

Compiling Equality in an Abstract Relational Model via Preference Tables and Translation Tables

by

Ensieh Mollazadeh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

© Ensieh Mollazadeh 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Borgida et al. have introduced a refinement to the *relational model* (RM) [1] which they call the *abstract relational model* (ARM) that extends the former in the following three ways:

1. the addition of a new abstract domain `eid` of entity identifiers to [Structured Query Language \(SQL\)](#) built-in concrete domains;
2. a capacity to resolve reference issues via PRIMARY KEY clauses is replaced by a new domain specific language for referring expression types; and
3. terms in SQL of the form “ $v.A$ ” can now have the form “ $v.A_1 \cdots A_k$ ” to more compactly encode navigation over foreign keys, thus yielding the language SQLP.

They have also proposed an algorithm for mapping ARM schemata to corresponding RM schemata via referring expression types and to subsequently map SQLP queries over the former to corresponding SQL queries over the latter, again via referring expression types. This mapping system relies on introducing so-called *preference tables* to enable coercion between alternative primary keys. Such tables, however, fail to account for circumstances in which explicit translation tables can exist to map between such alternatives in order to satisfy programmer intentions. In this thesis, we remedy this by extending their algorithm to enable the generation of such translation tables and their use in compiling SQLP.

Acknowledgements

First and for most, I would like to express my gratitude to Prof. Gran Weddell for his exceptional supervision of my master's study. He directed me through all the challenges, and taught me import academic skills necessary for successful completion of this thesis. I could have not taken this journey without his support.

Second, I would also like to thank David Toman for his co-supervision, feedback, and guidance. Furthermore, I am thankful to Tamer Özsu, and Richard Trefler for reviewing this thesis.

Finally, I would like to express my appreciation to my parents for their unconditional love and support. I am also very grateful to my husband, Hadi Zibaeenejad, for assisting me overcome difficulties in my study.

Table of Contents

List of Figures	vii
1 Introduction	1
1.1 Contributions	10
1.2 Thesis Outline	11
2 On Mapping ARM Schemata and SQLP via Preference Tables	12
2.1 Formal Definitions	14
2.2 Referring Expression Types	18
2.2.1 Global Data	21
2.2.2 ARM to RET Mapping	24
2.3 On Identity Resolution	33
2.3.1 A New Form of RET	36
2.4 Mapping an ARM schema to an RM schema	42
2.4.1 ARMtoRM on UNIV	48
3 Adding Translation Tables	53
3.1 ARM-to-RM mapping, considering translation tables	54

3.1.1	Identity Resolution with Translation Tables	55
3.1.2	On Reducing the Number of Translation Tables	57
3.2	SQLAtoSQL: Mapping an SQLA query to an SQL query	85
4	Conclusion	127
4.1	Future Study	128
	References	130
	Appendices	132
A	Table Declarations	133
A.1	Chapter 2, Example 2 UNIV table declarations	133
A.2	Chapter 2 RETs	134
A.3	Chapter 2, Example 2 concrete table declarations	135
A.4	Chapter 3, Example 1 abstract table declarations	136
A.5	Chapter 3, Example 1 RETs	137
A.6	Chapter 3, Example 1 concrete table declarations	138
A.7	Chapter 3, Example 1 translation tables:	139
A.8	Chapter 3, Example 2 abstract table declarations	140
A.9	Chapter 3, Example 2 RETs	141
A.10	Chapter3, Example 2 concrete table declarations	141
A.11	Chapter 3 Example 3 abstract table declarations for EMPLOYEE , PROFESSOR , STUDENT , VISITOR and CANADIAN	143
A.12	Chapter 3, Example 3 RETs	144
A.13	Chapter 3 Example 3 concrete table declarations for EMPLOYEE , PROFESSOR , STUDENT , VISITOR and CANADIAN	144

List of Figures

1.1	Concrete table declarations for tables of Σ'_1 .	2
1.2	Updated concrete table declarations for tables of Σ'_1 .	3
1.3	Abstract table declarations for tables of Σ_1 .	5
1.4	Abstract table declarations for tables of an ARM schema, UNIVERSITY.	6
1.5	Abstract table declarations for tables of Σ_2 .	8
1.6	Concrete table declarations for tables of Σ'_2 .	9
2.1	Surface syntax for ARM table clauses.	15
2.2	Integrity Constraints	16
2.3	UNIV, an ARM schema for a university.	17
2.4	A grammar for an SQLP query language and a grammar for an SQLA query language.	19
2.5	SUPERVISION, an ARM schema for a university.	21
2.6	A database for SUPERVISION.	21
2.7	SUPERVISION', the RM version of SUPERVISION.	22
2.8	A database for SUPERVISION' with eids replaced by strings .	23
2.9	Global data	25
2.10	PG for UNIV	30

2.11	PKG for UNIV	30
2.12	Global data for all tables in UNIV	31
2.13	The correct version of SUPERVISION' with eids replaced by strings. . .	41
2.14	A call-graph for algorithms and definitions for mapping an ARM schema to an RM schema.	47
2.15	The concrete relational schema UNIV'	51
3.1	Modified global data for PROFESSOR, STUDENT and PERSON.	65
3.2	The concrete relational schema SecondUNIV'.	70
3.3	The relational schema ThirdUNIV'.	71
3.4	The preference graph for ThirdUNIV.	71
3.5	Modified global data for PROFESSOR, STUDENT and PERSON.	72
3.6	The ARM schema UNIVPEOPLE	77
3.7	The preference graph for UNIVPEOPLE.	77
3.8	Global data for EMPLOYEE, PROFESSOR, STUDENT, VISITOR and CANADIAN. .	80
3.9	The relational schema UNIVPEOPLE'	86
3.10	A dependency graph of functions and procedures that are called to map an SQLA query to an SQL query.	87
3.11	Conversion of an SQLA query over SecondUNIV to an SQL query over SecondUNIV'.	100
3.12	Procedure and function calls with their result for Section 3.2 Example 1. .	103
3.13	Conversion of an SQLA query Q over ThirdUNIV to an SQL query Q' over ThirdUNIV'.	104
3.14	Procedure and function calls with their result for Section 3.2 Example 2. .	111
3.15	Conversion of an SQLA query Q over UNIVPEOPLE to an SQL query Q' over UNIVPEOPLE'.	113

3.16 Procedure and function calls with their result for Section 3.2 Example 3. . .	126
A.1 RTA(T)s for each table T in Σ	144

Chapter 1

Introduction

The *relational model* (RM) underlies the predominant means of data management that was proposed by Ted Codd in 1970 [3]. The model assumes all data is organized as a collection of tables, and derives from first-order predicate logic. Tables in RM consist of one or more named columns called *attributes*, where values occurring in columns are instances of concrete data types such as **integer**, **string**, **varchar**, and so on. Note that the names of columns are important in helping a user interpret the contents of the table. The contents consist of a finite collection of tuples, where each tuple encodes a meaningful relationship.

The standard interface to database systems that adopt RM is SQL. Indeed, the predominant means of data manipulation by deployed applications adopt RM as their underlying data model and SQL as the language of interaction.

Throughout this thesis, we have numerous occasions to introduce an example RM schema. In doing so, we will follow a naming protocol for RM tables that appends “-C” to the table name, as illustrated by our first example in Figure 1.1. This Figure indicates table declarations for tables that exist in an RM schema Σ'_1 . One possibility for the contents of the INSTRUCTOR-C table might be the tuple

(15104763, "David", 2) .

```

table INSTRUCTOR-C (name string, office integer, department string,
                    primary key (name, office))

table GRADUATE-C (gnum integer, name string, year integer, primary key (gnum))

table STAFF-C (snum integer, name string, salary integer, primary key (snum))

```

Figure 1.1: Concrete table declarations for tables of Σ'_1 .

Indeed, this is a relationship among three concrete data values. However, more particularly, the primary key clause suggests that the tuple is recording three concrete facts about a particular instructor, where the first two facts, the **name**, and **office** of the instructor serve the role of (indirectly) identifying a particular instructor by an expression that can be stated in English as “The instructor whose name is David and whose office number is 325”.

Moreover, **GRADUATE-C** entities are identified by their graduate numbers **gnum** and, **STAFF-C** entities are identified by their staff numbers **snum**. A tuple (15104763, "David", 2) in the **GRADUATE-C** table can record information about a graduate student whose name is "David", studying at the 2nd year of graduate school, and who is identified with his **gnum** which is 15104763. There is a possibility that this graduate student is also the instructor whose office number is 324, as indicated in the first tuple above. Therefore, this example illustrates the person who can be both a graduate student and an instructor.

On reflection, in a real world, a university member can work with different titles. For example, a person can be both a graduate student and a university staff member. Therefore, a data entity representing a person can appear in any combination of **INSTRUCTOR**, **GRADUATE** and **STAFF**. Now consider the question of finding the names of instructors who are also graduate students. Since there are different ways to identify graduates and instructors, there cannot exist an SQL query to compare them. More precisely, there cannot exist an SQL query to compare (**name**, **office**) with **gnum**. The same problem occurs for finding the common entities between every other two tables in Σ'_1 .

Borgida et al. have proposed an extension to RM and to SQL, called the *abstract*

```

table INSTRUCTOR-C (name string, office integer, department string,
                    primary key (name, office))

table GRADUATE-C (disc integer, f string, gnum integer, name string, year integer,
                 primary key (disc, f))

table STAFF-C (disc integer, f string, snum integer, name string, salary integer,
              primary key (disc, f))

```

Figure 1.2: Updated concrete table declarations for tables of Σ'_1 .

relational model (ARM) and SQLP, to resolve this issue [1]. In particular, SQLP replaces SQL's primary key declarations with a much more general facility for specifying so-called *referring expression types* (RETs). We illustrate how their resolution applies in this case later on, where we give a corresponding ARM schema defined in SQLP. However, for now, we illustrate the net impact on this RM schema in Figure 1.2. In particular, notice that two of the tables have two extra attributes: `disc` and `f`. To understand the value of `disc` and `f`, we should first visit the concept of *total order* which Borgida et al. have introduced in their earlier work [1]. The total order of tables in an RM schema is an ordered list of tables in the schema. For instance, the total order of Σ'_1 is (INSTRUCTOR, GRADUATE, STAFF). The offset of INSTRUCTOR, GRADUATE and STAFF which is their position in the total order is 1, 2 and 3 respectively.

The `disc` value of GRADUATE can be the offset of a table to the farthest left of it in the total order which here is INSTRUCTOR. In other words, if a graduate student is an instructor, then the `disc` attribute of GRADUATE is 1, and if not, then it is 2. The `disc` attribute of a staff member who is also an instructor, regardless of being a graduate student, is 1. If a staff member is not an instructor, but a graduate student, then the `disc` attribute is 2 and if he/she is neither a graduate student nor an instructor, then it is 3.

The value of the attribute `f` is the concatenation of attribute(s) of the concrete primary key of the table whose offset is in the `disc` column. For example, for those graduate students who are also instructors, the GRADUATE table contains the `disc` value of 1 and

the **f** value which is the concatenation of **name** and **office** of those instructors, since the primary key of **INSTRUCTOR** is (**name**, **office**). The following SQL query to “find the names of instructors who are also graduate students” can now be achieved, via **disc** and **f**:

```
select distinct i.name from INSTRUCTOR-C i
where exists( select * from GRADUATE-C g
              where g.disc = 1
              and g.f = Concat( i.name, i.office))
```

(1.1)

This SQL query resolves the referencing issues and finds the appropriate tuples. However, the protocol it uses, requires significant discipline and care on the part of programmers, which is far from ideal. Furthermore, there are various ways for representing and storing information in a relational database. Querying such information, using the notion of primary and foreign keys may be challenging for domain experts as they are not familiar with querying an actual relational schema. Therefore, the challenging level of writing such SQL queries make them more error-prone.

Borgida et al. have addressed this issue by a simple generalization of the RM schema, which they have called the *abstract relational model* (ARM). In particular, an ARM schema is formed by adding one additional domain for attributes, denoting entities which they have called *entity identifier* (**eid**). All abstract attributes have the domain of **eid**. Every table in the ARM schema has the abstract attribute **self** which functionally determines all other attributes in the table. Figure 1.3 indicates table declarations of tables in the ARM schema Σ that corresponds to the tables in Σ' , defined in Figure 1.2.

An abstract tuple (@eid1, "David", 325, "Computer Science") records three facts about the actual instructor itself which is @eid1, the value of column **self**. We define the SQL queries over an arbitrary ARM schema as SQLA queries. Figure 2.4 illustrates a grammar for an SQLP query language and a grammar for an SQLA query language. The SQLA query below indicates how to “find the names of instructors who are also graduate students”:

```

table INSTRUCTOR (self eid, name string, office integer, department string,
                  primary key (name, office))

table GRADUATE (self eid, gnum integer, name string, year integer,
               primary key (gnum), preference (INSTRUCTOR))

table STAFF (self eid, snum integer, name string, salary integer,
            primary key (snum), preference (INSTRUCTOR, GRADUATE))

```

Figure 1.3: Abstract table declarations for tables of Σ_1 .

```

select distinct i.name from INSTRUCTOR i
where exists( select * from GRADUATE g
              where g.self = i.self)

```

(1.2)

Note that this query is easier and simpler to write than the SQL query in (1.1), due to a less challenging protocol that the ARM provides. Therefore, SQLA queries are less error-prone, specially when writing complex conjunctive queries. However, there still exist challenges to write complex conjunctive queries even with SQLA queries. For this reason, Borgida et al. have introduced an extension to the standard SQL which they call SQLP such that it supports the usage of abstract domain attribute and attribute path. SQLP helps to simplify complex conjunctive queries by allowing implicit foreign key joins in the form of “path expressions”. Weicong Ma et al. conducted an empirical experiment to test and evaluate the performance of the ARM and SQLP among undergraduate and graduate students [8]. There is a significant statistical evidence that SQLP indeed requires less time for understanding and authoring queries, with no loss in accuracy.

Figure 1.4 illustrates a new ARM schema UNIVERSITY with four tables INSTRUCTOR, GRADUATE, STAFF and DEPARTMENT, such that they all have primary key clauses, and only GRADUATE and STAFF have preference clauses. The following indicates an SQLP query over the UNIVERSITY to find the names of the instructors whose department chair’s name is

```

table INSTRUCTOR (self eid, name string, office integer, department eid,
                  primary key (name, office), foreign key (department)
                  references DEPARTMENT)

table GRADUATE (self eid, gnum integer, name string, year integer,
                primary key (gnum), preference (INSTRUCTOR))

table STAFF (self eid, snum integer, name string, salary integer,
             primary key (snum), preference (INSTRUCTOR, GRADUATE))

table DEPARTMENT (self eid, deptname string, chair eid,
                  primary key (deptname), foreign key (chair)
                  references INSTRUCTOR)

```

Figure 1.4: Abstract table declarations for tables of an ARM schema, UNIVERSITY.

“Tom” :

```

select distinct i.name
from INSTRUCTOR i
where i.department.chair.name = "Tom"

```

The SQL version of this SQLP over the RM would be more complex and contains much more lines. Weicong Ma et al. show different examples of SQLP queries and their equivalent SQL over the RM [8].

Although ARM has made querying SQL easier, abstract attributes are invisible in the RM schema and they need to be replaced with concrete ones. In fact, an ARM schema and SQLA queries written over it need to be mapped to an RM schema and their corresponding SQL queries over the RM schema respectively. For this purpose, Borgida et al. have conducted a mapping system between ARM and RM [1]. In particular, they have implemented a mapping from SQLP to SQLA queries and then from SQLA to SQL queries over RM. We focus on the mapping of ARM to RM and SQLA to SQL over RM in this thesis.

The domain specific language of RETs mentioned above (for specifying referring expression types) introduced by Borgida et al. were the means of establishing concrete RM schemata that correspond to a given ARM schemata via an assignment of such a type to each table in a given ARM schema. One of our contributions is to introduce a much more user friendly front-end for specifying such an RET table assignment by reusing SQL’s PRIMARY KEY clause and adding a new PREFERENCE clause. Figure 1.3 illustrates the use of primary key clauses for three tables INSTRUCTOR, GRADUATE and STAFF in the ARM schema Σ_1 , and the use of preference clauses for tables GRADUATE and STAFF. Preference clauses in Figure 1.3 are replaced with `disc` and `f` in Figure 1.2, when Σ_1 is converted to the RM version Σ'_1 .

A preference table T_i-T_j -C contains three columns `disc`, `f` and the primary key of the right table which is T_j . For instance, since GRADUATE contains a preference clause, including INSTRUCTOR, there exists a preference table INSTRUCTOR-GRADUATE-C with attributes `disc`, `f` and `gnum`. The right outer join of GRADUATE-C with INSTRUCTOR-GRADUATE-C on the column `gnum` would change the GRADUATE-C table by adding `disc` and `f` to it, as shown in Figure 1.2, and it now has the same number of rows as INSTRUCTOR-GRADUATE-C.

Similarly, STAFF contains a preference clause, over INSTRUCTOR and GRADUATE. Therefore, there exist two preference tables INSTRUCTOR-STAFF-C and GRADUATE-STAFF-C. The right outer join of STAFF-C with both INSTRUCTOR-STAFF-C and GRADUATE-STAFF-C on the column `snum` would also extend STAFF-C by adding attributes `disc` and `f`, as also illustrated in Figure 1.2. Consequently, joining preference tables with the table that contains the preference clause removes the need to explicit tables, and that reduces the required storage.

This thesis introduces a new concept of *translation tables* to resolve equality issues when some preference tables are missing in an schema. For example, Σ_1 in Figure 1.3 without any preference clauses is the same as the ARM schema Σ_2 in Figure 1.5. Therefore, the concrete version of Σ_2 which is called Σ'_2 does not contain any preference tables with `disc` and `f` attributes, as Σ'_1 does in Figure 1.2. Consequently, an SQL query to find “the names of those instructors who are graduate students” fails to compile. We remedy this issue by creating three translation tables INSTRUCTOR-GRADUATE-C, INSTRUCTOR-STAFF-C and


```

table INSTRUCTOR (self eid, name string, office integer, department string,
                  primary key (name, office))

table GRADUATE (self eid, gnum integer, name string, year integer,
               primary key (gnum))

table STAFF (self eid, snum integer, name string, salary integer,
            primary key (snum))

```

Figure 1.5: Abstract table declarations for tables of Σ_2 .

GRADUATE-STAFF-C in Σ'_2 , as indicated in Figure 1.6.

A *translation table* is generated between every two pair of non-disjoint tables when there is no preference table between them, and it contains the intersection of the tables in the pair by storing their primary key attributes. For example, INSTRUCTOR-GRADUATE-C contains three columns `name`, `office` and `gnum`, and a tuple ("David", 325, 15104763) that refers to a graduate student who is also an instructor. Considering the translation table INSTRUCTOR-GRADUATE-C, the following SQL query finds the names of instructors who are also graduate students:

```

select distinct i.name from INSTRUCTOR-C i
where exists (select * from INSTRUCTOR-GRADUATE-C w, GRADUATE-C g
              where w.name = i.name
                 and w.office = i.office
                 and w.snum = g.gnum)

```

Furthermore, due to the notion of disc and f, preference tables are not likely to exist in existing database scenarios and more often, they need to be generated. Translation tables, however, are much more likely to exist in existing database scenarios, and this reduces the need to new translation tables. Despite of this advantage, there is a problem with translation tables that prevents them to fully replace preference tables: Translation tables result in storage overhead and possible loss of efficiency. The following example helps to better understand the former:

```

table INSTRUCTOR-C (name string, office integer, department string,
                    primary key (name, office))

table GRADUATE-C (gnum integer, name string, year int, primary key (gnum))

table STAFF-C (snum integer, name string, salary integer, primary key (snum))

table INSTRUCTOR-GRADUATE-C (name string, office integer, gnum integer,
                             primary key (name, office), foreign key (name, office)
                             references INSTRUCTOR, foreign key(gnum) references
                             GRADUATE-C)

table INSTRUCTOR-STAFF-C (name string, office integer, snum integer,
                         primary key (name, office), foreign key (name, office)
                         references INSTRUCTOR, foreign key (snum) references
                         STAFF)

table GRADUATE-STAFF-C (gnum integer, snum integer, primary key (gnum),
                       foreign key (gnum) references GRADUATE-C,
                       foreign key (snum) references STAFF)

```

Figure 1.6: Concrete table declarations for tables of Σ'_2 .

Suppose an schema with n tables, table T_i (**self** eid, A_i integer, primary key (A_i)), where $1 \leq i \leq n$. Also, each table T_i contains a single tuple (**@eid**, A_i), where **@eid** is the value of column **self** and it is the same for all n tables. Consider two cases:

1. First, there is a preference table T_i-T_j-C for every two tables T_i and T_j , where $i \leq j$. Therefore, there are $n(n-1)/2$ preference tables. However, only $(n-1)$ preference tables T_1-T_i-C contain columns (**disc**, **f**, A_1) with a corresponding tuple $(1, A_1, A_1)$, and the rest are empty. This implies an $O(n)$ store overhead.
2. Second, suppose there does not exist any preference table, and a compiler would produce all necessary translation tables T_i-T_j-C , where $i \leq j$. There are $n(n-1)/2$ of them, each contains a tuple (A_i, A_j) , and neither of them is empty. This implies

an $O(n^2)$ store overhead.

This example indicates the existence of storage overhead in translation tables.

The other advantage of preference tables over translation ones is that joining a preference table T_i-T_j-C with T_j-C reduces the number of tables by adding **disc** and **f** to T_j-C . In fact, joining any number of preference tables with T_j-C just adds **disc** and **f** to T_j-C . This removes the need to store preference tables explicitly. However, translation tables, cannot be joined, since joining them would result in a complex table with a lot of null values. For example, joining translation tables T_1-T_4-C , T_2-T_4-C and T_3-T_4-C with T_4-C would add the primary keys of T_1-C , T_2-C and T_3-C to T_4-C , with some null values.

There are, however, cases in which translation tables can be merged into one table. For example, consider a translation table T_i-T_j-C , and T_i is the subset of T_j . Then a left outer join between T_i-T_j-C and T_i-C on the primary key of T_i-C would add another column to T_i-C which contains the primary key of T_j-C for those T_j-C entities that are also an T_i-C entity. Chapter 3 contains the details about this optimization along with other optimization to absorb translation tables.

1.1 Contributions

Our first contribution relates to how Borgida et al. have introduced so-called referring expression types to generate identity resolution [1]. We introduce a new front-end for programmers to indirectly specify referring expression types via SQL’s primary key clause and a new preference clause, and we also define “**disc**” and “**f**” attributes to encode SQL’s primary keys in concrete tables. This has also required revisiting such types, in particular, replacing “**disc**” and “**f**” attributes with their underlying table and attributes names of abstract tables in order to express verifiable conditions that are sufficient to ensure identity resolution.

Our second and main contribution involves introducing translation tables. These tables provide the possibility of converting between referring expressions, such that a larger space

of referring expressions are allowed to ensure identity resolution. We demonstrate cases with the preference and primary key clauses that would fail to be identity resolving, and adding translation tables would change them to become identity resolving. We also explain mapping an ARM schema to the RM schema, considering the generation of translation tables, and illustrate three different examples to clarify this concept. The conversion of SQLA queries to SQL ones, via the use of translation tables is then defined, and it is shown on the three mentioned examples

1.2 Thesis Outline

In Chapter 2, we review the work of Borgida et al. which constitutes the definition for Abstract Relational Model (ARM) and the language SQLP [1]. We also review the concept of *Referring Expression Types* (RET), illustrate it by an example of an ARM schema, and show different RETs for tables in the schema. We further reuse primary key clauses and introduce new preference clauses, and illustrate how they can be indirectly used to generate RETs. Next, we review and explain the topic of *identity resolution*. Finally, we propose a mapping from an ARM schema to a RM one and indicate an example to clarify this method.

In Chapter 3, we introduce translation tables and how they are generated in an RM schema, and how they can be optimized. We show how identity resolution is extended to accommodate translation tables, and how a new mapping method from an ARM schema to an RM schema is accomplished when both preference tables and translation tables are present. We also explain how to convert an SQLA query to an SQL one by considering translation tables in the RM schema. Then, we present three case studies that consider progressively more involved circumstances to illustrate both ARM-to-RM mapping and SQLA-to-SQL conversion.

In Chapter 4, we review our contributions and discuss a number of future directions of research.

Chapter 2

On Mapping ARM Schemata and SQLP via Preference Tables

In this chapter, we provide a review of the work by Borgida et al. in which the notion of *abstract relational model* (ARM), SQLP and *referring expression types* (RET) were introduced [1]. One of our contribution to the thesis which is also presented in this chapter is to reuse SQL’s primary key clause and introduce a new preference clause as a much more user friendly front-end means of indirectly specifying these RETs. To paraphrase our thesis abstract, ARM extends the RM in the following three ways:

1. adding a new domain `eid`;
2. replacing the means of resolving reference issues via PRIMARY KEY clauses by a new domain specific language for referring expression types; and
3. allowing terms in SQL of the form “ $v.A$ ” to have the form “ $v.A_1 \cdots A_k$ ” to more compactly encode navigation over foreign keys, thus yielding the language SQLP.

The earlier work by Borgida et al. also proposed a so-called *referring expression type* (RET) assignment to generate an RM schema Σ' from an ARM schema Σ , and have outlined two steps in mapping an SQLP query over Σ to an SQL query over Σ' via RETs [1]:

1. map the original SQLP query over Σ to an SQL query also over Σ , which we call an SQLA¹ query in this thesis; then
2. map the SQLA query to a corresponding SQL query over Σ' .

In this chapter, we review the mapping between Σ and Σ' via RETs, by introducing the two main procedures $\text{ARMtoRET}(\Sigma)$ and $\text{ARMtoRM}(\Sigma)$, and the mapping of SQLA queries over Σ to the SQL queries over Σ' .

Furthermore, we present our proposal for a more user friendly means of specifying an RET assignment via the reuse of SQL's primary key clause and the introduction of a new preference clause. In order to map Σ to Σ' , Borgida et al. have introduced **disc** as an additional attribute for a concrete table $T\text{-C}$, when there exists a “;” in the $\text{RET}(T)$ [1]. We extend on this and introduce the string attribute **f**, in addition to **disc**, to store the primary key or the concatenation of all columns in the primary key of the table which its offset is in the **disc** column. Therefore, the combination (disc, f) is the primary key of $T\text{-C}$, when T contains a preference clause with some exceptions which we will explain further in this chapter.

Finally, this chapter is divided to four sections:

- Section 2.1 introduces ARM schema with a way to define table declaration for each table and different types of constraints are defined for tables of the ARM schema.
- Section 2.2 introduces referring expression types and explains a procedure to find a referring expression type for each table in the ARM schema.
- Section 2.3 presents the concept of identity resolution by defining a well-formed RTA, referring expressions and identity resolving type assignments.
- Section 2.4 illustrates a way of mapping the ARM schema to an RM one.

¹SQLA refers to SQL over Abstract schema

2.1 Formal Definitions

This section focuses on the ARM schema and explains how tables in the ARM schemata are declared. It also introduces table constraints in detail and proposes surface syntax used for ARM table declarations.

Definition 1 (ARM Schema)

Let TAB , AT , and CD be sets of *table names*, *attribute names* among which is **self**, and *domain names* CD consisting of the three *data types* **eid**, **integer** and **string**, where **eid** denotes an *abstract domain* of *entities*, and where **integer** and **string** denote the respective *concrete domains* of integers and *finite strings*. An *abstract relational model* (ARM) *schema* Σ is a set of *table declarations* of the form.

$$\text{table } T \text{ } (A_1 \text{ } D_1, \dots, A_k \text{ } D_k, \varphi_1, \dots, \varphi_n),$$

where $T \in \text{TAB}$, $A_i \in \text{AT}$, $D_i \in \text{CD}$, and φ_j are *clauses* attached to table T . We write $\text{TABLES}(\Sigma)$ to denote all table names declared in Σ , $\text{ATTRS}(T)$ to denote $\{A_1, \dots, A_k\}$, $\text{DOM}(T, A_i)$ to denote D_i , and $\text{ABS}(T)$ to denote $\{A_i \in \text{ATTRS}(T) \mid \text{DOM}(T, A_i) = \text{eid}\}$. Also, $\text{DOM}(T, A_i)$ must be **eid** when A_i is **self**.

In addition to attribute declarations “ $A_i \text{ } D_i$ ”, a table declaration can include clauses φ_j . A clause can express an *integrity constraint* such as a foreign key, or resolve reference issues for entities such as a primary key [1].

T is *reified* if **self** $\in \text{ATTRS}(T)$, and the ARM schema Σ is reified if each $T' \in \text{TABLES}(\Sigma)$ is reified. Σ is a *simple class-based schema*, if there is a single foreign key constraint for each $A \in \text{ABS}(T)$, not including **self**, for every $T \in \text{TABLES}(\Sigma)$. We write $\text{RAN}(T, A)$ to denote the table that the foreign key A in T references. Finally, Σ is a *relational model* (RM) schema if $\text{ABS}(T) = \emptyset$ for all $T \in \text{TABLES}(\Sigma)$. \square

Figure 2.1 defines surface syntax for constraints from ROSESEED project by Weddell [10]. The first two constraints are a contribution of this thesis that constitutes a front-end

$\langle \textit{preference} \rangle$	$::=$	<code>preference "(" { $\langle \textit{table-name} \rangle$ } ")</code>
$\langle \textit{primary-key} \rangle$	$::=$	<code>primary key "(" { $\langle \textit{attribute-name} \rangle$ } ")</code>
$\langle \textit{foreign-key} \rangle$	$::=$	<code>foreign key "(" { $\langle \textit{attribute-name} \rangle$ } ")</code> <code>references $\langle \textit{table-name} \rangle$ ["(" { $\langle \textit{attribute-name} \rangle$ } ")"]</code>
$\langle \textit{inclusion-dependency} \rangle$	$::=$	<code>inclusion dependency "(" { $\langle \textit{attribute-name} \rangle$ } ")</code> <code>references $\langle \textit{table-name} \rangle$ ["(" { $\langle \textit{attribute-name} \rangle$ } ")"]</code>
$\langle \textit{isa} \rangle$	$::=$	<code>isa "(" { $\langle \textit{table-name} \rangle$ } ")</code>
$\langle \textit{disjoint-from} \rangle$	$::=$	<code>disjoint from "(" { $\langle \textit{table-name} \rangle$ } ")</code>
$\langle \textit{cover-by} \rangle$	$::=$	<code>cover by "(" { [not] $\langle \textit{table-name} \rangle$ } ")</code>
$\langle \textit{path-functional-dependency} \rangle$	$::=$	<code>path functional dependency [with $\langle \textit{table-name} \rangle$]</code> <code>"(" { $\langle \textit{attribute-path} \rangle$ } ") determines $\langle \textit{attribute-path} \rangle$</code>
$\langle \textit{nominal} \rangle$	$::=$	<code>nominal</code>

Figure 2.1: Surface syntax for ARM table clauses.

for indirectly specifying the above-mentioned RETs via straightforward extensions to SQL’s primary key clause and the introduction of a new preference clause. All remaining clauses in the figure are integrity constraints on data and explained in more detail in Figure 2.2.

- Foreign key clauses are integrity constraints matching the semantics supported by standard SQL, and inclusion dependencies are their standard generalization.
- Inheritance (**isa**), disjointness or cover constraints are only meaningful when **self** occurs as one of the attributes of T and each T_i mentioned in the constraint. For inheritance constraints, any **self**-value occurring in T *must also* occur as a **self**-value in T_i , and, for disjointness, any **self**-value occurring in T *may not* occur as a **self**-value in T_i .

Cover constraints generalize what can be expressed with inheritance and disjointness. Here, for any **self**-value e occurring in T , there must exist a T_i mentioned in the constraint for which, when **not** precedes T_i , e does not occur as a **self**-value in T_i , or, when **not** does not precede T_i , e does occur as a **self**-value in T_i .

- Path functional dependencies are a generalization of functional dependencies that allow *attribute paths* in place of attributes. An attribute path **Pf** is either **self** or a dot-separated sequence of attribute names excluding **self**. A path functional dependency of the form

$$\begin{array}{c} \text{path function dependency with } T' \\ (\text{Pf}_1, \dots, \text{Pf}_m) \text{ determines Pf} \end{array}$$

is satisfied when any combination of a T -tuple and a T' -tuple that agree on the value of each Pf_i also agree on the value of **Pf**. (When not given, table T' defaults to table T .)

- Finally, a nominal constraint occurring in T is satisfied when there is a single tuple in T .

Figure 2.2: Integrity Constraints

UNIV in Figure 2.3 is an example of an ARM schema with seven tables for a university. Every abstract attribute is marked with a star (“*”) and every table contains the abstract attribute **self** of type **eid**. Appendix A.1 indicates table declarations for all seven tables. Except the table **PERSON**, every other table contains a primary key constraint and only

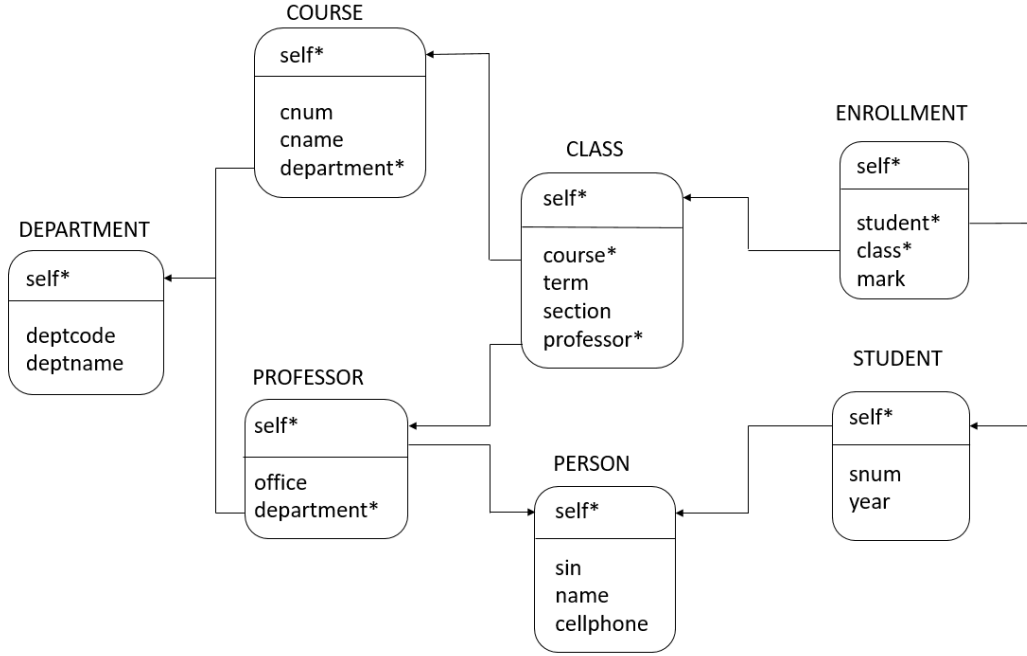


Figure 2.3: UNIV, an ARM schema for a university.

STUDENT and PERSON contain preference clauses.

In addition to the attribute **self**, there exist other abstract attributes in some tables which are foreign keys referencing some other tables. For example, the abstract attribute **department** in the COURSE and PROFESSOR tables is a foreign key, referencing the **self** attributes of the table DEPARTMENT. Both tables PROFESSOR and STUDENT are subsets of PERSON, since their table declarations contain an “**isa** PROFESSOR”.

Moreover, there exist disjoint constraints in all tables. For example, COURSE and DEPARTMENT are disjoint, since an entity can not be both a COURSE and a DEPARTMENT. In other words, the intersection between the **self** attributes of the COURSE and the **self** attributes of the DEPARTMENT table is empty. Also, PERSON is the only table with the “**cover by**” constraint. Thus, the values of **self** attributes of the PERSON are the union of the values of **self** attributes of PROFESSOR and STUDENT. An example of an $\langle \text{attribute-path} \rangle$

would be:

```
class.professor.department.deptcode.
```

Observe how this is just a sequence of foreign key joins. An example of a query using this $\langle attribute-path \rangle$ is the following:

```
select e.student, e.mark
from ENROLLMENT e
where e.class.professor.department.deptcode = "CS"
```

A simple grammar for fragment of an SQLP query is given in Figure 2.4a, and a simple grammar for fragment of an SQLA query is given in Figure 2.4b. This thesis focuses on converting an SQLA query to an SQL query.

2.2 Referring Expression Types

Borgida et al. have proposed a variety of SQLP queries as a means of indirectly referring to `eid` values, generally called *referring expressions*, and a *referring expression type* (RET), defined in the following definition, to denote a set of SQLP queries [1]. For an ARM schema Σ , identification issues are then resolved by indirectly associating an RET with each table in $TABLES(\Sigma)$. Here, a resolution of such issues will make it possible to translate any SQLA query over Σ to an equivalent SQL query devoid of any need to reference or compare `eids`.

$$\begin{aligned}
\langle query \rangle &::= \text{select distinct } x_1.Pf_1[\text{as } A_1], \dots, x_m.Pf_m[\text{as } A_m] \langle body \rangle \\
&| \langle query \rangle \text{ union } \langle query \rangle \\
\langle body \rangle &::= \text{from } T_1 x_1, \dots, T_n x_n [\text{where } \langle pred \rangle] \\
\langle pred \rangle &::= x_1.Pf_1 = x_2.Pf_2 \quad | \quad x.Pf_1 = c \quad | \quad \langle pred \rangle \text{ and } \langle pred \rangle \\
&| \langle pred \rangle \text{ or } \langle pred \rangle \quad | \quad \text{not } \langle pred \rangle \quad | \quad \text{exists (select * } \langle body \rangle)
\end{aligned}$$

(a) An SQLP fragment

$$\begin{aligned}
\langle query \rangle &::= \text{select distinct } x_1.A_1[\text{as } A_1], \dots, x_m.A_m[\text{as } A_m] \langle body \rangle \\
&| \langle query \rangle \text{ union } \langle query \rangle \\
\langle body \rangle &::= \text{from } T_1 x_1, \dots, T_n x_n [\text{where } \langle pred \rangle] \\
\langle pred \rangle &::= x_1.A_1 = x_2.A_2 \quad | \quad x.A_1 = c \quad | \quad \langle pred \rangle \text{ and } \langle pred \rangle \\
&| \langle pred \rangle \text{ or } \langle pred \rangle \quad | \quad \text{not } \langle pred \rangle \quad | \quad \text{exists (select * } \langle body \rangle)
\end{aligned}$$

(b) An SQLA fragment.

Figure 2.4: A grammar for an SQLP query language and a grammar for an SQLA query language.

Definition 2 (Referring Expression Types, and Assignments)

Let Σ be an ARM *schema*. A *referring expression type* (RET) *relative to* Σ has the following form:

$$T_1 \rightarrow (Pf_{1,1} = ?, \dots, Pf_{1,k_1} = ?); \dots; T_k \rightarrow (Pf_{k,1} = ?, \dots, Pf_{k,k_k} = ?), \quad (2.1)$$

consisting of a sequence of “;” separated *components*. Each component $T_i \rightarrow (Pf_{i,1} = ?, \dots, Pf_{i,k_i} = ?)$ consists in turn of a *guard*, written $\text{GUARD}(\text{RET}, i)$, which is the name “ T_i ” of a table in $\text{TABLES}(\Sigma)$ and the set of *attribute paths* $(Pf_{i,1}, \dots, Pf_{i,k_i})$, written $\text{KEYPFS}(T_i)$. A *referring type assignment* for Σ is a mapping RTA from $\text{TABLES}(\Sigma)$ to referring expression types relative to Σ . The $\text{RTA}(T)$ for a table $T \in \text{TABLES}(\Sigma)$ is in the form (2.1).

□

The purpose served by associating an RET of the form (2.1) with a table T is to denote a set of SQLP queries that can serve as substitutes for **eid** values occurring in T 's **self** column which are called *referring expressions*. The following example illustrates how this would work.

Consider SUPERVISION, an ARM schema for a university that contains three tables LECTURER, PROFESSOR and GRAD, such that PROFESSOR has a preference clause, containing LECTURER. Table GRAD is also disjoint with both LECTURER and PROFESSOR. A table declaration for each table in the SUPERVISION is illustrated in Figure 2.5, and the following indicates each table's RTA :

$$\begin{aligned}
 \text{RTA(LECTURER)} &:= \text{LECTURER} \rightarrow \text{enum} = ? \\
 \text{RTA(PROFESSOR)} &:= \text{LECTURER} \rightarrow \text{enum} = ?; \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?) \\
 \text{RTA(GRAD)} &:= \text{GRAD} \rightarrow (\text{name} = ?, \text{supervisor.disc} = ?, \text{supervisor.f} = ?)
 \end{aligned}
 \tag{2.2}$$

Since there exists an **eid** foreign key **supervisor** from GRAD, referencing PROFESSOR, every GRAD object has a **supervisor** who is a PROFESSOR entity. The following SQLP query illustrates how to find “the names of lecturers who are also professors”:

$$\begin{aligned}
 &\text{select le.name} \\
 &\text{from LECTURER le, PROFESSOR pr} \\
 &\text{where le.self} = \text{pr.self}
 \end{aligned}
 \tag{2.3}$$

A sample database for SUPERVISION is illustrated in Figure 2.6 with four tuples for each table. Every **eid** attribute in LECTURER, PROFESSOR and GRAD can be replaced with a *referring expression*, resulting in the RM version of SUPERVISION, called SUPERVISION', illustrated in Figure 2.7 and the database for it is indicated in Figure 2.8.

```

table LECTURER (self eid, enum integer, name string, office integer, deptname
               string, primary key (enum))

table PROFESSOR (self eid, name string, office integer, deptname string
                 primary key (name, office),
                 preference (LECTURER))

table GRAD (self eid, name string, year integer, supervisor eid,
            foreign key (supervisor) references PROFESSOR,
            primary key (name, supervisor)),
            disjoint with (LECTURER, PROFESSOR)

```

Figure 2.5: SUPERVISION, an ARM schema for a university.

```

LECTURER (self eid, enum integer, name string, office integer, deptname string){
  (@eid1, 1345, "David", 321, "CS"),
  (@eid4, 4654, "Alice", 264, "CS"),
  (@eid8, 5217, "Julia", ,530, "ECE"),
  (@eid9, 2736, "Tom", 421, "Math")

PROFESSOR (self eid, name string, office integer, deptname string){
  (@eid1, "David", 321, "CS"),
  (@eid2, "Sara", 512, "Math"),
  (@eid3, "Jack", 105, "ECE"),
  (@eid4, "Alice", 264, "CS")

GRAD (self eid, name string, year integer, supervisor eid){
  (@eid5, "Fred" , 2, @eid1),
  (@eid6, "John", 3, @eid2),
  (@eid7, "Nancy", 4, @eid3),
  (@eid10, "Mia", 5, @eid4)}

```

Figure 2.6: A database for SUPERVISION.

2.2.1 Global Data

This Subsection introduces a set of global data, shown in Figure 2.9, which are used throughout the thesis. Assuming Σ is an ARM schema and Σ' is the RM version of Σ , some of these global data are defined on Σ and some on Σ' . The first global variable is

```

table LECTURER (self string, enum integer, name string, office integer, deptname
                string, primary key (enum))

table PROFESSOR (self string, name string, office integer, deptname string
                 primary key (name, office),
                 preference (LECTURER))

table GRAD (self string, name string, year integer, supervisor-professor string,
            foreign key (supervisor-professor) references PROFESSOR,
            primary key (name, supervisor),
            disjoint with (LECTURER, PROFESSOR))

```

Figure 2.7: SUPERVISION', the RM version of SUPERVISION.

$PK(T)$ which is defined to be the primary key of a table $T \in \text{TABLES}(\Sigma)$. The variable $\text{CONC}(T)$ is the concrete view of $T \in \text{TABLES}(\Sigma')$. The primary key of $\text{CONC}(T)$ is then illustrated with $\text{PKC}(T)$ which can be computed via the algorithm $\text{GENCONCRETEPK}(T)$ (2). This algorithm is further explained in Section 2.2.2.

Moreover, every table T of $\text{TABLES}(\Sigma)$ has a position in the preference order PRO , which we refer to as the $\text{OFFSET}(T)$. Assuming $\text{OFFSET}(T) = i$, then $\text{TABLE}(i)$ represents T . If T contains a preference clause, then the set of offsets of tables in its preference clause is defined by the $\text{PREF}(T)$. The number of offsets of tables in the $\text{PREF}(T)$ is defined by the $\text{PREFNUM}(T)$. For example, if T has a preference clause, containing three tables in it, the $\text{PREFNUM}(T)$ is 3. Next, the variable $\text{DISJ}(T)$ represents the set of offsets of those tables which T is disjoint with.

Recalling $\text{RTA}(T)$ in the form (2.1), each table name T_i of the i th component, where $1 \leq i \leq k$ is called $\text{GUARD}(\text{RTA}(T), i)$ and the set of paths $(\text{Pf}_{i,1}, \dots, \text{Pf}_{i,k_i})$ for each T_i in the i th component is defined as $\text{KEYPFS}(T_i)$. For example, $\text{GUARD}(\text{RTA}(T), k)$ is T_k and $\text{KEYPFS}(T_k)$ is $(\text{Pf}_{k,1}, \dots, \text{Pf}_{k,k_k})$. When $\text{PKC}(T)$ is $(\text{disc}, \mathbf{f})$, $\text{KEYPFS}(T)$ indicates the \mathbf{f} value. When the primary key is not $(\text{disc}, \mathbf{f})$, $\text{KEYPFS}(T)$ is the same as $\text{PKC}(T)$. Also, the variable $\text{TABLENAME}(T)$ denotes the string for the name of a table T .

Moreover, variables $\text{TT}(T)$ and $\text{TTSET}(T)$ which are defined in Definition 12 of Chapter 3, are introduced when translation tables might exist in an schema. $\text{TT}(T)$ contains

```

LECTURER (self string, enum integer, name string, office integer, deptname string){
    ("select x.self from LECTURER x where x.enum = 1345", 1345, "David", 321,
    "CS"),
    ("select x.self from LECTURER x where x.enum = 4654", 4654, "Alice", 264,
    "CS"),
    ("select x.self from LECTURER x where x.enum = 5217", 5217, "Julia", 530,
    "Math"),
    ("select x.self from LECTURER x where x.enum = 2736", 2736, "Tom", 421,
    "ECE")

PROFESSOR (self string, name string, office integer, deptname string){
    ("select x.self from LECTURER x where x.enum = 1345", "David", 321, "CS"),
    ("select x.self from PROFESSOR x where x.name = "Sara" and x.office = 512"
    , "Sara", 512, "Math"),
    ("select x.self from PROFESSOR x where x.name = "Jack" and x.office = 105"
    , "Jack", 105, "ECE"),
    ("select x.self from LECTURER x where x.enum = 4654", "Alice", 264, "CS")}

GRAD (self string, name string, year integer, supervisor-professor string){
    ("select x.self from GRAD x where x.name = "Fred" and x.supervisor.disc = 1
    and supervisor.f = 1345", "Fred" , 2, "select x.self from LECTURER x where
    x.enum = 1354"),

    ("select x.self from GRAD x where x.name = "John" and supervisor-professor.
    disc = 2 and supervisor-professor.f = ("Sara", 512)", "John", 3,
    "select x.self from PROFESSOR x where x.name = "Sara" and x.office =
    512"),

    ("select x.self from GRAD x where x.name = "Nancy" and supervisor-professor.
    disc = 2 and supervisor-professor.f = ("Jack", 105)", "Nancy", 4,
    "select x.self from PROFESSOR x where x.name = "Jack" and x.office =
    105"),

    ("select x.self from GRAD x where x.name = "Mia" and supervisor-professor.
    disc = 1 and supervisor-professor.f = 4654", "Mia", 5,
    "select x.self from LECTURER x where x.enum = 4654")}

```

Figure 2.8: A database for SUPERVISION' with eids replaced by strings.

the set of offsets $i < \text{OFFSET}(T)$, where $\text{OFFSET}(T') = i$ and there exist translation tables $T'-T$ -C. $\text{TTSET}(\Sigma)$ contains the set of all translation tables that exist in Σ .

2.2.2 ARM to RET Mapping

As noted earlier, one of our contributions is a proposal for a front-end for referring expression types in the form of PRIMARY KEY and PREFERENCE clauses which is discussed in this subsection. Let Σ be an ARM schema such that every table $T \in \text{TABLES}(\Sigma)$ is provided with at least one of the two clauses **primary key** (A_1, \dots, A_k) and **preference** (T_1, \dots, T_m) . This subsection introduces a procedure $\text{ARMtoRET}(\Sigma)$ which generates RETs for $\text{TABLES}(\Sigma)$ via the use of provided primary key and preference clauses. Primary key and preference graphs are part of the translation of primary key and preference clauses to RETs. They are needed to detect programming errors, and are crucial in defining RTA that maps tables names to RETs.

Definition 3 (Preference graph (PG))

The *preference graph* (PG) of an ARM schema Σ is a directed graph $G = (N, E)$ in which N has a node for each $T \in \text{TABLES}(\Sigma)$ and E has an edge (T_i, T_j) , if T_i contains a preference clause “**preference** (T_1, \dots, T_k) ”, where $T_j \in (T_1, \dots, T_k)$. For a PG that does not have any cycles, a preference order $\text{PRO} = (T_1, \dots, T_m)$ denotes an arbitrary topological sort of the PG in a reversed order, and $\text{OFFSET}(T)$ denotes the offset of table T in this order. \square

Definition 4 (Primary key graph (PKG))

The *primary key graph* (PKG) of an ARM schema Σ is a directed graph $G = (N, E)$ in which N has a node for each $T \in \text{TABLES}(\Sigma)$ and E has an edge (T_i, T_j) , if T_i contains a primary key clause “**primary key** (A_1, \dots, A_k) ”, such that there exists an **eid** attribute A_m in (A_1, \dots, A_k) , where $\text{RAN}(T_i, A_m) = T_j$. Suppose there exists an edge (T_i, T_j) in E , and there exists a table T_k in $\text{TABLES}(\Sigma)$ such that the edge (T_j, T_k) exists in the PG. Then, by transitivity, an edge (T_i, T_k) is added to the PKG. For a PKG that does not have any cycles, a primary key order $\text{PKO} = (T_1, \dots, T_n)$ denotes an arbitrary topological sort of PKG in a reversed order in which the most left table T_1 only includes incoming edge(s) and the most right table T_n only contains outgoing edge(s). \square

The algorithm $\text{ARMtoRET}(\Sigma)$ and the auxiliary algorithms it calls are defined in the

$\text{PK}(T)$:	A set of attributes (A_1, \dots, A_k) , where “ primary key (A_1, \dots, A_k) ” is a clause in T .
$\text{CONC}(T)$:	A concrete view of T containing its concrete primary key $\text{PKC}(T)$.
$\text{PKC}(T)$:	A primary key of the concrete view of T .
$\text{RTA}(T)$:	A referring type assignment for table T .
PRO :	A preference order on the set $\text{TABLES}(\Sigma)$, for an ARM schema Σ .
$\text{PRO}(\text{RTA})$:	A “global” referring expression type.
$\text{OFFSET}(T)$:	A position of T in PRO .
$\text{TABLE}(i)$:	$T \in \text{TABLES}(\Sigma)$ such that $\text{OFFSET}(T) = i$.
$\text{PREF}(T)$:	A set of offsets $i < \text{OFFSET}(T)$ for which $\text{TABLE}(i)$ exists in T ’s preference clause. Every offset in $\text{PREF}(T)$ includes a possible value for disc in the table $\text{CONC}(T)$.
$\text{PREFNUM}(T)$:	The number of offsets in the $\text{PREF}(T)$.
$\text{DISJ}(T)$:	A set of offsets $i < \text{OFFSET}(T)$ such that $\text{TABLE}(i)$ is disjoint with T .
$\text{GUARD}(\text{RTA}(T), i)$:	A table name “ T_i ” in the i th component of the $\text{RTA}(T)$.
$\text{KEYPFS}(T)$:	A set of path $(\text{Pf}_1, \dots, \text{Pf}_i)$, where $1 \leq i \leq k$, that is achieved by converting the $\text{PK}(T)$ to a concrete version.
$\text{TABLENAME}(T)$:	The string for the name of a table T .
$\text{TT}(T\text{-C})$:	A set of offsets $i < \text{OFFSET}(T)$ for which there is a table T' for which $\text{OFFSET}(T') = i$, and for which there is a translation table $T'\text{-}T\text{-C}$.
$\text{TTSET}(\Sigma)$:	A set of all translation tables that exist in Σ .
Org-TABS :	A global variable defined in Chapter 3 to store every table “ T z” in the “from” clause of the SQL query Q .

Figure 2.9: Global data

rest of this Subsection to illustrate how RETs are generated from the provided primary key and preference clauses.

The algorithm $\text{ARMtoRET}(\Sigma)$ first checks if a PRO and a PKO can be defined on a preference graph and primary key graph respectively. Then, for every table T in the PRO, the global data $\text{OFFSET}(T)$, $\text{PK}(T)$, $\text{PREF}(T)$, $\text{TABLE}(\text{OFFSET}(T))$ and $\text{DISJ}(T)$ which are defined in Figure 2.9 are generated. Afterwards, $\text{ARMtoRET}(\Sigma)$ calls $\text{GENCONCRETEPK}(T)$ for every T in the PKO to generate the $\text{PKC}(T)$, and calls the $\text{GENRET}(T)$ for every T in the PRO to generate the $\text{RTA}(T)$.

Algorithm 1

```

procedure  $\text{ARMtoRET}(\Sigma)$ 
  if PRO is not defined by the PG of  $\Sigma$  then
    return error
  if PKO is not defined by the PKG of  $\Sigma$  then
    return error
  for every  $T$  in PRO, find the following global data
    if there does not exist any primary key clause in  $T$  then
       $\text{PREF}(T) := (1, \dots, m)$ , where “preference ( $T_1, \dots, T_m$ )” is a clause in  $T$ .
    if there does not exist any preference clause in  $T$  then
       $\text{PK}(T) := (A_1, \dots, A_k)$ , where “primary key ( $A_1, \dots, A_k$ )” is a clause in  $T$ .
    else
       $\triangleright$  Both primary key and preference clauses exist
       $\text{PK}(T) := (A_1, \dots, A_k)$ , where “primary key ( $A_1, \dots, A_k$ )” is a clause in  $T$ .
       $\text{PREF}(T) := (1, \dots, m)$ , where “preference ( $T_1, \dots, T_m$ )” is a clause in  $T$ .
      • Find the position of  $T$  in the PRO and assign the  $\text{OFFSET}(T)$  to this integer
      •  $\text{TABLE}(\text{OFFSET}(T)) := T$ 
      •  $\text{DISJ}(T) := (T'_1, \dots, T'_k)$ , where  $T$  contains the clause “disjoint from ( $T'_1, \dots, T'_k$ )”.
  for every  $T$  in the PKO
    Call  $\text{GENCONCRETEPK}(T)$ 
  for every  $T$  in the PRO
    Call  $\text{GENRET}(T)$ 

```

Procedure $\text{GENCONCRETEPK}(T)$ generates $\text{PKC}(T)$ as well as $\text{KEYPFS}(T)$.

There are four cases which the algorithm considers:

1. T does not have a primary key clause, but T contains the “**isa** T_j ” constraint, and T_j is in T ’s preference clause. In this case, $\text{PKC}(T)$ is the same as $\text{KEYPFS}(T)$ and they are both the same as $\text{PKC}(T_j)$.
2. T does not have a primary key clause, but T contains the “**cover by** (T_1, \dots, T_j) ” constraint, and the “**preference** (T_1, \dots, T_j) ” clause. In this case, $\text{PKC}(T)$ is $(\text{disc}, \mathbf{f})$ and $\text{KEYPFS}(T)$ is empty.
3. T contains only a primary key and does not have any preference clauses. In this case, $\text{PKC}(T)$ is the same as $\text{KEYPFS}(T)$. For each attribute A_i in $\text{PK}(T)$, if A_i is abstract and $\text{RAN}(T, A_i)$ is T_j , for each Pf_j in $\text{PKC}(T_j)$, A_i gets replaced by $A_i \circ \text{Pf}_j$. We introduce the composition operator \circ as follows:

$$\text{Pf}_1 \circ \text{Pf}_2 \equiv \begin{cases} \text{Pf}_1, & \text{if } \text{Pf}_2 \text{ is } \mathbf{self} \\ \text{Pf}_2, & \text{if } \text{Pf}_1 \text{ is } \mathbf{self} \\ \text{Pf}_1 . \text{Pf}_2, & \text{otherwise} \end{cases}$$

4. T contains both a primary key clause and a preference clause. $\text{PKC}(T)$ is then $(\text{disc}, \mathbf{f})$ and the $\text{KEYPFS}(T)$ is the value of \mathbf{f} which is computed as follows: for each attribute A_i in $\text{PK}(T)$, if A_i is abstract and $\text{RAN}(T, A_i)$ is T_j , for each Pf_j in $\text{PKC}(T_j)$, A_i is replaced by $A_i \circ \text{Pf}_j$.

Procedure $\text{GENRET}(T)$ finds the $\text{RTA}(T)$, according to two checks:

If $\text{PKC}(T)$ is $(\text{disc}, \mathbf{f})$, each component of $\text{RTA}(T)$ would be built by a table name T' and $\text{KEYPFS}(T')$, where $\text{OFFSET}(T')$ exists in $\text{PREF}(T)$. Otherwise, $\text{RTA}(T)$ is built by the table name T itself and its $\text{KEYPFS}(T)$. Function $\text{PRINTKEYPFS}(T)$ prints $\text{KEYPFS}(T')$.

Algorithm 2

```
procedure GENCONCRETEPK ( $T$ )
  if PK( $T$ ) is empty and  $T$  isa  $T_j$  and PREF ( $T$ ) = OFFSET( $T_j$ ) then
    PK( $T$ ) := PK( $T_j$ )
    KEYPFS( $T$ ) := PK( $T$ )
  if PK( $T$ ) is empty and  $T$  contains cover by  $(T_1, \dots, T_j)$  and
  PREF( $T$ ) =  $(T_1, \dots, T_j)$  then
    PK( $T$ ) := Concat (PK( $T$ ), "disc", "f")
    KEYPFS( $T$ ) := () ▷ KEYPFS is empty
  if PREF ( $T$ ) is empty then
    for each column  $A_i$  in PK( $T$ )
      if  $A_i$  is in ABS( $T$ ) and RAN( $T, A_i$ ) =  $T_j$  then
        for each path Pf $j$  in PKC( $T_j$ )
          PK( $T$ ) := Concat(PK( $T$ ),  $A_i \circ$  Pf $j$ )
          KEYPFS( $T$ ) = PK( $T$ )
        else ▷ Concrete attribute
          PK( $T$ ) := Concat(PK( $T$ ),  $A_i$ )
          KEYPFS( $T$ ) := PK( $T$ )
    else
      PK( $T$ ) = Concat(PK( $T$ ), "disc", "f")
      for each column  $A_i$  in PK( $T$ )
        if  $A_i$  is in ABS( $T$ ) and RAN( $T, A_i$ ) =  $T_j$  then
          for each path Pf $j$  in PKC( $T_j$ )
            KEYPFS( $T$ ) := Concat (KEYPFS( $T$ ),  $A_i \circ$  Pf $j$ )
          else ▷ Concrete attribute
            KEYPFS( $T$ ) := Concat (KEYPFS( $T$ ),  $A_i$ )
```

Algorithm 3

```
procedure GENRET ( $T$ )
  RTA( $T$ ) = ""
  if PKC( $T$ ) = "(disc, f)" then ▷  $T$  contains a preference clause
    x = 0 ▷ number of tables in the preference clause
    for each offset i in PREF( $T$ )
      GUARD(RTA( $T$ ), x) := TAB(i)
      if x < PREFNUM( $T$ ) then
        RTA( $T$ ) := Concat(RTA( $T$ ), RTA(TAB(i)), ";")
      else
        RTA( $T$ ) := Concat(RTA( $T$ ), RTA(TAB(i)))
    x := x + 1
  if KEYPFS( $T$ ) is not empty then ▷ In addition to a preference clause,
    ▷  $T$  contains a primary key clause
    RTA( $T$ ) := Concat(RTA( $T$ ), ";", PRINTKEYPFS( $T$ )) ▷ The last component
    is  $T$  itself
  else ▷  $T$  only contains a primary key clause
    RTA( $T$ ) := Concat(RTA( $T$ ), PRINTKEYPFS( $T$ ))
```

Algorithm 4

```
function PRINTKEYPFS ( $T$ )  
  RTA( $T$ ) := Concat("T", "→", "(")  
  for each  $Pf_i$  in KEYPFS( $T$ )  
    if  $Pf_i$  is not the last path attribute in KEYPFS( $T$ ) then  
      RTA( $T$ ) := Concat(RTA( $T$ ),  $Pf_i$ , "=", "?", ",", "  
    else  
      RTA( $T$ ) := Concat(RTA( $T$ ),  $Pf_i$ , "=", "?")  
  RTA( $T$ ) := Concat(RTA( $T$ ), ")")  
  return RTA( $T$ )
```

Considering UNIV in Figure 2.3 again, we illustrate how procedure ARMtoRET(UNIV) proceeds to find RTA(T) for each T in TABLES(UNIV). Table declarations for TABLES(UNIV) are illustrated in Appendix A.1. The only tables with preference clauses are STUDENT and PERSON and the only table that does not contain any primary key clause is PERSON. Both tables STUDENT and PROFESSOR are subsets of PERSON and they are also covered by PERSON.

ARMtoRET(UNIV) first generates a preference graph PG for UNIV, as indicated in Figure 2.10, where there does not exist any preference cycle. A possible topological sort of PG in the reverse order results in the following preference order:

PRO = (COURSE, DEPARTMENT, ENROLLMENT, CLASS, PROFESSOR, STUDENT, PERSON)

Thereafter, ARMtoRET(UNIV) generates a primary key graph PKG for UNIV, as illustrated in Figure 2.11, such that there does not exist any primary key cycle in PKG either. A topological sort of PKG in the reverse order results in the following primary key order:

Pko = (PERSON, PROFESSOR, STUDENT, DEPARTMENT, COURSE, CLASS, ENROLLMENT)

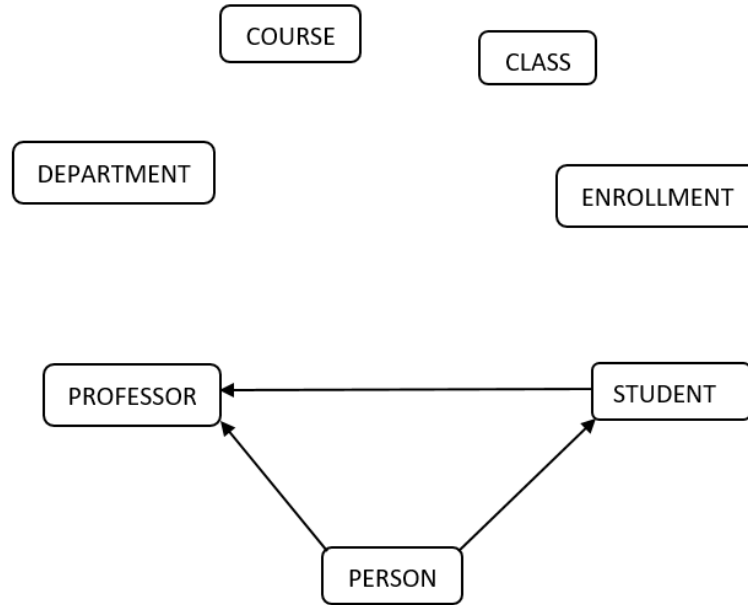


Figure 2.10: PG for UNIV

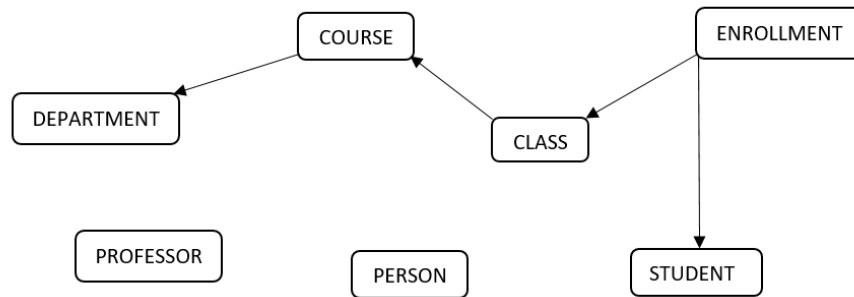


Figure 2.11: PKG for UNIV

ARMtoRET(UNIV) then iterates through every table T in PRO to find $\text{Pk}(T)$, $\text{Pref}(T)$, $\text{Offset}(T)$, $\text{Table}(\text{Offset}(T))$, and $\text{Disj}(T)$ which are illustrated in Figure 2.12 for COURSE, DEPARTMENT, ENROLLMENT, CLASS, PROFESSOR, STUDENT and PERSON.

```

Pk(COURSE) = (cnum, department)
Pref(COURSE) = ()
Offset(COURSE) = 1
TABLE(2) = COURSE
Disj(COURSE) = (DEPARTMENT, ENROLLMENT, CLASS, PROFESSOR, STUDENT, PERSON)

Pk(DEPARTMENT) = deptcode
Pref(DEPARTMENT) = ()
Offset(DEPARTMENT) = 2
TABLE(1) = DEPARTMENT
Disj(DEPARTMENT) = (COURSE, ENROLLMENT, CLASS, PROFESSOR, STUDENT, PERSON)

Pk(ENROLLMENT) = (student, class)
Pref(ENROLLMENT) = ()
Offset(ENROLLMENT) = 3
TABLE(4) = ENROLLMENT
Disj(ENROLLMENT) = (COURSE, DEPARTMENT, CLASS, PROFESSOR, STUDENT, PERSON)

Pk(CLASS) = (course, term, section)
Pref(CLASS) = ()
Offset(CLASS) = 4
TABLE(3) = CLASS
Disj(CLASS) = (COURSE, DEPARTMENT, ENROLLMENT, PROFESSOR, STUDENT, PERSON)

Pk(PROFESSOR) = (name, office)
Pref(PROFESSOR) = ()
Offset(PROFESSOR) = 5
TABLE(5) = PROFESSOR
Disj(PROFESSOR) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

Pk(STUDENT) = (snum)
Pref(STUDENT) = (5)
Offset(STUDENT) = 6
TABLE(6) = STUDENT
Disj(STUDENT) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

Pk(PERSON) = ()
Pref(PERSON) = (5, 6)
Offset(PERSON) = 7
TABLE(7) = PERSON
Disj(PERSON) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

```

Figure 2.12: Global data for all tables in UNIV

Thereafter, $\text{ARMtoRET}(\text{UNIV})$ iterates through every table T in PKO and calls procedure $\text{GENCONCRETEPK}(T)$ to generate $\text{PKC}(T)$ and $\text{KEYPFS}(T)$, as illustrated further:

$\text{PKC}(\text{PERSON}) = (\text{disc}, \text{f})$
 $\text{KEYPFS}(\text{PERSON}) = ()$

$\text{PKC}(\text{PROFESSOR}) = (\text{name}, \text{office})$
 $\text{KEYPFS}(\text{PROFESSOR}) = (\text{name}, \text{office})$

$\text{PKC}(\text{STUDENT}) = (\text{disc}, \text{f})$
 $\text{KEYPFS}(\text{STUDENT}) = (\text{snum})$

$\text{PKC}(\text{DEPARTMENT}) = \text{deptcode}$
 $\text{KEYPFS}(\text{DEPARTMENT}) = \text{deptcode}$

$\text{PKC}(\text{COURSE}) = (\text{cnum}, \text{department.deptcode})$
 $\text{KEYPFS}(\text{COURSE}) = (\text{cnum}, \text{department.deptcode})$

$\text{PKC}(\text{CLASS}) = (\text{course.cnum}, \text{course.department.deptcode}, \text{term}, \text{section})$
 $\text{KEYPFS}(\text{CLASS}) = (\text{course.cnum}, \text{course.department.deptcode}, \text{term}, \text{section})$

$\text{PKC}(\text{ENROLLMENT}) = (\text{student.snum}, \text{class.course.cnum}, \text{class.course.department.deptcode}, \text{class.term}, \text{class.section})$
 $\text{KEYPFS}(\text{ENROLLMENT}) = (\text{student.snum}, \text{class.course.cnum}, \text{class.course.department.deptcode}, \text{class.term}, \text{class.section})$

Then, $\text{ARMtoRET}(\text{UNIV})$ iterates through every T in PRO and calls procedure $\text{GENRET}(T)$ to generate $\text{RTA}(T)$, as shown below:

$\text{RTA}(\text{COURSE}) := \text{COURSE} \rightarrow (\text{cnum} = ?, \text{department.deptcode} = ?)$
 $\text{RTA}(\text{DEPARTMENT}) := \text{DEPARTMENT} \rightarrow \text{deptcode} = ?$
 $\text{RTA}(\text{ENROLLMENT}) := \text{ENROLLMENT} \rightarrow (\text{student.disc} = ?, \text{student.f} = ?, \text{class.}$
 $\quad \text{course.cnum} = ?, \text{class.course.department.deptcode} = ?,$
 $\quad \text{class.term} = ?, \text{class.section} = ?)$
 $\text{RTA}(\text{CLASS}) := \text{CLASS} \rightarrow (\text{course.cnum} = ?, \text{course.department.deptcode} = ?,$
 $\quad \text{term} = ?, \text{section} = ?)$
 $\text{RTA}(\text{PROFESSOR}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?)$
 $\text{RTA}(\text{STUDENT}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?,$
 $\text{RTA}(\text{PERSON}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?$

2.3 On Identity Resolution

Clearly, not all RTA assignments will work. For example, unless the `deptcode` of `DEPARTMENT` is unique, the RTA assignment for `DEPARTMENT` in Appendix A.3 is insufficient. We now consider when an RTA assignment qualifies as *well-formed*. We then show how satisfying this condition ultimately leads to an intuitive notion of *identity resolution*, in particular, the assurance that any SQL query over an ARM schema can be mapped to an equivalent query over a related RM schema.

Definition 5 (Well-Formed RTA)

Let Σ be an ARM schema and RTA a referring type assignment for Σ . Given a preference order $\text{PRO} := (T_{i_1}, \dots, T_{i_k})$ on the set $\text{TABLES}(\Sigma)$, define $\text{PRO}(\text{RTA})$ as the following referring expression type:

$$\text{RTA}(T_{i_1}); \dots; \text{RTA}(T_{i_k})$$

We say that RTA is *well-formed* with respect to PRO if the following conditions also hold for each $T \in \text{TABLES}(\Sigma)$.

1. $\text{RTA}(T) = \text{PRUNE}(\text{PRO}(\text{RTA}), T),$

2. $\Sigma \models ((\text{cover by } \{T_1, \dots, T_n\}) \in T)$, where $\{T_1, \dots, T_n\}$ are all tables occurring in the guards in the $\text{RTA}(T)$, and
3. for each component $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ of $\text{RTA}(T)$, the following also holds: (i) $\text{Pf}_{j,i}$ is well defined for T_j , for $1 \leq i \leq k_j$, and (ensuring *strong identification*) (ii) $\Sigma \models ((\text{path functional dependency } (\text{Pf}_{j,1}, \dots, \text{Pf}_{j,k_j}) \text{ determines self}) \in T_j)$.

□

The first condition of a well-formed RTA which is $\text{PRUNE}(\text{PRO}(\text{RTA}), T)$ is presented in Algorithm 5. It prunes $\text{PRO}(\text{RTA})$ with respect to T , by removing the j th component “ $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ ” of $\text{PRO}(\text{RTA})$ if one of the following condition is true:

1. T_j is covered by $\{T_1, \dots, T_{j-1}\}$
2. T is covered by $\{T_1, \dots, T_{j-1}\}$
3. T is disjoint from T_j

Algorithm 5

```

function PRUNE (PRO(RTA), T)
  pruneResult := PRO(RTA)
  if ( $\Sigma = (\text{cover by } \{T_1, \dots, T_{j-1}\}) \in T_j$ ) or
     $\Sigma = (\text{cover by } \{T_1, \dots, T_{j-1}\}) \in T$ ) or
     $\Sigma = (\text{disjoint with } T_j) \in T$ ) then
    remove the  $j$ th component “ $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ ” from pruneResult
  return pruneResult

```

The second condition of a well-formed RTA states that when tables $\{T_1, \dots, T_n\}$ exist in the guards of the $\text{RTA}(T)$, every **self** attribute in T must exist in at least one of the tables in $\{T_1, \dots, T_n\}$. Finally, the third condition of a well-formed RTA contains two parts:

1. $\text{Pf}_{j,i}$ must exist as an attribute in T_j . For example, suppose $\text{RTA}(\text{STUDENT}) := \text{address} = ?$, where the attribute **address** does not exist in **STUDENT**. Thus, it is not well defined and $\text{RTA}(\text{STUDENT})$ is not then well-formed.
2. $\text{Pf}_{j,i}$ must be unique for each **self** entity. In other words, there should not exist two **self** tuples with the same $\text{Pf}_{j,i}$. For example, suppose $\text{RTA}(\text{STUDENT}) := \text{name} = ?$. There can exist two or more students with the name “David”, so **name** does not strongly identify **STUDENT** and, therefore, $\text{RTA}(\text{STUDENT})$ is not well-formed.

Considering UNIV, the following illustrates the $\text{PRO}(\text{RTA})$:

$$\begin{aligned} \text{PRO}(\text{RTA}) := & \\ & \text{RTA}(\text{DEPARTMENT}); \text{RTA}(\text{COURSE}); \text{RTA}(\text{CLASS}); \text{RTA}(\text{DEPARTMENT}); \quad (2.4) \\ & \text{RTA}(\text{PROFESSOR}); \text{RTA}(\text{STUDENT}); \text{RTA}(\text{PERSON}) \end{aligned}$$

$\text{PRUNE}(\text{PRO}(\text{RTA}), T)$ for each T in $\text{TABLES}(\text{UNIV})$ is the same as the $\text{RTA}(T)$. For example, $\text{PRUNE}(\text{PRO}(\text{RTA}), \text{STUDENT})$ which is indicated as follows, is the same as the $\text{RTA}(\text{STUDENT})$:

$$\begin{aligned}
& \text{PRUNE}(\text{PRO}(\text{RTA}), \text{STUDENT}) = \\
& \text{PRUNE}((\text{DEPARTMENT} \rightarrow \text{deptcode} = ?; \\
& \quad \text{COURSE} \rightarrow (\text{cnum} = ?, \text{department.deptcode} = ?); \\
& \quad \text{CLASS} \rightarrow (\text{course.cnum} = ?, \text{course.department.deptcode} = ?, \text{term} = ?, \\
& \quad \quad \text{section} = ?); \\
& \quad \text{ENROLLMENT} \rightarrow (\text{student.snum} = ?, \text{class.course.cnum} = ?, \text{class.course.} \\
& \quad \quad \text{department.deptcode} = ?, \text{class.term} = ?, \text{class.} \\
& \quad \quad \text{section} = ?); \\
& \quad \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \\
& \quad \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?; \\
& \quad \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?), \text{STUDENT}) = \\
& \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?
\end{aligned} \tag{2.5}$$

Therefore, the first condition of a well-formed RTA is satisfied for UNIV. The second condition is also met for each T in $\text{TABLES}(\text{UNIV})$. For example, the table **PERSON** is covered by **PROFESSOR** and **STUDENT** and these two occur in the guards of the $\text{RTA}(\text{PERSON})$ (Appendix A.1). The third case is satisfied for each T in $\text{TABLES}(\text{UNIV})$ as well, since the $\text{RTA}(T)$ ensures strong identification.

2.3.1 A New Form of RET

A new form of (RET) *relative to* Σ , where Σ is an ARM schema can be illustrated in the following form in addition to the already introduced form in (2.1):

$$\begin{aligned}
\text{RET} ::= & T_i \rightarrow (\text{COND}_{i_1}, \dots, \text{COND}_{i_k}) \\
& | \quad \text{RET}; T_i \rightarrow (\text{COND}_{i_1}, \dots, \text{COND}_{i_k}) \\
\text{COND} ::= & \text{Pf} = ? \\
& | \quad \text{Pf.disc} = ?, \text{Pf.f} = ?
\end{aligned} \tag{2.6}$$

Considering the ARM schema SUPERVISION,

$\text{RTA}(\text{GRAD}) := \text{GRAD} \rightarrow (\text{name} = ?, \text{supervisor.disc} = ?, \text{supervisor.f} = ?),$

is in both the form (2.1) and (2.6). The referring expressions for $\text{RTA}(\text{GRAD})$ which have replaced the **self** column of **GRAD** in Figure 2.8, contain “**supervisor.disc** = c_1 and **supervisor.f** = c_2 ”, where c_1 and c_2 are constants. However, Since **disc** and **f** do not exist in the ARM schema Σ , such referring expressions can not be compiled on Σ . Therefore, for every $\text{RTA}(T)$ in Σ , each occurrence of “**Pf.disc** =?, **Pf.f** =?” is replaced by $\text{Pf} : \text{RTA}(\text{RAN}(T, \text{Pf}))$, and the **RET** form in (2.6) is changed to another form which is illustrated in the following definition:

Definition 6 (The New Referring Expression Types)

Let Σ be an ARM *schema*. A new *referring expression type* (**RET**) relative to Σ has the following form:

$$\begin{aligned} \text{RET} &::= T_i \rightarrow (\text{COND}_{i_1}, \dots, \text{COND}_{i_k}) \\ &\quad | \quad \text{RET}; T_i \rightarrow (\text{COND}_{i_1}, \dots, \text{COND}_{i_k}) \\ & \\ \text{COND} &::= \text{Pf} = ? \\ &\quad | \quad \text{Pf} : \text{RET}, \end{aligned} \tag{2.7}$$

where an **RET** consists of a sequence of “;” separated *components*. Each component $T_i \rightarrow (\text{COND}_{i_1}, \dots, \text{COND}_{i_k})$ consists in turn of a *guard*, written $\text{GUARD}(\text{RET}, i)$, which is the name T_i of a table in $\text{TABLES}(\Sigma)$ and each **COND** which is in the form $\text{Pf} = ?$ or $\text{Pf} : \text{RET}$.

□

Therefore, the $\text{RTA}(\text{GRAD})$ is now updated to the following:

$\text{RTA}(\text{GRAD}) := \text{GRAD} \rightarrow (\text{name} = ?,$
 $\quad \text{supervisor-professor} : \text{RTA}(\text{RAN}(\text{GRAD}, \text{supervisor-professor})))$
 $:= \text{GRAD} \rightarrow (\text{name} = ?,$
 $\quad \text{supervisor-professor} : \text{RTA}(\text{PROFESSOR})),$
 $:= \text{GRAD} \rightarrow (\text{name} = ?,$

supervisor-professor : (LECTURER \rightarrow enum = ? ;
 PROFESSOR \rightarrow (name = ?, office = ?))

Definition 7 (Referring Expressions)

Let DB denotes a database over an ARM schema Σ . Suppose an Ret is a special RET of the form (2.7) and $Cond$ is also a special COND of the form (2.7). A set of *referring expressions* for every Ret relative to the DB , written $RE(Ret, DB)$ and $EXTENDCOND(Cond, DB)$ have the form in Figure 2.8. The $EVAL(Re, DB)$ executes a special Re on DB and returns a table in which the **self** column contains an **eid** attribute corresponding to the Re . For example, if the Re is “select * from LECTURER x where x.enum = 1345”, the $EVAL(Re, DB)$ returns a table with the **self** column which contains the **eid** value of a lecturer whose **enum** is 1345.

$$\begin{aligned} RE("T \rightarrow (Cond_1, \dots, Cond_k)", DB) = \\ \{Q \mid Q = \text{"select x.self from } T \text{ x where } C_1 \text{ and } \dots \text{ and } C_k" \\ \text{where } C_i \in EXTENDCOND(Cond_i, DB), 1 \leq i \leq k, \\ \text{and } |EVAL(Q, DB)| = 1\}, \end{aligned}$$

$$\begin{aligned} RE("Ret' ; T \rightarrow (Cond_1, \dots, Cond_k)", DB) = \\ RE(Ret', DB) \cup \{Q_2 \in RE("T \rightarrow (Cond_1, \dots, Cond_k)", DB) \\ \mid \text{not exists } Q_1 \in RE(Ret', DB) \text{ s.t.} \\ EVAL("Q_1", DB) = EVAL("Q_2", DB)\}, \end{aligned} \tag{2.8}$$

$$EXTENDCOND("Pf = ?", DB) = \{“x.Pf = c” \mid c \text{ occurs as a constant in } DB\},$$

$$EXTENDCOND("Pf : Ret'", DB) = \{“x.Pf in (Q)” \mid Q \in RE(Ret', DB)\}.$$

□

According to the definition of referring expressions, the referring expressions for tables PROFESSOR and LECTURER of SUPERVISION remain the same as shown in Figure 2.8,

whereas the referring expression for **GRAD** is changed to the following general form below, where c_i is a constant:

$\text{RE}(\text{RTA}(\text{GRAD}), DB) =$

```
{ "select x.self from GRAD x where x.name =  $c_1$  and
    x.supervisor-professor in (select x.self from LECTURER x where
    x.enum =  $c_2$ ) "}
```

or

```
"select x.self from GRAD x where x.name =  $c_1$  and
    x.supervisor-professor in (select x.self from PROFESSOR x where
    x.name =  $c_3$  and x.office =  $c_4$ ) "}
```

The actual $\text{RE}(\text{RTA}(\text{GRAD}), DB)$ is then modified as follows:

$\text{RE}(\text{RTA}(\text{GRAD}), DB) =$

```
{ ("select x.self from GRAD x where x.name = "Fred" and
    x.supervisor-professor in (select x.self from LECTURER x where x.enum
    = 1345)"),
```

```
("select x.self from GRAD x where x.name = "John" and
    x.supervisor-professor in (select x.self from PROFESSOR x where x.name
    = Sara and x.office = 512)"),
```

```
("select x.self from GRAD x where x.name = "Nancy" and
    x.supervisor-professor in (select x.self from PROFESSOR x where x.name
    = Tom and x.office = 418)"),
```

```
("select x.self from GRAD x where x.name = "Mia" and
    x.supervisor-professor in (select x.self from LECTURER x where x.enum
    = 4654)"))}
```

(2.9)

Therefore, the correct table declarations for `LECTURER`, `PROFESSOR` and `GRAD` in the RM schema are now illustrated in Figure 2.13. When converting the SQLP query in (2.3) to an SQLA, the query is not changed and remains the same, since the `self` columns of type `string` now exist in the schema.

Proposition 8 Let Σ be an ARM schema and RTA is a well-formed referring type assignment for Σ . For every table T in $\text{TABLES}(\Sigma)$ and every database DB over Σ , there exists exactly one Re in the $\text{RE}(\text{RTA}(T), DB)$ for each `eid` occurring in the `self` column of T such that the following is true:

$$\text{EVAL}(Re, DB) = \{(\text{eid})\}$$

Definition 9 (A mapping from `eid` to `string`)

Given a database DB , with schema Σ , let M denotes a mapping from `eid` occurring in the DB to `strings`. We also write $M(Tab)$ to denote a table Tab' obtained from a given table Tab , where each `eid` in Tab is replaced by $M(\text{eid})$ and also write $M(DB)$ to denote a database DB' with schema Σ' , where Σ' is obtained from Σ by replacing any occurrence of the domain `eid` with the domain `string` and any table Tab in DB with table $M(Tab)$

Definition 10 (Identity Resolving Type Assignments)

Let Σ be an ARM schema and DB denotes a database over Σ . A mapping M is identity resolving if

$$M(\text{EVAL}(Q, DB)) = \text{EVAL}(Q, M(DB)) \quad (2.10)$$

holds for every SQLA query Q over the DB . □

An *identity issue* is the problem of compiling any query Q that contains $x.A = y.\text{self}$, where each side is an `eid`. In order to compile such queries, each side of the equality should be replaced by a string which we indicate it as $M(x.A) = M(y.\text{self})$, where M has to be identity resolving.

Proposition 11 Given a database DB with schema Σ and a well-formed RTA, there exists an M that is identity resolving.

```

LECTURER (self string, enum integer, name string, office integer, deptname string)
    = {("select x.self from LECTURER x where x.enum = 1345", 1345, "David",
321,"CS"),
("select x.self from LECTURER x where x.enum = 4654", 4654, "Alice", 264,
"CS"),
("select x.self from LECTURER x where x.enum = 5217", 5217, "Julia", 412,
"Math"),
("select x.self from LECTURER x where x.enum = 2736", 2736, "Tom", 418,
"ECE")}]

PROFESSOR (self string, name string, office integer, deptname string) = {
("select x.self from LECTURER x where x.enum = 1345", "David", 321,
"CS"),
("select x.self from PROFESSOR x where x.name = "Sara" and x.office =
512", "Sara", 512, "Math"),
("select x.self from PROFESSOR x where x.name = "Tom" and x.office =
418", "Tom", 418, "ECE"),
("select x.self from LECTURER x where x.enum = 4654", "Alice", 264,
"CS")}]

GRAD (self string, name string, year integer, supervisor string) = {
("select x.self from GRAD x where x.name = "Fred" and
x.supervisor in (select x.self from LECTURER x where x.enum = 1345)),
"Fred" , 2, "select x.self from LECTURER x where x.enum = 1354"),

("select x.self from GRAD x where x.name = "John" and
x.supervisor in (select x.self from PROFESSOR x where x.name = "Sara" and
x.office = 512)", "John", 3, "select x.self from PROFESSOR x where x.name =
"Sara" and x.office = 512"),

("select x.self from GRAD x where x.name = "Nancy" and
x.supervisor in (select x.self from PROFESSOR x where x.name = "Tom" and
x.office = 418)", "Nancy", 4, "select x.self from PROFESSOR x where
x.name = "Tom" "),

("select x.self from LECTURER x where x.enum = 4654", "Alice", 264, "CS'',
"Mia", 5, "select x.self from LECTURER x where x.enum = 4654"))}

```

Figure 2.13: The correct version of SUPERVISION' with eids replaced by strings.

The proof is a simple consequence of Proposition 8 such that M maps **eids** to referring expressions. In particular, for any **eid** occurring in a **self** column of table T in Σ , $M(\mathbf{eid})$ is *the unique* referring expression Re in $RE(RTA(T), DB)$ that must exist by Proposition 8, and for which $EVAL(Re, DB) = \{(\mathbf{eid})\}$.

2.4 Mapping an ARM schema to an RM schema

In Section 2.2, we have illustrated how **eids** in each table T in an ARM schema Σ are replaced by referring expressions such that Σ is converted to an RM schema Σ' . Figure 2.13 indicates a new database for SUPERVISION', where **eids** in SUPERVISION are replaced by correct referring expressions. Although the identity issues have been resolved via the new RM schema that has been achieved by replacing all the **eids** with referring expressions, storing such long strings is not efficient. Therefore, in converting an ARM schema Σ to an RM one, we remove the **self** column for each T in $TABLES(\Sigma)$ and replace it with $PKC(T)$ which can be a combination of **disc** and **f**. Also, for any **eid** foreign key A that references another table T' , A is replaced with $PKC(T')$. For example, the following illustrates a database for SUPERVISION', where the **self** columns have been removed, and columns **disc** and **f** are added, as required.

```
LECTURER (enum integer, name string, office integer, deptname string) = {
    (1345, "David", 321, "CS"),
    (4654, "Alice", 264, "CS"),
    (5217, "Julia", 412, "Math"),
    (2736, "Tom", 418, "ECE")}
```

```
PROFESSOR (disc integer, f string, name string, office integer, deptname
string) = {
    (1, "1345", "David", 321, "CS"),
    (2, "Sara 512 ", "Sara", 512, "Math"),
    (2, "Tom 418 ", "Tom", 418, "ECE"),
    (1, "4654", "Alice", 264, "CS")}
```

```
GRAD (name string, supervisor-disc integer, supervisor-f string, year
integer) = {
    ("Fred", 1, "1345" , 2),
    ("John", 2, "Sara 512", 3),
    ("Nancy", 2, "Tom 418" 4),
    ("Mia", 1, "4654", 5)}
```

In this section, function $\text{ARMtORM}(\Sigma)$ is introduced in Algorithm 6 to map an ARM schema Σ to an RM schema Σ' by finding a table declaration for each table T in $\text{TABLES}(\Sigma)$. $\text{ARMtORM}(\Sigma)$ calls procedure $\text{ARMtORET}(\Sigma)$ to generate $\text{RTA}(T)$ for each T in $\text{TABLES}(\Sigma)$. Then, if RTA is well-formed for Σ , $\text{ARMtORM}(\Sigma)$ starts generating a concrete table declaration for each T in Σ' . First, a primary key clause is generated for T . Then, every concrete attribute A_i of T , in addition to its domain, are added to the table declaration of T . If A_i is an `eid` attribute, and it references another table T' , string “ A_i- ” is concatenated to each element of $\text{PKC}(T')$, in addition to the domain of that element, and the result is then added to the table declaration of T . A foreign key clause is also generated at this point and added to the table declaration of T .

$\text{ARMtORM}(\Sigma)$ calls functions $\text{GENNAME}(\text{Pf})$ and $\text{PATHDOM}(T, \text{Pf})$. $\text{GENNAME}(\text{Pf})$ which is illustrated in Algorithm 8 inputs a path Pf in the form $B_1 \cdots B_k$ and converts

it to the form $B_1 \dots B_k$. $\text{PATHDOM}(T, \text{Pf})$ which is displayed in Algorithm 9 inputs a table T and a path Pf that exists as an attribute in T . Pf is assumed to be in the form $A_1 \dots A_j$, and $\text{PATHDOM}(T, \text{Pf})$ finds the domain of Pf and the table that Pf is referencing.

Function $\text{ADDNEWATTR}(A_i, T, T_j)$ which is indicated in Algorithm 7 inputs an abstract attribute A_i in a table T , such that it references a table T_j . This function concatenates “ A_i .” with each path Pf_j in $\text{PKC}(T_j)$ and generates a new attribute, and adds it to $\text{ATTRS}(\text{CONC}(T))$. Next, this function prefixes every path Pf_j in $\text{PKC}(T_j)$ with A_i , and adds the result path, along with the domain type of Pf_j to a string called table-decl. A foreign key clause including the combination of a prefix A_i and each path Pf_j in $\text{PKC}(T_j)$ is also added to table-decl. Figure 2.14 illustrates a graph for both algorithms and definitions that are introduced to map an ARM schema to an RM schema.

Algorithm 6

```
function ARMToRM ( $\Sigma$ )
  Call ARMToRET( $\Sigma$ ) to generate RTA( $T$ ) for each  $T$  in TABLES( $\Sigma$ )
  if RTA is well-formed then
    RM-table-decl-set := ()
    for each table  $T$  in TABLES( $\Sigma$ )
      table-decl := "("
      if PKC( $T$ ) == (disc, f) then
        table-decl := Concat(table-decl, disc, " integer, ", f, " string, ")
        table-decl := Concat(table-decl, "primary key", "( ", disc, f, ")")
        add (disc, f) to the ATTRS( $T - C$ )
      else
        primarykey = "("
        for each path  $Pf_i$  in PKC( $T$ )
          primarykey := Concat(primarykey, GENNAME( $Pf_i$ ))
        table-decl := Concat(table-decl, "primary key", primarykey, ")")
      for each attribute  $A_i$  in ATTRS( $T$ )
        if  $A_i$  is not in ABS( $T$ ) then
          table-decl := Concat(table-decl,  $A_i$ , " ", "DOM( $T, A_i$ )")
          add  $A_i$  to the ATTRS( $T - C$ )
        if  $A_i$  is in ABS( $T$ ) and RAN( $T, A_i$ ) =  $T_j$  then
          if PKC( $T_j$ ) == (disc, f) then
            table-decl := Concat(table-decl, (Concat( $A_i$ , "-", disc, "integer"),
              Concat( $A_i$ , "-", f, "string")))
            table-decl := Concat(table-decl, "foreign key", "( ",  $A_i$ , "-",
              disc, " ",  $A_i$ , "-", f, ") ", " ",
              "references",  $T_j$ )
            add  $A_i$ .disc and  $A_i$ .f to the ATTRS( $T - C$ )
          else
            addnew-decl := ADDNEWATTR( $A_i, T, T_j$ )
            table-decl := Concat(table-decl, addnew-decl)
             $\triangleright$  primary key of  $T_j$  is not (disc, f)
      for each path  $Pf_i$  in PKC( $T$ )
        if  $Pf_i$  is not in ATTRS( $T - C$ ) then
          table-decl := Concat(table-decl,  $Pf_i$ , " ", DOM( $T, Pf_i$ ))
          add  $Pf_i$  to the ATTRS( $T - C$ )
      table-decl := Concat(table-decl, ")")
      add table-decl to the RM-table-decl-set
    return RM-table-decl-set
```

Algorithm 7

```
function ADDNEWATTR ( $A_i, T, T_j$ )
  table-decl = "("
  foreignkey = "("
  for each path  $Pf_j$  in PKC( $T_j$ )
    pathName := GENNAME( $Pf_j$ )
    orgT, PfDom := PATHDOM( $T_j, Pf_j$ )
    foreignkey := Concat(foreignkey,  $A_i$ , "-", pathName)
    table-decl := Concat(table-decl,  $A_i$ , "-", pathName, " ", PfDom )
    add  $A_i.Pf_j$  to the ATTRS(CONC( $T$ ))
  table-decl := Concat(table-decl, "foreign key, foreignkey, ")
  return Concat(table-decl, "references",  $T_j$ )
```

Algorithm 8

```
function GENNAME (Pf)
  for each “.” in Pf, where Pf is of the form  $B_1 \dots B_k$ 
    convert “.” to “-”
  return  $B_1 - \dots - B_k$ 
```

Algorithm 9

```
function PATHDOM ( $T$ , Pf)
  table =  $T$ 
  pathDom = “ ”
  if Pf contains only one attribute  $A_j$  then
    pathDom = “DOM(table,  $A_j$ )”
    sourceTable = table
  else
    concrete =  $A_j$  ▷ storing the concrete attribute of the path( $A_j$ )
    Pfk =  $A_1 \dots A_{j-1}$  ▷ storing the abstract attributes of the path
    for each attribute  $A_k$  in Pfk
      table := RAN(table,  $A_k$ )
    pathDom := “DOM(table, concrete)”
    sourceTable := RAN( $T$ ,  $A_1$ )
  return sourceTable, pathDom
```

Considering UNIV in Figure 2.3, we illustrate how the algorithm ADDNEWATTR(course, CLASS, COURSE) proceeds when UNIV is mapped to its concrete version UNIV'. There exists an abstract attribute `course` in `CLASS` which references the table `COURSE`. The PKC(COURSE-C) is (department.deptcode, cnum). The ADDNEWATTR(course, CLASS, COURSE) calls the GENNAME(department.deptcode) which converts the “.” in `department.deptcode` to a “-”, and assigns a variable, called pathName to `department-deptcode`. Then the algorithm PATHDOM(COURSE, department.deptcode) is called to find the domain type of the `department.deptcode` which is `integer`. In order to convert `course` to concrete attributes, a new string, called foreignkey is introduced which stores `course-department-deptcode`, as indicated below:

$$\text{foreignkey} := \text{Concat}(\text{foreignkey}, \text{“course”}, \text{“-”}, \text{department-deptcode}).$$

Another string, called table-decl, is also generated to store the `course-department-deptcode`, along with its domain type `integer`:

$$\text{table-decl} := \text{Concat}(\text{table-decl}, \text{“course”}, \text{“-”}, \text{department-deptcode}, \text{integer}).$$

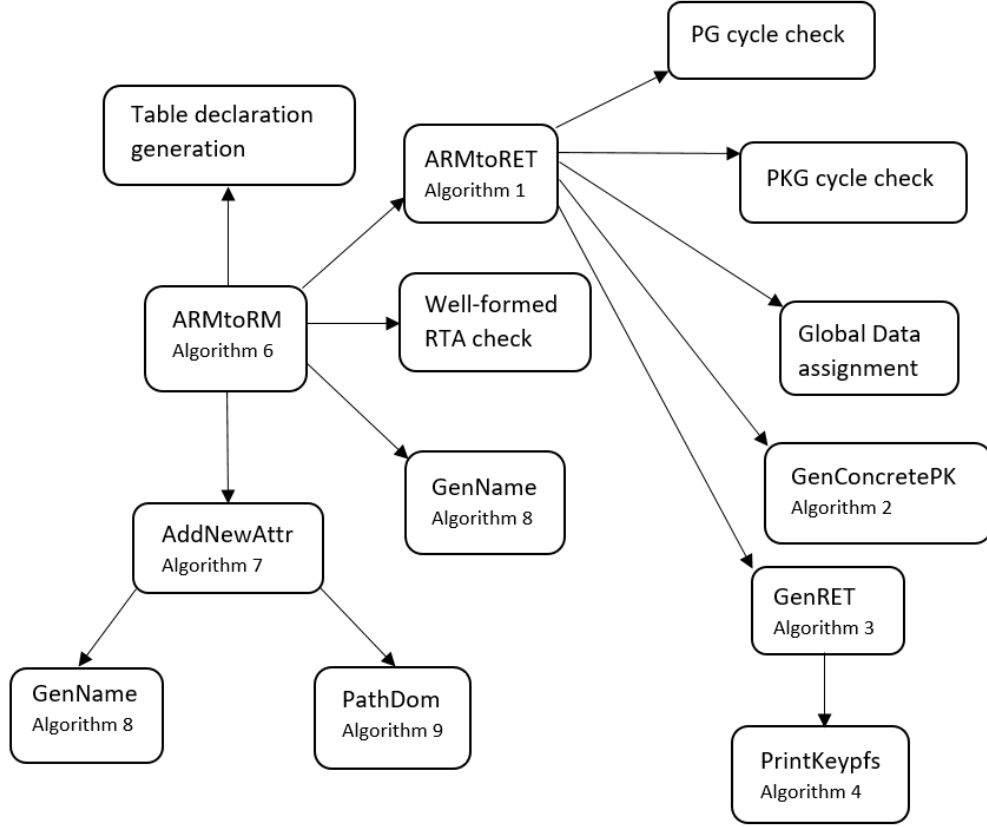


Figure 2.14: A call-graph for algorithms and definitions for mapping an ARM schema to an RM schema.

Then the new attribute `course.department.deptcode` is added to the `ATTRS(CLASS-C)`. The same process is iterated for the second attribute of `PKC(COURSE)` which is `cnum`. The `GENNAME(cnum)` returns `cnum`, since `cnum` is not a path and the `PATHDOM(COURSE, cnum)` returns the domain type of `cnum` which is `integer`. Then the string foreignkey and table-decl are updated as follows:

`foreignkey := Concat(foreignkey, "course", "-", cnum),`
`table-decl := Concat(table-decl, "course", "-", cnum, integer).`

Then the new attribute `course.cnum` is also added to the `ATTRS(CLASS-C)`. Thereafter, a

foreign key clause, using the foreignkey string is generated and added to the table-decl, as indicated:

```
table-decl := Concat(table-decl, "foreign key", foreign key, "references", COURSE).
```

Lastly, the `ADDNEWATTR(course, CLASS, COURSE)` returns the table-decl which stores the following string:

```
table-decl = "course-department-deptcode integer", "course-cnum integer",
             "foreign key(course-department-deptcode, course-cnum)
             references COURSE".
```

The next subsection illustrates converting UNIV in Figure 2.3 to UNIV' via procedure `ARMtoRM(UNIV)` and also indicates converting an SQLA query over UNIV to an SQL one over UNIV'.

2.4.1 ARMtoRM on UNIV

Considering UNIV, the following SQLP query illustrates the “names of professors who are also students”:

```
select pr.name
from PROFESSOR pr, STUDENT s
where pr.self = s.self
```

(2.11)

Attribute `name` in `PROFESSOR` is inherited from `PERSON`, since `PROFESSOR` is a subset of `PERSON`. Therefore, the SQLP query in (2.11) is simplified to the following SQLA:

```
select pe.name
from PROFESSOR pr, PERSON pe
where pr.self = pe.self
and exists( select * from STUDENT s
            where pe.self = s.self)
```

(2.12)

To map the SQLA query in (2.12) over UNIV to an SQL query over a concrete schema UNIV', first we need to map UNIV to UNIV' via procedure ARMToRM(UNIV).

ARMToRM(UNIV) first checks if RTA for UNIV is well-formed which is true, as explained in section 2.3. Then, for each table T in TABLES(UNIV), an empty string table-decl is generated to store the table declaration for T . We explain steps of this procedure for PROFESSOR and show the table declaration for PROFESSOR-C, STUDENT-C and PERSON-C in this subsection and Appendix A.3 illustrates the table declarations for all tables in UNIV'.

Since PKC(PROFESSOR) is (name, office), a primary key clause is added to the table-decl for PROFESSOR-C, as follows:

$$\text{table-decl} := \text{Concat}(\text{table-decl}, \text{"primary key"}, \text{"("}, \text{name}, \text{"}, \text{office}, \text{"})"})$$

ARMToRM(Σ) then iterates through ATTRS(PROFESSOR) which are office and department*. Since office is a concrete attribute, it is added to table-decl along with its data type, as shown here:

$$\text{table-decl} := \text{Concat}(\text{table-decl}, \text{office "integer"})$$

However, since department is an abstract attribute and it references DEPARTMENT and PKC(DEPARTMENT) = deptcode, function ADDNEWATTR(department, PROFESSOR, DEPARTMENT) is called and returns a new string named addnew-decl. This function generates the new attribute department-deptcode and adds it to ATTRS(PROFESSOR-C). department-deptcode, along with its data type which is integer is stored in addnew-decl, as shown below:

$$\text{addnew-decl} := \text{Concat}(\text{addnew-decl}, \text{department-deptcode "integer"})$$

A foreign key clause, including department-deptcode is generated as follows and also stored in addnew-decl :

$$\begin{aligned} \text{addnew-decl} := & \text{Concat}(\text{addnew-decl}, \text{"foreign key("}, \text{department-deptcode}, \text{"}), \\ & \text{"references"}, \text{DEPARTMENT}) \end{aligned}$$

ADDNEWATTR(department, PROFESSOR, DEPARTMENT) then returns addnew-decl which is concatenated with table-decl. Afterwards, ARMtoRM(Σ) iterates through each attribute in PKC(PROFESSOR) = (name, office), and since the attribute **name** has not existed in ATTRS(PROFESSOR-C)), it is added to table-decl as shown here:

$$\text{table-decl} := \text{Concat}(\text{table-decl}, \text{name "string"})$$

Therefore, the table declaration for concrete PROFESSOR-C is as follows:

```
table PROFESSOR-C (name string, office integer, department-deptcode integer,
                  primary key (name, office),
                  foreign key (department-deptcode) references DEPARTMENT)
```

ARMtoRM(UNIV) then generates the table declaration for STUDENT-C and PERSON-C for which PKC is (disc, f). Table declarations for these two tables are shown below:

```
table STUDENT-C (disc integer, f string, snum integer, year integer
                primary key (disc, f))

table PERSON-C (disc integer, f string, sin integer, name string, cellphone
               integer, primary key (disc, f))
```

A concrete relational schema UNIV' is indicated in Figure 2.15 in which a foreign key department-deptcode from PROFESSOR referencing DEPARTMENT is indicated by an arrow. The only tables whose PKC is (disc, f) are STUDENT and PERSON.

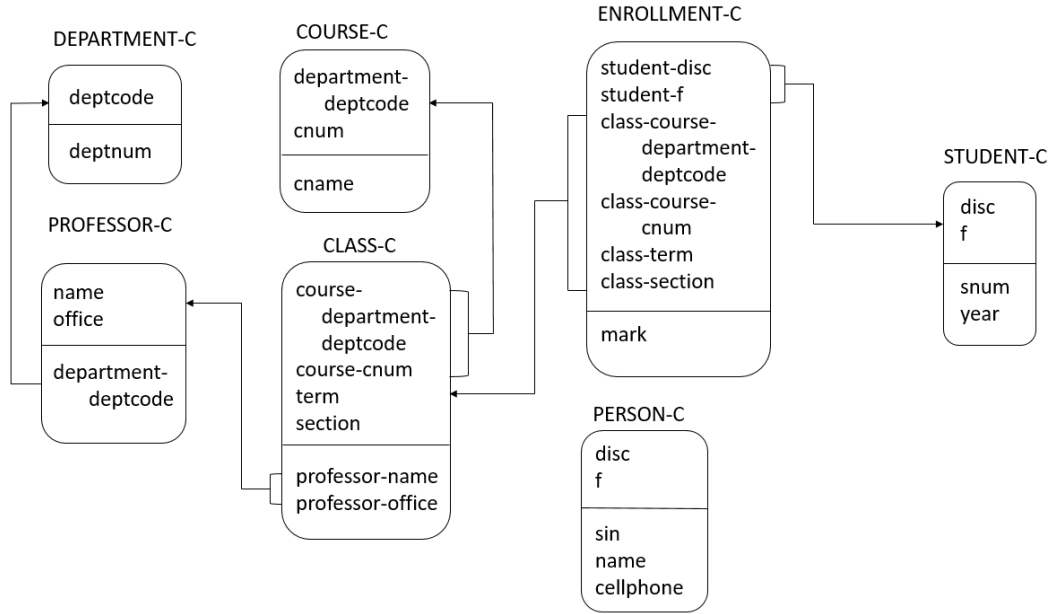


Figure 2.15: The concrete relational schema UNIV'

Considering the way Borgida et al. suggest in [1] for converting an SQLA query over UNIV to a SQL over UNIV', the `disc` attribute of `PERSON` is compared with `OFFSET(PROFESSOR) = 5` and the `f` attribute of `PERSON` is compared with `PKC(PROFESSOR) = (name, office)`. Also, the `disc` attribute of `STUDENT` is compared with `OFFSET(PROFESSOR) = 5` and the `f` attribute of `STUDENT` is compared with `PKC(PROFESSOR) = (name, office)`, as illustrated by the following SQL query:

```

select pe.name
from PROFESSOR pr
where exists( select * from PERSON pe
              where pe.disc = 5
              and pe.f = Concat( pr.name, pr.office)
              and where exists( select * from STUDENT s
                                where s.disc = 5
                                and s.f = Concat( pr.name, pr.office)))

```

(2.13)

Therefore, the SQLA query in (2.12) is converted to the SQL query in (2.13).

Chapter 3

Adding Translation Tables

This chapter introduces the new concept of *translation tables* to resolve reference issues in relational databases. At its core, a translation table from table $T_i\text{-}\mathbf{C}$ to table $T_j\text{-}\mathbf{C}$, hereon named $T_i\text{-}T_j\text{-}\mathbf{C}$, consists of columns encoding the ultimate primary keys of tables $T_i\text{-}\mathbf{C}$ and $T_j\text{-}\mathbf{C}$, and will contain a tuple for every `eid` occurring in common in the self columns of T_i and T_j . Recall that the use of “ $\text{-}\mathbf{C}$ ” for a table $T\text{-}\mathbf{C}$ implies that it exists in a concrete schema.

When necessary preference tables do not exist in an RM schema, translation tables should be generated to solve the equality issues and make the schema identity resolving. There are two approaches with translation tables. First, a translation table $T_i\text{-}T_j$ is introduced between tables T_i and T_j in the ARM. Second, a translation table $T_i\text{-}T_j\text{-}\mathbf{C}$ is introduced between tables $T_i\text{-}\mathbf{C}$ and $T_j\text{-}\mathbf{C}$ in the RM. This thesis follows the second approach.

There are two sections in this Chapter:

1. Section 3.1 introduces translation tables and illustrates how they affect mapping an ARM schema Σ to a RM schema Σ' , via the new ARMtoRM which has been updated from the ARMtoRM in Chapter 2 (Algorithm 6). The concept of identity resolution is also redefined in this section by considering translation tables. Then,

three examples are given to illustrate mapping Σ to Σ' , via the generation of translation tables when necessary.

2. Section 3.2 illustrates how to map SQLA queries over Σ to SQL ones over Σ' , considering translation tables, and via the new procedure SQLATOSQL. The three examples in 3.1 are recalled here to illustrate how different SQLA queries on Σ are mapped to SQL ones over Σ' in different circumstances.

3.1 ARM-to-RM mapping, considering translation tables

This section introduces the concept of *translation tables* which is the main contribution of this thesis. First, we explain how a translation table T_i-T_j-C is defined between two tables T_i-C and T_j-C , and the global data that is generated for ARM-to-RM mapping with respect to translation tables. Section 3.1.1 recalls the concept of identity resolution, now considering translation tables. In this section, we propose an updated version of procedure $\text{PRUNE}(\text{PRO}(\text{RTA}), T)$ which we call $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T)$, which will now have the option of introducing translation tables. This capability leads to a much larger variety of database schemata that are identity resolving. Also recall that PRO refers to a preference order on the tables comprising an ARM schema, and $\text{PRO}(\text{RTA})$ refers to a “global” referring expression type. (See Figure 2.9 and recall Section 2.3.)

New procedures and functions, that now use translation tables are introduced in Section 3.2. For example, function $\text{ABSORB}(T_i-T_j-C)$ checks whether T_i contains the constraint “isa T_j ” or whether T_j contains the constraint “isa T_i ”. Function $\text{REPLACE}(T_i-T_j-C)$ checks whether the translation table T_i-T_j-C can be computed from the join of two other translation tables. Furthermore, we propose procedure $\text{COMPRESSTTABLES}(\Sigma)$ which uses the information from $\text{ABSORB}(T_i-T_j-C)$ and $\text{REPLACE}(T_i-T_j-C)$ to reduce the number of unnecessary translation tables that can be computed via inheritance or the join of other translation tables respectively. Then, function $\text{UPDATEDARMTORM}(\Sigma)$ indicates

how function $\text{ARMToRM}(\Sigma)$ has been updated to map Σ to Σ' that can now contain translation tables. Finally, Section 3.3 includes three examples. The first one illustrates a scenario in which only translation tables are generated, whereas the second and third examples indicate different cases in which a combination of both preference and translation tables exists.

Definition 12 (Translation Table (TT))

A translation table T_i-T_j-C is a table in RM that is generated between tables T_i-C and T_j-C , where $\text{OFFSET}(T_i) < \text{OFFSET}(T_j)$ in PRO. T_i-T_j-C stores $\text{PKC}(T_i-C)$ and $\text{PKC}(T_j-C)$ for every common entity between T_i-C and T_j-C , such that $\text{PKC}(T_i-T_j-C)$ is always considered to be the same as $\text{PKC}(T_i)$. Also, the column(s) in the T_i-T_j-C which store(s) the sequence of attribute(s) in $\text{PKC}(T_i-C)$ is a foreign key, referencing T_i-C and the column(s), containing the sequence of attribute(s) in $\text{PKC}(T_j-C)$ is a foreign key, referencing T_j-C .

We have introduced global variables $\text{TT}(T)$ and $\text{TTSET}(\Sigma)$ in Figure 2.9 of Chapter 2, that are related to translation tables. More specifically, a $\text{TT}(T-C)$ is defined as a set of offsets $k < \text{OFFSET}(T)$ for which there exists a table T' for which $\text{OFFSET}(T') = k$, and for which there is a translation table $T'-T-C$. A $\text{TTSET}(\Sigma)$ is defined as a set of all translation tables in Σ . We also define $\text{CONCRAN}(T_i-C, \text{Pf}_j)$ to denote the table T_j-C which the foreign key Pf_j in the table T_i-C is referencing.

3.1.1 Identity Resolution with Translation Tables

Definition 13 (Well-formed RTA)

Let Σ be an ARM schema and RTA a referring type assignment for Σ . Given a preference order $\text{PRO} := (T_{i_1}, \dots, T_{i_k})$ on the set $\text{TABLES}(\Sigma)$, define $\text{PRO}(\text{RTA})$ as the following referring expression type:

$$\text{RTA}(T_{i_1}); \dots; \text{RTA}(T_{i_k})$$

We say that RTA is *well formed* if there is some preference order PRO such that the following conditions hold for each $T \in \text{TABLES}(\Sigma)$.

- $\text{RTA}(T) = \text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T)$,
- $\Sigma \models ((\text{cover by } \{T_1, \dots, T_n\}) \in T)$, where $\{T_1, \dots, T_n\}$ are all tables occurring in the guards in the $\text{RTA}(T)$, and
- for each component $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ of $\text{RTA}(T)$, the following also holds: (i) $\text{Pf}_{j,i}$ is well defined for T_j , for $1 \leq i \leq k_j$, and (ensuring *strong identification*) (ii) $\Sigma \models ((\text{path functional dependency } (\text{Pf}_{j,1}, \dots, \text{Pf}_{j,k_j}) \text{ determines self}) \in T_j)$.

□

The first condition of a well-formed RTA contains $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T)$ which is indicated in Algorithm 10. It works the same as $\text{PRUNE}(\text{PRO}(\text{RTA}), T)$ (Algorithm 5), with the difference that if the j th component “ $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ ” is not removed from $\text{PRO}(\text{RTA})$, and it is not a component of $\text{RTA}(T)$, $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T)$ removes it and generates a translation table $(T_j - T - C)$. The second and third conditions of a well-formed RTA are the same as the ones in Definition 10.

Algorithm 10

```

function UPDATEDPRUNE (PRO(RTA), T)
  pruneResult := PRO(RTA)
  if ( $\Sigma = (\text{cover by } \{T_1, \dots, T_{j-1}\}) \in T_j$ ) or
    ( $\Sigma = (\text{cover by } \{T_1, \dots, T_{j-1}\}) \in T$ ) or
    ( $\Sigma = (\text{disjoint with } T_j) \in T$ ) then
    remove the  $j$ th component “ $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ ” from pruneResult
  if “ $T_j \rightarrow (\text{Pf}_{j,1} = ?, \dots, \text{Pf}_{j,k_j} = ?)$ ” is not removed and it is not a component of
    RTA(T) then
    remove the  $j$ th component from pruneResult
    add the OFFSET( $T$ ) to the TT( $T_j$ )
    add the OFFSET( $T_j$ ) to the TT( $T$ )
    add the translation table  $T_j - T - C$  to the set TTSET( $\Sigma$ )
  return pruneResult

```

Definition of identity resolution, regarding translation tables remains the same as the one in Chapter 2 (Definition 10). Suppose M is an identity resolving mapping in a given database DB and an ARM schema Σ such that for every eid in the DB , the following is true:

$$\text{EVAL}(\text{M}(\text{eid}), DB) = (@\text{eid}).$$

Now if an $@\text{eid}$ exists in both T_i and T_j , where there exists a translation table $T_i-T_j-\mathbf{C}$, $\text{M}(\text{eid})$ is assigned to be the Re in $\text{RE}(\text{RTA}(T_i))$, as well as the Re in $\text{RE}(\text{RTA}(T_j))$, since $\text{OFFSET}(T_i) < \text{OFFSET}(T_j)$ in PRO .

3.1.2 On Reducing the Number of Translation Tables

As we have observed in $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T_j)$, if a table T_j is not disjoint with a table T_i that comes before T_j in the PRO , and there does not exist any preference table between T_i and T_j , then a translation table $T_i-T_j-\mathbf{C}$ is introduced between them to store common entities between $T_i-\mathbf{C}$ and $T_j-\mathbf{C}$. However, $T_i-T_j-\mathbf{C}$ can be removed if either of the following two conditions is satisfied:

1. If “ T_i *isa* T_j ” or “ T_j *isa* T_i ”
2. If $T_i-T_j-\mathbf{C}$ can be composed of joining two other translation tables TT_1 and TT_2

We introduce a procedure $\text{ABSORB}(T_i-T_j-\mathbf{C})$ in Algorithm 11 to check the first condition and a procedure $\text{REPLACE}(T_i-T_j-\mathbf{C})$ in Algorithm 12 to check the second one. The procedure $\text{ABSORB}(T_i-T_j-\mathbf{C})$ inputs the translation table $T_i-T_j-\mathbf{C}$ and returns “true on the right”, if the table declaration of T_i contains the clause “*isa* T_j ”. Since every entity in T_i is an entity in T_j , we say that $T_i-T_j-\mathbf{C}$ is absorbed on the right. Now if the table declaration of T_j contains the clause “*isa* T_i ”, the procedure $\text{ABSORB}(T_i-T_j-\mathbf{C})$ return “true on the left”. This means that $T_i-T_j-\mathbf{C}$ is absorbed on the left, since every entity of T_j is now an entity of T_i .

The procedure $\text{REPLACE}(T_i-T_j-\mathbf{C})$ checks whether $T_i-T_j-\mathbf{C}$ can be computed from the join of two other translation tables TT_1 and TT_2 . Consider an ARM schema Σ with a preference order PRO . Suppose tables T_i , T_j and T_k in Σ such that T_i is positioned before T_j in PRO . We now consider three different positions for T_k in PRO with respect to T_i and T_j . We refer to these possibilities as *replace possibilities* in the remainder of thesis:

1. The first possible order can be the order (T_i, T_k, T_j) as part of PRO. Considering such order, there can exist translation tables T_i-T_k-C , T_k-T_j-C and T_i-T_j-C . We assume that either $\text{ABSORB}(T_i-T_k-C)$ is “true on the left” or $\text{ABSORB}(T_k-T_j-C)$ is “true on the right”. If the former is the case, then T_i-T_k-C is removed and absorbed in T_i . Therefore, T_i contains an additional attribute $\text{PKC}(T_k)$ and the right outer join of T_i with T_k-T_j-C results in T_i-T_j-C . The translation table T_i-T_j-C is then removed and replaced as follows:

$$\text{REPLACE}(T_i-T_j-C) = (T_i-T_k-C, T_k-T_j-C)$$

2. The second possible order is (T_k, T_i, T_j) , in which there can exist translation tables T_k-T_i-C , T_k-T_j-C and T_i-T_j-C . We assume that either $\text{ABSORB}(T_k-T_i-C)$ is “true on the right” or $\text{ABSORB}(T_k-T_j-C)$ is “true on the right”. If the latter is true, then T_k-T_j-C is removed and absorbed in T_j by adding the additional attribute $\text{PKC}(T_k)$ to T_j . Then the right outer join of T_j with T_k-T_i-C results in T_i-T_j-C . The translation table T_i-T_j-C is then removed and obtained as follows:

$$\text{REPLACE}(T_i-T_j-C) = (T_k-T_i-C, T_k-T_j-C)$$

3. The third possibility is the order (T_i, T_j, T_k) which can result in translation tables T_i-T_j-C , T_i-T_k-C and T_j-T_k-C . We assume that either $\text{ABSORB}(T_i-T_j-C)$ is “true on the left” or $\text{ABSORB}(T_j-T_k-C)$ is “true on the left”. If the former is true, then T_i-T_j-C is removed and absorbed in T_i . Then, T_i contains the additional column $\text{PKC}(T_k)$ and the right outer join of T_i with T_j-T_k-C results in T_i-T_j-C . The translation table T_i-T_j-C is, therefore, removed and obtained as follows:

$$\text{REPLACE}(T_i-T_j-C) = (T_i-T_j-C, T_j-T_k-C)$$

Procedure $\text{COMPRESSTABLES}(\Sigma)$ uses the information from procedures $\text{ABSORB}(T_i-T_j-C)$ and $\text{REPLACE}(T_i-T_j-C)$ to optimize the number of translation tables that have been originally generated by procedure $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T_j)$. More precisely, procedure $\text{COMPRESSTABLES}(\Sigma)$ removes any translation table TT from $\text{TTSET}(\Sigma)$ if TT

is “absorbed on the left” or “absorbed on the right”, or if it can be replaced by two other translation tables TT_1 and TT_2 . If a translation table T_i-T_j-C is “absorbed on the left”, T_i-T_j-C is removed from $TTSET$ and each attribute A_k in $PKC(T_j-C)$ is added to T_i-C .

Function $UPDATEDARMTORM(\Sigma)$, indicated in Algorithm 14 maps an ARM schema Σ to an RM schema Σ' , via generating translation tables in Σ' . This function returns the set RM-table-decl-set which contains a concrete table declaration for each table in Σ' . Similar to procedure $ARMTORM(\Sigma)$ in Algorithm 6, procedure $UPDATEDARMTORM(\Sigma)$ first calls $ARMTORET(\Sigma)$ to generate $RTA(T)$ for each T in $TABLES(\Sigma)$. $ARMTORET(\Sigma)$, considering translation tables, has not been modified from the one in Chapter 2 (Algorithm 1). After generating RETs, the procedure $UPDATEDARMTORM(\Sigma)$ checks if the RTA of Σ is well-formed and if it is, procedure $COMPRESSTABLES(\Sigma)$ is called to update the $TTSET(\Sigma)$. Then, a concrete table declaration for each table T in Σ is generated and stored as the string table-decl, the same way as in $ARMTORM(\Sigma)$. Every table-decl for each T in Σ is added to the RM-table-decl-set. In the last part, function $BUILDTTs(TTSET(\Sigma))$ is called to generate the table declarations for all existing translation tables. A table declaration for each translation table in Σ' is also added to the RM-table-decl-set.

$BUILDTTs(TTSET(\Sigma))$ generates a table declaration for each translation table TT in the $TTSET(\Sigma)$. Function $TABLEFT(T_i-T_j-C)$ returns the left table of the translation table T_i-T_j-C which is T_i-C . Function $TABRIGHT(T_i-T_j-C)$ returns the right table of the translation table T_i-T_j-C which is T_j-C . This algorithm replaces the translation table T_a-T_b-C with two other translation tables TT_1 and TT_2 , according to the three cases explained in *replace possibilities*. Given a translation table T_i-T_j-C , if T_i is the subset of T_j , the translation table is said to be “absorbed on the left”, since T_i is on the left side of T_i-T_j-C . If T_j is the subset of T_i , the translation table is said to be “absorbed on the right”, since T_j is on the right side of T_i-T_j-C .

Algorithm 11

```
function ABSORB ( $T_i-T_j-C$ )  
  if  $T_i$  isa  $T_j$  then:  
  
    return “true on the left”  
  if  $T_j$  isa  $T_i$  then:  
  
    return “true on the right”  
  else  
  
    return false
```

Algorithm 12

```
function REPLACE ( $T_a-T_b-C$ )  
  
  for every  $T_k \in \text{PRO}$ , such that  $\text{OFFSET}(T_a) < \text{OFFSET}(T_k) < \text{OFFSET}(T_b)$  ▷ checking Replace case 1  
    if both  $T_a-T_k-C$  and  $T_k-T_b-C$  exist in  $\text{TTSET}(\Sigma)$  then  
      if (ABSORB( $T_a-T_k-C$ ) returns “true on the left”) or (ABSORB( $T_k-T_b-C$ )  
        returns “true on the right”) then  
        return ( $T_a-T_k-C$ ,  $T_k-T_b-C$ )  
  
  for every  $T_k \in \text{PRO}$ , such that  $\text{OFFSET}(T_k) < \text{OFFSET}(T_a) < \text{OFFSET}(T_b)$  ▷ checking Replace case 2  
    if both  $T_k-T_a-C$  and  $T_k-T_b-C$  exist in  $\text{TTSET}(\Sigma)$  then  
      if (ABSORB( $T_k-T_a-C$ ) returns “true on the right”) or (ABSORB( $T_k-T_b-C$ )  
        returns “true on the right”) then  
        return ( $T_k-T_a-C$ ,  $T_k-T_b-C$ )  
  
  for every  $T_k \in \text{PRO}$ , such that  $\text{OFFSET}(T_a) < \text{OFFSET}(T_b) < \text{OFFSET}(T_k)$  ▷ checking Replace case 3  
    if both  $T_a-T_k-C$  and  $T_b-T_k-C$  exist in  $\text{TTSET}(\Sigma)$  then  
      if (ABSORB( $T_a-T_k-C$ ) returns “true on the left”) or (ABSORB( $T_b-T_k-C$ )  
        returns “true on the left”) then  
        return ( $T_a-T_k-C$ ,  $T_b-T_k-C$ )  
  
  return null
```

Algorithm 13

```
procedure COMPRESSTABLES ( $\Sigma$ )
  finalTTSet := ()
  for every translation table ( $T_i-T_j-C$ ) in TTSET( $\Sigma$ ) ▷ Absorbed on the left check
    if ABSORB( $T_i-T_j-C$ ) returns “true on the left” then
      if PKC( $T_j-C$ ) = (disc, f) then
        if PKC( $T_i-C$ ) = (disc, f) then
          NewAttr1 := Concat(TABlename( $T_j-C$ ), “-”, disc)
          NewAttr2 := Concat(TABlename( $T_j-C$ ), “-”, f)
          Add the attributes NewAttr1 and NewAttr2 to the ATTRS( $T_i-C$ )
          CONCRAN( $T_i-C$ , (NewAttr1, NewAttr2)) =  $T_j$ 
        else
          Add the attributes disc and f to the ATTRS( $T_i-C$ )
          CONCRAN( $T_i-C$ , (disc, f)) =  $T_j-C$ 
      else ▷ PKC( $T_j-C$ ) is not (disc, f)
        if PKC( $T_j-C$ ) does not appear as attribute(s) in ATTRS( $T_i$ ) then
          for each path  $Pf_j$  in PKC( $T_j-C$ )
            Add  $Pf_j$  to the ATTRS( $T_i-C$ )
            CONCRAN( $T_i-C$ ,  $Pf_j$ ) =  $T_j-C$  ▷ Absorbed on the right check
    if ABSORB( $T_i-T_j-C$ ) returns “true on the right” then
      if PKC( $T_i-C$ ) = (disc, f) then
        if PKC( $T_i-C$ ) = (disc, f) then
          NewAttr1 := Concat(TABlename( $T_i$ ), “-”, disc)
          NewAttr2 := Concat(TABlename( $T_i$ ), “-”, f)
          Add the attributes NewAttr1 and NewAttr2 to the ATTRS(CONC( $T_j$ ))
          CONCRAN( $T_j-C$ , (NewAttr1, NewAttr2)) =  $T_i-C$ 
        else
          Add the attributes disc and f to the ATTRS( $T_j-C$ )
          CONCRAN( $T_j-C$ , (disc, f)) =  $T_i-C$ 
      else ▷ PKC( $T_i$ ) is not (disc, f)
        if PKC( $T_i-C$ ) does not appear as attribute(s) in ATTRS( $T_j$ ) then
          for each path  $Pf_i$  in PKC( $T_i-C$ )
            Add  $Pf_i$  to the ATTRS( $T_j-C$ )
            CONCRAN( $T_j-C$ ,  $Pf_i$ ) =  $T_i-C$ 
    else
      finalTTSet := Concat(finalTTSet,  $T_i-T_j-C$ ) ▷ Replace check
  if REPLACE( $T_i-T_j-C$ ) returns ( $TT_1$ ,  $TT_2$ ) then
    if  $TT_1$  is not in finalTTSet then
      finalTTSet := Concat(finalTTSet,  $TT_1$ )
    if  $TT_2$  is not in finalTTSet then
      finalTTSet := Concat(finalTTSet,  $TT_2$ )
  else
    finalTTSet := Concat(finalTTSet,  $T_i-T_j-C$ )
  TTSET( $\Sigma$ ) := finalTTSet
```

Algorithm 14

```
function UPDATEDARMToRM ( $\Sigma$ )
  RM-table-decl-set := ()
  Call ARMToRET( $\Sigma$ ) to generate RTA( $T$ ) for each  $T$  in TABLES( $\Sigma$ )
  if RTA is well-formed then
    call COMPRESSTTABLES( $\Sigma$ ) to update TTSET( $\Sigma$ )
    for each table  $T$  in TABLES( $\Sigma$ )
      table-decl := "("
      if ATTRS( $T$ -C) is non-empty then
        for each path  $Pf_i$  in ATTRS( $T$ -C)
           $T_j = \text{CONCRAN}(T\text{-C}, Pf_i)$ 
          if  $Pf_i$  contains disc then
            table-decl := Concat(table-decl, GENNAME( $Pf_i$ ), " ", "integer")
          if  $Pf_i$  contains f then
            table-decl := Concat(table-decl, GENNAME( $Pf_i$ ), " ", "string")
          else
             $T_q, \text{pfDom} := \text{PATHDOM}(T_j, Pf_i)$ 
            table-decl := Concat(table-decl, GENNAME( $Pf_i$ ), " ",  $\text{pfDom}$ )
            table-decl := Concat(table-decl, "foreign key", "(",  $Pf_i$ , ")", "references",  $T_j$ )
      if PKC( $T$ ) == (disc, f) then
        table-decl := Concat(table-decl, disc, " ", "integer", " ", f, " ", "string", " ")
        table-decl := Concat(table-decl, "primary key", "(", " ", disc, f, " )")
        add (disc, f) to the ATTRS( $T$ -C)
      else
        primarykey = "("
        for each path  $Pf_i$  in PKC( $T$ )
          primarykey := Concat(primarykey, GENNAME( $Pf_i$ ))
        table-decl := Concat(table-decl, "primary key", primarykey, " )")
      for each attribute  $A_i$  in ATTRS( $T$ )
        if  $A_i$  is not in ABS( $T$ ) then
          table-decl := Concat(table-decl,  $A_i$ , " ", "DOM( $T$ ,  $A_i$ )")
          add  $A_i$  to the ATTRS( $T$ -C)
        if  $A_i$  is in ABS( $T$ ) and  $\text{RAN}(T, A_i) = T_j$  then
          if PKC( $T_j$ ) == (disc, f) then
            table-decl := Concat(table-decl, (Concat( $A_i$ , "-", disc, "integer"), Concat( $A_i$ , "-", f, "string")))
            table-decl := Concat(table-decl, "foreign key", "(",  $A_i$ , "-", disc, " ",  $A_i$ , "-", f, " )", " ", "references",  $T_j$ )
            add  $A_i$ .disc and  $A_i$ .f to the ATTRS( $T$ -C)
          else ▷ primary key of  $T_j$  is not (disc, f)
            table-decl := ADDNEWATTR( $A_i$ ,  $T_j$ , table-decl)
      for each path  $Pf_i$  in PKC( $T$ )
        if  $Pf_i$  is not in ATTRS( $T$ -C) then
          table-decl := Concat(table-decl, GENNAME( $Pf_i$ ), " ", DOM( $T$ ,  $Pf_i$ ))
          add  $Pf_i$  to the ATTRS( $T$ -C)
      table-decl := Concat(table-decl, " )")
      add table-decl to the RM-table-decl-set
  TT-table-decl-set := BUILDTTs(TTSET( $\Sigma$ ))
  add the table declaration for each translation table in the TT-table-decl-set to the RM-table-decl-set
  return RM-table-decl-set
```

Algorithm 15

```
function BUILDTTs (TTSET( $\Sigma$ ))
  TT-table-decl-set := ()
  for each  $TT$  in TTSET( $\Sigma$ )
    table-decl = "("
    if PKC(TABLEFT( $TT$ )) == (disc, f) then
      table-decl := Concat(table-decl, disc, " integer ", f, " string, ")
      table-decl := Concat(table-decl, "primary key", "(, disc, f, ")")
      table-decl := Concat(table-decl, "foreign key", "(, disc, f, ",
        "references", TABLEFT( $TT$ ))
    else
      primarykey := "("
      foreignkey := "("
      for each path  $Pf_i$  in PKC(TABLEFT( $TT$ ))
        pfName := GENNAME( $Pf_i$ )
        primary-key := Concat(primary-key, pfName)
         $T_q$ , pfDom := PATHDOM(TABLEFT( $TT$ ),  $Pf_i$ )
        table-decl := Concat(table-decl, pfName, " ", pfDom)
        foreignkey := Concat(foreignkey, pfName)
      table-decl := Concat(table-decl, "primary key", primarykey, ")")
      table-decl := Concat(table-decl, "foreign key ", foreignkey, ") ", references,  $T_q$ )
    if PKC(TABRIGHT( $TT$ )) == (disc, f) then
      table-decl := Concat(table-decl, disc, " integer ", f, " string, ")
      table-decl := Concat(table-decl, "foreign key", "(, disc, f, ",
        "references", TABRIGHT( $TT$ ))
    else
      foreignkey := "("
      for each path  $Pf_i$  in PKC(TABRIGHT( $TT$ ))
        pfName := GENNAME( $Pf_i$ )
         $T_q$ , pfDom := PATHDOM(TABRIGHT( $TT$ ),  $Pf_i$ )
        table-decl := Concat(table-decl, pfName, " ", pfDom)
        foreignkey := Concat(foreignkey, pfName)
      table-decl := Concat(table-decl, "foreign key ", foreignkey, ") ", references,  $T_q$ )
      table-decl := Concat(table-decl, ")")
      TT-table-decl-set := Concat(TT-table-decl-set, table-decl)
  return TT-table-decl-set
```

Algorithm 16

```
function TABLEFT ( $T_i$ - $T_j$ -C)
  return  $T_i$ -C
```

Algorithm 17

```
function TABRIGHT ( $T_i$ - $T_j$ -C)
  return  $T_j$ -C
```

The following three examples illustrate how the algorithms work. The first two use the ARM schema UNIV, illustrated in Figure 2.3 with different databases, and the third one uses a new ARM schema called UNIVPEOPLE and indicated in Figure 3.6. Example 1 reflects on the existence of translation tables only, whereas Example 2 and 3 explain a scenario in which both preference and translation tables exist. The ARM schema UNIVPEOPLE in Example 3 provides a more involved and complex situation than the other two.

Example 1.

This example illustrates a case in which only translation tables are generated and no preference tables exist, as illustrated in a relational schema in Figure 3.2. The ARM schema in this example is called SecondUNIV, and it is based on UNIV in Figure 2.3 with modified table declarations for PROFESSOR, STUDENT and PERSON, such that neither of these tables contains a preference clause and they all contain primary keys. Both STUDENT and PROFESSOR are also subsets of PERSON. Table declarations for tables DEPARTMENT, COURSE, CLASS and ENROLLMENT in SecondUNIV remain the same as the ones in UNIV. Table declarations of all tables in SecondUNIV are illustrated in Appendix A.4 and RETs are indicated in Appendix A.5.

Since there does not exist any preference clause in SecondUNIV, translation tables should be generated to keep the schema identity resolving. This example illustrates how mapping SecondUNIV to its RM version SecondUNIV' through the algorithm UPDATEDARMToRM(SecondUNIV) results in the generation of translation tables PROFESSOR-STUDENT-C, PROFESSOR-PERSON-C and STUDENT-PERSON-C. Function UPDATEDARMToRM(SECONDUNIV) first calls procedure ARMToRET(SecondUNIV) to generate RETs for TABLE(SecondUNIV). ARMToRET(SecondUNIV) first generates a preference graph PG for SecondUNIV which does not contain any edge, since there does not exist any preference clause. Therefore, there does not exist any preference cycle and a possible preference order can be as follows:

PRO = (COURSE, STUDENT, DEPARTMENT, PROFESSOR, ENROLLMENT, CLASS, PERSON)

```

PK(PROFESSOR) = (name, office)
PREF(PROFESSOR) = ()
OFFSET(PROFESSOR) = 5
TABLE(5) = PROFESSOR
DISJ(PROFESSOR) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

OFFSET(STUDENT) = 6
TABLE(6) = STUDENT
PK(STUDENT) = (snum)
PREF(STUDENT) = ()
DISJ(STUDENT) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

OFFSET(PERSON) = 7
TABLE(7) = PERSON
PK(PERSON) = (sin)
PREF(PERSON) = ()
DISJ(PERSON) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

```

Figure 3.1: Modified global data for PROFESSOR, STUDENT and PERSON.

ARMTORET(Σ) then generates a PKG for SecondUNIV which is similar to PKG of UNIV in Figure 2.11. As illustrated in Figure 2.11, there is no primary key cycle in PKG and the following primary key order can be used:

PKO = (PERSON, PROFESSOR, STUDENT, DEPARTMENT, COURSE, CLASS, ENROLLMENT)

ARMTORET(SecondUNIV) then iterates through every table T in PRO and finds values for $PK(T)$, $PREF(T)$, $OFFSET(T)$, $TABLE(OFFSET(T))$, and $DISJ(T)$ which remain the same as before for DEPARTMENT, COURSE, CLASS and ENROLLMENT, as indicated in Figure 2.12, and change for PROFESSOR, STUDENT and PERSON, as illustrated in Figure 3.1.

Thereafter, ARMTORET(SecondUNIV) iterates through every table T in PKO and calls procedure GENCONCRETEPK(T) to generate a $PKC(T)$ and $KEYPFS(T)$ for each T . For example, GENCONCRETEPK(PROFESSOR) generates $PKC(PROFESSOR)$ and $KEYPFS-(PROFESSOR)$, as illustrated here:

$$\begin{aligned}
&\text{PKC}(\text{PERSON}) = (\text{sin}) \\
&\text{KEYPFS}(\text{PERSON}) = (\text{sin}) \\
\\
&\text{PKC}(\text{PROFESSOR}) = (\text{name}, \text{office}) \\
&\text{KEYPFS}(\text{PROFESSOR}) = (\text{name}, \text{office}) \\
\\
&\text{PKC}(\text{STUDENT}) = (\text{snum}) \\
&\text{KEYPFS}(\text{STUDENT}) = (\text{snum})
\end{aligned} \tag{3.1}$$

The result of calling $\text{GENCONCRETEPK}(\text{STUDENT})$ and $\text{GENCONCRETEPK}(\text{PERSON})$ are also illustrated in (3.1). Since neither of tables in SecondUNIV contains a preference clause, PKC of neither of them is of the form (disc, f) . Thus, for every T in $\text{TABLES}(\text{SecondUNIV})$, $\text{PKC}(T)$ and $\text{KEYPFS}(T)$ are the same, as indicated in (3.1).

Then, $\text{ARMTORET}(\text{SecondUNIV})$ iterates through every T in PRO and calls procedure $\text{GENRET}(T)$ to generate the following $\text{RTA}(T)$ for PROFESSOR , STUDENT and PERSON :

$$\begin{aligned}
&\text{RTA}(\text{STUDENT}) := \text{STUDENT} \rightarrow \text{snum} = ?, \\
&\text{RTA}(\text{PROFESSOR}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?) \\
&\text{RTA}(\text{PERSON}) := \text{PERSON} \rightarrow (\text{sin} = ?)
\end{aligned} \tag{3.2}$$

Once RETs are generated by $\text{ARMTORET}(\text{SecondUNIV})$, $\text{UPDATEDARMTORM}(\text{SecondUNIV})$ checks whether or not the RTA for SecondUNIV is well-formed. As explained in Definition 13, one of the condition for being a well-formed RTA is satisfying the following:

$$\text{RTA}(T) := \text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T)$$

However, since PROFESSOR , STUDENT and PERSON are not disjoint and there does not exist any preference table between them, $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), \text{STUDENT})$ generates the translation table $\text{PROFESSOR-STUDENT-C}$ and $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), \text{PERSON})$

generates the translation tables **PROFESSOR-PERSON-C** and **STUDENT-PERSON-C** to make the RTA for SecondUNIV identity resolving. The other two conditions of being a well-formed RTA, explained in Definition 13, are also met for SecondUNIV, so the RTA is well-formed.

UPDATEDARMtORM(SecondUNIV) then calls procedure COMPRESSTABLES(SecondUNIV) to reduce the number of translation tables if possible. Since **PROFESSOR** and **STUDENT** are subsets of **PERSON**, the translation tables **PROFESSOR-PERSON-C** and **STUDENT-PERSON-C** are both “absorbed on the left” and can be removed from TTSET(SecondUNIV). Thus, PKC(**person**) which is **sin** is added to both ATTRS(**STUDENT-C**) and ATTRS(**PROFESSOR-C**), and both CONCRAN(**STUDENT-C**, **sin**) and CONCRAN(**PROFESSOR-C**, **sin**) are set to **PERSON**.

The translation table **PROFESSOR-STUDENT-C** can also be computed from the join of two translation tables **PROFESSOR-PERSON** and **STUDENT-PERSON** via case 3 of REPLACE in *replace possibilities*, as shown below:

$$\text{REPLACE}(\text{PROFESSOR-STUDENT-C}) = (\text{PROFESSOR-PERSON-C}, \text{STUDENT-PERSON-C})$$

Since **PROFESSOR-PERSON-C** and **STUDENT-PERSON-C** are “absorbed on left”, **PROFESSOR-STUDENT-C** can also be computed from the join of two translation tables **PROFESSOR-C** and **STUDENT-C** on the column **person**, as indicated here:

$$\text{REPLACE}(\text{PROFESSOR-STUDENT-C}) = (\text{PROFESSOR-C}, \text{STUDENT-C})$$

Then, for each table T in SecondUNIV, an empty string table-decl is generated to store the table declaration of T . Since the ATTRS(**PROFESSOR-C**) already contains the attribute **sin** by absorbing **PERSON**, the table-decl is updated as follows:

```
table-decl := Concat(table-decl, sin, “integer”)
table-decl := Concat(table-decl, “foreign key”, sin, “refernces”, PERSON-C)
```

Since PKC(**PROFESSOR**) is (**name**, **office**), the following primary key clause is added to the table-decl of **PROFESSOR-C**:

table-decl = Concat(table-decl, “primary key”, “(”, name, “”, office, “)”)

UPDATEDARMToRM(SecondUNIV) then iterates through ATTRS(PROFESSOR) which are **office** and **department**. Since **office** is a concrete attribute, it is added to the table-decl along with its data type, as shown here:

table-decl := Concat(table-decl, office “integer”)

However, **department** is an abstract attribute in PROFESSOR that references DEPARTMENT, and PKC(DEPARTMENT) is **deptcode**. Function ADDNEWATTR(**department**, PROFESSOR, DEPARTMENT) is called to generate a new concrete attribute **department-deptcode** and add it to ATTRS(PROFESSOR-C). The attribute **department-deptcode**, along with its data type which is **integer** is stored in a new string, called addnew-decl, as shown below:

addnew-decl := Concat(addnew-decl, department-deptcode “integer”)

Then a foreign key clause, including **department-deptcode** is generated as follows and also stored in the addnew-decl:

addnew-decl := Concat(addnew-decl, “foreign key(”, department-deptcode, “)”,
“references”, DEPARTMENT))

ADDNEWATTR(**department**, PROFESSOR, DEPARTMENT) then returns addnew-decl which is concatenated with the table-decl. Afterwards, UPDATEDARMToRM(SecondUNIV) iterates through each attribute in PKC(PROFESSOR) = (**name**, **office**), and since attribute **name** has not existed in the ATTRS(PROFESSOR-C), it is added to the table-decl, as shown here:

table-decl := Concat(table-decl, name, “string”)

Therefore, table declaration for PROFESSOR-C is as follows:

```
table PROFESSOR-C (name string, office integer, department-deptcode integer,
                  sin integer, primary key (name, office),
                  foreign key (department-deptcode) references DEPARTMENT,
                  foreign key (sin) references PERSON-C).
```

UPDATEDARMtORM(SecondUNIV) then generates the table declaration for STUDENT-C and PERSON-C as follows:

```
table STUDENT-C (snum integer, year integer, sin integer,
                primary key (snum),
                foreign key (sin) references PERSON-C).

table PERSON-C (sin integer, name string, cellphone integer,
                primary key (sin)).
```

The concrete relational schema SecondUNIV' is indicated in Figure 3.2 in which there are two arrows going out from PROFESSOR-C, indicating two foreign keys `department-deptcode` and `sin`, referencing the tables DEPARTMENT-C and PERSON-C respectively, and one arrow from STUDENT-C to PERSON-C, indicating the foreign key `sin`. Appendix A.6 illustrates table declaration for all tables in SecondUNIV'.

Example 2.

This example illustrates a case in which a combination of translation tables and preference tables exists in a relational schema, as illustrated in the relational schema in Figure 3.3. This example introduces an ARM schema, called ThirdUNIV which is based on UNIV in Figure 2.3 with some changes to the tables PROFESSOR, STUDENT and PERSON, such that only PERSON contains a preference clause and they all contain primary keys. Both STUDENT and PROFESSOR are subsets of PERSON. Table declarations for tables in ThirdUNIV are illustrated in Appendix A.8 and RETs are indicated in Appendix A.9.

Furthermore, we explain how function UPDATEDARMtORM(ThirdUNIV) maps ThirdUNIV to its RM version ThirdUNIV', indicated in Figure 3.3. Since PERSON has a preference clause containing PROFESSOR and STUDENT, the absorbed translation tables

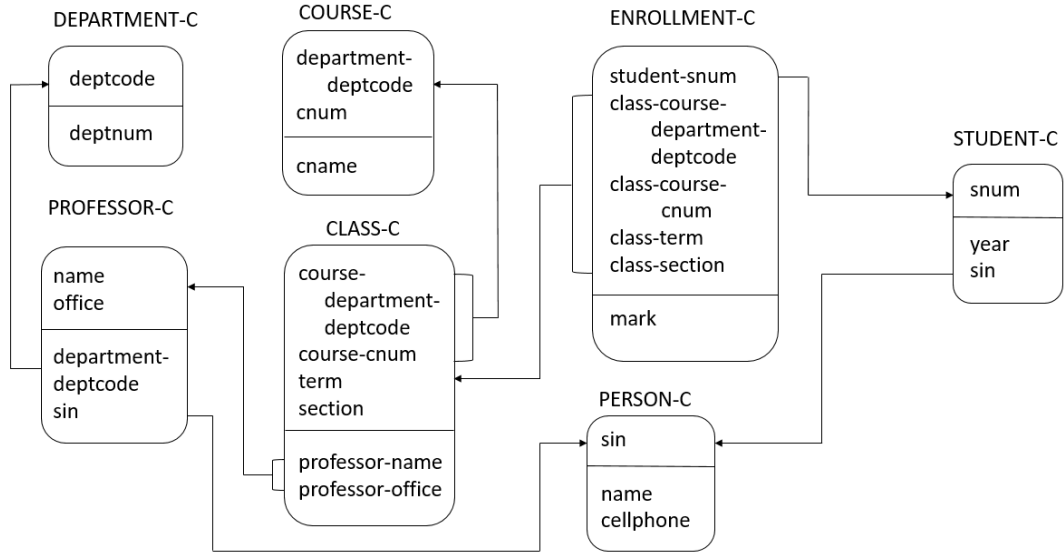


Figure 3.2: The concrete relational schema SecondUNIV'.

PROFESSOR-PERSON-C and STUDENT-PERSON-C that have been generated for SecondUNIV', no longer are generated for ThirdUNIV'. However, translation table PROFESSOR-STUDENT-C should be generated to find the professors who are also students to make ThirdUNIV' identity resolving.

UPDATEDARMtoRM(ThirdUNIV) first calls procedure ARMtoRET(ThirdUNIV) to generate RETs for ThirdUNIV. ARMtoRET(ThirdUNIV) first generates a preference graph PG for ThirdUNIV with two edges from PERSON to PROFESSOR and STUDENT, as illustrated in Figure 3.4. Since, there does not exist any preference cycle in PG, a possible preference order can be as follows:

PRO = (DEPARTMENT, COURSE, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON)

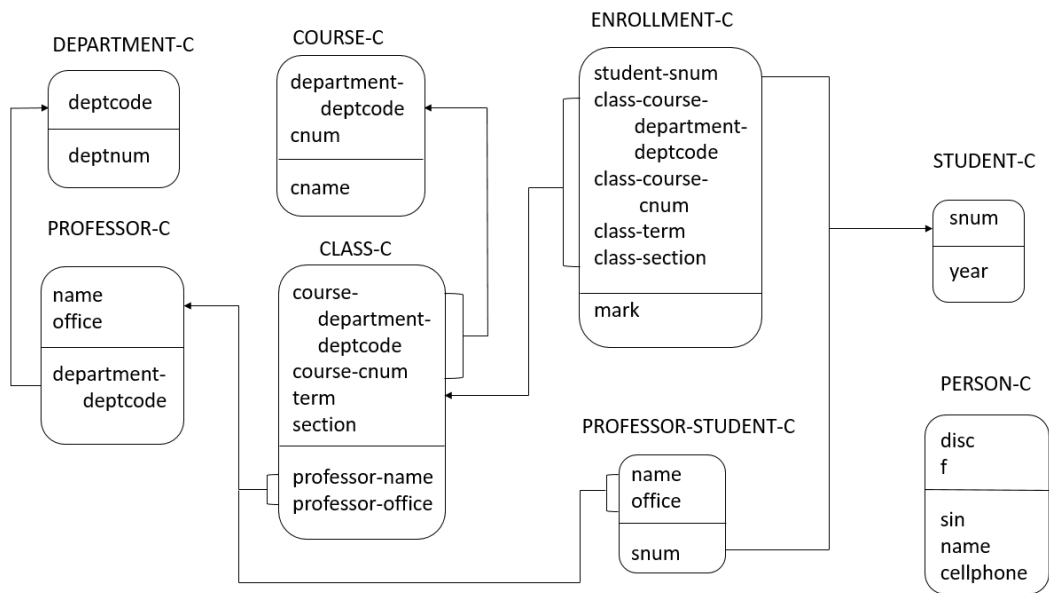


Figure 3.3: The relational schema ThirdUNIV'.

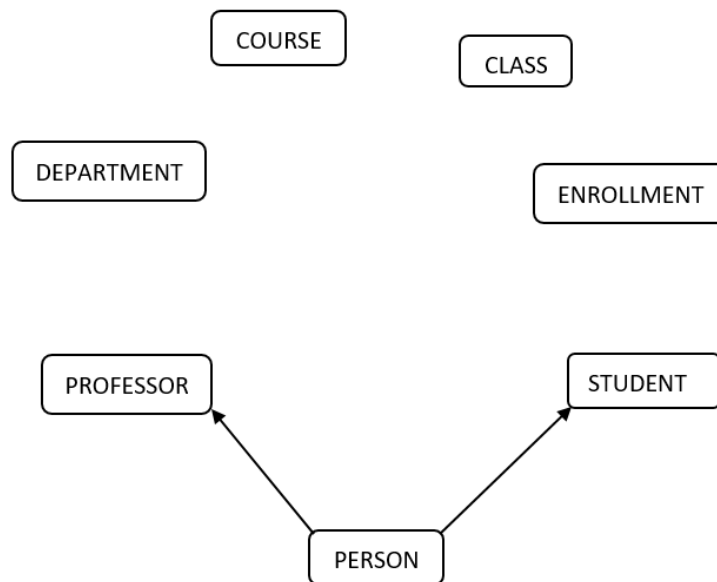


Figure 3.4: The preference graph for ThirdUNIV.


```

OFFSET(PROFESSOR) = 5
TABLE(5) = PROFESSOR
PK(PROFESSOR) = (name, office)
PREF(PROFESSOR) = ()
DISJ(PROFESSOR) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

OFFSET(STUDENT) = 6
TABLE(6) = STUDENT
PK(STUDENT) = (snum)
PREF(STUDENT) = ()
DISJ(STUDENT) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

OFFSET(PERSON) = 7
TABLE(7) = PERSON
PK(PERSON) = (sin)
PREF(PERSON) = (5, 6)
DISJ(PERSON) = (DEPARTMENT, COURSE, CLASS, ENROLLMENT)

```

Figure 3.5: Modified global data for PROFESSOR, STUDENT and PERSON.

Afterwards, $\text{ARMtoRET}(\text{ThirdUNIV})$ generates a primary key graph PKG for ThirdUNIV , which is the same as UNIV 's PKG in Figure 2.11, and it does not have any primary key cycle. Therefore, a possible primary key order can be as follows:

$\text{PKO} = (\text{PERSON}, \text{PROFESSOR}, \text{STUDENT}, \text{DEPARTMENT}, \text{COURSE}, \text{CLASS}, \text{ENROLLMENT})$

$\text{ARMtoRET}(\text{ThirdUNIV})$ then iterates through every table T in PRO and finds global data $\text{PK}(T)$, $\text{PREF}(T)$, $\text{OFFSET}(T)$, $\text{TABLE}(\text{OFFSET}(T))$, and $\text{DISJ}(T)$ which remain the same for DEPARTMENT , COURSE , CLASS and ENROLLMENT , as indicated in Figure 2.12, and are updated for PROFESSOR , STUDENT and PERSON , as shown in Figure 3.5.

Thereafter, $\text{ARMtoRET}(\text{ThirdUNIV})$ iterates through every table T in PKO and calls procedure $\text{GENCONCRETEPK}(T)$ to generate $\text{PKC}(T)$ and $\text{KEYPFS}(T)$ for each T . For example, $\text{GENCONCRETEPK}(\text{PERSON})$ generates $\text{PKC}(\text{PERSON})$ which is of the form (disc, f) and $\text{KEYPFS}(\text{PERSON})$ which is sin , as indicated here:

$$\begin{aligned}
& \text{PKC}(\text{PERSON}) = (\text{disc}, \text{f}) \\
& \text{KEYPFS}(\text{PERSON}) = (\text{sin}) \\
\\
& \text{PKC}(\text{PROFESSOR}) = (\text{name}, \text{office}) \\
& \text{KEYPFS}(\text{PROFESSOR}) = (\text{name}, \text{office}) \\
\\
& \text{PKC}(\text{STUDENT}) = (\text{snum}) \\
& \text{KEYPFS}(\text{STUDENT}) = (\text{snum})
\end{aligned} \tag{3.3}$$

The result of $\text{GENCONCRETEPK}(\text{PROFESSOR})$ and $\text{GENCONCRETEPK}(\text{STUDENT})$ are also illustrated in (3.3). Then, $\text{ARMTORET}(\text{ThirdUNIV})$ iterates through every T in PRO and calls procedure $\text{GENRET}(T)$ to generate the following $\text{RTA}(T)$ when T is PROFESSOR , STUDENT and PERSON :

$$\begin{aligned}
& \text{RTA}(\text{PROFESSOR}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?) \\
& \text{RTA}(\text{STUDENT}) := \text{STUDENT} \rightarrow \text{snum} = ? \\
& \text{RTA}(\text{PERSON}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?; \\
& \quad \text{PERSON} \rightarrow \text{sin} = ?
\end{aligned} \tag{3.4}$$

Once RETs have been generated by $\text{ARMTORET}(\text{ThirdUNIV})$, $\text{UPDATEDARMTORM}(\text{ThirdUNIV})$ checks whether or not the RTA for ThirdUNIV is well-formed. To satisfy the first condition of being a well-formed RTA in Definition 13, $\text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), \text{STUDENT})$ generates the translation table **PROFESSOR-STUDENT-C**, since there does not exist any preference table between **PROFESSOR** and **STUDENT**, and these two tables are not disjoint either. The other two conditions of a well-formed RTA in Definition 13 are also met for ThirdUNIV, so the RTA is well-formed.

$\text{UPDATEDARMTORM}(\text{ThirdUNIV})$ then calls $\text{COMPRESSTTABLES}(\text{ThirdUNIV})$ to optimize the number of translation tables in $\text{TTSets}(\Sigma)$ if possible. However, the transla-

tion table PROFESSOR-STUDENT-C is not absorbed and can not be computed from the join of other tables. Therefore, PROFESSOR-STUDENT-C remains in the TTSET(Σ). Thereafter, UPDATEDARMtORM(ThirdUNIV) iterates through each table T in TABLES(Σ) and generates the empty string table-decl to store the table declaration for T . Table declaration for PROFESSOR-C and STUDENT-C are as follows:

```
table PROFESSOR-C (name string, office integer, department-deptcode integer,
                    primary key (name, office),
                    foreign key (department-deptcode) references DEPARTMENT).
table STUDENT-C (snum integer, year integer,
                 primary key (snum)).
```

Since PKC(PERSON) is (disc, f), a primary key clause is added to table-decl for PERSON-C, as follows:

```
table-decl := Concat(table-decl, "primary key", "(", disc, ",", f, ")")
```

UPDATEDARMtORM(ThirdUNIV) then iterates through each attribute A_i in ATTRS(PERSON-C) which is the set (disc, f, sin, name, cellphone), and adds each concrete A_i , along with its data type to table-decl, as follows:

```
table-decl := Concat(table-decl, disc, " integer")
table-decl := Concat(table-decl, f, " string")
table-decl := Concat(table-decl, sin, " integer")
table-decl := Concat(table-decl, name, " string")
table-decl := Concat(table-decl, cellphone, " integer")
```

Thus, the final table-decl for PERSON-C is as indicated:

```
table PERSON-C (disc integer, f string, id integer, name string,
                cellphone integer, primary key (disc, f)).
```

UPDATEDARMToRM(ThirdUNIV) then calls function BUILDTTs(TTSET(ThirdUNIV)) to generate a table declaration for PROFESSOR-STUDENT-C. BUILDTTs(TTSET(ThirdUNIV)) creates an empty string table-decl for PROFESSOR-STUDENT-C which stores its table declaration. Since TABLEFT(PROFESSOR-STUDENT-C) is PROFESSOR and PKC(PROFESSOR) is (name, office), table-decl is updated as follows:

```
table-decl := Concat(table-decl, name, " string")
table-decl := Concat(table-decl, office, " integer")
```

The primary key and one of the foreign keys of PROFESSOR-STUDENT-C is then generated as follows:

```
table-decl := Concat(table-decl, "primary key ", "(" , name, office, ")")
table-decl := Concat(table-decl, "foreign key", "( "name, office ")",
                    "references PROFESSOR")
```

Since TABRIGHT(PROFESSOR-STUDENT-C) is STUDENT and PKC(STUDENT) is snum, table-decl is modified as indicated:

```
table-decl := Concat(table-decl, snum, " integer")
```

The second foreign key of PROFESSOR-STUDENT-C is also generated as illustrated:

```
table-decl := Concat(table-decl, "foreign key", "( "snum ")", "references STUDENT")
```

The final table declaration for PROFESSOR-STUDENT-C is as follows:

```
table PROFESSOR-STUDENT-C (name string, office integer, snum integer,
    primary key (name, office),
    foreign key (name, office) references PROFESSOR-C,
    foreign key (snum) references STUDENT-C).
```

The concrete relational schema ThirdUNIV' is indicated in Figure 3.3 in which there exists only one translation table PROFESSOR-STUDENT-C with two foreign keys snum and (name, office), referencing STUDENT and PROFESSOR respectively.

Example 3.

This example introduces a new ARM schema called UNIVPEOPLE which is illustrated in Figure 3.6, to propose a more in-depth scenario in which both preference and translation tables exist. UNIVPEOPLE contains five tables **EMPLOYEE**, **PROFESSOR**, **STUDENT**, **VISITOR** and **CANADIAN**, where **PROFESSOR** is the subset of **EMPLOYEE** and the only table with a preference clause is **VISITOR** which contains a preference over **PROFESSOR** and **STUDENT** in terms. **VISITOR** is also disjoint with **CANADIAN**. Table declarations for TABLES(UNIVPEOPLE) and RETs are indicated in Appendix A.11 and A.12 respectively.

Similar to the previous two examples, we illustrate how function UPDATEDARM-TOARM(UNIVPEOPLE) is compiled to map UNIVPEOPLE to its RM version UNIVPEOPLE'. UPDATEDARMTOARM(UNIVPEOPLE) first calls procedure ARMTORET(UNIVPEOPLE) to generate RETs for TABLES(UNIVPEOPLE). ARMTORET(UNIVPEOPLE) first generates a preference graph PG for UNIVPEOPLE, in which there does not exist any preference cycle, as illustrated in Figure 3.7. Thus, the following can be a possible preference order:

$$\text{PRO} = (\text{EMPLOYEE}, \text{PROFESSOR}, \text{STUDENT}, \text{VISITOR}, \text{CANADIAN})$$

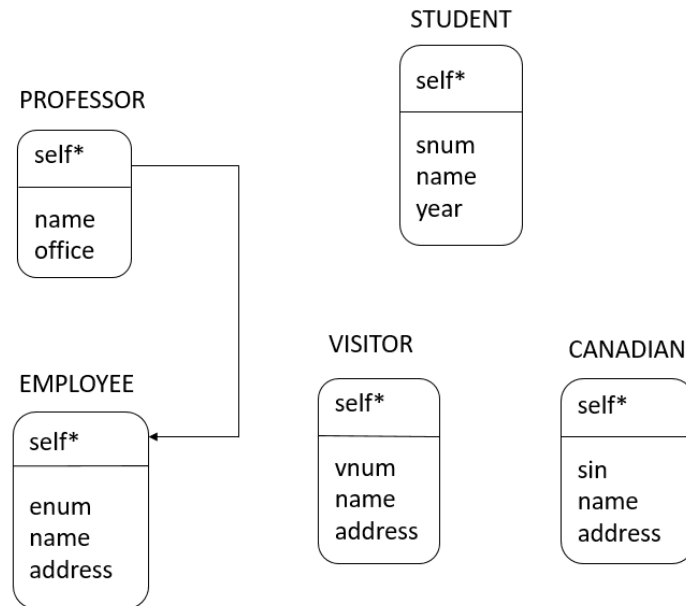


Figure 3.6: The ARM schema UNIVPEOPLE

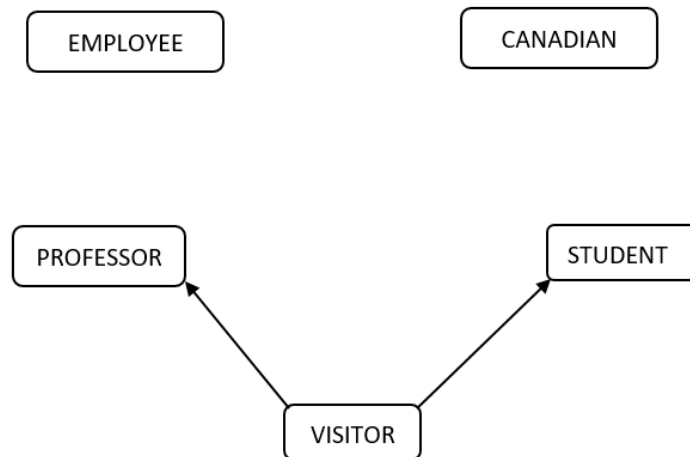


Figure 3.7: The preference graph for UNIVPEOPLE.

Afterwards, $\text{ARMtoRET}(\text{UNIVPEOPLE})$ generates a primary key graph PKG for UNIVPEOPLE , in which there does not exist any edge, since none of the tables in UNIVPEOPLE contains an abstract attribute in its primary key that references another table. Therefore, there does not exist any primary key cycle in PKG and any primary key order can be picked. One possible order can be as follows:

$$\text{PKO} = (\text{STUDENT}, \text{CANADIAN}, \text{PROFESSOR}, \text{VISITOR}, \text{EMPLOYEE})$$

$\text{ARMtoRET}(\text{UNIVPEOPLE})$ then iterates through each table T in PRO and finds $\text{OFFSET}(T)$, $\text{TABLE}(\text{OFFSET}(T))$, $\text{PK}(T)$, $\text{PREF}(T)$ and $\text{DISJ}(T)$ for EMPLOYEE , PROFESSOR , STUDENT , VISITOR and CANADIAN , as indicated in Figure 3.8. Thereafter, $\text{ARMtoRET}(\text{UNIVPEOPLE})$ iterates through each table T in PKO and calls the procedure $\text{GENCONCRETEPK}(T)$ to generate $\text{PKC}(T)$ and $\text{KEYPFS}(T)$ for each T , as shown here:

$$\text{PKC}(\text{EMPLOYEE}) = (\text{enum})$$

$$\text{KEYPFS}(\text{EMPLOYEE}) = (\text{enum})$$

$$\text{PKC}(\text{PROFESSOR}) = (\text{name}, \text{office})$$

$$\text{KEYPFS}(\text{PROFESSOR}) = (\text{name}, \text{office})$$

$$\text{PKC}(\text{STUDENT}) = (\text{snum})$$

$$\text{KEYPFS}(\text{STUDENT}) = (\text{snum})$$

$$\text{PKC}(\text{VISITOR}) = (\text{disc}, \text{f})$$

$$\text{KEYPFS}(\text{VISITOR}) = (\text{visanum})$$

$$\text{PKC}(\text{CANADIAN}) = (\text{sin})$$

$$\text{KEYPFS}(\text{CANADIAN}) = (\text{sin})$$

$\text{PKC}(T)$ and $\text{KEYPFS}(T)$ for all tables except VISITOR are the same, since neither of them has a preference clause. For VISITOR , however, $\text{PKC}(T)$ is the set (disc, f) and $\text{KEYPFS}(T)$

is `visanum`. Then, `ARMTORET(UNIVPEOPLE)` iterates through each table T in `PRO` and calls procedure `GENRET(T)` to generate `RTA(T)` for each T , as illustrated here:

```

RTA(EMPLOYEE)  := EMPLOYEE → enum = ?
RTA(PROFESSOR) := PROFESSOR → (name = ?, office = ?)
RTA(STUDENT)   := STUDENT → snum = ?
RTA(VISITOR)   := PROFESSOR → (name = ?, office = ?);
                STUDENT → snum = ?; VISITOR → visanum = ?
RTA(CANADIAN)  := CANADIAN → sin = ?

```

Once RETs have been generated by procedure `ARMTORET(UNIVPEOPLE)`, function `UPDATEDARMTORET(UNIVPEOPLE)` checks whether or not the RTA for `UNIVPEOPLE` is well-formed by checking the following condition for each T in `UNIVPEOPLE` and the other two conditions which are all explained in Definition 13.

$$\text{RTA}(T) := \text{UPDATEDPRUNE}(\text{PRO}(\text{RTA}), T)$$

Function `UPDATEDPRUNE(PRO(RTA), EMPLOYEE)` returns `RTA(EMPLOYEE)` which satisfies the above condition. However, `UPDATEDPRUNE(PRO(RTA), PROFESSOR)` does not return `RTA(PROFESSOR)`, since there does not exist any preference table between `EMPLOYEE` and `PROFESSOR`, even though they are not disjoint. Therefore, `UPDATEDPRUNE(PRO(RTA), PROFESSOR)` generates the translation table `EMPLOYEE-PROFESSOR-C` to find the intersection entities between `EMPLOYEE` and `PROFESSOR` and make `UNIVPEOPLE` identity resolving. There also does not exist any preference table between `EMPLOYEE` and `STUDENT` and they are not disjoint either, so `UPDATEDPRUNE(PRO(RTA), STUDENT)` then generates the translation tables `EMPLOYEE-STUDENT-C` and `PROFESSOR-STUDENT-C`. Next, `UPDATEDPRUNE(PRO(RTA), VISITOR)` only generates the translation table `EMPLOYEE-VISITOR-C`, since `VISITOR` includes a preference clause containing both `PROFESSOR` and `STUDENT`. Thereafter, `UPDATEDPRUNE(PRO(RTA), CANADIAN)` generates the translation tables `EMPLOYEE-CANADIAN-C`, `PROFESSOR-CANADIAN-C` and `STUDENT-CANADIAN-C`. All these translation tables are stored in the set `TTSET(UNIVPEOPLE)`, as indicated here:


```

OFFSET(EMPLOYEE) = 1
TABLE(1) = EMPLOYEE
PK(EMPLOYEE) = (enum)
PREF(EMPLOYEE) = ()
DISJ(EMPLOYEE) = ()

OFFSET(PROFESSOR) = 2
TABLE(2) = PROFESSOR
PK(PROFESSOR) = (name, office)
PREF(PROFESSOR) = ()
DISJ(PROFESSOR) = ()

OFFSET(STUDENT) = 3
TABLE(3) = STUDENT
PK(STUDENT) = (snum)
PREF(STUDENT) = ()
DISJ(STUDENT) = ()

OFFSET(VISITOR) = 4
TABLE(4) = VISITOR
PK(VISITOR) = (visanum)
PREF(VISITOR) = (2, 3)
DISJ(VISITOR) = (CANADIAN)

OFFSET(CANADIAN) = 5
TABLE(5) = CANADIAN
PK(CANADIAN) = (sin)
PREF(CANADIAN) = ()
DISJ(CANADIAN) = (VISITOR)

```

Figure 3.8: Global data for EMPLOYEE, PROFESSOR, STUDENT, VISITOR and CANADIAN.

```

TTSet(UNIVPEOPLE) = (EMPLOYEE-PROFESSOR-C, EMPLOYEE-STUDENT-C,
  PROFESSOR-STUDENT-C, EMPLOYEE-VISITOR-C, EMPLOYEE-CANADIAN-C,
  PROFESSOR-CANADIAN-C, STUDENT-CANADIAN-C)

```

RTA of UNIVPEOPLE also satisfies the condition 2 and 3 of being a well-formed RTA as explained in Definition 13. UPDATEDARMtoRM(UNIVPEOPLE) then calls procedure COMPRESSTABLES(UNIVPEOPLE) to update TTSet(UNIVPEOPLE) by removing unnecessary translation tables. For example, EMPLOYEE-PROFESSOR-C is “absorbed on the right”, since PROFESSOR is the subset of EMPLOYEE. Therefore, the attribute PKC(EMPLOYEE) which is `enum` is added to PROFESSOR-C, and EMPLOYEE-PROFESSOR-C is removed from TTSet(UNIVPEOPLE). Considering the second case of REPLACE in *replace possibilities*, translation tables PROFESSOR-STUDENT-C and PROFESSOR-CANADIAN-C, can be removed from TTSet(UNIVPEOPLE) and replaced as follows:

REPLACE(PROFESSOR-STUDENT-C) = (EMPLOYEE-PROFESSOR-C, EMPLOYEE-STUDENT-C)
 REPLACE(PROFESSOR-CANADIAN-C) = (EMPLOYEE-PROFESSOR-C, EMPLOYEE-CANADIAN-C)

Therefore, COMPRESSTABLES(UNIVPEOPLE) updates TTSet(UNIVPEOPLE), as indicated:

TTSet(UNIVPEOPLE) = (EMPLOYEE-STUDENT-C, EMPLOYEE-VISITOR-C,
 EMPLOYEE-CANADIAN-C, STUDENT-CANADIAN-C)

Afterwards UPDATEDARMtoRM(UNIVPEOPLE) iterates through each table T in UNIVPEOPLE and generates the empty string table-decl to store the table declaration for T . Thus, the table declaration for EMPLOYEE-C is as follows:

table EMPLOYEE-C (`enum integer`, `name string`, `address string`,
 `primary key (enum)`)

Since PKC(EMPLOYEE) which is `enum` has been already added to ATTRS(PROFESSOR-C) by absorbing EMPLOYEE, table-decl for PROFESSOR-C is updated as follows:

table-decl := Concat(table-decl, `enum`, “`integer`”)
 table-decl := Concat(table-decl, “foreign key”, `enum`, “references”, EMPLOYEE-C)

Since PKC(PROFESSOR) is (`name`, `office`), a primary key clause is added to the table-decl for PROFESSOR-C, as follows:

table-decl := Concat(table-decl, “primary key”, “(”, name, “”, office, “)”)

UPDATEDARMtoRM(UNIVPEOPLE) then iterates through ATTRS(PROFESSOR) which are **name** and **office**. Since they are both concrete attributes, they are added to the table-decl along with their data types, as shown here:

table-decl := Concat(table-decl, name “string”)
table-decl := Concat(table-decl, office “integer”)

Therefore, the table declaration for PROFESSOR-C is as follows:

```
table PROFESSOR-C (name string, office integer, enum integer,
                    primary key (name, office),
                    foreign key (enum) references EMPLOYEE-C).
```

The table declaration for STUDENT-C is computed similarly and illustrated below:

```
table STUDENT-C (snum integer, name string, year integer,
                 primary key (snum)).
```

Afterwards, UPDATEDARMtoRM(UNIVPEOPLE) generates the table-decl for VISITOR-C. Since PKC(VISITOR) is (disc, f), a primary key clause is added to the table-decl, as follows:

table-decl := Concat(table-decl, “primary key”, “(”, disc, “”, f, “)”)

UPDATEDARMtoRM(UNIVPEOPLE) then iterates through each attribute A_i in ATTRS(VISITOR) which is (vnum, name, address), and adds them to ATTRS(VISITOR-C). Each A_i , along with its data type is also added to table-decl, as follows:

table-decl := Concat(table-decl, vnum, “ integer”)
table-decl := Concat(table-decl, name, “ string”)
table-decl := Concat(table-decl, address, “ string”)

UPDATEDARMtoRM(UNIVPEOPLE) iterates through each attribute in PKC(VISITOR) which is (disc, f) and, since they do not already exist in ATTRS(VISITOR-C), adds them to ATTRS(VISITOR-C). Attributes (disc, f) are also added to table-decl, along with their data types, as indicated:

```

table-decl := Concat(table-decl, disc, " integer")
table-decl := Concat(table-decl, f, " string")

```

Thus, the final table-decl for VISITOR-C is:

```

table VISITOR-C (disc integer, f string, vnum integer, name string, address string,
primary key (disc disc, f)).

```

UPDATEDARMToRM(UNIVPEOPLE) generates a table-decl for CANADIAN-C similar to previous tables:

```

table CANADIAN-C (sin integer, name string, address string,
primary key (sin)).

```

UPDATEDARMToRM(UNIVPEOPLE) then calls function BUILDTTs(TTSet(UNIVPEOPLE)) to generate table declarations for translation tables. BUILDTTs(TTSet(UNIVPEOPLE)) iterates through each translation table *TT* in TTSet(UNIVPEOPLE) and generates an empty string table-decl for each *TT* to store *TT*'s table declaration in it. BUILDTTs(TTSet(UNIVPEOPLE)) first generates an empty string TT-decl for EMPLOYEE-STUDENT-C which stores its table declaration. Since the left table of EMPLOYEE-STUDENT-C is EMPLOYEE which is returned by function TABLEFT(EMPLOYEE-STUDENT-C), and PKC(EMPLOYEE) is enum, TT-decl is updated as follows:

```

TT-decl := Concat(TT-decl, enum, " integer")

```

Primary key and one of the foreign keys of EMPLOYEE-STUDENT-C is then generated as shown:

```

TT-decl := Concat(TT-decl, "primary key ", "( ", enum, " ")
TT-decl := Concat(TT-decl, "foreign key", "( "enum ", "references EMPLOYEE")

```

Since the right table of EMPLOYEE-STUDENT-C is STUDENT which is returned by function TABRIGHT(EMPLOYEE-STUDENT-C), and PKC(STUDENT) is snum, table-decl is modified as indicated:

$$\text{TTe-decl} := \text{Concat}(\text{TT-decl}, \text{snum}, " \text{ integer} ")$$

The second foreign key of **EMPLOYEE-STUDENT** is also generated as illustrated:

$$\text{TT-decl} := \text{Concat}(\text{TT-decl}, " \text{ foreign key} ", "(\text{ snum}) ", " \text{ references STUDENT} ")$$

Therefore, the final table declaration for **EMPLOYEE-STUDENT-C** is computed:

```
table EMPLOYEE-STUDENT-C (enum integer, snum integer,
                           primary key (enum),
                           foreign key (enum) references EMPLOYEE-C,
                           foreign key (snum) references STUDENT-C).
```

Table declaration for **EMPLOYEE-VISITOR-C**, **EMPLOYEE-CANADIAN-C** and **STUDENT-CANADIAN-C** are generated similarly and illustrated:

```
table EMPLOYEE-VISITOR-C (enum integer, disc integer, f string,
                           primary key (enum),
                           foreign key (enum) references EMPLOYEE-C,
                           foreign key (disc, f) references VISITOR-C).
```

```
table EMPLOYEE-CANADIAN-C (enum integer, sin integer,
                           primary key (enum),
                           foreign key (enum) references EMPLOYEE-C,
                           foreign key (sin) references CANADIAN-C).
```

```
table STUDENT-CANADIAN-C (snum integer, sin integer,
                           primary key (snum),
                           foreign key (snum) references STUDENT-C,
                           foreign key (sin) references CANADIAN-C).
```

Final table declarations for all tables are illustrated in Appendix [A.11](#), and **UNIVPEOPLE'** is presented in Figure [3.9](#).

3.2 SQLAtoSQL: Mapping an SQLA query to an SQL query

Recall from Figure 2.4b, the syntax for SQLA queries, that mapping such queries to equivalent formulations over our generated concrete RM schema requires rewriting subexpressions of the form “ $x_1.A_1 = x_2.A_2$ ”. For example, consider the following SQLA query:

```
select *  
from T1 x, T2 y  
where x.A = y.B,
```

where there exists tables T_i and T_j , such that $\text{RAN}(T_1, A) = T_i$ and $\text{RAN}(T_2, B) = T_j$. If A and B are **self** attributes, then T_1 and T_2 are the same as T_i and T_j respectively, and this SQLA query is changed to the following one:

```
select *  
from T1 x, T2 y  
where x.self = y.self
```

We have introduced procedure $\text{SQLAToSQL}(x, A, T_i, y, B, T_j)$ to compile $x.A(T_i) = y.B(T_j)$ by replacing it with an string which has the following format:

```
exists (select * from WHERE-TABS  
       where WHERE-EQS)
```

Variable WHERE-TABS in this query, stores the table names in the “from” clause, and variable WHERE-EQS, stores the string in the “where” clause.

Figure 3.10 illustrates a dependency graph of all procedures and functions that are executed to convert an SQLA query to an SQL query. Every node in this graph illustrates a procedure/function, and there exists an edge from node A to B if procedure/function A calls procedure/function B. The node for $\text{SQLAToSQL}(Q)$ in this graph is the only

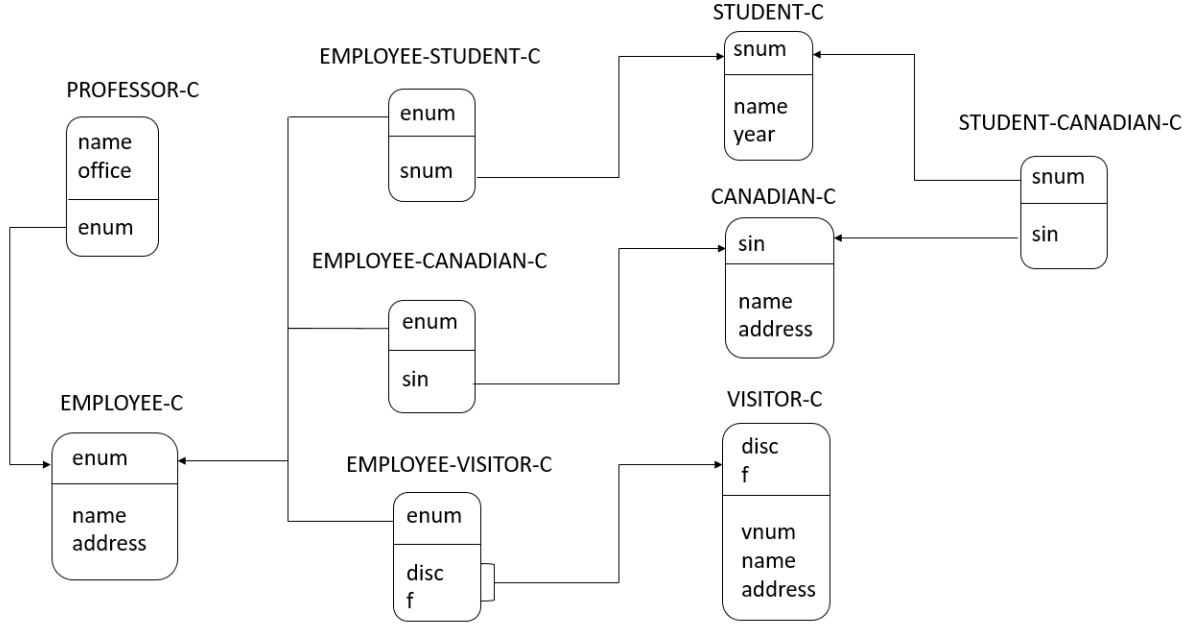


Figure 3.9: The relational schema UNIVPEOPLE'

node without incoming edges, as procedure $\text{SQLAToSQL}(Q)$ only calls other procedures/functions.

We present pseudo code for each procedure/function together with an overview that outlines its purpose and how it executes. A more in-depth explanation of how $\text{SQLAToSQL}(Q)$ works will then follow by appealing to a sequence of three progressively more involved examples.

Consider an SQLA query Q over an ARM schema Σ , which has the format in (3.2), procedure $\text{SQLAToSQL}(Q)$ which is introduced in Algorithm 18 maps Q to an SQL query Q' over the RM schema Σ' . $\text{SQLAToSQL}(Q)$ does that by processing as follows:

1. $\text{SQLAToSQL}(Q)$ defines a global variable, called Org-TABS, and adds every table “ T z ” in the “from” clause of Q to this set.
2. For every occurrence of “ $x.A = y.B$ ”, where both A and B are of type `eid`, $\text{SQLAToSQL}(Q)$

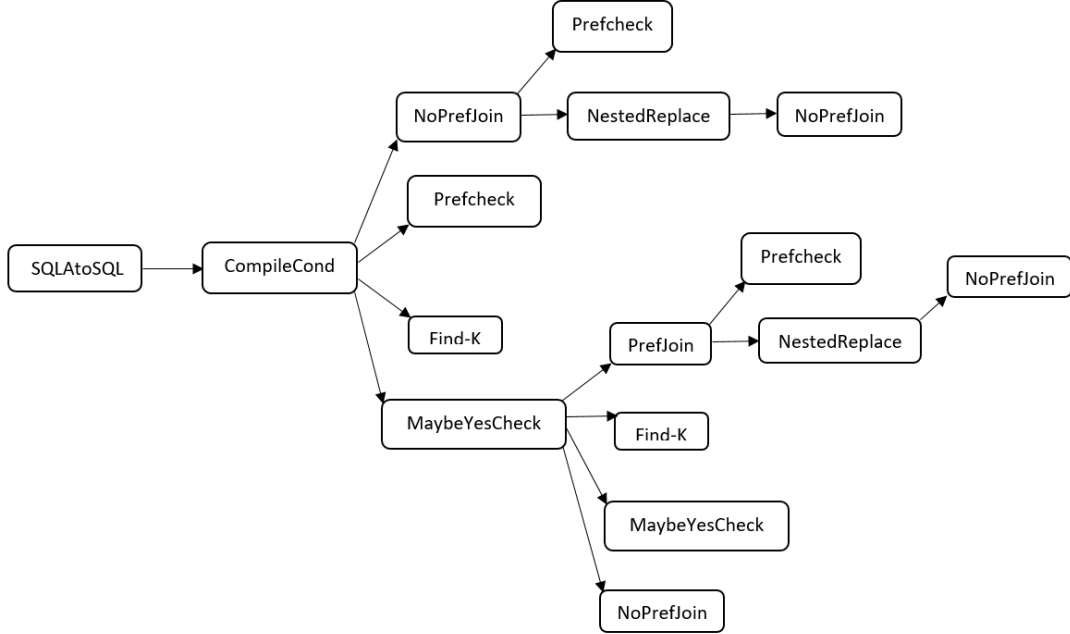


Figure 3.10: A dependency graph of functions and procedures that are called to map an SQLA query to an SQL query.

SQL(Q) generates a set, called E-WHERE-TABS. If “ $x.A = y.B$ ” exists in an “exists” clause, every table “ T_r ” in the “from” clause of the “exists” clause is added to E-WHERE-TABS.

3. SQLATOSQL(Q) replaces any occurrence of “ $x.A = y.B$ ” in Q with an SQL query in the format (3.2), that is computed by function `COMPILECOND(x, A, T_1 , y, B, T_2 , E-WHERE-TABS)`.

Considering the SQLA query Q in (3.2) again, function `COMPILECOND(x, A, $\text{RAN}(\text{FROMTAB}(x, Q), A)$, y, B, $\text{RAN}(\text{FROMTAB}(y, Q), B)$, E-WHERE-TABS)` which is defined in Algorithm 19, is called to compile “ $x.A = y.B$ ” by replacing it with an SQL query Q' that has the format in (3.2). Function calls `FROMTAB(x, Q)` and `FROMTAB(y, Q)` return tables T_1 and T_2 respectively, since Q contains “ $T_1 x$ ” and “ $T_2 y$ ” in its “from” clause.

Therefore, $\text{COMPILECOND}(x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ replaces $x.A(T_1) = y.B(T_2)$ with an SQL query in the format (3.2), by checking the following conditions:

1. If there exists a translation table between T_1 and T_2 , then function $\text{NOPREFJOIN}(x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ or function $\text{NOPREFJOIN}(y, B, T_2, x, A, T_1, \text{E-WHERE-TABS})$ is called, depending on the position of T_1 and T_2 in PRO .
2. If there is a preference clause between T_1 and T_2 , then function $\text{PREFCHECK}(x.A(T_1), y.B(T_2))$ is called first to compare primary keys of T_1 and T_2 . However, if $\text{OFFSET}(T_1)$ is not the first offset in $\text{PREF}(T_2)$ or $\text{OFFSET}(T_2)$ is not the first offset in $\text{PREF}(T_1)$, then comparing their primary keys is not enough to capture every intersection entity between T_1 and T_2 , so function $\text{FIND-K}(T_1, T_2, k)$ is called, when k is initially set to 0. If k is not 0, then function $\text{MAYBEYESCHECK}(k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ is called to compute any remaining common entities between T_1 and T_2 and illustrate them in an SQL query which has the format in (3.2).

Function $\text{FIND-K}(T_1, T_2, k)$ which is illustrated in Algorithm 20, inputs two tables T_1 and T_2 , such that $\text{OFFSET}(T_1) < \text{OFFSET}(T_2)$, and k is an integer, where it is initially set to 0. This function finds a set, which is called possible- k , and it contains possible table offsets such that each offset p in this set should have the following conditions:

1. p is greater than k .
2. p is less than $\text{OFFSET}(T_1)$.
3. $\text{TAB}(p)$ should not be disjoint with T_1 and T_2 .
4. There should exists a translation table between $\text{TAB}(p)$ and at least one of T_1 or T_2 .

The minimum p in the possible- k is the new k which is returned by the $\text{FIND-K}(T_1, T_2, 0)$.

Function $\text{MAYBEYESCHECK}(k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ which is indicated in Algorithm 21, checks the following three cases for offset k :

1. If k is in $\text{PREF}(T_1)$ and in $\text{TT}(T_2)$: function $\text{PREFJOIN}(r, \text{self}, \text{TAB}(k), y, B, T_2, x, A, T_1, \text{E-WHERE-TABS})$ is called to find common entities between three tables $\text{TAB}(k)$, T_1 and T_2 , and returns a WHERE-TABS and a WHERE-EQS for an SQL query. The input “ r ” in this function refers to tuples in $\text{TAB}(k)$.
2. If k is in $\text{PREF}(T_2)$ and in $\text{TT}(T_1)$: function $\text{PREFJOIN}(r, \text{self}, \text{TAB}(k), x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ is called to find common entities between three tables $\text{TAB}(k)$, T_1 and T_2 , and returns a WHERE-TABS and a WHERE-EQS for an SQL query.
3. If k is in $\text{TT}(T_1)$ and in $\text{TT}(T_2)$: function $\text{NOPREFJOIN}(r, \text{self}, \text{TAB}(k), x, A, T_1, \text{E-WHERE-TABS})$ is called to find common entities between $\text{TAB}(k)$ and T_1 , and it returns a WHERE-TABS1 and a WHERE-EQS1 for part of an SQL query. Then, function $\text{NOPREFJOIN}(r, \text{self}, \text{TAB}(k), y, B, T_2, \text{E-WHERE-TABS})$ is called to find common entities between $\text{TAB}(k)$ and T_2 , and it returns a WHERE-TABS2 and a WHERE-EQS2 for part of an SQL query. Strings in WHERE-TABS1 and WHERE-TABS2 are concatenated and stored as the variable WHERE-TABS and strings in WHERE-EQS1 and WHERE-EQS2 are concatenated and stored as the variable WHERE-EQS. The input “ r ” in these two functions refer to tuples in $\text{TAB}(k)$.

Using the WHERE-TABS and WHERE-EQS that are generated by each of the above cases, an SQL query in the format (3.2) is generated and stored in an string E . Afterwards, this function calls $\text{FIND-K}(T_1, T_2, k)$ to generate a new k . If new k is 0, $\text{MAYBEYESCHECK}(k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ returns E and if not, it calls $\text{MAYBEYESCHECK}(k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ with the new k to do the same process and extends E .

Considering function $\text{TRANSLATIONTCHECK}(g, \text{self}, T_k, e, \text{self}, T_m, \text{E-WHERE-TABS}, \text{WHERE-TABS})$ in Algorithm 22, the input “ g ” refers to tuples in table T_k , and the input “ e ” refers to tuples in table T_m . This function is called when there exists a translation table T_k-T_m-C . It checks for type of T_k-T_m-C , and returns an SQL query that has the format in (3.2). This algorithm works as follows:

1. If T_k-T_m-C is absorbed on the left, common entities between T_k-C and T_m-C are found by checking the absorbed column(s) in T_k-C which include(s) the primary key of T_m-C .
2. If T_k-T_m-C is absorbed on the right, common entities between T_k-C and T_m-C are found by checking the absorbed column(s) in T_m-C which include(s) the primary key of T_k-C .
3. If T_k-T_m-C can be computed from the join of other two translation tables TT_1 and TT_2 , $\text{NESTEDREPLACE}(TT_1, TT_2, \text{E-WHERE-TABS})$ is called to join TT_1 and TT_2 via the use of primary keys of tables involved in these translation tables.
4. If T_k-T_m-C is not absorbed and can not be computed from the join of other translation tables:
 - (a) If T_k-T_m-C does not exist in Org-TABS or E-WHERE-TABS, it is added to the WHERE-TABS.
 - (b) Join T_k-C and T_m-C with T_k-T_m-C by comparing their primary keys.

Considering function $\text{NOPREFJOIN}(r, C, T_k, z, D, T_m)$ in Algorithm 23, the input “r” refers to tuples in table T_k , where the input “C” is a column in T_k , and the input “z” refers to tuples in table T_m , where the input “D” is a column in T_m . This function is called when there exists a translation table T_k-T_m-C . It executes as follows and returns strings WHERE-TABS and WHERE-EQS which are part of an SQL query in the format (3.2).

1. If C is an abstract foreign key referencing T_k , function $\text{PREFCHECK}(r.C(T_k), g.\text{self}(T_k))$ is called to compile “ $r.C = g.\text{self}$ ” in a relational schema.
2. If each of T_k and T_m does not already exist in Org-TABS or E-WHERE-TABS, it is added to WHERE-TABS.
3. Call function $\text{TRANSLATIONTCHECK}(g, \text{self}, T_k, e, \text{self}, T_m)$ to check for special cases of the translation table T_k-T_m-C .

Considering function `PREFJOIN(r, C, T_k , z, D, T_m , s, E, T_n , E-WHERE-TABS)` in Algorithm 24, the input “r” refers to tuples in table T_k , where the input “C” is a column in T_k , the input “z” refers to tuples in table T_m , where the input “D” is a column in T_m , and the input “s” refers to tuples in table T_n , where the input “E” is a column in T_n . This function is called when there exists a translation table T_k - T_m -C, and T_n contains `PREF(T_k)`. This algorithm joins T_k with T_m and T_n accordingly to find common entities between them, and it returns an SQL query in the format (3.2) to illustrate such entities. Detail of how this algorithm executes is as follows:

1. If attribute C is an `eid` foreign key referencing T_k , function `PREFCHECK(r.C(T_k), g.self(T_k))` is called to find entities in T_k which C is referencing. The same holds for D and E.
2. If each of tables T_k , T_m and T_n does not already exist in E-WHERE-TABS or Org-TABS, it is added to WHERE-TABS.
3. Find common attributes between T_n and T_k to compile `h.self(T_n) = g.self(T_k)`, by calling `PREFCHECK(h.self(T_n), g.self(T_k))`.
4. Call function `TRANSLATIONTCHECK(g, self, T_k , e, self, T_m , E-WHERE-TABS)` to check for special cases of the translation table T_k - T_m -C.

Function `NESTEDREPLACE(TT_1 , TT_2 , E-WHERE-TABS)` which is presented in Algorithm 25, finds left table of TT_1 by calling `TABLEFT(TT_1)` and right table of TT_1 by calling `TABRIGHT(TT_1)`. This algorithm further calls `NOREFJOIN(p, self, TABLEFT(TT_1), z, self, TABRIGHT(TT_1), E-WHERE-TABS)` to join `TABLEFT(TT_1)` and `TABRIGHT(TT_1)` with TT_1 , and returns a WHERE-TABS1 and WHERE-EQS1 for generating an SQL query. Then, this algorithm calls `NOREFJOIN(s, self, TABLEFT(TT_2), t, self, TABRIGHT(TT_2), E-WHERE-TABS)` to join `TABLEFT(TT_2)` and `TABRIGHT(TT_2)` with TT_2 , and returns a WHERE-TABS2 and WHERE-EQS2 for generating an SQL query. Thereafter, string in WHERE-TABS1 is concatenated with string in WHERE-TABS2 and stored in a variable called WHERE-TABS, and string in WHERE-EQS1 is concatenated with string in

WHERE-EQS2 and stored in a variable called WHERE-EQS. `NESTEDREPLACE(TT1, TT2, E-WHERE-TABS)` returns WHERE-TABS and WHERE-EQS for an SQL query in the format (3.2).

Function `PREFCHECK(x.A(T1), y.B(T2))` which is introduced in Algorithm 26, compares `PKC(T1)` with `PKC(T2)` as follows:

1. If neither T_1 nor T_2 has a preference clause, each attribute A_1, \dots, A_k of `PKC(T1)` is compared with each attribute B_1, \dots, B_m of `PKC(T2)` in terms.
2. If T_1 has a preference clause and T_2 does not, `disc` attribute of $T_1\text{-C}$ is compared with `OFFSET(T2)` and `f` attribute of $T_1\text{-C}$ is compared with concatenation of each attribute B_1, \dots, B_m of `PKC(T2)`.
3. If T_2 has a preference clause and T_1 does not, `disc` attribute of $T_2\text{-C}$ is compared with `OFFSET(T1)` and `f` attribute of $T_2\text{-C}$ is compared with concatenation of each attribute A_1, \dots, A_k of `PKC(T1)`.
4. If both T_1 and T_2 have a preference clause, `disc` and `f` attributes of $T_1\text{-C}$ are compared with `disc` and `f` attributes of $T_2\text{-C}$ respectively.

Algorithm 18

```
procedure SQLToSQL (Q)
  Org-TABS = ()
  for each table  $T$  in  $Q$ 's "from" clause
    Add  $T$  GENVARNAME( $T$ ) to the set Org-TABS
  while there exists " $x.A = y.B$ ", where  $\text{DOM}(\text{FROMTAB}(x, Q), A) = \text{DOM}(\text{FROMTAB}(y, Q), B) = \text{eid}$ 
    E-WHERE-TABS = ()
    if " $x.A = y.B$ " exists in an "exists" clause  $E$  then
      for each table  $T$  in  $E$ 's "from" clause
        Add  $T$  GENVARNAME( $T$ ) to the set E-WHERE-TABS
      Replace " $x.A = y.B$ " by  $\text{COMPILECOND}(x, A, \text{RAN}(\text{FROMTAB}(x, Q), A), y, B, \text{RAN}(\text{FROMTAB}(y, Q), B), \text{E-WHERE-TABS})$ 
```

Algorithm 19

```
function COMPILECOND ( $x, A, T_1, y, B, T_2, \text{E-WHERE-TABS}$ )
   $E := "$ "
  if  $\text{OFFSET}(T_2)$  is in  $\text{TT}(T_1)$  then
    if  $\text{OFFSET}(T_1) \leq \text{OFFSET}(T_2)$  then
      WHERE-TABS, WHERE-EQS =  $\text{NOPREFJOIN}(x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ 
    else
      WHERE-TABS, WHERE-EQS =  $\text{NOPREFJOIN}(y, B, T_2, x, A, T_1, \text{E-WHERE-TABS})$ 
      Iterate through each equality statement in WHERE-EQS, and reverse the left side of it with its right side.
    if WHERE-TABS is empty then
       $E := \text{Concat}(E, \text{WHERE-EQS})$ 
    else
       $E := \text{Concat}(E, \text{WHERE-EQS})$ 
       $E := \text{Concat}(E, \text{exists}(\text{select } * \text{ from WHERE-TABS where WHERE-EQS}))$ 
  else
     $\triangleright T_1$  and  $T_2$  are either disjoint or there is preference clause between them.
     $E := \text{Concat}(E, "(, \text{PREFCHECK}(x.A(T_1), y.B(T_2)), ")")$ 
    if  $\text{OFFSET}(T_1)$  is not the first offset in  $\text{PREF}(T_2)$  or  $\text{OFFSET}(T_2)$  is not the first offset in  $\text{PREF}(T_1)$  then
       $k = 0$ 
       $k := \text{FIND-K}(T_1, T_2, k)$ 
      if  $k = 0$  then
        return  $E$ 
       $E := \text{Concat}(E, \text{MAYBEYESCHECK}(k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS}))$ 
  return  $E$ 
```

Algorithm 20

```
function FIND-K ( $T_1, T_2, k$ )
  Possible-k :=  $p : p > k$ 
               and  $p < \min(\text{OFFSET}(T_1), \text{OFFSET}(T_2))$ 
               and  $p \text{ not in } (\text{DISJ}(T_1) \cup (\text{DISJ}(T_2)))$ 
               and  $p \text{ in } (TT(T_1) \cup TT(T_2))$ 

  if Possible-k is non-empty then
     $k := \min(\text{Possible-k})$ 
  else
     $k := 0$ 
  return k
```

Algorithm 21

```
function MAYBEYESCHECK ( $k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS}$ )
   $E := \text{“”}$ 
  if  $k \text{ in Pref}(T_1)$  then
     $r = \text{GENVARNAME}(\text{TAB}(k))$ 
     $\text{WHERE-TABS}, \text{WHERE-EQS} = \text{PREFJOIN}(r, \text{self}, \text{TAB}(k), y, B, T_2, x, A, T_1, \text{E-WHERE-TABS})$ 
  if  $k \text{ in pref}(T_2)$  then
     $r = \text{GENVARNAME}(\text{TAB}(k))$ 
     $\text{WHERE-TABS}, \text{WHERE-EQS} = \text{PREFJOIN}(r, \text{self}, \text{TAB}(k), x, A, T_1, y, B, T_2, \text{E-WHERE-TABS})$ 
     $\triangleright k \text{ is not in neither Pref}(T_1) \text{ nor Pref}(T_2)$ 
  else
     $r = \text{GENVARNAME}(\text{TAB}(k))$ 
     $\text{WHERE-TABS1}, \text{WHERE-EQS1} = \text{NOPREFJOIN}(r, \text{self}, \text{TAB}(k), x, A, T_1, \text{E-WHERE-TABS})$ 
     $\text{E-WHERE-TABS} = \text{Concat}(\text{E-WHERE-TABS}, \text{WHERE-TABS1})$ 
     $\text{WHERE-TABS2}, \text{WHERE-EQS1} = \text{NOPREFJOIN}(r, \text{self}, \text{TAB}(k), y, B, T_2, \text{E-WHERE-TABS})$ 
     $\text{WHERE-TABS} := \text{Concat}(\text{WHERE-TABS}, \text{WHERE-TABS1}, \text{WHERE-TABS2})$ 
     $\text{WHERE-EQS} := \text{Concat}(\text{WHERE-EQS}, \text{WHERE-EQS1}, \text{WHERE-EQS2})$ 
   $E := \text{Concat}(E, \text{“or exists(select * from WHERE-TABS where WHERE-EQS”})$ 
   $k = \text{FIND-K}(T_1, T_2, k)$ 
  if  $k = 0$  then return E
   $E := \text{Concat}(E, \text{MAYBEYESCHECK}(k, x, A, T_1, y, B, T_2, \text{E-WHERE-TABS}))$ 
  return E
```

Algorithm 22

```
function TRANSLATIONTCHECK (g, self,  $T_k$ , e, self,  $T_m$ , E-WHERE-TABS)
  WHERE-TABS := " "
  if ABSORB( $T_k - T_m - \mathbf{C}$ ) is "true on the left" then                                 $\triangleright$  ( $T_k - T_m - C$ ) is absorbed on
                                                                                       the left
    WHERE-EQS := Concat(WHERE-EQS, "("), "and", "(", PREFCHECK(g.self( $T_m$ ), e.self( $T_m$ )),
                        ")")
  if ABSORB( $T_k - T_m - \mathbf{C}$ ) is "true on the right" then                                 $\triangleright$  ( $T_k - T_m - C$ ) is absorbed
                                                                                       on the right
    WHERE-EQS := Concat(WHERE-EQS, "(", "and", "(", PREFCHECK(g.self( $T_k$ ), e.self( $T_k$ )),
                        ")")
  if REPLACE( $T_k - T_m - \mathbf{C}$ ) == ( $TT_1$ ,  $TT_2$ ) then                                 $\triangleright$  ( $T_k - T_m - \mathbf{C}$ ) is replaced by
                                                                                       joining  $TT_1$  and  $TT_2$ 
    WHERE-TABS1, WHERE-EQS1 := NESTEDREPLACE( $TT_1$ ,  $TT_2$ , E-WHERE-TABS)
    WHERE-TABS := Concat(WHERE-TABS, WHERE-TABS1)
    WHERE-EQS := Concat(WHERE-EQS, WHERE-EQS1)
  else                                 $\triangleright$   $T_k - T_m - \mathbf{C}$  is neither absorbed nor replaced
    if  $T_k - T_m - \mathbf{C}$  w is not in (Org-TABS  $\cup$  E-WHERE-TABS) then
      WHERE-TABS := Concat(WHERE-TABS,  $T_k - T_m - \mathbf{C}$  w)
    WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(g.self ( $T_k$ ), w.self( $T_k$ )), ")", "and",
                        "(", PREFCHECK(w.self( $T_m$ ), e.self( $T_m$ )), ")")
  return WHERE-TABS, WHERE-EQS
```

Algorithm 23

```
function NOPREFJOIN (r, C,  $T_k$ , z, D,  $T_m$ , E-WHERE-TABS)
  WHERE-TABS = ""
  WHERE-EQS = ""

   $\triangleright$  Checking if D is an eid foreign key referencing  $T_m$ 
  if D <> self then
    e = GENVARNAME( $T_m$ )
    WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(z.D( $T_m$ ), e.self( $T_m$ )), ")")
  else
    e = z
     $\triangleright$  Checking if C is an eid foreign key referencing  $T_k$ 
    if C <> self then
      g = GENVARNAME( $T_k$ )
      WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(r.C( $T_k$ ), g.self( $T_k$ )), ")")
    else
      g = r
       $\triangleright$  The next two "If" statements check if  $T_k$  and  $T_m$ 
      already exist in the Org-TABS or E-WHERE-TABS
    if  $T_k$  g is not in (Org-TABS  $\cup$  E-WHERE-TABS) then
      WHERE-TABS := Concat(WHERE-TABS,  $T_k$  g)
    if  $T_m$  e is not in (Org-TABS  $\cup$  E-WHERE-TABS) then
      WHERE-TABS := Concat(WHERE-TABS,  $T_m$  e)
    E-WHERE-TABS := Concat(E-WHERE-TABS, WHERE-TABS)
    WHERE-TABS1, WHERE-EQS1 = TRANSLATIONTCHECK(g, self,  $T_k$ , e, self,  $T_m$ , E-WHERE-
      TABS)
    WHERE-TABS := Concat(WHERE-TABS, WHERE-TABS1)
    WHERE-EQS := Concat(WHERE-EQS, WHERE-EQS1)
  return WHERE-TABS, WHERE-EQS
```

Algorithm 24

```
function PREFJOIN (r, C,  $T_k$ , z, D,  $T_m$ , s, E,  $T_n$ , E-WHERE-TABS)
  WHERE-TABS = ""
  WHERE-EQS = ""

   $\triangleright$  Checking if D is an eid foreign key referencing  $T_m$ 
  if D <> self then
    e = GENVARNAME( $T_m$ )
    WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(z.D( $T_m$ ), e.self( $T_m$ )), ")")
  else
    e = z
     $\triangleright$  Checking if C is an eid foreign key referencing  $T_k$ 
  if C <> self then
    g = GENVARNAME( $T_k$ )
    WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(r.C( $T_k$ ), g.self( $T_k$ )), ")")
  else
    g = r
     $\triangleright$  Checking if E is an eid foreign key referencing  $T_n$ 
  if E <> self then
    h = GENVARNAME( $T_n$ )
    WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(s.E( $T_n$ ), h.self( $T_n$ )), ")")
  else
    h = s
     $\triangleright$  The next three "If" statements check if  $T_k$ ,  $T_m$  and  $T_n$ 
    already exist in the Org-TABS or E-WHERE-TABS
  if  $T_k$  g is not in (Org-TABS  $\cup$  E-WHERE-TABS) then
    WHERE-TABS := Concat(WHERE-TABS,  $T_k$  g)
  if  $T_m$  e is not in (Org-TABS  $\cup$  E-WHERE-TABS) then
    WHERE-TABS := Concat(WHERE-TABS,  $T_m$  e)
  if  $T_n$  h is not in (Org-TABS  $\cup$  E-WHERE-TABS) then
    WHERE-TABS := Concat(WHERE-TABS,  $T_n$  h)

     $\triangleright$  Compiling h.self( $T_n$ ) = g.self( $T_k$ ), where  $T_n$  contains PREF( $T_k$ )
  WHERE-EQS := Concat(WHERE-EQS, "(", PREFCHECK(h.self( $T_n$ ), g.self( $T_k$ )), ")")
  E-WHERE-TABS := Concat(E-WHERE-TABS, WHERE-TABS)
  WHERE-TABS1, WHERE-EQS1 = TRANSLATIONTCHECK(g, self,  $T_k$ , e, self,  $T_m$ , E-WHERE-
    TABS)
  WHERE-TABS := Concat(WHERE-TABS, WHERE-TABS1)
  WHERE-EQS := Concat(WHERE-EQS, WHERE-EQS1)
  return WHERE-TABS, WHERE-EQS
```

Algorithm 25

```
function NESTEDREPLACE ( $TT_1, TT_2, \text{E-WHERE-TABS}$ )
  WHERE-TABS = ""
  WHERE-EQS = ""
  p = GENVARNAME(TABLEFT( $TT_1$ ))
  z = GENVARNAME(TABRIGHT( $TT_1$ ))
  s = GENVARNAME(TABLEFT( $TT_2$ ))
  t = GENVARNAME(TABRIGHT( $TT_2$ ))
  WHERE-TABS1, WHERE-EQS1 = NoPREFJOIN(p, self, TABLEFT( $TT_1$ ), z,
                                         self, TABRIGHT( $TT_1$ ), E-WHERE-TABS)
  E-WHERE-TABS = Concat(E-WHERE-TABS, WHERE-TABS1)
  WHERE-TABS2, WHERE-EQS2 = NoPREFJOIN(s, self, TABLEFT( $TT_2$ ), t,
                                         self, TABRIGHT( $TT_2$ ), E-WHERE-TABS)
  WHERE-TABS := Concat(WHERE-TABS, WHERE-TABS1, WHERE-TABS2)
  WHERE-EQS := Concat(WHERE-EQS, WHERE-EQS1, WHERE-EQS2)
  return WHERE-TABS, WHERE-EQS
```

Algorithm 26

```
function PREFCHECK ( $x.A(T_1), y.B(T_2)$ )
   $A_1, \dots, A_k := \text{PKC}(T_1)$ 
   $B_1, \dots, B_m := \text{PKC}(T_2)$ 
  if PREFNUM( $T_1$ ) = 0 and PREFNUM( $T_2$ ) = 0 then :
    E := Concat(E, x.comp(A,  $A_1$ ), ..., x.comp(A,  $A_k$ )) =
      Concat(y.comp(B,  $B_1$ ), ..., y.comp(B,  $B_m$ ))"
  if PREFNUM( $T_1$ ) > 0 and PREFNUM( $T_2$ ) = 0 then:
    E := x.comp(A, disc) = OFFSET( $T_2$ ) and x.comp(A, f) =
      Concat(y.comp(B,  $B_1$ ), ..., y.comp(B,  $B_m$ ))
  if PREFNUM( $T_1$ ) = 0 and PREFNUM( $T_2$ ) > 0 then:
    E := OFFSET( $T_1$ ) = y.comp(B, disc) and Concat(x.comp(A,  $A_1$ ), ...,
      x.comp(A,  $A_k$ )) = y.comp(B, f)"
  if PREFNUM( $T_1$ ) > 0 and PREFNUM( $T_2$ ) > 0 then:
    E := x.Comp(A, disc) = y.Comp(B, disc) and x.Comp(A, f) =
      y.Comp(A, f)
  else:
    E := False
  return E
```

Example 1.

This example uses the same schema, SecondUNIV, as Example 1 of Section 3.1 where only translation tables are involved. The ARM schema for SecondUNIV is based on UNIV in Figure 2.3. Table declarations and RETs for SecondUNIV are illustrated in Appendices A.4 and A.5, respectively. The following SQLP query over SecondUNIV indicates how to find the names of all professors who are also students:

```
select pr.name
from PROFESSOR pr, STUDENT s
where pr.self = s.self
```

This query is then converted to an SQLA query over SecondUNIV, as shown in Figure 3.11a.

The RM version of SecondUNIV is called SecondUNIV' and to compile the SQLA query Q in Figure 3.11a and map it to an SQL one over SecondUNIV', the algorithm SQLAToSQL(Q) is called. It first defines a set called Org-TABS, which here is the set ("PROFESSOR", "PERSON"). Recall that for every occurrence of "x.A = y.B", where both A and B are of type `eid`, SQLAToSQL(Q) generates a set called E-WHERE-TABS. If "x.A = y.B" exists in an "exists" clause, every table "T r" in the "from" clause of the "exists" clause is added to the E-WHERE-TABS. The variable E-WHERE-TABS for "pr.self = pe.self" is empty, since it does not exist in an "exists" clause. SQLAToSQL(Q) replaces "pr.self = pe.self" with an SQL query that is computed by `COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, self)`, and has the format in (3.2). Since "pe.self = s.self" exists in an "exists" clause, E-Where-TABS for "pe.self = s.self" is the set (STUDENT). SQLAToSQL(Q) then replaces "pe.self = s.self" with an SQL query that is computed by `COMPILECOND(pe, self, PERSON, s, self, STUDENT, (STUDENT))`, and also has the format in (3.2).

Function `COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, self)`, (()) first checks for the existence of a translation table between PROFESSOR and PERSON. Since the translation table PROFESSOR-PERSON-C exists, function `NO_PREFJOIN(pr, self, PROFESSOR,`

```

select pe.name
from PROFESSOR-C pr, PERSON-C pe
where pr.self = pe.self
and exists(select * from STUDENT-C s
           where pe.self = s.self)

```

(a) An SQLA query Q over SecondUNIV.

```

select pe.name
from PROFESSOR-C pr, PERSON-C pe
and pr.sin = pe.sin
and exists (select * from STUDENT-C s
            where pe.self = s.self)

```

(b) An intermediate SQL query Q' over SecondUNIV'.

```

select pe.name
from PROFESSOR-C pr, PERSON-C pe
where pr.sin = pe.sin
and exists(select * from STUDENT-C s
            where pe.sin = s.sin)

```

(c) An SQL query Q' over SecondUNIV'

Figure 3.11: Conversion of an SQLA query over SecondUNIV to an SQL query over SecondUNIV'.

pe, self, PERSON, ()) is called to find the intersection entities between PROFESSOR-C and PERSON-C, and returns a WHERE-EQS and a WHERE-TABS for an SQL query in the format (3.2) that illustrates such entities. This function first checks for existence of tables PROFESSOR and PERSON in either Org-WHERE-TABS or E-WHERE-TABS. Since both PROFESSOR and PERSON already exist in Org-WHERE-TABS, they are not added to WHERE-TABS, and WHERE-TABS remains empty. Then, it calls function TRANSLATIONTCHECK(pr, self, PROFESSOR, pe, self, PERSON, ()) to check for the type of translation table PROFESSOR-PERSON-C. Since PROFESSOR-PERSON-C is “absorbed on the left”, the attribute PKC(PERSON) = sin is added to PROFESSOR-C, as also explained in Example 1 of 3.1. TRANSLATIONTCHECK(pr, self, PROFESSOR, pe, self, PERSON, ()) re-

turns an empty WHERE-TABS and WHERE-EQS which illustrates the common entities between PROFESSOR-C and PERSON-C, as follows:

$$\begin{aligned}\text{WHERE-EQS} &:= \text{PREFCHECK}(\text{pr. self}(\text{PERSON}), \text{pe. self}(\text{PERSON})) \\ \text{WHERE-TABS} &:= \text{“ ”}\end{aligned}\tag{3.5}$$

WHERE-EQS is simplified to the following WHERE-EQS, where NOPREFJOIN(pr, self, PROFESSOR, pe, self, PERSON, ()) returns WHERE-EQS and WHERE-TABS, as follows:

$$\begin{aligned}\text{WHERE-EQS} &:= \text{“pr.sin = pe.sin”} \\ \text{WHERE-TABS} &:= \text{“ ”}\end{aligned}\tag{3.6}$$

Thereafter, COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, self), () returns the following string:

$$\text{pr.sin} = \text{pe.sin}\tag{3.7}$$

Therefor, the SQLA query in Figure 3.11a is now converted to an intermediate SQL in Figure 3.11b. The next function COMPILECOND(pe, self, PERSON, s, self, STUDENT, (STUDENT)) first checks for the existence of a translation table between STUDENT and PERSON. Since there exists the translation table STUDENT-PERSON-C, the algorithm NOPREFJOIN(s, self, STUDENT, pe, self, PERSON, (STUDENT)) is called to find the common entities between PROFESSOR-C and PERSON-C, and returns WHERE-EQS and WHERE-TABS for an SQL query in the format (3.2) that illustrates such entities. This function first checks for the existence of tables STUDENT and PERSON in either Org-WHERE-TABS or E-WHERE-TABS. Since STUDENT already exists in E-WHERE-TABS and PERSON exists in Org-WHERE-TABS, they are not added to WHERE-TABS, and WHERE-TABS remains empty. Then, it calls function TRANSLATIONTCHECK(s, self, STUDENT, pe, self, PERSON, (STUDENT)) to check for the type of translation table STUDENT-PERSON-C. Since STUDENT-PERSON-C is “absorbed on the left”, the attribute sin is added to STUDENT-C. The following illustrates common entities between STUDENT-C and PERSON-C that are stored in WHERE-EQS and

an empty WHERE-TABS.

$$\begin{aligned}\text{WHERE-EQS} &:= \text{PREFCHECK}(s.\text{self}(\text{PERSON}), \text{pe}.\text{self}(\text{PERSON})) \\ \text{WHERE-TABS} &:= \text{“ ”}\end{aligned}\tag{3.8}$$

WHERE-EQS is simplified to the following, where both $\text{TRANSLATIONTCHECK}(s, \text{self}, \text{STUDENT}, \text{pe}, \text{self}, \text{PERSON}, (\text{STUDENT}))$ and $\text{NOPREFJOIN}(s, \text{self}, \text{STUDENT}, \text{pe}, \text{self}, \text{PERSON}, (\text{STUDENT}))$ return WHERE-EQS and WHERE-TABS, as follows:

$$\begin{aligned}\text{WHERE-EQS} &:= \text{“s.sin = pe.sin”} \\ \text{WHERE-TABS} &:= \text{“ ”}\end{aligned}\tag{3.9}$$

Notice that $\text{COMPILECOND}(\text{pe}, \text{self}, \text{PERSON}, s, \text{self}, \text{STUDENT}, (\text{STUDENT}))$ should compile $\text{pe}.\text{self} = s.\text{self}$, but WHERE-EQS in (3.9) is instead $s.\text{sin} = \text{pe}.\text{sin}$. Therefore, $\text{COMPILECOND}(\text{pe}, \text{self}, \text{PERSON}, s, \text{self}, \text{STUDENT}, (\text{STUDENT}))$ reverses the left side of the equality of the string in WHERE-EQS in (3.9) with the right side, and returns the following result:

$$\text{pe.sin} = s.\text{sin}\tag{3.10}$$

Therefore, the SQLA query in Figure 3.11a is then converted to an SQL query in Figure 3.11c.

Figure 3.12 illustrates the algorithms that the procedure $\text{SQLAToSQL}(Q)$ calls to convert the SQLA query in Figure 3.11a to the SQL query in Figure 3.11c by compiling both $\text{pr}.\text{self} = \text{pe}.\text{self}$ and $\text{pe}.\text{self} = s.\text{self}$. The first column contains a function or a procedure call with its inputs, and the second column contains what the procedure produces or what the function returns. Except the first row, every other row contains a function that is called by a function or a procedure in the previous row.

Algorithm name	Result
SQLAToSQL(Q)	Figure 3.11c
COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, ())	(3.7)
NO_PREFJOIN(pr, self, PROFESSOR, pe, self, PERSON, ())	(3.6)
TRANSLATIONTCHECK(pr, self, PROFESSOR, pe, self, PERSON, ())	(3.6)
PREFCHECK(pr.self(PERSON), pe.self(PERSON))	(3.5)
COMPILECOND(pe, self, PERSON, s, self, STUDENT, (STUDENT))	(3.10)
NO_PREFJOIN(s, self, STUDENT, pe, self, PERSON, (STUDENT))	(3.9)
TRANSLATIONTCHECK(s, self, STUDENT, pe, self, PERSON, (STUDENT))	(3.9)
PREFCHECK(s.self(PERSON), pe.self(PERSON))	(3.8)

Figure 3.12: Procedure and function calls with their result for Section 3.2 Example 1.

Example 2.

This example uses the same schema, ThirdUNIV, as Example 2 of Section 3.1 where both translation and preference tables are involved. The ARM schema for ThirdUNIV is based on UNIV in Figure 2.3. Table declarations and RETs for ThirdUNIV are illustrated in Appendices A.8 and A.9, respectively. This example illustrates how the SQLA query Q in Figure 3.13a over ThirdUNIV, is mapped to an SQL query Q' over the RM schema ThirdUNIV'.

Procedure SQLAToSQL(Q) is called to map Q to Q'. Recall that SQLAToSQL(Q) defines the variable Org-TABS to store every table “T z” in the “from” clause of Q. Org-TABS here is the set (PROFESSOR, PERSON). As explained earlier in this section, for every occurrence of “x.A = y.B”, where both A and B are of type eid, SQLAToSQL(Q) generates a set called E-WHERE-TABS. If “x.A = y.B” exists in an “exists” clause, every table “T r” in the “from” clause of the “exists” clause is added to E-WHERE-TABS.


```

select pe.name
from PROFESSOR pr, PERSON pe
where pr.self = pe.self
and exists(select * from STUDENT s
           where pe.self = s.self)

```

(a) An SQLA query Q over ThirdUNIV.

```

select pe.name
from PROFESSOR-C pr, PERSON-C pe
where pr.5 = pe.disc
and Concat(pr.name, pr.office) = pe.f
and exists (select * from STUDENT s
            where pe.self = s.self)

```

(b) An intermediate SQL query Q' over ThirdUNIV'.

```

select pe.name
from PROFESSOR-C pr, PERSON-C pe
where pr.5 = pe.disc
and Concat(pr.name, pr.office) = pe.f
and exists (select * from STUDENT-C s
            where pe.disc = s.6
            and pe.f = s.snum
            or exists (select * from PROFESSOR-STUDENT-C w
                      where pe.disc = pr.5
                      and pe.f = Concat(pr.name, pr.office)
                      and pr.5 = w.5
                      and Concat(pr.name, pr.office) = Concat(w.name, w.office)
                      and w.snum = s.snum)

```

(c) An SQL query Q' over ThirdUNIV'.

Figure 3.13: Conversion of an SQLA query Q over ThirdUNIV to an SQL query Q' over ThirdUNIV'.

Since “pr.self = pe.self” does not exist in an “exists” clause, E-WHERE-TABS is empty here and SQLATOSQL(Q) calls function COMPILECOND(pr, self, PROFESSOR, pe, self,

PERSON, ()) to compile and replace “pr.self = pe.self” with an SQL query which has the format in (3.2). Now consider “pe.self = s.self” where it exists in an “exists” clause, E-WHERE-TABS is then the set (STUDENT). SQLAToSQL(Q) then calls function COMPILECOND(pe, self, PERSON, s, self, STUDENT, (STUDENT)) to compile and replace “pe.self = s.self” with an SQL query which has the format in (3.2).

COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, ()) first checks for the existence of a translation table between PROFESSOR and PERSON. However, there does not exist such translation table, since PERSON contains PREF(PROFESSOR). Since PROFESSOR is the first table in PERSON’s preference clause, all PROFESSOR entities who are also PERSON entities can be accessed by only comparing PKC(PROFESSOR) and PKC(PERSON), as follows:

$$\text{PREFCHECK}(\text{pr.self}(\text{PROFESSOR}), \text{pe.self}(\text{PERSON})) \quad (3.11)$$

Since PROFESSOR-C does not contain disc and f attributes, a disc attribute for PROFESSOR-C is considered to be OFFSET(PROFESSOR) = 5 and an f attribute for PROFESSOR-C is considered to be the concatenation of columns in PKC(PROFESSOR) which is Concat(name, office). Therefore, the following illustrates the result of computation in (3.11), such that it is returned by COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, ()):

$$\text{pr.5} = \text{pe.disc} \text{ and } \text{Concat}(\text{pr.name}, \text{pr.office}) = \text{pe.f} \quad (3.12)$$

String “pr.self = pe.self” in Figure 3.13a is replaced by the string in (3.12), as shown in Figure 3.13b.

COMPILECOND(pe, self, PERSON, s, self, STUDENT, (STUDENT)) first checks if there exists a translation table between PERSON and STUDENT and, since PERSON contains PREF(-STUDENT), there does not exist any translation table between them. Since STUDENT is the second table in PERSON’s preference clause, not all STUDENT entities who are also PERSON entities can be found by only comparing PKC(STUDENT) and PKC(PERSON), as follows:

$$\text{PREFCHECK}(\text{pe.self}(\text{PERSON}), \text{s.self}(\text{STUDENT})) \quad (3.13)$$

Since $\text{OFFSET}(\text{STUDENT})$ is 6 and $\text{PKC}(\text{STUDENT})$ is `snum`, this is simplified to the following:

$$\text{pe.disc} = \text{s.6} \text{ and } \text{pe.f} = \text{s.snum} \quad (3.14)$$

Consider a person entity who is both a professor and a student entity. This entity then has a `disc` attribute, which is the same as $\text{OFFSET}(\text{PROFESSOR}) = 5$. Therefore, this entity is not captured by the comparison in (3.14). To resolve this issue and find all common entities between `STUDENT-C` and `PERSON-C`, “`pe.self = s.self`” is replaced by an string `E` that is a logical disjunction. Therefore, `E` contains two parts that are separated by an “`or`” statement. The first part is simply a comparison between $\text{PKC}(\text{PERSON})$ and $\text{PKC}(\text{STUDENT})$, as indicated in (3.13) and simplified in (3.14). The second part of `E`, including the “`or`” statement, is an SQL query in the following format:

$$\begin{aligned} &\text{or exists (select * from WHERE-TABS} \\ &\quad \text{where WHERE-EQS)} \end{aligned} \quad (3.15)$$

The above SQL query is computed by function $\text{MAYBEYESCHECK}(5, \text{pe}, \text{self}, \text{PERSON}, \text{s}, \text{self}, \text{STUDENT}, (\text{STUDENT}))$. This function inputs an integer `k` which is 5 here. The variable `k` is computed by function $\text{FIND-K}(\text{PERSON}, \text{STUDENT}, k)$, when `k` is initially set to 0. $\text{FIND-K}(\text{PERSON}, \text{STUDENT}, 0)$ finds the set `possible-k` which contains possible table offsets, such that each offset `p` in this set should have the following conditions:

1. `p` is greater than `k`,
2. `p` is less than $\text{OFFSET}(\text{STUDENT})$,
3. $\text{TAB}(p)$ should not be disjoint with `STUDENT` and `PERSON`,
4. `p` should exist in at least one of the $\text{TT}(\text{STUDENT-C})$ or $\text{TT}(\text{PERSON-C})$.

The minimum `p` in `possible-k` is the new `k` which is returned by $\text{FIND-K}(\text{PERSON}, \text{STUDENT}, 0)$. Since $\text{OFFSET}(\text{STUDENT}) = 6$ and $\text{OFFSET}(\text{PERSON}) = 7$, the only possible offset

greater than 0 and less than 6 which is not disjoint with both **STUDENT** and **PERSON**, and exists in $(TT(STUDENT-C) \cup TT(PERSON-C))$, is $OFFSET(PROFESSOR) = 5$. Therefore, $FIND-K(PERSON, STUDENT, 0)$ returns $k = 5$.

The following illustrates a hypothetical maybe-yes list for each **STUDENT** and **PERSON**:

$$\begin{aligned}
 \text{STUDENT} : & \{ \text{no} , \text{no} , \text{no} , \text{no} , \text{maybe} \} \\
 \text{PRO} = & \{ \text{DEPARTMENT}, \text{COURSE}, \text{CLASS}, \text{ENROLLMENT}, \text{PROFESSOR}, \text{STUDENT}, \text{PERSON} \} \\
 \text{PERSON} : & \{ \text{no} , \text{no} , \text{no} , \text{no} , \text{yes} \}
 \end{aligned} \tag{3.16}$$

For each table T in **PRO**, such that T is not disjoint with **STUDENT** and $OFFSET(T) < OFFSET(STUDENT)$, if there exists a translation table T -**STUDENT**, string “maybe” is added to the **STUDENT**’s maybe-yes list at the position of $OFFSET(T)$. If $PREF(STUDENT)$ contains $OFFSET(T)$, then string “yes” is added to the **STUDENT**’s maybe-yes list, at the position of $OFFSET(T)$, and if T is disjoint with **STUDENT**, string “no” is added at the position of $OFFSET(T)$. **PERSON**’s maybe-yes list is then filled out similarly.

Since **STUDENT** is disjoint with **DEPARTMENT**, **COURSE**, **CLASS** and **ENROLLMENT**, string “no” is added to the position 1, 2, 3 and 4 of the **STUDENT**’s maybe-yes list. Since **STUDENT** entities who are also **PROFESSOR** entities can be captured through the translation table **STUDENT-PROFESSOR-C**, string “maybe” is added to the **STUDENT**’s maybe-yes list at the position of $OFFSET(PROFESSOR)$ which is 5, as shown in (3.16).

Since **PERSON** is also disjoint with **DEPARTMENT**, **COURSE**, **CLASS** and **ENROLLMENT**, string “no” is added to the position 1, 2, 3 and 4 of the **PERSON**’s maybe-yes list. Since **PERSON** contains $PREF(PROFESSOR)$, string “yes” is added to the **PERSON**’s maybe-yes list at the position 5, as also indicated in (3.16).

Function `MAYBEYESCHECK(5, pe, self, PERSON, st, self, STUDENT, (STUDENT))` is then called to compute the second part of **E** in the format (3.15), via the idea of a maybe-yes list for each **STUDENT** and **PERSON**. This algorithm checks for the relation of **PROFESSOR** with both **STUDENT** and **PERSON** in three cases :

1. The yes-maybe case: **PROFESSOR** exists in **PERSON**’s preference clause, and there is a

translation table between PROFESSOR and STUDENT.

2. The maybe-yes case: There exists a translation table between PROFESSOR, and PERSON and PROFESSOR exists in STUDENT's preference clause.
3. The maybe-maybe case: There exists a translation table between PROFESSOR and STUDENT, and between PROFESSOR and PERSON.

Clearly, the first case is true and the other two are false. Therefore, function PREFJOIN(pr, self, PROFESSOR, s, self, STUDENT, pe, self, PERSON, (STUDENT)) is called to join PERSON-C with PROFESSOR-C and PROFESSOR-STUDENT-C with both PROFESSOR-C and STUDENT-C.

Since all three tables PROFESSOR-C, STUDENT-C and PERSON-C exist in either Org-TABS or E-WHERE-TABS, neither of them is added to WHERE-TABS and it remains empty. PREFJOIN(pr, self, PROFESSOR, s, self, STUDENT, pe, self, PERSON, (STUDENT)) first finds the common attributes between PERSON and PROFESSOR and stores them in the following WHERE-EQS.

$$\begin{aligned} \text{WHERE-EQS} &:= \text{PREFCHECK}(\text{pe.self}(\text{PERSON}), \text{pr.self}(\text{PROFESSOR})) \\ \text{WHERE-TABS} &:= \text{“ ”} \end{aligned} \quad (3.17)$$

They are simplified to the following:

$$\begin{aligned} \text{WHERE-TABS} &:= \text{“ ”} \\ \text{WHERE-EQS} &:= (\text{pe.disc} = \text{pr.f} \text{ and } \text{pe.f} = \text{Concat}(\text{pr.name}, \text{pr.office})) \end{aligned} \quad (3.18)$$

Then, PREFJOIN(pr, self, PROFESSOR, s, self, STUDENT, pe, self, PERSON, (STUDENT)) calls function TRANSLATIONTCHECK(pr, self, PROFESSOR, s, self, STUDENT, (STUDENT)) to check for the type of PROFESSOR-STUDENT and join it with both PROFESSOR and STUDENT.

TRANSLATIONTCHECK(pr, self, PROFESSOR, s, self, STUDENT, (STUDENT)) first checks whether PROFESSOR-STUDENT-C is absorbed or can be computed from the join of other tables or neither. Since PROFESSOR-STUDENT-C is not absorbed and can not be computed from the join of other tables, “PROFESSOR-STUDENT-C w” is added to the following

WHERE-TABS. The following WHERE-EQS is also generated to contain an string that joins PROFESSOR-STUDENT-C with both PROFESSOR and STUDENT:

$$\begin{aligned} \text{WHERE-TABS} &:= (\text{PROFESSOR-STUDENT-C } w) \\ \text{WHERE-EQS} &:= (\text{PREFCHECK}(pr).self(\text{PROFESSOR}), w.self(\text{PROFESSOR}), \\ &\quad \text{PREFCHECK}(w.self(\text{STUDENT}), s.self(\text{STUDENT}))) \end{aligned} \quad (3.19)$$

The above WHERE-EQS is then simplified to the following, and TRANSLATIONTCHECK($pr, self, \text{PROFESSOR}, s, self, \text{STUDENT}, (\text{STUDENT})$) returns the following WHERE-TABS and WHERE-EQS:

$$\begin{aligned} \text{WHERE-TABS} &:= (\text{PROFESSOR-STUDENT-C } w) \\ \text{WHERE-EQS} &:= (\text{Concat}(pr.name, pr.office) = \text{Concat}(w.name, w.office), \\ &\quad w.snum = s.snum) \end{aligned} \quad (3.20)$$

Thereafter, PREFJOIN($pr, self, \text{PROFESSOR}, pe, self, \text{PERSON}, s, self, \text{STUDENT}$) concatenates WHERE-EQS in (3.17) with WHERE-EQS in (3.20) and returns the following WHERE-TABS and WHERE-EQS:

$$\begin{aligned} \text{WHERE-TABS} &:= (\text{PROFESSOR-STUDENT-C } w) \\ \text{WHERE-EQS} &:= (pe.disc = pr.5 \\ &\quad \text{and } pe.f = \text{Concat}(pr.name, pr.office) \\ &\quad \text{and } pr.5 = w.5 \\ &\quad \text{and } \text{Concat}(pr.name, pr.office) = \text{Concat}(w.name, w.office) \\ &\quad \text{and } w.snum = s.snum) \end{aligned} \quad (3.21)$$

Afterwards, FIND-K(PERSON, STUDENT, 5) is called to find the minimum k . Since there does not exist any tables in PRO with an offset greater than 5 and less than OFFSET(STUDENT) = 6, FIND-K(PERSON, STUDENT, 5) returns $k = 0$. Therefore, MAYBEYES-CHECK(5, $pe, self, \text{PERSON}, st, self, \text{STUDENT}, (\text{STUDENT})$) uses WHERE-TABS and WHERE-EQS in (3.21) to build the second part of the string E, containing the “or”

clause, and returns it, as follows:

```

or exists( select * from PROFESSOR-STUDENT-C w
          where pe.disc = pr.5
          and pe.f = Concat(pr.name, pr.office)
          and pr.5 = w.5
          and Concat(pr.name, pr.office) = Concat(w.name, w.office)
          and w.snum = s.snum)

```

(3.22)

COMPILECOND(pe, self, PERSON, st, self, STUDENT, (STUDENT)) then concatenates the string in (3.22) with the comparison in (3.14) and returns the following string:

```

pe.disc = s.6 and pe.f = s.snum
or exists( select * from PROFESSOR-STUDENT-C w
          where pe.disc = pr.5
          and pe.f = Concat(pr.name, pr.office)
          and pr.5 = w.5
          and Concat(pr.name, pr.office) = Concat(w.name, w.office)
          and w.snum = s.snum)

```

(3.23)

Therefore, “pe.self = s.self” in Figure 3.11a is replaced by the string in (3.23) and the SQLA query in Figure 3.11a is converted to the SQL query in Figure 3.13c. Figure 3.14 illustrates functions that procedure SQLATOSQL(Q) calls to convert the SQLA query in Figure 3.13a to the SQL query in Figure 3.13c by compiling “pr.self = pe.self”. The first column contains a function or a procedure call with its inputs, and the second column contains what the procedure produces or what the function returns. Except the first row, every other row contains a function that is called by a function or a procedure in the previous row.

Algorithm name	Result
SQLAtoSQL(Q)	Figure 3.13c
COMPILECOND(pr, self, PROFESSOR, pe, self, PERSON, ())	(3.12)
PREFCHECK(pr.self (PROFESSOR), pe.self (PERSON))	(3.12)
COMPILECOND(pe, self, PERSON, s, self, STUDENT, (STUDENT))	(3.23)
PREFCHECK(pe.self (PERSON), s.self (STUDENT))	(3.14)
FIND-K(PERSON, STUDENT, 0)	k = 5
MAYBEYESCHECK(5, pe, self, PERSON, s, self, STUDENT, (STUDENT))	(3.22)
PREFJOIN(pr, self, PROFESSOR, s, self, STUDENT, pe, self, PERSON, (STUDENT))	(3.21)
PREFCHECK(pe.self (PERSON), pr.self (PROFESSOR))	(3.21)
TRANSLATIONTCHECK(pr, self, PROFESSOR, s, self, STUDENT, (STUDENT))	(3.20)
PREFCHECK(pr.self (PROFESSOR), w.left (PROFESSOR))	(3.21)
PREFCHECK(w.self (STUDENT), s.self, (STUDENT)))	(3.21)
FIND-K(PERSON, STUDENT, 5)	k = 0

Figure 3.14: Procedure and function calls with their result for Section 3.2 Example 2.

Example 3.

This example uses the ARM schema UNIVPEOPLE in Figure [3.6](#) that has been introduced in Example [3](#) of Section 3.1, such that a combination of both translation and preference tables are generated for the RM version of UNIVPEOPLE which is called UNIPEOPLE', as indicated in Figure [3.9](#). Consider the SQLA query Q in Figure [3.15a](#) which finds the names of all students who are visitors from other universities in UNIVPEOPLE. This Example illustrates how to map Q to an SQL query Q' over UNIVPEOPLE'.

Procedure SQLAToSQL(Q) is called to map Q to Q'. Recall that SQLAToSQL(Q) defines a global variable called Org-TABS, and then it iterates through each table in Q's "from" clause and store it in Org-TABS where Org-TABS for this example is the set (STUDENT, VISITOR). Also recall that for every occurrence of "x.A = y.B", where both A and B are of type `eid`, SQLAToSQL(Q) generates a set called E-WHERE-TABS. If "x.A = y.B" exists in an "exists" clause, every table "T r" in the "from" clause of the "exists" clause is added to E-WHERE-TABS. In this example, SQLAToSQL(Q) considers "s.self = v.self", for which the set E-WHERE-TABS is defined empty, since "s.self = v.self" does not exist in an "exists" clause. Thereafter, SQLAToSQL(Q) calls function COMPILECOND(s, self, STUDENT, v, self, VISITOR, ()) to compile "s.self = v.self" and replace it with an string E which has the format in (3.2). This function first checks for an existence of a translation table between STUDENT and VISITOR. Since STUDENT exists in VISITOR's preference clause, there does not exist any translation table between them. Since STUDENT is the second table in VISITOR's preference clause, not all STUDENT entities who are also VISITOR entities can be found by only comparing PKC(STUDENT) and PKC(VISITOR), as shown here:

$$\text{PREFCHECK}(\text{s.self}(\text{STUDENT}), \text{v.self}(\text{VISITOR})) \quad (3.24)$$

Since OFFSET(STUDENT) is 3 and PKC(STUDENT) is `snum`, this computation results in the following:

$$\text{s.3} = \text{v.disc and s.snum} = \text{v.f} \quad (3.25)$$

```

select s.snum
from STUDENT s, VISITOR v
where (s.self = v.self)

```

(a) An SQLA query Q over UNIVPEOPLE.

```

select s.snum
from STUDENT-C s, VISITOR-C v
where (s.3 = v.disc and s.snum = v.f)

```

(b) An intermediate SQL query Q' over UNIVPEOPLE'.

```

select s.snum
from STUDENT-C s, VISITOR-C v
where (s.3 = v.disc and s.snum = v.f)
or exists( select * from EMPLOYEE-C e, EMPLOYEE-STUDENT-C z, EMPLOYEE-VISITOR-C r
           where (e.enum = z.enum)
                 and (z.snum = s.snum)
                 and (e.enum = r.enum)
                 and (r.disc = v.disc and r.f = v.f))

```

(c) An intermediate SQL query Q' over UNIVPEOPLE'.

```

select s.snum
from STUDENT-C s, VISITOR-C v
where s.3 = v.disc and s.snum = v.f
or exists( select * from EMPLOYEE-C e, EMPLOYEE-STUDENT-C z, EMPLOYEE-VISITOR-C r
           where (e.enum = z.enum)
                 and (z.snum = s.snum)
                 and (e.enum = r.enum)
                 and (r.disc = v.disc and r.f = v.f))
or exists( select * from PROFESSOR-C p, EMPLOYEE-C e, EMPLOYEE-STUDENT-C z
           where (v.disc = p.2 and v.f = Concat(p.name, p.office))
                 and (p.enum = e.enum)
                 and (e.enum = z.enum)
                 and (z.snum = s.snum)

```

(d) An SQL query Q' over UNIVPEOPLE'.

Figure 3.15: Conversion of an SQLA query Q over UNIVPEOPLE to an SQL query Q' over UNIVPEOPLE'.

Consider a visitor entity who is both a professor and a student entity, this entity has a `disc` attribute which is the same as $\text{OFFSET}(\text{PROFESSOR}) = 2$. Therefore, the comparison in (3.25) fails to find such entity. In order to resolve this issue and find all common entities between `STUDENT-C` and `VISITOR-C`, string “`s.self = v.self`” should be replaced with an string `E` that is a logical disjunction. In particular, `E` might contain only one “`or`” clause or multiple ones. The first part of `E` before the “`or`” statement, is simply a comparison between $\text{PKC}(\text{STUDENT})$ and $\text{PKC}(\text{VISITOR})$, as indicated in (3.24), and simplified in (3.25). The SQLA query in Figure 3.15a is now converted to an SQL query in Figure 3.15b, such that it includes the first part of `E`. The second part of `E` which includes the “`or`” statement is an SQL query in the format (3.15), and it is computed by function `MAYBEYESCHECK(1, s, self, STUDENT, v, self, VISITOR, ())`. This function inputs an integer `k` which is 1 here. We further explain how `k` is computed by function `FIND-K(STUDENT, VISITOR, k)`, when `k` is initially set to 0. `FIND-K(STUDENT, VISITOR, 0)` finds a set called possible-`k` which contains possible table offsets, such that each offset `p` in this set should have the following conditions:

1. `p` is greater than `k`,
2. `p` is less than $\text{OFFSET}(\text{STUDENT})$,
3. `TAB(p)` should not be disjoint with `STUDENT` and `VISITOR`,
4. `p` should exist in at least one of `TT(STUDENT-C)` or `TT(VISITOR-C)`.

The minimum `p` in possible-`k`, is the new `k` which is returned by `FIND-K(STUDENT, VISITOR, 0)`. Since $\text{OFFSET}(\text{STUDENT}) = 3$ and $\text{OFFSET}(\text{VISITOR}) = 4$, possible-`k` is the set (1, 2), where `TAB(1)` is table `EMPLOYEE` and `TAB(2)` is table `PROFESSOR`. Thus, `FIND-K(STUDENT, VISITOR, 0)` would return `k = 1`, since, the minimum integer in possible-`k` is 1.

Similar to the previous example, a hypothetical maybe-yes list is computed for each

STUDENT and VISITOR, by considering the offsets in possible-k, as shown in the following:

$$\begin{aligned}
&\text{STUDENT : } \{\text{maybe}, \quad \text{maybe}\} \\
&\text{PRO} = \quad \{\text{EMPLOYEE}, \text{PROFESSOR}, \text{STUDENT}, \text{VISITOR}, \text{CANADIAN}\} \\
&\text{VISITOR : } \{\text{maybe}, \quad \text{yes}\}
\end{aligned} \tag{3.26}$$

For each table T in PRO, such that T is not disjoint with STUDENT and $\text{OFFSET}(T) < \text{OFFSET}(\text{STUDENT})$, if there exists a translation table T -STUDENT, string “maybe” is added to the STUDENT’s maybe-yes list at the position of $\text{OFFSET}(T)$. If STUDENT has a preference clause, containing $\text{OFFSET}(T)$, string “yes” is added to the STUDENT’s maybe-yes list, at the position of $\text{OFFSET}(T)$, and if T is disjoint with STUDENT, string “no” is added at the position of $\text{OFFSET}(T)$. VISITOR’s maybe-yes list is then filled out similarly.

Firstly, we explain how STUDENT’s maybe-yes list is filled out:

Consider offset 1 in possible-k, EMPLOYEE entities who are also STUDENT entities can be achieved through the translation table EMPLOYEE-STUDENT-C. Therefore, an string “maybe” is added to the STUDENT’s maybe-yes list. Consider offset 2 in possible-k, PROFESSOR entities who are also STUDENT entities can be achieved through the translation table PROFESSOR-STUDENT-C, so another string “maybe” is added to the STUDENT’s maybe-yes list, as indicated in (3.26).

Secondly, we explain how VISITOR’s maybe-yes list is filled out:

Consider offset 1 in possible-k, EMPLOYEE entities who are also VISITOR entities can be achieved through the translation table EMPLOYEE-VISITOR-C. Therefore, an string “maybe” is added to the VISITOR’s maybe-yes list. Consider offset 2 in possible-k, there does not exist any translation table between PROFESSOR and VISITOR, since VISITOR contains $\text{PREF}(\text{PROFESSOR})$. Therefore, an string “yes” is added to the VISITOR’s maybe-yes list, as shown in (3.26).

Following that $\text{FIND-K}(\text{STUDENT}, \text{VISITOR}, 0)$ returns $k = 1$, $\text{COMPILECOND}(s, \text{self}, \text{STUDENT}, v, \text{self}, \text{VISITOR}, ())$ calls function $\text{MAYBEYESCHECK}(1, s, \text{self}, \text{STUDENT}, v, \text{self}, \text{VISITOR}, ())$ to compute the second part of E, containing the “or” clause, in the format (3.15), via the idea of a maybe-yes list for each STUDENT and VISITOR. This function

first checks for the relation of **EMPLOYEE** with both **STUDENT** and **VISITOR** in three cases :

1. The yes-maybe case: **EMPLOYEE** exists in **STUDENT**'s preference clause, and there exists a translation table between **EMPLOYEE** and **VISITOR**.
2. The maybe-yes case: There exists a translation table between **EMPLOYEE** and **STUDENT**, and **EMPLOYEE** exists in **VISITOR**'s preference clause.
3. The maybe-maybe case: There exist translation tables between **EMPLOYEE** and **STUDENT**, and between **EMPLOYEE** and **VISITOR**.

Clearly, the third case is true, since there exist translation tables **EMPLOYEE-STUDENT-C** and **EMPLOYEE-VISITOR-C**, and the other two cases are false. Thus, **MAYBEYESCHECK**(1, s, self, **STUDENT**, v, self, **VISITOR**, ()) calls function **NO_PREFJOIN**(e, self, **EMPLOYEE**, s, self, **STUDENT**, ()) to find the common entities between **EMPLOYEE-C** and **STUDENT-C** and function **NO_PREFJOIN**(e, self, **EMPLOYEE**, v, self, **VISITOR**, ("EMPLOYEE", "EMPLOYEE-STUDENT-C")) to find the common entities between **EMPLOYEE-C** and **VISITOR-C**. The former returns a **WHERE-TABS1** and a **WHERE-EQS1** that are part of an SQL query in the format (3.2). Since **EMPLOYEE** does not already exist in **Org-TABS** or **E-WHERE-TABS**, it is added to the following **WHERE-TABS1** and **E-WHERE-TABS**. However, **WHERE-EQS1** is still empty, as follows:

$$\begin{aligned}\text{WHERE-TABS1} &:= (\text{EMPLOYEE-C } e) \\ \text{WHERE-EQS1} &:= ()\end{aligned}$$

Then, **NO_PREFJOIN**(e, self, **EMPLOYEE**, s, self, **STUDENT**, ()) calls function **TRANSLATIONTCHECK**(e, self, **EMPLOYEE**, s, self, **STUDENT**, (**EMPLOYEE**)) to check for the type of **EMPLOYEE-STUDENT-C** and find the common attributes between **EMPLOYEE-C** and **STUDENT-C** accordingly. Since **EMPLOYEE-STUDENT-C** is neither absorbed or can be computed from the join of other tables, it is added to the following **TTWHERE-TABS1**, and the common attributes between **EMPLOYEE-C** and **STUDENT-C** are stored in the following

TTWHERE-EQS1:

```
TTWHERE-TABS1 := (EMPLOYEE-STUDENT-C z)
TTWHERE-EQS1 := ((PREFCHECK(e.self(EMPLOYEE), z.self(EMPLOYEE))) and
(PREFCHECK(z.self(STUDENT), s.self(STUDENT))))
```

TRANSLATIONTCHECK(e, self, EMPLOYEE, s, self, STUDENT, (EMPLOYEE)) returns TTWHERE-EQS1 and TTWHERE-TABS1 as part of an SQL query in the format (3.2), and computation in TTWHERE-EQS1 is simplified as follows:

$$\begin{aligned} \text{TTWHERE-TABS1} &= (\text{EMPLOYEE-STUDENT-C } z) \\ \text{TTWHERE-EQS1} &= (e.\text{enum} = z.\text{enum} \text{ and } z.\text{snum} = s.\text{snum}) \end{aligned} \quad (3.27)$$

NO_PREFJOIN(e, self, EMPLOYEE, s, self, STUDENT, ()) would then concatenate WHERE-TABS1 with TTWHERE-TABS1 and concatenate WHERE-EQS1 with TTWHERE-EQS1, as follows:

$$\begin{aligned} \text{WHERE-TABS1} &= (\text{EMPLOYEE-C } e, \text{ EMPLOYEE-STUDENT-C } z) \\ \text{WHERE-EQS1} &= (e.\text{enum} = z.\text{enum} \text{ and } z.\text{snum} = s.\text{snum}) \end{aligned} \quad (3.28)$$

Since TTWHERE-TABS1 in (3.27), will be included in an “exists” clause of an SQL query in the form (3.2), it is added to E-WHERE-TABS. Function NO_PREFJOIN(e, self, EMPLOYEE, v, self, VISITOR, (“EMPLOYEE”, “EMPLOYEE-STUDENT-C”))) returns WHERE-TABS2 and WHERE-EQS2 that are part of an SQL query in the format (3.2). Since both EMPLOYEE and VISITOR already exist in Org-TABS and E-WHERE-TABS, they are not added to WHERE-TABS. Then, this function calls function TRANSLATIONTCHECK(e, self, EMPLOYEE, v, self, VISITOR, (“EMPLOYEE”, “EMPLOYEE-STUDENT-C”)) to check for the type of EMPLOYEE-STUDENT-C and find the common attributes between EMPLOYEE-C and VISITOR-C accordingly. Since EMPLOYEE-VISITOR-C is neither absorbed or can be replaced by other tables, it is added to the following TTWHERE-TABS2, and common attributes between EMPLOYEE-C and VISITOR-C are stored in the following TTWHERE-EQS2:

$$\text{TTWHERE-TABS2} := (\text{EMPLOYEE-VISITOR-C } r)$$

$$\text{TTWHERE-EQS2} := (\text{PREFCHECK}(e.\text{self}(\text{EMPLOYEE}), r.\text{self}(\text{EMPLOYEE}))), \text{ and,}$$

$$(\text{PREFCHECK}(r.\text{self}(\text{VISITOR}), v.\text{self}(\text{VISITOR})))$$

$\text{TRANSLATIONTCHECK}(e, \text{self}, \text{EMPLOYEE}, v, \text{self}, \text{VISITOR}, (\text{"EMPLOYEE"}, \text{"EMPLOYEE-STUDENT-C"}))$ returns TTWHERE-TABS2 and TTWHERE-EQS2 that are part of an SQL query in the format (3.2) and computation in TTWHERE-EQS2 is simplified to the following:

$$\text{TTWHERE-TABS2} := (\text{EMPLOYEE-VISITOR-C } r)$$

$$\text{TTWHERE-EQS2} := ((e.\text{enum} = r.\text{enum}) \text{ and } (r.\text{disc} = v.\text{disc} \text{ and } r.f = v.f))$$
(3.29)

$\text{NOPREFJOIN}(e, \text{self}, \text{EMPLOYEE}, v, \text{self}, \text{VISITOR}, (\text{"EMPLOYEE"}, \text{"EMPLOYEE-STUDENT-C"}))$ then returns WHERE-TABS2 and WHERE-EQS2 which are the same as TTWHERE-TABS2 and TTWHERE-EQS2 . Thereafter, $\text{MAYBEYESCHECK}(1, s, \text{self}, \text{STUDENT}, v, \text{self}, \text{VISITOR}, ())$ concatenates WHERE-TABS1 and WHERE-TABS2 which results to the following WHERE-TABS , and concatenates WHERE-EQS1 and WHERE-EQS2 which results to the following WHERE-EQS :

$$\text{WHERE-TABS} := (\text{EMPLOYEE-C } e, \text{EMPLOYEE-STUDENT-C } z, \text{EMPLOYEE-VISITOR-C } r)$$

$$\text{WHERE-EQS} := ((e.\text{enum} = z.\text{enum}) \text{ and } (z.\text{snum} = s.\text{snum})),$$

$$((e.\text{enum} = r.\text{enum}) \text{ and } (r.\text{disc} = v.\text{disc} \text{ and } r.f = v.f))$$

$\text{MAYBEYESCHECK}(1, s, \text{self}, \text{STUDENT}, v, \text{self}, \text{VISITOR}, ())$ uses theses WHERE-TABS and WHERE-EQS to generate the second part of E which is in the format (3.15), as shown

here:

```

or exists( select * from EMPLOYEE-C e, EMPLOYEE-STUDENT-C z, EMPLOYEE-VISITOR-C r
           where (e.enum = z.enum)
           and (z.snum = s.snum)
           and (e.enum = r.enum)
           and (r.disc = v.disc and r.f = v.f))

```

(3.30)

The SQLA in Figure 3.15a is now converted to the intermediate SQL in Figure 3.15c. Thereafter, MAYBEYESCHECK(1, s, self, STUDENT, v, self, VISITOR, ()) calls function FIND-K(STUDENT, VISITOR, 1) to find the next k. possible-k is now the set (2), so k = 2 is returned, where TAB(2) is PROFESSOR. The hypothetical maybe-yes list for both STUDENT and VISITOR, in relation to PROFESSOR, has been introduced in (3.26). Using k = 2 as an input, MAYBEYESCHECK(1, s, self, STUDENT, v, self, VISITOR, ()) calls MAYBEYESCHECK(2, s, self, STUDENT, v, self, VISITOR, ()) to compute another “or” clause in the format (3.15), by finding the intersection between PROFESSOR and both STUDENT and VISITOR. Therefore, this function first checks for the relation of PROFESSOR with both STUDENT and VISITOR in three cases :

1. The yes-maybe case: PROFESSOR exists in the STUDENT’s preference clause, and there exists a translation table between PROFESSOR and VISITOR.
2. The maybe-yes case: There exists a translation table between PROFESSOR and STUDENT, and PROFESSOR exists in the VISITOR’s preference clause.
3. The maybe-maybe case: There exist translation tables between PROFESSOR and STUDENT, and between PROFESSOR and VISITOR.

Clearly, the second case is true, since PREF(VISITOR) contains offset 2 and there exists translation table PROFESSOR-STUDENT-C, and the other two cases are false. Therefore, MAYBEYESCHECK(2, s, self, STUDENT, v, self, VISITOR, ()) calls function PREFJOIN(p, self, PROFESSOR, s, self, STUDENT, v, self, VISITOR, ()) to join VISITOR with PROFESSOR

and PROFESSOR-STUDENT-C with both PROFESSOR and STUDENT, and illustrate the result as WHERE-TABS and WHERE-EQS for an SQL query in the format (3.2). The following WHERE-EQS illustrates common attributes between VISITOR-C and PROFESSOR-C. Since PROFESSOR does not exist in either Org-Tabs or E-WHERE-TABS, it is added to the following WHERE-TABS:

$$\begin{aligned}\text{WHERE-EQS} &:= (\text{PREFCHECK}(v.\text{self}(\text{VISITOR}), p.\text{self}(\text{PROFESSOR}))) \\ \text{WHERE-TABS} &:= (\text{PROFESSOR-C } p)\end{aligned}$$

The computation in the WHERE-EQS is simplified as follows:

$$\begin{aligned}\text{WHERE-EQS} &:= (v.\text{disc} = p.2 \text{ and } v.f = \text{Concat}(p.\text{name}, p.\text{office})) \\ \text{WHERE-TABS} &:= (\text{PROFESSOR-C } p)\end{aligned}\tag{3.31}$$

Thereafter, E-WHERE-TABS is updated to be (PROFESSOR), and PREFJOIN(p, self, PROFESSOR, s, self, STUDENT, v, self, VISITOR, ()) calls function TRANSLATIONTCHECK(p, self, PROFESSOR, s, self, STUDENT, (PROFESSOR)) to check for the type of PROFESSOR-STUDENT-C which can be replaced as follows:

$$\text{REPLACE}(\text{PROFESSOR-STUDENT-C}) = (\text{EMPLOYEE-PROFESSOR-C}, \text{EMPLOYEE-STUDENT-C})$$

TRANSLATIONTCHECK(p, self, PROFESSOR, s, self, STUDENT, (PROFESSOR)) then calls function NESTEDREPLACE(EMPLOYEE-PROFESSOR-C, EMPLOYEE-STUDENT-C, (PROFESSOR)) to find the common attributes between EMPLOYEE-C and PROFESSOR-C, via NOPREFJOIN(e, self, EMPLOYEE, p, self, PROFESSOR, (PROFESSOR)), and common attributes between EMPLOYEE-C and STUDENT-C, via NOPREFJOIN(e, self, EMPLOYEE, s, self, STUDENT, (PROFESSOR, EMPLOYEE)).

NOPREFJOIN(e, self, EMPLOYEE, p, self, PROFESSOR, (PROFESSOR)) returns WHERE-TABS1 and WHERE-EQS1 for an SQL query in the format (3.2). Since EMPLOYEE does not already exist in E-WHERE-TABS or Org-TABS, it is added to both E-WHERE-TABS

and WHERE-TABS1 as follows, and WHERE-EQS1 is still empty, as shown here:

$$\begin{aligned}\text{WHERE-TABS1} &:= (\text{EMPLOYEE-C } e) \\ \text{WHERE-EQS1} &:= ()\end{aligned}\tag{3.32}$$

E-WHERE-TABS is then updated to be the set (PROFESSOR, EMPLOYEE).

NO_PREFJOIN(*e*, *self*, EMPLOYEE, *p*, *self*, PROFESSOR, (PROFESSOR)) then calls function TRANSLATIONTCHECK(*e*, *self*, EMPLOYEE, *p*, *self*, PROFESSOR, (PROFESSOR, EMPLOYEE)) to check for the type of EMPLOYEE-PROFESSOR-C and find common attributes between EMPLOYEE-C and PROFESSOR-C accordingly. Since EMPLOYEE-PROFESSOR-C is absorbed on PROFESSOR and PROFESSOR already exists in E-WHERE-TABS, this function returns empty TTWHERE-TABS1 and the following TTWHERE-EQS1 to illustrate common attributes between EMPLOYEE-C and PROFESSOR-C:

$$\begin{aligned}\text{TTWHERE-TABS1} &:= () \\ \text{TTWHERE-EQS1} &:= (\text{PREFCHECK}(p.\text{self}(\text{EMPLOYEE}), e.\text{self}(\text{EMPLOYEE})))\end{aligned}$$

Computation in TTWHERE-EQS1 is simplified as follows:

$$\begin{aligned}\text{TTWHERE-TABS1} &:= () \\ \text{TTWHERE-EQS1} &:= (p.\text{enum} = e.\text{enum})\end{aligned}\tag{3.33}$$

TTWHERE-TABS1 and TTWHERE-EQS1 are then concatenated to WHERE-TABS1 and WHERE-EQS1 respectively, and NO_PREFJOIN(*e*, *self*, EMPLOYEE, *p*, *self*, PROFESSOR, (PROFESSOR)) returns the following WHERE-TABS1 and WHERE-EQS1:

$$\begin{aligned}\text{WHERE-TABS1} &:= (\text{EMPLOYEE-C } e) \\ \text{WHERE-EQS1} &:= (p.\text{enum} = e.\text{enum})\end{aligned}\tag{3.34}$$

Then NESTEDREPLACE(EMPLOYEE-PROFESSOR-C, EMPLOYEE-STUDENT-C, (PROFESSOR)) calls NO_PREFJOIN(*e*, *self*, EMPLOYEE, *s*, *self*, STUDENT, (PROFESSOR, EMPLOYEE)) which returns WHERE-TABS2 and WHERE-EQS2 for an SQL query in the format (3.2). This

function calls function `TRANSLATIONTCHECK(e, self, EMPLOYEE, s, self, STUDENT, (PROFESSOR, EMPLOYEE))` to check for the type of `EMPLOYEE-STUDENT-C` and find common attributes between `EMPLOYEE-C` and `STUDENT-C` accordingly. Common attributes between `EMPLOYEE-C` and `STUDENT-C` are stored in the following `TTWHERE-EQS2`. Since `EMPLOYEE-STUDENT-C` is neither absorbed or can be replaced by other tables, it is stored in the following `TTWHERE-TABS2`:

$$\begin{aligned} \text{TTWHERE-EQS2} &:= (\text{PREFCHECK}(e.\text{self}(\text{EMPLOYEE}), z.\text{self}(\text{EMPLOYEE}))) \\ &\quad \text{and } (\text{PREFCHECK}(z.\text{self}(\text{STUDENT}), s.\text{self}(\text{STUDENT}))) \\ \text{TTWHERE-TABS2} &:= (\text{EMPLOYEE-STUDENT-C } z) \end{aligned}$$

Computation in `TTWHERE-EQS2` is simplified to the following:

$$\begin{aligned} \text{TTWHERE-EQS2} &:= ((e.\text{enum} = z.\text{enum}) \text{ and } (z.\text{snum} = s.\text{snum})) \\ \text{TTWHERE-TABS2} &:= (\text{EMPLOYEE-STUDENT-C } z) \end{aligned} \tag{3.35}$$

`TTWHERE-TABS2` and `TTWHERE-EQS2` are then concatenated to `WHERE-TABS2` and `WHERE-EQS2` respectively, and `NO_PREFJOIN(e, self, EMPLOYEE, s, self, STUDENT, (PROFESSOR, EMPLOYEE))` returns `WHERE-TABS2` and `WHERE-EQS2` which are the same as `TTWHERE-TABS2` and `TTWHERE-EQS2` respectively.

Afterwards, `NESTEDREPLACE(EMPLOYEE-PROFESSOR-C, EMPLOYEE-STUDENT-C, (PROFESSOR))` returns `NWHERE-TABS` and `NWHERE-EQS` for an SQL query in the format (3.2). `NWHERE-TABS` is computed by concatenating the `WHERE-TABS1` in (3.34) and `TTWHERE-TABS2` in (3.35), and `NWHERE-EQS` is computed by concatenating `WHERE-EQS1` in (3.34) and `TTWHERE-EQS2` in (3.35), as illustrated:

$$\begin{aligned} \text{NWHERE-TABS} &:= (\text{EMPLOYEE-C } e, \text{ EMPLOYEE-C-STUDENT-C } z) \\ \text{NWHERE-EQS} &:= ((p.\text{enum} = e.\text{enum}) \\ &\quad \text{and } (e.\text{enum} = z.\text{enum}) \text{ and } (z.\text{snum} = s.\text{snum})) \end{aligned} \tag{3.36}$$

`TRANSLATIONTCHECK(p, self, PROFESSOR, s, self, STUDENT, (PROFESSOR))` returns the

same WHERE-TABS and WHERE-EQS as NWHERE-TABS and NWHERE-EQS respectively. `PREFJOIN(p, self, PROFESSOR, s, self, STUDENT, v, self, VISITOR, ())` also returns WHERE-TABS which is computed by concatenating WHERE-TABS in (3.31) and (3.36), and WHERE-EQS is computed by concatenating WHERE-EQS in (3.31) and (3.36), as illustrated:

$$\begin{aligned}
\text{WHERE-TABS} &:= (\text{PROFESSOR-C } p, \text{ EMPLOYEE-C } e, \text{ EMPLOYEE-STUDENT-C } z) \\
\text{WHERE-EQS} &:= ((v.\text{disc} = p.2 \text{ and } v.\text{f} = \text{Concat}(p.\text{name}, p.\text{office})) \\
&\quad \text{and } (p.\text{enum} = e.\text{enum}) \\
&\quad \text{and } (e.\text{enum} = z.\text{enum}) \text{ and } (z.\text{snum} = s.\text{snum}))
\end{aligned} \tag{3.37}$$

`MAYBEYESCHECK(2, s, self, STUDENT, v, self, VISITOR, ())` uses above WHERE-TABS and WHERE-EQS to compute the second “or” clause of E in the format (3.15), as indicated here:

$$\begin{aligned}
&\text{or exists(select * from PROFESSOR-C } p, \text{ EMPLOYEE-C } e, \text{ EMPLOYEE-STUDENT-C } z \\
&\quad \text{where } (v.\text{disc} = p.2 \text{ and } v.\text{f} = \text{Concat}(p.\text{name}, p.\text{office})) \\
&\quad \text{and } (p.\text{enum} = e.\text{enum}) \\
&\quad \text{and } (e.\text{enum} = z.\text{enum}) \\
&\quad \text{and } (z.\text{snum} = s.\text{snum})
\end{aligned} \tag{3.38}$$

Thereafter, this function calls `FIND-K(STUDENT, VISITOR, 2)` which returns $k = 0$.

`MAYBEYESCHECK(1, s, self, STUDENT, v, self, VISITOR, ())` concatenates the first “or” clause of E in (3.30) with the second in (3.38) and returns the following “or” clauses:

```

or exists( select * from EMPLOYEE-C e, EMPLOYEE-STUDENT-C z, EMPLOYEE-VISITOR-C r
          where (e.enum = z.enum)
          and (z.snum = s.snum)
          where (e.enum = r.enum)
          and(r.disc = v.disc and r.f = v.f))
or exists( select * from PROFESSOR-C p, EMPLOYEE-C e, EMPLOYEE-STUDENT-C z
          where (v.disc = p.2 and v.f = Concat(p.name, p.office))
          and (p.enum = e.enum)
          and (e.enum = z.enum)
          and (z.snum = s.snum)

```

(3.39)

COMPILECOND(s, self, STUDENT, v, self, VISITOR, ()) concatenates the first part of E in (3.25) with the “or” clauses in (3.39) and returns the following string:

```

s.3 = v.disc and s.snum = v.f
or exists( select * from EMPLOYEE-C e, EMPLOYEE-STUDENT-C z, EMPLOYEE-VISITOR-C r
          where (e.enum = z.enum)
          and (z.snum = s.snum)
          where (e.enum = r.enum)
          and(r.disc = v.disc and r.f = v.f))
or exists( select * from PROFESSOR-C p, EMPLOYEE-C e, EMPLOYEE-STUDENT-C z
          where (v.disc = p.2 and v.f = Concat(p.name, p.office))
          and (p.enum = e.enum)
          and (e.enum = z.enum)
          and (z.snum = s.snum)

```

(3.40)

Therefore, SQLATOSQL(Q) replaces “s.self = v.self” with the string in Figure 3.40 and returns the SQL query in Figure 3.15d.

Figure 3.16 illustrates functions that procedure SQLATOSQL(Q) calls to convert the

SQLA query in Figure 3.15a to the SQL query in Figure 3.15d by compiling “s.**self** = v.**self**”. The first column contains a function or a procedure call with its inputs, and the second column contains what the procedure produces or what the function returns. Except the first row, every other row contains a function that is called by a function or a procedure in the previous row.

Algorithm name	Return
SQLAtoSQL(Q)	Figure 3.15d
COMPILECOND(s, self, STUDENT, v, self, VISITOR, ())	(3.40)
PREFCHECK(s.self(STUDENT), v.self(VISITOR))	(3.24)
FIND-K(STUDENT, VISITOR, 0)	k = 1
MAYBEYESCHECK(1, s, self, STUDENT, v, self, VISITOR, ())	(3.39)
NO_PREFJOIN(e, self, EMPLOYEE, s, self, STUDENT, ())	(3.28)
TRANSLATIONTCHECK(e, self, EMPLOYEE, s, self, STUDENT, (EMPLOYEE))	(3.27)
NO_PREFJOIN(e, self, EMPLOYEE, v, self, VISITOR, (EMPLOYEE, EMPLOYEE-STUDENT-C))	(3.29)
TRANSLATIONTCHECK(e, self, EMPLOYEE, v, self, VISITOR, (EMPLOYEE, EMPLOYEE-STUDENT-C))	(3.29)
FIND-K(STUDENT, VISITOR, 1)	k = 2
MAYBEYESCHECK(2, s, self, STUDENT, v, self, VISITOR, ())	(3.38)
PREFJOIN(p, self, PROFESSOR, s, self, STUDENT, v, self, VISITOR, ())	(3.37)
TRANSLATIONTCHECK(p, self, PROFESSOR, s, self, STUDENT, (PROFESSOR))	(3.36)
NESTEDREPLACE(EMPLOYEE-PROFESSOR-C, EMPLOYEE-STUDENT, (PROFESSOR))	(3.36)
NO_PREFJOIN(e, self, EMPLOYEE, p, self, PROFESSOR, (PROFESSOR))	(3.34)
TRANSLATIONTCHECK(e, self, EMPLOYEE, p, self, PROFESSOR, (PROFESSOR, Employee))	(3.33)
NO_PREFJOIN(e, self, EMPLOYEE, s, self, STUDENT, (PROFESSOR, EMPLOYEE))	(3.35)
TRANSLATIONTCHECK(e, self, EMPLOYEE, s, self, STUDENT, (PROFESSOR, EMPLOYEE))	(3.35)
FIND-K(STUDENT, VISITOR, 2)	k = 0

Figure 3.16: Procedure and function calls with their result for Section 3.2 Example 3.

Chapter 4

Conclusion

Our first contribution relates to how identity resolution is accomplished in Borgida et al.[\[1\]](#) via so-called referring expression types. Primarily, this involved the introduction of a new front-end for programmers to specify such types via SQL’s PRIMARY KEY clause and a new PREFERENCE clause as well as the introduction of new “**disc**” and “**f**” attributes to encode SQL’s primary keys in concrete tables. This has also required revisiting such types, in particular, replacing “**disc**” and “**f**” attributes with their underlying table and attributes names of abstract tables in order to express verifiable conditions that are sufficient to ensure identity resolution.

Our main contribution is to introduce translation tables. Such tables provide an explicit way to convert between referring expressions, and enable a larger space of possible referring expression types that ensure identity resolution. We illustrate situations in which programmer supplied PREFERENCE and PRIMARY KEY clauses would fail to be identity resolving, and adding translation tables would result in the identity resolution. Mapping an ARM schema to the RM schema, considering the generation of translation tables is also well-explained and three different examples are given to clarify this concept. We then illustrate converting SQLA queries to SQL ones on the three mentioned examples, via the use of translation tables.

4.1 Future Study

There are two directions to the future study which we introduce briefly:

1. This thesis takes a top-down approach to mapping ARM schemata to RM schemata, by introducing preference and translation tables on demand. More precisely, when An ARM schema is not identity resolving, due to a missed preference clause, we have added a translation table to keep the schema identity resolving. However, a compliment to this approach is a bottom-up approach in which specific preference and translation tables are already given. The complexity in this approach appears when a translation table is absorbed or it can be replaced by two translation tables. For example, if a translation table $T_i - T_j$ is absorbed on the T_i , only T_i is given with an additional column which stores the $\text{PKC}(T_j)$. Using this column, the intersection between T_i and T_j can be found. If $T_i - T_j$ can be replaced by two translation tables TT_1 and TT_2 , in the RM schema of a bottom-up approach, only TT_1 and TT_2 are given and $T_i - T_j$ does not exist. Thus, for an identity-resolution check, $T_i - T_j$ should be computed via the existence of TT_1 and TT_2 . Otherwise, the identity resolution conditions are not met.
2. The definition of identity resolution in this thesis requires all SQLA queries to map to SQL queries that compute the same result on a generated RM schema. More precisely, a mapping M between `eid` and `string`, has been defined to be *identity resolving*, if for every SQLA query Q and any database DB , the following holds:

$$M(\text{EVAL}(Q, DB)) = \text{EVAL}(Q, M(DB)) \quad (4.1)$$

Another direction for future work would be to relax this “full” or “complete” notion of identity resolution to something, called *partial identity resolution*. In this new approach, M can be partial identity resolving for some SQLA queries. Therefore, not all SQLA queries require to map to their corresponding SQL queries. A possible definition for partial identity resolution can be as follows:

There exists a subset of tables S , where there exists M such that, for all SQLA queries Q only mentioning tables in S and all databases DB , the following holds:

$$M(\text{EVAL}(Q, DB)) = \text{EVAL}(Q, M(DB)). \quad (4.2)$$

Both directions are important in the context of the problem of structured data integration. This is the problem of a data server that aims to provide a client with access to a collection of structured data sources via an integrating schema.

References

- [1] Alexander Borgida, David Toman, and Grant Weddell. On referring expressions in information systems derived from conceptual modelling. In *International Conference on Conceptual Modeling*, pages 183–197. Springer, 2016.
- [2] Alexander Borgida, David Toman, and Grant Weddell. On referring expressions in query answering over first order knowledge bases. In *Fifteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2016.
- [3] EF E, F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *ACM SIGMOD Record*, 38(1):17–36, 2009.
- [4] Ramez Elmasri and Shamkant B Navathe. Fundamentals of database systems. 2016.
- [5] Georg Gottlob, Stanislav Kikot, Roman Kontchakov, Vladimir Podolskii, Thomas Schwentick, and Michael Zakharyashev. The price of query rewriting in ontology-based data access. *Artificial Intelligence*, 213:42–59, 2014.
- [6] Terry Halpin. Modeling of linguistic reference schemes. *International Journal of Information System Modeling and Design (IJISMD)*, 6(4):1–23, 2015.
- [7] Lina Lubyte and Sergio Tessaris. Automatic extraction of ontologies wrapping relational data sources. In *International Conference on Database and Expert Systems Applications*, pages 128–142. Springer, 2009.

- [8] Weicong Ma, C Maria Keet, Wayne Oldford, David Toman, and Grant Weddell. The utility of the abstract relational model and attribute paths in sql. In *European Knowledge Acquisition Workshop*, pages 195–211. Springer, 2018.
- [9] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997.
- [10] Grant Weddell. ROSESEED: Sqlp to sql.

Appendices

Appendix A

Table Declarations

A.1 Chapter 2, Example 2 UNIV table declarations

```
table DEPARTMENT (self eid, deptcode integer, deptname string,  
                  primary key (deptcode))  
                  disjoint with (COURSE, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON))
```

```
table COURSE (self eid, cnum integer, cname string, department eid,  
              primary key (cnum, department),  
              foreign key (department) references DEPARTMENT),  
              disjoint with (DEPARTMENT, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON))
```

```
table CLASS (self eid, course eid, term integer, section integer, professor eid  
             primary key (course, term, section),  
             foreign key (course) references COURSE),  
             foreign key (professor) references PROFESSOR),  
             disjoint with (DEPARTMENT, COURSE, ENROLLMENT, PROFESSOR, STUDENT, PERSON))
```

```

table ENROLLMENT (self eid, student eid, class eid, mark integer,
                  primary key (student, class),
                  foreign key (student) references STUDENT),
                  foreign key (class) references CLASS),
                  disjoint with (DEPARTMENT, COURSE, CLASS, PROFESSOR, STUDENT, PERSON))

table PROFESSOR (self eid, office integer, department eid
                 primary key (name, office),
                 isa(PERSON),
                 foreign key (self) references PERSON,
                 foreign key (department) references DEPARTMENT)
                 disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

table STUDENT (self eid, snum integer, year integer
               primary key (snum),
               preference (PROFESSOR),
               isa(PERSON),
               foreign key (self) references PERSON)
               disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

table PERSON (self eid, sin integer, name string, cellphone integer,
              preference (PROFESSOR, STUDENT),
              covered by (PROFESSOR, STUDENT),
              disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

```

A.2 Chapter 2 RETs

RTA(DEPARTMENT):= DEPARTMENT \rightarrow deptcode = ?

RTA(COURSE):= COURSE \rightarrow (deptcode = ?, cnum = ?)

$\text{RTA}(\text{CLASS}) := \text{CLASS} \rightarrow (\text{deptcode} = ?, \text{cnum} = ?, \text{term} = ?, \text{section} = ?)$
 $\text{RTA}(\text{ENROLLMENT}) := \text{ENROLLMENT} \rightarrow (\text{snum} = ?, \text{deptcode} = ?, \text{cnum} = ?, \text{term} = ?, \text{section} = ?)$
 $\text{RTA}(\text{PROFESSOR}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?)$
 $\text{RTA}(\text{STUDENT}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?$
 $\text{RTA}(\text{PERSON}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?); \text{STUDENT} \rightarrow \text{snum} = ?$

A.3 Chapter 2, Example 2 concrete table declarations

```
table DEPARTMENT-C (deptcode integer, deptname string,
                    primary key (deptcode))
```

```
table COURSE-C (cnum integer, cname string, department-deptcode integer,
               primary key (cnum, department-deptcode),
               foreign key (department-deptcode) references DEPARTMENT)
```

```
table CLASS-C (course-department-deptcode integer, course-cnum integer, term integer,
              section integer, professor-name string, professor-office integer,
              primary key (deptcode, cnum, term, section),
              foreign key (name, office) references PROFESSOR)
```

```
table ENROLLMENT-C (student-disc integer, student-f string, class-course-department-
                  deptcode integer, class-course-cnum integer, class-term integer,
                  class-section integer,
                  primary key (student-disc, student-f, class-course-department-
                  deptcode, class-course-cnum, class-term, class-section),
                  foreign key (class-course-department-deptcode, class-course-cnum,
                  class-term, class-section) references CLASS,
                  foreign key (student-disc, student-f) references STUDENT)
```



```

table PROFESSOR-C (name string, office integer, department-deptcode integer,
                    primary key (name, office),
                    foreign key (department-deptcode) references DEPARTMENT)
table STUDENT-C (disc integer, f string, snum integer, year integer
                 primary key (disc, f))
table PERSON-C (disc integer, f string, sin integer, name string, cellphone integer
                primary key (disc, f))

```

A.4 Chapter 3, Example 1 abstract table declarations

```

table DEPARTMENT (self eid, deptcode integer, deptname string,
                  primary key (deptcode))
                  disjoint with (COURSE, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON))

table COURSE (self eid, cnum integer, cname string, department eid,
              primary key (cnum, department),
              foreign key (department) references DEPARTMENT),
              disjoint with (DEPARTMENT, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON))

table CLASS (self eid, course eid, term integer, section integer, professor eid
             name string, office integer,
             primary key (course, term, section),
             foreign key (course) references COURSE),
             foreign key (professor) references PROFESSOR),
             disjoint with (DEPARTMENT, COURSE, ENROLLMENT, PROFESSOR, STUDENT, PERSON))

```

```

table ENROLLMENT (self eid, student eid, class eid, mark integer,
                  primary key (student, class),
                  foreign key (student) references STUDENT),
                  foreign key (class) references CLASS),
                  disjoint with (DEPARTMENT, COURSE, CLASS, PROFESSOR, STUDENT, PERSON))

table PROFESSOR (self eid, office integer, department eid,
                 primary key(name, office),
                 isa (PERSON),
                 foreign key (self) references PERSON,
                 foreign key (department) references DEPARTMENT
                 disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

table STUDENT (self eid, snum integer, year integer,
               primary key(snum),
               isa (PERSON),
               foreign key (self) references PERSON
               disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

table PERSON (self eid, sin integer, name string, cellphone integer,
              primary key (sin),
              disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

```

A.5 Chapter 3, Example 1 RETs

RTA(DEPARTMENT):= DEPARTMENT \rightarrow deptcode = ?

RTA(COURSE):= COURSE \rightarrow (deptcode = ?, cnum = ?)

RTA(CLASS):= CLASS \rightarrow (deptcode = ?, cnum = ?, term = ?, section = ?)

$\text{RTA}(\text{ENROLLMENT}) := \text{ENROLLMENT} \rightarrow (\text{snum} = ?, \text{deptcode} = ?, \text{cnum} = ?, \text{term} = ?, \\ \text{section} = ?)$
 $\text{RTA}(\text{PROFESSOR}) := \text{PROFESSOR} \rightarrow (\text{name} = ?, \text{office} = ?)$
 $\text{RTA}(\text{STUDENT}) := \text{STUDENT} \rightarrow \text{snum} = ?$
 $\text{RTA}(\text{PERSON}) := \text{PERSON} \rightarrow \text{sin} = ?$

A.6 Chapter 3, Example 1 concrete table declarations

```

table DEPARTMENT-C (deptcode integer, deptname string,
                    primary key (deptcode))

table COURSE-C (cnum integer, cname string, deptcode integer,
               primary key (cnum, deptcode),
               foreign key (deptcode) references DEPARTMENT)

table CLASS-C (deptcode integer, cnum integer, term integer, section integer,
              name string, office integer,
              primary key (deptcode, cnum, term, section),
              foreign key (name, office) references PROFESSOR)

table ENROLLMENT-C (snum integer, deptcode integer, cnum integer,
                   term integer, section integer,
                   primary key (snum, deptcode, cnum, term, section),
                   foreign key (deptcode, cnum, term, section) references CLASS,
                   foreign key (snum) references STUDENT)

table PROFESSOR-C (office integer, deptcode integer, sin integer,
                  primary key (name, office),
                  foreign key (sin) references PERSON,
                  foreign key (deptcode) references DEPARTMENT-C)

```

```
table STUDENT-C (snum integer, year integer, sin integer
                primary key (snum),
                foreign key (sin) references PERSON-C)
```

```
table PERSON-C (sin integer, name string, cellphone integer,
                primary key (sin))
```

A.7 Chapter 3, Example 1 translation tables:

```
table PERSON-STUDENT-C (id integer, snum integer,
                        primary key (id),
                        foreign key (id) references PERSON-C,
                        foreign key (snum) references STUDENT-C)
```

```
table PERSON-PROFESSOR-C (id integer, name string, office integer,
                           primary key (id),
                           foreign key (id) references PERSON-C,
                           foreign key (name, office) references PROFESSOR-C)
```

```
table STUDENT-PROFESSOR-C (snum integer, name string, office integer
                           primary key (id),
                           foreign key (id) references PERSON-C,
                           foreign key (snum) references STUDENT-C)
```

A.8 Chapter 3, Example 2 abstract table declarations

```
table DEPARTMENT (self eid, deptcode integer, deptname string,  
                  primary key (deptcode))  
                  disjoint with (COURSE, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON))  
  
table COURSE (self eid, cnum integer, cname string, department eid,  
              primary key (cnum, department),  
              foreign key (department) references DEPARTMENT),  
              disjoint with (DEPARTMENT, CLASS, ENROLLMENT, PROFESSOR, STUDENT, PERSON))  
  
table CLASS (self eid, department eid, cnum integer, term integer, section integer,  
             name string, office integer, primary key (department, cnum, term, section),  
             foreign key (name, office) references PROFESSOR),  
             disjoint with (DEPARTMENT, COURSE, ENROLLMENT, PROFESSOR, STUDENT, PERSON))  
  
table ENROLLMENT (self eid, snum integer, department eid, cnum integer,  
                  term integer, section integer,  
                  primary key (snum, department, cnum, term, section),  
                  foreign key (department, cnum, term, section) references CLASS),  
                  foreign key (snum) references STUDENT),  
                  disjoint with (DEPARTMENT, COURSE, CLASS, PROFESSOR, STUDENT, PERSON))  
  
table PROFESSOR (self eid, office integer, department eid  
                 primary key (name, office),  
                 isa (PERSON),  
                 foreign key (self) references PERSON,  
                 foreign key (department) references DEPARTMENT  
                 disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))
```

```

table STUDENT (self eid, snum integer, year integer,
               primary key (snum),
               isa (PERSON),
               foreign key (self) references PERSON
               disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

```

```

table PERSON (self eid, sin integer, name string, cellphone integer
              primary key (sin),
              preference (PROFESSOR, STUDENT),
              disjoint with (DEPARTMENT, COURSE, CLASS, ENROLLMENT))

```

A.9 Chapter 3, Example 2 RETs

```

RTA(DEPARTMENT):= DEPARTMENT → deptcode = ?
RTA(COURSE):= COURSE → (deptcode = ?, cnum = ?)
RTA(CLASS):= CLASS → (deptcode = ?, cnum = ?, term = ?, section = ?)
RTA(ENROLLMENT):= ENROLLMENT → (snum = ?, deptcode = ?, cnum = ?, term = ?,
                                section = ?)
RTA(PROFESSOR):= PROFESSOR → (name = ?, office = ?)
RTA(STUDENT):= STUDENT → snum = ?
RTA(PERSON):= PROFESSOR → (name = ?, office = ?); STUDENT → snum = ?;
                    PERSON → sin = ?

```

A.10 Chapter3, Example 2 concrete table declarations

```

table DEPARTMENT-C (deptcode integer, deptname string,
                    primary key (deptcode))

```

```

table COURSE-C (cnum integer, cname string, deptcode integer,
                primary key (cnum, deptcode),
                foreign key (deptcode) references DEPARTMENT)

table CLASS-C (deptcode integer, cnum integer, term integer, section integer,
               name string, office integer,
               primary key (deptcode, cnum, term, section),
               foreign key (name, office) references PROFESSOR)

table ENROLLMENT-C (snum integer, deptcode integer, cnum integer,
                   term integer, section integer,
                   primary key (snum, deptcode, cnum, term, section),
                   foreign key (department, cnum, term, section) references CLASS,
                   foreign key (snum) references STUDENT)

table PROFESSOR-C (name string, office integer, deptcode integer,
                  primary key (name, Office),
                  foreign key (deptcode) references DEPARTMENT-C)

table STUDENT-C (snum integer, year integer,
                 primary key (snum))

table PERSON-C (disc integer, f string, id integer, name string, cellphone integer
               primary key (disc, f))

table PROFESSOR-STUDENT-C (name string, office integer, snum integer
                           primary key (name, office),
                           foreign key (name, office) references PROFESSOR-C),
                           foreign key (snum) references STUDENT-C))

```

A.11 Chapter 3 Example 3 abstract table declarations for EMPLOYEE, PROFESSOR, STUDENT, VISITOR and CANADIAN

```
table EMPLOYEE (self eid, enum integer, name string, address string
                primary key (enum))
```

```
table PROFESSOR (self eid, name string, office integer,
                 primary key (name, office),
                 isa (EMPLOYEE))
```

```
table STUDENT (self eid, snum integer, name string, year integer,
               primary key (snum))
```

```
table VISITOR (self eid, vnum integer, name string, address string
               primary key (vnum),
               disjoint from CANADIAN,
               preference (PROFESSOR, STUDENT))
```

```
table CANADIAN (self eid, sin integer, name string, address string
                primary key (sin),
                disjoint from VISITOR))
```


A.12 Chapter 3, Example 3 RETs

```
RTA(EMPLOYEE):= EMPLOYEE → enum = ?  
RTA(PROFESSOR):= PROFESSOR → (name = ?, office = ?)  
RTA(STUDENT):= STUDENT → snum = ?  
RTA(VISITOR):= PROFESSOR → (name = ?, office = ?); STUDENT → snum = ?;  
                VISITOR → visanum = ?  
RTA(CANADIAN):= CANADIAN → sin = ?
```

Figure A.1: $\text{RTA}(T)$ s for each table T in Σ

A.13 Chapter 3 Example 3 concrete table declarations for EMPLOYEE, PROFESSOR, STUDENT, VISITOR and CANADIAN

```
table EMPLOYEE (enum integer, name string, address string,  
                primary key (enum))
```

```
table PROFESSOR (name string, office integer,  
                primary key (name, office))
```

```
table STUDENT (snum integer, name string, year integer,  
               primary key (snum))
```

```
table VISITOR (disc integer, f string, vnum integer, name string, address string,  
              primary key (disc, f))
```

```
table CANADIAN (sin integer, name string, address string,  
               primary key (sin))
```