Rewriting of Plain SO Tgds into Nested Tgds

Rihan Hai
RWTH Aachen University, Germany
hai@dbis.rwth-aachen.de

Christoph Quix
Hochschule Niederrhein & Fraunhofer FIT, Germany
christoph.quix@hs-niederrhein.de

ABSTRACT

Schema mappings express the relationships between sources in data interoperability scenarios and can be expressed in various formalisms. Source-to-target tuple-generating dependencies (s-t tgds) can be easily used for data transformation or query rewriting tasks. Second-order tgds (SO tgds) are more expressive as they can also represent the composition and inversion of s-t tgds. Yet, the expressive power of SO tgds comes with the problem of undecidability for some reasoning tasks. Nested tgds and plain SO tgds are mapping languages that are between s-t tgds and SO tgds in terms of expressivity, and their properties have been studied in the recent years. Nested tgds are less expressive than plain SO tgds, but the logical equivalence problem for nested tgds is decidable. However, a detailed characterization of plain SO tgds that have an equivalent nested tgd is missing. In this paper, we present an algorithmic solution for translating plain SO tgds into nested tgds. The algorithm computes one or more nested tgds, if a given plain SO tgd is rewritable. Furthermore, we are able to give a detailed characterization of those plain SO tgds for which an equivalent nested tgd exists, based on the structural properties of the source predicates and Skolem functions in the plain SO tgd. In the evaluation, we show that our algorithm covers a larger subset of plain SO tgds than previous approaches and that a rewriting can be computed efficiently although the algorithm has the exponential complexity.

PVLDB Reference Format:
Rihan Hai, Christoph Quix. Rewriting of Plain SO Tgds into Nested Tgds. *PVLDB*, 12(11): 1526-1538, 2019.
DOI: https://doi.org/10.14778/3342263.3342631

1. INTRODUCTION

Schema mappings are high-level specifications that describe the relationship between different schemas, and have been intensively studied for data exchange and data integration. In this work, we focus on the problem of rewriting schema mappings between different classes of mappings to enable reasoning and improve the understandability of mappings. We first give an overview of existing classes of mappings, before we define our research questions.

1.1 Background: Mapping Dependencies

Schema mappings are often expressed as logical sentences, e.g., source-to-target tuple generating dependencies (s-t tgd), which are also known as Global-Local-as-View (GLAV) assertions [22]. A s-t tgd is a first-order (FO) sentence in the form of $\forall \mathbf{x}\ (\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y}\ \psi(\mathbf{x}, \mathbf{y}))$, where $\varphi(\mathbf{x})$ is a conjunction of atomic formulas over the source schemas, and $\psi(\mathbf{x}, \mathbf{y})$ is a conjunction of atoms over the target schema.

By allowing the "nesting" of tgds, we obtain a more powerful mapping language, nested tuple-generating dependencies (nested tgds) [12, 26]. In the Clio system [12], nested GLAV mappings (expressible by nested tgds) are proven to be more accurate for describing the relationship between source and target schemas, and also more efficient for data transformation in data exchange systems. The following FO formula λ is an example of a nested tgd.

$$\forall x_1\ (S_1(x_1) \rightarrow \exists y_1\ (T_1(x_1, y_1) \wedge \forall x_2\ (S_2(x_2) \rightarrow \exists y_2\ T_2(x_2, y_1, y_2)\ )\ )\ )$$

Another well-known mapping language is the class of second-order tgds (SO tgds) [10]. The composition of two GLAV mappings expressed as tgds, may be only expressible as an SO tgd. Moreover, SO tgds are also a good abstraction for data transformation, as they are rules with Skolem functions. Skolem functions have been used earlier (independent of SO tgds) for the creation of object identifiers in the context of practical applications such as data integration, data exchange, and ontology-based data access [19, 20, 8].

An SO tgd is a second-order formula, including existentially quantified function symbols, followed by a conjunction of FO formulas similar to tgds. SO tgds allow nested function terms and equalities between different terms. Besides their semantics (FO vs. SO), the main difference between (nested) tgds and SO tgds is how they express the missing information. For instance, in a data exchange scenario, there might be elements in the target schema without corresponding elements in the source schema. The missing values are represented by existentially quantified variables in (nested) tgds (e.g., y_1, y_2 in λ), whereas SO tgds use function terms.

By forbidding the usage of nested function terms and equalities in SO tgds, we obtain the class of plain SO tgds. They are a good alternative to SO tgds, especially with mapping composition and inversion tasks [3]. The following formula μ is an example of a plain SO tgd, which is logically equivalent to the nested tgd λ.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342631

Source Schema
Department(DeptName)
Group(GroupID, DeptName)
Budget(BudgetID, DeptName)
Target Schema
Department(DeptName, DeptID)
Group(GroupID, DeptID, Leader)
Budget(BudgetID, DeptID, Amount)
(a)

Schema mapping as a plain SO tgd μ_1:

$\exists f_d, f_l, f_a \; \forall n, g, b$

$\sigma_1 : [D(n) \rightarrow D'(n, f_d(n))] \wedge$

$\sigma_2 : [D(n) \wedge G(g, n) \rightarrow G'(g, f_d(n), f_l(n, g))] \wedge$

$\sigma_3 : [D(n) \wedge B(b, n) \rightarrow B'(b, f_d(n), f_a(n, b))]$

(b)

Nested tgd λ_1 :

$\sigma_1 : [\forall n \; D(n) \rightarrow \exists y_d \; D'(n, y_d) \wedge$

$\sigma_2 : [\forall g \; G(g, n) \rightarrow \exists y_l \; G'(g, y_d, y_l)] \wedge$

$\sigma_3 : [\forall b \; B(b, n) \rightarrow \exists y_a \; B'(b, y_d, y_a)]]$

(c)

Figure 1: Running example

$$\mu : \exists f_1, f_2 \; \forall x_1, x_2 \; (S_1(x_1) \rightarrow T_1(x_1, f_1(x_1))) \wedge \\ (S_1(x_1) \wedge S_2(x_2) \rightarrow T_2(x_2, f_1(x_1), f_2(x_1, x_2)))$$

Many existing data exchange and integration applications [27, 12, 23, 7, 1] generate mappings with Skolem functions as grouping keys for set elements, and the specification of their schema mappings can be expressed in plain SO tgds.

There is a strict inclusion hierarchy between the different classes of mappings, i.e., tgds are a strict subset of nested tgds; nested tgds are a strict subset of plain SO tgds; and plain SO tgds are a strict subset of SO tgds. For every nested tgd, we can find an equivalent plain SO tgd (e.g., λ is logically equivalent to μ), but not vice versa.

1.2 Motivation and Problem Definition

Although SO tgds and plain SO tgds have stronger expressive power, they are less desirable specifications in practice compared to dependencies with FO semantics. To begin with, FO and SO logics correspond to algorithmic behaviors with different complexities [21]. For example, in a data exchange system the source data needs to be checked whether it satisfies the given mappings, which corresponds to the problem of model-checking. It is proven that the data complexity of the model checking problem for plain SO tgds is NP-complete while the same problem is merely LOGSPACE for nested tgds [25, 21]. Moreover, some reasoning tasks are undecidable for SO tgds, such as logical equivalence [11], which plays a fundamental role in mapping optimization [9]. The decidability of the logical equivalence problem for plain SO tgds is an open question. Yet, logical equivalence for nested tgds is decidable. In addition, FO tgds are more user-friendly and can be easily translated to SQL [24, 21].

Given schema mappings generated from data exchange applications as plain SO tgds, it would be desirable if we can tell whether there are equivalent nested tgds.

Q1: Given a schema mapping expressed as a plain SO tgd μ, is there a single nested tgd λ or a set of nested tgds Λ logically equivalent to μ?

Simplification of schema mappings with SO semantics has been addressed in various ways [24, 25, 6]. The main intuition of these approaches is to discover an ordering of the arguments of the functions in a plain SO tgd, and replace the function quantifiers with existential variables. This is a common procedure for transforming SO to FO formulas, known as *de-Skolemization.* However, $Q1$ is more complicated than the de-Skolemization problem studied in the aforementioned works. This leads to the problem that although some plain SO tgds have FO semantics, and can be expressed by nested tgds, they cannot be discovered by the existing approaches.

We demonstrate the insufficiency of existing works and motivate our work with the following example.

Example 1.1. Consider the data exchange scenario with source schema and target schema given in Fig. 1a. The source schema with *Department (D)*, *Group (G)*, and *Budget (B)* should be mapped to the target schema with corresponding relations but additional attributes. The plain SO tgd μ_1 in Fig. 1b is the logical representation of such a schema mapping, either generated by mapping systems [12, 16] or defined by a mapping designer [1]. There are certain target values missing in the source schema and need to be represented by Skolem functions. For instance, *DeptID* (d) does not exist in the source schema, thus the Skolem function $f_d(n)$ specifies that the new target instances of d depend on the source instance values of *DeptName* (n). Similarly, $f_l(n, g)$ indicates that *Leader* (l) depends on the source values of department name n and *GroupID* (g), and all the tuples with the same values of n and g should have the identical value for l. Likewise the third Skolem function $f_a(n, b)$ specifies values for *Amount* (a). We use labels σ_1, σ_2 and σ_3 to distinguish the three implications in μ_1.

A direct application of the approaches based on de-Skolemization [24, 25] is not feasible for answering $Q1$ given μ_1. In μ_1, there is an ordering between f_d and f_l since the argument of f_d is a subset of the arguments of f_l. There also exists an ordering between f_d and f_a, but no ordering between f_l and f_a since their argument sets do not have a containment relationship. Thus, μ_1 is not rewritable by [24, 25]. The state-of-the-art approach [6] has proposed a partitioning method that "cuts" the given plain SO tgd into smaller blocks based on the disjoint sets of Skolem functions, then performs the de-Skolemization in each block. Since all three implications in μ_1 contains f_d, [6] considers μ_1 as a whole block and tries to order f_d, f_l, and f_a, which fails for the same reason and gives a negative answer to $Q1$.

Yet, there exists a nested tgd λ_1 (Fig. 1c) which is logically equivalent to μ_1. The reason why existing solutions fail to answer $Q1$ is two-fold. (i) A deeper analysis of structural properties of nested tgds, and a characterization of plain SO tgds which have equivalent nested tgds are missing. Intuitively, plain SO tgds have a 'flat' structure with regard to the implications, e.g., μ_1 is the conjunction of three non-nested implications. When existing works try to construct the new FO mappings, the transformation is limited to flat structures. However, nested tgds are FOs with a specific hierarchical structure, which we elaborate in Sec. 2.2. For instance, λ_1 has a two-level hierarchy with σ_2 and σ_3 nested under σ_1. (ii) As a direct adaption of *de-Skolemization* from mathematical logic, existing works focus on Skolem functions and overlook the remaining components of schema mappings. We will show that the source relations in mappings (e.g., $D(n)$ in σ_1, $D(n) \wedge G(g,n)$ in σ_2 of μ_1) also play an important role in rewriting plain SO tgds to nested tgds.

Therefore, we propose a novel approach which partitions the given plain SO tgd in a "divide and conquer" manner, and tries to discover whether the given plain SO tgd has a tree-like structure similar to nested tgds. For $Q1$, we are also able to find multiple solutions, i.e., several sets of nested tgds equivalent to the input plain SO tgd, which is not discussed in existing works.

Plain SO tgds are more expressive than nested tgds. It implies that there exist plain SO tgds not logically equivalent to any nested tgd, e.g., the below plain SO tgd μ':

$$\exists f_1, f_2 \forall x_1, x_2, x_3 \ (S(x_1,x_2,x_3) \rightarrow T(f_1(x_1,x_2), f_2(x_1,x_3)))$$

It would be convenient to have a set of conditions to differentiate the plain SO tgds that have equivalent nested tgds (e.g., μ), with the ones that have not (e.g., μ').

Q2: How can we characterize the class of plain SO tgds, for which a rewriting to nested tgds is possible?

We will propose a sufficient condition and show that it reveals the native structural characterization of plain SO tgds rewritable to nested tgds, which is more general than the conditions given in [6]. There is also a theoretical tool given in [21] to tell when a plain SO tgd does not have equivalent nested tgds, but it is difficult to be directly applied in an algorithmic approach.

Q3: Can we define and implement an efficient algorithm that is able to translate a relevant subset of plain SO tgds into logically equivalent nested tgds?

We provide a checking condition which can be efficiently implemented. Finally, we need to verify that our algorithms and their implementation are correct and complete with respect to the identified class of rewritable plain SO tgds. We designed an evaluation procedure to address the final research question.

Q4: How can we evaluate the correctness, completeness, and performance of the algorithms to rewrite plain SO tgds into nested tgds?

We summarize our main contributions as below:

- We propose a novel approach for rewriting plain SO tgds into logically equivalent nested tgds. We refine our approach such that it can support more general cases, which covers a wider range of plain SO tgds than the state-of-the-art approach (addressing Q1 and Q3).
- We provide a sufficient condition to tell when plain SO tgds have logically equivalent nested tgds (addressing Q2).
- We design an evaluation procedure, based on configurable generators for nested tgds and plain SO tgds, to process generated mappings with our algorithms. By intensive experiments, we demonstrate the correctness, completeness, and performance of our approach (addressing Q4).

The remainder of the paper is organized as follows: first we introduce preliminary concepts in Sec. 2. Then, we discuss our basic approach in Sec. 3. Some refinements of the algorithms, the theoretical analysis, definition of the sufficient condition, and a complexity analysis are addressed in Sec. 4. The extensive evaluation of our solutions is presented in Sec. 5. Finally, we compare our results with related works (Sec.6) and conclude the paper in Sec. 7.

2. PRELIMINARIES

2.1 Schema and Schema Mapping

A **relational schema R** is a finite sequence of relation symbols $\mathbf{R} = \langle R_1, \dots, R_k \rangle$, where each R_i has a fixed arity. An instance I over $\mathbf{R}$, is a k-tuple $(R_1^I, \dots, R_k^I)$, where each finite relation R_i^I has the same arity as R_i.

Let $\mathbf{S}$ and $\mathbf{T}$ be a source and a target schema sharing no relation symbols. A **schema mapping** $\mathcal{M}$ between $\mathbf{S}$ and $\mathbf{T}$ is a triple $\mathcal{M} = \langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$, where Σ is a set of dependencies over $(\mathbf{S}, \mathbf{T})$. The dependencies Σ can be expressed as logical formulas (e.g., tgds) over the source and target schemas. Let I be an instance and θ be a first-order formula (e.g., tgds, nested tgds), or a second-order formula (e.g., plain SO tgds, SO tgds). We denote I satisfying θ as $I \models \theta$. Let Θ be a finite set of formulas, we use $I \models \Theta$ to denote that $I \models \theta$ holds for every $\theta \in \Theta$. For a quantifier $Q \in \{\forall, \exists\}$ in θ, the *scope* of Q is the range in θ controlled by Q.

Let Λ and Λ' be two finite sets of source-to-target constraints, expressed in a certain class of mapping dependencies. We only consider the case that the set of instances is finite. If for every source instance I and every target instance J such that $(I,J) \models \Lambda$, we have that $(I,J) \models \Lambda'$ then Λ implies Λ', denoted as $\Lambda \models \Lambda'$. Λ and Λ' are **logically equivalent**, iff $\Lambda \models \Lambda'$ and $\Lambda' \models \Lambda$, denoted as $\Lambda \equiv \Lambda'$ [21].

2.2 Nested Tgds

Nested GLAV mappings can be expressed by a finite set of nested tgds. Nested tgds [26] are often considered as the best choice for describing the correlation of mappings [14].

Definition 2.1 ([21]). A **nested tuple-generating dependency** is a first-order sentence that can be generated by the following recursive definition (the variables are partitioned into two disjoint sets X and Y representing the universally and existentially quantified variables, respectively):

$$\phi ::= \alpha \mid \forall \vec{x} \ (\beta_1 \wedge \dots \wedge \beta_k \rightarrow \exists \vec{y} \ (\phi_1 \wedge \dots \wedge \phi_l))$$

where each $x_i \in X$, each $y_i \in Y$, α is an atomic formula over the target schema, and each β_j is an atomic formula over the source schema containing only variables from X such that each x_i occurs in some β_j.

In this work, we use *parts* [21, 14] marked by inline labels σ_i to indicate the nesting order, as shown in Fig. 2. We use square brackets to make the nesting hierarchy clearer. If a part σ_i is nested inside another part σ_j, then we refer to σ_j as the parent of σ_i, and σ_i as a child of σ_j. For instance, $parent(\sigma_2) = parent(\sigma_3) = \sigma_1$ and $child(\sigma_3) = \sigma_4$. With transitive closure of parents, the *ancestor* of a part is defined. For instance, $ancestor(\sigma_4) = \{\sigma_1, \sigma_3\}$. A **pattern** of a nested tgd is a tree with its nodes marked by the inline labels of parts. If two parts in a nested tgd have a parent-child relationship, then there is an edge between the nodes presenting these two parts. The pattern of λ_2 is also shown in the right part of Fig. 2.

2.3 From Nested Tgds to Plain SO Tgds

To prepare for the definition of plain SO tgds, we first introduce the notion of *plain terms*. Given a collection $\mathbf{x}$ of variables and a collection $\mathbf{f}$ of function symbols, a **plain term** (based on $\mathbf{x}$ and $\mathbf{f}$) is defined as: (1) Each variable x in $\mathbf{x}$ is a plain term; (2) Assume f is a k-ary function symbol

$\sigma_1 : [\forall x_1\ S_1(x_1) \rightarrow \exists y_1$

$\quad \sigma_2 : [\forall x_2\ S_2(x_2) \rightarrow R_2(y_1, x_2)] \wedge$

$\quad \sigma_3 : [\forall x_3\ S_3(x_1,x_3) \rightarrow R_3(y_1, x_3) \wedge$

$\qquad \sigma_4 : \forall x_4\ [S_4(x_3, x_4) \rightarrow \exists y_2\ R_4(y_2, x_4)]]]$

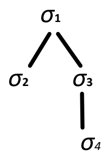


Figure 2: Nested tgd λ_2 and its pattern [21]

in $\mathbf{f}$ of the form $f(u_1, \ldots, u_k)$, where each u_i ($1 \leqslant i \leqslant k$) is a variable in $\mathbf{x}$, then $f(u_1, \ldots, u_k)$ is a plain term.

Definition 2.2 ([3])**.** Given schemas $\mathbf{S}$ and $\mathbf{T}$ with no relation symbols in common, a **plain second-order tgd (plain SO tgd)** from $\mathbf{S}$ to $\mathbf{T}$ is a formula of the form:

$$\exists \mathbf{f}\ (\forall \mathbf{x_1}(\varphi_1 \rightarrow \psi_1)) \wedge\ \ldots\ \wedge \forall \mathbf{x_n}(\varphi_n \rightarrow \psi_n)), \text{ where}$$

(1) each member of $\mathbf{f}$ is a function symbol,
(2) each formula φ_i ($1 \leq i \leq n$) is a conjunction of relational atoms $S(y_1, \ldots, y_k)$, where S is a k-ary relation symbol of schema $\mathbf{S}$ and $y_1, \ldots, y_k$ are (not necessarily distinct) variables in $\mathbf{x_i}$,
(3) each ψ_i is a conjunction of relational atomic formulas over $\mathbf{T}$ mentioning plain terms built from $\mathbf{x_i}$ and $\mathbf{f}$, and
(4) each variable in $\mathbf{x_i}$ ($1 \leq i \leq n$) appears in some relational atom of φ_i.

Skolemization is the process of replacing every existentially quantified variable y in a first-order formula λ with a new Skolem function in the form of $f(\mathbf{x})$ (also referred to as a *Skolem term*), where $\mathbf{x}$ is the vector of universal variables in the scope of y. That is, when performing Skolemization over a nested tgd, the arguments of f are the universal variables from the part where $\exists y$ appears, and its ancestors. For instance, $\exists y_2$ is in the part σ_4 of λ_2, the universal variables from σ_4 and its ancestors are $\langle x_1, x_3, x_4 \rangle$. Thus, y_2 is replaced by the Skolem function $f_2(x_1, x_3, x_4)$. The following example shows the Skolemized nested tgd δ_2, which is the Skolemized form of λ_2.

Example 2.1. δ_2: the Skolemized form of λ_2 [21].

$\sigma_1 : [\forall x_1\ S_1(x_1) \rightarrow \exists f_1$

$\quad \sigma_2 : [\forall x_2\ S_2(x_2) \rightarrow R_2(f_1(x_1), x_2)] \wedge$

$\quad \sigma_3 : [\forall x_3\ S_3(x_1, x_3) \rightarrow R_3(f_1(x_1), x_3) \wedge$

$\qquad \sigma_4 : [\forall x_4\ S_4(x_3, x_4) \rightarrow \exists f_2\ R_4(f_2(x_1, x_3, x_4), x_4)]]]$

A Skolemized nested tgd is not necessarily a plain SO tgd, since it may still contain nested implications, e.g., δ_2. In [14], a **normalization** procedure performed on Skolemized nested tgds is introduced, which is referred to as "nested-to-so". A first-order formula $\varphi \rightarrow (\psi \wedge [\varphi_1 \rightarrow \psi_1])$ can be equivalently rewritten to $[\varphi \rightarrow \psi] \wedge [\varphi \wedge \varphi_1 \rightarrow \psi_1]$. By applying this procedure recursively from the outermost nesting level to the root level of the Skolemized nested tgd, we can obtain its normalized form, as in the following example.

Example 2.2. Plain SO tgd μ_2: the normalized form of δ_2.

$\exists f_1, f_2$

$\sigma_1 : [S_1(x_1) \rightarrow \text{true}] \wedge$

$\sigma_2 : [S_1(x_1) \wedge S_2(x_2) \rightarrow R_2(f_1(x_1), x_2)] \wedge$

$\sigma_3 : [S_1(x_1) \wedge S_3(x_1, x_3) \rightarrow R_3(f_1(x_1), x_3)] \wedge$

$\sigma_4 : [S_1(x_1) \wedge S_3(x_1, x_3) \wedge S_4(x_3, x_4) \rightarrow R_4(f_2(x_1, x_3, x_4), x_4)]$

Note that Skolemization and normalization, as well as their inverse processes de-Skolemization and de-normalization, are transformations which preserve logical equivalence. For instance, λ_2, δ_2 and μ_2 are logically equivalent to each other. Along the lines of nested tgds, we refer to each implication of a plain SO tgd as a *part*. The part σ_1 of μ_2 is a tautology which could be removed, but we keep it for clarity. For simplicity, we often suppress writing the universal quantifiers $\forall \mathbf{x}$ in plain SO tgds.

3. BASIC APPROACH

In this section, we introduce the basic method to rewrite plain SO tgds to logically equivalent nested tgds. It has two steps: (i) *Nest* (Sec. 3.1) that analyses the logical structure of the plain SO tgd and generates a Skolemized nested tgd, and (ii) *DeSkolemization* (Sec. 3.2) which considers the Skolem functions and aims at replacing them with existential variables. We omit some details to focus first on the basic ideas of the approach. In Sec. 4, we will discuss some refinements of the algorithms to make them more general.

3.1 Reverse Logical Normalization

Our approach is based on the observation on Example 2.2. That is, on the left-hand sides of the implications of plain SO tgd μ_2, the subset hierarchy of the sets of relational atoms forms a tree. In the following, $LH(\sigma)$ (resp., $RH(\sigma)$) refers to the set of relational atoms on the left-hand (resp., right-hand) side of a part, e.g., in Example 2.2 $LH(\sigma_1) = S_1(x_1)$. $parts(\mu)$ denotes all parts of a plain SO tgd or nested tgd. Since in μ_2 $LH(\sigma_1) \subset LH(\sigma_2)$ and $LH(\sigma_1) \subset LH(\sigma_3) \subset LH(\sigma_4)$, it implies a tree structure shown in Fig. 2. Thus, we first aim at recognizing this tree structure based on the left-hand sides. Note that some plain SO tgds might have more than one tree structure for their left-hand sides.

Example 3.1. Consider the following plain SO tgd μ_3:

$\sigma_1 : [\ S_1(x_1) \rightarrow T_1(x_1, f_1(x_1))] \wedge$

$\sigma_2 : [\ S_1(x_1) \wedge S_2(x_1, x_2) \rightarrow T_2(x_2, f_1(x_1))] \wedge$

$\sigma_3 : [\ S_3(x_3) \wedge S_4(x_3, x_4) \rightarrow T_3(x_3, f_2(x_3, x_4))]$

This plain SO tgd is equivalent to two nested tgds with $S_1(x_1)$ and $S_3(x_3) \wedge S_4(x_3, x_4)$, respectively, at the root levels. The corresponding pattern has two unconnected subgraphs: $\sigma_1 \rightarrow \sigma_2$, and σ_3 as a separate node.

Thus, our nesting approach for plain SO tgds has to take into account that **multiple** nested tgds may have to be created. Our algorithm *Nest* (cf. Alg. 1) works recursively. Its input is a *parent* part (for recursive calls, *null* in the initial invocation) and a plain SO tgd separated into its parts. The input plain SO tgd is also checked for consistency [6, 24], which means that Skolem functions can be used only with the same list of arguments, and variables may not be repeated in the argument list of a Skolem function. Plain SO tgds generated by Skolemization procedure in Sec. 2.3 are consistent; for inconsistent tgds, the replacement of Skolem terms with existential variables would not be possible. We have implemented the method checking consistency before calling Alg. 1, which is not the focus of this work.

In Alg. 1, we first take an arbitrary part p_i (we will see in an example in the next section, that this choice might make a difference, and that we backtrack over this choice to find all alternative results). The sets C and S separate the parts

Algorithm 1: Nest

```
Input: A parent part $parent$, and a set of parts
       $P = \{p_1, \dots, p_n\}$ of a consistent plain SO tgd
Output: A set of Skolemized nested tgds, identified
by the root parts
1  if $P = \emptyset$ then Return $\emptyset$ // Stop recursion
2  $lh \leftarrow LH(p_i)$ for an arbitrary $p_i \in P$
3  $C, S \leftarrow \emptyset$ // Children and sibling parts
4  $p_r \leftarrow null$
5  foreach $p \in P$ do
6      if $lh \cap LH(p) \neq \emptyset$ then
7          $lh \leftarrow lh \cap LH(p)$; $C \leftarrow C \cup \{p\}$
8          if $lh = LH(p)$ then $p_r \leftarrow p$
9      else $S \leftarrow S \cup \{p\}$
   // If $p_r$ cannot be the root of the new subtree, create a
      new part $p_r$ with empty right-hand side as root
10 if $lh \neq LH(p_r)$ then $p_r \leftarrow [lh \rightarrow true]$
11 if $parent \neq null$ then add $p_r$ to $parent.children$
12 $C \leftarrow C \backslash \{p_r\}$
13 $Nest(p_r, PrepareChildren(C, p_r))$
   // Nest remaining parts as siblings & return all roots
14 return $Nest(parent, S) \cup \{p_r\}$
```

into two disjoint sets: C holds parts which have overlapping antecedents with p_i and will be an element in the subtree rooted at p_r; S has the parts which will be represented in sibling trees of the tree rooted at p_r (note that in a recursive call, p_r and its siblings have the same parent).

The separation of the parts in C and S is controlled by the check in line 6 which checks whether there is an overlap on the left-hand sides of the parts in C so far (lh is the intersection of all $LH(p)$ for each part p in C) and the current part. If this is not the case, then the current part p will be added to S, otherwise it will be added to C. The condition in line 8 checks whether the current part p can be the root of the subtree for all parts in C.

After all parts have been processed, we need to check whether p_r still represents the intersection of all parts in C, and, thus, can be used as the root of the new tree. If that is not the case, then we need to create a new 'virtual' root part with an empty right-hand side (line 10).

We add p_r to the children of the given parent, and call the algorithm recursively with p_r as parent and the rest parts in C (obtained by removing p_r from C in line 12) as children. The procedure *PrepareChildren* removes the left-hand side of p_r from the left-hand sides of the parts in C. The final step (line 14) then processes the sibling nodes.

Example 3.2. Suppose we apply the algorithm to μ_3 from Example 3.1. Let's assume, we randomly take σ_2 as the initial part, and take σ_1 as the first element p in the for-loop. σ_1 will be considered as the root element p_r, as $LH(\sigma_1) = LH(\sigma_2) \cap LH(\sigma_1)$. σ_3 will be inserted into S as it does not have an overlap with σ_1. The recursive call in line 13 is $Nest(\sigma_1, \{[S_2(x_1, x_2) \rightarrow T_2(x_2, f_1(x_1))]\})$, i.e., $S_1(x_1)$ has been removed in σ_2. This will result in a nested tgd δ_{31} with the pattern tree $\sigma_1 \rightarrow \sigma_2$. The processing of σ_3 with the recursive call in line 14 will lead to a second Skolemized 'nested' tgd δ_{32}.

$$\begin{aligned}\delta_{31}: \quad & [S_1(x_1) \rightarrow T_1(x_1, f_1(x_1)) \wedge \\ & \qquad [S_2(x_1) \rightarrow T_2(x_2, f_1(x_1))]] \\ \delta_{32}: \quad & S_3(x_3) \wedge S_4(x_4) \rightarrow T_3(x_3, f_2(x_3, x_4))\end{aligned}$$

Algorithm 2: DeSkolemization

```
Input: A Skolemized nested tgd $\delta$
Output: A nested tgd $\lambda$ if successful; $null$ if failed
1  Initialization: $\lambda \leftarrow \delta$
2  foreach Skolem term $f \in \lambda$ do
3      $replaced \leftarrow false$
4      $p \leftarrow FirstPart(f)$
5      $P \leftarrow Ancestors(p) \cup \{p\}$
6      foreach $p_i \in P$ do
7          $X \leftarrow GetUniversalVars(p_i)$
8          if $Arg(f) = X$ then
9              replace every appearance of $f$ in $\lambda$ with $y_f$
10             append $y_f$ to the exist. variable list of $p_i$
11             $replaced \leftarrow true$; break
12     if $replaced = false$ then return $null$
13 return $\lambda$
```

3.2 Reverse Skolemization

Since the result of the *Nest* procedure might still contain Skolem functions, these have to be replaced with existential variables. Alg. 2 performs the *de-Skolemization.* Its input is a single Skolemized nested tgd δ generated by Alg. 1. Alg. 2 replaces the Skolemized nested tgd δ with Skolem functions into a nested tgd with existential variables. To illustrate the algorithm, we use the following example which is known to have a logically equivalent nested tgd [14].

Example 3.3. Given is a plain SO tgd μ_4 which describes the schema mapping between source schemas *Department* (D), *Group* (G), *Employees* (E) and the corresponding target schemas. Variables d, g, e are the keys of each relation.

$$\begin{aligned}&\exists f_d, f_g \\ &\sigma_1: [D(d) \rightarrow D'(f_d(d))] \wedge \\ &\sigma_2: [D(d) \wedge G(d,g) \rightarrow G'(f_d(d), f_g(d,g))] \wedge \\ &\sigma_3: [D(d) \wedge G(d,g) \wedge E(d,g,e) \rightarrow E'(f_d(d), f_g(d,g), e)]\end{aligned}$$

The *Nest* algorithm will produce the following output δ_4.

$$\begin{aligned}&\exists f_d, f_g \\ &\sigma_1: D(d) \rightarrow D'(f_d(d)) \wedge \\ &\quad \sigma_2: [G(d,g) \rightarrow G'(f_d(d), f_g(d,g)) \wedge \\ &\qquad \sigma_3: [E(d,g,e) \rightarrow E'(f_d(d), f_g(d,g), e)]]\end{aligned}$$

After initialization, for each Skolem term f in λ the procedure $FirstPart(f)$ finds the part p where f first appears. For instance, in Example 3.3 f_d appears in all three parts σ_1, σ_2 and σ_3 of δ_4. However, its corresponding existential variables should belong to only one part in a nested tgd, i.e., the first part where it appears in the pattern tree. In δ_4, the first part in the hierarchy where f_d appears is σ_1. Similarly, the first part of f_g is σ_2. Recall that when we Skolemize a nested tgd, the Skolem function arguments are the set of universal variables from the ancestors and current part of the existential variable. Therefore, to transform a Skolem function f back to an existential variable, we need to check the universal variables of the part where it first appears (p in line 4) and all ancestors of this part ($Ancestors(p)$), whose union we denote as P in line 5. We iterate each part p_i in P, and check its universal variables by $GetUniversalVars(p_i)$ in line 7. When we find a part p_i with the same set of universal variables as the arguments of f (if-statement in line

8), we can replace every occurrence of f in λ with a new existential variable y_f, which is also inserted into the existential variable list of p_i. For δ_4 we replace $f_d(d)$ with y_d in σ_1 as d is the only universal variable in σ_1, and we replace $f_g(d,g)$ with y_g in σ_2 as d and g are the universal variables of σ_2. If such a replacement is impossible, then a nested tgd cannot be generated and the algorithm fails.

Example 3.4. The output of Alg. 2 applied to δ_4 from Example 3.3 (for clarity, now we include all quantifiers).

$$\begin{aligned}\sigma_1 : [\forall d\ D(d) \rightarrow \exists y_d\ D'(y_d) \wedge \\ \sigma_2 : [\forall g\ G(d,g) \rightarrow \exists y_g\ G'(y_d, y_g) \wedge \\ \sigma_3 : [\forall e\ E(d,g,e) \rightarrow E'(y_d, y_g, e)]\ ]\ ]\end{aligned}$$

4. EXTENSION AND ANALYSIS OF THE APPROACH

In this section, we will extend the basic algorithms, study their theoretical properties, and give a better characterization of the subset of plain SO tgds, which are rewritable into a single nested tgd or a set of nested tgds.

4.1 Extension of Nest Algorithm

We consider first the *Nest* algorithm. It may generate different results because we take initially an arbitrary part p_i. Depending on the choice made, the resulting 'Skolemized' nested tgds might have different structures. Fig. 3 shows an example. The input plain SO tgd μ_5 in Fig. 3a have four parts with only pairwise overlaps.

Fig. 3b, 3c and 3d show the possible results which are generated by our approach. For simplicity, we just show the relation names and omit the variables. Each solution is a pair of nested tgds. In one nested tgd of each pair, the root part is either S_1, S_2, or S_3, depending on the sequence in which the parts are processed (i.e., the arbitrary pick in line 2 and the order of elements in the for-loop in line 5 of Alg. 1). All results are obtained in principle by applying classical boolean transformations, e.g., $A \rightarrow (B \wedge (C \rightarrow D))$ is equivalent to $(A \rightarrow B) \wedge (A \wedge C \rightarrow D)$, but they are not logically equivalent as we ignore the Skolem functions.

Thus, to have correct results we also have to verify the correct handling of Skolem functions in the transformation procedure. Separating the parts of a plain SO tgd in the *Nest* algorithm into the sets C and S results in multiple nested tgds as output (as in Example 3.2). If two nested tgds have the same Skolem term, then this set of nested tgds is not a valid rewriting. In the example of Fig. 3, this applies to $\{\lambda_{31}, \lambda_{32}\}$ in Fig. 3d which share f_1.

In the approach presented in [6], the concept of *maximal partitions* is used to describe a similar situation. A plain SO tgd is partitioned into several blocks with disjoint sets of Skolem functions. In our approach, possible partitions are created implicitly by the *Nest* procedure. To verify that the partitions handle Skolem functions correctly, we defined a 'main' procedure *BuildNestedTGD* (cf. Alg. 3).

After preparing the input and invoking the *Nest* algorithm, in line 7 we test whether there is an overlap in Skolem functions of Skolemized nested tgds in one result set (the function $skf(\delta)$ returns all Skolem terms in δ). In that case, we skip the current solution $\mathbf{N}$ and generate a new solution with *Nest*, if possible. For instance, in Fig. 3d λ_{31} and λ_{32} share f_1, thus they are not included in the final result. Please note that the implementation of the algorithms which we evaluated in Sec. 5 is more efficient, as we sort the parts by the number of source relations, and handle the part with the lowest number of source relations first. The optimizations on implementation level would make the algorithm harder to understand; thus, we use a simpler way to present the idea of the algorithm. We also check the result of the *DeSkolemization* algorithm to see whether the Skolem functions could be correctly replaced by existential variables. The final result $\mathbf{R}$ of *BuildNestedTGD* may contain multiple solutions, i.e., each solution is a set of nested tgds $\Lambda = \{\lambda_1, \ldots, \lambda_n\}$, and Λ is logically equivalent to the input μ, which we formalize as the following theorem.[1]

Algorithm 3: BuildNestedTGD

```
   Input: A consistent plain SO tgd $\mu$
   Output: A set of solutions, each solution is a set of
   nested tgds; or $\emptyset$ in case of failure
1  $P \leftarrow$ all parts of $\mu$
2  $\mathbf{R} \leftarrow \emptyset$ // Result set
3  while not all possibilities have been considered in
    Nest do
4      $\mathbf{N} \leftarrow Nest(null, P)$
5      $\mathbf{N}' \leftarrow \emptyset$
6      foreach $\delta \in \mathbf{N}$ do
7          if $\exists \delta' \in \mathbf{N}$ with
            $\delta \neq \delta'$ and $skf(\delta) \cap skf(\delta') \neq \emptyset$ then
8              Skip solution N and generate next result
                for Nest
9          $\lambda \leftarrow DeSkolemization(\delta)$
10         if $\lambda \neq null$ then Add $\lambda$ to $\mathbf{N}'$
11         else Skip solution N and generate next result
            for Nest
12     Add $\mathbf{N}'$ to $\mathbf{R}$
13 return R
```

Theorem 1. *Given a plain SO tgd μ, if* BuildNestedTGD(μ) *produces a nonempty result $\mathbf{R} = \{\Lambda_1, \ldots, \Lambda_m\}$, then each solution in $\mathbf{R}$ is logically equivalent to the input μ, i.e., $\forall \Lambda_i \in \mathbf{R},\ \Lambda_i \equiv \mu$.*

Summary. Theorem 1 answers the question $Q1$ raised in Sec. 1. That is, *BuildNestedTGD* produces correct solutions which are logically equivalent to the given plain SO tgd μ. Notably, it also reveals the possibility that there may exist multiple correct solutions, which is **not** studied in previous works. For instance, in Fig. 3 the final results are two solutions $\Lambda_1 = \{\lambda_{11}, \lambda_{12}\}$ in Fig. 3b and $\Lambda_2 = \{\lambda_{21}, \lambda_{22}\}$ in Fig. 3c with both equivalent to μ_5. This gives the possibility to choose the 'better' solution, e.g., in terms of performance for mapping execution or in terms of understandability. We did not yet examine this issue in detail, but the nesting depth (height of pattern tree) and the size of nested tgds are good indicators for their complexity.

4.2 Rewritability Conditions

In this section, we provide the formal checking condition to tell whether a given plain SO tgd has an equivalent set of nested tgds. We first give a deeper analysis of our approach *BuildNestedTGD*.

[1]The proofs of Theorem 1, Theorem 2, and Theorem 3: `http://dbis.rwth-aachen.de/cms/staff/hai/proof`

$\exists f_1, f_2$
$\sigma_1 : [S_1(x_1) \rightarrow T_1(x_1, f_1(x_1))] \wedge$
$\sigma_2 : [S_1(x_1) \wedge S_2(x_1, x_2) \rightarrow T_2(x_2, f_1(x_1))] \wedge$
$\sigma_3 : [S_2(x_1, x_2) \wedge S_3(x_1, x_3) \rightarrow T_3(x_3, f_2(x_1, x_2, x_3))] \wedge$
$\sigma_4 : [S_1(x_1) \wedge S_3(x_1, x_3) \rightarrow T_4(x_1, x_3)]$

(a)

$\lambda_{11} : [S_1 \rightarrow T_1 \wedge$
$[S_2 \rightarrow T_2] \wedge$
$[S_3 \rightarrow T_4]\ ]$

$\lambda_{12} : S_2 \wedge S_3 \rightarrow T_3$

(b)

$\lambda_{21} : [S_1 \rightarrow T_1 \wedge$
$[S_2 \rightarrow T_2]\ ]$
$\lambda_{22} : [S_3 \rightarrow$
$[S_2 \rightarrow T_3] \wedge$
$[S_1 \rightarrow T_4]\ ]$

(c)

$\lambda_{31} : [S_2 \rightarrow$
$[S_1 \rightarrow T_2] \wedge$
$[S_3 \rightarrow T_3]\ ]$
$\lambda_{32} : [S_1 \rightarrow T_1 \wedge$
$[S_3 \rightarrow T_4]\ ]$

(d)

Figure 3: (a) An input plain SO tgd μ_5; (b)-(d) three candidate results of *Nest*

As discussed, we require that the input to *BuildNested-TGD* is a *consistent* plain SO tgd [24, 6]. Consistency requires that every appearance of the same function has the same arguments, and no variables in the argument list are repeated. This condition is also required in general second-order logic for de-Skolemization [13].

Hierarchical Ordering. Our goal is to answer the question $Q2$, i.e., check when a plain SO tgd can be "nested back" to nested tgds. A plain SO tgd μ has a set of parts, and for each part there are source relations in the left-hand (LH) of the implication and target relations in the right-hand (RH) of the implication. If μ can be equivalently transformed into a nested tgd with a tree pattern, then μ itself should possess certain properties in its left-hand side and right-hand side. More specifically, the source relations of a plain SO tgd should form a tree structure. Moreover, the Skolem function arguments should also form a corresponding tree. For instance, we can observe that in Example 2.2 the containment of Skolem function arguments corresponds to the tree pattern in Fig. 2.

A natural question is whether we should rely on left-hand side (source relation overlaps) or right-hand side (Skolem function argument inclusion) to build the pattern from the given plain SO tgd. The *Nest* algorithm detects a tree-structured pattern from left-hand side of a plain SO tgd. The reason why we have chosen left-hand side to build the tree structure rather than relying on Skolem functions is as follows. Skolem functions can be used either as a replacement for unknown values on the target side, or to structure and group the data on the target side. The former type of Skolem functions are not really helpful to identify the nesting structure, while the latter might not be available in the case of flat relational data. Nevertheless, after discovering the tree pattern using source relations, we still need to check whether the obtained pattern confirms to the right-hand side of the given plain SO tgd. That is, for μ to be qualified to be transformed into a nested tgd, the sets of arguments of Skolem functions should also form a tree, whose structure is 'compatible' with the discovered pattern of the left-hand side. Alg. 2 checks this condition, i.e., whether the arguments of a Skolem function are equal to the universal variables in some node (part) in the tree.

In what follows we summarize the above discussion, and provide the formalizations for our rewritability conditions. First, we characterize the result of the *Nest* procedure. As the procedure partitions and structures the left-hand sides of a plain SO tgd, we call the result *LH-Trees.*

Definition 4.1 (**LH-Trees**)**.** Let μ be a plain SO tgd with parts $\sigma_1, \ldots, \sigma_n$ and relational atoms $S_1, \ldots, S_m$ on the left-hand sides of its parts. The *LH-Trees* of μ is a set of trees with the relational atoms as nodes. For each part σ_i in μ with the sets of relational atoms $\mathbf{S_{i1}}, \ldots, \mathbf{S_{ik}}$, there exists a path $\mathbf{S_{i1}}, \ldots, \mathbf{S_{ik}}$ in a tree from the root $\mathbf{S_{i1}}$ to the node $\mathbf{S_{ik}}$.

For example, Fig. 4 shows the LH-trees of μ_5 to produce λ_{11} and λ_{12} in Fig. 3b. t_1 is built from σ_1, σ_2, σ_4; σ_1 corresponds to the root node of t_1 while σ_2 and σ_4 correspond to the two paths from root node S_1 to the two leaf nodes S_2 and S_3 respectively. Note that a node of LH-tree can contain the conjunction of multiple relational atoms. For instance, t_2 built from σ_3 of μ_5 has one root node $\mathbf{S_{31}} = \{S_2, S_3\}$.

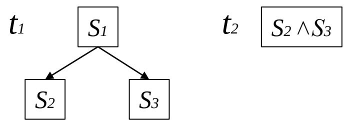


Figure 4: LH-trees for λ_{11} and λ_{12} in Fig. 3b

It is easy to see, that the result of the *Nest* algorithm can be considered as LH-trees. As discussed above, considering the left-hand sides alone is not sufficient for checking the rewritability. Therefore, we add also the Skolem functions to the LH-Trees, and get *LH-Skolem-Trees*.

Definition 4.2 (**LH-Skolem-Trees**)**.** Let μ be a plain SO tgd and $T = \{t_1, \ldots, t_n\}$ be the set of LH-Trees of μ. The *LH-Skolem-Trees* for μ are obtained from T as below:

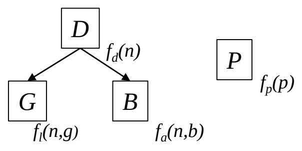
- If σ_i is a part in μ with the sets of relational atoms $\mathbf{S_{i1}}, \ldots, \mathbf{S_{ik}}$ and a Skolem term $f(\mathbf{x})$ appearing on its right-hand side, and $\mathbf{S_{i1}}, \ldots, \mathbf{S_{ik}}$ is a path in a LH-tree $t \in T$, then $f(\mathbf{x})$ is attached to the node $\mathbf{S_{ik}}$.
- If the Skolem term $f(\mathbf{x})$ is attached to some node $\mathbf{S}$ in a tree, then it will be removed from all nodes in the sub-tree rooted at $\mathbf{S}$, i.e., we just keep the first occurrence of a Skolem term in the tree.

Example 4.1. Please consider the plain SO tgd μ_1 in Fig. 5a (continued from the running example, but we added a new relation P for *Project*, p for *ProjectName* and f_p for *ProjectID*), the corresponding LH-Skolem-trees are shown in the middle. The plain SO tgd μ_1 can be divided into two blocks: $\{\sigma_1, \sigma_2, \sigma_3\}$ as they overlap in the relational atom $D(n)$, and $\{\sigma_4\}$ which is the only part with $P(p)$. The Skolem term $f_d(n)$ is attached only to D as it appears in σ_1 (the occurrences attached to G and B are removed as stated in the definition). The LH-Skolem-tree for σ_4 has only one node P as there is only one relational atom in that part.

The result of *Nest* algorithm is shown in Fig. 5c. It can be seen to have the same structure as LH-Skolem-trees of μ_1 in Fig. 5b, i.e., each (Skolemized) nested tgd corresponds to one tree.

$$\sigma_1 : [D(n) \to D'(n, f_d(n))] \wedge$$
$$\sigma_2 : [D(n) \wedge G(g,n) \to G'(g, f_d(n), f_l(n,g))] \wedge$$
$$\sigma_3 : [D(n) \wedge B(b,n) \to B'(b, f_d(n), f_a(n,b))] \wedge$$
$$\sigma_4 : [P(p) \to P'(p, f_p(p)]$$

(a)



(b)

$$\delta_{61} : [D(n) \to D'(n, f_d(n)) \wedge$$
$$[G(g,n) \to G'(g, f_d(n), f_l(n,g))] \wedge$$
$$[B(b,n) \to B'(b, f_d(n), f_a(n,b))]]$$
$$\delta_{62} : [P(p) \to P'(p, f_p(p)]$$

(c)

Figure 5: (a) Extended plain SO tgd μ_1 from Fig. 1b; (b) LH-Skolem-Trees of μ_1; (c) Result of *Nest*

Now, we also have to verify whether the Skolem functions are correctly placed in the LH-Skolem-tree, i.e., whether there is a hierarchical order of the Skolem functions. The idea of *linearity* was proposed in [6] and states that in each new block, the sets of arguments of all Skolem Functions can be ordered linearly, i.e., should be contained in each other. For the plain SO tgd μ_1 in Fig. 5a, $Arg(f_d) \subset Arg(f_l)$, $Arg(f_d) \subset Arg(f_a)$, but $Arg(f_l) \not\subseteq Arg(f_a)$ and $Arg(f_a) \not\subseteq Arg(f_l)$; thus, μ_1 does not satisfy *linearity* and cannot be rewritten according to [6]. The reason is that the approach of [6] tries to find whether the given plain SO tgds have a 'linear' structure, which would be a criteria for tgds. Yet, as we have shown in Sec. 2.2, nested tgds have a tree structure. This is the root cause why linearity is too limited for finding the plain SO tgds with logically equivalent nested tgds.

Observing the example of Fig. 5a, which can be rewritten into two nested tgds, we need to verify only argument lists of Skolem functions in a path of the tree from the root to a leaf node, instead of the whole tree (or block). Since a given plain SO tgd may have multiple sets of LH-Skolem-trees (e.g., μ_5 in Fig. 3a), the verification may also be different. This idea is implemented in our *DeSkolemization* procedure and leads to the definition of *hierarchically ordered*.

Definition 4.3 (**Hierarchically-Ordered**). Let μ be a plain SO tgd. μ is *hierarchically ordered* if there exists a set of LH-Skolem-trees T of μ satisfying:
(1) distinct trees in T have disjoint Skolem functions;
(2) for every Skolem function $f(\mathbf{x})$ in a tree $t \in T$, its arguments are the same as the universal variables of the node $\mathbf{S_{ik}}$ where $f(\mathbf{x})$ is attached, i.e., assume that the path from root to $\mathbf{S_{ik}}$ is $\mathbf{S_{i1}}, \ldots, \mathbf{S_{ik}}$, and the set of universal variables appearing in $\mathbf{S_{i1}}, \ldots, \mathbf{S_{ik}}$ is X_{ik}, then the arguments of f satisfy $\mathbf{x} = X_{ik}$.

μ_1 in Fig. 5a is hierarchically ordered. Because in Fig. 5b the LH-Skolem-trees of μ_1 do not share any Skolem functions; the arguments of Skolem function $Arg(f_d) = X_D = \{n\}$, $Arg(f_l) = X_G = \{n,\ g\}$, $Arg(f_a) = X_B = \{n,\ b\}$, $Arg(f_p) = X_P = \{p\}$. The rewriting result of μ_1 consists of two nested tgds, λ_1 in Fig. 1c which is de-Skolemized from δ_{61}, and the nested tgd $\forall p\ P(p) \to \exists y_p\ P'(p, y_p)$ which is de-Skolemized from δ_{62}. Note that a plain SO tgd μ may have multiple sets of LH-trees (e.g., Fig. 4 shows one of three sets of LH-trees of μ_5), which lead to multiple sets of LH-Skolem-trees. As long as one set of LH-Skolem-trees of μ satisfies Definition 4.3, μ is rewritable. Finally, we present the below sufficient condition to check whether a plain SO tgd can be rewritten to a set of nested tgds.

Theorem 2. *Let μ be a plain SO tgd. If μ is both consistent and hierarchically-ordered, then there exists a set of nested tgds Λ such that $\mu \equiv \Lambda$.*

Theorem 2 is the sufficient and necessary condition for Alg. 3 to succeed, since Alg. 1 produces *LH-Trees* of μ and hierarchical ordering is examined by Alg. 2 and 3. If μ is not hierarchically ordered, then Alg. 3 will try all possibilities and finally return $\emptyset$.

Summary. Besides answering $Q2$, Theorem 2 provides a valuable insight regarding characterizations of plain SO tgds rewritable to nested tgds. That is, there exists a proper subclass of plain SO tgds which can always be rewritten to logically equivalent nested tgds, i.e., the plain SO tgds which are consistent and hierarchically-ordered. Comparing to the linearity [6], the condition hierarchically-ordered provides a more precise portrait of the structural properties of a plain SO tgd that guarantee its rewritability. Moreover, Theorem 2 also reveals that not only Skolem functions, but also the left-hand sides of a plain SO tgd, i.e., source schemas, may affect its rewritability to nested tgds, which is not studied in existing works.

4.3 Complexity

In order to answer $Q3$ regarding efficiency, we analyze the complexity of our approach and show that the core algorithms have the polynomial-time complexity. To analyze the complexity of the algorithms, we consider the following parameters: m is the number of parts in the plain SO tgd; s and t denote the average number of source (resp., target) relations in a part; a is the average number of universal variables in a source relation; b is the average number of distinct Skolem functions appearing in a target relation; and the average arity of a target relation is c.

Theorem 3. *Let μ be a plain SO tgd and the quantities of its parameters are fixed. Then,* Nest *and* DeSkolemization *are polynomial-time algorithms, and* BuildNestedTGD *is an exponential-time algorithm.*

In our implementation, we sort the parts by the number of source relations, and handle the part with the lowest number of source relations first, which is often a good choice. In the evaluation, we will show that with input plain SO tgds rewritable, the exponential complexity is often only relevant for the case when searching for all solutions.

5. EVALUATION

With intensive experiments we have evaluated our approach in the following aspects. In Sec. 5.1, we introduce the mapping test suite that generates plain SO tgds as the input of our algorithms. In Sec. 5.2, we demonstrate experimentally the correctness of our approach. In Sec. 5.3, we use [6] as baseline and compare it with our approach regarding the rewriting rate. Furthermore, in Sec. 5.4 we explore the hierarchical ordering of plain SO tgds with diverse characteristics. Finally, we examine the performance of our

Table 1: Parameters of nested tgd generator

Parameter	Description	Value
$\pi_{TreeHeight}$	Pattern Tree height	h
$\pi_{NumLeaves}$	Number of child leaves per node in pattern	k
$\pi_{FullLeaves}$	Whether generate a full pattern tree	T/F
$\pi_{NumParts}$	Number of parts per nested tgd	m
$\pi_{NumSrcRels}$	Number of source relations per part	s
$\pi_{AritySrcRels}$	Arity of source relations	a
$\pi_{NumTgtRels}$	Number of target relations per part	t
$\pi_{ArityTgtRels}$	Arity of target relations	c
$\pi_{NumExistVars}$	Number of distinct existential variables per target relation	b

algorithms in Sec. 5.5, which confirms with our complexity analysis in Sec. 4.3. The experiments are performed on a Intel i7 1.8GHz machine with four cores and 24GB RAM.

5.1 Mapping Generators

To examine the generality of our approach, it is important that we have plain SO tgds with a huge variety of characteristics as input. There are existing tools for generating schema mappings, such as iBench [4] which is extended from STBenchmark [2]. Notably, although iBench can be used for logical mapping generation, it does not meet the experimental requirements in this work. To explore the relationship between plain SO tgds and nested tgds, our experiments need plain SO tgds as input which can be rewritten to nested tgds. However, iBench uses the mapping languages of tgds and plain SO tgds,[2] but not nested tgds; and its proposed mapping primitives [4, 5] do not cover nested mappings. That is, the plain SO tgds generated by iBench correspond to a set of tgds instead of nested tgds, if they are rewritable.

Thus, we have implemented a mapping test suite including a nested tgd generator and a plain SO tgd generator.[3] The test suite also includes tools for Skolemizing and normalizing a nested tgd; checking whether two given nested tgds are structurally equivalent (one can be transformed to the other by renaming variables); and checking whether two given plain SO tgds are structurally equivalent. We focus on the introduction of the two generators.

Nested Tgd Generator. Our nested tgd generator has two running modes, *single* and *batch*. In the single mode, the user can customize every variable/relation name such that the generator produces a specific nested tgd. For instance, we can apply this mode to generate λ_2 in Fig. 2. To populate a number of nested tgds, the batch mode can be configured with several parameters, as in Table 1. The *value* column indicates whether the parameter is a boolean parameter (values as T/F) or a quantitative parameter.

Plain SO tgd Generator. Similarly, the plain SO tgd generator also has two running modes, and allows a set of parameters as shown in Table 2. For the plain SO tgd generation no pattern tree related parameters are required, and we use $\pi_{NumFuncs}$ instead of $\pi_{NumExistVars}$ for each target

[2]http://dblab.cs.toronto.edu/project/iBench/

[3]http://dbis.rwth-aachen.de/cms/staff/hai/test/tools

Table 2: Parameters of plain SO tgd generator

Parameter	Description	Value
$\pi_{NumParts}$	Number of parts	m
$\pi_{NumSrcRels}$	Number of source relations per part	s
$\pi_{AritySrcRels}$	Arity of source relations	a
$\pi_{NumTgtRels}$	Number of target relations per part	t
$\pi_{ArityTgtRels}$	Arity of target relations	c
$\pi_{NumFuncs}$	Number of distinct Skolem functions per target relation	b
$\pi_{FuncArgMode}$	How Skolem function arguments are chosen	*Relation* *Part* *All*

relation. As consistency is not our focus in this work, for the following experiments our plain SO tgd generator generates plain SO tgds satisfying consistency.

Similar to previous studies [6, 4], to study how the Skolem functions affect the rewritability, we have designed three modes of $\pi_{FuncArgMode}$ for Skolem function argument generation, adapted from iBench. In *Relation* and *Part* modes the generator chooses the whole set of universal variables from a source relation or a whole part. For instance, μ_3 in Example 3.1 could be a possible output of the *Relation* mode. With the *All* mode, the arguments of a Skolem function will include the universal variables from all parts that have been generated. In preliminary experiments, we have also tested the cases when the generator randomly picks up universal variables as Skolem function arguments, which leads to the generated plain SO tgds rarely rewritable.

5.2 Correctness and Completeness

With Theorem 1, we have proved theoretically the soundness of our approach. To provide a more practical perspective, we design experiments and show that our approach generates nested tgds equivalent to the input plain SO tgds. The main difficulty lies on the unavailability of ground truth. Given a random, complex plain SO tgd, there is no known method to determine whether it is rewritable to a finite set of nested tgds (an open question raised in [21]).

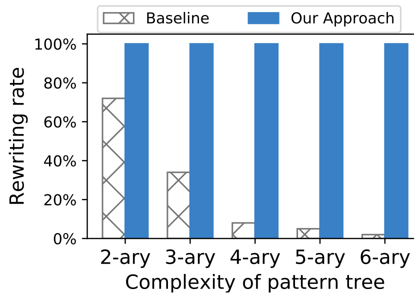
Check with existing literatures. Therefore, we have collected 14 plain SO tgds from existing literatures [3, 6, 21, 14, 4] whose rewritability is known, together with all the examples we have used in this paper. We have processed these plain SO tgds with our approach, and all the rewriting results have been verified to be correct. The detailed results including the input plain SO tgds, their source literatures and our results can be found online.[4]

Apply the nested tgd generator. In addition, to examine the correctness of our approach with complex mappings, we have designed a 2nd experiment using the nested tgd generator and populated 1,088,230 nested tgds, which are transformed to plain SO tgds by Skolemization and normalization. Table 3 shows the value ranges of the quantitative parameters in Table 1 applied in this experiment. For the remaining boolean parameter in Table 1, we have used different combinations of values. In this way, we obtain rewritable plain SO tgds whose equivalent nested tgds

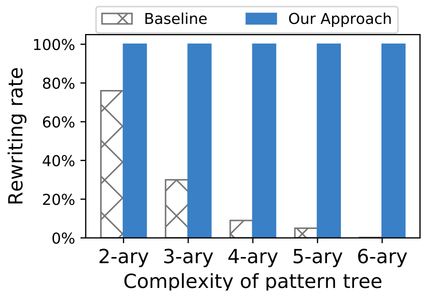
[4]http://dbis.rwth-aachen.de/cms/staff/hai/corr/results

Table 3: Nested tgd generator parameter value range

Parameter	Min	Max
$\pi_{TreeHeight}$	2	6
$\pi_{NumLeaves}$	1	5
$\pi_{NumParts}$	2	10
$\pi_{NumSrcRels}$	1	10
$\pi_{AritySrcRels}$	1	10
$\pi_{NumTgtRels}$	1	10
$\pi_{ArityTgtRels}$	1	10
$\pi_{NumExistVars}$	1	8



(a) $\pi_{NumSrcRels} = 2, \pi_{AritySrcRels} = 4, \pi_{NumTgtRels} = 1, \pi_{ArityTgtRels} = 3, \pi_{NumExistVars} = 1$



(b) $\pi_{NumSrcRels} = 4, \pi_{AritySrcRels} = 6, \pi_{NumTgtRels} = 2, \pi_{ArityTgtRels} = 5, \pi_{NumExistVars} = 2$

Figure 6: Rewritability comparison with baseline approach

are available. We refer to such plain SO tgds as *transformed plain SO tgds*, which are transformed from nested tgds with preserving logical equivalence, and the ground truth of their rewritability is known. We processed these plain SO tgds with our approach *BuildNestedTGD*, and **all** of them can be rewritten to nested tgds. Finally, we applied the tool in our mapping test suite to check whether the newly generated nested tgds are structurally equivalent to the original nested tgds (unique except for renaming of variables). By doing so, we verified that all the rewriting results are correct.

Due to the absence of the ground truth, we cannot fully validate the completeness of our approach. However, in this experiment we are able to show that with the transformed plain SO tgds whose equivalent nested tgds are known, our approach succeeds in finding their solutions.

5.3 Comparison With Baseline Approach

In this experiment, we compare our approach with the state-of-the-art approach [6]. In Sec. 4.2 we have explained why our proposed condition *hierarchically-ordered* is more general than *linearity* in a formal manner. Now we demonstrate this claim more intuitively with experiments.

Experimental setting. To guarantee the availability of ground truth, similar to Sec. 5.2 in this experiment we also first generate nested tgds, then Skolemize and normalize them to obtain transformed plain SO tgds as input. In this way all input plain SO tgds have equivalent nested tgds. For each nested tgd, we generate k-ary pattern trees, and we increase the value of k. The pattern tree height is set as the same with the value of k.

Results. We have run tests with different parameter values with similar observations. Here, we report the rewriting rates of two sets of parameters in Fig. 6a and b. We observe that our approach can rewrite all input plain SO tgds without being affected by the complexity of pattern trees, while the rewriting rate of baseline approach decreases with the increasing pattern tree complexity. The difference becomes significant with a higher value of k. Because linearity is a restrictive description of rewritable plain SO tgds. When the number of children per node (k) increases, the pattern tree structures of initial nested tgds become more complex, and less input plain SO tgds fall into the linear structure.

Summary. These results indicate the considerable gains by applying our proposed approach in terms of rewritability. More importantly, it shows that our approach describes more precisely the structural properties of the rewritable plain SO tgds, especially with the complex cases.

5.4 Rewritable Plain SO Tgds

Answering $Q2$, Theorem 2 has provided the characterization of rewritable plain SO tgds. Now we further explore how the rewritability is affected by the structural properties of the given plain SO tgds, e.g., number of parts, source relations and Skolem functions, etc.

Experimental setting. We generate input plain SO tgds directly by the plain SO tgd generator introduced in Sec. 5.1, which we refer to as *generated plain SO tgds*. Note that different from the transformed plain SO tgds in previous experiments using the nested tgd generator, in this experiment we do not have the ground truth whether a generated plain SO tgd has equivalent nested tgds. Therefore, given such an input plain SO tgd μ, when the rewriting process succeeds and returns a set of nested tgds Λ, we verify the correctness of the results by Skolemizing and normalizing Λ back to a new plain SO tgd μ', then check whether μ' and μ are structurally equivalent, i.e., unique unto renaming. In this way, we have verified that all the rewriting results are correct. All tests in this experiment share the same parameter values unless explicitly given in Fig. 7: $\pi_{NumParts} = 3$, $\pi_{NumSrcRels} = 1$, $\pi_{NumTgtRels} = 1$, $\pi_{AritySrcRels} = 5$, $\pi_{ArityTgtRels} = 4$, $\pi_{NumFuncs} = 1$.

Results. First we study how the rewritability is affected by Skolem functions in terms of their types and proportion. We generate plain SO tgd with three modes of Skolem function arguments ($\pi_{FuncArgMode}$) in Table 2. Fig. 7a illustrates how the rewriting rate in percentage (y-axis) varies with the increasing proportion of Skolem functions in target relation arity (x-axis). By comparing the three lines in Fig. 7a, we can observe that there is a considerable rewritability difference due to the choices of Skolem function arguments. *Relation* mode has the lowest rewriting rate which is close to zero with the increasing proportion of Skolem functions, as in this mode the minimum unit for universal variable selection as Skolem function arguments is a source relation, which makes it less likely that the Skolem functions satisfy hierarchical ordering. The rewriting rate for *Part* mode is higher since the minimal unit for Skolem function argument selection is a part. The plain SO tgds generated in *All* mode are always rewritable. Because in this mode all the universal variables generated in the scope of the current part are included in the argument list of the Skolem function of this part. Therefore, our approach can always find its equivalent nested tgd (similar to Example 3.3). Moreover, by observing the lines of *Relation* and *Part* modes in Fig. 7a

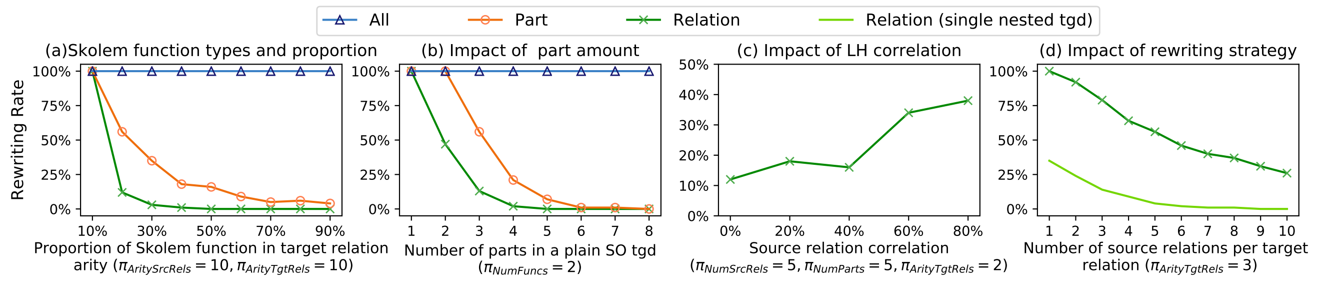


Figure 7: Rewriting rate with varying values of different parameters

we also find that the rewritability declines with the increasing proportion of Skolem functions in the arity of a target relation. Because if one Skolem function is unqualified then the whole plain SO tgd fails to satisfy hierarchical ordering. With more Skolem functions, it becomes more difficult for the generated plain SO tgd to meet the requirement of hierarchical ordering.

Next we study whether the rewritability of our solution is affected by other structural properties of input plain SO tgds. In Fig. 7b we have run the test by increasing the number of parts in a plain SO tgd (x-axis). We observe that except the *All* mode in which a rewriting is always possible, the rewriting rates in the other two modes decline with the increasing amount of parts. This is expected since with more parts, there is a higher possibility that a Skolem function does not include universal variables from the "potential ancestor" parts, and the plain SO tgd fails to be hierarchically-ordered.

We have run tests by varying the values of other parameters in Table 2, e.g., number of source relations per part, which showed little impact on rewritability.[5] Furthermore, we vary the ratio of source relation correlation (number of source relations from previously generated parts divided by number of source relations in the current part). For instance, in Fig. 1b, σ_2 and σ_3 both have two source relations with one relation $D(n)$ from σ_1, thus the ratio of source relation correlation is 50%. We set the Skolem function generation as *Relation* mode. Fig. 7c indicates that the rewriting rate gains a noticeable increase with the rise of the correlation ratio. Because when the left-hand sides of parts are more correlated, it is more likely to build a LH-tree, which contributes to a higher likelihood of satisfying Theorem 2.

Finally we compare the different rewriting strategies, i.e., rewrite the given plain SO tgd to a solution (a set of nested tgds) or a single nested tgd. Fig. 7d shows the result with the Skolem function generation set as *Relation* mode. Here the rewriting rate is significantly higher when we rewrite the given plain SO tgds to a set instead of one nested tgd. This is expected as there are plain SO tgds rewritable to a finite set of nested tgds instead of one nested tgd.

Summary. The rewritability of plain SO tgds is affected by the types and proportion of Skolem functions. Such an observation aligns with the previous work [6]. A more valuable discovery is that the rewritability is a rather complex problem, and its factors are a combination of Skolem functions and other structural properties (amount of parts, correlation of left-hand), and rewriting strategies. It is also the key insight of *hierarchical ordering*. That is, the Skolem function arguments need to "match" with the tree-like patterns built from source relations of the given plain SO tgd.

[5]`http://dbis.rwth-aachen.de/cms/staff/hai/comp/results`.

5.5 Performance

Experimental setting. We have directly generated plain SO tgds using the plain SO tgd generator and set the Skolem function as *All* mode, thus the input plain SO tgds are all rewritable. We mainly report the rewriting time with regard to varying number of parts,[6] as it has the most significant contribution to complexity (see the proof of Theorem 3).

Results. Fig. 8a shows the rewriting time in *milliseconds* when the algorithm *BuildNestedTGD* terminates with a valid solution found, while Fig. 8b provides the rewriting time in *seconds* for searching all solutions. The trends how rewriting time changes with increasing number of parts is consistent with our analysis in the following aspects: (i) In Fig. 8a we observe that our rewriting algorithms are quite efficient, e.g., for a plain SO tgd with 8 parts of 32 source relations, our approach just needs 71 milliseconds in average. (ii) Fig. 8b indicates that by changing the rewriting strategy to multiple solutions, the rewriting time increases significantly when the input plain SO tgds have a larger number of parts. This is expected as the search of all possible solutions has the exponential complexity. To make it clearer, in Fig. 8c we show the number of candidate solutions with varying values of parts, which presents a similar trend as in Fig. 8b. In Fig. 8d we use the number of tested solutions as x-axis, and observe that the rewriting time scales well with the number of tested solutions.

Nevertheless, FO semantics discovery is not a frequent operation in data exchange and data integration applications. In Fig. 8b for a complex mapping with 8 parts it takes less than 23 minutes in average, which is still endurable. For practical use a mapping designer can first run our approach to find one solution. If she is interested in finding all possible rewriting solutions for comparison, then she can also run our algorithms to find all solutions of the mapping expressed as a plain SO tgd.

Summary. The experimental results confirm with our previous complexity analysis. Besides showing that our approach can efficiently find one valid solution for the given rewritable plain SO tgd, we also find out the exponential complexity of finding all possible solutions in such cases, which is not discussed in the existing works.

[6] Full report: `http://dbis.rwth-aachen.de/cms/staff/hai/perfm/results`.

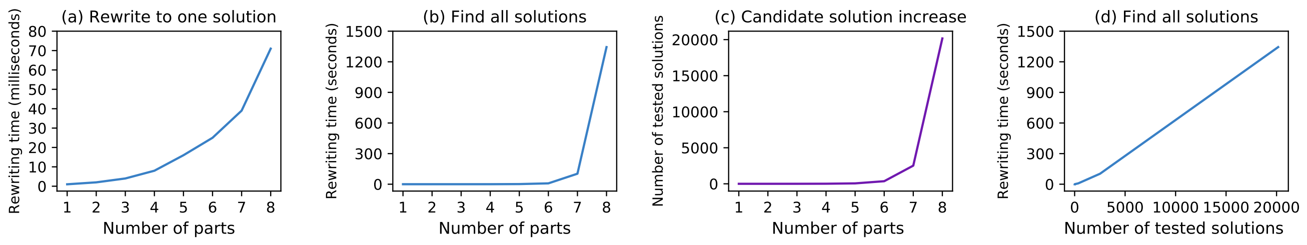


Figure 8: Performance test ($\pi_{NumSrcRels} = 4, \pi_{AritySrcRels} = 5, \pi_{NumTgtRels} = 3, \pi_{ArityTgtRels} = 4, \pi_{NumFuncs} = 2$)

6. RELATED WORK

In this work, we study the problem of transforming plain SO tgds to dependencies with first-order semantics, e.g., nested tgds. The origin of this problem is the predicate variables elimination from classical SO logic (see [13] for an overall introduction). However, our focus is plain SO tgds, a class of SO formulas with specific syntax and practical applications in data exchange and data integration systems.

The main research question of [24] is the compositions of schema mappings. The direct output of their composition algorithms are SO tgds. To make the mapping results applicable for further data transformations such as SQL queries, they proposed an algorithm that transfers SO tgds back to tgds. A sound but not complete, P-time checkable condition is also proposed, which tells when the SO tgds can be transformed to its equivalent tgds. However, in terms of expressive power the class of tgds is a strict subclass of nested tgds. Thus the de-Skolemization algorithm proposed in [24] covers less plain SO tgds than our approach.

Similar remarks hold for [25], which studies the complexities of the model checking problems for different mapping dependencies. The checking conditions are given to tell when an input SO tgd μ without nested function terms can be translated into FO sentences. That is, the variables constituting the argument list of every Skolem function in μ should be consistent; there should exist a linear ordering $\vec{X}$ of the arguments of Skolem functions of μ. Then by applying standard de-Skolemization procedure that replaces Skolem functions with existentially quantified variables, μ is transformed to FO sentences. No explicit algorithms are given in [25]. The limitation of using this method to solve $Q1$ is also that only tgds rather than nested tgds are considered.

We mainly compared our solution with [6], which explicitly defines the checking conditions of *consistency* and *linearity*, and provides the algorithms to rewrite plain SO tgds. In particular, a partitioning method *maximum partition* is proposed, which cuts the plain SO tgds to multiple blocks, and then checks each block for linearity. [6] has shown that linearity is more general and its approach can rewrite more plain SO tgds than [24, 25]. The approach in [6] rewrites a given plain SO tgd into a set of tgds or FO sentences. The results do not always satisfy the syntax of nested tgds, and their decidability of logical equivalence is unknown. Unlike our approach, in [6] the nesting of mappings (i.e., nesting of implications in nested tgds) is not considered. Such a property is the main reason why nested mappings are more efficient than basic mappings [12, 16]. Moreover, we have proven formally (Sec. 4.2) and showed experimentally (Sec. 5.3) that even with a focus on rewritability, our algorithmic approach with the condition of a hierarchical order is more general than the linearity based approach [6].

In [21] a tool, i.e., *Gaifman graph of nulls* is proposed to tell apart plain SO tgds and nested tgds. For every nested GLAV mapping, its Gaifman graph of nulls has bounded path length (Theorem 4.16, [21]). However, the decidability of determining whether a given plain SO tgd has bounded path length in its Gaifman graph of nulls is unknown. In this work we provide a sufficient condition (Theorem 2), which is easy to implement.

7. CONCLUSION

We have studied the problem whether a given plain SO tgd has first-order semantics and can be rewritten into nested tgds. We proposed an algorithmic approach to transform plain SO tgds to one or a set of nested tgds while preserving the logical equivalence. We also defined a sufficient condition to tell whether the input plain SO tgd is rewritable based on the structural properties. We have formally proved the correctness of our approach, which we also demonstrated experimentally through enumerating a massive number of plain SO tgds with different characteristics and sizes. Although we cannot verify the completeness of our approach due to the lack of the ground truth, we showed that it is more general than existing works, and applies to a wide range of plain SO tgds. We have formally explored the complexity of rewriting tasks, and experimentally showed that our algorithms have a satisfying performance.

Our current approach investigates the native properties of the input schema mapping, i.e., its logical equivalence to first-order mapping dependences. In the future, we plan to extend our approach by considering additional input, e.g., (relaxed) functional dependencies [17]. Moreover, in our data lake system [15], we generate mappings as plain SO tgds [16], and use them to rewrite queries in a polystore-based setting [18]. We are currently working on extending our proposed algorithms for refining the generated mappings to boost the mapping execution for data exchange or data integration tasks in a heterogeneous data lake.

Acknowledgements: The authors would like to thank the German Research Foundation DFG for the kind support within the Cluster of Excellence "Internet of Production" (Project ID: EXC 2023/390621612).

References

[1] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: mapping understanding and design by example. In *2008 IEEE 24th International Conference on Data Engineering (ICDE)*, pages 10–19, 2008.

[2] B. Alexe, W. C. Tan, and Y. Velegrakis. STBenchmark: towards a benchmark for mapping systems. *PVLDB*, 1(1):230–244, 2008.

[3] M. Arenas, J. Pérez, J. Reutter, and C. Riveros. The language of plain SO-tgds: Composition, inversion and structural properties. *Journal of Computer and System Sciences*, 79(6):763–784, 2013.

[4] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[5] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The iBench Integration Metadata Generator: Extended Version. Technical report, 2015. `http://dblab.cs.toronto.edu/project/iBench/docs/iBench-TR-2015.pdf`.

[6] P. C. Arocena, B. Glavic, and R. J. Miller. Value invention in data exchange. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 157–168. ACM, 2013.

[7] A. Bonifati, E. Q. Chang, T. Ho, L. V. S. Lakshmanan, R. Pottinger, and Y. Chung. Schema mapping and query translation in heterogeneous P2P XML databases. *The VLDB Journal*, 19(2):231–256, 2010.

[8] A. Bonifati, W. Nutt, R. Torlone, and J. V. den Bussche. Mapping-equivalence and oid-equivalence of single-function object-creating conjunctive queries. *The VLDB Journal*, 25(3):381–397, 2016.

[9] R. Fagin, P. G. Kolaitis, A. Nash, and L. Popa. Towards a theory of schema-mapping optimization. In *Proc. 27th ACM Symposium on Principles of Database Systems (PODS)*, pages 33–42, 2008.

[10] R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Transactions on Database Systems (TODS)*, 30(4):994–1055, 2005.

[11] I. Feinerer, R. Pichler, E. Sallinger, and V. Savenkov. On the undecidability of the equivalence of second-order tuple generating dependencies. *Information Systems*, 48:113–129, 2015.

[12] A. Fuxman, M. A. Hernandez, H. Ho, R. J. Miller, P. Papotti, and L. Popa. Nested mappings: schema mapping reloaded. In *Proceedings of the 32nd international conference on Very large data bases (VLDB)*, pages 67–78, 2006.

[13] D. M. Gabbay, R Schmidt, and A Szalas. *Second Order Quantifier Elimination: Foundations, Computational Aspects and Applications.* College Publications, 2008.

[14] G. Gottlob, R. Pichler, and E. Sallinger. Function Symbols in Tuple-Generating Dependencies: Expressive Power and Computability. In *Proc. 34th ACM Symposium on Principles of Database Systems (PODS)*, pages 65–77, 2015.

[15] R. Hai, S. Geisler, and C. Quix. Constance: an intelligent data lake system. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 2097–2100, 2016.

[16] R. Hai, C. Quix, and D. Kensche. Nested schema mappings for integrating JSON. In *International Conference on Conceptual Modeling (ER)*, pages 397–405, 2018.

[17] R. Hai, C. Quix, and D. Wang. Relaxed functional dependency discovery in heterogeneous data lakes. In *International Conference on Conceptual Modeling (ER)*, 2019.

[18] R. Hai, C. Quix, and C. Zhou. Query rewriting for heterogeneous data lakes. In *European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 35–49, 2018.

[19] R. Hull and M. Yoshikawa. ILOG: declarative creation and manipulation of object identifiers. In *16th International Conference on Very Large Data Bases (VLDB)*, pages 455–468, 1990.

[20] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 393–402, 1992.

[21] P. G. Kolaitis, R. Pichler, E. Sallinger, and V. Savenkov. Nested dependencies: structure and reasoning. In *Proc. 33rd ACM Symposium on Principles of Database Systems (PODS)*, pages 176–187, 2014.

[22] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. 21st ACM Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.

[23] B. Marnette, G. Mecca, and P. Papotti. Scalable data exchange with functional dependencies. *PVLDB*, 3(1):105–116, 2010.

[24] A. Nash, P. A. Bernstein, and S. Melnik. Composition of mappings given by embedded dependencies. *ACM Transactions on Database Systems (TODS)*, 32(1):4, 2007.

[25] R. Pichler and S. Skritek. The complexity of evaluating tuple generating dependencies. In *Proceedings of the 14th International Conference on Database Theory (ICDT)*, pages 244–255, 2011.

[26] B. ten Cate and P. G. Kolaitis. Structural characterizations of schema-mapping languages. In *Proceedings of the 12th International Conference on Database Theory (ICDT)*, pages 63–72, 2009.

[27] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 371–382, 2004.