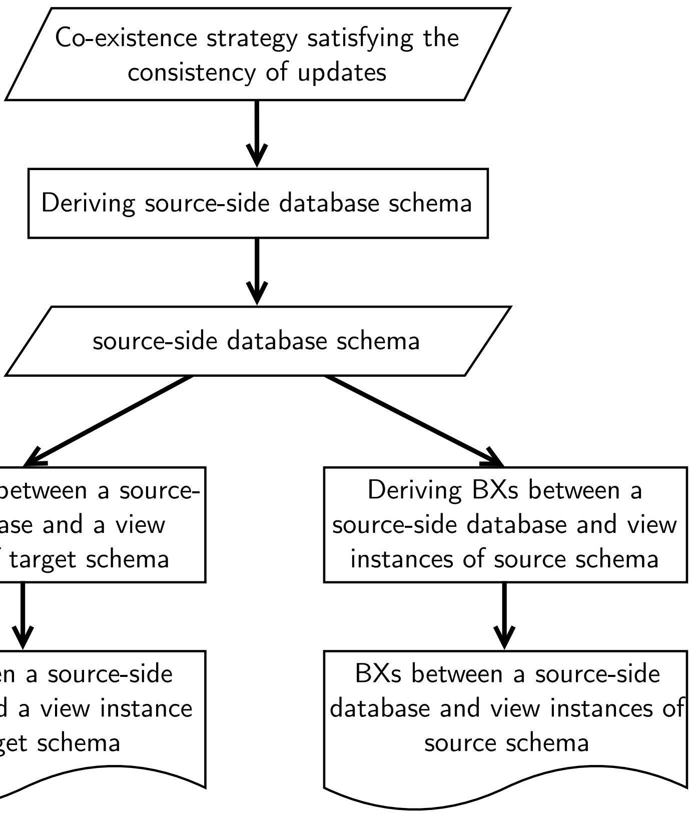

Co-existence strategy satisfying the consistency of updates
Deriving source-side database schema
source-side database schema
Deriving BX between a source-side database and a view instance of target schema
Deriving BXs between a source-side database and view instances of source schema
BX between a source-side database and a view instance of target schema
BXs between a source-side database and view instances of source schema

Figure 4.2: Procedure of deriving BXs between a source-side database and view instances of schemas.

## Procedure to Derive BXs

Figure 4.2 shows a procedure of deriving BXs between the source-side database and the view instances of schemas. This is the following procedure after a given co-existence strategy is verified whether the *consistency* of updates is satisfied in Figure 1.2.

The first step is deriving the source-side database schema. Given a co-existence strategy that specifies source schema and target schema, the source-side database schema is defined so that the database is union of the base relations corresponding to the relations of source schema and the auxiliary relations required for a realization of a co-existence strategy by $BX_{trg}$.

The next step is deriving BXs based on the following policy:

1. $BX_{trg}$ as bidirectional transformation between the database and the view instance

$T$ of target schema is derived so that a co-existence strategy is realized between the base relations of the database and the view instance $T$. *get* of $BX_{trg}$ follows schema evolution specified in a co-existence strategy. *put* of $BX_{trg}$ follows backward update sharing specified in a co-existence strategy.

2. Updates against the view instance $T$ are transformed to auxiliary relations as needed so that the updated view instance can be computed from the updated base and auxiliary relations without loss or gain. Recall the *consistency* of updates assures that schema evolution after backward update sharing results in a subset of initially inserted (deleted) tuples against a relation of target schema. Thus some initially inserted (deleted) tuples may be lost (gained) in the result. Suppose schema evolution after backward update sharing between the base relations and the updated view instance of target schema results in lost and gained tuples. In that case, they are stored in an auxiliary relation for lost tuples and an auxiliary relation for gained tuples, respectively. The source-side database schema defines two auxiliary relation names for such two auxiliary relations.

3. $BX_{src.i}$ for each $i$ ($i \in [1, n]$) is derived as identity mapping between the base relation and the view instance $S_i$ and additional transformations of its *put* to delete tuples of auxiliary relations for lost tuples. When schema evolution transforms inserted and deleted tuples against $S_i$ to inserted and deleted tuples against $T$ and such tuples exist in the auxiliary relation for lost tuples, these tuples are not lost tuples anymore. Thus such tuples are deleted from the auxiliary relation.

In the following section, we explain the details of deriving BXs shown above. Subsection 4.2.1 shows an outline of deriving BXs through examples, Subsection 4.2.2 explains deriving the source-side database schema in the procedure, Subsection 4.2.3 explains deriving BX between the source-side database and the view instance of target schema, and Subsection 4.2.4 explains deriving BXs between the source-side database and the view instances of source schema.

## 4.2 Deriving BX to Realize Co-Existence Strategies

This section explains how to derive BXs from a given co-existence strategy. We first give its outline, then explain derivation algorithms and the correctness of them.

### 4.2.1 Outline

We show an outline of deriving bidirectional transformations based on the policy in the overview thorough examples.

#### Derivation of Bidirectional Transformations between View Instances and Base Relations

Let us start with a simple co-existence strategy that does not require auxiliary relations to realize the strategy. Let $S_1(X, Y, Z, W)$ be a relation of source schema and $T(X, Y, Z)$ be a relation of target schema. A co-existence strategy consists of schema evolution by projection from $S_1(X, Y, Z, W)$ to $T(X, Y, Z)$ and backward update sharing to set a default value into a projected away attribute $W$. Datalog rules of the strategy are as follows:

schema evolution:

$$t(X, Y, Z) :- s_1(X, Y, Z, W). \tag{4.5}$$

backward update sharing:

$$+s_1(X, Y, Z, W) :- +t(X, Y, Z), \neg s_1(X, Y, Z, \_), W = \text{‘}\mathtt{w}\text{’}. \tag{4.6}$$

$$-s_1(X, Y, Z, W) :- -t(X, Y, Z), s_1(X, Y, Z, W). \tag{4.7}$$

By following the policy of deriving BXs, we define the source-side database schema. Let $B_{S_1}(X, Y, Z, W)$ be a base relation corresponding to the relation $S_1(X, Y, Z, W)$ of source schema. Let $A_T^{lost}(X, Y, Z)$ and $A_T^{gain}(X, Y, Z)$ be auxiliary relations for lost and gained tuples even though auxiliary relations are not utilized in this example.

By following the first policy, we derive transformations of bidirectional transformation $BX_{trg}$ between the base relation $B_{S_1}(X, Y, Z, W)$ and the view instance $T(X, Y, Z)$

of target schema. Rule (4.5) of schema evolution are replaced to $get_{trg}$ of $BX_{trg}$ by replacing the predicate symbol $s_1$ to $b\_s_1$ for the base relation.

$get_{trg}$:

$$t(X, Y, Z) :- b\_s_1(X, Y, Z, W). \tag{4.8}$$

Rules (4.6) – (4.7) of backward update sharing are replaced to $pur_{trg}$ of $BX_{trg}$ by replacing the predicate symbol $s_1$ to $b\_s_1$ and adding supplemental rules. Following Datalog rules are derived.

$put_{trg}$:

$$+b\_s_1(X, Y, Z, W) :- +t(X, Y, Z), \neg b\_s_1(X, Y, Z, \_), W = \text{‘w’}. \tag{4.9}$$
$$-b\_s_1(X, Y, Z, W) :- -t(X, Y, Z), b\_s_1(X, Y, Z, W). \tag{4.10}$$
$$+t(X, Y, Z) :- t'(X, Y, Z), \neg t(X, Y, Z). \tag{4.11}$$
$$-t(X, Y, Z) :- \neg t'(X, Y, Z), t(X, Y, Z). \tag{4.12}$$
$$b\_s_1{}'(X, Y, Z, W) :- b\_s_1(X, Y, Z, W), \neg - b\_s_1(X, Y, Z, W). \tag{4.13}$$
$$b\_s_1{}'(X, Y, Z, W) :- +b\_s_1(X, Y, Z, W). \tag{4.14}$$

where predicates with symbols $+t$, $-t$, $t'$, and $b\_s_1{}'$ correspond to relations $\Delta_T^+$, $\Delta_T^-$, $T'$, and $B_{S_1}{}'$ respectively. Relations $\Delta_T^+$ and $\Delta_T^-$ are sets of inserted and deleted tuples against the view instance $T$. $T'$ is the updated view instance of target schema and $B_{S_1}{}'$ is the updated base relation.

Rules (4.9) – (4.10) are derived from rules (4.6) – (4.7) of backward update sharing. Rules (4.11) – (4.14) are supplemental rules. Rules (4.11) – (4.12) computes relations $\Delta_T^+$ and $\Delta_T^-$ for predicate $+t(X, Y, W)$ and $-t(X, Y, W)$ appearing in bodies of rules (4.9) – (4.10). Since $\Delta_T^+$ and $\Delta_T^-$ are sets of inserted and deleted tuples against the view instance $T$, they are computed as results of set difference between the non-updated original view instance $T$ and the updated view instance $T'$[1]. Rules (4.13) – (4.14) express transformations to the updated base relation $B_{S_1}{}'$ by application of delta relation as

[1] $T$ is computed by $get_{trg}$ from a base relation.

${B_{S_1}}' = (B_{S_1} \cap \neg\Delta^{-}_{B_{S_1}}) \cup \Delta^{+}_{B_{S_1}}$.

By following the second policy, we consider necessity of Datalog rules to transform into the auxiliary relations. Rules (4.9) – (4.14) of $put_{trg}$ derived from backward update sharing transform any inserted (deleted) tuples against the view instance $T$ to inserted (deleted) tuples against the base relation $B_{S_1}$ if they do not exist (exist) in it. Then rule (4.8) of $get_{trg}$ derived from schema evolution transforms the updated base relation ${B_{S_1}}'$ to the updated view instance $T'$ without loss or gain. Datalog rules to transform lost or gained tuples to the auxiliary relations are not necessary.

By following the third policy, bidirectional transformation $BX_{src.1}$ is derived as identity mapping between the base relation $B_{S_1}$ and the view instance $S_1$ of source schema. Since it is rather simple, we skip to show it here and explain details in Subsection 4.2.4 later.

The following example shows another co-existence strategy that requires Datalog rules to utilize auxiliary relations.

### Derivation of Bidirectional Transformations Utilizing Auxiliary Relations

We show a derivation of Datalog rules to transform into auxiliary relations. Suppose a co-existence strategy consisting of schema evolution expressed by rule (4.5) and the following backward update sharing added selection condition $Y < 100$.

$$+s_1(X,Y,Z,W) :- +t(X,Y,Z), \neg s_1(X,Y,Z,\_), Y < 100, W = \text{‘w’}. \tag{4.15}$$

$$-s_1(X,Y,Z,W) :- -t(X,Y,Z), s_1(X,Y,Z,W), Y < 100. \tag{4.16}$$

By following the first policy, Datalog rules of $put_{trg}$ of $BX_{trg}$ are derived by replacing the predicate symbol $s_1$ appearing in rules (4.15) – (4.16) to $b\_s_1$ as follows and by adding rules (4.11) – (4.14).

$$+b\_s_1(X,Y,Z,W) :- +t(X,Y,Z), \neg b\_s_1(X,Y,Z,\_), W = \text{‘w’}, Y < 100. \tag{4.17}$$

$$-b\_s_1(X,Y,Z,W) :- -t(X,Y,Z), b\_s_1(X,Y,Z,W), Y < 100. \tag{4.18}$$

These rules transform inserted and deleted tuples against the view instance $T(X,Y,Z)$ to inserted and deleted tuples against the base relation $B_{S_1}(X,Y,Z,W)$ if their value of $Y$ is less than 100. Otherwise, they do not transform inserted and deleted tuples against

$T$ to the base relation $B_{S_1}$. Since such tuples are not transformed to the base relation, transformation from the updated base relation by rule (4.8) of schema evolution does not compute all originally inserted and deleted tuples against $T$. Thus lost and gained tuples occur.

By following the second policy, such lost and gained tuples are inserted into the auxiliary relations $A_T^{lost}(X, Y, Z)$ and $A_T^{gain}(X, Y, Z)$ respectively so that the updated view instance of target schema can be computed from the updated database without loss or gain. We define the following Datalog rules to express insertion of lost and gained tuples to auxiliary relations:

$$t''(X, Y, Z) :- b\_s_1'(X, Y, Z, W). \tag{4.19}$$

$$+a\_lost\_t(X, Y, Z) :- t'(X, Y, Z), \neg t''(X, Y, Z), \neg a\_lost\_t(X, Y, Z). \tag{4.20}$$

$$+a\_gain\_t(X, Y, Z) :- \neg t'(X, Y, Z), t''(X, Y, Z), \neg a\_gain\_t(X, Y, Z). \tag{4.21}$$

where predicates $t''(X, Y, Z)$, $a\_lost\_t(X, Y, Z)$, $a\_gain\_t(X, Y, Z)$, $+a\_gain\_t(X, Y, Z)$, and $+a\_gain\_t(X, Y, Z)$ correspond to relations $T''$, $A_T^{lost}$, $A_T^{gain}$, $\Delta^+_{A_T^{lost}}$, and $\Delta^+_{A_T^{gain}}$ respectively. Rule (4.19) expresses a transformation to a relation $T''$ as a result of schema evolution from the updated base relation $B_{S_1}{}'$ that is updated by backward transformation of rule (4.17) and (4.18). Rule (4.20) expresses that lost tuples which exist in the updated view instance $T'$ but do not exist in $T''$ are inserted into the auxiliary relation $A_T^{lost}$ if they do not exist in it. Rule (4.21) expresses that gained tuples which do not exist in the updated view instance $T'$ but exist in $T''$ are inserted into the auxiliary relation $A_T^{gain}$ if they do not exist in it.

Datalog rules to delete lost and gained tuples from the auxiliary relations are defined as follows:

$$-a\_lost\_t(X, Y, Z) :- a\_lost\_t(X, Y, Z), \neg t'(X, Y, Z). \tag{4.22}$$

$$-a\_gain\_t(X, Y, Z) :- a\_gain\_t(X, Y, Z), t'(X, Y, Z). \tag{4.23}$$

$$-a\_gain\_t(X, Y, Z) :- a\_gain\_t(X, Y, Z), \neg t''(X, Y, Z). \tag{4.24}$$

where predicates $-a\_gain\_t(X, Y, Z)$, and $-a\_gain\_t(X, Y, Z)$ correspond to relations $\Delta^-_{A_T^{lost}}$, and $\Delta^-_{A_T^{gain}}$ respectively. If lost tuples in the auxiliary relation $A_T^{lost}$ do not exist in the updated view instance $T'$, they are not lost tuples anymore and must be deleted from $A_T^{lost}$. Rule (4.22) expresses such transformation to a set of deleted tuples from $A_T^{lost}$. If gained tuples in the auxiliary relation $A_T^{gain}$ exist in the updated view instance $T'$ or do not exist in $T''$ as a result of schema evolution after backward update sharing, they are not gained tuples anymore. They must be deleted from $A_T^{gain}$. Rule (4.23) and (4.24) express such transformation to a set of deleted tuples from $A_T^{gain}$.

The updated auxiliary relations $A_T^{lost'}$ and $A_T^{gain'}$ are defined by following rules as applications of delta relations.

$$a\_lost\_t(X,Y,Z)' :- a\_lost\_t(X,Y,Z), \neg -a\_lost\_t(X,Y,Z). \tag{4.25}$$

$$a\_lost\_t(X,Y,Z)' :- +a\_lost\_t(X,Y,Z). \tag{4.26}$$

$$a\_gain\_t(X,Y,Z)' :- a\_gain\_t(X,Y,Z), \neg -a\_gain\_t(X,Y,Z). \tag{4.27}$$

$$a\_gain\_t(X,Y,Z)' :- +a\_gain\_t(X,Y,Z). \tag{4.28}$$

Finally $put_{trg}$ is derived as rules (4.11) – (4.14) and (4.17) – (4.28).

We derive $get_{trg}$ to compute the view instance $T$ without loss or gain by adding lost tuples in the auxiliary relation $A_T^{lost}$ and deleting gained tuples in the auxiliary relation $A_T^{gain}$ against a result of schema evolution from the base relation $B_{S_1}$. Datalog rules to express such transformation are defined as follows:

$$t\_evo(X,Y,Z) :- b\_s_1(X,Y,Z,W). \tag{4.29}$$

$$t(X,Y,Z) :- t\_evo(X,Y,Z), \neg a\_gain\_t(X,Y,Z). \tag{4.30}$$

$$t(X,Y,Z) :- a\_lost\_t(X,Y,Z). \tag{4.31}$$

By following the third policy, rule (4.29) is derived from rule (4.5) of schema evolution by replacing a predicate symbol $s_1$ to $b\_s_1$ and replacing a predicate symbol $t$ in head to $t\_evo$ corresponding to a relation $T^{evo}$. Rules (4.30) and (4.31) express a transformation that a set of gained tuples as $A_T^{gain}$ is deleted and a set of lost tuples as $A_T^{lost}$ is added against a relation $T_{evo}$.

By following the third policy, bidirectional transformation $BX_{src.1}$ is identity mapping between a base relation $B_{S_1}(X,Y,Z,W)$ and a view instance $S_1(X,Y,Z)$ of source schema and transformations to delete tuples of auxiliary relations. As with the former example, we show details of it in Subsection 4.2.4 later.

In the following subsections, we show algorithms to derive the source-side database

**Algorithm 4.1** Deriving Source-Side Database Schema

```
Input: a co-existence strategy $P$
Output: a set of relation names of source-side database schema $src\_db$
    $n \leftarrow$ a number of relation names of source schema defined in $P$
    $base \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        $s_i, \vec{X_i} \leftarrow$ $i$-th relation name and attributes of source schema defined in $P$
        $b\_s_i \leftarrow \{ (\text{"}b\_\text{"} \,\&\, s_i, \vec{X_i}) \}$        // base relation name
        $base \leftarrow base \cup b\_s_i$
    end for
    $t, \vec{Y}$        $\leftarrow$ a relation name and attributes of target schema defined in $P$
    $a\_lost\_t \leftarrow \{ (\text{"}a\_lost\_\text{"} \,\&\, t, \vec{Y}) \}$        // auxiliary relation name for lost tuples
    $a\_gain\_t \leftarrow \{ (\text{"}a\_gain\_\text{"} \,\&\, t, \vec{Y}) \}$        // auxiliary relation name for gained tuples
    $src\_db$      $\leftarrow base \cup a\_lost\_t \cup a\_gain\_t$
    return $src\_db$
```

schema, BXs between the database and the view instance of target schema, and BXs between the database and the view instances of source schema.

### 4.2.2 Deriving Source-Side Database Schema

We give Algorithm 4.1 to derive the source-side database schema. The input is a co-existence strategy. The output is a set of relation names of the base relations and the auxiliary relations.

The following example shows relation names that the algorithm outputs from a given co-existence strategy.

**Example 8.** Rewrite the co-existence strategy of the running example introduced in Section 3.1 for readability.

`% schemas:`

source: ver1# $s_1(X\text{:string}, Y\text{:int}, Z\text{:int}, W\text{:string})$.

target: ver2# $t(X\text{:string}, Y\text{:int}, Z\text{:int})$.

pk($s_1$, $['X']$).

pk($t$, $['X']$).

```
% schema evolution:
```

$t(X, Y, Z) :- s_1(X, Y, Z, W).$

```
% backward update sharing:
```

$+s_1(X, Y, Z, W) :- +t(X, Y, Z), \neg s_1(X, Y, Z, \_), Y < 100, W = \text{` '}.$

$-s_1(X, Y, Z, W) :- -t(X, Y, Z), s_1(X, Y, Z, W), Y < 100.$

```
% constraint:
```

$\bot() :- s_1(X, Y, Z, W), Y \leq 0.$

$\bot() :- t(X, Y, Z), Y \leq 0.$

Note that schema evolution and backward update sharing of this co-existence strategy are the same with rules (4.5) and (4.15) – (4.16) introduced in the previous subsection.

Since the co-existence strategy specifies one relation name of source schema, Algorithm 4.1 sets $n$ as 1. Then the algorithm sets $s_1$ as $s_1$, $\vec{X_1}$ as $\{X, Y, Z, W\}$, $t$ as $t$, $\vec{Y}$ as $\{X, Y, Z\}$ from relation names $s_1(X, Y, Z, W)$ of source schema and $t(X, Y, Z)$ of target schema.

Based on them, the algorithm outputs the following relation names: $b\_s_1(X, Y, Z, W)$ as the base relation name, $a\_lost\_t(X, Y, Z)$ as the auxiliary relation name for lost tuples, and $a\_gain\_t(X, Y, Z)$ as the auxiliary relation name for gained tuples. □

### 4.2.3 Deriving BX between Source-Side Database and View Instance of Target Schema

We give Algorithm 4.2 to derive BX between the source-side database and view instance of target schema. The input is a co-existence strategy satisfying the *consistency* of updates. The output is a bidirectional transformation $BX_{trg}$. For readability, transformations are described by relational algebra in the algorithm. The outline of the procedure to derive BX is shown in Subsection 4.2.1. In the algorithm, a derivation of constraints for a primary key follows the derivation method in [71].

The following example shows how the algorithm derives BX.

**Example 9.** Suppose the co-existence strategy in Example 8. Algorithm 4.2 outputs Datalog rules of BXs between the source-side database and the view instances of schemas from the co-existence strategy. Since schema evolution and backward update sharing of the strategy are the same with rules (4.5) and (4.15) – (4.16) treated in the previous subsection, the algorithm outputs following rules of $get_{trg}$: $r_{evo}$ as a rule (4.29), and $r_T$ as rules (4.30) – (4.31).

The algorithm outputs following rules of $put_{trg}$: $r_{base}$ as rules (4.17) – (4.18), $r_{aux}$ as rules (4.20) – (4.21) and (4.22) – (4.24), $r'_{base}$ as rules (4.13) – (4.14), $r'_{aux}$ as rules (4.25) – (4.28), $r_{\Delta T}$ as of rules (4.11) – (4.12), and $r_{T''}$ as a rule (4.19). In addition, the algorithm outputs Datalog rules of $c$ for a constraint that predicate symbol $t$ appears in its body, $c_{aux}$ for constraints applied to the auxiliary relations, and $c_{pk}$ for constraints of a primary key.

$c$:

$$\bot() :- t'(X, Y, Z), Y \leq 0.$$

$c_{aux}$:

$$\bot() :- a\_lost\_t(X, Y, Z), a\_gain\_t(X, Y, Z).$$
$$\bot() :- a\_gain\_t(X, Y, Z), \neg t\_evo(X, Y, Z).$$

where a predicate $t\_evo(X, Y, Z)$ corresponds to a relation $T^{evo}(X, Y, Z)$.

$c_{pk}$:

$$\bot() :- t'(X, Y, Z), t'(X, Y1, Z1), Y \neq Y1.$$
$$\bot() :- t'(X, Y, Z), t'(X, Y1, Z1), Z \neq Z1.$$

□

**Algorithm 4.2** Deriving BX between Source-Side Database and View Instance of Target Schema

```
Input: a co-existence strategy $P$ satisfying the consistency of updates
Output: bidirectional transformation $BX_{trg}$
    // Specification described in $P$
    $n$ ← a number of relation names of source schema
    for $i = 1$ to $n$ do
        $s_i, \vec{X_i}$ ← $i$-th relation name and attributes of source schema
                                                // corresponding to a relation $S_i(\vec{X_i})$
    end for
    $t, \vec{Y}$ ← a relation name and attributes of target schema
                                                // corresponding to a view instance $T(\vec{Y})$
    $C$ ← a set of constraints
    $f$ ← a set of Datalog rules to transform to $T(\vec{Y})$          // schema evolution
    for $i = 1$ to $n$ do
        $g^+_{s_i}$ ← a set of Datalog rules to transform $\Delta^+_{S_i}(\vec{X_i})$   // backward update sharing
        $g^-_{s_i}$ ← a set of Datalog rules to transform $\Delta^-_{S_i}(\vec{X_i})$   // backward update sharing
    end for

    // base relations $B_{S_i}(X_i)$ and auxiliary relations $A^{lost}_T(\vec{Y})$ and $A^{gain}_T(\vec{Y})$
    $src\_db$ ← a set of base and auxiliary relation names defined by Algorithm 4.1

    // $get_{trg}$
    $r_{evo}$ ← a set of Datalog rules of $f$ to transform to $T^{evo}(\vec{Y})$ by replacing predicate
              symbols $s_i$ for all $i$ ($i \in [1, n]$) to base relation names corresponding to
              $B_{S_i}(\vec{X_i})$ and $t$ to a predicate symbol corresponding to $T^{evo}(\vec{Y})$
    $r_T$ ← a set of Datalog rules to transform to $T(\vec{Y})$ as
              $T(\vec{Y}) = (T^{evo}(\vec{Y}) \cap \neg A^{gain}_T(\vec{Y})) \cup A^{lost}_T(\vec{Y})$
    $get_{trg}$ ← $r_{evo} \cup r_T$

    // $put_{trg}$
    $r_{base}$ ← a set of Datalog rules of $g^+_{s_i}$ and $g^-_{s_i}$ for all $i$ ($i \in [1, n]$) to transform to
              $\Delta^+_{B_{S_i}}(\vec{X_i})$ and $\Delta^+_{B_{S_i}}(\vec{X_i})$ by replacing predicate symbols $s_j$ for all $j$ ($j \in [1, n]$)
              to a base relation names corresponding to $B_{S_j}(\vec{X_j})$
    $r_{aux}$ ← a set of Datalog rules to transform to $\Delta^+_{A^{lost}_T}(\vec{Y})$, $\Delta^-_{A^{lost}_T}(\vec{Y})$, $\Delta^+_{A^{gain}_T}(\vec{Y})$, and
              $\Delta^-_{A^{gain}_T}(\vec{Y})$ as
              $\Delta^+_{A^{lost}_T}(\vec{Y}) = T'(\vec{Y}) \cap \neg T''(\vec{Y}) \cap \neg A^{lost}_T(\vec{Y})$,
              $\Delta^-_{A^{lost}_T}(\vec{Y}) = A^{lost}_T(\vec{Y}) \cap \neg T'(\vec{Y})$,
              $\Delta^+_{A^{gain}_T}(\vec{Y}) = \neg T'(\vec{Y}) \cap T''(\vec{Y}) \cap \neg A^{gain}_T(\vec{Y})$, and
              $\Delta^-_{A^{gain}_T}(\vec{Y}) = (A^{gain}_T(\vec{Y}) \cap T'(\vec{Y})) \cup (A^{gain}_T(\vec{Y}) \cap \neg T''(\vec{Y}))$
```

```
$r'_{base} \leftarrow \emptyset$                                  // Transformations to updated $B_{S_i}'$ for all i
for $i = 1$ to $n$ do
    $r'_i$ $\leftarrow$ a set of Datalog rules to transform updated $B_{S_i}'(\vec{X_i})$ as
        $B_{S_i}'(\vec{X_i}) = (B_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{B_{S_i}}(\vec{X_i})) \cup \Delta^+_{B_{S_i}}(\vec{X_i})$
    $r'_{base} \leftarrow r'_{base} \cup r'_i$
end for
$r'_{aux}$ $\leftarrow$ a set of Datalog rules to transform updated $A_T^{lost\prime}(\vec{Y})$ and $A_T^{gain\prime}(\vec{Y})$ as
        $A_T^{lost\prime}(\vec{Y}) = (A_T^{lost}(\vec{Y}) \cap \neg\Delta^-_{A_T^{lost}}(\vec{Y})) \cup \Delta^+_{A_T^{lost}}(\vec{Y}))$ and
        $A_T^{gain\prime}(\vec{Y}) = (A_T^{gain}(\vec{Y}) \cap \neg\Delta^-_{A_T^{gain}}(\vec{Y})) \cup \Delta^+_{A_T^{gain}}(\vec{Y}))$
$r_{\Delta T}$ $\leftarrow$ a set of Datalog rules to transform to $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$ as
        $\Delta^+_T(\vec{Y}) = T'(\vec{Y}) \cap \neg T(\vec{Y})$ and
        $\Delta^-_T(\vec{Y}) = \neg T'(\vec{Y}) \cap T(\vec{Y})$
$r_{T''}$ $\leftarrow$ a set of Datalog rules of $f$ to transform to $T''(\vec{Y})$ by replacing predicate
        symbols $s_i$ for all $i$ ($i \in [1, n]$) into base relation names corresponding to
        $B_{S_i}'(\vec{X_i})$ and a predicate symbol $t$ into a relation name corresponding to
        $T''(\vec{Y})$.
$c$ $\leftarrow$ a set of constrains in $C$ that a predicate with symbol $t$ appears in body and
        predicate symbols $t$ and $s_j$ for all $j$ ($j \in [1, n]$) are replaced to relation
        names corresponding to $T'(\vec{Y})$ and $B_{S_j}(\vec{X_j})$ respectively.
$c_{aux}$ $\leftarrow$ a set of constraints of
        $A_T^{lost}(\vec{Y}) \cap A_T^{gain}(\vec{Y}) = \emptyset$ and
        $A_T^{gain}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) = \emptyset$
$c_{pk}$ $\leftarrow$ a set of constrains for primary key of relation name $t$ if it is specified
$put_{trg} \leftarrow r_{base} \cup r_{aux} \cup r'_{base} \cup r'_{aux} \cup r_{\Delta T} \cup r_{evo} \cup r_{T''} \cup c \cup c_{aux} \cup c_{pk}$
$BX_{trg} \leftarrow \{(get_{trg}, put_{trg})\}$
return $BX_{trg}$
```

### Properties

Based on transformations of $get_{trg}$ and $put_{trg}$ derived by Algorithm 4.2, the following lemmas and a proposition are satisfied. Lemmas state that particular relationships of the auxiliary relations and the base relations are kept. Note that they are given as constraints in the algorithm.

**Lemma 4.1** (Disjointness of Auxiliary Relations of Source-Side Database)**.** *The auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ are disjoint.*

$$A_T^{lost}(\vec{Y}) \cap A_T^{gain}(\vec{Y}) = \emptyset$$

□

The formal proof is available in Appendix A.1.

**Lemma 4.2** (Inclusion of Auxiliary Relation of Source-Side Database)**.** *The auxiliary relation $A_T^{gain}(\vec{Y})$ for gained tuples is included in the relation $T^{evo}(\vec{Y})$ defined in Algorithm 4.2.*

$$A_T^{gain}(\vec{Y}) \subseteq T^{evo}(\vec{Y})$$

□

The proof is available in Appendix A.2.

In order to realize a co-existence of schema by the view-based approach, the updated view instance must be recomputed from the updated database without loss or gain, and the non-updated view instance must not change the database. The following proposition states a bidirectional transformation $BX_{trg}$ derived by Algorithm 4.2 is well-behaved and satisfies these features.

**Proposition 4.3** (Well-Behaveness of BX between Source-Side Database and View Instance of Target Schema)**.** *Given a co-existence strategy between relations of source schema and a relation of target schema and turning each relation to view instance, bidirectional transformation $BX_{trg}$ derived by the Algorithm 4.2 is well-behaved by satisfying* GetPut *and* PutGet *laws.* □

The proof is available in Appendix A.3.

### 4.2.4 Deriving BX between Source-Side Database and View Instance of Source Schema

We give Algorithm 4.3 to derive BXs between the source-side database and the view instances of source schema. The input is a co-existence strategy satisfying the *consistency* of updates. The output is bidirectional transformations $BX_{src.i}$ for all $i$ ($i \in [1, n]$) expressed by Datalog rules. For readability, transformations are described by relational algebra in the following and the algorithm. A derivation of constraints for a primary key follows the derivation method in [71]

The algorithm derives identity mapping between the view instance of source schema $S_i(\vec{X_i})$ and the base relation $B_{S_i}(\vec{X_i})$ of the database. The algorithm defines $get_{src.i}$ as following:

$$S_i(\vec{X_i}) = B_{S_i}(\vec{X_i}) \tag{4.32}$$

It is obvious that this transformation is identity mapping. The algorithm defines $r'_{base}$ of $put_{src.i}$ as a set of transformations from the updated view instance $S_i{}'(\vec{X_i})$ to the updated base relation $B_{S_i}{}'(\vec{X_i})$ as follows:

$$\Delta^+_{B_{S_i}}(\vec{X_i}) = S_i{}'(\vec{X_i}) \cap \neg B_{S_i}(\vec{X_i}) \tag{4.33}$$
$$\Delta^-_{B_{S_i}}(\vec{X_i}) = \neg S_i{}'(\vec{X_i}) \cap B_{S_i}(\vec{X_i}) \tag{4.34}$$
$$B_{S_i}{}'(\vec{X_i}) = (B_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{B_{S_i}}(\vec{X_i})) \cup \Delta^+_{B_{S_i}}(\vec{X_i}) \tag{4.35}$$

Rules (4.33) and (4.34) for transformations to inserted and deleted tuples against the base relation $B_{S_i}(\vec{X_i})$ respectively. Rules (4.35) is an application of delta relation to transform into the updated base relation $B_{S_i}{}'(\vec{X_i})$. The following shows these transformations are identify mapping between $S_i{}'(X_i)$ and $B_{S_i}{}'(X_i)$. Since all relations appearing in rules have the same attributes $\vec{X_i}$, it is omitted.

$$\begin{aligned}
B_{S_i}{}' &= \{\text{Rule (4.35)}\}\\
&\quad (B_{S_i} \cap \neg\Delta^-_{B_{S_i}}) \cup \Delta^+_{B_{S_i}}\\
&= \{\text{Substitute rules (4.33) and (4.34)}\}\\
&\quad (B_{S_i} \cap \neg(\neg S_i{}' \cap B_{S_i})) \cup (S_i{}' \cap \neg B_{S_i})
\end{aligned}$$

$$\begin{aligned}
&= \{\text{Deformation of a formula by set operation}\} \\
&\quad (B_{S_i} \cap \neg B_{S_i}) \cup (B_{S_i} \cap S_i{}') \cup (S_i{}' \cap \neg B_{S_i}) \cup (B_{S_i} \cup S_i{}') \cup (\neg B_{S_i} \cap S_i{}') \\
&= \{\text{Deformation of a formula by set operation and } A \cap \neg A = \emptyset\} \\
&\quad S_i{}'
\end{aligned}$$

The algorithm defines $r'_{aux}$ as transformations to the update auxiliary relation $A_T^{lost\prime}$ for lost tuples because lost tuples can be disappeared when updates against the view instance $S_i$ of source schema are shared with the view instance $T$ of target schema by a transformation of schema evolution. Transformations of $r'_{aux}$ are defined as followings:

$$\Delta^-_{A_T^{lost}}(\vec{Y}) = (\Delta^+_T(Y) \cap A_T^{lost}(Y)) \cup (\Delta^-_T(Y) \cap A_T^{lost}(Y)) \tag{4.36}$$

$$A_T^{lost\prime}(\vec{Y}) = A_T^{lost}(\vec{Y}) \cap \neg\Delta^-_{A_T^{lost}}(\vec{Y}) \tag{4.37}$$

A relation $\Delta^+_T(\vec{Y})$ ($\Delta^-_T(\vec{Y})$) is a set of tuples that newly appears (disappears) in the updated view instance of target schema as a result of schema evolution from the updated base relations but does not exist (exists) in the original view instance of target schema as a result of schema evolution from the original base relations. If tuples of $\Delta^+_T(\vec{Y})$ exist in $A_T^{lost}$, they are not lost tuples anymore. If tuples of $\Delta^-_T(\vec{Y})$ exist in $A_T^{lost}$, such tuples must not appear in the view instance $T$ of target schema as lost tuples. Rule (4.36) transforms such tuples into tuples tuples to be deleted from $A_T^{lost}$. Rule (4.37) expresses an application of delta relation to transform into the updated auxiliary relation $A_T^{lost\prime}(\vec{Y})$. Relations $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$ are defined as transformations of $r_T$, $r_{T'}$ and $r_{\Delta T}$ in the algorithm.

**Algorithm 4.3** Deriving BXs between Source-Side Database and View Instances of Source Schema

```
Input: a co-existence strategy $P$ satisfying the consistency of updates
Output: bidirectional transformations of source schema $BX_{src}$
    // Specification described in $P$
    $n \leftarrow$ a number of relation names of source schema
    for $i = 1$ to $n$ do
        $s_i, \vec{X_i} \leftarrow$ a relation name and attributes of source schema
                                        //corresponding to a view instance $S_i(\vec{X_i})$
    end for
    $t, \vec{Y} \leftarrow$ a relation name and attributes of target schema
                                        //corresponding to a relation $T(\vec{Y})$
    $C \leftarrow$ a set of constraints in $P$
    $f \leftarrow$ a set of Datalog rules to transform $T(\vec{Y})$          //schema evolution

    // base relations $B_{S_i}(X_i)$ and auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$
    $src\_db \leftarrow$ a set of base and auxiliary relation names defined by Algorithm 4.1

    $BX_{src} \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        // $get_{src.i}$
        $get_{src.i} \leftarrow$ a set of Datalog rule to transform to $S_i(\vec{X_i})$ as $S_i(\vec{X_i}) = B_{S_i}(\vec{X_i})$

        // $put_{src.i}$
        $r'_{base} \leftarrow$ a set of Datalog rules to transform to $B_{S_i}{}'(\vec{X_i})$ as
            $\Delta^+_{B_{S_i}}(\vec{X_i}) = S_i{}'(\vec{X_i}) \cap \neg B_{s_i}(\vec{X_i})$,
            $\Delta^-_{B_{S_i}}(\vec{X_i}) = \neg S_i{}'(\vec{X_i}) \cap B_{s_i}(\vec{X_i})$, and
            $B_{S_i}{}'(\vec{X_i}) = (B_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{B_{S_i}}(\vec{X_i})) \cup \Delta^+_{B_{S_i}}(\vec{X_i})$
        $r_{evo} \leftarrow$ a set of Datalog rules of $f$ to transform to $T^{evo}(\vec{Y})$ by replacing
            predicate symbols $s_j$ for all $j$ ($j \in [1, n]$) to base relation names
            corresponding to $B_{S_j}(\vec{X_j})$ and a predicate $t$ to a predicate symbol
            corresponding to $T^{evo}(\vec{Y})$
        $r'_{evo} \leftarrow$ a set of Datalog rules of $f$ to transform to $T^{evo\prime}(\vec{Y})$ by replacing
            predicate symbols $s_i$ and $t$ to base relation names corresponding to
            $B_{S_i}{}'(\vec{X_i})$ and $T^{evo\prime}(\vec{Y})$, and $s_j$ ($j \neq i$) to a relation names corresponding
            to $B_{S_j}(\vec{X_j})$ if it exist.
        $r_{\Delta evo} \leftarrow$ a set of Datalog rules to transform to $\Delta^+_{T^{evo}}(\vec{Y})$ and $\Delta^-_{T^{evo}}(\vec{Y})$ as
            $\Delta^+_{T^{evo}}(\vec{Y}) = T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y})$ and
            $\Delta^-_{T^{evo}}(\vec{Y}) = \neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y})$
        $r'_{aux} \leftarrow$ a set of Datalog rules to transform to $A_T^{lost\prime}(\vec{Y})$ as
            $\Delta^-_{A_T^{lost}}(\vec{Y}) = (\Delta^+_{T^{evo}}(\vec{Y}) \cap A_T^{lost}(\vec{Y})) \cup (\Delta^-_{T^{evo}}(\vec{Y}) \cap A_T^{lost}(\vec{Y}))$ and
            $A_T^{lost\prime}(\vec{Y}) = A_T^{lost}(\vec{Y}) \cap \neg\Delta^-_{A_T^{lost}}(\vec{Y})$
        $c$ $\leftarrow$ a set of constrains in $C$ that a predicate with symbol $t$ does not appears
            in body and predicate symbol $s_i$ and $s_j$ $(j \neq i)$ are replaced to a
            corresponding relation names of $B_{S_i}'(\vec{X_i})$ and $B_{S_j}(\vec{X_j})$ respectively.
        $c_{pk}$ $\leftarrow$ a set of constrains for primary key of relation name $s_i$ if it is specified
        $put_{src.i} \leftarrow r'_{base} \cup r_{evo} \cup r'_{evo} \cup r_{\Delta evo} \cup r'_{aux} \cup c \cup c_{pk}$
        $BX_{src.i} \leftarrow \{(get_{src.i}, put_{src.i})\}$
        $BX_{src} \leftarrow BX_{src} \cup BX_{src.i}$
    end for
    return $BX_{src}$
```

The following examples show how the algorithm derives BXs.

**Example 10.** Suppose the co-existence strategy in Example 8. Algorithm 4.3 derives a transformation of $get_{src.1}$ as follows:

$$s_1(X, Y, Z, W) \mathrel{:-} b\_s_1(X, Y, Z, W).$$

where the predicate with symbol $b\_s_1$ corresponds to the base relation $B_{S_1}$.

The algorithm derives the following transformations of $put_{src.1}$.

$r'_{base}$:

$$\begin{aligned}
+b\_s_1(X, Y, Z, W) &\mathrel{:-} s_1{}'(X, Y, Z, W), \neg b\_s_1(X, Y, Z, W).\\
-b\_s_1(X, Y, Z, W) &\mathrel{:-} \neg s_1{}'(X, Y, Z, W), b\_s_1(X, Y, Z, W).\\
b\_s_1{}'(X, Y, Z, W) &\mathrel{:-} b\_s_1(X, Y, Z, W), \neg - b\_s_1(X, Y, Z, W).\\
b\_s_1{}'(X, Y, Z, W) &\mathrel{:-} +b\_s_1(X, Y, Z, W).
\end{aligned}$$

where predicates with symbols $s_1{}'$ and $b\_s_1{}'$ correspond to the updated view instance $S_1{}'$ and $B_{S_1}{}'$ respectively.

$r_{evo}$:

$$t\_evo(X, Y, Z) \mathrel{:-} b\_s_1(X, Y, Z, W).$$

A rule of $r_{evo}$ is derived by replacing the predicate symbols $t$ and $s_1$ in the rule of schema evolution into the predicate symbols $t\_evo$ and $b\_s_1$ respectively.

$r'_{evo}$:

$$t\_evo'(X, Y, Z) \mathrel{:-} b\_s_1{}'(X, Y, Z, W).$$

A rule of $r_{evo'}$ is derived by replacing predicate symbol $t$ and $s_1$ appearing in the rule of schema evolution into predicate symbols $t\_evo'$ and $b\_s_1{}'$ respectively.

$r_{\Delta evo}$:

$$+t\_evo(X,Y,Z) :- t\_evo'(X,Y,Z), \neg t\_evo(X,Y,Z).$$
$$-t\_evo(X,Y,Z) :- \neg t\_evo'(X,Y,Z), t\_evo(X,Y,Z).$$

where predicates with symbols $+t\_evo$ and $-t\_evo$ correspond to relations $\Delta^+_{T^{evo}}$ and $\Delta^-_{T^{evo}}$ respectively.

$r'_{aux}$:

$$-a\_lost\_t(X,Y,Z) :- +t\_evo(X,Y,Z), a\_lost\_t(X,Y,Z).$$
$$-a\_lost\_t(X,Y,Z) :- -t\_evo(X,Y,Z), a\_lost\_t(X,Y,Z).$$
$$a\_lost\_t'(X,Y,Z) :- a\_lost\_t(X,Y,Z) \neg - a\_lost\_t(X,Y,Z).$$

where predicates with symbols $a\_lost\_t$, $a\_lost\_t'$ $+a\_lost\_t$, $-a\_lost\_t$, correspond to relations $A^{lost}_T$, $A^{lost'}_T$, $\Delta^+_{A^{lost}_T}$, and $\Delta^-_{A^{lost}_T}$ respectively.

In addition, the algorithm outputs Datalog rules of $c$ for a constraint that predicate symbol $t$ does not appear in its body and $c_{pk}$ for constraints of a primary key.

$c$:

$$\perp() :- s_1'(X,Y,Z,W), Y \leq 0.$$

$c_{pk}$:

$$\perp() :- s_1'(X,Y,Z,W), s_1'(X,Y1,Z1,W1), Y \neq Y1.$$
$$\perp() :- s_1'(X,Y,Z,W), s_1'(X,Y1,Z1,W1), Z \neq Z1.$$
$$\perp() :- s_1'(X,Y,Z,W), s_1'(X,Y1,Z1,W1), W \neq W1.$$

□

In the algorithm, derivation of $r_{T'}$ is rather complicated. The following example shows how it is derived when two relations of source schema are specified.

**Example 11.** Suppose a co-existence strategy with schema evolution by join in Figure

3.5. Its schema evolution is as follows:

$$t(X, Y, Z) :- s_1(X, Y), s_2(X, Z).$$

This strategy defines two relation names $s_1$ and $s_2$ of source schema. Thus the algorithm sets $n = 2$. It derives the following Datalog rule for $r_{evo'}$ of $put_{src.1}$.

$$t\_evo'(X, Y, Z) :- b\_s_1{}'(X, Y), b\_s_2(X, Z).$$

Predicate symbols $t$, $s_1$ and $s_2$ appearing in the Datalog rule of schema evolution are replaced into $t\_evo'$, $b\_s_1{}'$ and $b\_s_2$ respectively.

The algorithm derives the following Datalog rule for $r_{evo'}$ of $put_{src.2}$.

$$t\_evo'(X, Y, Z) :- b\_s_1(X, Y), b\_s_2{}'(X, Z).$$

Predicate symbols $t$, $s_1$ and $s_2$ appearing in schema evolution are replaced into $t\_evo'$, $b\_s_1$ and $b\_s_2{}'$ respectively. □

**Properties**

The following proposition states that a bidirectional transformations $BX_{src.i}$ for each $i$ ($i \in [1, n]$) derived by Algorithm 4.3 is well-behaved.

**Proposition 4.4** (Well-Behaveness of BXs between Source-Side Database and View Instances of Source Schema)**.** *Given a co-existence strategy between relations $S_i$ ($i \in [1, n]$) of source schema and a relation of target schema and turning each relation to view instance, bidirectional transformation $BX_{src.i}$ for each $i$ derived by the Algorithm 4.3 are well-behaved by satisfying* GetPut *and* PutGet *laws.* □

The proof is available in Appendix A.4.

### 4.2.5 Correctness of Derivation Algorithms

We first show schema evolution and backward update sharing of a co-existence strategy are realized by the derived BXs, then show the soundness of the proposed algorithms to derive BXs.

### Realization of Schema Evolution

Schema evolution of a co-existence strategy is realized if a result of transformations by schema evolution and derived BXs are the same.

**Theorem 4.5** (Realization of Schema Evolution by BXs between Source-Side Database and View Instances of Schemas)**.** *Given a co-existence strategy, the derived bidirectional transformations by Algorithm 4.2 and 4.3 realize its schema evolution.* □

Suppose a co-existence strategy between relations $S_i$ ($i \in [1, n]$) of source schema and a relation $T$ of target schema, and $BX_{src.i}$ for all $i$ and $BX_{trg}$ as derived bidirectional transformations from the co-existence strategy by Algorithm 4.2 and 4.3. Recall source schema instance **S** is union of relations $S_i$ for all $i$. Schema evolution of the co-existence strategy transforms **S** to $T$. On the other hand, for each $i$, $put_{src.i}$ of $BX_{src.i}$ transforms the view instance $S_i$ to the source-side database $\mathbf{D}_s$, and $get_{trg}$ of $BX_{trg}$ transforms $\mathbf{D}_s$ to the view instance $T$. If a result of these transformations from union of $S_i$ to $T$ via $\mathbf{D}_s$ is equivalent to a result of the transformation by schema evolution, schema evolution is realized by the derived BXs. Further details and the formal proof are available in Appendix A.5.

### Realization of Backward Update Sharing

Backward update sharing of a co-existence strategy is realized if a result of transformations by backward update sharing and derived BXs are the same.

**Theorem 4.6** (Realization of Backward Update Sharing by BXs between Source-Side Database and View Instances of Schemas)**.** *Given a co-existence strategy, the derived bidirectional transformations by Algorithm 4.2 and 4.3 realize its backward update sharing.* □

Suppose the co-existence strategy and derived bidirectional transformations mentioned above. Backward update sharing of the co-existence strategy transforms sets of inserted and deleted tuples against the relation $T$ of target schema to sets of inserted and deleted tuples to each relation $S_i$ ($i \in [1, n]$) of source schema. They update the relation $S_i$ to $S_i{}'$. On the other hand, $put_{trg}$ of $BX_{trg}$ transforms sets of inserted and deleted tuples against a view instance $T$ to sets of inserted and deleted tuples against the source-side database $\mathbf{D}_s$ and $get_{src.i}$ of $BX_{src.i}$ for each $i$ transforms the updated source-side database $\mathbf{D}_s{}'$ to the updated view instance $S_i{}'$. If a result of these transformations is equivalent to a result of the transformation by backward update sharing, backward update sharing is realized by the derived BXs. Further details and the formal proof are available in Appendix A.6.

#### Soundness

Given a co-existence strategy, the derived BXs by Algorithm 4.3 and 4.2 are well-behaved and realize its schema evolution and backward update sharing. Thus the algorithms are sound.

## 4.3 Evaluation

This section shows experimental results and an evaluation of the proposed method.

### 4.3.1 Implementation

We have implemented the verification of the *consistency* of updates and Algorithm 4.1, 4.2, and 4.3 into a prototype. The prototype is coded in OCaml[2]. The input to the prototype is co-existence strategies written by the proposed DSL. The output from the prototype is Datalog programs of BXs between the source-side or target-side database and view instances of schemas.

The verification of the *consistency* of updates is implemented as satisfiability check of a Datalog program explained in Chapter 3. The prototype transforms a Datalog program of a co-existence strategy into a program expressed by first-order logic and then verifies its satisfiability by using SMT solver, Z3. If its satisfiability is satisfied, the prototype outputs Datalog programs of BXs by Algorithm 4.1, 4.2, and 4.3. The generated Datalog programs of BXs are checked whether the well-behaveness is satisfied just in case and transformed into SQL programs to run on a commercial RDBMS, PostgreSQL by BIRDS [71]. BIRDS is a bidirectionalization engine between a view instance and a database and generates a SQL program file specifying a view

[2]The full source code is available at https://github.com/JumpeiTanaka/COXS/

derived from *get* of a given BX described by Datalog rules and triggers from *put* of the BX.

### 4.3.2 Experimental Result

We conduct an experiment to evaluate our approach. The experiment investigates a practical relevance of our proposed method in describing co-existence strategies, deriving BXs to realize strategies, and performance of writing and reading on view instances of both source schema and target schema while sharing updates with each. In order to evaluate them, we define a benchmark as co-existence strategies to be evaluated from the following points:

- Co-existence strategies predefined for SMOs of existing work [34, 35].
- Co-existence strategies consisting of schema evolution expressed by basic operations of relational algebra, selection, projection, union, Cartesian product, and set difference, and backward update sharing paired with schema evolution.

For the first point, SMOs' predefined strategies are obtained from Figure 2 and appendix of [34]. For the second point, since practitioners describe a script of data migration for schema evolution by SQL program [2], the expressive power of the proposed DSL must be relational complete to describe schema evolution. Thus, we evaluate co-existence strategies consisting of schema evolution expressed by the five base operations mentioned above.

The experiment is run on a Core i5 machine with 2 GHz and 16 GB memory and PostgreSQL 10.16.

Table 4.1 shows experimental results of benchmark co-existence strategies described by the proposed DSL and derived BXs between the source-side database and the view instances of source schema and target schema. The first column of the table shows SMO's operator[3]. The third and fourth columns show schema evolution and backward update sharing of co-existence strategies. We experimented with co-existence strategies consisting of schema evolutions equivalent to transformations by basic relational algebra operations and several types of backward update sharing paired with each

[3]Since SMO `DECOMPOSE TABLE` is inverse SMO of `JOIN TABLE`, we evaluated `JOIN TABLE` only. `DECOMPOSE TABLE` is specified by swapping relations of source schema and target schema.

schema evolution. The fifth column is checked for SMO's predefined co-existence strategies. Since SMOs do not directly support schema evolution by Cartesian product and set difference, co-existence strategies consisting of such schema evolution are added.

For example, the table shows that SMO `DROP COLUMN` expresses modification of a relation $S_1$ of source schema to a relation $T$ of target schema by dropping columns (attributes). Its co-existence strategy consists of schema evolution that expresses dropping columns by projection and backward update sharing that transforms any updates against $T$ to $S_1$. This SMO predefines one auxiliary relation name to realize its co-existence strategy. On the other hand, the proposed DSL describes its co-existence strategy as an 8 LOC Datalog program. Our method derives BXs as 41 LOC of Datalog programs and 701 LOC of SQL programs by defining two auxiliary relation names of the source-side database schema.

Table 4.1: Results of derived BXs between the source-side database and view instances of source schema and target schema.

| SMOs | | co-existence strategy | | | the proposed method | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Operator | # of aux. | schema evolution | backward update sharing | SMO | DSL [LOC] | # of aux. | BXs [LOC] | SQL [LOC] | No. |
| DROP COLUMN ($S_1 \rightarrow T$) | 1 | projection | share all with $S_1$ | ✓ | 8 | 2 | 41 | 701 | 1 |
| | | | share with $S_1$ if a condition is satisfied | | 8 | 2 | 41 | 701 | 2 |
| | | | share only deletion with $S_1$ if a condition is satisfied | | 6 | 2 | 38 | 546 | 3 |
| | | | not share with $S_1$ | | 3 | 2 | 33 | 419 | 4 |
| ADD COLUMN ($S_1 \rightarrow T$) | 1 | outer join (pk) | share all with $S_1$ | ✓ | 17 | 2 | 76 | 4209 | 5 |
| | | | not share with $S_1$ | | 9 | 2 | 66 | 797 | 6 |
| JOIN TABLE ($S_1, S_2 \rightarrow T$) | 0 | outer join (pk) | share all with $S_1$ and $S_2$ | ✓ | 15 | 2 | 80 | 3664 | 7 |
| | | | not share with $S_1$ and $S_2$ | | 9 | 2 | 72 | 800 | 8 |
| | 0 | outer join (fk) | share all with $S_1$ and $S_2$ | ✓ | 12 | 2 | 78 | 2007 | 9 |
| | | | not share with $S_1$ and $S_2$ | | 5 | 2 | 70 | 845 | 10 |
| | 0 | inner join (pk) | share all with $S_1$ and $S_2$ | ✓ | 11 | 2 | 62 | 1192 | 11 |
| | | | not share with $S_1$ and $S_2$ | | 7 | 2 | 56 | 698 | 12 |
| | 2 | inner join (cond.) | share all with $S_1$ and $S_2$ | ✓ | 16 | 2 | 66 | 4158 | 13 |
| | | | not share with $S_1$ and $S_2$ | | 4 | 2 | 50 | 630 | 14 |
| SPLIT TABLE ($S_1 \rightarrow T$) | 1 | selection | share with $S_1$ without condition | ✓ | 7 | 2 | 39 | 636 | 15 |
| | | | share with $S_1$ if selection condition is satisfied | | 7 | 2 | 39 | 636 | 16 |
| | | | not share with $S_1$ | | 5 | 2 | 36 | 456 | 17 |
| MERGE TABLE ($S_1, S_2 \rightarrow T$) | 1 | union | share with $S_1$ and $S_2$ if conditions are satisfied | ✓ | 13 | 2 | 68 | 1481 | 18 |
| | | | share insertion with $S_1$ and deletion with $S_2$ | | 8 | 2 | 65 | 1197 | 19 |
| | | | not share with $S_1$ and $S_2$ | | 8 | 2 | 124 | 731 | 20 |
| ($S_1, S_2 \rightarrow T$) | - | Cartesian product | share with $S_1$ and $S_2$ | | 18 | 2 | 66 | 3973 | 21 |
| ($S_1, S_2 \rightarrow T$) | - | set difference | share with $S_1$ and not share with $S_2$ | | 6 | 2 | 53 | 860 | 22 |

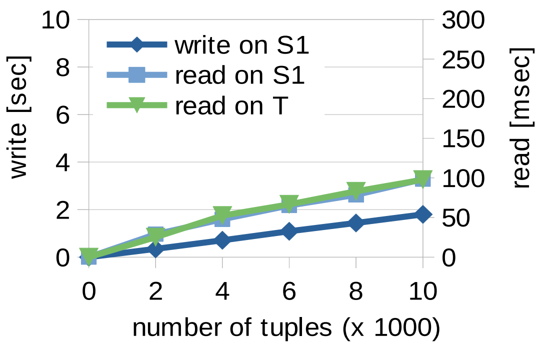
Figure 4.3 shows writing and reading while updates are shared by schema evolution realized on the source-side database. Each graph shows relationships between the number of executed tuples and execution time of writing (insertion) to $S_1$ of source schema, reading its result as the view instances $S_1$ of source schema, and reading of the view instance $T$ of target schema as a result of update sharing by schema evolution of each co-existence strategy. Figure 4.3 (a) is a result of the co-existence strategy No.1 in Table 4.1 for schema evolution by projection. Figure 4.3 (b), (c), (d), (e) are results of the co-existence strategies No.13 for schema evolution by inner join by a given condition, No.15 for schema evolution by selection, No.18 for schema evolution by union, and No.22 for schema evolution by set difference respectively. Since join is a combination of Cartesian product and selection, we experimented inner join of Figure 4.3 (b) instead of Cartesian product. In writing on $S_1$ of (b) and (e), a view instance $S_2$ of source schema has 10,000 tuples in advance.

Results show that reading time is almost the same for view instances of source schema and target schema except for (c) of schema evolution by selection. Since schema evolution by selection discards tuples, tuples of view instance $T$ of target schema are less than tuples of view instance $S_1$ of source schema. Then the reading time of view instance $T$ is faster than the reading time of $S_1$. The execution time of writing varies depending on strategies of backward update sharing. While schema evolution by projection, join, and set difference ((a), (b), (d) respectively) show linear relationships between the number of executed tuples and executed time, schema evolution by selection and union ((c) and (e) respectively) show non-linear increases of execution time against the increase of executed tuples.

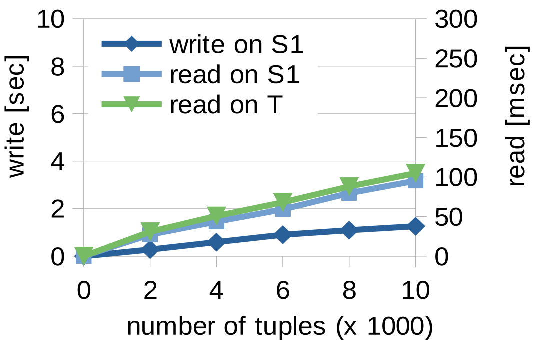
Figure 4.4 shows performances of writing and reading while updates are shared by backward update sharing realized on the source-side database. Each graph shows relationships between the number of executed tuples and execution time of writing (insertion) to $T$ of target schema, reading its result as the view instances $T$ of target schema, and reading of the view instance $S_1$ of source schema as a result of update sharing by backward update sharing of each co-existence strategy. In the same manner with Figure 4.3, Figure 4.4 (a), (b), (c), (d), (e) are results of the co-existence strategy No.1 for schema evolution by projection, the co-existence strategies No.13 for schema evolution by inner join by a given condition, No.15 for schema evolution by selection, No.18 for schema evolution by union, and No.22 for schema evolution by set difference respectively. In writing on $T$ of (b) and (e), the view instance $S_2$ of source schema already has 10,000 tuples in advance. Note that the scale of writing time in Figure 4.4 (b) is different from others.

Results show that reading time is almost the same for view instances of source schema and target schema except for (c) of schema evolution by selection. The execution time of writing varies depending on strategies. While schema evolution by join and set difference ((b), (d) respectively) show linear relationships between the number of executed tuples and executed time, schema evolution by projection, selection and, union ((a), (c), and (e) respectively) show non-linear increases of execution time against the increase of the executed tuples. Figure 4.4 (b) shows the execution time of writing on backward update sharing against schema evolution by join takes more time than others because it updates the base relation corresponding to the view instance $S_1$ of source schema by checking join condition.

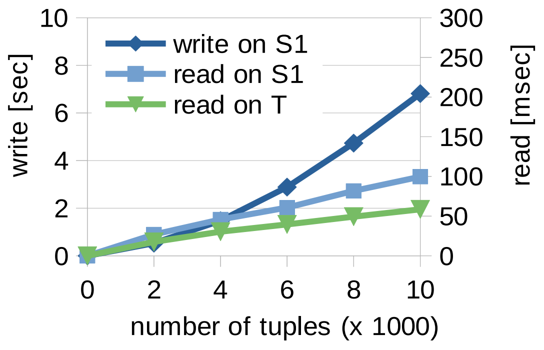
Furthermore, we investigate the variation of performance depending on the auxiliary relation. Figure 4.4 (f) shows performances depending on the ratio of recorded tuples in the auxiliary relation for lost tuples and the base relation. This experiment is performed based on the co-existence strategy No.15 in Table 4.1 for schema evolution by selection. Its selection condition of backward update sharing is varied so that inserted tuples are stored in 0%, 50%, and 100% in the auxiliary relation (100%, 50%, and 0% in the base relation). The result shows a case of 50% is the slowest while cases of 0% and 100% are almost the same performance.



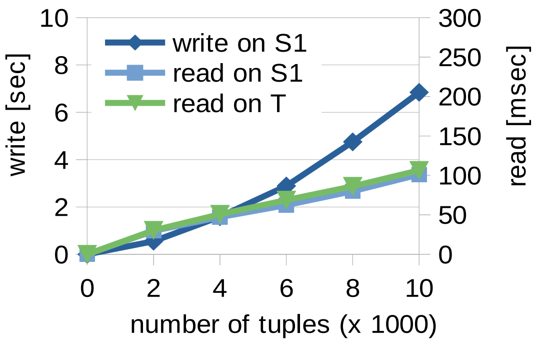
(a) projection



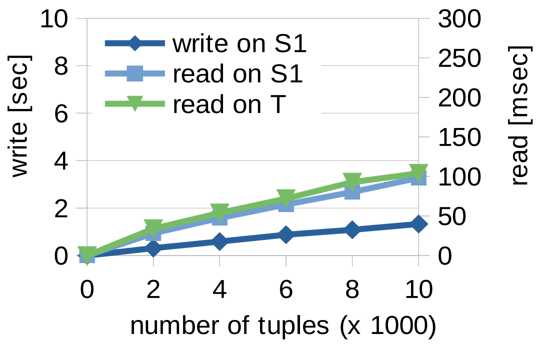
(b) join



(c) selection

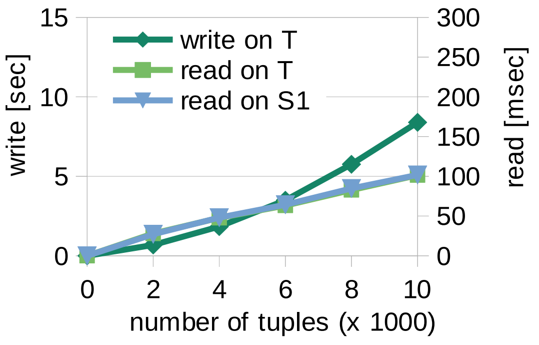


(d) union

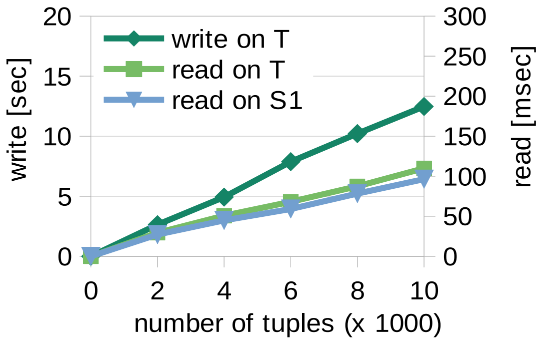


(e) set difference

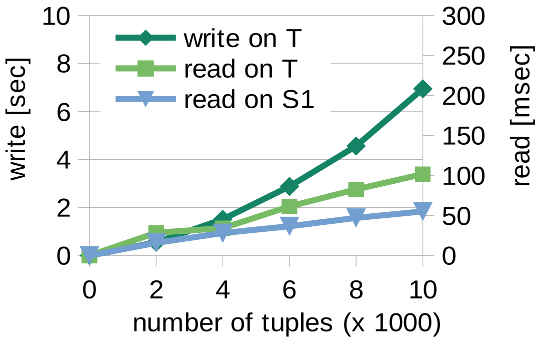
Figure 4.3: Performance of writing and reading with update sharing by schema evolution realized on the source-side database.



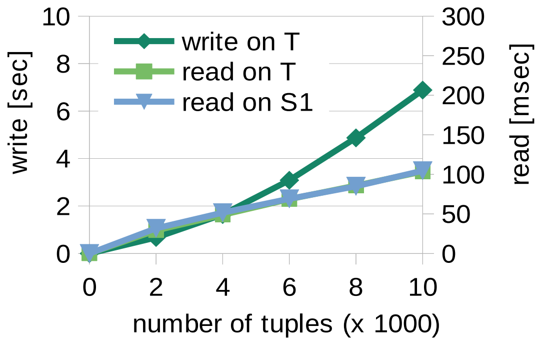
(a) projection



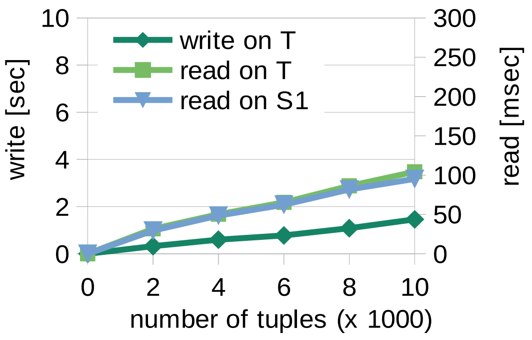
(b) join



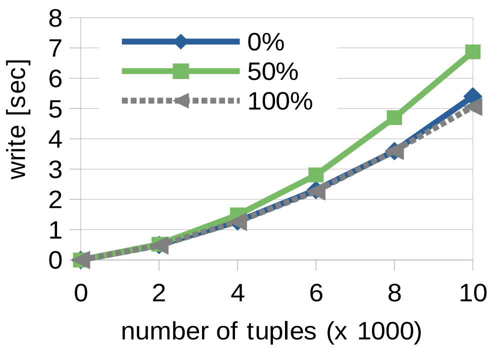
(c) selection



(d) union



(e) set difference



(f) auxiliary relation for lost tuples

Figure 4.4: Performance of writing and reading with update sharing by backward update sharing realized on the source-side database

### 4.3.3 Discussion

Table 4.1 shows that the benchmark co-existence strategies mentioned above are describable by the proposed DSL, and BXs to realize them are automatically derived. All strategies are described by less than 20 LOC. The proposed method makes co-existence strategies programmable with a reasonable amount of description. The table also shows that the existing work predefines auxiliary relation names for SMO's strategies on a one-by-one basis. On the other hand, our proposed method realizes any co-existence strategies based on two auxiliary relation names systematically derived by Algorithm 4.1 from a given co-existence strategy.

Figure 4.3 and Figure 4.4 show that the sampled co-existence strategies of Table 4.1 are realized by sharing updates between view instances of schemas based on the derived BXs. The performance of reading is almost the same regardless of strategies. However, the performance of writing is much slower than reading. It shows non-linear increases of execution time in some co-existence strategies against the increase of executed tuples. Figure 4.4 (f) shows that the performance of writing is the worst when the view instance is evenly transformed to the base relation and the auxiliary relation. These results suggest that improving performance for the writing on view instances makes the proposed method more practical.

## 4.4 Related Work

This section shows related work about realization of a co-existence strategy.

#### Practical Operation of Schema Evolution

Due to difficulty to realize both schema evolution and backward update sharing of co-existence strategies, schema evolution has been tread mainly in practical operation. For practitioners, Ambler et al. [2] summarize the refactoring databases as design patterns of more than 100 cases to define a new schema based on an original schema by SQL program, for example, dropping a column and merging tables. Since it is challenging to correctly manage thousands of SQL program files for all changes of schema evolution, tools, Flyway [28], Liquibase [49], MyBatis migration [53], DBDeploy [23], and DB maestro [23], support managing a bunch of SQL program files for each change and correctly executing them.

These strong tools make our work practical because our implemented prototype generates BXs to realize a co-existence strategy as SQL program files.

### NoSQL databases

NoSQL database is another angle of schema evolution [21, 64]. When an application is not sensitive for consistency or availability of data, NoSQL database provides a more flexible schema or schemaless model to reduce the cost of modifying schemas. However, we focus on relational database because NoSQL relaxes ACID properties and is not a full replacement for relational database.

### Data Integration

While each database is independently developed for each purpose, the same data would exist in some databases with different formats or attribute names and additional information. Data integration aims to provide the unified view sharing data among databases.

Designing a shared global schema with mapping to each database is one data integration technique. Such a top-down approach works well if the problem domain is well understood. As a bottom-up approach, Ng et al. [56] propose PeerDB, Halevy and Tatarinov [31, 68] propose Pizza as a peer-to-peer-based data sharing system. Halevy [32] also propose PDMS with decentralized and extensible mediators for data sharing. Ives et al. [40] propose ORCHESTRA as a successor project of Piazza. ORCHESTRA is a collaborative data sharing systems (CDSS) in which a database shares data in the first step, copies updated data by other databases, and decides whether accept updates by others or not. Karvounarakis et al. [41] enhance CDSS to utilize provenance semirings [32] to decide whether accept an update or not by knowing which database originally updates data.

Even though there is a gap between the data integration based on multiple databases and the view-based co-existence of schema on one database, data integration gives ideas about how to control data sharing by specifying which database originally updates data. In this thesis, we control update sharing by auxiliary relations to specify which view instance of a schema originally updates tuples.

### Multitenancy Databases

Multitenancy database for cloud and SaaS is another angle of co-existence of schemas. In cloud and SaaS, data of multiple customers (tenants) for cloud applications is stored in a database. Basically application and data structure are common for all customers. However, slight variations of each customer exist. Aulbach [6, 7] proposes a set of generic schema designs ranging from base tables and a methodology to customize them for each tenant while allowing data sharing between tenants. Shengqi et al. [61] propose multi-version metadata of Basic-Table combined with Extension-Table for efficient schema evolution. As enterprise applications, Weissman and Bobrowski [76] report metadata-driven architecture for multitenancy database of CRM (client relationship management) system, and Chen et al. [17] report multitenancy databases of a mobile company.

Objectives of multitenacy database is close to one of co-existence of schemas. However, multitenacy databases have worked for simple schema evolution such as addling and deleting columns and do not support relational database. In this thesis, we propose programmable strategies for co-existence relational database schemas.

### Virtual Database

View-based virtual databases have been proposed for high flexibility of schema modification [70, 63, 5]. While these works focus on providing flexibility to design a consolidated view for a particular purpose from a source database, co-existence schema requires handling arbitrary updates and sharing data by following a certain strategy. In this thesis, we propose a view-based co-existence schema to handle the above features.

### View Update Problem

In order to realize a co-existence of schemas by the view-based approach, deriving transformations between a database and a view instance is required. It is known as view update problem in the database community.

In general, ambiguity exists that a translator from an update of view instance to an update of a database is not uniquely decided from a given view because a view may discard information of a database. Bancilhon and Spyratos [8] introduce a constant complement to keep lost information by view for translation of an update of view instance, Dayal and Bernstein [22] clarifies conditions to correctly translate an update of view instance, Lechtenbörger [47, 46] studies how to compute a complement efficiently. Keller [43] proposes a dialog-based method to specify a relevant update translator. Masunaga [50] proposes a semantic-based methodology to decide which update translator is relevant.

Our proposed method to design a co-existence strategy and derive bidirectional transformations is similar to Keller's dialog-based method. While Keller's dialog describes how to translate any updates of view instance to the database, our co-existence strategy directly describes how to share data between schemas. Since a co-existence strategy designs a case that update is shared and not shared, some updates on a schema might be discarded in data sharing. Unlike a complement to keep lost information in the computation of a view instance, we propose to utilize auxiliary relations to keep lost information in data sharing.

#### Designing of Bidirectional Transformation

In the programming language community, view update problem has been widely discussed as bidirectional transformations [69, 16]. The pioneering work of Foster et al. [29] propose the first bidirectional programming languages, lenses, for defining views of tree-structured data. The lenses enable to construct a bidirectional transformation by concatenating primitive operators. Each primitive operator is *well-behaved* bidirectional transformation consisting of *get* and *put*. Several lenses are proposed for each purpose. Bohannon et al. [12] propose lenses for view update of a relational database, called relational lenses. It defines primitive operators consisting of *get* as view by enriching the SQL expression for defining a view instance of projection, selection, and join from base relations of a database and *put* to transform updates of a view instance to the base relations. Due to the ambiguity defining update translator from a given view, the effectiveness of lenses comes from limiting a user's knowledge and control of an update translator by *put* in advance.

However, the expressive power of lenses is limited because *put* is predefined. Matsuda et al. [51] propose a bidirectionalization technique to derive *put* from a given *get* by deriving a complement automatically. To fully reflect a user's intention for designing *put*, a putback-based bidirectionalization technique is proposed to derive *get* from a given *put* so that they consist of *well-behaved* bidirectional transformation. [29, 51, 57, 25, 37, 26, 44]. Tran et al. [71] apply the putback-based bidirectionalization technique to view update problem of relational database and propose BIRDS. It provides a language to formally describe an update transformation from a view instance to a database as *put* and automatically derives a view as *get*.

While the putback-based bidirectionalization technique provides full expressiveness of *put*, it is challenging for a user to describe *put*. In order to compute the updated target data from the updated source data, bidirectional transformation assumes any updates of the target data must be transformed to source data. Therefore, a user needs to describe all cases of update transformation from the target data to the source data. On the other hand, our proposed DSL makes user's work easier by limiting a description of a co-existence strategy only for cases to share data, by giving the *consistency* of updates as a relaxed version of PutGet, and by automatically deriving *well-behaved* bidirectional transformations to realize a co-existence strategy.

# 5 Realization of Co-Existence Strategies on Target-Side Database

This chapter presents how to realize co-existence strategies on the target-side database. In Chapter 4, co-existence strategies are realized on the source-side database having base relations corresponding to relations of source schema and auxiliary relations. In this chapter, co-existence strategies are realized on the target-side database having a base relation corresponding to a relation of target schema and other auxiliary relations. We explain deriving BXs between the target-side database and view instances of schemas as a realization of co-existence strategies on the target-side database in the overall procedure shown in Figure 1.2 of Chapter 1. We start with an overview and then explain how to derive BXs and mapping for data migration from a source-side database to a target-side database. We show their evaluation with experimental results. Related work is explained at the end of this chapter.

## 5.1 Overview

Before we discuss the details of deriving bidirectional transformations, we present an overview of what bidirectional transformations realize co-existence strategies on a target-side database, how to derive them, how to derive mapping for data migration, and relation with the later sections.

### BX to Realize Co-Existence Strategies

We realize co-existence strategies by two types of bidirectional transformations following the view-based approach (Figure 5.1). By giving a co-existence strategy specifying a relationship between relations $S_i$ ($i \in [1, n]$) of source schema and a relation $T$ of target schema, we turn $S_i$ for all $i$ and $T$ into view instances. $\mathbf{D}_t$ is a target-side database. It is union of a base relation corresponding to a relation of target schema and auxiliary relations for supplemental information. These auxiliary relations have different data structures from auxiliary relations of the source-side database. Details are explained later. In this chapter, we shorten a target-side database as a database unless specified otherwise.

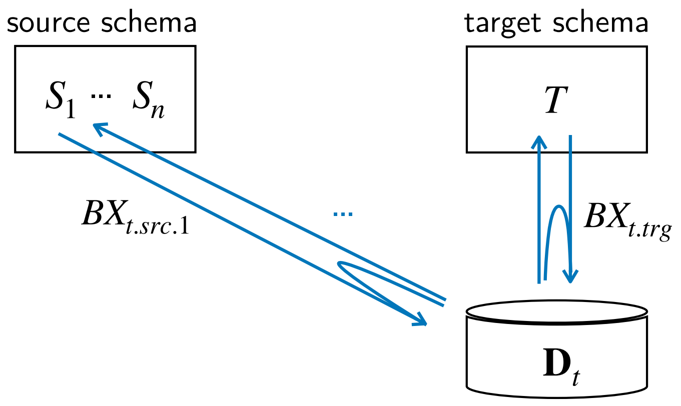


Figure 5.1: Realization of a co-existence strategy by bidirectional transformations on the target-side database.

For each $i$ ($i \in [1, n]$), $BX_{t.src.i}$ is bidirectional transformations between view instance $S_i$ of source schema and database $\mathbf{D}_t$. Its *get* (denoted as $get_{t.src.i}$) and *put* (denoted as $put_{t.src.i}$) are as followings:

$$get_{t.src.i}(\mathbf{D}_t) = S_i \tag{5.1}$$

$$put_{t.src.i}(\mathbf{D}_t, S_i{}') = \mathbf{D}_t{}' \tag{5.2}$$

where $S_i{}'$ is an updated view instance and $\mathbf{D}_t{}'$ is an updated database. We derive this bidirectional transformation so that it realizes schema evolution of a co-existence strategy between view instances of source schema and a base relation of database.

$BX_{t.trg}$ is a bidirectional transformation between view instance $T$ of target schema and database $\mathbf{D}_t$. Its *get* (denoted as $get_{t.trg}$) and *put* (denoted as $put_{t.trg}$) are as followings:

$$get_{t.trg}(\mathbf{D}_t) = T \tag{5.3}$$

$$put_{t.trg}(\mathbf{D}_t, T') = \mathbf{D}_t{}' \tag{5.4}$$

where $T'$ is an updated view instance. We derive this bidirectional transformation so that it realizes backward update sharing of a co-existence strategy between view instances of target schema and a base relation and auxiliary relations of the database.

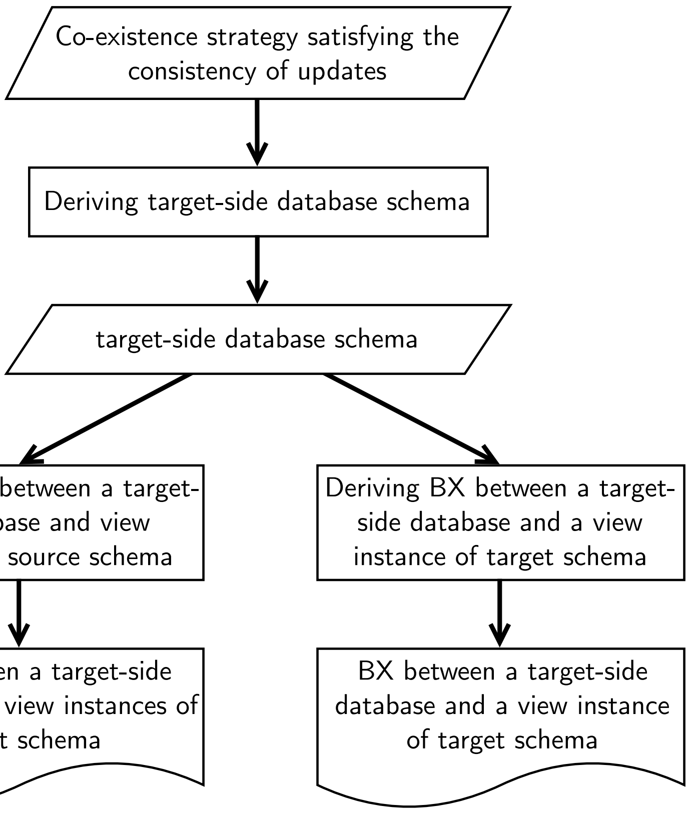

Co-existence strategy satisfying the consistency of updates
Deriving target-side database schema
target-side database schema
Deriving BXs between a target-side database and view instances of source schema
Deriving BX between a target-side database and a view instance of target schema
BXs between a target-side database and view instances of target schema
BX between a target-side database and a view instance of target schema

Figure 5.2: Procedure of deriving BXs between a target-side database and view instances of schemas.

These bidirectional transformations make updates against a view instance of sources schema shared with a view instance of target schema and vice versa through updates of database. In the forward direction, $put_{t.src.i}$ transforms an updated view instance $S_i{}'$ of source schema into an updated database $\mathbf{D}_t{}'$. And then $get_{t.trg}$ transforms the updated $\mathbf{D}_t{}'$ into an updated view instance $T'$ of target schema. In the backward direction, $put_{t.trg}$ then $get_{t.src.i}$ transforms an updated $T'$ into an updated $S_i{}'$ through an updated database $\mathbf{D}_t{}'$.

**Procedure to Derive BXs**

Figure 5.2 shows a procedure of deriving bidirectional transformations between a target-side database and view instances of schemas. This is the following procedure after a given co-existence strategy is verified whether the *consistency* of updates is satisfied in Figure 1.2.

The first step is deriving a target-side database schema. Given a co-existence strategy that specifies source schema and target schema, a target-side database schema is defined so that the database is union of a base relation corresponding to a relation of target schema and auxiliary relations required for a realization of a co-existence strategy by bidirectional transformations mentioned above.

Then the next step is deriving bidirectional transformations based on the following policy:

1. Schema evolution is replaced with *put* of $BX_{t.src.i}$ so that it transforms a view instance of source schema to a base relation corresponding to a view instance of target schema. Since schema evolution may project away attributes of a view instance of source schema, such values of projected away attributes are transformed to a complement, an auxiliary relation for lost attributes.

2. When a view instance of source schema is updated, the updated view instance must be recomputed from the updated database without loss or gain. If lost or gained tuples occurs in the recomputed view instance from the updated base relation and auxiliary relation by the first policy, *put* of $BX_{t.src.i}$ transforms such lost and gained tuples to other auxiliary relations for each.

3. Backward update sharing is replaced with *put* of $BX_{t.trg}$ so that it transforms a view instance of target schema to a base relation and auxiliary relation for lost attributes. Inserted and deleted tuples against a view instance of target schema are transformed to a base relation if backward update sharing transforms them to inserted and deleted tuples against relations of source schema. Values of the projected away attributes in inserted and deleted tuples against relations of source schema are transformed to an auxiliary relation for lost attributes.

4. When a view instance of target schema is updated, the updated view instance must be recomputed from an updated database without loss or gain. If lost or gained tuples occurs in a recomputed view instance of target schema from the updated base relation and auxiliary relation by the third policy, *put* of $BX_{t.trg}$ transforms such lost and gained tuples to other auxiliary relations for each.

Five types of auxiliary relations are utilized: auxiliary relation for lost attributes, auxiliary relation for lost tuples against a view instance of source schema, auxiliary relation for gained tuples against a view instance of source schema, auxiliary relation for lost tuples against a view instance of target schema, and auxiliary relation for gained tuples against a view instance of target schema.

In the following section, we explain details of the procedure deriving bidirectional transformations. Subsection 5.2.1 shows an outline of deriving bidirectional transformations through examples. Subsection 5.2.2 explains deriving a target-side database schema. Subsection 5.2.3 explains deriving bidirectional transformations between a target-side database and a view instance of source schema. Subsection 5.2.4 explains deriving bidirectional transformations between a target-side database and a view instance of source schema.

## Data Migration

After a co-existence strategy is realized on a source-side database and update sharing between view instances of source schema and target schema are operated based on that, a realization of the strategy on the target-side database might be required due to an increase of access to data of target schema or decrease of access to data of source schema. In that case, data migration from the source-side database to the target-side database is required. Or data migration from the target-side database to the source-side database might be required if access to data of source schema increases again. We migrate data by the following policy:

- The source-side database is migrated to the target-side database by the following two steps: transforming the source-side database to view instances of source schema and target schema by *get* of $BX_{src}$ and $BX_{trg}$ derived from a given co-existence strategy by the method shown in Chapter 4, then transforming these view instances to the target-side database by *put* of $BX_{t.src}$ and $BX_{t.trg}$ derived from the co-existence strategy by the method introduced in this chapter.
- The target-side database is migrated to the source-side database by the following two steps: transforming the target-side database to view instances of source schema and target schema by *get* of $BX_{t.src}$ and $BX_{t.trg}$ derived from the co-existence strategy by the method introduced in this chapter, then transforming

these view instances to the source-side database by *put* of $BX_{src}$ and $BX_{trg}$ derived from the co-existence strategy by the method shown in Chapter 4.

We explain the details of data migration in Section 5.3.

## 5.2 Deriving BX to Realize Co-Existence Strategies

This section explains how to derive bidirectional transformations between a target-side database and view instances from a given co-existence strategy. We first give its outline and then explain the derivation algorithms and the correctness of them.

### 5.2.1 Outline

We show an outline of how to derive bidirectional transformations based on the policy in the overview through the example. Suppose the co-existence strategy that consists of schema evolution ruled by (4.5) and backward update sharing ruled by (4.15) – (4.16) in Section 4.2. Let attribute $X$ be primary key for a relation $S_1(X, Y, Z, W)$ of source schema and a relation $T(X, Y, Z)$ of target schema.

#### Definition of Target-Side Database

By turning $S_1(X, Y, Z, W)$ and $T(X, Y, Z)$ to view instances, we define data structure of target-side database. We define target-side database schema based on the policy of deriving bidirectional transformations,

Following the first policy, let a base relation of the database be $B_T(X, Y, Z)$ that has the same attributes with a view instance $T(X, Y, Z)$ of target schema. Let an auxiliary relation be $A^c_{S_1}(X, W)$ as a complement having values of projected away attributes by schema evolution. Its attributes consist of key $X$[1] to map tuples in $S_1$ and $T$ and an projected away attribute $W$ by rule (4.5) of schema evolution.

Following the second policy, let $A^{lost}_{S_1}(X, Y, Z, W)$ and $A^{gain}_{S_1}(X, Y, Z, W)$ be auxiliary relations for lost tuples and gained tuples against a view instance $S_1$ respectively. They

[1]Since attribute $X$ is a primary key for both $S_1(X, Y, Z, W)$ and $T(X, Y, Z)$ and common for both, it is key to map tuples between them. If primary keys are not specified, a key to map tuples between the relations is common attributes of them. In this example, if $X$ is not specified as a primary key of $S_1$ and $T$, a key is common attributes $X$, $Y$, and $Z$.

have the same attributes with a view instance $S_1(X, Y, Z, W)$.

Following the fourth policy, let $A_T^{lost}(X, Y, Z)$ and $A_T^{gain}(X, Y, Z)$ be auxiliary relations for lost and gained tuples against a view instance $T$ respectively. They have the same attributes with a view instance $T(X, Y, Z)$.

## Derivation of *put* of $BX_{t.src.i}$

We start to derive $put_{t.src.1}$ of $BX_{t.src.1}$ from the co-existence strategy. Recall $put_{t.src.1}$ transforms to an updated database from a pair of database and an updated view instance $S_1{}'$ of source schema.

First, we replace rule (4.5) of schema evolution to transformations of $put_{t.src.1}$ by following the first policy. We derive transformations to the updated base relation $B_T{}'$ of the database from a pair of the database and the updated $S_1{}'$.

$$t\_evo(X, Y, Z) \;:-\; s_1(X, Y, Z, W). \tag{5.5}$$
$$t\_evo'(X, Y, Z) \;:-\; s_1{}'(X, Y, Z, W). \tag{5.6}$$
$$+b\_t(X, Y, Z) \;:-\; t\_evo'(X, Y, Z), \neg t\_evo(X, Y, Z), \neg b\_t(X, Y, Z). \tag{5.7}$$
$$-b\_t(X, Y, Z) \;:-\; \neg t\_evo'(X, Y, Z), t\_evo(X, Y, Z), b\_t(X, Y, Z). \tag{5.8}$$
$$b\_t'(X, Y, Z) \;:-\; b\_t(X, Y, Z), \neg - b\_t(X, Y, Z). \tag{5.9}$$
$$b\_t'(X, Y, Z) \;:-\; +b\_t(X, Y, Z). \tag{5.10}$$

where predicates $s_1{}'(X, Y, Z, W)$, $t\_evo(X, Y, Z)$, $t\_evo'(X, Y, Z)$, $b\_t(X, Y, Z)$, and $b\_t'(X, Y, Z)$ correspond to the updated view instance $S_1{}'$ of source schema, the result of schema evolution $T^{evo}$, the result of schema evolution from the updated view instances of source schema $T^{evo\prime}$, the base relation $B_T$, and the updated base relation $B_T{}'$ respectively. Rule (5.5) is equivalent to rule (4.5) of schema evolution and expresses a transformation to a set of tuples $T^{evo}$ from the view instance $S_1$ by schema evolution. The view instance $S_1$ is computed from database by $get_{t.src.1}$ introduced later. Rule (5.6) expresses a transformation to a set of tuples $T^{evo\prime}$ from the updated view instance $S_1{}'$ by schema evolution.

Rules (5.7) and (5.8) are a core of a transformation to the updated base relation $B_T{}'$. Rule (5.7) ((5.8)) expresses a transformation to a set of inserted tuples to (deleted tuples from) the base relation $B_T$. If tuples exist (do not exist) in $T^{evo\prime}$ but do not exist

(exist) in $T^{evo}$ and the base relation $B_T$, such tuples are added to (deleted from) the base relation $B_T$. Rule (5.9) and (5.10) are application of delta relation to transform to the updated base relation $B_T{}'$.

Second, we handle values of a projected away attribute $W$ by schema evolution. Such values are transformed to the auxiliary relation $A^c_{S_1}$ for the projected away attribute by following the second policy. As transformations of $put_{t.src.1}$, we define transformations to the updated auxiliary relation $A^c_{S_1}{}'$ of database from a pair of the database and the updated view instance $S_1{}'$ as follows:

$$+s_1(X,Y,Z,W) \coloneq s_1{}'(X,Y,Z,W), \neg s_1(X,Y,Z,W). \tag{5.11}$$
$$-s_1(X,Y,Z,W) \coloneq \neg s_1{}'(X,Y,Z,W), s_1(X,Y,Z,W). \tag{5.12}$$
$$+a\_c\_s_1(X,W) \coloneq +s_1(X,Y,Z,W), +b\_t(X,Y,Z), \neg a\_c\_s_1(X,W). \tag{5.13}$$
$$-a\_c\_s_1(X,W) \coloneq -s_1(X,Y,Z,W), -b\_t(X,Y,Z), a\_c\_s_1(X,W). \tag{5.14}$$
$$a\_c\_s_1{}'(X,W) \coloneq a\_c\_s_1(X,W), \neg - a\_c\_s_1(X,W). \tag{5.15}$$
$$a\_c\_s_1{}'(X,W) \coloneq +a\_c\_s_1(X,W). \tag{5.16}$$

where predicates $a\_c\_s_1(X,W)$, and $a\_c\_s_1{}'(X,W)$ correspond to the auxiliary relation $A^c_{S_1}$ and the updated auxiliary relation $A^c_{S_1}{}'$ respectively. Rule (5.11) ((5.12)) expresses a transformation to inserted (deleted) tuples against the view instance $S_1$. If tuples exist (do not exist) in the updated view instance $S_1{}'$ but do not exist (exist) in the original view instance $S_1$, such tuples are inserted (deleted) tuples against $S_1$.

Rules (5.13) and (5.14) are core of a transformation to the updated auxiliary relation $A^c_{S_1}{}'$. Rules express a transformation to a set of inserted and deleted tuples against $A^c_{S_1}$. These rules are equivalent to the following expressions by relational algebra.

$$\Delta^+_{A^c_{S_1}}(X,W) = \pi_{\{X,W\}}(\Delta^+_{S_1}(X,Y,Z,W) \bowtie \Delta^+_{B_T}(X,Y,Z)) \cap \neg A^c_{S_1}(X,W)$$
$$\Delta^-_{A^c_{S_1}}(X,W) = \pi_{\{X,W\}}(\Delta^-_{S_1}(X,Y,Z,W) \bowtie \Delta^-_{B_T}(X,Y,Z)) \cap A^c_{S_1}(X,W)$$

where predicates $+s_1(X,Y,Z,W)$, $-s_1(X,Y,Z,W)$, $+a\_c\_s_1(X,W)$, $-a\_c\_s_1(X,W)$, $+s_1(X,Y,Z,W)$, $+b\_t(X,Y,Z)$, and $-b\_t(X,Y,Z)$ in Datalog rules correspond to sets of inserted and deleted tuples of $\Delta^+_{S_1}$, $\Delta^-_{S_1}$, $\Delta^+_{A^c_{S_1}}$, $\Delta^-_{A^c_{S_1}}$, $\Delta^+_{B_T}$, and $\Delta^-_{B_T}$ respectively. Join $\Delta^+_{S_1} \bowtie \Delta^+_{B_T}$ results in inserted tuples to the view instance $S_1$ with values of a projected away attribute by schema evolution. Then if its projection to values of a key $X$ and the projected away attribute $W$ does not exist in the auxiliary relation $A^c_{S_1}$, such tuples are inserted into it. Similarly, join $\Delta^-_{S_1} \bowtie \Delta^-_{B_T}$ results in deleted tuples from the view instance $S_1$ with values of the projected away attribute by schema evolution. Then if its projection to values of the key $X$ and the projected away attribute $W$ exist in the auxiliary relation $A^c_{S_1}$, such tuples are deleted from it. Rule (5.15) and (5.16) are application of delta relation to transform to the updated auxiliary relation $A^c_{S_1}{}'$.

Third, we derive transformations of $put_{t.src.1}$ to update auxiliary relations $A^{lost}_{S_1}$ and $A^{gain}_{S_1}$ by following the third policy. Since this outline is almost same with transformations to auxiliary relations $A^{gain}_T$ and $A^{loss}_T$ explained in Section 4.2, we skip its explanation here and show details in the algorithm introduced in Section 5.2.3 later.

#### Derivation of *get* of $BX_{t.src.i}$

We define transformations of $get_{t.src.1}$ of $BX_{t.src.1}$ by following the third policy as follows:

$$s_1\_evo(X,Y,Z,W) :- b\_t(X,Y,Z), a\_c\_s_1(X,W). \tag{5.17}$$

$$s_1(X,Y,Z,W) :- s_1\_evo(X,Y,Z,W), \neg a\_gain\_s_1(X,Y,Z,W). \tag{5.18}$$

$$s_1(X,Y,Z,W) :- a\_lost\_s_1(X,Y,Z,W). \tag{5.19}$$

where predicate $s_1\_evo(X,Y,Z,W)$ in Datalog rule corresponds to a relation $S^{evo}_1$. Rule (5.17) is equivalent to the following expression by relational algeba.

$$S^{evo}_1(X,Y,Z,W) = \pi_{\{X,Y,Z,W\}}(B_T(X,Y,Z) \bowtie A^c_{S_1}(X,W))$$

This expression is a transformation to the relation $S^{evo}_1$ by projection to attributes of the view instance $S_1$ from join of the base relation $B_T$ and the auxiliary relation for lost attributes $A^c_{S_1}$. Join of $B_T$ and $A^c_{S_1}$ results in tuples which have values of projected away attributes by schema evolution.

If $S^{evo}_1$ does not cause any lost or gained tuples against the view instance $S_1$, $S^{evo}_1$ is enough to compute $S_1$. Recall auxiliary relations $A^{lost}_{S_1}$ and $A^{gain}_{S_1}$ are for such lost and gained tuples against updates of view instance $S_1$. To compute the view instance of source schema $S_1$ without loss or gain, rules (5.18) and (5.19) express transformations to a view instance $S_1$ by deleting gained tuples in auxiliary relation $A_{S_1}^{gain}$ and adding lost tuples in auxiliary relation $A_{S_1}^{lost}$ against a relation $S_1^{evo}$.

**Derivation of *put* of $BX_{t.trg}$**

We derive $put_{t.trg}$ of $BX_{t.src.1}$ from the co-existence strategy so that $put_{t.trg}$ transforms to the updated database from a pair of the database and the updated view instance $T'$ of target schema.

First, following the fourth policy, we define transformations of $put_{t.trg}$ to the updated base relation $B_T{}'$ of the database from a pair of the database and the updated $T'$ as follows:

$$+t(X,Y,Z) \text{ :- } t'(X,Y,Z), \neg t(X,Y,Z). \tag{5.20}$$

$$-t(X,Y,Z) \text{ :- } \neg t'(X,Y,Z), t(X,Y,Z). \tag{5.21}$$

$$+s_1(X,Y,Z,W) \text{ :- } +t(X,Y,Z), \neg s_1(X,Y,Z,\_), Y < 100, W = \text{` '}. \tag{5.22}$$

$$-s_1(X,Y,Z,W) \text{ :- } -t(X,Y,Z), s_1(X,Y,Z,W), Y < 100. \tag{5.23}$$

$$+b\_t(X,Y,Z) \text{ :- } +s_1(X,Y,Z,W), +t(X,Y,Z), \neg b\_t(X,Y,Z). \tag{5.24}$$

$$-b\_t(X,Y,Z) \text{ :- } -s_1(X,Y,Z,W), -t(X,Y,Z), b\_t(X,Y,Z). \tag{5.25}$$

$$b\_t'(X,Y,Z) \text{ :- } b\_t(X,Y,Z), \neg - b\_t(X,Y,Z). \tag{5.26}$$

$$b\_t'(X,Y,Z) \text{ :- } +b\_t(X,Y,Z). \tag{5.27}$$

where predicates $t'(X,Y,Z)$, $b\_t(X,Y,Z)$, and $b\_t'(X,Y,Z)$ correspond to the updated view instance $T'$ of target schema, the base relation $B_T$, and the updated base relation $B_T{}'$ respectively. Predicates $+s_1(X,Y,Z,W)$, $-s_1(X,Y,Z,W)$, $+t(X,Y,Z)$, and $-t(X,Y,Z)$ correspond to sets of inserted and deleted tuples of $\Delta_{S_1}^+$, $\Delta_{S_1}^-$, $\Delta_T^+$, and $\Delta_T^-$ respectively.

Rule (5.20) ((5.21)) expresses a transformation to inserted (deleted) tuples against a view instance $T$. If tuples exist (do not exist) in an updated view instance $T'$ but do not exist (exist) in the original view instance $T$, such tuples are inserted (deleted) tuples against $T$. Rules (5.22) and (5.23) are transformations of backward update sharing. They are a core of a transformation to an updated base relation $B_T{}'$. Rules express transformations to sets of inserted and deleted tuples against $B_T$. These rules are equivalent to the following expressions by relational algebra.

$$\Delta^+_{B_T}(X,Y,Z) = \pi_{\{X,Y,Z\}}(\Delta^+_{S_1}(X,Y,Z,W) \bowtie \Delta^+_T(X,Y,Z)) \cap \neg B_T(X,Y,Z)$$
$$\Delta^-_{B_T}(X,Y,Z) = \pi_{\{X,Y,Z\}}(\Delta^-_{S_1}(X,Y,Z,W) \bowtie \Delta^-_T(X,Y,Z)) \cap B_T(X,Y,Z)$$

Join $\Delta^+_{S_1} \bowtie \Delta^+_T$ results in tuples inserted into the view instance $T$ and the view instance $S_1$ by backward update transformation. Then its projection to attributes of the base relation $B_T$ is a set of inserted tuples to $B_T$ if a result of projection does not exist in $B_T$. Similarly, join $\Delta^-_{S_1} \bowtie \Delta^-_T$ results in tuples deleted from the view instance $T$ and the view instance $S_1$ by backward update transformation. Then its projection to attributes of the base relation $B_T$ is a set of deleted tuples from $B_T$, if a result of projection exist in $B_T$. Rule (5.24) and (5.25) are application of delta relation to transform to an updated base relation $B_T{}'$.

Second, we define transformations to the updated auxiliary relation for lost attributes $A^c_{S_1}{}'$ of the database from a pair of the database and the updated view instance $T'$. Following the fourth policy, we define such transformations as follows:

$$+a\_c\_s_1(X,W) :- +s_1(X,Y,Z,W), \neg a\_c\_s_1(X,W). \tag{5.28}$$
$$-s_1{}'(X,Y,Z,W) :- -s_1(X,Y,Z,W), \neg + s_1(X,\_,\_,\_). \tag{5.29}$$
$$-a\_c\_s_1(X,W) :- -s_1'(X,Y,Z,W), a\_c\_s_1(X,W). \tag{5.30}$$
$$a\_c\_s_1{}'(X,W) :- a\_c\_s_1(X,W), \neg - a\_c\_s_1(X,W). \tag{5.31}$$
$$a\_c\_s_1{}'(X,W) :- +a\_c\_s_1(X,W). \tag{5.32}$$

where predicates $a\_c\_s_1(X,W)$, $a\_c\_s_1{}'(X,W)$, and $-s_1{}'(X,Y,Z,W)$ correspond to the auxiliary relation $A^c_{S_1}$, the updated auxiliary relation $A^c_{S_1}{}'$, and the relation $\Delta^-_{S_1}{}'$ respectively. Predicates $+a\_c\_s_1(X,W)$ and $+s_1(X,Y,Z,W)$ correspond to relations $\Delta^+_{A^c_{S_1}}(X,W)$ and $\Delta^+_{S_1}(X,Y,Z,W)$ respectively.

Rule (5.28) expresses a transformation to a set of inserted tuples to the auxiliary relation $A^c_{S_1}$. This rule is equivalent to the following expression by relational algebra.

$$\Delta^+_{A^c_{S_1}}(X,W) = \pi_{\{X,W\}}(\Delta^+_{S_1}(X,Y,Z,W)) \cap \neg A^c_{S_1}(X,W)$$

Note that $\Delta^+_{S_1}(X,Y,Z,W)$ as a set of deleted tuples from $S_1(X,Y,Z,W)$ is computed by rule (5.22) of backward update sharing. If tuples resulted in projection of $\Delta^{+}_{S_1}(X, Y, Z, W)$ to a set of tuples with attribute $X$ and $W$ do not exist in the auxiliary relation $A^{c}_{S_1}(X, W)$, such tuples are added to $A^{c}_{S_1}(X, W)$.

Rule (5.29) is to exclude deleted tuples from $S_1$ in replacement of tuples of $S_1$. When tuples of $T$ are replaced, backward update sharing may result in sets of inserted and deleted tuples against $S_1$ with the same values of the key. For example, suppose a tuple $\langle x1, y1, z1, w1\rangle$ of $S_1$, a tuple $\langle x1, w1\rangle$ of $A^{c}_{S_1}$, an inserted tuple $\langle x1, y2, z2, w1\rangle$ of $\Delta^{+}_{S_1}$, and a deleted tuple $\langle x1, y1, z1, w1\rangle$ of $\Delta^{-}_{S_1}$ to replace the tuple $\langle x1, y1, z1, w1\rangle$ of $S_1$ to $\langle x1, y2, z2, w1\rangle$. All tuples have the same value $x1$ of attribute $X$ as the key. In this case, we do not need to delete the tuple $\langle x1, w1\rangle$ of $A^{c}_{S_1}$ by $\langle x1, y1, z1, w1\rangle$ of $\Delta^{-}_{S_1}$ because a value of attribute $W$ is not updated. The rule expresses a transformation to $\Delta^{-}_{S_1}{}'$ by excluding such tuples from $\Delta^{-}_{S_1}$.

Rule (5.30) expresses a transformation to a set of deleted tuples from the auxiliary relation $A^{c}_{S_1}$. This rule is equivalent to the following expression by relational algebra.

$$\Delta^{-}_{A^{c}_{S_1}}(X, W) = \pi_{\{X,W\}}(\Delta^{-}_{S_1}{}'(X, Y, Z, W)) \cap A^{c}_{S_1}(X, W)$$

If tuples as a result of projection $\Delta^{-}_{S_1}{}'(X, Y, Z, W)$ to attributes $X$ and $W$ exist in the auxiliary relation $A^{c}_{S_1}(X, W)$, such tuples are deleted from it. Rules (5.31) and (5.32) are application of delta relation to transform the updated auxiliary relation $A^{c}_{S_1}{}'$.

Third, we derive transformations $put_{t.trg}$ to update auxiliary relations $A^{lost}_{T}$ and $A^{gain}_{T}$ by following the fifth policy. Since this outline is almost same with transformations to auxiliary relations $A^{gain}_{T}$ and $A^{loss}_{T}$ explained in Section 4.2, we skip its explanation here and show details in the algorithm introduced in Section 5.2.4 later.

### Derivation of *get* of $BX_{t.trg}$

We define $get_{t.trg}$ of $BX_{t.trg}$ by following the fifth policy as follows:

$$t(X, Y, Z) :- b\_t(X, Y, Z), \neg a\_gain\_t(X, Y, Z). \tag{5.33}$$

$$t(X, Y, Z) :- a\_lost\_t(X, Y, Z). \tag{5.34}$$

If tuples of a base relation $B_T$ do not cause any lost or gained tuples against view instance $T$, $B_T$ is enough to compute it. Recall auxiliary relations $A^{lost}_{T}$ and $A^{gain}_{T}$ are

**Algorithm 5.1** Deriving Target-Side Database Schema

```
Input: a co-existence strategy $P$
Output: a set of relation names of target-side database schema $tar\_db$
    $t, \vec{Y} \leftarrow$ a relation name and attributes of target schema defined in $P$
    $base \leftarrow \{ ("b\_" \,\&\, t, \vec{Y}) \}$    // base relation name
    $a\_lost\_t \leftarrow \{ ("a\_lost\_" \,\&\, t, \vec{Y}) \}$    // auxiliary relation name for lost tuple
    $a\_gain\_t \leftarrow \{ ("a\_gain\_" \,\&\, t, \vec{Y}) \}$    // auxiliary relation name for gained tuple
    $n \leftarrow$ a number of relation names of source schema defined in $P$
    $src\_aux \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        $s_i, \vec{X_i} \leftarrow$ $i$-th relation name and attributes of source schema defined in $P$
        $key \leftarrow$ primary key of $s_i$ and $t$ if both are the same, otherwise $\vec{X_i} \cap \vec{Y}$.
        $\vec{Z_i} \leftarrow key \cup (\vec{X_i} \cap \neg\vec{Y})$
        $a\_c\_s_i \leftarrow \{ ("a\_c\_" \,\&\, s_i, \vec{Z_i}) \}$    //auxiliary relation name for lost attributes
        $a\_lost\_s_i \leftarrow \{ ("a\_lost\_" \,\&\, s_i, \vec{X_i}) \}$    //auxiliary relation name for lost tuples
        $a\_gain\_s_i \leftarrow \{ ("a\_gain\_" \,\&\, s_i, \vec{X_i}) \}$ //auxiliary relation name for gained tuples
        $src\_aux \leftarrow arc\_aux \cup a\_c\_s_i \cup a\_lost\_s_i \cup a\_gain\_s_i$
    end for
    $tar\_db \leftarrow base \cup src\_aux$
    return $tar\_db$
```

for such lost and gained tuples against updates of view instance $T$. To compute a view instance of target schema $T$ without loss or gain, rules (5.33) and (5.34) express transformations to a view instance $T$ by deleting gained tuples in auxiliary relation $A_T^{gain}$ and adding lost tuples in auxiliary relation $A_T^{lost}$ against a base relation $B_T$.

### 5.2.2 Deriving Target-Side Database Schema

We give Algorithm 5.1 to derive the target-side database schema. The input is a co-existence strategy. The output is a set of relation names of base relations and auxiliary relations. A base relation name corresponds to a relation name of target schema. The number of derived auxiliary relation names varies depending on the number of relation names of source schema. The algorithm defines three auxiliary relation names for one relation name of source schema and two auxiliary relation names for one relation name of target schema.

The following example shows relation names which the algorithm outputs from a given co-existence strategy.

**Example 12.** Suppose the co-existence strategy in Example 8 as input to Algorithm 5.1. Since the co-existence strategy specifies one relation name of source schema, the algorithm sets $n$ as 1. Then, the algorithm sets $s_1$ as $s_1$, $\vec{X_1}$ as $\{X, Y, Z, W\}$, $t$ as $t$, $\vec{Y}$ as $\{X, Y, Z\}$ from relation names $s_1(X, Y, Z, W)$ of source schema and $t(X, Y, Z)$ of target schema.

Based on them, the algorithm outputs the following relation names: $b\_t(X, Y, Z)$ as the base relation name, $a\_lost\_t(X, Y, Z)$ as the auxiliary relation name for lost tuples of a view instance $T(X, Y, Z)$ of target schema, $a\_gain\_t(X, Y, Z)$ as the auxiliary relation name for gained tuples of the view instance $T(X, Y, Z)$, $a\_c\_s_1(X, W)$ as the auxiliary relation name for lost attributes of a view instance $S_1(X, Y, Z, W)$ of source schema, $a\_lost\_s_1(X, Y, Z, W)$ as the auxiliary relation name for lost tuples of $S_1(X, Y, Z, W)$, and $a\_gain\_s_1(X, Y, Z, W)$ as the auxiliary relation name for gained tuples of $S_1(X, Y, Z, W)$. □

### 5.2.3 Deriving BX between Target-Side Database and View Instance of Source Schema

We give Algorithm 5.2 to derive bidirectional transformations between the view instance $S_i$ of source schema and the target-side database. The input is a co-existence strategy satisfying the *consistency* of updates. The output is bidirectional transformations $BX_{t.src.i}$ for all $i$ ($i \in [1, n]$) expressed by Datalog rules. The outline to derive bidirectional transformation is shown in Subsection 5.2.1. In the algorithm, a derivation of constraints for a primary key follows a derivation method in [71].

**Algorithm 5.2** Deriving BXs between Target-Side Database and View Instances of Source Schema

```
Input: a co-existence strategy $P$ satisfying the *consistency* of updates
Output: bidirectional transformations of source schema $BX_{t.src}$
    // Specification described in $P$
    $n \leftarrow$ a number of relation names of source schema
    for $i = 1$ to $n$ do
        $s_i, \vec{X_i} \leftarrow$ a relation name and attributes of source schema
                                        // corresponding to a relation $S_i(\vec{X_i})$
    end for
    $t, \vec{Y} \leftarrow$ a relation name and attributes of target schema
                                        // corresponding to a view instance $T(\vec{Y})$
    $C \leftarrow$ a set of constraints
    $f \leftarrow$ a set of Datalog rules to transform to $T(\vec{Y})$        //schema evolution

    // updated view instance $S_i'(X_i)$
    // base relation $B_T(\vec{Y})$
    // auxiliary relations $A^c_{S_i}(\vec{Z_i})$, $A^{lost}_{S_i}(\vec{X_i})$, $A^{gain}_{S_i}(\vec{X_i})$, $A^{lost}_T(\vec{Y})$, and $A^{gain}_T(\vec{Y})$
    $trg\_db \leftarrow$ a set of base and auxiliary relation names defined by Algorithm 5.1
    $r_S \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        $r_{S_i} \leftarrow$ a set of Datalog rules to transform to a view instance $S_i(\vec{X_i})$ as
                $S_i^{evo}(\vec{X_i}) = \pi_{attr(S_i)}(B_T(\vec{Y}) \bowtie A^c_{S_i}(\vec{Z_i}))$ and
                $S_i(\vec{X_i}) = (S_i^{evo}(\vec{X_i}) \cap \neg A^{gain}_{S_i}(\vec{X_i})) \cup A^{lost}_{S_i}(\vec{X_i})$
        $r_S \leftarrow r_S \cup r_{S_i}$
    end for
    $BX_{t.src} \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        // $get_{t.src.i}$
        $get_{t.src.i} \leftarrow r_{S_i}$
        // $put_{t.src.i}$
        $r_{evo} \leftarrow$ a set of Datalog rules of $f$ to transform to $T^{evo}(\vec{Y})$ by replacing a
                predicate symbol $t$ to a predicate symbol corresponding to $T^{evo}(\vec{Y})$
        $r'_{evo} \leftarrow$ a set of Datalog rules of $f$ to transform to $T^{evo\prime}(\vec{Y})$ by replacing
                predicate symbols $s_i$ and $t$ to predicate symbols corresponding
                to $S_i'(\vec{X_i})$ and $T^{evo\prime}(\vec{Y})$
        $r'_{base} \leftarrow$ a set of Datalog rules to transform to updated $B_T'(\vec{Y})$ as
                $\Delta^+_{B_T}(\vec{Y}) = T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \cap \neg B_T(\vec{Y})$,
                $\Delta^-_{B_T}(\vec{Y}) = \neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y}) \cap B_T(\vec{Y})$, and
                $B_T'(\vec{Y}) = (B_T(\vec{Y}) \cap \neg \Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y})$
    $r'_{aux.c} \leftarrow$ a set of Datalog rules to transform to updated $A^{c}_{S_i}{}'(\vec{Z_i})$ as
        $\Delta^{+}_{S_i}(\vec{X_i}) = S_i{}'(\vec{X_i}) \cap \neg S_i(\vec{X_i})$,
        $\Delta^{-}_{S_i}(\vec{X_i}) = \neg S_i{}'(\vec{X_i}) \cap S_i(\vec{X_i})$,
        $\Delta^{+}_{A^{c}_{S_i}}(\vec{Z_i}) = \pi_{\vec{Z_i}}(\Delta^{+}_{S_i}(\vec{X_i}) \bowtie \Delta^{+}_{B_T}(\vec{Y})) \cap \neg A^{c}_{S_i}(\vec{Z_i})$,
        $\Delta^{-}_{A^{c}_{S_i}}(\vec{Z_i}) = \pi_{\vec{Z_i}}(\Delta^{-}_{S_i}(\vec{X_i}) \bowtie \Delta^{-}_{B_T}(\vec{Y})) \cap A^{c}_{S_i}(\vec{Z_i})$, and
        $A^{c}_{S_i}{}'(\vec{Z_i}) = (A^{c}_{S_i}(\vec{Z_i}) \cap \neg \Delta^{-}_{A^{c}_{S_i}}(\vec{Z_i})) \cup \Delta^{+}_{A^{c}_{S_i}}(\vec{Z_i})$
    $r'_{aux} \leftarrow$ a set of Datalog rules to transform to updated $A^{lost\prime}_{S_i}(\vec{X_i})$, $A^{gain\prime}_{S_i}(\vec{X_i})$,
        $A^{lost\prime}_{T}(\vec{T})$, and $A^{gain\prime}_{T}(\vec{Y})$ as
        $S_i{}''(\vec{X_i}) = \pi_{attr(S_i)}(B_T{}'(\vec{Y}) \bowtie A^{c}_{S_i}{}'(\vec{Z_i}))$,
        $\Delta^{+}_{A^{lost}_{S_i}}(\vec{X_i}) = S_i{}'(\vec{X_i}) \cap \neg S_i{}''(\vec{X_i}) \cap \neg A^{lost}_{S_i}(\vec{X_i})$,
        $\Delta^{-}_{A^{lost}_{S_i}}(\vec{X_i}) = \neg S_i{}'(\vec{X_i}) \cap A^{lost}_{S_i}(\vec{X_i})$,
        $\Delta^{+}_{A^{gain}_{S_i}}(\vec{X_i}) = \neg S_i{}'(\vec{X_i}) \cap S_i{}''(\vec{X_i}) \cap \neg A^{gain}_{S_i}(\vec{X_i})$,
        $\Delta^{-}_{A^{gain}_{S_i}}(\vec{X_i}) = (S_i{}'(\vec{X_i}) \cap A^{gain}_{S_i}(\vec{X_i})) \cup (\neg S_i{}''(\vec{X_i}) \cap A^{gain}_{S_i}(\vec{X_i}))$,
        $\Delta^{-}_{A^{lost}_{T}}(\vec{Y}) = (\Delta^{+}_{B_T}(\vec{Y}) \cap A^{lost}_{T}(\vec{Y})) \cup (\Delta^{-}_{B_T}(\vec{Y}) \cap A^{lost}_{T}(\vec{Y}))$,
        $\Delta^{-}_{A^{gain}_{T}}(\vec{Y}) = (\Delta^{+}_{B_T}(\vec{Y}) \cap A^{gain}_{T}(\vec{Y})) \cup (\Delta^{-}_{B_T}(\vec{Y}) \cap A^{gain}_{T}(\vec{Y}))$,
        $A^{lost\prime}_{S_i}(\vec{X_i}) = (A^{lost}_{S_i}(\vec{X_i}) \cap \neg \Delta^{-}_{A^{lost}_{S_i}}(\vec{X_i})) \cup \Delta^{+}_{A^{lost}_{S_i}}(\vec{X_i})$,
        $A^{gain\prime}_{S_i}(\vec{X_i}) = (A^{gain}_{S_i}(\vec{X_i}) \cap \neg \Delta^{-}_{A^{gain}_{S_i}}(\vec{X_i})) \cup \Delta^{+}_{A^{gain}_{S_i}}(\vec{X_i})$,
        $A^{lost\prime}_{T}(\vec{Y}) = A^{lost}_{T}(\vec{Y}) \cap \neg \Delta^{-}_{A^{lost}_{T}}(\vec{Y})$, and
        $A^{gain\prime}_{T}(\vec{Y}) = A^{gain}_{T}(\vec{Y}) \cap \neg \Delta^{-}_{A^{gain}_{T}}(\vec{Y})$
    $c \leftarrow$ a set of constrains in $C$ that a predicate with symbol $t$ does not appears
        in body and predicate symbol $s_i$ is replaced to a corresponding predicate
        symbols of $S_i{}'(\vec{X_i})$.
    $c_{aux} \leftarrow$ a set of constraints of
        $A^{lost}_{S_i}(\vec{X_i}) \cap A^{gain}_{S_i}(\vec{X_i}) = \emptyset$ and
        $A^{gain}_{S_i}(\vec{X_i}) \cap \neg S^{evo}_{i}(\vec{X_i}) = \emptyset$
    $c_{pk} \leftarrow$ a set of constrains for primary key of relation name $s_i$ if it is specified
    $put_{src.i} \leftarrow r_S \cup r_{evo} \cup r'_{evo} \cup r'_{base} \cup r'_{aux.c} \cup r'_{aux} \cup c \cup c_{aux} \cup c_{pk}$
    $BX_{t.src.i} \leftarrow \{(get_{t.src.i}, put_{t.src.i})\}$
    $BX_{t.src} \leftarrow BX_{t.src} \cup BX_{t.src.i}$
**end for**
**return** $BX_{t.src}$
```

The following example shows how the algorithm derives bidirectional transformation.

**Example 13.** Suppose the co-existence strategy in Example (8) as input to Algorithm (5.2). The algorithm sets $n$ as 1 because the co-existence strategy specifies one relation name of source schema. Based on relation names derived in Example (12), the algorithm outputs bidirectional transformation $BX_{t.src.1}$

The algorithm outputs transformations of $get_{t.src.1}$ consisting of Datalog rules (5.17) – (5.17) shown in Subsection 5.2.1. The algorithm outputs the following transformations of $put_{t.src.1}$: $r_T$ as rule (5.5), $r_{T'}$ as rule (5.6), $r'_{base}$ as rules (5.7) – (5.10), and $r'_{aux.c}$ as rules (5.11) – (5.16). Further the algorithm outputs the following Datalog rules of $r'_{aux}$ to transform to the updated auxiliary relations $\Delta^{lost'}_{S_1}$, $\Delta^{gain'}_{S_1}$, $\Delta^{lost'}_{T}$, and $\Delta^{gain'}_{T}$.

$r'_{aux}$:

$$\begin{aligned}
s_1{}''(X,Y,Z,W) &:- b\_t'(X,Y,Z) \bowtie a\_c\_s_1{}'(X,W).\\
+a\_lost\_s_1(X,Y,Z,W) &:- s_1{}'(X,Y,Z,W), \neg s_1{}''(X,Y,Z,W), \neg a\_lost\_s_1\_ins(X,Y,Z,W).\\
-a\_lost\_s_1(X,Y,Z,W) &:- \neg s_1{}'(X,Y,Z,W), a\_lost\_s_1(X,Y,Z,W).\\
+a\_gain\_s_1(X,Y,Z,W) &:- \neg s_1{}'(X,Y,Z,W), s_1{}''(X,Y,Z,W), \neg a\_gain\_s_1\_ins(X,Y,Z,W).\\
-a\_gain\_s_1(X,Y,Z,W) &:- s_1{}'(X,Y,Z,W), a\_gain\_s_1(X,Y,Z,W).\\
-a\_gain\_s_1(X,Y,Z,W) &:- \neg s_1{}''(X,Y,Z,W), a\_gain\_s_1(X,Y,Z,W).\\
-a\_lost\_t(X,Y,Z) &:- +b\_t(X,Y,Z), a\_lost\_t(X,Y,Z).\\
-a\_lost\_t(X,Y,Z) &:- -b\_t(X,Y,Z), a\_lost\_t(X,Y,Z).\\
-a\_gain\_t(X,Y,Z) &:- +b\_t(X,Y,Z), a\_gain\_t(X,Y,Z).\\
-a\_gain\_t(X,Y,Z) &:- -b\_t(X,Y,Z), a\_gain\_t(X,Y,Z).\\
a\_lost\_s_1{}'(X,Y,Z,W) &:- a\_lost\_s_1(X,Y,Z,W), \neg -a\_lost\_s_1(X,Y,Z,W).\\
a\_lost\_s_1{}'(X,Y,Z,W) &:- +a\_lost\_s_1(X,Y,Z,W).\\
a\_gain\_s_1{}'(X,Y,Z,W) &:- a\_gain\_s_1(X,Y,Z,W), \neg -a\_gain\_s_1(X,Y,Z,W).\\
a\_gain\_s_1{}'(X,Y,Z,W) &:- +a\_gain\_s_1(X,Y,Z,W).\\
a\_lost\_t'(X,Y,Z,W) &:- a\_lost\_t(X,Y,Z,W), \neg -a\_lost\_t(X,Y,Z,W).\\
a\_gain\_t'(X,Y,Z,W) &:- a\_gain\_t(X,Y,Z,W), \neg -a\_gain\_t(X,Y,Z,W).
\end{aligned}$$

where predicates with symbols $a\_lost\_s_1{}'$, $a\_gain\_s_1{}'$, $a\_lost\_t'$, and $a\_gain\_t'$ correspond to the updated auxiliary relations $A^{lost'}_{S_1}$, $A^{gain'}_{S_1}$, $A^{lost'}_{T}$, and $A^{gain'}_{S_1}$ respectively.

The algorithm outputs Datalog rules of $c$ for a constraint that predicate symbol $t$ does not appears in its body, $c_{aux}$ for constraints applied to the auxiliary relations, and $c_{pk}$ for constraints of a primary key.

$c$:

$$\bot() :- s_1{}'(X, Y, Z, W), Y \leq 0.$$

$c_{aux}$:

$$\bot() :- a\_lost\_s_1(X, Y, Z, W), a\_gain\_s_1(X, Y, Z, W).$$
$$\bot() :- a\_gain\_s_1(X, Y, Z, W), \neg s_1\_evo(X, Y, Z, W).$$

where predicates with symbols $s_1\_evo$ corresponds to a relation $S^{evo}_1$.

$c_{pk}$:

$$\bot() :- s_1{}'(X, Y, Z, W), s_1{}'(X, Y1, Z1, W1), Y \neq Y1.$$
$$\bot() :- s_1{}'(X, Y, Z, W), s_1{}'(X, Y1, Z1, W1), Z \neq Z1.$$
$$\bot() :- s_1{}'(X, Y, Z, W), s_1{}'(X, Y1, Z1, W1), W \neq W1.$$

□

## Properties

Based on transformations of $get_{t.src.i}$ and $put_{t.src.i}$ for each $i$ ($i \in [1, n]$) derived by Algorithm 5.2, the following lemmas and a proposition are satisfied. Lemmas state that particular relationships of the auxiliary relations and the base relation are kept. Note that they are given as constraints in the algorithm.

**Lemma 5.1** (Disjointness of Auxiliary Relations of Target-Side Database for Source Schema's View Instance)**.** *The auxiliary relations $A^{lost}_{S_i}(\vec{X}_i)$ and $A^{gain}_{S_i}(\vec{X}_i)$ for source* *schema's view instance $S_i(\vec{X_i})$ are disjoint for each i ($i \in [1, n]$).*

$$A^{lost}_{S_i}(\vec{X_i}) \cap A^{gain}_{S_i}(\vec{X_i}) = \emptyset$$

□

The proof is available in Appendix A.7.

**Lemma 5.2** (Inclusion of Auxiliary Relation of Target-Side Database for Source Schema's View Instance)**.** *The auxiliary relations $A^{gain}_{S_i}(\vec{X_i})$ for source schema's view instance $S_i(\vec{X_i})$ is included in the relation $S^{evo}_i(\vec{X_i})$ defined in Algorithm 5.2 for each i ($i \in [1, n]$).*

$$A^{gain}_{S_i}(\vec{X_i}) \subseteq S^{evo}_i(\vec{X_i})$$

□

The proof is available in Appendix A.8.

The following proposition states that a bidirectional transformations $BX_{t.src.i}$ for each $i$ ($i \in [1, n]$) derived by Algorithm 5.2 is well-behaved.

**Proposition 5.3** (Well-Behaveness of BX between Target-Side Database and View Instance of Source Schema)**.** *Given a co-existence strategy between relations of source schema and a relation of target schema and turning each relation to view instance, bidirectional transformation $BX_{t.src.i}$ for each i ($i \in [1, n]$) derived by Algorithm 5.2 is well-behaved by satisfying* GetPut *and* PutGet *laws.* □

The proof is available in Appendix A.9.

### 5.2.4 Deriving BX between Target-Side Database and View Instance of Target Schema

We give Algorithm 5.3 to derive a bidirectional transformation $BX_{t.trg}$ between a view instance of target schema and target-side database. The input is a co-existence strategy satisfying the *consistency* of updates. The output is a bidirectional transformation $BX_{t.trg}$ expressed by Datalog rules. The outline to derive a bidirectional transformation is shown in Subsection 5.2.1. In the algorithm, a derivation of constraints from a definition of primary key follows the derivation method in [71].

The following example shows how the algorithm derives a bidirectional transformation.

**Example 14.** Suppose the co-existence strategy in Example 8 as input to Algorithm 5.3. The algorithm sets $n$ as 1 because the co-existence strategy specifies one relation name of source schema. Based on relation names derived in Example 12, the algorithm outputs a bidirectional transformation $BX_{t.trg}$.

The algorithm outputs transformations of $get_{t.trg}$ consisting of Datalog rules (5.33) and (5.34) shown in Subsection 5.2.1. The algorithm outputs the following transformations of $put_{t.trg}$: $r_S$ as (5.17) – (5.19), $r_{\Delta T}$ as (5.20) – (5.21), $r_{back}$ as (5.22) – (5.25) and (5.28) – (5.30), $r'_{base}$ as (5.26) – (5.27).

Further the algorithm outputs the following Datalog rules of $r'_{aux.T}$ and $r'_{aux.S}$ to transform to the updated auxiliary relations $\Delta_T^{lost\prime}$, $\Delta_T^{gain\prime}$, $\Delta_{S_1}^{lost\prime}$, and $\Delta_{S_1}^{gain\prime}$

$r'_{aux.T}$:

$$\begin{aligned}
+a\_lost\_t(X,Y,Z) &:- t'(X,Y,Z), \neg b\_t'(X,Y,Z), \neg a\_lost\_t(X,Y,Z).\\
-a\_lost\_t(X,Y,Z) &:- \neg t'(X,Y,Z), a\_lost\_t(X,Y,Z).\\
+a\_gain\_t(X,Y,Z) &:- \neg t'(X,Y,Z), b\_t'(X,Y,Z), \neg a\_gain\_t(X,Y,Z).\\
-a\_gain\_t(X,Y,Z) &:- t'(X,Y,Z), a\_gain\_t(X,Y,Z).\\
-a\_gain\_t(X,Y,Z) &:- \neg b\_t'(X,Y,Z), a\_gain\_t(X,Y,Z).\\
a\_lost\_t'(X,Y,Z) &:- a\_lost\_t(X,Y,Z), \neg -a\_lost\_t(X,Y,Z).\\
a\_lost\_t'(X,Y,Z) &:- +a\_lost\_t(X,Y,Z).\\
a\_gain\_t'(X,Y,Z) &:- a\_gain\_t(X,Y,Z), \neg -a\_gain\_t(X,Y,Z).\\
a\_gain\_t'(X,Y,Z) &:- +a\_gain\_t(X,Y,Z).
\end{aligned}$$

$r'_{aux.S}$:

$$\begin{aligned}
-a\_lost\_s_1(X,Y,Z,W) &:- +s_1(X,Y,Z,W), a\_lost\_s_1(X,Y,Z,W).\\
-a\_lost\_s_1(X,Y,Z,W) &:- -s_1(X,Y,Z,W), a\_lost\_s_1(X,Y,Z,W).
\end{aligned}$$

$$\begin{aligned}
-a\_gain\_s_1(X,Y,Z,W) &:- +s_1(X,Y,Z,W), a\_gain\_s_1(X,Y,Z,W).\\
-a\_gain\_s_1(X,Y,Z,W) &:- -s_1(X,Y,Z,W), a\_gain\_s_1(X,Y,Z,W).\\
a\_c\_s_1{}'(X,W) &:- a\_c\_s_1(X,W), \neg -a\_c\_s_1(X,W).\\
a\_c\_s_1{}'(X,W) &:- +a\_c\_s_1(X,W).\\
a\_lost\_s_1{}'(X,Y,Z,W) &:- a\_lost\_s_1(X,Y,Z,W), \neg -a\_lost\_s_1(X,Y,Z,W).\\
a\_gain\_s_1{}'(X,Y,Z,W) &:- a\_gain\_s_1(X,Y,Z,W), \neg -a\_gain\_s_1(X,Y,Z,W).
\end{aligned}$$

where predicates $a\_c\_s_1{}'$(X,W), $a\_lost\_s_1{}'(X,Y,Z,W)$, and $a\_gain\_s_1{}'(X,Y,Z,W)$ correspond to the updated auxiliary relations $A^{c}_{S_1}{}'$, $A^{lost'}_{S_1}$, and $A^{gain'}_{S_1}$ respectively.

The algorithm outputs Datalog rules of $c$ for a constraint that predicate symbol $t$ appears in its body, $c_{aux}$ for constraints applied to the auxiliary relations, and $c_{pk}$ for constraints of a primary key.

$c$:

$$\bot() :- t'(X,Y,Z), Y \leq 0.$$

$c_{aux}$:

$$\begin{aligned}
\bot() &:- a\_lost\_t(X,Y,Z), a\_gain\_t(X,Y,Z).\\
\bot() &:- a\_gain\_t(X,Y,Z), \neg b\_t(X,Y,Z).
\end{aligned}$$

$c_{pk}$:

$$\begin{aligned}
\bot() &:- t'(X,Y,Z), t'(X,Y1,Z1), Y \neq Y1.\\
\bot() &:- t'(X,Y,Z), t'(X,Y1,Z1), Z \neq Z1.
\end{aligned}$$

□

**Algorithm 5.3** Deriving BX between Target-Side Database and View Instance of Target Schema

```
Input: a co-existence strategy $P$ satisfying the *consistency* of updates
Output: bidirectional transformations of source schema $BX_{t.trg}$
    $n \leftarrow$ a number of relation names of source schema
    for $i = 1$ to $n$ do
        $s_i, \vec{X_i} \leftarrow$ a relation name and attributes of source schema
                                        // corresponding to a relation $S_i(\vec{X_i})$
    end for
    $t, \vec{Y} \leftarrow$ a relation name and attributes of target schema
                                        // corresponding to a view instance $T(\vec{Y})$
    $C \leftarrow$ a set of constraints in $P$
    $f \leftarrow$ a set of Datalog rules to transform to $T(\vec{Y})$        //schema evolution
    for $i = 1$ to $n$ do
        $g^+_{s_i} \leftarrow$ a set of Datalog rules to transform $\Delta^+_{S_i}(\vec{X_i})$   // backward update sharing
        $g^-_{s_i} \leftarrow$ a set of Datalog rules to transform $\Delta^-_{S_i}(\vec{X_i})$   // backward update sharing
    end for

    // updated view instance $T'(\vec{Y})$
    // base relation $B_T(\vec{Y})$
    // auxiliary relations $A^c_{S_i}(\vec{Z_i})$, $A^{lost}_{S_i}(\vec{X_i})$, $A^{gain}_{S_i}(\vec{X_i})$, $A^{lost}_T(\vec{T})$, and $A^{gain}_T(\vec{T})$
    $trg\_db \leftarrow$ a set of base and auxiliary relation names defined by Algorithm 5.1

    // $get_{t.trg}$
    $r_T \leftarrow$ a set of Datalog rules to transform to a view instance $T(\vec{Y})$ as
        $T(\vec{Y}) = (B_T(\vec{Y}) \cap \neg A^{gain}_T(\vec{Y})) \cup A^{lost}_T(\vec{Y})$
    $get_{t.trg} \leftarrow r_T$

    // $put_{t.trg}$
    $r_S \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        $r_{S_i} \leftarrow$ a set of Datalog rules to transform to $S_i(\vec{X_i})$ as
            $S^{evo}_i(\vec{X_i}) = \pi_{attr(S_i)}(B_T(\vec{Y}) \bowtie A^c_{S_i}(\vec{Z_i}))$ and
            $S_i(\vec{X_i}) = (S^{evo}_i(\vec{X_i}) \cap \neg A^{gain}_{S_i}(\vec{X_i})) \cup A^{lost}_{S_i}(\vec{X_i})$
        $r_S \leftarrow r_S \cup r_{S_i}$
    end for
    $r_{\Delta T} \leftarrow$ A set of Datalog rules to transform to $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$as
        $\Delta^+_T(\vec{Y}) = T'(\vec{Y}) \cap \neg T(\vec{Y})$, and
        $\Delta^-_T(\vec{Y}) = \neg T'(\vec{Y}) \cap T(\vec{Y})$
    $r_{back} \leftarrow \emptyset$
    for $i = 1$ to $n$ do
        $r_{\Delta S_i} \leftarrow g^+_{s_i} \cup g^-_{s_i}$
        $r_i \leftarrow$ a set of Datalog rules to transform to $\Delta^+_{B_T}(\vec{Y})$, $\Delta^-_{B_T}(\vec{Y})$, $\Delta^+_{A^c_{S_i}}(\vec{Z_i})$, and
            $\Delta^-_{A^c_{S_i}}(\vec{Z_i})$ as
            $\Delta^+_{B_T}(\vec{Y}) = \pi_{attr(T)}(\Delta^+_{S_i}(\vec{X_i}) \bowtie \Delta^+_T(\vec{Y})) \cap \neg B_T(\vec{Y})$,
            $\Delta^-_{B_T}(\vec{Y}) = \pi_{attr(T)}(\Delta^-_{S_i}(\vec{X_i}) \bowtie \Delta^-_T(\vec{Y})) \cap B_T(\vec{Y})$,
            $\Delta^+_{A^c_{S_i}}(\vec{Z_i}) = \pi_{\vec{Z_i}}(\Delta^+_{S_i}(\vec{X_i})) \cap \neg A^c_{S_i}(\vec{Z_i})$,
            $\Delta^-_{S_i}{}'(\vec{X_i}) = \Delta^-_{S_i}(\vec{X_i}) \bowtie (\pi_{key}(\Delta^-_{S_i}(\vec{X_i})) \cap \neg\pi_{key}(\Delta^+_{S_i}(\vec{X_i})))$, and
            $\Delta^-_{A^c_{S_i}}(\vec{Z_i}) = \pi_{\vec{Z_i}}(\Delta^-_{S_i}{}'(\vec{X_i})) \cap A^c_{S_i}(\vec{Z_i})$
    $r_{back} \leftarrow r_{back} \cup r_{\Delta S_i} \cup r_i$
end for
$r'_{base} \leftarrow$ a set of Datalog rules to transform to $B_T{}'(\vec{Y})$ as
            $B_T{}'(\vec{Y}) = (B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y})$
$r'_{aux.T} \leftarrow$ a set of Datalog rules transform to $A^{lost}_T{}'(\vec{Y})$ and $A^{gain}_T{}'(\vec{Y})$ as
            $\Delta^+_{A^{lost}_T}(\vec{Y}) = T'(\vec{Y}) \cap \neg B_T{}'(\vec{Y}) \cap \neg A^{lost}_T(\vec{Y})$,
            $\Delta^-_{A^{lost}_T}(\vec{Y}) = \neg T'(\vec{Y}) \cap A^{lost}_T(\vec{Y})$,
            $\Delta^+_{A^{gain}_T}(\vec{Y}) = \neg T'(\vec{Y}) \cap B_T{}'(\vec{Y}) \cap \neg A^{gain}_T(\vec{Y})$,
            $\Delta^-_{A^{gain}_T}(\vec{Y}) = (T'(\vec{Y}) \cap A^{gain}_T(\vec{Y})) \cup (\neg B_T{}'(\vec{Y}) \cap A^{gain}_T(\vec{Y}))$,
            $A^{lost}_T{}'(\vec{Y}) = (A^{lost}_T(\vec{Y}) \cap \neg\Delta^-_{A^{lost}_T}(\vec{Y})) \cup \Delta^+_{A^{lost}_T}(\vec{Y})$, and
            $A^{gain}_T{}'(\vec{Y}) = (A^{gain}_T(\vec{Y}) \cap \neg\Delta^-_{A^{gain}_T}(\vec{Y})) \cup \Delta^+_{A^{gain}_T}(\vec{Y})$
$r'_{aux.S} \leftarrow \emptyset$
for $i = 1$ to $n$ do
    $r'_{aux.S_i} \leftarrow$ a set of Datalog rules transform to $A^c_{S_i}{}'(\vec{Z_i})$, $A^{lost}_{S_i}{}'(\vec{X_i})$, and $A^{gain}_{S_i}{}'(\vec{X_i})$
            as
            $\Delta^-_{A^{lost}_{S_i}}(\vec{X_i}) = (\Delta^+_{S_i}(\vec{X_i}) \cap A^{lost}_{S_i}(\vec{X_i})) \cup (\Delta^-_{S_i}(\vec{X_i}) \cap A^{lost}_{S_i}(\vec{X_i}))$,
            $\Delta^-_{A^{gain}_{S_i}}(\vec{X_i}) = (\Delta^+_{S_i}(\vec{X_i}) \cap A^{gain}_{S_i}(\vec{X_i})) \cup (\Delta^-_{S_i}(\vec{X_i}) \cap A^{gain}_{S_i})(\vec{X_i})$,
            $A^c_{S_i}{}'(\vec{Z_i}) = (A^c_{S_i}(\vec{Z_i}) \cap \neg\Delta^-_{A^c_{S_i}}(\vec{Z_i})) \cup \Delta^+_{A^c_{S_i}}(\vec{Z_i})$,
            $A^{lost}_{S_i}{}'(\vec{X_i}) = A^{lost}_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{A^{lost}_{S_i}}(\vec{X_i})$, and
            $A^{gain}_{S_i}{}'(\vec{X_i}) = A^{gain}_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{A^{gain}_{S_i}}(\vec{X_i})$
    $r'_{aux.S} \leftarrow r'_{aux.S} \cup r'_{aux.S_i}$
end for
$c \leftarrow$ a set of constrains in $C$ that a predicate with symbol $t$ is replaced to a
        corresponding predicate symbol of $T'(\vec{Y})$.
$c_{aux} \leftarrow$ a set of constraints of
        $A^{lost}_T(\vec{Y}) \cap A^{gain}_T(\vec{Y}) = \emptyset$
        $A^{gain}_T(\vec{Y}) \cap \neg B_T(\vec{Y}) = \emptyset$
$c_{pk} \leftarrow$ a set of constrains for primary key of relation name $t$ if it is specified
$put_{t.trg} \leftarrow r_S \cup r_{\Delta T} \cup r_{back} \cup r'_{base} \cup r'_{aux.T} \cup r'_{aux.S} \cup c \cup c_{aux} \cup c_{pk}$
$BX_{t.trg} \leftarrow \{(get_{t.trg}, put_{t.trg})\}$
return $BX_{t.trg}$
```

### Properties

Based on transformations of $get_{t.trg}$ and $put_{t.trg}$ derived by Algorithm 5.3, the following lemmas and a proposition are satisfied. Lemmas state that particular relationships of the auxiliary relations and the base relation are kept. Note that they are given as constraints in the algorithm.

**Lemma 5.4** (Disjointness of Auxiliary Relations of Target-Side Database)**.** *The auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ are disjoint.*

$$A_T^{lost}(\vec{Y}) \cap A_T^{gain}(\vec{Y}) = \emptyset$$

□

The proof is available in Appendix A.10.

**Lemma 5.5** (Inclusion of Auxiliary Relation of Target-Side Database)**.** *The auxiliary relation $A_T^{gain}(\vec{Y})$ is included in the base relation $B_T(\vec{Y})$.*

$$A_T^{gain}(\vec{Y}) \subseteq B_T(\vec{Y})$$

□

The proof is available in Appendix A.11.

The following proposition states the bidirectional transformation $BX_{t.trg}$ derived by Algorithm 5.3 is well-behaved.

**Proposition 5.6** (Well-Behaveness of BX between Target-Side Database and View Instance of Target Schema)**.** *Given a co-existence strategy between relations of source schema and a relation of target schema and turning each relation to view instance, the bidirectional transformation $BX_{t.trg}$ derived by Algorithm 5.3 is well-behaved by satisfying* GetPut *and* PutGet *laws.* □

The proof is available in Appendix A.12.

### 5.2.5 Correctness of the Algorithms

In the same manner with the correctness of the derivation algorithms in Chapter 4, we show schema evolution and backward update sharing of a co-existence strategy are realized by the derived BXs, and the soundness of the proposed algorithms to derive BXs.

### Realization of Schema Evolution

Schema evolution of a co-existence strategy is realized if a result of transformation by schema evolution and the derived BXs are the same. The following theorem states that the derived bidirectional transformations realize schema evolution.

**Theorem 5.7** (Realization of Schema Evolution by BXs between Target-Side Database and View Instances of Schemas)**.** *Given a co-existence strategy, the derived bidirectional transformations by Algorithm 5.2 and 5.3 realize its schema evolution.* □

Suppose a co-existence strategy between relations $S_i$ $(i \in [1, n])$ of source schema and a relation $T$ of target schema, and $BX_{t.src.i}$ for all $i$ and $BX_{t.trg}$ as derived bidirectional transformations from the co-existence strategy by Algorithm 5.2 and 5.3. Recall source schema instance S is union of relations $S_i$ for all $i$. Schema evolution of the co-existence strategy transforms S to $T$. On the other hand, for each $i$, $put_{t.src.i}$ of $BX_{t.src.i}$ transforms a view instance $S_i$ to a source-side database $\mathbf{D}_t$, and $get_{t.trg}$ of $BX_{t.trg}$ transforms $\mathbf{D}_t$ to a view instance $T$. If a result of these transformations from union of $S_i$ to $T$ via $\mathbf{D}_t$ is equivalent to a result of the transformation by schema evolution, schema evolution is realized by the derived bidirectional transformations. Further details and the formal proof are available in Appendix A.13.

### Realization of Backward Update Sharing

Backward update sharing of a co-existence strategy is realized if a result of transformation by backward update sharing and the derived BXs are the same. The following theorem states that the derived bidirectional transformations realize backward update sharing.

**Theorem 5.8** (Realization of Backward Update Sharing by BXs between Source-Side Database and View Instances of Schemas)**.** *Given a co-existence strategy, the derived bidirectional transformations by Algorithm 5.2 and 5.3 realize its backward update sharing.* □

Suppose the co-existence strategy and the derived bidirectional transformations mentioned above. Backward update sharing of the co-existence strategy transforms to sets of inserted and deleted tuples against a relation $T$ of target schema to sets of inserted and deleted tuples to the relation $S_i$ of source schema for each $i$ ($i \in [1, n]$). They update the relation $S_i$ to $S_i{}'$. On the other hand, $put_{t.trg}$ of $BX_{t.trg}$ transforms the updated $T'$ of target schema by sets of inserted and deleted tuples against it to the updated source-side database $\mathbf{D}_t{}'$ and $get_{t.src.i}$ of $BX_{t.src.i}$ for each $i$ transforms it to the updated view instance $S_i{}'$. If a result of these transformations is equivalent to a result of the transformation by backward update sharing, backward update sharing is realized by the derived bidirectional transformations. Further details and the formal proof are available in Appendix A.14.

### Soundness

Given a co-existence strategy, the derived BXs by Algorithm 5.2 and 5.3 are well-behaved and realize its schema evolution and backward update sharing. Thus the algorithms are sound.

## 5.3 Data Migration

This section explains how to migrate data between the source-side database and the target-side database.

### Data Migration from Source-Side Database to Target-Side Database

Data migration from the source-side database to the target-side database consists of two steps. Recall $BX_{src}$ and $BX_{trg}$ are bidirectional transformations between the source-side database and view instances of source schema and target schema. $BX_{t.src}$ and $BX_{t.trg}$ are bidirectional transformations between target-side database and view instances of source schema and target schema respectively.

Figure 5.3 shows steps of a data migration from the source-side database $\mathbf{D}_s$ to the target-side database $\mathbf{D}_t$. Figure 5.3 (a) depicts the first step. $\mathbf{D}_s$ is transformed to the view instances $S_i$ of source schema by $get_{src.i}$ of $BX_{src}$ for each $i$, and then these view instances are transformed to the target-side database one-by-one by $put_{t.src.i}$.

Eventually its state ends up $\mathbf{D}_t^{tmp}$. Figure 5.3 (b) depicts the second step to finally migrate data from the source-side database $\mathbf{D}_s$ to the target-side database $\mathbf{D}_t$. $\mathbf{D}_s$ is transformed to the view instance $T$ of target schema by $get_{trg}$ of $BX_{src}$, and then a pair of $\mathbf{D}_t^{tmp}$ and the view instance $T$ are transformed to the target-side database $\mathbf{D}_t$ by $put_{t.trg}$.

### Data Migration from Target-Side Database to Source-Side Database

Figure 5.4 shows steps of data migration from the target-side database $\mathbf{D}_t$ to the source-side database $\mathbf{D}_t$. Figure 5.4 (a) depicts the first step. $\mathbf{D}_t$ is transformed to the view instances $S_i$ of source schema by $get_{t.src.i}$ of $BX_{t.src}$ for each $i$, and then these view instances are transformed to the source-side database one-by-one by $put_{src.i}$. Eventually its state ends up $\mathbf{D}_s^{tmp}$. Figure 5.4 (b) depicts the second step to finally migrate data from the target-side database $\mathbf{D}_t$ to the source-side database $\mathbf{D}_s$. $\mathbf{D}_t$ is transformed to the view instance $T$ of target schema by $get_{t.trg}$ of $BX_{t.trg}$, and then a pair of $\mathbf{D}_s^{tmp}$ and the view instance $T$ are transformed to the target-side database $\mathbf{D}_s$ by $put_{trg}$.

The correctness of the data migration requires that view instances of schemas computed from a migrated database must be the same as view instances of schemas computed from an original database. The following proposition states such property.

**Proposition 5.9** (Data Migration)**.** *Suppose BXs derived by Algorithm 4.3, 4.2, 5.2, and 5.3 from a given co-existence strategy. View instances of source schema and target schema are equivalent when they are computed from the source-side database and the migrated target-side database based on the derived BXs. View instances of source schema and target schema are equivalent when they are computed from the target-side database and the migrated source-side database based on the derived BXs.* □

The proof is available in Appendix A.15.

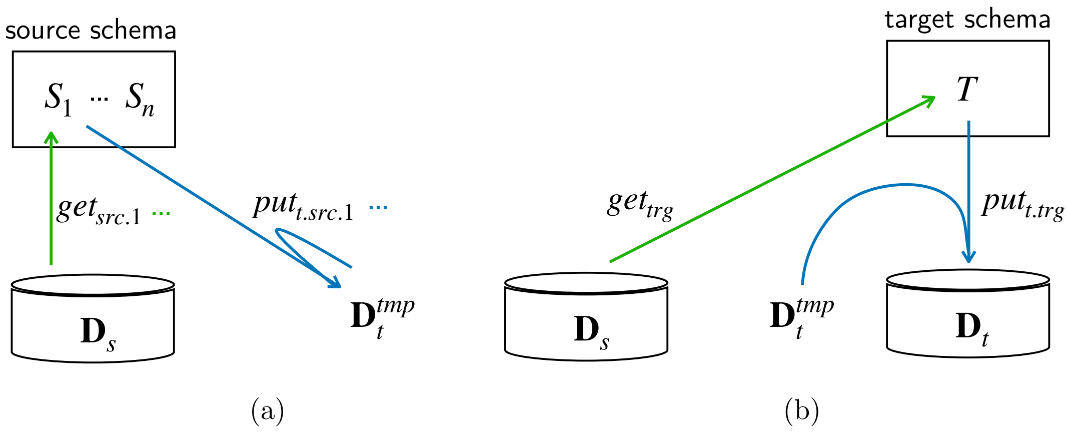


Figure 5.3: Data Migration from a source-side database to a target-side database.

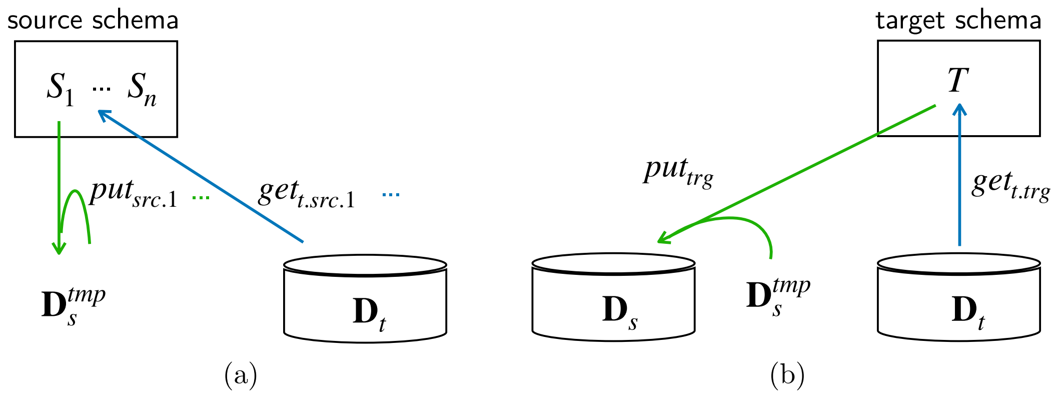


Figure 5.4: Data Migration from a target-side database to a source-side database.

## 5.4 Evaluation

This section shows experimental results and an evaluation of the proposed method.

### 5.4.1 Implementation

We have implemented a prototype in OCaml to verify the *consistency* of updates and realization of co-existence strategy on a target-side database in the same manner as Chapter 4. This prototype derives Datalog programs of bidirectional transformations by Algorithm 5.1, 5.2, and 5.3. All other components of the prototype are the same as the prototype explained in Chapter 4.

### 5.4.2 Experimental Result

To evaluate our approach, we conduct two experiments. The first experiment aims to investigate derived BXs from co-existence strategies described by the proposed DSL and performance of writing and reading on view instances of both source schema and target schema while sharing updates with each. In the second experiment, we evaluate the performance of the data migration between the source-side database and the target-side database. We utilize the same co-existence strategies in Table 4.1 of Chapter 4 as the benchmark to perform evaluations. The experiments are run on a Core i5 machine with 2 GHz and 16 GB memory and PostgreSQL 10.16.

#### Deriving BXs between Target-Side Database and View Instances of Schemas

Table 5.1 shows experimental results of derived BXs between the target-side database and view instances of source schema and target schema from co-existence strategies described by the proposed DSL. The structure of the table is the same as in Table 4.1. For example, the table shows that a co-existence strategy of SMO `DROP COLUMN` (a co-existence strategy No.1) is described as an 8 LOC Datalog program by the proposed DSL. Our method derives 75 LOC of BXs as Datalog programs, 1891 LOC of SQL programs, and five auxiliary relation names of the target-side database schema. The number of derived auxiliary relation names varies depending on the number of relation names of source schema. For example, the algorithm derives 5 auxiliary relation names from a co-existence strategy No.13 that specifies 3 ($= 3 \times 1$) auxiliary relation names from one relation of source schema and 2 auxiliary relation names from one relation of target schema. On the other hand, the algorithm derives 8 auxiliary relation names from a co-existence strategy No.16 that specifies 6 ($= 3 \times 2$) auxiliary relation names from two relations of source schema and 2 auxiliary relation names from one relation of target schema.

Table 5.1: Results of derived BXs between the target-side database and view instances of source schema and target schema.

| SMOs | | co-existence strategy | | | the proposed method | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Operator | # of aux. | schema evolution | backward update sharing | SMO | DSL [LOC] | # of aux. | BXs [LOC] | SQL [LOC] | No. |
| `DROP COLUMN` $(S_1 \to T)$ | 1 | projection | share all with $S_1$ | ✓ | 8 | 5 | 75 | 1891 | 1 |
| | | | share with $S_1$ if a condition is satisfied | | 8 | 5 | 75 | 1891 | 2 |
| | | | share only deletion with $S_1$ if a condition is satisfied | | 6 | 5 | 68 | 1390 | 3 |
| | | | not share with $S_1$ | | 3 | 5 | 59 | 1057 | 4 |
| `ADD COLUMN` $(S_1 \to T)$ | 1 | outer join (pk) | share all with $S_1$ | ✓ | 17 | 8 | 153 | 7452 | 5 |
| | | | not share with $S_1$ | | 9 | 8 | 128 | 4248 | 6 |
| `JOIN TABLE` $(S_1, S_2 \to T)$ | 0 | outer join (pk) | share all with $S_1$ and $S_2$ | ✓ | 15 | 8 | 154 | 8032 | 7 |
| | | | not share with $S_1$ and $S_2$ | | 9 | 8 | 131 | 5720 | 8 |
| | 1 | outer join (fk) | share all with $S_1$ and $S_2$ | ✓ | 12 | 8 | 147 | 6698 | 9 |
| | | | not share with $S_1$ and $S_2$ | | 5 | 8 | 131 | 4297 | 10 |
| | 2 | inner join (pk) | share all with $S_1$ and $S_2$ | ✓ | 11 | 8 | 144 | 4356 | 11 |
| | | | not share with $S_1$ and $S_2$ | | 7 | 8 | 123 | 2744 | 12 |
| | 3 | inner join (cond.) | share all with $S_1$ and $S_2$ | ✓ | 16 | 8 | 144 | 12582 | 13 |
| | | | not share with $S_1$ and $S_2$ | | 4 | 8 | 113 | 2358 | 14 |
| `SPLIT TABLE` $(S_1 \to T)$ | 1 | selection | share with $S_1$ without condition | ✓ | 7 | 5 | 73 | 1717 | 15 |
| | | | share with $S_1$ if selection condition is satisfied | | 7 | 5 | 73 | 1729 | 16 |
| | | | not share with $S_1$ | | 5 | 5 | 62 | 1141 | 17 |
| `MERGE TABLE` $(S_1, S_2 \to T)$ | 5 | union | share with $S_1$ and $S_2$ if conditions are satisfied | ✓ | 13 | 8 | 146 | 5162 | 18 |
| | | | share insertion with $S_1$ and deletion with $S_2$ | | 8 | 8 | 136 | 4428 | 19 |
| | | | not share with $S_1$ and $S_2$ | | 8 | 8 | 124 | 3468 | 20 |
| $(S_1, S_2 \to T)$ | - | Cartesian product | share all with $S_1$ and $S_2$ | | 18 | 8 | 144 | 9102 | 21 |
| $(S_1, S_2 \to T)$ | - | set difference | share with $S_1$ and not share with $S_2$ | | 6 | 8 | 124 | 3436 | 22 |

Figure 5.5 shows performances of writing and reading while updates are shared by schema evolution realized on the target-side database. Each graph shows relationships between the number of executed tuples and execution time of writing (insertion) to $S_1$ of source schema, reading its result as the view instances $S_1$ of source schema, and reading of the view instance $T$ of target schema as a result of update sharing by schema evolution of each co-existence strategy. In the same manner with Figure 4.3 of Chapter 4, Figure 5.5 (a), (b), (c), (d), (e) are results of the co-existence strategy No.1 for schema evolution by projection, No.13 for schema evolution by inner join with a given condition, No.15 for schema evolution by selection, No.18 for schema evolution by union, and No.22 for schema evolution by set difference respectively. In writing on $S_1$ of (b) and (e), a view instance $S_2$ of source schema has 10,000 tuples in advance.

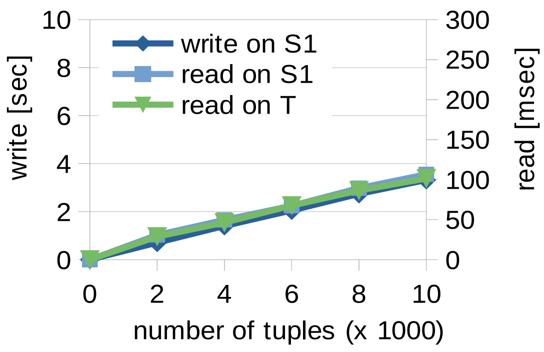
Results show that reading time is almost the same for view instances of source schema and target schema regardless of co-existence strategies except for reading on the view instance $T$ resulted in schema evolution of selection in (c). As discussed in Chapter 4, reading on the view instance $T$ is faster than $S_1$ because schema evolution by selection discards tuples and tuples of the view instance $T$ of target schema are less than tuples of view instance $S_1$ of source schema. The execution time of writing shows linear relationships between the number of executed tuples and executed time, even though execution time varies depending on strategies.

Figure 5.6 shows performances of writing and reading while updates are shared by backward update sharing realized on the target-side database. Each graph shows relationships between the number of executed tuples and execution time of writing (insertion) to $T$ of target schema, reading its result as the view instances $T$ of target schema, and reading of the view instance $S_1$ of source schema as a result of update sharing by backward update sharing of each co-existence strategy. Figure 5.6 (a) – (e) are results of backward update sharing in the co-existence strategies corresponding to (a) – (e) of Figure 5.5. In writing on $T$ of (b) and (e), the view instance $S_2$ of source schema already has 10,000 tuples in advance. Note that the scale of writing time in Figure 5.6 (b) is different from others.

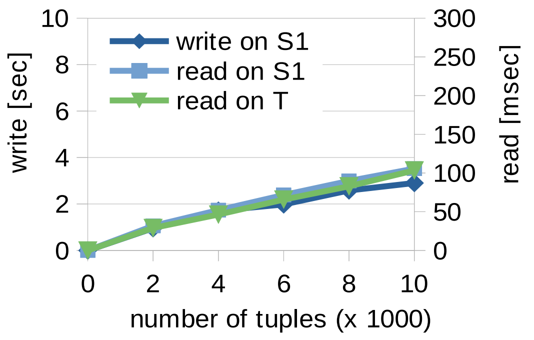
Results show that reading time is almost the same for view instances of source schema and target schema except for (c) of schema evolution by selection. Reading on the view instance $S_1$ is faster than $T$ because backward update sharing for schema evolution by selection discards tuples and tuples of the view instance $S_1$ of source schema are less than tuples of view instance $T$ of target schema.

The execution time of writing varies depending on strategies of backward update sharing. Results of backward update sharing for schema evolution by selection and set difference ((c) and (e) respectively) show linear relationships between the number of executed tuples and executed time. On the other hand, a non-linear increase of execution time against the increase of executed tuples are shown in the results of backward update sharing for schema evolution by projection, join, and union ((a), (b), and (d), respectively). Especially, backward update sharing for schema evolution by join shows a much longer execution time of writing.

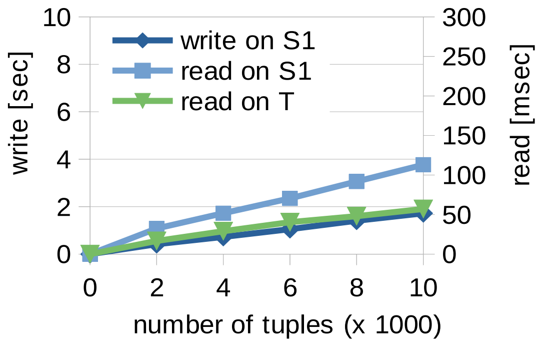
Furthermore, we investigate performance depending on the structure of the auxiliary relation for lost attributes. Figure 5.5 (f) shows performances of writing depending on the portion of projected away attributes by schema evolution. This experiment is performed based on the co-existence strategy No.1 in Table 5.1 for schema evolution by projection. For example, in the graph, a legend of "1/10 (10 LOC)" says about a co-existence strategy that consists of schema evolution to discard one attribute from 10 attributes and 10 LOC of backward update sharing. The result shows that the execution time of writing becomes longer when the portion of discarded attributes increases and LOC of backward update sharing increases.



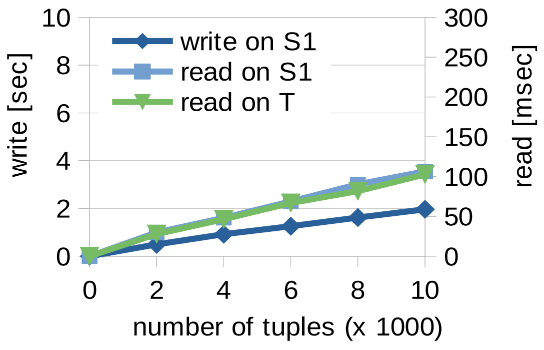
(a) projection



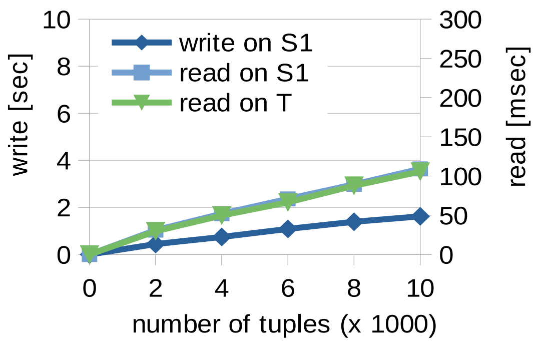
(b) join



(c) selection

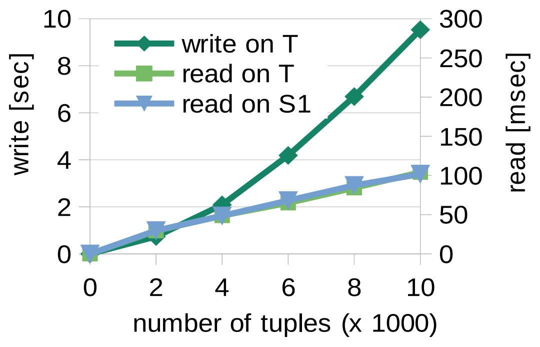


(d) union

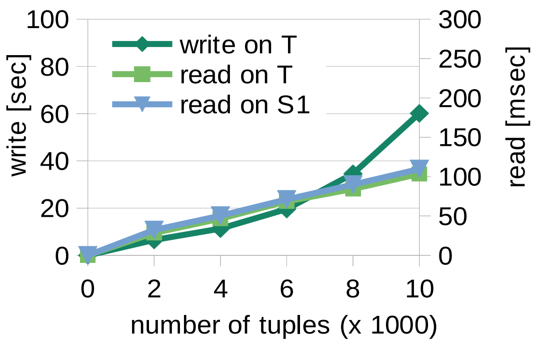


(e) set difference

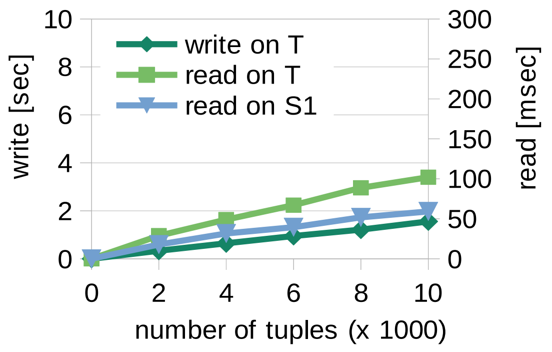
Figure 5.5: Performance of writing and reading with update sharing by schema evolution realized on the target-side database.



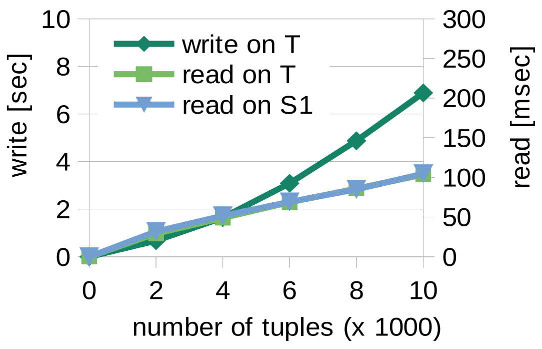
(a) projection



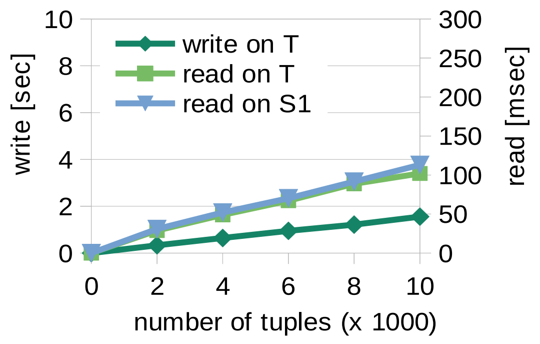
(b) join



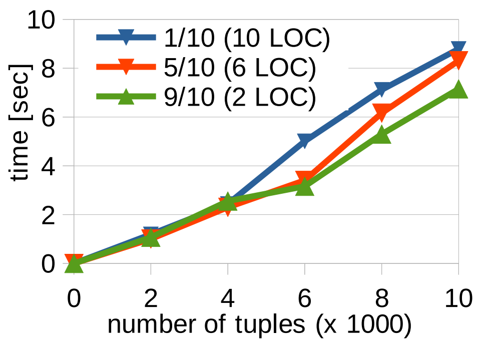
(c) selection



(d) union



(e) set difference



(f) auxiliary relation for lost attributes

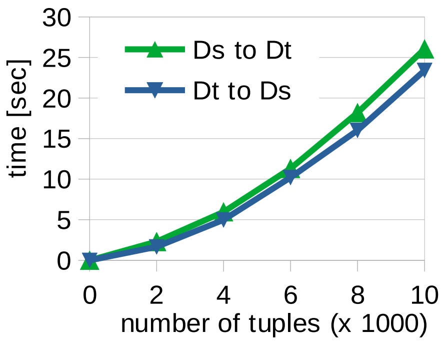
Figure 5.6: Performance of writing and reading with update sharing by backward update sharing realized on the target-side database.

### Performance of Data Migration

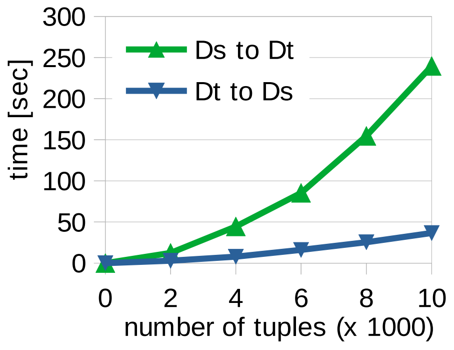
Figure 5.7 shows the performance of data migration between the source-side database instance $\mathbf{D}_s$ and the target-side database instance $\mathbf{D}_t$ by the method explained in Section 5.3. Experiments are performed based on co-existence strategies experimented in Figure 4.4 and Figure 5.6. Note that the scale of writing time in Figure 5.6 (b) is different from others.

Results show a non-linear relationship between the increase of migrated tuples and the increase of the execution time of data migration regardless of co-existence strategies and a direction of data migration. A data migration consists of reading (computing) view instances of schemas and writing (inserting) them to a database. However, results of data migrations show a longer execution time than a summation of their execution time shown in Figure 4.3, 4.4, 5.5, and 5.6. Since *put* of the second step in a data migration refers to the database resulted in the first step, e.g., $\mathbf{D}_t^{tmp}$ in Figure 5.3 and $\mathbf{D}_s^{tmp}$ in Figure 5.4, its execution time shows non-linear increase against increase of executed tuples and becomes longer than results of writing in Figure 4.3, 4.4, 5.5, and 5.6, in which writing (*put*) refers to an empty database [2].

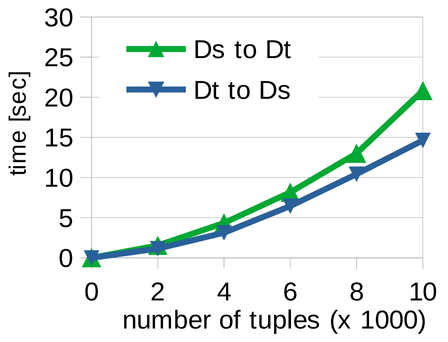
[2]Exceptions are (b) and (e) of Figure 4.3, 4.4, 5.5, and 5.6 in which the base relation and auxiliary relations have tuples in advance so that the view instance $S_2$ has 10,000 tuples. However, these base and auxiliary relations do not contain tuples to consist of tuples of the view instance $S_1$ while $\mathbf{D}_s^{tmp}$ or $\mathbf{D}_t^{tmp}$ has such tuples.



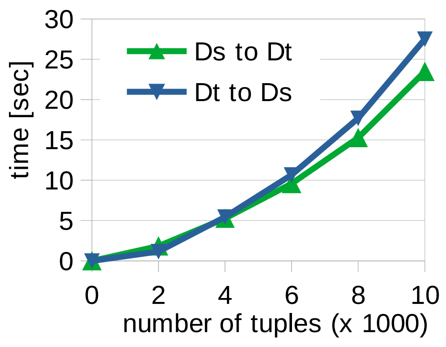
(a) projection



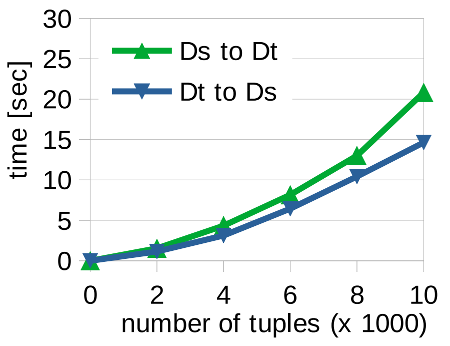
(b) join



(c) selection



(d) union



(e) set difference

Figure 5.7: Performance of data migration between the source-side database $\mathbf{D}_s$ and the target-side database $\mathbf{D}_t$.

### 5.4.3 Discussion

Table 5.1 shows that the benchmark co-existence strategies mentioned above are describable by the proposed DSL, and BXs to realize them are automatically derived. While the existing work predefines auxiliary relation names for SMO's strategies on a one-by-one basis, the proposed method realizes any co-existence strategies on a target-side database with systematically derived auxiliary relation names by Algorithm 5.1 from a given co-existence strategy.

Figure 5.5 and Figure 5.6 show that the sampled co-existence strategies from Table 5.1 are realized by sharing updates between view instances of schemas based on the derived BXs. The performance of reading is almost the same regardless of strategies. However, in some co-existence strategies, the performance of writing is much slower than reading and causes non-linear increases of execution time against the increase of executed tuples. Figure 5.6 (f) shows that the performance of writing is varied by rules of backward update sharing for schema evolution by projection even though the same type of auxiliary relation is utilized. However, its variation is smaller than the variation of the execution time of writing by various co-existence strategies. These results suggest that improving performance for the writing on view instances of target schema makes the proposed method more practical.

Figure 5.5 shows that the derived BXs realize data migrations between the source-side database and the target-side database. The result reveals that execution time of data migration is a non-linear increase against the number of executed tuples because the referred database by *put* (writing) is not empty. Again, it suggests that improving performance for the writing on view instances of target schema would contribute to more practical usefulness.

## 5.5 Related Work

Almost related work for the realization of co-existence strategy on the target-side database is common with related work introduced in Chapter 4. This section shows related work about complement when a transformation from a database to a view instance discards information.

A data warehouse is an integrated and time-varying collection of data from multiple source databases. To maintain a data warehouse efficiently by getting only updates of source databases without querying all data, a data warehouse needs to keep uncollected data for future updates. For example, dangling tuples of one out of two relations should be separately kept when a data warehouse collects data by join of two relations. Halevy [30] proposes a data warehouse based on a materialized view. Then, Laurent [45] proposes a methodology to derive minimal auxiliary views as complements to maintain uncollected data for efficient maintenance of data warehouse. Even though there is a gap between a data warehouse that partially collects data of each database and the view-based co-existence schemas that provides a part of the database to each view instance of schemas, a notion of auxiliary view for uncollected data is similar to a notion of auxiliary relation for unshared data.

Several researchers have investigated the complement in view update problem [8, 22, 47, 46]. In bidirectional transformation, Matsuda et al. [51] propose a methodology to derive *put* from *get* by deriving a view complement function accompanying a transformation of tree-structured data.

In this chapter, based on a relational database, we propose a methodology to derive the auxiliary relation as complement so that projected away attributes by schema evolution are kept, and the view instance of source schema can be computed from the base relation corresponding to the view instance of target schema and the auxiliary relation as a complement.

# 6
# Conclusion

This thesis presented a methodology to make the co-existence of relational database schemas programmable. First, we presented the DSL to make co-existence strategies describable. We also presented the *consistency* of updates and its verification method so that a described co-existence strategy does not cause additional updates when updates are shared between relations of schemas.

Second, we presented a method to realize co-existence strategies described by the DSL on the source-side database by the view-based approach. A source-side database schema is systematically derived from a given co-existence strategy. Then, BXs between the source-side database and view instances of source schema and target schema are derived. We implemented the proposed method and evaluated its usefulness. The results show that co-existence strategies defined in the existing work [34] and other strategies are describable by the proposed DSL, BXs to realize them are automatically derived, and the described co-existence strategies are realized based on the derived BXs by sharing updates between view instances of source schema and target schema.

Third, we presented a method to realize co-existence strategies described by the DSL on the target-side database instead of the source-side database. The target-side database schema is systematically derived from a given co-existence strategy. Then, BXs between the target-side database and view instances of source schema and target schema are derived. We implemented the proposed method and evaluated its usefulness. The results show that the described co-existence strategies are realized based on the derived BXs by sharing updates between view instances of source schema and target schema.

Fourth, we presented the method for data migration between the source-side database and the target-side database based on the derived BXs. We evaluated its usefulness by showing the execution time of data migration.

Possible future works are considered from each achievement of this thesis. The handling of more than two relations of target schema is considered about DSL. For example, duplicated tuples can appear in relations of target schema if a relation of source schema is evolved to two relations of target schema by selection with overlapped conditions. To specify backward update sharing for such duplicated tuples, a description of backward update sharing must refer to multiple relations of target schema. DSL would be designed by considering reasonable restrictions to handle multiple relations of target schema.

About deriving BXs to realize a co-existence strategy on the source-side or the target-side database, performance improvement of wiring will make the proposed methods more practical. In our proposed method, the derived BXs compute whole a view instance many times to compute sets of inserted and deleted tuples as differences between an updated view instance and a computed view instance from a database. If we utilize a technique of incrementalization in the database community, the performance of writing would be improved. Further experiments and analysis with large data sizes are also required for suitable design of performance improvement.

About data migration, performance improvement will make the proposed methods more practical. Since our method utilizes views to migrate data between databases, mapping to directly transform the source-side database to the target-side database and vice versa would contribute to the performance.

# Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. Foundations of databases, volume 8. Addison-Wesley Reading, 1995.

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

[3] José Andany, Michel Léonard, and Carole Palisser. Management of schema evolution in databases. In VLDB, pages 161–170, 1991.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. Feature-oriented software product lines. Springer, 2016.

[5] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Variational databases. In Proceedings of The 16th International Symposium on Database Programming Languages, pages 1–4, 2017.

[6] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1195–1206, 2008.

[7] Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and data sharing in evolving multi-tenant databases. In 2011 IEEE 27th International Conference on Data Engineering, pages 99–110. IEEE, 2011.

[8] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. ACM Trans. Database Syst., 6(4):557–575, December 1981.

[9] Vince Bárány, Balder Ten Cate, and Luc Segoufin. Guarded negation. Journal of the ACM (JACM), 62(3):1–26, 2015.

[10] Vince Bárány, Balder Ten Cate, and Martin Otto. Queries with guarded negation. Proceedings of the VLDB Endowment, 5(11):1328–1339, 2012.

[11] Philip A. Bernstein and Sergey Melnik. Model management 2.0: manipulating richer mappings. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 1–12, 2007.

[12] 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, pages 338–347, 2006.

[13] Michael L. Brodie and Jason. T. Liu. The power and limits of relational technology in the age of information ecosystems. On The Move Federated Conferences and Workshops, 2010.

[14] Loredana Caruccio, Giuseppe Polese, and Genoveffa Tortora. Synchronization of queries and views upon schema evolutions: A survey. ACM Transactions on Database Systems (TODS), 41(2):1–41, 2016.

[15] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). IEEE transactions on knowledge and data engineering, 1(1):146–166, 1989.

[16] Haitao Chen and Husheng Liao. A survey to view update problem. International Journal of Computer Theory and Engineering, 3(1):23, 2011.

[17] Jianjun Chen, Yu Chen, Zhibiao Chen, Ahmad Ghazal, Guoliang Li, Sihao Li, Weijie Ou, Yang Sun, Mingyi Zhang, and Minqi Zhou. Data management at huawei: Recent accomplishments and future challenges. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 13–24. IEEE, 2019.

[18] Carlo A. Curino, Hyun J. Moon, Alin Deutsch, and Carlo Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. Proceedings of the VLDB Endowment, 4(2):117–128, 2010.

[19] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. Proceedings of the VLDB Endowment, 1(1):761–772, 2008.

[20] Carlo A. Curino, Letizia Tanca, Hyun J. Moon, and Carlo Zaniolo. Schema evolution in wikipedia: toward a web information system benchmark. In In International Conference on Enterprise Information Systems (ICEIS). Citeseer, 2008.

[21] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on nosql stores. ACM Computing Surveys (CSUR), 51(2):1–43, 2018.

[22] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. ACM Transactions on Database Systems (TODS), 7(3):381–416, 1982.

[23] DBbeploy. http://dbdeploy.com/, 2021.

[24] Eladio Domínguez, Jorge Lloret, Ángel L Rubio, and María A Zapata. Medea: A database evolution architecture with traceability. Data & Knowledge Engineering, 65(3):419–441, 2008.

[25] Sebastian Fischer, Zhengjiang Hu, and Hugo Pacheco. "putback" is the essence of bidirectional programming. Technical report, GRACE Center, National Institute of infomatics, 2012.

[26] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. A clear picture of lens laws. In International Conference on Mathematics of Program Construction, pages 215–223. Springer, 2015.

[27] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. Journal of Systems and Software, 123:176–189, 2017.

[28] Flayway. https://flywaydb.org/, 2021.

[29] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic

approach to the view update problem. ACM SIGPLAN Notices, 40(1):233–246, 2005.

[30] Alon Y. Halevy. Answering queries using views: A survey. The VLDB Journal, 10(4):270–294, 2001.

[31] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: Data management infrastructure for semantic web applications. In Proceedings of the 12th International Conference on World Wide Web, WWW '03, page 556–567, New York, NY, USA, 2003. Association for Computing Machinery.

[32] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation for large-scale semantic data sharing. The VLDB Journal, 14(1):68–83, 2005.

[33] Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. Codel–a relationally complete language for database evolution. In East European Conference on Advances in Databases and Information Systems, pages 63–76. Springer, 2015.

[34] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, pages 1101–1116, 2017.

[35] Kai Herrmann, Hannes Voigt, Torben Bach Pedersen, and Wolfgang Lehner. Multi-schema-version data management: data independence in the twenty-first century. The VLDB Journal, 27(4):547–571, 2018.

[36] Philip Howard. Data migration report, 2011.

[37] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. Validity checking of putback transformations in bidirectional programming. In International Symposium on Formal Methods, pages 1–15. Springer, 2014.

[38] Richard Hull. Relative information capacity of simple relational database schemata. SIAM Journal on Computing, 15(3):856–886, 1986.

[39] Jez Humble and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.

[40] Zachary G. Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. Orchestra: Rapid, collaborative sharing of dynamic data. In CIDR, pages 107–118, 2005.

[41] Grigoris Karvounarakis, Todd J. Green, Zachary G. Ives, and Val Tannen. Collaborative data sharing via update exchange and provenance. ACM Transactions on Database Systems (TODS), 38(3):1–42, 2013.

[42] 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, pages 154–163, 1985.

[43] Arthur M. Keller. Choosing a view update translator by dialog at view definition time. In VLDB. Citeseer, 1986.

[44] Hsiang-Shang Ko and Zhenjiang Hu. An axiomatic basis for bidirectional programming. Proc. ACM Program. Lang., 2(POPL), December 2017.

[45] Dominique Laurent, Jens Lechtenbörger, Nicolas Spyratos, and Gottfried Vossen. Monotonic complements for independent data warehouses. The VLDB Journal, 10(4):295–315, December 2001.

[46] Jens Lechtenbörger. The impact of the constant complement approach towards view updating. In Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 49–55, 2003.

[47] Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. ACM Transactions on Database Systems (TODS), 28(2):175–208, 2003.

[48] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. ACM Transactions on Database Systems (TODS), 25(1):83–127, 2000.

[49] Liquibase. https://www.liquibase.org/, 2021.

[50] Yoshifumi Masunaga. A relational database view update translation mechanism. In Proceedings of the 10th International Conference on Very Large Data Bases, pages 309–320. Morgan Kaufmann Publishers Inc., 1984.

[51] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. SIGPLAN Not., 42(9):47–58, October 2007.

[52] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. ACM computing surveys (CSUR), 37(4):316–344, 2005.

[53] MyBatis migration. https://mybatis.org/migrations/, 2019.

[54] Renée J Miller, Yannis E Ioannidis, and Raghu Ramakrishnan. The use of information capacity in schema integration and translation. In VLDB, volume 93, pages 120–133, 1993.

[55] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. Proceedings of the VLDB Endowment, 1(1):882–895, 2008.

[56] Wee Siong Ng, Beng Chin Ooi, K-L Tan, and Aoying Zhou. Peerdb: A p2p-based system for distributed data sharing. In Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405), pages 633–644. IEEE, 2003.

[57] Hugo José Pereira Pacheco. Bidirectional Data Transformation by Calculation. PhD thesis, University of Minho, 2012.

[58] John F. Roddick. Schema evolution in database systems - an annotated bibliography. SIGMOD record, 21(4):35–40, 1992.

[59] John F. Roddick. A survey of schema versioning issues for database systems. Information and Software Technology, 37(7):383–393, 1995.

[60] Robert E. Schuler and Carl Kessleman. A high-level user-oriented framework for database evolution. In Proceedings of the 31st International Conference on Scientific and Statistical Database Management, pages 157–168, 2019.

[61] Wu Shengqi, Zhang Shidong, and Kong Lanju. Schema evolution via multi-version metadata in saas. Procedia Engineering, 29:1107–1112, 2012.

[62] Ioannis Skoulis, Panos Vassiliadis, and Apostolos Zarras. Open-source databases: Within, outside, or beyond lehman's laws of software evolution? In International Conference on Advanced Information Systems Engineering, pages 379–393. Springer, 2014.

[63] William Spoth, Bahareh Sadat Arab, Eric S Chan, Dieter Gawlick, Adel Ghoneimy, Boris Glavic, Beda Hammerschmidt, Oliver Kennedy, Seokki Lee, Zhen Hua Liu, et al. Adaptive schema databases. In CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings, 2017.

[64] Uta Störl, Meike Klettke, and Stefanie Scherzinger. Nosql schema evolution and data migration: State-of-the-art and opportunities. In EDBT, pages 655–658, 2020.

[65] Jumpei Tanaka, Vang-Dang Tran, and Zhenjiang Hu. Toward programmable strategy for co-existence of relational schemes. In Proceedings of Forth International Workshop on Software Foundations for Data Interoperability (SFDI 2020) collocated with VLDB 2020, pages 138–151. Springer, 2020.

[66] Jumpei Tanaka, Vang-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Toward co-existing database schemas based on bidirectional transformation. In Proceedings of Third Workshop on Software Foundations for Data Interoperability (SFDI2019+) collocated with the 11th Asia-Pasific Symposium on Internetware (Internetware 2019), 2019.

[67] Jumpei Tanaka, Vang-Dang Tran, Hiroyuki Kato, and Zhenjiang Hu. Programmable strategy for co-existence of schemas. IPSJ (PRO), 14(5):15–33, 2021.

[68] Igor Tatarinov, Zachary Ives, Jayant Madhavan, Alon Halevy, Dan Suciu, Nilesh Dalvi, Xin (Luna) Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The piazza peer data management project. SIGMOD Rec., 32(3):47–52, September 2003.

[69] James F. Terwilliger, Anthony Cleve, and Carlo A. Curino. How clean is your sandbox? In International Conference on Theory and Practice of Model Transformations, pages 1–23. Springer, 2012.

[70] James F. Terwilliger, Lois M. L. Delcambre, David Maier, Jeremy Steinhauer, and Scott Britell. Updatable and evolvable transforms for virtual databases. Proc, VLDB Endow., 3(1-2):309–319, 2010.

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

[72] D. Ullman Jeffrey. Principles of Database and Knowledge-Base Systems: Volume I. Computer Science Press, 1988.

[73] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. SIGPLAN Not., 35(6):26–36, June 2000.

[74] Panos Vassiliadis, Apostolos V. Zarras, and Ioannis Skoulis. How is life for a table in an evolving relational schema? birth, death and everything in between. In International Conference on Conceptual Modeling, pages 453–466. Springer, 2015.

[75] Bob Wall and Rafal Angryk. Minimal data sets vs. synchronized data copies in a schema and data versioning system. In Proceedings of the 4th workshop on Workshop for Ph. D. students in information & knowledge management, pages 67–74, 2011.

[76] Craig D. Weissman and Steve Bobrowski. The design of the force.com multitenant internet application development platform. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pages 889–896, 2009.

[77] Antoni Wolski and Kyösti Laiho. Rolling upgrades for continuous services. In International Service Availability Symposium, pages 175–189. Springer, 2004.

# A Proofs

## A.1 Proof of Lemma 4.1

We prove this lemma by induction. Attributes $\vec{Y}$ of relations are omitted in below.

Let $T(n)$ be a view instance of target schema at $n$ step. $T(n)$ is updated from the view instance $T(n-1)$ at $n-1$ step. In the same manner, let $B_{S_i}(n)$, $A_T^{lost}(n)$, and $A_T^{gain}(n)$ be base relations for each $i$ ($i \in [1,n]$) and auxiliary relations at $n$ step that $B_{S_i}(n)$, $A_T^{lost}(n)$, and $A_T^{gain}(n)$ are updated from $B_{S_i}(n-1)$, $A_T^{lost}(n-1)$, and $A_T^{gain}(n-1)$ at $n-1$ step respectively. Given a view instance $T(n)$, $put_{trg}$ transforms $B_{S_i}(n-1)$ for each $i$ ($i \in [1,n]$), $A_T^{lost}(n-1)$, and $A_T^{gain}(n-1)$ into updated $B_{S_i}(n)$, $A_T^{lost}(n)$, and $A_T^{gain}(n)$. Let $\mathbf{B_S}(n)$ be union of base relations $B_{S_i}(n)$ for all $i$. Let $T^{evo}(n)$ be $T^{evo}(n) = f(\mathbf{B_S}(n))$ where $f$ is a transformation of schema evolution.

(Base) As step 0, let $A_T^{lost}(0)$ and $A_T^{gain}(0)$ be empty set. It is obvious the following equation is satisfied.

$$A_T^{lost}(0) \cap A_T^{gain}(0) = \emptyset$$

(Induction) As an induction hypothesis, suppose $A_T^{lost}(n-1)$ and $A_T^{gain}(n-1)$ are disjoint and satisfy the following equation.

$$A_T^{lost}(n-1) \cap A_T^{gain}(n-1) = \emptyset$$

We show that $A_T^{lost}(n) \cap A_T^{gain}(n) = \emptyset$ is satisfied.

$$\begin{aligned}
A_T^{lost}(n) \cap A_T^{gain}(n) &= \{\text{Apply delta relations to } A_T^{lost}(n-1) \text{ and } A_T^{gain}(n-1)\} \\
&\quad \Big((A_T^{lost}(n-1) \cap \neg\Delta^-_{A_T^{lost}}) \cup \Delta^+_{A_T^{lost}}\Big) \\
&\quad \cap \Big((A_T^{gain}(n-1) \cap \neg\Delta^-_{A_T^{gain}}) \cup \Delta^+_{A_T^{gain}}\Big) \\
&= \{\text{Substitute transformations to } \Delta^+_{A_T^{lost}}, \Delta^-_{A_T^{lost}}, \Delta^+_{A_T^{gain}}, \text{ and } \Delta^-_{A_T^{gain}} \\
&\quad \text{defined in Algorithm 4.2 by replacing updated } T' \text{ to } T(n) \text{ and} \\
&\quad \text{updated } T'' \text{ to } T''(n).\} \\
&\quad \Big((A_T^{lost}(n-1) \cap \neg(A_T^{lost}(n-1) \cap \neg T(n))) \\
&\quad\quad \cup \ (T(n) \cap \neg T''(n) \cap \neg A_T^{lost}(n-1))\Big) \\
&\quad \cap \Big((A_T^{gain}(n-1) \cap \neg((A_T^{gain}(n-1) \cap T(n)) \\
&\quad\quad \cup ((A_T^{gain}(n-1) \cap \neg T''(n))) \\
&\quad\quad \cup (\neg T(n) \cap T''(n) \cap \neg A_T^{gain}(n-1))\Big) \\
&= \{\text{Deformation of a formula by set operation and applying} \\
&\quad A \cap \neg A = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

Thus $A_T^{lost} \cap A_T^{gain} = \emptyset$. □

## A.2 Proof of Lemma 4.2

Since $A_T^{gain}(\vec{Y}) \subseteq T^{evo}(\vec{Y})$ is equivalent to $A_T^{gain}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) = \emptyset$, we prove the following equation by induction.

$$A_T^{gain}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) = \emptyset$$

Attributes $\vec{Y}$ of relations are omitted in below.

Let $T(n)$ be a view instance of target schema at $n$ step. $T(n)$ is updated from the view instance $T(n-1)$ at $n-1$ step. In the same manner, let $B_{S_i}(n)$ and $A_T^{gain}(n)$ be base relations for each $i$ ($i \in [1, n]$) and auxiliary relation at $n$ step that $B_{S_i}(n)$ and $A_T^{gain}(n)$ are updated from $B_{S_i}(n-1)$ and $A_T^{gain}(n-1)$ at $n-1$ step respectively. Given a view instance $T(n)$, $put_{trg}$ transforms an auxiliary relation $A_T^{gain}(n-1)$ and base relations $B_{S_i}(n-1)$ for each $i$ ($i \in [1, n]$) into updated $A_T^{gain}(n)$ and $B_{S_i}(n)$. Let $\mathbf{B_S}(n)$ be union of base relations $B_{S_i}(n)$ for all $i$ and $T^{evo}(n)$ be $T^{evo}(n) = f(\mathbf{B_S}(n))$ where $f$ is a transformation of schema evolution. Since Algorithm 4.2 defines $T''$ as a result of schema evolution $f$ from the updated base relations $B_{S_i}{}'$ for all $i$ and $B_{S_i}{}'$ is expressed as $B_{S_i}(n)$, $T''$ is equivalent to $T^{evo}(n)$.

(Base) As step 0, let $A_T^{gain}(0)$ and $B_{S_i}(0)$ for all $i$ be empty set. Then $\mathbf{B_S}(0)$ is empty set. $T^{evo}(0)$ is also empty set because a result of transformation $f$ from empty set $\mathbf{B_S}(0)$ is empty set. Now, it is obvious the following equation is satisfied.

$$A_T^{gain}(0) \cap \neg T^{evo}(0) = \emptyset$$

(Induction) As an induction hypothesis, suppose the following equation.

$$A_T^{gain}(n-1) \cap \neg T^{evo}(n-1) = \emptyset.$$

We show that $A_T^{gain}(n) \cap \neg T^{evo}(n) = \emptyset$ is satisfied.

$$\begin{aligned} A_T^{gain}(n) \cap \neg T^{evo}(n) = {} & \{\text{Apply delta relations to } A_T^{gain}(n-1)\} \\ & ((A_T^{gain}(n-1) \cap \neg \Delta^-_{A_T^{gain}}) \cup \Delta^+_{A_T^{gain}}) \cap \neg T^{evo}(n) \end{aligned}$$

$$
\begin{aligned}
&= \{\text{Substitute transformations to } \Delta^{+}_{A_T^{gain}}, \text{ and } \Delta^{-}_{A_T^{gain}} \text{defined in} \\
&\quad \text{Algorithm 4.2 by replacing updated } T' \text{ to } T(n) \text{ and } T'' \text{ to} \\
&\quad T^{evo}(n).\} \\
&\quad \Big((A_T^{gain}(n-1) \cap \\
&\quad \neg((A_T^{gain}(n-1) \cap T(n)) \cup ((A_T^{gain}(n-1) \cap \neg T^{evo}(n))\big) \\
&\quad \cup (\neg T(n) \cap T^{evo}(n) \cap \neg A_T^{gain}(n-1))\Big) \cap \neg T^{evo}(n) \\
&= \{\text{Deformation of a formula by set operation and applying} \\
&\quad A \cap \neg A = \emptyset\} \\
&\quad \emptyset
\end{aligned}
$$

Thus $A_T^{gain} \cap \neg T^{evo} = \emptyset$. □

## A.3 Proof of Proposition 4.3

First, we prove the following GetPutis satisfied.

$$put_{trg}(\mathbf{D}_s, get_{trg}(\mathbf{D}_s)) = \mathbf{D}_s$$

where the database $\mathbf{D}_s$ is union of the base relation $B_{S_i}$ for all $i$ ($i \in [1, n]$) and auxiliary relations $A_T^{lost}$ and $A_T^{gain}$. We prove GetPut by showing a result of $put_{trg}$ after $get_{trg}$ does not update the base relations and the auxiliary relations of the database. Since a result of $get_{trg}$ is not updated in the proof of GetPut, let $T'(\vec{Y})$ be $T(\vec{Y})$ as a result of schema evolution. The following shows that $\Delta_T^+(\vec{Y})$ and $\Delta_T^-(\vec{Y})$ become empty set when $T'(\vec{Y}) = T(\vec{Y})$ is substituted into transformations to $\Delta_T^+(\vec{Y})$ and $\Delta_T^-(\vec{Y})$ defined in Algorithm 4.2. Note that attributes are omitted if all relations in a formula have the same attributes.

$$
\begin{aligned}
\Delta_T^+ &= \{\text{Definition in Algorithm 4.2}\} \\
&\quad T' \cap \neg T \\
&= \{\text{Substitute } T' = T\}
\end{aligned}
$$

$$
\begin{aligned}
& T \cap \neg T \\
= & \{A \cap \neg A = \emptyset\} \\
& \emptyset \\
\Delta_T^- = & \{\text{Definition by Algorithm 4.2}\} \\
& \neg T' \cap T \\
= & \{\text{Substitute } T' = T\} \\
& \neg T \cap T \\
= & \{A \cap \neg A = \emptyset\} \\
& \emptyset
\end{aligned}
$$

When $\Delta_T^+(\vec{Y})$ and $\Delta_T^-(\vec{Y})$ are empty sets, backward update transformation does not transform anything to $\Delta_{S_i}^+$ and $\Delta_{S_i}^-$ as inserted and deleted tuples against relations $S_i$ of source schema for each $i$ ($i \in [1, n]$). Since rules of $r_{base}$ are transformations to $\Delta_{B_{S_i}}^+$ and $\Delta_{B_{S_i}}^-$ against the base relation $B_{S_i}$ based on backward update transformation, $\Delta_{B_{S_i}}^+$ and $\Delta_{B_{S_i}}^-$ are empty sets when $\Delta_T^+(\vec{Y})$ and $\Delta_T^-(\vec{Y})$ are empty set. Thus base relations $B_{S_i}$ are not updated.

When base relations are not updated, $T''(\vec{Y})$ is equivalent to $T^{evo}(\vec{Y})$ as a result of schema evolution from the non-updated base relations. Thus $T''(\vec{Y}) = T^{evo}(\vec{Y})$. The following shows that $\Delta_{A_T^{lost}}^+$, $\Delta_{A_T^{lost}}^-$, $\Delta_{A_T^{gain}}^+$, and $\Delta_{A_T^{gain}}^-$ are empty set when $T'(\vec{Y}) = T(\vec{Y})$ and $T''(\vec{Y}) = T^{evo}(\vec{Y})$ are substituted into transformations to them defined in the Algorithm 4.2.

$$
\begin{aligned}
\Delta_{A_T^{lost}}^+ = & \{\text{Definition in Algorithm 4.2}\} \\
& T' \cap \neg T'' \cap \neg A_T^{lost} \\
= & \{T' = T, T'' = T^{evo}\} \\
& T \cap \neg T^{evo} \cap \neg A_T^{lost} \\
= & \{T = (T^{evo} \cap \neg A_T^{gain}) \cup A_T^{lost} \text{ by Algorithm 4.2}\} \\
& ((T^{evo} \cap \neg A_T^{gain}) \cup A_T^{lost}) \cap \neg T^{evo} \cap \neg A_T^{lost} \\
= & \{\text{Deformation of a formula by set operation}\} \\
& (T^{evo} \cap \neg A_T^{gain} \cap \neg T^{evo} \cap \neg A_T^{lost}) \cup (A_T^{lost} \cap \neg T^{evo} \cap \neg A_T^{lost})
\end{aligned}
$$

$$
\begin{aligned}
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset \\
\Delta^{-}_{A_T^{lost}} &= \{\text{Definition in Algorithm 4.2}\} \\
&\quad A_T^{lost} \cap \neg T' \\
&= \{T' = T\} \\
&\quad A_T^{lost} \cap \neg T \\
&= \{T = (T^{evo} \cap \neg A_T^{gain}) \cup A_T^{lost} \text{ by Algorithm 4.2}\} \\
&\quad A_T^{lost} \cap \neg((T^{evo} \cap \neg A_T^{gain}) \cup A_T^{lost}) \\
&= \{\text{De Morgan's laws}\} \\
&\quad A_T^{lost} \cap (\neg(T^{evo} \cup A_T^{gain}) \cap \neg A_T^{lost}) \\
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset \\
\Delta^{+}_{A_T^{gain}} &= \{\text{Definition in Algorithm 4.2}\} \\
&\quad \neg T' \cap T'' \cap \neg A_T^{gain} \\
&= \{T' = T, T'' = T^{evo}\} \\
&\quad \neg T \cap T^{evo} \cap \neg A_T^{gain} \\
&= \{T = (T^{evo} \cap \neg A_T^{gain}) \cup A_T^{lost} \text{ by Algorithm 4.2}\} \\
&\quad \neg((T^{evo} \cap \neg A_T^{gain}) \cup A_T^{lost}) \cap T^{evo} \cap \neg A_T^{gain} \\
&= \{\text{Deformation of a formula by set operation}\} \\
&\quad (\neg T^{evo} \cap \neg A_T^{gain} \cap T^{evo} \cap \neg A_T^{gain}) \cup (A_T^{gain} \cap \neg A_T^{lost} \cap T^{evo} \cap \neg A_T^{gain}) \\
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset \\
\Delta^{-}_{A_T^{gain}} &= \{\text{Definition in Algorithm 4.2}\} \\
&\quad (A_T^{gain} \cap T') \cup (A_T^{gain} \cap \neg T'') \\
&= \{T' = T, T'' = T^{evo}\} \\
&\quad (A_T^{gain} \cap T) \cup (A_T^{gain} \cap \neg T^{evo}) \\
&= \{\text{Deformation of a formula by set operation}\}
\end{aligned}
$$

$$
\begin{aligned}
& (A_T^{gain} \cap A_T^{lost}) \cup (A_T^{gain} \cap \neg T^{evo}) \\
= & \{\text{Lemma 4.1 and Lemma 4.2}\} \\
& \emptyset
\end{aligned}
$$

Thus auxiliary relations $A_T^{lost}$ and $A_T^{gain}$ are not updated. Since both base relations and auxiliary relations are not updated, GetPut is satisfied.

Second, we prove the following PutGet is satisfied.

$$get_{trg}(put_{trg}(\mathbf{D}_s, T')) = T'$$

A result of $put_{trg}(\mathbf{D}_s, T')$ is the updated database. The updated database is union of the updated base relations $S_i{}'$ for all $i$ ($i \in [1, n]$) and the updated auxiliary relations $A_T^{lost\prime}$ and $A_T^{gain\prime}$. PutGet is proved as follows.

$$
\begin{aligned}
get_{trg}(put_{trg}(\mathbf{D}_s, T')) = & \{\text{Definition of } get_{trg} \text{ in Algorithm 4.2 and } T''(\vec{Y}) \text{ is a result of} \\
& \text{schema evolution from the updated base relations}\} \\
& (T'' \cap \neg A_T^{gain\prime}) \cup A_T^{lost\prime} \\
= & \{\text{Substitute definition of } A_T^{lost\prime} \text{ and } A_T^{gain\prime} \text{ in Algorithm 4.2}\} \\
& (T'' \cap \neg((A_T^{gain} \cap \neg\Delta^-_{A_T^{gain}}) \cup \Delta^+_{A_T^{gain}})) \\
& \cup ((A_T^{lost} \cap \neg\Delta^-_{A_T^{lost}}) \cup \Delta^+_{A_T^{lost}}) \\
= & \{\text{Substitute definition of } \Delta^+_{A_T^{lost}}, \Delta^-_{A_T^{lost}}, \Delta^+_{A_T^{gain}} \text{ and } \Delta^-_{A_T^{gain}} \\
& \text{in Algorithm 4.2}\} \\
& (T'' \cap \neg((A_T^{gain} \cap \neg((A_T^{gain} \cap T') \cup (A_T^{gain} \cap \neg T'')))\cup \\
& (\neg T' \cap T'' \cap \neg A_T^{gain}))) \cup ((A_T^{lost} \cap \neg(A_T^{lost} \cap \neg T'))\cup \\
& (T' \cap \neg T'' \cap \neg A_T^{lost})) \\
= & \{\text{Deformation of a formula by set operation}\} \\
& T' \cap (T'' \cup A_T^{lost} \cup \neg T'') \cap (T'' \cup A_T^{lost} \cup \neg A_T^{lost}) \\
= & \{A \cap (B \cup \neg B) = A\} \\
& T'
\end{aligned}
$$

Thus PUTGET is satisfied. □

## A.4 Proof of Proposition 4.4

First, we prove the following GETPUT is satisfied.

$$put_{src.i}(\mathbf{D}_s, get_{src.i}(\mathbf{D}_s)) = \mathbf{D}_s$$

where the database $\mathbf{D}_s$ is union of the base relation $B_{S_i}(\vec{X_i})$ for all $i$ ($i \in [1, n]$) and the auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$. We prove GETPUT by showing a result of $put_{src.i}$ after $get_{src.i}$ does not update the base relations and the auxiliary relations of the database. $get_{src.i}$ defined in Algorithm 4.3 transforms the base relation $B_{S_i}(\vec{X_i})$ into the view instance $S_i(\vec{X_i})$ as $S_i(\vec{X_i}) = B_{S_i}(\vec{X_i})$. Since a result of $get_{src.i}$ is not updated in the proof of GETPUT, the update view instance $S_i{}'(\vec{X_i})$ appearing in $put_{src.i}$ is equivalent to $S_i(\vec{X_i})$ as a result of $get_{src.i}$. Thus, $S_i{}'(\vec{X_i}) = S_i(\vec{X_i}) = B_{S_i}(\vec{X_i})$. The following shows that $\Delta^+_{B_{S_i}}(\vec{X_i})$ and $\Delta^-_{B_{S_i}}(\vec{X_i})$ for each $i$ become empty set when $S_i{}'(\vec{X_i}) = B_{S_i}(\vec{X_i})$ is substituted into transformations to $\Delta^+_{B_{S_i}}(\vec{X_i})$ and $\Delta^-_{B_{S_i}}(\vec{X_i})$ defined in Algorithm 4.3. Note that attributes are omitted if all relations in a formula have the same attributes.

$$\begin{aligned}
\Delta^+_{B_{S_i}} &= \{\text{Definition in Algorithm 4.3}\} \\
&\quad S_i{}' \cap \neg B_{S_i} \\
&= \{S_i{}' = B_{S_i}\} \\
&\quad B_{S_i} \cap \neg B_{S_i} \\
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset \\
\Delta^-_{B_{S_i}} &= \{\text{Definition in Algorithm 4.3}\} \\
&\quad \neg S_i{}' \cap B_{S_i} \\
&= \{S_i{}' = B_{S_i}\} \\
&\quad \neg B_{S_i} \cap B_{S_i} \\
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

Algorithm 4.3 defines that $T^{evo}(\vec{Y})$ is a result of schema evolution from the base relations and $T^{evo\prime}(\vec{Y})$ is a result of schema evolution from the updated base relations. Since each base relation $B_{S_i}$ is not updated in the proof of GetPut, $T^{evo}(\vec{Y})$ is equivalent to $T^{evo\prime}(\vec{Y})$. Based on that, the following shows that $\Delta^+_{T^{evo}}(\vec{Y})$ and $\Delta^-_{T^{evo}}(\vec{Y})$ become empty set when $T^{evo}(\vec{Y}) = T^{evo\prime}(\vec{Y})$ is substituted into transformations to $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$ defined in Algorithm 4.3.

$$\begin{aligned}
\Delta^+_{T^{evo}} &= \{\text{Definition in Algorithm 4.3}\}\\
&\quad T^{evo\prime} \cap \neg T^{evo}\\
&= \{T^{evo} = T^{evo\prime} \text{ and } A \cap \neg A = \emptyset\}\\
&\quad \emptyset\\
\Delta^-_{T^{evo}} &= \{\text{Definition in Algorithm 4.3}\}\\
&\quad \neg T^{evo\prime} \cap T^{evo}\\
&= \{T^{evo} = T^{evo\prime} \text{ and } A \cap \neg A = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

By substituting $\Delta^+_{T^{evo}}(\vec{Y}) = \emptyset$ and $\Delta^-_{T^{evo}}(\vec{Y}) = \emptyset$ into a transformation to $\Delta^-_{A^{lost}_T}(\vec{Y})$ defined in the algorithm, it becomes empty set.

$$\begin{aligned}
\Delta^-_{A^{lost}_T} &= \{\text{Definition in Algorithm 4.3}\}\\
&\quad (\Delta^+_{T^{evo}} \cap A^{lost}_T) \cup (\Delta^-_{T^{evo}} \cap A^{lost}_T)\\
&= \{\Delta^+_{T^{evo}} = \emptyset \text{ and } \Delta^-_{T^{evo}} = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

The algorithm does not specify a transformation into the updated auxiliary relation $A^{gain\prime}_T$. Therefore, $get_{src.i}$ after $put_{src.i}$ does not update the base relations and the auxiliary relations of the database. Thus GetPut is satisfied.

Second, we prove the following PutGet is satisfied.

$$get_{src.i}(put_{src.i}(\mathbf{D}_s, S_i{}')) = S_i{}'$$

Algorithm 4.3 defines $put_{src.i}$ consisting of identity mapping from the view instance

$S_i(\vec{X_i})$ to the base relation $B_{S_i}(\vec{X_i})$ and a transformation from the view instance $S_i(\vec{X_i})$ to the auxiliary relation $A_T^{lost}$. The algorithm defines $get_{src.i}$ as identity mapping from the base relation $B_{S_i}(\vec{X_i})$ to the view instance $S_i(\vec{X_i})$. Therefore, $put_{src.i}$ transforms the updated view instance $S_i'(\vec{X_i})$ to the updated base relation $B_{S_i}'(\vec{X_i})$ and the auxiliary relation $A_T^{lost\prime}(\vec{Y})$ and then $get_{src.i}$ transforms the updated base relation $B_{S_i}'(\vec{X_i})$ into the updated view instance $S_i'(\vec{X_i})$. Thus PutGet is satisfied. □

## A.5 Proof of Theorem 4.5

Suppose a co-existence strategy strategy satisfying the *consistency* of updates between relations $S_i(\vec{X_i})$ $(i \in [1, n])$ of source schema and a relation $T(\vec{Y})$ of target schema. Recall that schema evolution is specified as a transformation $f$ from source schema instance $\mathbf{S}$ to a relation $T(\vec{Y})$.

$$T(\vec{Y}) = f(\mathbf{S}) \tag{A.1}$$

Source schema instance $\mathbf{S}$ is union of relations $S_i(\vec{X_i})$ $(i \in [1, n])$ of source schema.

$BX_{src.i}$ for all $i$ are derived by Algorithm 4.3 from the strategy. $BX_{trg}$ is derived by Algorithm 4.2 from the strategy. $put_{src.i}$ of $BX_{src.i}$ for each $i$ transforms the updated view instance $S_i'$ to the updated source-side database $\mathbf{D}_s'$. $get_{trg}$ of $BX_{trg}$ transforms the database $\mathbf{D}_s$ to the view instance $T$. The derived BXs realize schema evolution of a co-existence strategy if the following two cases are satisfied: as an initial state, a transformation from union of $S_i$ to $T$ via $\mathbf{D}_s$ by $BX_{src.i}$ for all $i$ and $BX_{trg}$ is equivalent to a result of the transformation by schema evolution, and as forward update sharing, a transformation of updates against $S_i$ to updates against $T$ by $BX_{src.i}$ and $BX_{trg}$ is equivalent to a result of update sharing by schema evolution.

First, it is proved that a result of schema evolution is equivalent to a result of $get_{trg}$ of $BX_{trg}$ after $put_{src.i}$ of $BX_{src.i}$ for all $i$ from a pair of the view instance $S_i(\vec{X_i})$ and the database when the base relations $B_{S_i}(\vec{X_i})$ for all $i$ and the auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ are empty set as the initial state. Algorithm 4.3 defines $put_{src.i}$ that consists of a transformation from the updated view instance $S_i'(\vec{X_i})$ to the update base relation $B_{S_i}'(\vec{X_i})$ as identity mapping and a transformation from the updated view instance $S_i'(\vec{X_i})$ to the updated auxiliary relation $A_T^{lost\prime}(\vec{Y})$ by applying a set of deleted tuples $\Delta^-_{A_T^{lost}}(\vec{Y})$ to the original auxiliary relation $A_T^{lost}(\vec{Y})$. Therefore, when the view instance is $S_i(\vec{X_i})$ and the base relations and the auxiliary relations are empty set, $put_{src.i}$ transforms $S_i(\vec{X_i})$ into the base relation $B_{S_i}(\vec{X_i})$ as $S_i(\vec{X_i}) = B_{S_i}(\vec{X_i})$ and the auxiliary relations as empty set.

Algorithm 4.2 defines $get_{trg}$ that transforms the base relations and the auxiliary relations of the database into the view instance $T(\vec{Y})$ as follows:

$$T(\vec{Y}) = (T^{evo}(\vec{Y}) \cap \neg A_T^{gain}(\vec{Y})) \cup A_T^{lost}(\vec{Y}) \tag{A.2}$$

where $T^{evo}(\vec{Y})$ is defined by the algorithm as a result of the transformation $f$ from union of the base relations $B_{S_i}$ $(i \in [1, n])$. Now $S_i(\vec{X_i}) = B_{S_i}(\vec{X_i})$ for all $i$, and $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ are empty sets. Therefore, left-hand side of equation (A.2) is equivalent to left-hand side of equation (A.1) as $f(\mathbf{S})$. Thus, a result of schema evolution is equivalent to a result of $get_{trg}$ after $put_{src.i}$ for all $i$ from a pair of the view instance $S_i(\vec{X_i})$ and the database when the base relations and the auxiliary relations of the database are empty set.

Second, it is proved that a transformation of updates against $S_i$ to updates against $T$ by $get_{trg}$ of $BX_{trg}$ after $put_{src.i}$ of $BX_{src.i}$ is equivalent to a result of update sharing by schema evolution. $get_{trg}$ defined by equation (A.2) expresses that the view instance $T(\vec{Y})$ is computed by deleting tuples in the auxiliary relation $A_T^{gain}(\vec{Y})$ and adding tuples in the auxiliary relation $A_T^{lost}(\vec{Y})$ against $T^{evo}(\vec{Y})$. $T^{evo}(\vec{Y})$ is a result of schema evolution $f$ from union of the base relations $B_{S_i}$ $(i \in [1, n])$. Let $\mathbf{B_S}'$ be union of base relations that one of them is updated to $B_{S_i}'(\vec{X_i})$ by $put_{src.i}$ from the updated view instance $S_i'(\vec{X_i})$. Algorithm 4.3 defines the updated $T^{evo\prime}(\vec{Y})$ as a result of the transformation $f$ of schema evolution from $\mathbf{B_S}'$. When the base relations and auxiliary relations are updated to $B_{S_i}'(\vec{X_i})$ and $A_T^{lost\prime}(\vec{Y})$ by $put_{src.i}$ from the updated $S_i'(\vec{X_i})$, the updated view instance $T'(\vec{Y})$ is computed by $get_{trg}$ as follows:

$$T'(\vec{Y}) = (T^{evo\prime}(\vec{Y}) \cap \neg A_T^{gain}(\vec{Y})) \cup A_T^{lost\prime}(\vec{Y}) \tag{A.3}$$

If tuples that do not exist in $T^{evo}(\vec{Y})$ and newly appear in $T^{evo\prime}(\vec{Y})$ are not deleted by tuples of $A_T^{gain}(\vec{Y})$ and tuples that exist in $T^{evo}(\vec{Y})$ and newly disappear from $T^{evo\prime}(\vec{Y})$ are not added by tuples of $A_T^{lost\prime}(\vec{Y})$ as a result of right-hand side of equation (A.3), a transformation of updates against $S_i(\vec{X_i})$ to updates against $T(\vec{Y})$ by the BXs is equivalent to a result of update sharing by schema evolution.

Since $put_{src.i}$ does not change the auxiliary relation $A_T^{gain}(\vec{Y})$ from the updated view instance $S_i{}'(\vec{X_i})$ and $A_T^{gain}(\vec{Y}) \subseteq T^{evo}(\vec{Y})$ by Lemma 4.2, tuples that do not exist in $T^{evo}(\vec{Y})$ and newly appear in $T^{evo}{}'(\vec{Y})$ are not deleted as a result of right-hand side of equation (A.3).

On the other hand, $put_{src.i}$ transforms to the updated $A_T^{lost}{}'(\vec{Y})$ by deleting a set of tuples $\Delta^-_{A_T^{lost}}(\vec{Y})$ as follows:

$$\Delta^+_{T^{evo}}(\vec{Y}) = T^{evo}{}'(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \tag{A.4}$$
$$\Delta^-_{T^{evo}}(\vec{Y}) = \neg T^{evo}{}'(\vec{Y}) \cap T^{evo}(\vec{Y}) \tag{A.5}$$
$$\Delta^-_{A_T^{lost}}(\vec{Y}) = (\Delta^+_{T^{evo}}(\vec{Y}) \cap A_T^{lost}(\vec{Y})) \cup (\Delta^-_{T^{evo}}(\vec{Y}) \cap A_T^{lost}(\vec{Y})) \tag{A.6}$$
$$A_T^{lost}{}'(\vec{Y}) = A_T^{lost}(\vec{Y}) \cap \neg\Delta^-_{A_T^{lost}}(\vec{Y}) \tag{A.7}$$

Rule (A.4) express that $\Delta^+_{T^{evo}}(\vec{Y})$ is a set of newly appeared tuples in $T^{evo}{}'(\vec{Y})$ as a result of schema evolution $f$ from $\mathbf{B_S}'$ while they do not exist in $T^{evo}(\vec{Y})$ as a result of schema evolution from the original base relations. Rule (A.5) express that $\Delta^-_{T^{evo}}(\vec{Y})$ is a set of newly disappeared tuples from $T^{evo}{}'(\vec{Y})$ as a result of schema evolution $f$ from $\mathbf{B_S}'$ while they exist in $T^{evo}(\vec{Y})$ as a result of schema evolution from the original base relations. Rule (A.6) expresses that tuples in $\Delta^+_{T^{evo}}(\vec{Y})$ or $\Delta^-_{T^{evo}}(\vec{Y})$ are deleted from the auxiliary relation $A_T^{lost}(\vec{Y})$ if they exist in it. Based on rule (A.7), $A_T^{lost}{}'(\vec{Y})$ does not have tuples of $\Delta^-_{T^{evo}}(\vec{Y})$ that is a set of tuples newly disappeared from $T^{evo}{}'(\vec{Y})$. Thus, tuples that exist in $T^{evo}(\vec{Y})$ and newly disappear from $T^{evo}{}'(\vec{Y})$ are not added as a result of right-hand side of equation (A.3).

Therefore, schema evolution of a co-existence strategy is realized by the BXs derived by Algorithm 4.2 and 4.3. □

## A.6 Proof of Theorem 4.6

Suppose a co-existence strategy strategy satisfying the *consistency* of updates between relations $S_i(\vec{X_i})$ $(i \in [1, n])$ of source schema and a relation $T(\vec{Y})$ of target schema. Recall that backward update sharing is specified as transformations $g^+_{s_i}$ and $g^-_{s_i}$ for each $i$ ($i \in [1, n]$) from a pair of source schema instance S and delta relation $\Delta T(\vec{Y})$ to $\Delta^{+}_{S_i}(\vec{X}_i)$ and $\Delta^{-}_{S_i}(\vec{X}_i)$ as sets of inserted and deleted tuples against a relation $S_i$ of source schema.

$$\Delta^{+}_{S_i}(\vec{X}_i) = g^{+}_{s_i}(\mathrm{S}, \Delta T(\vec{Y}))$$
$$\Delta^{-}_{S_i}(\vec{X}_i) = g^{-}_{s_i}(\mathrm{S}, \Delta T(\vec{Y}))$$

where source schema instance S is union of all relations $S_i$ of source schema. Such $\Delta^{+}_{S_i}(\vec{X}_i)$ and $\Delta^{-}_{S_i}(\vec{X}_i)$ update the relation $S_i(\vec{X}_i)$ to $S_i{}'(\vec{X}_i)$.

$BX_{src.i}$ for all $i$ are derived by Algorithm 4.3 from the strategy. $BX_{trg}$ is derived by Algorithm 4.2 from the strategy. $put_{trg}$ of $BX_{trg}$ transforms $\Delta T(\vec{Y})$ as the delta relation of the view instance $T(\vec{Y})$ of target schema into the updated database $\mathbf{D}_S{}'$. The base relations $B_{S_i}(\vec{X}_i)$ for each $i$ of the database is updated to $B_{S_i}{}'(\vec{X}_i)$ as follows:

$$\Delta^{+}_{B_{S_i}}(\vec{X}_i) = g^{+}_{s_i}(\mathbf{B}_{\mathrm{S}}, \Delta T(\vec{Y}))$$
$$\Delta^{-}_{B_{S_i}}(\vec{X}_i) = g^{-}_{s_i}(\mathbf{B}_{\mathrm{S}}, \Delta T(\vec{Y}))$$
$$B_{S_i}{}'(\vec{X}_i) = (B_{S_i}(\vec{X}_i) \cap \neg\Delta^{-}_{B_{S_i}}(\vec{X}_i)) \cup \Delta^{+}_{B_{S_i}}(\vec{X}_i)$$

where $\mathbf{B}_{\mathrm{S}}$ is union of the base relations $B_{S_i}(\vec{X}_i)$ for all $i$.

The updated base relation $B_{S_i}{}'(\vec{X}_i)$ is transformed into the updated view instance $S_i{}'(\vec{X}_i)$ of source schema by $get_{src.i}$ as follows:

$$S_i{}'(\vec{X}_i) = B_{S_i}{}'(\vec{X}_i)$$

Therefore, a result of the transformation by backward update sharing is equivalent to a result of transformations of updates against the view instance $T(\vec{Y})$ of target schema to the updated view instance $S_i{}'(\vec{X}_i)$ of source schema by $put_{trg}$ and $get_{src.i}$. Thus, backward update sharing is realized by the derived BXs. □

## A.7 Proof of Lemma 5.1

We prove this lemma by induction. Attributes $\vec{X}_i$ of relations are omitted in the following.

Let $S_i(n)$ be a view instance of source schema at $n$ step. $S_i(n)$ is updated from the view instance $S_i(n-1)$ at $n-1$ step. Given the view instance $S_i(n)$, $put_{t.src.i}$ transforms auxiliary relations $A^{lost}_{S_i}(n-1)$ and $A^{gain}_{S_i}(n-1)$ into updated $A^{lost}_{S_i}(n)$ and $A^{gain}_{S_i}(n)$.

(Base) As step 0, let $A^{lost}_{S_i}(0)$ and $A^{gain}_{S_i}(0)$ be empty set. It is obvious the following equation is satisfied.

$$A^{lost}_{S_i}(0) \cap A^{gain}_{S_i}(0) = \emptyset$$

(Induction) As an induction hypothesis, suppose $A^{lost}_{S_i}(n-1)$ and $A^{gain}_{S_i}(n-1)$ are disjoint and satisfy the following equation.

$$A^{lost}_{S_i}(n-1) \cap A^{gain}_{S_i}(n-1) = \emptyset$$

We show that $A^{lost}_{S_i}(n) \cap A^{gain}_{S_i}(n) = \emptyset$ is satisfied.

$$\begin{aligned}
A^{lost}_{S_i}(n) \cap A^{gain}_{S_i}(n) &= \{\text{Apply delta relations to } A^{lost}_{S_i}(n-1) \text{ and } A^{gain}_{S_i}(n-1)\}\\
&\quad \Big((A^{lost}_{S_i}(n-1) \cap \neg\Delta^-_{A^{lost}_{S_i}}) \cup \Delta^+_{A^{lost}_{S_i}}\Big)\\
&\quad \cap \Big((A^{gain}_{S_i}(n-1) \cap \neg\Delta^-_{A^{gain}_{S_i}}) \cup \Delta^+_{A^{gain}_{S_i}}\Big)\\
&= \{\text{Substitute transformations to } \Delta^+_{A^{lost}_{S_i}}, \Delta^-_{A^{lost}_{S_i}}, \Delta^+_{A^{gain}_{S_i}}, \text{ and } \Delta^-_{A^{gain}_{S_i}}\\
&\quad \text{defined in Algorithm 5.2 by replacing updated } S_i{}' \text{ to } S_i(n) \text{ and}\\
&\quad \text{updated } S_i{}'' \text{ to } S_i{}''(n).\}\\
&\quad \Big((A^{lost}_{S_i}(n-1) \cap \neg(\neg S_i(n) \cap A^{lost}_{S_i}(n-1)))\\
&\quad\quad \cup\ (S_i(n) \cap \neg S_i{}''(n) \cap \neg A^{lost}_{S_i}(n-1))\Big)\\
&\quad \cap \Big((A^{gain}_{S_i}(n-1) \cap \neg((S_i(n) \cap A^{gain}_{S_i}(n-1))\\
&\quad\quad \cup (\neg S_i{}''(n) \cap A^{gain}_{S_i}(n-1)))\\
&\quad\quad \cup (\neg S_i(n) \cap S_i{}''(n) \cap \neg A^{gain}_{S_i}(n-1))\Big)\\
&= \{\text{Deformation of a formula by set operation and applying}
\end{aligned}$$

$$A \cap \neg A = \emptyset\}$$

$$\emptyset$$

Thus $A^{lost}_{S_i} \cap A^{gain}_{S_i} = \emptyset$. □

## A.8 Proof of Lemma 5.2

Since $A^{gain}_{S_i}(\vec{X_i}) \subseteq S^{evo}_i(\vec{X_i})$ is equivalent to $A^{gain}_{S_i}(\vec{X_i}) \cap \neg S^{evo}_i(\vec{X_i}) = \emptyset$, we prove the following equation by induction.

$$A^{gain}_{S_i}(\vec{X_i}) \cap \neg S^{evo}_i(\vec{X_i}) = \emptyset$$

Let $S_i(\vec{X_i})(n)$ be a view instance of source schema at $n$ step. $S_i(\vec{X_i})(n)$ is updated from the view instance $S_i(\vec{X_i})(n-1)$ at $n-1$ step. Let the base relation $B_T(\vec{Y})$, the auxiliary relations $A^{gain}_{S_i}(\vec{X_i})$ and $A^c_{S_i}(\vec{Z_i})$ be $B_T(\vec{Y})(n-1)$, $A^{gain}_{S_i}(\vec{X_i})(n-1)$, and $A^c_{S_i}(\vec{Z_i})(n-1)$ at $n-1$ step and $B_T{}'(\vec{Y})$, $A^{gain\prime}_{S_i}(\vec{X_i})$ and $A^c_{S_i}{}'(\vec{Z_i})$ be $B_T(\vec{Y})(n)$, $A^{gain}_{S_i}(\vec{X_i})(n)$, and $A^c_{S_i}(\vec{Z_i})(n)$ at $n$ step. Given the view instance $S_i(\vec{X_i})(n)$, $B_T(\vec{Y})(n-1)$, $A^{gain}_{S_i}(\vec{X_i})(n-1)$ and $A^c_{S_i}(\vec{Z_i})(n-1)$ are updated to $B_T(\vec{Y})(n)$, $A^{gain}_{S_i}(\vec{X_i})(n)$ and $A^c_{S_i}(\vec{Z_i})(n)$ respectively by $put_{t.src.i}$. By following the definition in Algorithm 5.2, $S^{evo}_i(\vec{X_i})(n)$ is transformed from $B_T(\vec{Y})(n)$ and $A^c_{S.i}(\vec{Z_i})(n))$ as follows:

$$S^{evo}_i(\vec{X_i})(n) = \pi_{attr(S_i)}(B_T(\vec{Y})(n) \bowtie A^c_{S.i}(\vec{Z_i})(n))$$

Also, by following the definition in Algorithm 5.2, $S_i''(\vec{X_i})(n)$ is transformed from $B_T(\vec{Y})(n)$ and $A^c_{S.i}(\vec{Z_i})(n))$ as follows:

$$S_i''(\vec{X_i})(n) = \pi_{attr(S_i)}(B_T(\vec{Y})(n) \bowtie A^c_{S.i}(\vec{Z_i})(n))$$

Thus

$$S^{evo}_i(\vec{X_i})(n) = S_i''(\vec{X_i})(n)$$

Attributes $\vec{X_i}$ of relations are omitted in below.

(Base) As step 0, let $B_T(\vec{Y})(0)$, $A^{gain}_{S_i}(\vec{X_i})(0)$ and $A^c_{S_i}(\vec{Z_i})(0)$ be empty set. $S^{evo}_i(\vec{X_i})(0)$ is also empty set because a result of transformation $f$ from empty set $B_T(\vec{Y})(0)$ and $A^c_{S_i}(\vec{Z_i})(0)$ is empty set. Now, it is obvious that the following equation is satisfied.

$$A^{gain}_{S_i}(0) \cap \neg S^{evo}_i(0) = \emptyset$$

(Induction) As an induction hypothesis, suppose the following equation.

$$A^{gain}_{S_i}(n-1) \cap \neg S^{evo}_i(n-1) = \emptyset$$

We show that $A^{gain}_{S_i}(n) \cap \neg S^{evo}_i(n) = \emptyset$ is satisfied.

$$\begin{aligned}
A^{gain}_{S_i}(n) \cap \neg S^{evo}_i(n) &= \{\text{Apply delta relations to } A^{gain}_{S_i}(n-1)\} \\
&\quad ((A^{gain}_{S_i}(n-1) \cap \neg \Delta^-_{A^{gain}_{S_i}}) \cup \Delta^+_{A^{gain}_{S_i}}) \cap \neg S^{evo}_i(n) \\
&= \{\text{Substitute transformations to } \Delta^+_{A^{gain}_{S_i}}, \text{ and } \Delta^-_{A^{gain}_{S_i}} \text{ defined by} \\
&\quad \text{Algorithm 5.2 by replacing updated } S_i{}' \text{ to } S_i(n) \text{ and } S_i{}'' \text{ to} \\
&\quad S^{evo}_i(n).\} \\
&\quad \Big((A^{gain}_{S_i}(n-1) \cap \neg((S_i(n) \cap A^{gain}_{S_i}(n-1)) \\
&\qquad \cup (\neg S^{evo}_i(n) \cap (A^{gain}_{S_i}(n-1))) \cup (\neg S_i(n) \cap S^{evo}_i(n) \cap \neg A^{gain}_{S_i})\Big) \\
&\qquad \cap \neg S^{evo}_i(n) \\
&= \{\text{Deformation of a formula by set operation and applying} \\
&\quad A \cap \neg A = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

Thus $A^{gain}_{S_i} \cap \neg S^{evo}_i = \emptyset$. □

## A.9 Proof of Proposition 5.3

First, we prove the following GetPut is satisfied.

$$put_{t.src.i}(\mathbf{D}_t, get_{t.src.i}(\mathbf{D}_t)) = \mathbf{D}_t$$

where the database $\mathbf{D}_t$ is union of the base relation $B_T$ and the auxiliary relations $A^c_{S_i}$, $A^{lost}_{S_i}$, $A^{gain}_{S_i}$ for all $i$ ($i \in [1, n]$), $A^{lost}_T$ and $A^{gain}_T$. We prove GetPut is satisfied by showing a result of $put_{t.src.i}$ after $get_{t.src.i}$ does not update base relations and auxiliary relations of the database.

Since a result of $get_{t.src.i}$ is not updated in the proof of GetPut, let $S_i'(\vec{X_i})$ be equivalent to $S(\vec{X_i})$ as a result of schema evolution. Then relations $T^{evo\prime}(\vec{Y})$ and $T^{evo}(\vec{Y})$ defined in Algorithm 5.2 are the same because both are results of schema evolution $f$ from non-updated view instances $S_i$ for all $i$ ($i \in [1, n]$). By substituting $T^{evo\prime}(\vec{Y}) = T^{evo}(\vec{Y})$ into transformations to $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^-_{B_T}(\vec{Y})$ defined in the algorithm, they become empty set as follows. We omit attributes if all relations in a formula have the same attributes in the following.

$$\begin{aligned}
\Delta^+_{B_T} &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad T^{evo\prime} \cap \neg T^{evo} \cap \neg B_T \\
&= \{T^{evo\prime} = T^{evo}\} \\
&\quad T^{evo} \cap \neg T^{evo} \cap \neg B_T \\
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset \\
\Delta^-_{B_T} &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad \neg T^{evo\prime} \cap T^{evo} \cap B_T \\
&= \{T^{evo\prime} = T^{evo}\} \\
&\quad \neg T^{evo} \cap T^{evo} \cap B_T \\
&= \{A \cap \neg A = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

Thus the base relation $B_t$ is not updated.

By substituting $\Delta^{+}_{B_T}(\vec{Y}) = \emptyset$ and $\Delta^{-}_{B_T}(\vec{Y}) = \emptyset$ into transformations to $\Delta^{+}_{A^c_{S_i}}(\vec{Z_i})$ and $\Delta^{-}_{A^c_{S_i}}(\vec{Z_i})$ defined in the algorithm, they become empty set as follows:

$$\begin{aligned}
\Delta^{+}_{A^c_{S_i}}(\vec{Z_i}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad \pi_{\vec{Z_i}}(\Delta^{+}_{S_i}(\vec{X_i}) \bowtie \Delta^{+}_{B_T}(\vec{Y})) \cap \neg A^c_{S_i}(\vec{Z_i}) \\
&= \{\Delta^{+}_{B_T}(\vec{Y}) = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

$$\begin{aligned}
\Delta^{-}_{A^c_{S_i}}(\vec{Z_i}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad \pi_{\vec{Z_i}}(\Delta^{-}_{S_i}(\vec{X_i}) \bowtie \Delta^{-}_{B_T}(\vec{Y})) \cap A^c_{S_i}(\vec{Z_i}) \\
&= \{\Delta^{-}_{B_T}(\vec{Y}) = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

Thus the auxiliary relation $A^c_{S_i}$ is not updated.

Since $B_T(\vec{Y})$ and $A^c_{S_i}(\vec{Z_i})$ are not updated, $B_T{}'(\vec{Y}) = B_T(\vec{Y})$ and $A^c_{S_i}{}'(\vec{Z_i}) = A^c_{S_i}(\vec{Z_i})$. Then $S_i{}''(\vec{X_i})$ defined in the algorithm is equivalent to $S^{evo}_i(\vec{X_i})$ as follows:

$$\begin{aligned}
S_i{}''(\vec{X_i}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad \pi_{attr(S_i)}(B_T{}'(\vec{Y}) \bowtie A^c_{S_i}{}'(\vec{Z_i})) \\
&= \{B_T{}'(\vec{Y}) = B_T(\vec{Y}) \text{ and } A^c_{S_i}{}'(\vec{Z_i}) = A^c_{S_i}(\vec{Z_i})\} \\
&\quad \pi_{attr(S_i)}(B_T(\vec{Y}) \bowtie A^c_{S_i}(\vec{Z_i})) \\
&= \{\text{Definition to transform to } S^{evo}_i(\vec{X_i}) \text{ in Algorithm 5.2}\} \\
&\quad S^{evo}_i(\vec{X_i})
\end{aligned}$$

By substituting $S_i{}'(\vec{X_i}) = S_i(\vec{X_i})$ and $S_i{}''(\vec{X_i}) = S^{evo}_i(\vec{X_i})$ into transformations to $\Delta^{+}_{A^{lost}_{S_i}}(\vec{X_i})$ and $\Delta^{-}_{A^{lost}_{S_i}}(\vec{X_i})$ defined in the algorithm, they become empty set as follows:

$$\begin{aligned}
\Delta^{+}_{A^{lost}_{S_i}} &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad S_i{}' \cap \neg S_i{}'' \cap \neg A^{lost}_{S_i}
\end{aligned}$$

$$\begin{aligned}
&= \{S_i{}' = S_i \text{ and } S_i{}'' = S_i^{evo}\}\\
&\quad S_i \cap \neg S_i{}^{evo} \cap \neg A_{S_i}^{lost}\\
&= \{\text{Substitute definition of } S_i \text{ in Algorithm 5.2}\}\\
&\quad ((S_i^{evo} \cap \neg A_{S_i}^{gain}) \cup A_{S_i}^{lost}) \cap \neg S_i{}^{evo} \cap \neg A_{S_i}^{lost}\\
&= \{\text{Distributive law}\}\\
&\quad (S_i^{evo} \cap \neg A_{S_i}^{gain} \cap \neg S_i{}^{evo} \cap \neg A_{S_i}^{lost}) \cup (A_{S_i}^{lost} \cap \neg S_i{}^{evo} \cap \neg A_{S_i}^{lost})\\
&= \{A \cap \neg A = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

$$\begin{aligned}
\Delta^-_{A_{S_i}^{lost}} &= \{\text{Definition in Algorithm 5.2}\}\\
&\quad \neg S_i{}' \cap A_{S_i}^{lost}\\
&= \{S_i{}' = S_i \text{ and substitute definition of } S_i \text{ in Algorithm 5.2}\}\\
&\quad \neg((S_i^{evo} \cap \neg A_{S_i}^{gain}) \cup A_{S_i}^{lost}) \cap A_{S_i}^{lost}\\
&= \{\text{De Morgan's law}\}\\
&\quad (\neg(S_i^{evo} \cap \neg A_{S_i}^{gain}) \cap \neg A_{S_i}^{lost}) \cap A_{S_i}^{lost}\\
&= \{A \cap \neg A = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

Thus, the auxiliary relation $A_{S_i}^{lost}$ is not updated.

By substituting $S_i{}'(\vec{X}_i) = S_i(\vec{X}_i)$ and $S_i{}''(\vec{X}_i) = S_i^{evo}(\vec{X}_i)$ into transformations to $\Delta^+_{A_{S_i}^{gain}}(\vec{X}_i)$ and $\Delta^-_{A_{S_i}^{gain}}(\vec{X}_i)$ defined in the algorithm, they become empty set as follows:

$$\begin{aligned}
\Delta^+_{A_{S_i}^{gain}} &= \{\text{Definition in Algorithm 5.2}\}\\
&\quad \neg S_i{}' \cap S_i{}'' \cap \neg A_{S_i}^{gain}\\
&= \{S_i{}' = S_i \text{ and } S_i{}'' = S_i^{evo}\}\\
&\quad \neg S_i \cap S_i{}^{evo} \cap \neg A_{S_i}^{gain}\\
&= \{\text{Substitute definition of } S_i \text{ in Algorithm 5.2}\}\\
&\quad \neg((S_i^{evo} \cap \neg A_{S_i}^{gain}) \cup A_{S_i}^{lost}) \cap S_i{}^{evo} \cap \neg A_{S_i}^{gain}\\
&= \{\text{De Morgan's law and Distributive law}\}
\end{aligned}$$

$$
\begin{aligned}
& (\neg S_i^{evo} \cap \neg A_{S_i}^{lost} \cap S_i{}^{evo} \cap \neg A_{S_i}^{gain}) \cup (A_{S_i}^{gain} \cap \neg A_{S_i}^{lost} \cap S_i{}^{evo} \cap \neg A_{S_i}^{gain}) \\
= & \{A \cap \neg A = \emptyset\} \\
& \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\Delta^-_{A_{S_i}^{gain}} = & \{\text{Definition in Algorithm 5.2}\} \\
& (S_i{}' \cap A_{S_i}^{gain}) \cup (\neg S_i{}'' \cap A_{S_i}^{gain}) \\
= & \{S_i{}' = S_i \text{ and } S_i{}'' = S_i^{evo}\} \\
& (S_i \cap A_{S_i}^{gain}) \cup (\neg S_i{}^{evo} \cap A_{S_i}^{gain}) \\
= & \{\text{Substitute definition of } S_i \text{ in Algorithm 5.2}\} \\
& ((S_i^{evo} \cap \neg A_{S_i}^{gain}) \cup A_{S_i}^{lost}) \cap A_{S_i}^{gain}) \cup (\neg S_i{}^{evo} \cap A_{S_i}^{gain}) \\
= & \{\text{Distributive law}\} \\
& (S_i^{evo} \cap \neg A_{S_i}^{gain} \cap A_{S_i}^{gain}) \cup (A_{S_i}^{lost} \cap A_{S_i}^{gain}) \cup (\neg S_i{}^{evo} \cap A_{S_i}^{gain}) \\
= & \{A \cap \neg A = \emptyset \text{ , Lemma 5.1 and Lemma 5.2 }\} \\
& \emptyset
\end{aligned}
$$

Thus the auxiliary relation $A_{S_i}^{gain}(\vec{X}_i)$ is not updated.

By substituting $\Delta^+_{B_T}(\vec{Y}) = \emptyset$ and $\Delta^-_{B_T}(\vec{Y}) = \emptyset$ into transformations to $\Delta^-_{A_T^{lost}}(\vec{Y})$ and $\Delta^-_{A_T^{gain}}(\vec{Y})$ defined in the algorithm, they become empty set as follows:

$$
\begin{aligned}
\Delta^-_{A_T^{lost}} = & \{\text{Definition in Algorithm 5.2}\} \\
& (\Delta^+_{B_T} \cap A_T^{lost}) \cup (\Delta^-_{B_T} \cap A_T^{lost}) \\
= & \{\Delta^+_{B_T} = \emptyset \text{ and } \Delta^-_{B_T} = \emptyset\} \\
& \emptyset \\
\Delta^-_{A_T^{gain}} = & \{\text{Definition in Algorithm 5.2}\} \\
& (\Delta^+_{B_T} \cap A_T^{gain}) \cup (\Delta^-_{B_T} \cap A_T^{gain}) \\
= & \{\Delta^+_{B_T} = \emptyset \text{ and } \Delta^-_{B_T} = \emptyset\} \\
& \emptyset
\end{aligned}
$$

Thus the auxiliary relation $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ are not updated. Since the base relation and the auxiliary relations are not updated, GetPut is satisfied.

Second, we prove the following PutGet is satisfied.

$$get_{t.src.i}(put_{t.src.i}(\mathbf{D}_t, S_i{}')) = S_i{}'$$

A result of $put_{t.src.i}(\mathbf{D}_t, S_i{}')$ is the updated database $\mathbf{D}_t{}'$. The updated database $\mathbf{D}_t{}'$ consists of the updated base relation $B_T{}'(\vec{Y})$, the updated auxiliary relations $A^c_{S_i}{}'(\vec{Z}_i)$, $A^{lost}_{S_i}{}'(\vec{X}_i)$, $A^{gain}_{S_i}{}'(\vec{X}_i)$, $A^{lost}_T{}'(\vec{Y})$, and $A^{gain}_T{}'(\vec{Y})$. Also, Algorithm 5.2 defines $S_i{}''(\vec{X}_i)$ as $S_i{}''(\vec{X}_i) = \pi_{attr(S_i)}(B_T{}'(\vec{Y}) \bowtie A^c_{S_i}{}'(\vec{X}_i))$. Based on them, PutGet is proved as follows.

$$\begin{aligned}
get_{t.src.i}(put_{t.src.i}(\mathbf{D}_t, S_i{}')) &= \{\text{Definition of } get_{t.src.i} \text{ in Algorithm 5.2 based on the updated} \\
&\quad\ \text{database by } put_{t.src.i}(\mathbf{D}_t, S_i{}')\} \\
&\quad\ (\pi_{attr(S_i)}(B_T{}'(\vec{Y}) \bowtie A^c_{S_i}{}'(\vec{X}_i)) \cap \neg A^{gain}_{S_i}{}') \cup A^{lost}_{S_i}{}' \\
&= \{S_i{}''(\vec{X}_i) = \pi_{attr(S_i)}(B_T{}'(\vec{Y}) \bowtie A^c_{S_i}{}'(\vec{X}_i))\} \\
&\quad\ (S_i'' \cap \neg A^{gain}_{S_i}{}') \cup A^{lost}_{S_i}{}' \\
&= \{\text{Substitute definition of } A^{lost}_{S_i}{}'(\vec{X}_i) \text{ and } A^{gain}_{S_i}{}'(\vec{X}_i) \\
&\quad\ \text{in Algorithm 5.2}\} \\
&\quad\ \Big(S_i'' \cap \neg((A^{gain}_{S_i} \cap \neg\Delta^-_{A^{gain}_{S_i}}) \cup \Delta^+_{A^{gain}_{S_i}})\Big) \\
&\quad\ \cup ((A^{lost}_{S_i} \cap \neg\Delta^-_{A^{lost}_{S_i}}) \cup \Delta^+_{A^{lost}_{S_i}}) \\
&= \{\text{Substitute definition of } \Delta^+_{A^{lost}_{S_i}}, \Delta^-_{A^{lost}_{S_i}}, \Delta^+_{A^{gain}_{S_i}}, \text{ and } \Delta^-_{A^{gain}_{S_i}} \\
&\quad\ \text{in Algorithm 5.2}\} \\
&\quad\ \Big(S_i'' \cap \neg\Big((A^{gain}_{S_i} \cap \neg((S_i' \cap A^{gain}_{S_i}) \cup (\neg S_i'' \cap A^{gain}_{S_i}))) \\
&\quad\ \cup (\neg S_i{}' \cap S_i{}'' \cap \neg A^{gain}_{S_i})\Big)\Big) \\
&\quad\ \cup \Big((A^{lost}_{S_i} \cap \neg(\neg S_i{}' \cap A^{lost}_{S_i})) \cup (S_i{}' \cap \neg S_i{}'' \cap \neg A^{lost}_{S_i})\Big) \\
&= \{\text{Deformation of a formula by set operation}\} \\
&\quad\ (S_i{}'' \cap \neg A^{gain}_{S_i} \cap S_i{}') \cup (S_i{}'' \cap S_i{}' \cap A^{gain}_{S_i}) \cup (A^{lost}_{S_i} \cap S_i{}') \\
&\quad\ \cup (S_i{}' \cap \neg S_i{}'' \cap \neg A^{lost}_{S_i})
\end{aligned}$$

$$
\begin{aligned}
&= \{(A \cap \neg B) \cup (A \cap B) = A \text{ and distributive law}\}\\
&\quad S_i{}' \cap (S_i{}'' \cup A^{lost}_{S_i} \cup S_i{}') \cap (S_i{}'' \cup A^{lost}_{S_i} \cup \neg A^{lost}_{S_i})\\
&= \{(A \cup \neg A) \cap B = B\}\\
&\quad S_i{}'
\end{aligned}
$$

Thus PutGet is satisfied. □

## A.10 Proof of Lemma 5.4

We prove this lemma by induction. Attributes $\vec{Y}$ of relations are omitted in the following.

Let $T(n)$ be the view instance of target schema at $n$ step. $T(n)$ is updated from the view instance $T(n-1)$ at $n-1$ step. Given the view instance $T(n)$, $put_{t.trg}$ transforms the base relation $B_T(n-1)$, the auxiliary relations $A^{lost}_T(n-1)$ and $A^{gain}_T(n-1)$ into updated $B_T(n)$, $A^{lost}_T(n)$ and $A^{gain}_T(n)$ respectively.

(Base) As step 0, let $A^{lost}_T(0)$ and $A^{gain}_T(0)$ be empty set. It is obvious the following equation is satisfied.

$$A^{lost}_T(0) \cap A^{gain}_T(0) = \emptyset$$

(Induction) As an induction hypothesis, suppose $A^{lost}_T(n-1)$ and $A^{gain}_T(n-1)$ are disjoint and satisfy the following equation.

$$A^{lost}_T(n-1) \cap A^{gain}_T(n-1) = \emptyset$$

We show that $A^{lost}_T(n) \cap A^{gain}_T(n) = \emptyset$ is satisfied.

$$
\begin{aligned}
A^{lost}_T(n) \cap A^{gain}_T(n) &= \{\text{Apply delta relations to } A^{lost}_T(n-1) \text{ and } A^{gain}_T(n-1)\}\\
&\quad \left((A^{lost}_T(n-1) \cap \neg\Delta^-_{A^{lost}_T}) \cup \Delta^+_{A^{lost}_T}\right)\\
&\quad \cap \left((A^{gain}_T(n-1) \cap \neg\Delta^-_{A^{gain}_T}) \cup \Delta^+_{A^{gain}_T}\right)
\end{aligned}
$$

= {Substitute transformations to $\Delta^{+}_{A_T^{lost}}$, $\Delta^{-}_{A_T^{lost}}$, $\Delta^{+}_{A_T^{gain}}$, and $\Delta^{-}_{A_T^{gain}}$ defined in Algorithm 5.3 by replacing updated $T'$ to $T(n)$ and updated $B_T{}'$ to $B_T(n)$.}

$$\Big(\big(A_T^{lost}(n-1) \cap \neg(\neg T(n) \cap A_T^{lost}(n-1))\big) \\ \cup\ (T(n) \cap \neg B_T(n) \cap \neg A_T^{lost}(n-1))\Big) \\ \cap \Big(\big(A_T^{gain}(n-1) \cap \neg((T(n) \cap A_T^{gain}(n-1)) \\ \cup (\neg B_T(n) \cap A_T^{gain}(n-1))\big) \\ \cup (\neg T(n) \cap B_T(n) \cap \neg A_T^{gain}(n-1))\Big)$$

= {Deformation of a formula by set operation and applying $A \cap \neg A = \emptyset$}

$\emptyset$

Thus $A_T^{lost} \cap A_T^{gain} = \emptyset$. □

## A.11 Proof of Lemma 5.5

Since $A_T^{gain}(\vec{Y}) \subseteq B_T(\vec{Y})$ is equivalent to $A_T^{gain}(\vec{Y}) \cap \neg B_T(\vec{Y}) = \emptyset$, we prove the following equation by induction.

$$A_T^{gain}(\vec{Y}) \cap \neg B_T(\vec{Y}) = \emptyset$$

Let $T(n)$ be the view instance of target schema at $n$ step. $T(n)$ is updated from the view instance $T(n-1)$ at $n-1$ step. Given the view instance $T(n)$, $put_{t.trg}$ transforms the base relation $B_T(n-1)$ and the auxiliary relation $A_T^{gain}(n-1)$ into updated $B_T(n)$ and $A_T^{gain}(n)$ respectively.

(Base) As step 0, let $B_T(0)$ and $A_T^{gain}(0)$ be empty set. It is obvious that the following equation is satisfied.

$$A_T^{gain}(0) \cap \neg B_T(0) = \emptyset$$

(Induction) As an induction hypothesis, suppose the following equation.

$$A_T^{gain}(n-1) \cap \neg B_T(n-1) = \emptyset$$

We show that $A_T^{gain}(n) \cap \neg B_T(n) = \emptyset$ is satisfied.

$$\begin{aligned}
A_T^{gain}(n) \cap \neg B_T(n) =\ & \{\text{Apply delta relations to } A_T^{gain}(n-1)\} \\
& ((A_T^{gain}(n-1) \cap \neg \Delta^-_{A_T^{gain}}) \cup \Delta^+_{A_T^{gain}}) \cap \neg B_T(n) \\
=\ & \{\text{Substitute transformations to } \Delta^+_{A_T^{gain}}, \text{ and } \Delta^-_{A_T^{gain}} \text{defined in} \\
& \text{Algorithm 5.3 by replacing updated } T' \text{ to } T(n) \text{ and} \\
& B_T{}' \text{ to } B_T(n).\} \\
& \Big((A_T^{gain}(n-1) \cap \neg((T(n) \cap A_T^{gain}(n-1)) \\
& \quad \cup (\neg B_T(n) \cap (A_T^{gain}(n-1))) \\
& \quad \cup (\neg T(n) \cap B_T(n) \cap \neg A_T^{gain}(n-1))\Big) \\
& \quad \cap \neg B_T(n) \\
=\ & \{\text{Deformation of a formula by set operation and applying} \\
& \ A \cap \neg A = \emptyset\} \\
& \emptyset
\end{aligned}$$

Thus $A_T^{gain} \cap \neg B_T = \emptyset$. □

## A.12 Proof of Proposition5.6

First, we prove following GETPUTis satisfied.

$$put_{t.trg}(\mathbf{D}_t, get_{t.trg}(\mathbf{D}_t)) = \mathbf{D}_t$$

where the database $\mathbf{D}_t$ is union of the base relation $B_T$ and the auxiliary relations $A^c_{S_i}$, $A^{lost}_{S_i}$, $A^{gain}_{S_i}$ for all $i$ ($i \in [1, n]$), $A^{lost}_T$ and $A^{gain}_T$. We prove GETPUT by showing a result of $put_{t.trg}$ after $get_{t.trg}$ does not update the base relations and the auxiliary relations of the database.

Since a result of $get_{t.trg}$ is not updated in the proof of GetPut, the update view instance $T'(\vec{Y})$ appearing in $put_{t.trg}$ is equivalent to $T(\vec{Y})$ as a result of $get_{t.trg}$. By substituting $T'(\vec{Y}) = T(\vec{Y})$ into transformations to $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$ defined in the algorithm, they become empty set as follows. We omit attributes if all relations in a formula have the same attributes in the following.

$$\begin{aligned}
\Delta^+_T &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad T' \cap \neg T\\
&= \{T' = T \text{ and } A \cap \neg A = \emptyset\}\\
&\quad \emptyset\\
\Delta^-_T &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \neg T' \cap T\\
&= \{T' = T \text{ and } A \cap \neg A = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

By substituting $\Delta^+_T(\vec{Y}) = \emptyset$ and $\Delta^-_T(\vec{Y}) = \emptyset$ into transformations to $\Delta^+_{B_t}(\vec{Y})$ and $\Delta^-_{B_t}(\vec{Y})$ defined in the algorithm, they become empty set as follows:

$$\begin{aligned}
\Delta^+_{B_T}(\vec{Y}) &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \pi_{attr(T)}(\Delta^+_{S_i}(\vec{X_i}) \bowtie \Delta^+_T(\vec{Y})) \cap \neg B_T(\vec{Y})\\
&= \{\text{Substitute } \Delta^+_T(\vec{Y}) = \emptyset\}\\
&\quad \emptyset\\
\Delta^-_{B_T}(\vec{Y}) &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \pi_{attr(T)}(\Delta^-_{S_i}(\vec{X_i}) \bowtie \Delta^-_T(\vec{Y})) \cap B_T(\vec{Y})\\
&= \{\text{Substitute } \Delta^-_T(\vec{Y}) = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

Thus base relation $B_T$ is not updated. Updated $B_T{}'$ is equivalent to $B_T$.

By substituting $T'(\vec{Y}) = T(\vec{Y})$ and $B_T{}'(\vec{Y}) = B_T(\vec{Y})$ into transformations to $\Delta^+_{A_T{}^{lost}}(\vec{Y})$ and $\Delta^-_{A_T{}^{lost}}(\vec{Y})$ defined in the algorithm, they become empty set as follows:

$$
\begin{aligned}
\Delta^{+}_{A_T{}^{lost}} &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad T' \cap \neg B_T{}' \cap \neg A_T^{lost}\\
&= \{T' = T, B_T{}' = B_T\}\\
&\quad T \cap \neg B_T \cap \neg A_T^{lost}\\
&= \{\text{Substitute definition of } T \text{ in Algorithm 5.3}\}\\
&\quad ((B_T \cap \neg A_T^{gain}) \cup A_T^{lost}) \cap \neg B_T \cap \neg A_T^{lost}\\
&= \{\text{Distributive law}\}\\
&\quad (B_T \cap \neg A_T^{gain} \cap \neg B_T \cap \neg A_T^{lost}) \cup (A_T^{lost} \cap \neg B_T \cap \neg A_T^{lost})\\
&= \{A \cap \neg A = \emptyset\}\\
&\quad \emptyset\\
\Delta^{-}_{A_T{}^{lost}} &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \neg T' \cap A_T^{lost}\\
&= \{T' = T \text{ and substitute definition of } T \text{ in Algorithm 5.3}\}\\
&\quad \neg((B_T \cap \neg A_T^{gain}) \cup A_T^{lost}) \cap A_T^{lost}\\
&= \{\text{De Morgan's law}\}\\
&\quad (\neg(B_T \cap \neg A_T^{gain}) \cap \neg A_T^{lost}) \cap A_T^{lost}\\
&= \{A \cap \neg A = \emptyset\}\\
&\quad \emptyset
\end{aligned}
$$

Thus the auxiliary relation $A_T^{lost}$ is not updated.

By substituting $T'(\vec{Y}) = T(\vec{Y})$ and $B_T{}'(\vec{Y}) = B_T(\vec{Y})$ into transformations to $\Delta^{+}_{A_T{}^{gain}}(\vec{Y})$ and $\Delta^{-}_{A_T{}^{gain}}(\vec{Y})$ defined in the algorithm, they become empty set as follows:

$$
\begin{aligned}
\Delta^{+}_{A_T{}^{gain}} &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \neg T' \cap B_T{}' \cap \neg A_T^{gain}\\
&= \{T' = T, B_T{}' = B_T\}\\
&\quad \neg T \cap B_T \cap \neg A_T^{gain}
\end{aligned}
$$

$$
\begin{aligned}
&= \{\text{Substitute definition of } T \text{ in Algorithm 5.3}\}\\
&\quad \neg((B_T \cap \neg A_T^{gain}) \cup A_T^{lost}) \cap B_T \cap \neg A_T^{gain}\\
&= \{\text{De Morgan's law and Distributive law}\}\\
&\quad (\neg B_T \cap \neg A_T^{lost} \cap B_T \cap \neg A_T^{gain}) \cup (A_T^{gain} \cap \neg A_T^{lost} \cap B_T \cap \neg A_T^{gain})\\
&= \{A \cap \neg A = \emptyset\}\\
&\quad \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\Delta^-_{A_T{}^{gain}} &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad (T' \cap A_T^{gain}) \cup (\neg B_T{}' \cap A_T^{gain})\\
&= \{T' = T, B_T{}' = B_T \text{ and substitute definition of } T \text{ in Algorithm 5.3}\}\\
&\quad (((B_T \cap \neg A_T^{gain}) \cup A_T^{lost}) \cap A_T^{gain}) \cup (\neg B_T \cap A_T^{gain})\\
&= \{\text{Distributive law}\}\\
&\quad (B_T \cap \neg A_T^{gain} \cap A_T^{gain}) \cup (A_T^{lost} \cap A_T^{gain}) \cap (\neg B_T \cap A_T^{gain})\\
&= \{A \cap \neg A = \emptyset \text{ , Lemma 5.4 and Lemma 5.5 }\}\\
&\quad \emptyset
\end{aligned}
$$

Thus the auxiliary relation $A_T^{gain}$ is not updated.

Since $\Delta_T^+$ and $\Delta_T^-$ are empty set, $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ as results of backward updated sharing $g^+_{s_i}$ and $g^-_{s_i}$ are empty set for all $i$ ($i \in [1, n]$). By substituting $\Delta^+_{S_i}(\vec{X}_i) = \emptyset$ and $\Delta^-_{S_i}(\vec{X}_i) = \emptyset$ into transformations to $\Delta^+_{A^c_{S_i}}(\vec{Z}_i)$, $\Delta^-_{A^c_{S_i}}(\vec{Z}_i)$, $\Delta^-_{A^{lost}_{S_i}}(\vec{X}_i)$, and $\Delta^-_{A^{gain}_{S_i}}(\vec{X}_i)$ for each $i$, they become empty set as follows:

$$
\begin{aligned}
\Delta^+_{A^c_{S_i}}(\vec{Z}_i) &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \pi_{\vec{Z}_i}(\Delta^+_{S_i}(\vec{X}_i)) \cap \neg A^c_{S_i}(\vec{Z}_i)\\
&= \{\Delta^+_{S_i}(\vec{X}_i) = \emptyset\}\\
&\quad \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\Delta^-_{A^c_{S_i}}(\vec{Z}_i) &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad \pi_{\vec{Z}_i}(\Delta^-_{S_i}{}'(\vec{X}_i)) \cap A^c_{S_i}(\vec{Z}_i)\\
&= \{\Delta^-_{S_i}{}'(\vec{X}_i) = \Delta^-_{S_i}(\vec{X}_i) \bowtie (\pi_{key}(\Delta^-_{S_i}(\vec{X}_i)) \cap \neg \pi_{key}(\Delta^+_{S_i}(\vec{X}_i))) \text{ and } \Delta^-_{S_i}(\vec{X}_i) = \emptyset\}\\
&\quad \emptyset
\end{aligned}
$$

$$\begin{aligned}
\Delta^{-}_{A^{lost}_{S_i}}(\vec{X_i}) &= \{\text{Definition in Algorithm 5.3}\} \\
&\quad (\Delta^{+}_{S_i}(\vec{X_i}) \cap A^{lost}_{S.i}(\vec{X_i})) \cup (\Delta^{-}_{S_i}(\vec{X_i}) \cap A^{lost}_{S.i}(\vec{X_i})) \\
&= \{\Delta^{+}_{S_i}(\vec{X_i}) = \emptyset, \Delta^{-}_{S_i}(\vec{X_i}) = \emptyset\} \\
&\quad \emptyset \\
\Delta^{-}_{A^{gain}_{S_i}}(\vec{X_i}) &= \{\text{Definition in Algorithm 5.3}\} \\
&\quad (\Delta^{+}_{S_i}(\vec{X_i}) \cap A^{gain}_{S.i}(\vec{X_i})) \cup (\Delta^{-}_{S_i}(\vec{X_i}) \cap A^{gain}_{S.i})(\vec{X_i}) \\
&= \{\Delta^{+}_{S_i}(\vec{X_i}) = \emptyset, \Delta^{-}_{S_i}(\vec{X_i}) = \emptyset\} \\
&\quad \emptyset
\end{aligned}$$

Thus the auxiliary relations $A^{c}_{S_i}$, $A^{lost}_{S_i}$ and $A^{gain}_{S_i}$ are not updated. Since the base relation and the auxiliary relations are not updated, GetPut is satisfied.

Second, we prove following PutGet is satisfied.

$$get_{t.trg}(put_{t.trg}(\mathbf{D}_t, T')) = T'$$

A result of $put_{t.trg}(\mathbf{D}_t, T')$ is the updated database $\mathbf{D}_t{}'$. The updated database $\mathbf{D}_t{}'$ consists of the updated base relation $B_T{}'(\vec{Y})$, the updated auxiliary relations $A^{c}_{S_i}{}'(\vec{Z_i})$, $A^{lost}_{S_i}{}'(\vec{X_i})$, $A^{gain}_{S_i}{}'(\vec{X_i})$, $A^{lost}_{T}{}'(\vec{Y})$, and $A^{gain}_{T}{}'(\vec{Y})$. PutGet is proved as follows.

$$\begin{aligned}
get_{t.trg}(put_{t.trg}(\mathbf{D}_t, T')) &= \{\text{Definition of } get_{t.trg} \text{ in Algorithm 5.3 based on the} \\
&\quad \text{updated database by } put_{t.trg}(\mathbf{D}_t, T')\} \\
&\quad (B'_T \cap \neg A^{gain\prime}_T) \cup A^{lost\prime}_T \\
&= \{\text{Substitute definition of } A^{lost\prime}_T(\vec{X_i}) \text{ and } A^{gain\prime}_T(\vec{X_i}) \\
&\quad \text{in Algorithm 5.3}\} \\
&\quad \left(B'_T \cap \neg((A^{gain}_T \cap \neg\Delta^{-}_{A^{gain}_T}) \cup \Delta^{+}_{A^{gain}_T})\right) \\
&\quad \cup ((A^{lost}_T \cap \neg\Delta^{-}_{A^{lost}_T}) \cup \Delta^{+}_{A^{lost}_T}) \\
&= \{\text{Substitute definition of } \Delta^{+}_{A^{lost}_T}, \Delta^{-}_{A^{lost}_T}, \Delta^{+}_{A^{gain}_T}, \text{ and } \Delta^{-}_{A^{gain}_T} \\
&\quad \text{in Algorithm 5.3}\}
\end{aligned}$$

$$
\begin{aligned}
&\Big(B_T' \cap \neg\Big((A_T^{gain} \cap \neg((T' \cap A_T^{gain}) \cup (\neg B_T{}' \cap A_T^{gain})))\\
&\quad \cup (\neg T' \cap B_T{}' \cap \neg A_T^{gain})\Big)\Big)\\
&\quad \cup \Big((A_T^{lost} \cap \neg(\neg T' \cap A_T^{lost})) \cup (T' \cap \neg B_T{}' \cap \neg A_T^{lost})\Big)\\
= &\ \{\text{Deformation of a formula by set operation}\}\\
&(B_T{}' \cap \neg A_T^{gain} \cap T') \cup (B_T{}' \cap T' \cap A_T^{gain}) \cup (A_T^{lost} \cap T')\\
&\cup (T' \cap \neg B_T{}' \cap \neg A_T^{lost})\\
= &\ \{(A \cap \neg B) \cup (A \cap B) = A \text{ and distributive law}\}\\
&T' \cap (B_T{}' \cup A_T^{lost} \cup T') \cap (B_T{}' \cup A_T^{lost} \cup \neg A_T^{lost})\\
= &\ \{(A \cup \neg A) \cap B = B\}\\
&T'
\end{aligned}
$$

Thus PutGet is satisfied. □

## A.13 Proof of Theorem 5.7

Suppose a co-existence strategy strategy satisfying the *consistency* of updates between relations $S_i(\vec{X_i})$ $(i \in [1, n])$ of source schema and a relation $T(\vec{Y})$ of target schema. Recall that schema evolution is specified as a transformation $f$ from source schema instance S to a relation $T(\vec{Y})$.

$$T(\vec{Y}) = f(\mathbf{S}) \tag{A.8}$$

Source schema instance **S** is union of relations $S_i(\vec{X_i})$ $(i \in [1, n])$ of source schema.

$BX_{t.src.i}$ for all $i$ are derived by Algorithm 5.2 from the co-existence strategy. $put_{t.src.i}$ of $BX_{t.src.i}$ for each $i$ transforms the updated view instance $S_i{}'$ to the updated target-side database $\mathbf{D}_t{}'$. $BX_{t.trg}$ is derived by Algorithm 5.3 from the co-existence strategy. $get_{t.trg}$ of $BX_{t.trg}$ transforms the database $\mathbf{D}_t$ to the view instance $T$. The derived BXs realize schema evolution of a co-existence strategy if the following two cases are satisfied: as an initial state, a transformation from union of the view instance $S_i$ to the view instance $T$ via $\mathbf{D}_t$ by $BX_{t.src.i}$ for all $i$ and $BX_{t.trg}$ is equivalent to a result of the transformation by schema evolution, and as forward update sharing, a transformation of updates against $S_i$ to updates against $T$ via updated $\mathbf{D}_t{}'$ by $BX_{t.src.i}$ and $BX_{t.trg}$ is equivalent to a result of update sharing by schema evolution.

First, it is proved that a result of schema evolution is equivalent to a result of $get_{t.trg}$ of $BX_{t.trg}$ after $put_{t.src.i}$ of $BX_{t.src.i}$ for all $i$ from the view instances $S_i(\vec{X_i})$ for all $i$ and empty set of the base relation and the auxiliary relations as the initial state. Given empty set of the base relation, empty set of the auxiliary relations, and the instance of source schema $\mathbb{S}$ as union of the view instance $S_i(\vec{X_i})$ for all $i$, the transformations of $put_{t.src.i}$ of $BX_{t.src.i}$ compute $T^{evo}(\vec{Y})$ as empty set and $T^{evo\prime}(\vec{Y})$ as a result of schema evolution $f(\mathbb{S})$. By substituting such $T^{evo}(\vec{Y}) = \emptyset$, $T^{evo\prime}(\vec{Y}) = f(\mathbb{S})$ and empty set of the base relation into the transformation to $\Delta^+_{B_T}$ and $\Delta^-_{B_T}$ in $put_{t.src.i}$, the updated base relation $B_T{}'(\vec{Y})$ becomes $f(\mathbb{S})$ and the updated auxiliary relations $A^{lost}_T(\vec{Y})'$ and $A^{gain}_T(\vec{Y})$ become empty set as follows:

$$\begin{aligned}
B_T{}'(\vec{Y}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad (B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y}) \\
&= \{\text{Substitute definiton of } \Delta^+_{B_T}(\vec{Y}) \text{ and } \Delta^-_{B_T}(\vec{Y}) \text{ in Algorithm 5.2, } B_T(\vec{Y}) = \emptyset, \\
&\quad T^{evo}(\vec{Y}) = \emptyset, \text{ and } T^{evo\prime}(\vec{Y}) = f(\mathbb{S})\} \\
&\quad f(\mathbb{S}) \\
A^{lost\prime}_T(\vec{Y}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad A^{lost}_T(\vec{Y}) \cap \neg\Delta^-_{A^{lost}_T}(\vec{Y}) \\
&= \{\text{Substitute definiton of } \Delta^-_{A^{lost}_T}(\vec{Y}) \text{ in Algorithm 5.2 and } A^{lost}_T(\vec{Y}) = \emptyset, \\
&\quad \emptyset \\
A^{gain\prime}_T(\vec{Y}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad A^{gain}_T(\vec{Y}) \cap \neg\Delta^-_{A^{gain}_T}(\vec{Y}) \\
&= \{\text{Substitute definiton of } \Delta^-_{A^{gain}_T}(\vec{Y}) \text{ in Algorithm 5.2 and } A^{gain}_T(\vec{Y}) = \emptyset, \\
&\quad \emptyset
\end{aligned}$$

Then, by renaming the updated $B_T{}'(\vec{Y})$, $A^{lost\prime}_T(\vec{Y})$, and $A^{gain\prime}_T(\vec{Y})$ to the current state of the base relation and auxiliary relations as $B_T(\vec{Y})$, $A^{lost}_T(\vec{Y})$, and $A^{gain}_T(\vec{Y})$ respectively, they are transformed to the view instance $T(\vec{Y})$ of target schema by $get_{t.trg}$ of $BX_{t.trg}$ as follows:

$$\begin{aligned}
T(\vec{Y}) &= \{\text{Definition in Algorithm 5.3}\}\\
&\quad (B_T(\vec{Y}) \cap \neg A_T^{gain}(\vec{Y})) \cup A_T^{lost}(\vec{Y})\\
&= \{\text{Substitute } A_T^{gain}(\vec{Y}) = \emptyset \text{ and } A_T^{lost}(\vec{Y}) = \emptyset\}\\
&\quad B_T(\vec{Y})\\
&= \{\text{Substitute } B_T(\vec{Y}) = f(\mathrm{S})\}\\
&\quad f(\mathrm{S})
\end{aligned}$$

Thus a result of schema evolution expressed by the equation (A.8) is equivalent to a result of $get_{t.trg}$ of $BX_{t.trg}$ after $put_{t.src.i}$ of $BX_{t.src.i}$ for all $i$ based on the view instances $S_i(\vec{X_i})$ for all $i$ and empty set of the base relations and the auxiliary relations.

Second, it is proved that a transformation of updates against $S_i(\vec{X_i})$ to updates against $T(\vec{Y})$ by $get_{t.trg}$ of $BX_{trg}$ after $put_{t.src.i}$ of $BX_{t.src.i}$ is equivalent to a result of update sharing by schema evolution. Let $\mathrm{S}'$ be union of relations of source schema that one relation $S_i(\vec{X_i})$ is updated to $S_i'(\vec{X_i})$. $T'(\vec{Y})$ is a result of schema evolution from $\mathrm{S}'$ as follows:

$$T'(\vec{Y}) = f(\mathrm{S}') \tag{A.9}$$

By tuning the relations of source schema into the view instance, let S be union of view instances of source schema and S′ be union of view instances of source schema that one view instance $S_i(\vec{X_i})$ is updated to $S_i'(\vec{X_i})$. Then $put_{t.src.i}$ transforms S and S′ by schema evolution $f$ as follows:

$$\begin{aligned}
T^{evo}(\vec{Y}) &= f(\mathrm{S})\\
T^{evo\prime}(\vec{Y}) &= f(\mathrm{S}')
\end{aligned}$$

$put_{t.src.i}$ transforms $T^{evo}(\vec{Y})$ and $T^{evo\prime}(\vec{Y})$ into the updated base relation $B_T{}'(\vec{Y})$ and the updated auxiliary relations $A_T^{lost\prime}(\vec{Y_i})$ and $A_T^{gain\prime}(\vec{Y_i})$ by computing sets of inserted and deleted tuples $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^-_{B_T}(\vec{Y})$ as follows:

$$\begin{aligned}
\Delta^+_{B_T}(\vec{Y}) &= T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \cap \neg B_T(\vec{Y}) \\
\Delta^-_{B_T}(\vec{Y}) &= \neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y}) \cap B_T(\vec{Y}) \\
B_T{}'(\vec{Y}) &= (B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y}) \\
\Delta^-_{A^{lost}_T}(\vec{Y}) &= (\Delta^+_{B_T}(\vec{Y}) \cap A^{lost}_T)(\vec{Y}) \cup (\Delta^-_{B_T}(\vec{Y}) \cap A^{lost}_T(\vec{Y})) \\
A^{lost\prime}_T(\vec{Y}) &= A^{lost}_T(\vec{Y}) \cap \neg\Delta^-_{A^{lost}_T}(\vec{Y}) \\
\Delta^-_{A^{gain}_T}(\vec{Y}) &= (\Delta^+_{B_T}(\vec{Y}) \cap A^{gain}_T(\vec{Y})) \cup (\Delta^-_{B_T}(\vec{Y}) \cap A^{gain}_T(\vec{Y})) \\
A^{gain\prime}_T(\vec{Y}) &= A^{gain}_T(\vec{Y}) \cap \neg\Delta^-_{A^{gain}_T}(\vec{Y})
\end{aligned}$$

$B_T{}'(\vec{Y})$ is equivalent to $T^{evo\prime}(\vec{Y})$ as follows:

$$\begin{aligned}
B_T{}'(\vec{Y}) &= \{\text{Definition in Algorithm 5.2}\} \\
&\quad (B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y}) \\
&= \{\text{Substitute } \Delta^-_{B_T}(\vec{Y}) \text{ and } \Delta^-_{B_T}(\vec{Y}) \text{ defined in Algorithm 5.2 }\} \\
&\quad \Big(B_T(\vec{Y}) \cap \neg\big(\neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y}) \cap B_T(\vec{Y})\big)\Big) \\
&\quad \cup \Big(T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \cap \neg B_T(\vec{Y})\Big) \\
&= \{\text{Substitute } B_T(\vec{Y}) = f(\mathbf{S}) \text{ and } T^{evo}(\vec{Y}) = f(\mathbf{S})\} \\
&\quad \Big(T^{evo}(\vec{Y}) \cap \neg\big(\neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y}) \cap T^{evo}(\vec{Y})\big)\Big) \\
&\quad \cup \Big(T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \cap \neg T^{evo}(\vec{Y})\Big) \\
&= \{\text{Deformation of a formula by set operation}\} \\
&\quad \big(T^{evo}(\vec{Y}) \cap T^{evo\prime}(\vec{Y})\big) \cup \big(T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y})\big) \\
&= \{(A \cap B) \cup (A \cap \neg B) = A\} \\
&\quad T^{evo\prime}(\vec{Y})
\end{aligned}$$

Then $get_{t.trg}$ transforms these updated base relation and auxiliary relations into the updated view instance $T'(\vec{Y})$ of target schema as follows:

$$T'(\vec{Y}) = \{\text{Definition in Algorithm 5.2}\}$$

$$
\begin{aligned}
&(B_T{}'(\vec{Y}) \cap \neg A_T^{gain\prime}(\vec{Y})) \cup A_T^{lost\prime}(\vec{Y}) \\
=\ &\{\text{Substitute } B_T{}'(\vec{Y}) = f(\mathrm{S}') \text{ because of } B_T{}'(\vec{Y}) = T^{evo\prime}(\vec{Y}) \text{ and} \\
&T^{evo\prime}(\vec{Y}) = f(\mathrm{S}')\} \\
&\left(f(\mathrm{S}') \cap \neg A_T^{gain\prime}(\vec{Y})\right) \cup A_T^{lost\prime}(\vec{Y})
\end{aligned}
$$

Because of $B_T(\vec{Y}) = f(\mathrm{S})$ and $B_T{}'(\vec{Y}) = f(\mathrm{S}')$, the updates from $f(\mathrm{S})$ to $f(\mathrm{S}')$ are equivalent to the updates from $B_T(\vec{Y})$ to $B_T{}'(\vec{Y})$ when union of view instances of source schema is updated from S to S′. As mentioned above, $put_{t.src.i}$ updates the base relation $B_T(\vec{Y})$ to $B_T{}'(\vec{Y})$ by applying sets of inserted and deleted tuples $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^-_{B_T}(\vec{Y})$. It also updates the auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ to $A_T^{lost\prime}(\vec{Y})$ and $A_T^{gain\prime}(\vec{Y})$ by excluding tuples of $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^-_{B_T}(\vec{Y})$. Thus, inserted and deleted tuples from $f(\mathrm{S})$ to $f(\mathrm{S}')$ as a result of update sharing by schema evolution $f$ are not deleted and inserted by the updated auxiliary relations $A_T^{lost\prime}(\vec{Y})$ and $A_T^{gain\prime}(\vec{Y})$ in the transformation to $T'(\vec{Y})$ by $get_{t.trg}$. Therefore, transformation of updates against $S_i(\vec{X}_i)$ to updates against $T(\vec{Y})$ by $get_{t.trg}$ after $put_{t.src.i}$ is equivalent to a result of update sharing by schema evolution. □

## A.14 Proof of Theorem 5.8

Suppose a co-existence strategy strategy satisfying the *consistency* of updates between relations $S_i(\vec{X}_i)$ ($i \in [1, n]$) of source schema and a relation $T(\vec{Y})$ of target schema. Recall that backward update sharing is specified as transformations $g^+_{s_i}$ and $g^-_{s_i}$ for each $i$ ($i \in [1, n]$) from a pair of source schema instance S and delta relation $\Delta T(\vec{Y})$ to $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ as sets of inserted and deleted tuples against a relation $S_i$ of source schema.

$$\Delta^+_{S_i}(\vec{X}_i) = g^+_{s_i}(\mathrm{S}, \Delta T(\vec{Y})) \tag{A.10}$$

$$\Delta^-_{S_i}(\vec{X}_i) = g^-_{s_i}(\mathrm{S}, \Delta T(\vec{Y})) \tag{A.11}$$

where source schema instance S is union of all relations $S_i$ of source schema. Such $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ update the relation $S_i(\vec{X}_i)$ to $S_i{}'(\vec{X}_i)$.

On the other hand, $BX_{t.trg}$ is derived by Algorithm 5.3 from the co-existence strategy. $put_{t.trg}$ of $BX_{t.trg}$ transforms $\Delta T(\vec{Y})$ as the delta relation of the view instance $T(\vec{Y})$ of target schema into the updated database $\mathbf{D}_T{}'$. The base relation $B_T(\vec{Y})$ and auxiliary relations $A^c_{S_i}(\vec{Z_i})$, $A^{lost}_{S_i}(\vec{X_i})$, and $A^{gain}_{S_i}(\vec{X_i})$ of the database are updated to $B_T{}'(\vec{Y})$, $A^c_{S_i}{}'(\vec{Z_i})$, $A^{lost'}_{S_i}(\vec{X_i})$, and $A^{gain'}_{S_i}(\vec{X_i})$ as follows:

$$\Delta^+_{B_T}(\vec{Y}) = \pi_{attr(T)}(\Delta^+_{S_i}(\vec{X_i}) \bowtie \Delta^+_T(\vec{Y})) \cap \neg B_T(\vec{Y}) \tag{A.12}$$
$$\Delta^-_{B_T}(\vec{Y}) = \pi_{attr(T)}(\Delta^-_{S_i}(\vec{X_i}) \bowtie \Delta^-_T(\vec{Y})) \cap B_T(\vec{Y}) \tag{A.13}$$
$$B_T{}'(\vec{Y}) = (B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y}) \tag{A.14}$$

$$\Delta^+_{A^c_{S_i}}(\vec{Z_i}) = \pi_{\vec{Z_i}}(\Delta^+_{S_i}(\vec{X_i})) \cap \neg A^c_{S_i}(\vec{Z_i}) \tag{A.15}$$
$$\Delta^-_{S_i}{}'(\vec{X_i}) = \Delta^-_{S_i}(\vec{X_i}) \bowtie (\pi_{key}(\Delta^-_{S_i}(\vec{X_i})) \cap \neg\pi_{key}(\Delta^+_{S_i}(\vec{X_i}))) \tag{A.16}$$
$$\Delta^-_{A^c_{S_i}}(\vec{Z_i}) = \pi_{\vec{Z_i}}(\Delta^-_{S_i}{}'(\vec{X_i})) \cap A^c_{S_i}(\vec{Z_i}) \tag{A.17}$$
$$A^c_{S_i}{}'(\vec{Z_i}) = (A^c_{S_i}(\vec{Z_i}) \cap \neg\Delta^-_{A^c_{S_i}}(\vec{Z_i})) \cup \Delta^+_{A^c_{S_i}}(\vec{Z_i}) \tag{A.18}$$

$$\Delta^-_{A^{lost}_{S.i}}(\vec{X_i}) = (\Delta^+_{S_i}(\vec{X_i}) \cap A^{lost}_{S.i}(\vec{X_i})) \cup (\Delta^-_{S_i}(\vec{X_i}) \cap A^{lost}_{S.i}(\vec{X_i})) \tag{A.19}$$
$$A^{lost'}_{S_i}(\vec{X_i}) = A^{lost}_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{A^{lost}_{S_i}}(\vec{X_i}) \tag{A.20}$$

$$\Delta^-_{A^{gain}_{S.i}}(\vec{X_i}) = (\Delta^+_{S_i}(\vec{X_i}) \cap A^{gain}_{S.i}(\vec{X_i})) \cup (\Delta^-_{S_i}(\vec{X_i}) \cap A^{gain}_{S.i})(\vec{X_i}) \tag{A.21}$$
$$A^{gain'}_{S_i}(\vec{X_i}) = A^{gain}_{S_i}(\vec{X_i}) \cap \neg\Delta^-_{A^{gain}_{S_i}}(\vec{X_i}) \tag{A.22}$$

where $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ are results of backward updated sharing of (A.10) and (A.11) based on S as union of view instances $S_i(\vec{X_i})$ for all $i$ of source schema and $\Delta T(\vec{Y})$ as the delta relation of the view instance $T(\vec{Y})$ of target schema.

The updated base relations and auxiliary relations are transformed into the updated view instance $S_i{}'(\vec{X}_i)$ of source schema by $get_{t.src.i}$ as follows:

$$S_i^{evo}{}'(\vec{X}_i) = \pi_{attr(S_i)}(B_T{}'(\vec{Y}) \bowtie A^c_{S.i}{}'(\vec{Z}_i)) \tag{A.23}$$

$$S_i{}'(\vec{X}_i) = (S_i^{evo}{}'(\vec{X}_i) \cap \neg A^{gain}_{S_i}{}'(\vec{X}_i)) \cup A^{lost}_{S_i}{}'(\vec{X}_i) \tag{A.24}$$

Let $S_i^{evo}{}'(\vec{X}_i)$ be updated by $\Delta^+_{S_i^{evo}}(\vec{X}_i)$ and $\Delta^-_{S_i^{evo}}(\vec{X}_i)$ as sets of inserted and deleted tuples against $S_i^{evo}(\vec{X}_i)$. In the transformation to the updated view instance $S_i{}'(\vec{X}_i)$ by (A.24), if $\Delta^+_{S_i^{evo}}(\vec{X}_i)$ and $\Delta^-_{S_i^{evo}}(\vec{X}_i)$ are equivalent to $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ respectively, tuples of $\Delta^+_{S_i^{evo}}(\vec{X}_i)$ are not deleted by tuples of the updated auxiliary relation $A^{gain}_{S_i}{}'(\vec{X}_i)$, and tuples of $\Delta^-_{S_i^{evo}}(\vec{X}_i)$ are not added by tuples of the updated auxiliary relation $A^{lost}_{S_i}{}'(\vec{X}_i)$, the updated view instance $S_i{}'(\vec{X}_i)$ is computed by applying all tuples of $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ transformed by backward update sharing. Thus, the derived BXs realize backward update sharing of a co-existence strategy.

First, it is shown that $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ are equivalent to $\Delta^+_{S_i^{evo}}(\vec{X}_i)$ and $\Delta^-_{S_i^{evo}}(\vec{X}_i)$ respectively when tuples of $\Delta^+_{S_i}(\vec{X}_i)$ do not exist in $S_i^{evo}(\vec{X}_i)$ and tuples of $\Delta^-_{S_i}(\vec{X}_i)$ exist in $S_i^{evo}(\vec{X}_i)$. Since backward update sharing is restricted to be the *monotonic* so that $\Delta^+_{S_i}(\vec{X}_i)$ is transformed from $\Delta^+_T(\vec{Y})$ and $\Delta^-_{S_i}(\vec{X}_i)$ is transformed from $\Delta^-_T(\vec{Y})$, tuples of $\Delta^+_{S_i}(\vec{X}_i)$ and $\Delta^-_{S_i}(\vec{X}_i)$ are not generated without tuples of $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$ respectively. Thus, when tuples are inserted to the view instance $T(\vec{Y})$ by $\Delta^+_T(\vec{Y})$, $\Delta^+_{S_i}(\vec{X}_i)$ as a result of backward update sharing is transformed to both $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^+_{A^c_{S_i}}$ by (A.12) and (A.15) respectively if tuples of $\Delta^+_{S_i}(\vec{X}_i)$ do not exist in $S_i^{evo}(\vec{X}_i)$ defined as $S_i^{evo}(\vec{X}_i) = \pi_{attr(S_i)}(B_T(\vec{Y}) \bowtie A^c_{S.i}(\vec{Z}_i))$ by $get_{t.src.i}$. Attributes $\vec{Z}_i$ consist of key to uniquely specify a tuple having attributes $\vec{Y}$. When tuples are deleted from the view instance $T(\vec{Y})$ by $\Delta^-_T(\vec{Y})$, $\Delta^-_{S_i}(\vec{X}_i)$ as a result of backward update sharing is transformed to both $\Delta^-_{B_T}(\vec{Y})$ and $\Delta^-_{A^c_{S_i}}$ by (A.13), (A.16), and (A.17) respectively if tuples of $\Delta^+_{S_i}(\vec{X}_i)$ exist in $S_i^{evo}(\vec{X}_i)$. When tuples of the view instance $T(\vec{Y})$ is replaced by $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$, tuples of $B_T(\vec{Y})$ is replaced by $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^-_{B_T}(\vec{Y})$ while tuples of $A^c_{S_i}$ is not replaced due to (A.18). Therefore, dangling tuples do not exit in $B_T(\vec{Y})$ and $A^c_{S.i}(\vec{Z}_i)$. $\Delta^+_{S_i^{evo}}(\vec{X}_i)$ and $\Delta^-_{S_i^{evo}}(\vec{X}_i)$ are specified as follows:

$$\Delta^+_{S_i^{evo}}(\vec{X}_i) = \pi_{attr(S_i)}(\Delta^+_{B_T}(\vec{Y}) \bowtie \Delta^+_{A^c_{S.i}}(\vec{Z}_i))$$

$$\Delta^-_{S_i^{evo}}(\vec{X}_i) = \pi_{attr(S_i)}(\Delta^-_{B_T}(\vec{Y}) \bowtie \Delta^-_{A^c_{S.i}}(\vec{Z}_i))$$

Since $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ are transformed to a pair of $\Delta^+_{B_T}(\vec{Y})$ and $\Delta^+_{A^c_{S.i}}(\vec{Z_i})$ and a pair of pair of $\Delta^-_{B_T}(\vec{Y})$ and $\Delta^-_{A^c_{S.i}}(\vec{Z_i})$, $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ are equivalent to $\Delta^+_{S^{evo}_i}(\vec{X_i})$ and $\Delta^-_{S^{evo}_i}(\vec{X_i})$ respectively when tuples of $\Delta^+_{S_i}(\vec{X_i})$ do not exist in $S^{evo}_i(\vec{X_i})$ and tuples of $\Delta^-_{S_i}(\vec{X_i})$ exist in $S^{evo}_i(\vec{X_i})$.

Second, it is shown that the updated view instance $S_i{}'(\vec{X_i})$ of source schema is a result of adding tuples of $\Delta^+_{S_i}(\vec{X_i})$ and deleting tuples of $\Delta^-_{S_i}(\vec{X_i})$ against the original view instance $S_i(\vec{X_i})$ when tuples of $\Delta^+_{S_i}(\vec{X_i})$ do not exist in $S^{evo}_i(\vec{X_i})$ and tuples of $\Delta^-_{S_i}(\vec{X_i})$ exist in $S^{evo}_i(\vec{X_i})$. (A.24) transfroms $S^{evo\prime}_i(\vec{X_i})$ updated by $\Delta^+_{S_i{}^{evo}}(\vec{X_i})$ and $\Delta^-_{S_i{}^{evo}}(\vec{X_i})$, $A^{lost\prime}_{S_i}(\vec{X_i})$, and $A^{gain\prime}_{S_i}(\vec{X_i})$ into the updated view instance $S_i{}'(\vec{X_i})$. (A.20) transforms to the updated auxiliary relation $A^{lost\prime}_{S_i}(\vec{X_i})$ by excluding tuples of $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ from the original $A^{lost}_{S_i}(\vec{X_i})$. (A.22) transforms to the updated auxiliary relation $A^{gain\prime}_{S_i}(\vec{X_i})$ by excluding tuples of $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ from the original $A^{gain}_{S_i}(\vec{X_i})$. Thus, tuples of $\Delta^+_{S_i{}^{evo}}(\vec{X_i})$ are not deleted from $S^{evo\prime}_i(\vec{X_i})$ and tuples of $\Delta^-_{S_i{}^{evo}}(\vec{X_i})$ are not added to $S^{evo\prime}_i(\vec{X_i})$ in the transformation to $S_i{}'(\vec{X_i})$ by (A.24) because $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ are equivalent to $\Delta^+_{S^{evo}_i}(\vec{X_i})$ and $\Delta^-_{S^{evo}_i}(\vec{X_i})$ respectively when tuples of $\Delta^+_{S_i}(\vec{X_i})$ do not exist in $S^{evo}_i(\vec{X_i})$ and tuples of $\Delta^-_{S_i}(\vec{X_i})$ exist in $S^{evo}_i(\vec{X_i})$. Therefore, the updated view instance $S_i{}'(\vec{X_i})$ is a result of adding tuples of $\Delta^+_{S_i}(\vec{X_i})$ equivalent to $\Delta^+_{S_i{}^{evo}}(\vec{X_i})$ and deleting tuples of $\Delta^-_{S_i}(\vec{X_i})$ equivalent to $\Delta^-_{S_i{}^{evo}}(\vec{X_i})$ against the original view instance $S_i(\vec{X_i})$ when tuples of $\Delta^+_{S_i}(\vec{X_i})$ do not exist in $S^{evo}_i(\vec{X_i})$ and tuples of $\Delta^-_{S_i}(\vec{X_i})$ exist in $S^{evo}_i(\vec{X_i})$.

Third, it is shown that the updated view instance $S_i{}'(\vec{X_i})$ of source schema is a result of adding tuples of $\Delta^+_{S_i}(\vec{X_i})$ and deleting tuples of $\Delta^-_{S_i}(\vec{X_i})$ against the original view instance $S_i(\vec{X_i})$ even when tuples of $\Delta^+_{S_i}(\vec{X_i})$ exist in $S^{evo}_i(\vec{X_i})$ and tuples of $\Delta^-_{S_i}(\vec{X_i})$ does not exist in $S^{evo}_i(\vec{X_i})$. In such case, as the above shows, $S_i{}^{evo\prime}(\vec{X_i})$ is equivalent to $S_i{}^{evo}(\vec{X_i})$ because the base relation $B_T(\vec{Y})$ and the auxiliary relation $A^c_{S_i}(\vec{X_i})$ are not updated to $B_T{}'(\vec{Y})$ and $A^c_{S_i}{}'(\vec{X_i})$ by $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$.

On the other hand, tuples of $\Delta^+_{S_i}(\vec{X_i})$ existing in $S^{evo}_i(\vec{X_i})$ are deleted from the auxiliary relation $A^{gain}_{S_i}(\vec{X_i})$ if they exist in $A^{gain}_{S_i}(\vec{X_i})$. Thus, such tuples of $\Delta^+_{S_i}(\vec{X_i})$ appear in $S_i{}'(\vec{X_i})$ as a result of (A.24). Tuples of $\Delta^-_{S_i}(\vec{X_i})$ not existing in $S^{evo}_i(\vec{X_i})$ are deleted from the auxiliary relation $A^{loss}_{S_i}(\vec{X_i})$ if they exist in $A^{loss}_{S_i}(\vec{X_i})$. Thus, such tuples of $\Delta^-_{S_i}(\vec{X_i})$ do not appear in $S_i{}'(\vec{X_i})$ as a result of (A.24). Therefore, the updated view instance $S_i{}'(\vec{X_i})$ of source schema is a result of adding tuples of $\Delta^+_{S_i}(\vec{X_i})$ and deleting tuples of $\Delta^-_{S_i}(\vec{X_i})$ against the original view instance $S_i(\vec{X_i})$.

From the above, it is shown that a result of the transformation by backward update sharing is equivalent to a result of transformations of updates against the view instance $T(\vec{Y})$ of target schema to the updated view instance $S_i{}'(\vec{X_i})$ of source schema by $put_{t.trg}$ and $get_{t.src.i}$. Thus, backward update sharing is realized by the derived BXs. □

## A.15 Proof of Proposition 5.9

Suppose a co-existence strategy satisfying the *consistency* of updates between relations $S_i(\vec{X_i})$ ($i \in [1, n]$) of source schema and a relation $T(\vec{Y})$ of target schema. Let $\mathbf{S}$ be union of $S_i(\vec{X_i})$ for all $i$. Let $BX_{src.i}$ for all $i$ and $BX_{trg}$ be the derived bidirectional transformations from the co-existence strategy by Algorithm 4.3 and Algorithm 4.2 respectively. Let $BX_{t.src.i}$ for all $i$ and $BX_{t.trg}$ be the derived bidirectional transformations from the co-existence strategy by Algorithm 5.2 and Algorithm 5.3 respectively.

Suppose we migrate the source-side database $\mathbf{D}_s$ to the target-side database$\mathbf{D}_t$ by the derived BXs. Also suppose that a view instance $T(\vec{Y})$ of target schema is updated by a delta relation $\Delta T(\vec{Y})$ against a previous state $T^{pre}(\vec{Y})$ of target schema and a view instance $S_i(\vec{X_i})$ for each $i$ of source schema is updated from a previous state $S_i^{pre}(\vec{X_i})$ of source schema by sets of inserted and deleted tuples $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$. $\Delta^+_{S_i}(\vec{X_i})$ and $\Delta^-_{S_i}(\vec{X_i})$ are transformed from $\Delta T(\vec{Y})$ by backward update sharing of the co-existence strategy. The delta relation $\Delta T(\vec{Y})$ is union of sets of inserted and deleted tuples $\Delta^+_T(\vec{Y})$ and $\Delta^-_T(\vec{Y})$. Union of the view instances $S_i(\vec{X_i})$ for all $i$ is source schema instance $\mathbf{S}$.

Based on the derived BXs, the view instance $S_i(\vec{X_i})$ of source schema for each $i$ is transformed from the source-side database $\mathbf{D}_s$ by $get_{src.i}$ of $BX_{src.i}$ and the view instance $T(\vec{Y})$ of target schema is transformed from the source-side database $\mathbf{D}_s$ by $get_{trg}$ of $BX_{trg}$. In the first step of data migration, the target-side database is updated by $put_{t.src.i}$ in a stepwise manner. Let $\mathbf{D}_t^{tmp}(i)$ be a target-side database at $i$-th step of data migration as a result of $put_{t.src.i}$ from a pair of the database $\mathbf{D}_t^{tmp}(i-1)$ and the view instance $S_i(\vec{X_i})$, $\mathbf{D}_t^{tmp}(0)$ be empty set, and $\mathbf{D}_t^{tmp}(n)$ be $\mathbf{D}^{tmp}$. For example, $\mathbf{D}_t^{tmp}(0)$ as empty set is updated to $\mathbf{D}_t^{tmp}(1)$ by $put_{t.src.1}$ from a pair of $\mathbf{D}_t^{tmp}(0)$ and $S_1(\vec{X_1})$, $\mathbf{D}_t^{tmp}(1)$ is updated to $\mathbf{D}_t^{tmp}(2)$ by $put_{t.src.2}$ from a pair of $\mathbf{D}_t^{tmp}(1)$ and $S_2(\vec{X_2})$, and eventually $\mathbf{D}_t^{tmp}(n-1)$ is updated to $\mathbf{D}_t^{tmp}(n)$ by $put_{t.src.n}$ from a pair of $\mathbf{D}_t^{tmp}(n-1)$ and $S_n(\vec{X_n})$. $\mathbf{D}_t^{tmp}(n)$ is equivalent to $\mathbf{D}_t^{tmp}$. In the second step of data migration, $put_{t.trg}$ transforms a pair of $\mathbf{D}_t^{tmp}$ and the view instance $T(\vec{Y})$ that is transformed from the source-side database $\mathbf{D}_s$ by $get_{trg}$ into the migrated target-side database $\mathbf{D}_t$. Then $get_{t.src.i}$ of $BX_{t.src.i}$ transforms the migrated target-side database $\mathbf{D}_t$ into the view instance $S_i{}'(\vec{X_i})$ of source schema. $get_{t.trg}$ of $BX_{t.trg}$ transforms $\mathbf{D}_t$ into the view instance $T'(\vec{Y})$ of target schema. Thus, by proving $S_i(\vec{X_i}) = S_i{}'(\vec{X_i})$ and $T(\vec{Y}) = T'(\vec{Y})$, we prove that view instances of source schema and target schema are equivalent when they are computed from the original source-side database and the migrated target-side database based on the derived BXs.

First, we prove $T(\vec{Y}) = T'(\vec{Y})$. $T'(\vec{Y})$ is a result of $get_{t.trg}$ from the migrated target-side database $\mathbf{D}_t$. $\mathbf{D}_t$ is a result of $put_{t.trg}$ from a pair of $\mathbf{D}_t^{tmp}$ and the view instance $T(\vec{Y})$. Such transformation to $T'(\vec{Y})$ is expressed as follows:

$$T'(\vec{Y}) = get_{t.trg}(put_{t.trg}(\mathbf{D}_t^{tmp}, T(\vec{Y})))$$

Since Proposition 5.6 says $get_{t.trg}$ and $put_{t.trg}$ of $BX_{t.trg}$ satisfy PutGet, $T(\vec{Y}) = T'(\vec{Y})$.

Second, we prove $S_i(\vec{X_i}) = S_i{}'(\vec{X_i})$ for each $i$. Let $S_i{}^{tmp}(\vec{X_i})$ be a result of $get_{t.src.i}$ from $\mathbf{D}_t^{tmp}(i)$. Such transformation to $S_i{}^{tmp}(\vec{X_i})$ is expressed as follows:

$$\begin{aligned} S_i{}^{tmp}(\vec{X_i}) &= \{S_i{}^{tmp}(\vec{X_i}) \text{ is a result of } get_{t.src.i} \text{ from the target-side database } \mathbf{D}_t^{tmp}(i)\} \\ &\quad get_{t.src.i}(\mathbf{D}_t^{tmp}(i)) \\ &= \{\mathbf{D}_t^{tmp}(i) \text{ is a result of } put_{t.src.i} \text{ from a pair of } \mathbf{D}_t^{tmp}(i-1) \text{ and the view} \\ &\quad \text{instance } S_i(\vec{X_i}))\} \\ &\quad get_{t.src.i}\Big(put_{t.src.i}(\mathbf{D}_t^{tmp}(i-1), S_i(\vec{X_i}))\Big) \end{aligned}$$

Since Proposition 5.3 says $get_{t.src.i}$ and $put_{t.src.i}$ of $BX_{t.t.src.i}$ satisfy PutGet as follows:

$$S_i{}^{tmp}(\vec{X_i}) = S_i(\vec{X_i}) \tag{A.25}$$

On the other hand, in a process of data migration, $put_{t.src.1}$ at 1st step makes the updated base relation $B_T{}'(\vec{Y})$ to be a result of schema evolution $f(\mathbf{S})$ from $\mathbf{D}_t^{tmp}(0)$ as union of empty set and $\mathbf{S}$ as union of the view instance $S_i(\vec{X_i})$ of source schema for all $i$ as follows:

$$B_T{}'(\vec{Y}) = \{\text{Definition in Algorithm 5.2}\}$$

$$
\begin{aligned}
&(B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y})\\
=\ &\{\text{Substitute definitions of } \Delta^+_{B_T}(\vec{Y}) \text{ and } \Delta^-_{B_T}(\vec{Y}) \text{ in Algorithm 5.2}\}\\
&\left(B_T(\vec{Y}) \cap \neg(\neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y}) \cap B_T(\vec{Y}))\right)\\
&\ \cup \left(T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \cap \neg B_T(\vec{Y})\right)\\
=\ &\{\text{Substitute } B_T(\vec{Y}) = \emptyset\}\\
&T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y})\\
=\ &\{\text{Since the view instance of source schema are empty set and updated to}\\
&S_i(\vec{X_i}) \text{ for all } i \text{ at 1st step, } T^{evo}(\vec{Y}) = \emptyset \text{ and } T^{evo\prime}(\vec{Y}) = f(\mathrm{S}) \text{ by definition in}\\
&\text{Algorithm 5.2 }\}\\
&f(\mathrm{S})
\end{aligned}
$$

By turning the updated base relation $B_T{}'(\vec{Y})$ to the current base relation $B_T(\vec{Y}) = f(\mathrm{S})$ of $\mathbf{D}^{tmp}_t(1)$, $put_{t.src.2}$ at 2nd step makes the updated base relation $B_T{}'(\vec{Y})$ to be a result of schema evolution $f(\mathrm{S})$ as follows:

$$
\begin{aligned}
B_T{}'(\vec{Y}) =\ &\{\text{Definition in Algorithm 5.2}\}\\
&(B_T(\vec{Y}) \cap \neg\Delta^-_{B_T}(\vec{Y})) \cup \Delta^+_{B_T}(\vec{Y})\\
=\ &\{\text{Substitute definitions of } \Delta^+_{B_T}(\vec{Y}) \text{ and } \Delta^-_{B_T}(\vec{Y}) \text{ in Algorithm 5.2}\}\\
&\left(B_T(\vec{Y}) \cap \neg(\neg T^{evo\prime}(\vec{Y}) \cap T^{evo}(\vec{Y}) \cap B_T(\vec{Y}))\right)\\
&\ \cup \left(T^{evo\prime}(\vec{Y}) \cap \neg T^{evo}(\vec{Y}) \cap \neg B_T(\vec{Y})\right)\\
=\ &\{\text{Substitute } B_T(\vec{Y}) = f(\mathrm{S}), T^{evo}(\vec{Y}) = \emptyset, \text{ and } T^{evo\prime}(\vec{Y}) = f(\mathrm{S})\}\\
&f(\mathrm{S})
\end{aligned}
$$

By turning the updated base relation $B_T{}'(\vec{Y})$ to the current base relation $B_T(\vec{Y}) = f(\mathrm{S})$ of $\mathbf{D}^{tmp}_t(2)$, $put_{t.src.3}$ at 3rd step makes the updated base relation $B_T{}'(\vec{Y})$ to be $f(\mathrm{S})$ in the same manner. Therefore, the base relations of $\mathbf{D}^{tmp}_t(i)$ for each $i$ and $\mathbf{D}^{tmp}_t$ are a result of schema evolution $f(\mathrm{S})$.

Also $put_{t.src.1}$ at 1st step makes the updated base relation $A^{lost\prime}_T(\vec{Y})$ and $A^{gain\prime}_T(\vec{Y})$ to be empty set based on $\mathbf{D}_t^{tmp}(0)$ as union of empty set as follows:

$$\begin{aligned}
A_T^{lost\prime}(\vec{Y}) &= \{\text{Definition in Algorithm 5.2}\}\\
&\quad A_T^{lost}(\vec{Y}) \cap \neg\Delta^-_{A_T^{lost}}(\vec{Y})\\
&= \{\text{Substitute definitions of } \Delta^-_{A_T^{lost}}(\vec{Y}) \text{ in Algorithm 5.2}\}\\
&\quad A_T^{lost}(\vec{Y}) \cap \neg\Big((\Delta^+_{B_T}(\vec{Y}) \cap A_T^{lost}(\vec{Y})) \cup (\Delta^-_{B_T}(\vec{Y}) \cap A_T^{lost}(\vec{Y}))\Big)\\
&= \{\text{Substitute } A_T^{lost}(\vec{Y}) = \emptyset\}\\
&\quad \emptyset\\
A_T^{gain\prime}(\vec{Y}) &= \{\text{Definition in Algorithm 5.2}\}\\
&\quad A_T^{gain}(\vec{Y}) \cap \neg\Delta^-_{A_T^{gain}}(\vec{Y})\\
&= \{\text{Substitute definitions of } \Delta^-_{A_T^{gain}}(\vec{Y}) \text{ in Algorithm 5.2}\}\\
&\quad A_T^{gain}(\vec{Y}) \cap \neg\Big((\Delta^+_{B_T}(\vec{Y}) \cap A_T^{gain}(\vec{Y})) \cup (\Delta^-_{B_T}(\vec{Y}) \cap A_T^{gain}(\vec{Y}))\Big)\\
&= \{\text{Substitute } A_T^{lost}(\vec{Y}) = \emptyset\}\\
&\quad \emptyset
\end{aligned}$$

By turning the updated auxiliary relations $A_T^{lost\prime}(\vec{Y})$ and $A_T^{gain\prime}(\vec{Y})$ as empty set to the current auxiliary relations $A_T^{lost\prime}(\vec{Y})$ and $A_T^{gain\prime}(\vec{Y})$ as empty set, $put_{t.src.2}$ at 2nd step makes the updated auxiliary relations to be empty set again. Therefore, the auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ of $\mathbf{D}_t^{tmp}(i)$ at $i$-th step and $\mathbf{D}_t^{tmp}$ are empty set. Then $get_{t.src.i}$ transforms the base relation as $f(\mathrm{S})$ and $i$-th auxiliary relations $A_{S_i}^{c}(\vec{Z}_i)$, $A_{S_i}^{lost}(\vec{X}_i)$, and $A_{S_i}^{gain}(\vec{X}_i)$ of $\mathbf{D}_t^{tmp}(i)$ into the view instance $S_i^{tmp}(\vec{X}_i)$.

On the other hand, the target-side database $\mathbf{D}_t^{tmp}$ is union of the base relation as $f(\mathrm{S})$, auxiliary relations $A_{S_i}^{c}(\vec{Z}_i)$, $A_{S_i}^{lost}(\vec{X}_i)$, and $A_{S_i}^{gain}(\vec{X}_i)$ that are update by $put_{t.src.i}$ for each $i$, and other auxiliary relations $A_T^{lost}(\vec{Y})$ and $A_T^{gain}(\vec{Y})$ as empty set. Thus, $get_{t.src.i}(\mathbf{D}_t^{tmp}(i))$ and $get_{t.src.i}(\mathbf{D}_t^{tmp})$ is equivalent and $S_i(\vec{X}_i)$ is equivalent to $get_{t.src.i}(\mathbf{D}_t^{tmp})$ as follows:

$$\begin{aligned}
S_i(\vec{X}_i) &= \{(\text{A.25})\}\\
&\quad S_i^{tmp}(\vec{X}_i)\\
&= \{S_i^{tmp}(\vec{X}_i) = get_{t.src.i}(\mathbf{D}_t^{tmp}(i)) \text{ and } get_{t.src.i}(\mathbf{D}_t^{tmp}(i)) = get_{t.src.i}(\mathbf{D}_t^{tmp})\}
\end{aligned}$$

$$get_{t.src.i}(\mathbf{D}_t^{tmp})$$

In the second step of data migration, $put_{t.trg}$ transforms a pair of $\mathbf{D}_t^{tmp}$ and the view instance $T(\vec{Y})$ that is transformed from the source-side database $\mathbf{D}_s$ by $get_{trg}$ into the migrated target-side database $\mathbf{D}_t$. Then, the view instance $S_i{}'(\vec{X}_i)$ is transformed from the migrated $\mathbf{D}_t$ by $get_{t.src.i}$. In order to prove $S_i(\vec{X}_i) = S_i{}'(\vec{X}_i)$, we prove $S_i{}^{tmp}(\vec{X}_i) = S_i{}'(\vec{X}_i)$ because of $S_i{}^{tmp}(\vec{X}_i) = S_i(\vec{X}_i)$ as (A.25) shows.

Sets of newly appeared and disappeared tuples $\Delta_T^{+\prime}(\vec{Y})$ and $\Delta_T^{-\prime}(\vec{Y})$ as a result of schema evolution after backward update sharing from the delta relation $\Delta T(\vec{Y})$ against $T^{pre}(\vec{Y})$ are expressed as followings because source schema instance $\mathbf{S}$ is union of the view instance $S_i(\vec{X}_i)$ for all $i$ that are updated by backward update sharing:

$$\Delta_T^{+\prime}(\vec{Y}) = f(\mathbf{S}) \cap \neg T^{pre}(\vec{Y})$$
$$\Delta_T^{-\prime}(\vec{Y}) = \neg f(\mathbf{S}) \cap T^{pre}(\vec{Y})$$

Since the *consistency* of updates is satisfied, the followings are satisfied.

$$\Delta_T^{+\prime}(\vec{Y}) \subseteq \Delta_T^{+}(\vec{Y})$$
$$\Delta_T^{-\prime}(\vec{Y}) \subseteq \Delta_T^{-}(\vec{Y})$$

Let $\Delta T'(\vec{Y})$ be union of $\Delta_T^{+\prime}(\vec{Y})$ and $\Delta_T^{-\prime}(\vec{Y})$, and $\Delta_{S_i}^{+}{}'(\vec{X}_i)$ and $\Delta_{S_i}^{-}{}'(\vec{X}_i)$ be a result of backward updates sharing from $\Delta T'(\vec{Y})$. Since Datalog rules of backward update sharing are restricted so that predicates corresponding to $\Delta_T^{+}(\vec{Y})$ and $\Delta_T^{-}(\vec{Y})$ are not negated, backward update sharing is monotonic. Thus $\Delta_{S_i}^{+}{}'(\vec{X}_i)$ and $\Delta_{S_i}^{-}{}'(\vec{X}_i)$ are subsets of $\Delta_{S_i}^{+}$ and $\Delta_{S_i}^{-}$ respectively as follows:

$$\Delta_{S_i}^{+}{}'(\vec{X}_i) \subseteq \Delta_{S_i}^{+}(\vec{X}_i)$$
$$\Delta_{S_i}^{-}{}'(\vec{X}_i) \subseteq \Delta_{S_i}^{-}(\vec{X}_i)$$

Since the base relation $B_T(\vec{Y})$ as $f(\mathbf{S})$ and auxiliary relations $A_{S_i}^{c}(\vec{Z}_i)$, $A_{S_i}^{lost}(\vec{X}_i)$, and $A_{S_i}^{gain}(\vec{X}_i)$ of $\mathbf{D}_t^{tmp}$ are a result of $put_{t.src.i}$ based on $S_i(\vec{X}_i)$ applied $\Delta_{S_i}^{+}(\vec{X}_i)$ and $\Delta_{S_i}^{-}(\vec{X}_i)$, these base relations and auxiliary relations are not updated as a result of $put_{t.trg}$ based on $\Delta_{S_i}^{+}{}'(\vec{X}_i)$ and $\Delta_{S_i}^{-}{}'(\vec{X}_i)$. Then, the view instance $S_i^{tmp}(\vec{X}_i)$ transformed from these base relations and auxiliary relations of the $\mathbf{D}_t^{tmp}$ by $get_{t.src.i}$ is equivalent to the view instance $S_i{}'(\vec{X_i})$ transformed from these base relations and auxiliary relations of the migrated $\mathbf{D}_t$ by $get_{t.src.i}$.

$$S_i{}^{tmp}(\vec{X_i}) = S_i{}'(\vec{X_i}) \tag{A.26}$$

(A.25) and (A.26) conclude $S_i(\vec{X_i}) = S_i{}'(\vec{X_i})$ for each $i$.

Suppose we migrate a target-side database $\mathbf{D}_t$ to a source-side database$\mathbf{D}_s$ by the derived BXs. A view instance $S_i(\vec{X_i})$ for each $i$ of source schema and a view instance of $T(\vec{Y})$ of target schema are transformed from the original target-side database $\mathbf{D}_t$ by $get_{t.src.i}$ of $BX_{t.src.i}$ and $get_{t.trg}$ of $BX_{t.trg}$ respectively. A view instance $S_i{}'(\vec{X_i})$ for each $i$ of source schema and a view instance of $T'(\vec{Y})$ of target schema are transformed from the migrated source-side database $\mathbf{D}_s$ by $get_{src.i}$ of $BX_{src.i}$ and $get_{trg}$ of $BX_{trg}$ respectively. Thus, by proving $S_i{}'(\vec{X_i}) = S_i(\vec{X_i})$ and $T'(\vec{Y}) = T(\vec{Y})$, we prove that view instances of source schema and target schema are equivalent when they are computed from the migrated source-side database and the original target-side database based on the derived BXs.

First, we prove $T'(\vec{Y}) = T(\vec{Y})$. $T'(\vec{Y})$ is a result of $get_{trg}$ from the migrated source-side database $\mathbf{D}_s$. $\mathbf{D}_s$ is a result of $put_{trg}$ from a pair of $\mathbf{D}_s^{tmp}$ and a view instance $T(\vec{Y})$ where $\mathbf{D}_s^{tmp}$ is a result of $put_{src.i}$ from a pair of the source-side database as empty set and view instances $S_i(\vec{X_i})$ for all $i$. Such transformation to $T'(\vec{Y})$ is expressed as follows:

$$T'(\vec{Y}) = get_{trg}(put_{trg}(\mathbf{D}_s^{tmp}, T(\vec{Y})))$$

Proposition 4.4 says $get_{trg}$ and $put_{trg}$ of $BX_{trg}$ satisfy PutGet. Thus $T(\vec{Y}) = T'(\vec{Y})$.

Second, we prove $S_i(\vec{X_i}) = S_i{}'(\vec{X_i})$ for each $i$. In the first step of the data migration, the view instance $S_i(\vec{X_i})$ is transformed to the corresponding base relation $B_{S_i}(\vec{X_i})$ of the source-side database $\mathbf{D}_s^{tmp}$ as identity mapping. In the second step of the data migration, $\mathbf{D}_s^{tmp}$ is transformed to $\mathbf{D}_s$ by $put_{trg}$ from the view instance $T(\vec{Y})$. However, $put_{trg}$ does not update the base relations corresponding to view instances $S_i(\vec{X_i})$ for each $i$ because $S_i(\vec{X_i})$ is already a result of backward update sharing from the delta relation $\Delta T(\vec{Y})$ to make the view instance $T(\vec{Y})$. The view instance $S_i{}'(\vec{X_i})$ is computed from the base relations of $\mathbf{D}_s$ by $get_{src.i}$ while the base relations are not updated from base relations of $\mathbf{D}_s^{tmp}$. Thus $S_i(\vec{X_i}) = S_i{}'(\vec{X_i})$ for each $i$. □