

Conceptual interpretation of foreign keys

Jean-Luc Hainaut

May 2010¹

Abstract

Foreign keys form a major structuring construct in relational databases and in standard files. In reverse engineering processes, they have long been interpreted as the implementation of *many-to-one* relationship types. Though one could naively think they are useless, or at least unnecessary, in hierarchical and network models, foreign keys also appear very frequently in IMS, CODASYL, TOTAL/IMAGE and even in OO databases. Besides the standard version of foreign key, according to which a set of columns (fields) in a table (file) is used to designate rows (records) in another table, a careful analysis of existing (both modern and legacy) databases puts into light a surprisingly large variety of non standard forms of foreign keys. Most of them are quite correct, and perfectly fitted to the requirements the developer had in mind. However, their conceptual interpretation can prove much more difficult to formalize than the standard forms.

The aim of this study is to classify, to analyze and to interpret some of the most frequent variants of foreign keys that have been observed in operational files and databases.

1. The first version appeared as Chapter 12 of *Introduction to Database Reverse Engineering*, 2002

1 Introduction

The term *foreign key* was introduced to designate a frequent structural pattern observed in relational databases. Considering a relation schema S with candidate (most often primary) key A on the one hand, and a set F of attributes of relation schema R on the other hand, F is a foreign key of R to S if, at any time, for each tuple r from the extension of R , such that $r.F$ is not null, a tuple s exists in the extension of S such that $r.F = s.A$ (Figure 1). This property means that the set of values of F that appears in the extension of R is a part of the set of values of A of the extension of S . A foreign key induces an inclusion constraint the right-hand side of which is the value set of a candidate (primary) key. The foreign key F acts as a *reference* to the tuples of S . Hence the name *referential constraint* commonly given to the inclusion property. Maintaining this constraint ensures the so-called *referential integrity* of the database.



Figure 1: Two representations of the concept of foreign key in the relational model: attribute $R.F$ is a reference to S , a role that is expressed by an inclusion constraint (left) or the tag *ref* (right).

F is called a *foreign key* to suggest that it is a copy of the (primary) key of a foreign relation. Of course, both tables can be the same, in which case the foreign key references tuples of its own table.

The concept of a set of fields used as a reference to records is not limited to the relational model, and can be found in practically all databases. Of course, it is an integral part of all value-based models, that is, models in which all the information is represented by explicit field values, and by aggregates of values. The pure relational model (excluding the object-relational variants) is value-based, but all standard file structure models also are of that kind. To define relationships between records or rows, we generally use reference fields.

Other models include specific constructs to associate data entities. Such is the case of the CODASYL DGTG models, and their numerous commercial avatars (IDMS, IDS, IDS2, UDS, Vax DBMS, etc.) and of IBM IMS model, in which records (or segments) can be linked through association mechanisms the implementation of which (generally pointer-based) can be ignored to a large extent by the programmers. Object-oriented databases also offer structures for associating objects, namely through object-attributes or explicit associations. TOTAL/IMAGE databases make use of a mixed construct, based on explicit associations in one direction, and foreign keys in the other one.

In pure relation schemas, a foreign key will generally be noted by the referential constraint they define (e.g., $R[F] \subseteq S[A]$). In DMS-independent schemas, we will prefer the more neutral notation $R.F \longrightarrow S.A$, or, when there is no ambiguity about the target

identifier, $R.F \longrightarrow S$. This notation should not be confused with that of functional dependency, which is similar.

Basic and derived foreign keys

The origin of a foreign key, say F , as it appears in the schema in concern is an important issue. In some cases, F has been declared in the DDL source program, so that it can be considered a basic property expressing, in most cases, a *many-to-one* relationship type.

In other cases, F has been recovered in the Data Structure Extraction phase through elicitation techniques such as those discussed in Section **XX**. Such foreign keys can be basic or derived (Figure 2). Indeed, a derived foreign key exhibits the same static and dynamic characteristics as basic ones. In particular, data analysis and program analysis² can detect both types of foreign keys.

```

DEPT (D#, NAME)
ACCOUNT (A#, D#, AVAIL)
EXPENSE (E#, AMOUNT, A#, D#)
ACCOUNT[D#] ⊆ DEPT[D#] (FK1)
EXPENSE[A#, D#] ⊆ ACCOUNT[A#, D#] (FK2)
EXPENSE[D#] ⊆ DEPT[D#] (FK3)

```

Figure 2: Basic (FK1, FK2) and derived (FK3) foreign keys. Derived foreign keys must be discarded.

If all the basic foreign keys have been recovered, then identifying derived foreign keys is an easy task, when considering the following inference rules of inclusion constraints:

1. let M and N be two lists of domains such that $N \subseteq M$, and R and S two relation schemas defined on supersets of M

$$R[M] \subseteq S[M] \Rightarrow R[N] \subseteq S[N]$$
2. let J, K, L be subsets of an arbitrary set P ,
$$J \subseteq K \wedge K \subseteq L \Rightarrow J \subseteq L$$

Applying these rules to the schema 2 leads to the expressions,

$$\text{EXP}[A\#, D\#] \subseteq \text{ACT}[A\#, D\#] \Rightarrow \text{EXP}[D\#] \subseteq \text{ACT}[D\#]$$

$$\text{EXP}[D\#] \subseteq \text{AC}[D\#] \wedge \text{AC}[D\#] \subseteq \text{DEPT}[D\#] \Rightarrow \text{EXP}[D\#] \subseteq \text{DEPT}[D\#]$$

... that prove that FK3 is derivable from FK1 and FK2, and therefore can be discarded.

Conversely, some derived foreign keys may happen to be kept in the schema, and erroneously considered basic, if not all their basic foreign keys have been elicited. In this case, the schema is both incomplete and flawed and will rapidly lead to data inconsistencies, should it be used for example to migrate the data to a new database governed by this schema. For instance, due to careless data structure extraction, the schema of Figure 2 could have included the foreign

2. Through such constructs as *by-pass* joins:

```
select * from DEPT D, EXPENSE E where D.D# = E.D#.
```

keys FK1 and FK3 only.

Fortunately, heuristics exist which can locate some of such patterns, and help the analyst rebuild the correct schema (see Section 7.5).

Two puzzling observations

1. One of the most surprising observations in actual *non value-based* databases, is that they most often include foreign keys, despite the availability of explicit constructs to express associations. For instance, many CODASYL databases, and almost all medium to large size IMS databases include hundreds to thousands of foreign keys. As a consequence, studying foreign keys and their interpretation must be considered a major domain of interest in data reverse engineering, whatever the physical model according to which the legacy data are organized.
2. The second, even more disturbing, observation, is that, besides standard foreign keys described here above as the implementation of *many-to-one* relationship types, database developers have used this apparently straightforward concept to code a large variety of sophisticated structures. Though some patterns can be considered truly *perverse*, most of them are clever implementations of complex structures that have never been described in the literature. This proves, if need be, that reverse engineering can provide invaluable, and often original, techniques for database design.

Structure of this chapter

We will first discuss the most frequent standard (Section 2) and non-standard (Section 3) foreign key patterns, as well as inclusion constraints (Section 4) and propose natural and intuitive interpretations. Then, we will address complex and tricky structures (Section 5) that cannot be solved by simple transformations, and temporal foreign keys (6). Finally we will examine some awkward and even incorrect patterns (Section 7). A series of tables will synthesize the main patterns (Section 8), while exercises will conclude the study.

Terminology

This report, just like many of those which cope with the conceptualization process, develops techniques that concern both logical and conceptual constructs. We will in particular use the terms relations, tables, record types, files, segments and entity types on the one hand, and fields, attributes and columns on the other hand. The question is, should we use all these terms, or, on the contrary, should we base the discussion on a unique set of generic terms that are valid whatever the model of the legacy database? For instance, must we define a foreign key as a set of *attributes that reference tuples of a relation* (relational theory), as a set of *columns referencing rows in a table* (RDBMS), as a set of *fields referencing records in a file* (COBOL files), or as a set of *attributes referencing entities of a given type* (GER model)?

Obviously, we should adopt a common vocabulary, in order to make the techniques developed as general as possible and applicable to as many past, current and future models as pos-

sible. Hence the following conventions that will be used throughout this chapter, with some minor exceptions, notably when we deal with specific models, in which case we naturally adopt the proper terminology.

<i>we use the term</i>	<i>to denote, according to the context</i>
entity	tuple, row, record, segment, entity, object, etc.
entity type	relation, table, record type, segment type, entity type, object class (or type), etc.
attribute	attribute, column, field, etc.
identifier	candidate key, unique key, (alternate) record key, etc.
primary identifier	primary key, record key, etc.
foreign key	any kind of attribute set used to reference entities.

2 Standard foreign keys and basic variants

2.1 The standard foreign key

The most common form of foreign key strictly obeys the definition stated in the relational theory. It is made up of a group of one or several mandatory attributes that targets the primary key of a relation. The components of both keys, considered pairwise, are defined on the same domains. Since it has been described in all database introductory textbooks, even the most elementary ones, this form will be called *standard*. Figure 3 (left) shows a typical standard foreign key.

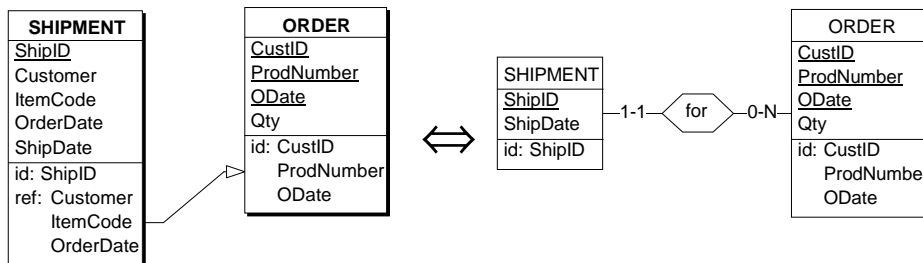


Figure 3: The standard foreign key pattern and its conceptual interpretation.

The transformational interpretation is straightforward (Figure 3, right): the foreign key components are removed, and replaced with a functional³ relationship type. The cardinality of the source role (for.SHIPMENT) is [1-1] and that of the target role (for.ORDER) is [0-N].

From this pattern, we can derive several variants that will be described in this section. They are independent, so that they can occur simultaneously. Their synthesis is presented in Section 8.1.

2.2 Optional foreign key

In its simplest form, such a foreign key comprises one **optional attribute**. It translates in the same way as standard pattern, i.e., into a functional relationship type, except for the source role, which now is optional (cardinality [0-1]), as shown in Figure 4.

3. Let us recall that we call functional any *many-to-one* relationship type, since it expresses a *function* between its roles. As a particular case, *one-to-one* relationship types also are functional. On the contrary, *one-to-many*, *many-to-many*, N-ary relationship types as well as those with attributes are called *non-functional*.

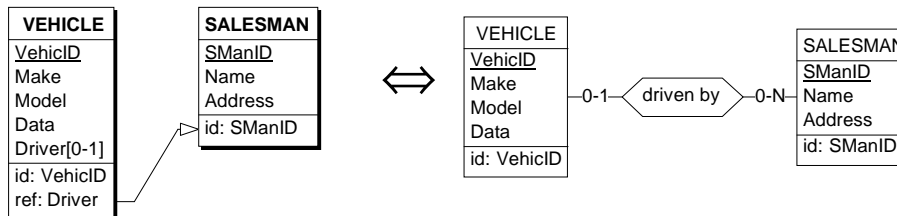


Figure 4: An optional foreign key expresses an optional relationship type.

2.3 Optional multi-component foreign key

If the foreign key comprises several optional attributes, then an additional *coexistence* constraint must hold among these components to make the transformation valid. The interpretation is through an optional relationship type (Figure 5).

Unfortunately, in most cases we encountered, optional multi-component foreign keys were incompletely extracted, in that the coexistence constraints were not identified. Any database that includes entities in which some attributes are valued (i.e., they have *non null* values) while the others are not, cannot be interpreted according to the rule stated above.

If the coexistence constraint cannot be asserted through the Data Structure Extraction phase, then four cases⁴ must be considered, defining the partitioning of STUDENT instances into four subtypes, among which STUD_TY only can be the source of a (mandatory) foreign key.

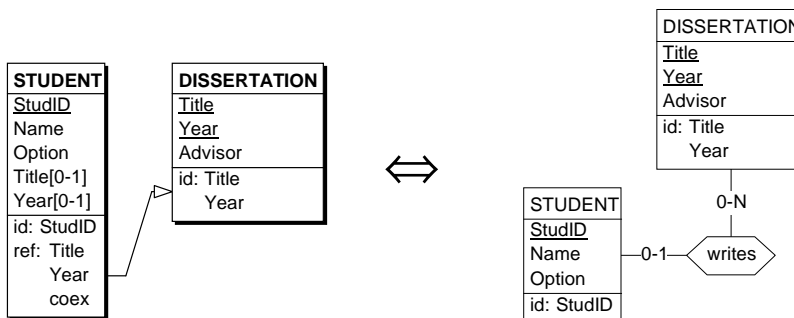


Figure 5: An optional multi-component foreign key should comprise optional components among which a coexistence constraint holds. It translates into an optional relationship type.

- Title = null & Year = null \Rightarrow subset of the STUDENT entities that do not fall in STUD_T nor STUD_Y; this subtype, with

4. In general 2ⁿ, where n is the number of components of the foreign key.

attributes {StdID, Name, Option}, can be left declared;

- Title \neq null & Year = null \Rightarrow subtype STUD_T with mandatory attributes {StdID, Name, Option, Title};
- Title = null & Year \neq null \Rightarrow subtype STUD_Y with mandatory attributes {StdID, Name, Option, Year};
- Title \neq null & Year \neq null \Rightarrow subtype STUD_TY with mandatory attributes {StdID, Name, Option, Title, Year}.

A valid presentation of this partitioning is illustrated in the Figure 6. Interpreting the mandatory foreign key is as usual. The pattern in which some attributes are optional while others are mandatory will be analyzed in Section 7.4.

2.4 Total, or equality, foreign key

Each value of a *total* (or *equality*) foreign key is the primary key value of a target entity **and conversely**. In other words, the primary key of each target entity must match at least one source entity. This foreign key expresses a relationship type whose target role is mandatory (Figure 7, right), that is, its cardinality is [1-J] (i.e., [1-N] or [1-1]).

2.5 Identifying foreign key

An *identifying foreign key* also is an **identifier** of the source entity type. The resulting relationship type is *one-to-one* (Figure 8, right), that is, the cardinality of the target role is [K-1] (i.e., [0-1] or [1-1]).

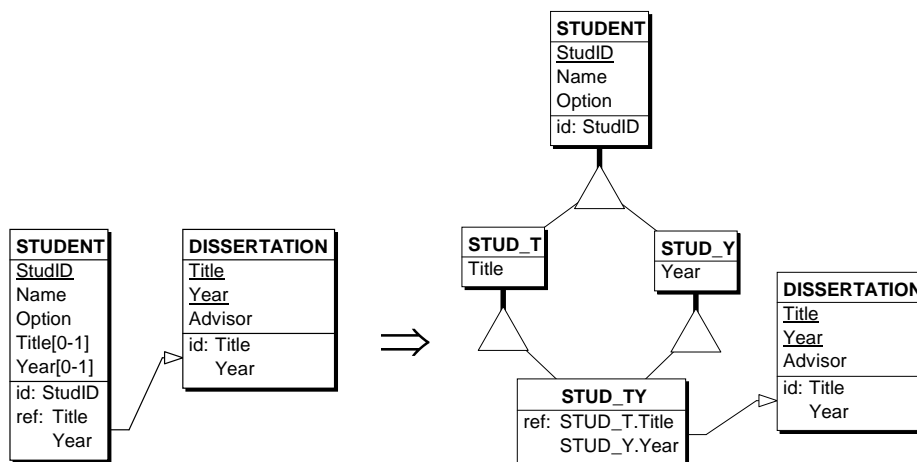


Figure 6: Surprisingly, removing (or forgetting) the coexistence constraint leads to a much more

complex data structure. The right side schema makes all the properties of the left side schema explicit.

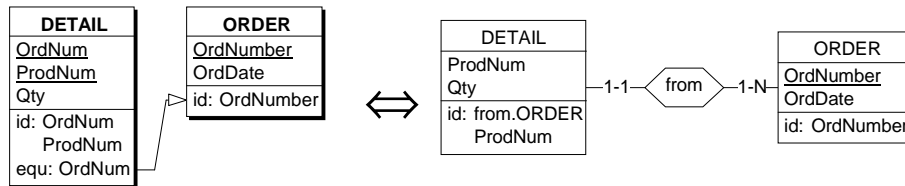


Figure 7: A total, or *equality* foreign key (tagged with keyword *equ* instead of *ref*), translates into a mandatory target role.

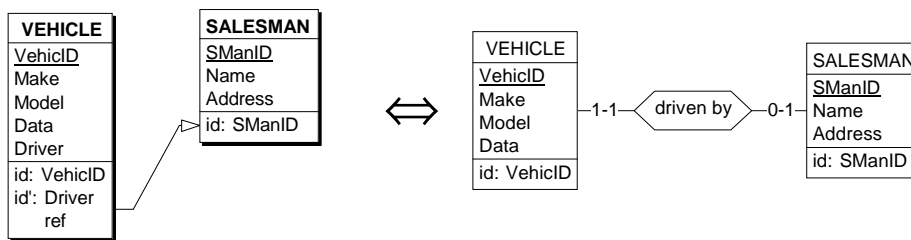


Figure 8: A foreign key which also is an identifier translates into a *one-to-one* relationship type.

An interesting pattern sometimes occurs, in which the foreign key also is the primary (or a secondary) identifier of the source entity type (Figure 9). Some authors propose algorithms in which this pattern is interpreted as an IS-A relation from the source side (subtype) to the target one (supertype). This interpretation obviously is not always valid as witnessed by our example. This point is discussed in Section **XXXXX**.

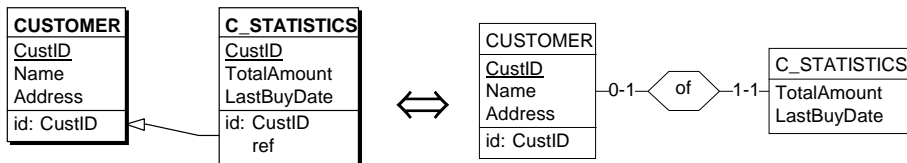


Figure 9: A source entity type whose primary identifier is a foreign key loses this identifier.

2.6 Cyclic foreign key

The source entity type also is the target entity type, leading to a pattern often known as *self-referencing entity type*. The foreign key translates into a cyclic relationship type (Figure 10).

Most such foreign keys are optional. Indeed, let us assume that, in our example, *each product has a substitute*, a fact that would be expressed by a mandatory foreign key. To insert an entity concerning a product, we would have to set the attribute Substitute to the Product code of its substitute, which must already be recorded in the database. Three problems arise: (1) products must be recorded in a specific order so that the above-mentioned constraint is met, (2) a product that has no substitute (yet) cannot be recorded, (3) as a particular case, the first product cannot be recorded.⁵

If a cyclic foreign key happens to be mandatory, then further analyzing it through Data Extraction techniques certainly is worth the effort to check whether it actually is mandatory.

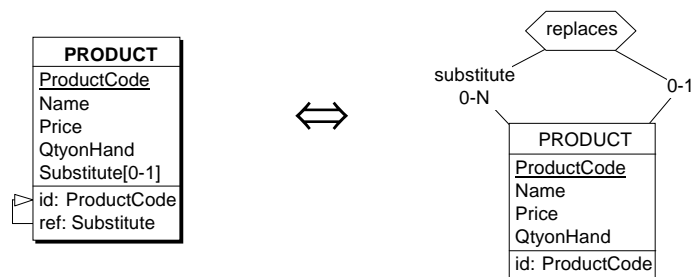


Figure 10: A cyclic foreign key is interpreted as a cyclic relationship type.

5. Unless it is considered its own substitute!

3 Non-standard foreign keys

These patterns are direct extensions of the basic concept, or extensions that accommodate more complex source or target structures. They induce no particularly difficult problems, but must be carefully detected and interpreted.

3.1 Secondary foreign key

The *secondary foreign key* targets a secondary identifier of the target entity type instead of its primary identifier. The interpretation is the same as for standard foreign keys (Figure 11). Though this form is an integral part of the relational theory (that mentions *candidate* target keys only), the practice favors the use of primary keys, hence the qualifier *non-standard*.

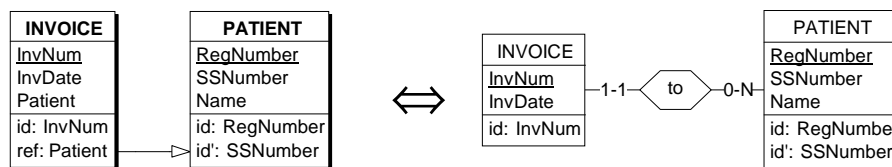


Figure 11: Target secondary identifiers behave just like primary ones.

Unlike primary identifiers, secondary identifiers can be made up of optional components. This poses no particular interpretation problems except for *total foreign key* patterns, that must first be transformed as shown in Figure 12.

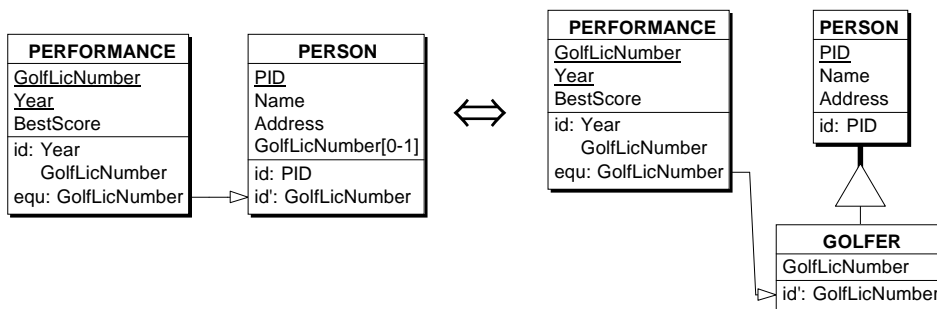


Figure 12: A total (equ) foreign key targeting an optional secondary identifier requires some cleaning before being interpreted correctly.

3.2 Multi-target foreign key

A *multi-target foreign key* references more than one target entity type, so that each source en-

tity simultaneously references an entity of each target type. Since this pattern is equivalent to a series of as many foreign keys as there are target entity types (Figure 13), we will discuss the latter variant. There are two ways to interpret this pattern.

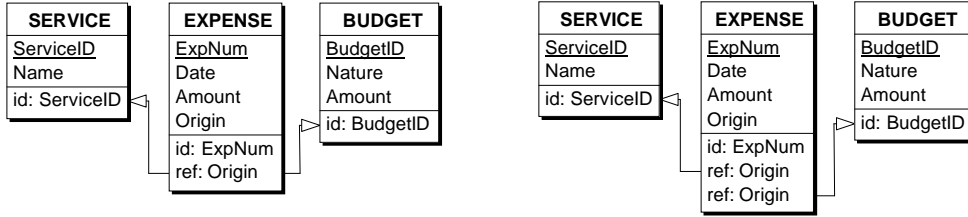


Figure 13: The two variants of multi-target foreign key. An expense has been incurred by a service, and has been charged to a budget. Budgets happen to be identified by the ID of their service. The same foreign key Origin designates both the service that incurs the expense, and the budget which it has been charged to.

The first approach is the most straightforward and consists in defining a relationship type for each foreign key, removing their common components, then defining an integrity constraint that ensures that the values of the target identifiers are the same (Figure 14, left). If the foreign key is optional, then the roles {by.EXPENSE, on.EXPENSE} are optional too and a co-existence constraint must be stated among the roles {by.SERVICE, on.BUDGET}.

The second approach derives from the observation that each foreign key is trivially embedded in the other one (since $\{A3\} \subseteq \{A3\}$). Therefore, the pattern belongs to the *embedded foreign key* family, and can be solved according to the interpretation we will develop in Section 7.5. Hence the schema of Figure 14, right, or its symmetrical version, where EXPENSE is associated with SERVICE instead. It is based on the hypothesis that foreign key EXPENSE.Origin \rightarrow SERVICE is a transitive foreign key deriving from explicit foreign key EXPENSE.Origin \rightarrow BUDGET and implicit foreign key BUDGET.BudgetID \rightarrow SERVICE, still unidentified.

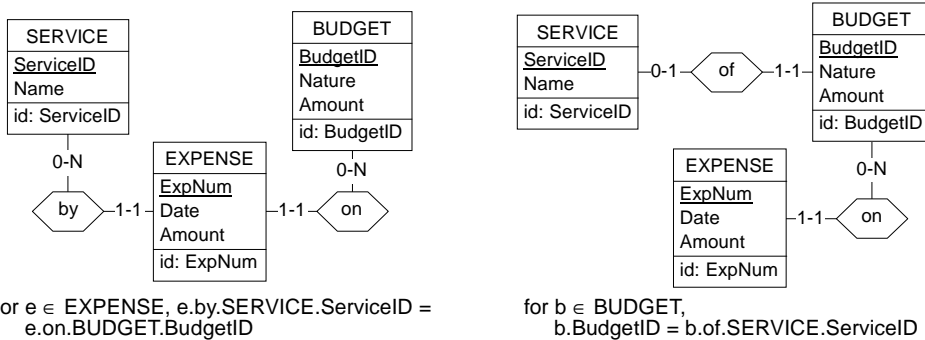


Figure 14: Two interpretations of a multi-target foreign key.

3.3 Alternate foreign key

Each value of an *alternate foreign key* references an entity in one among several target entity types (Figure 15). The entity type that is actually referenced is determined by a definite condition on the source entity, for instance on the value structure of the foreign key.

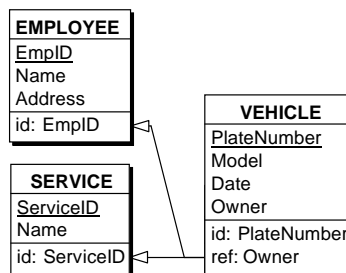


Figure 15: Each value of the foreign key VEHICLE.Owner references either an EMPLOYEE entity or a SERVICE entity.

This pattern can be interpreted as a functional relationship type with a multi-ET role (Figure 16, left) or as an equivalent set of relationship types among which an exclusive constraint holds (Figure 16, right).

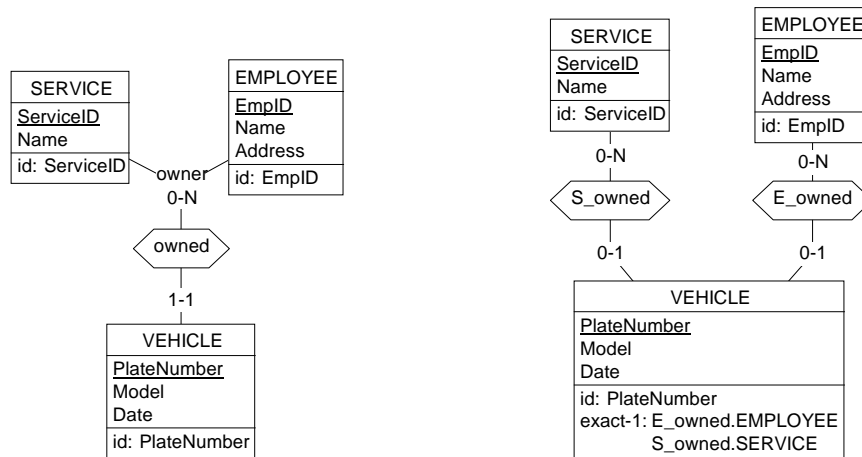


Figure 16: Two equivalent interpretations of an alternate foreign key.

3.4 Hierarchical foreign key to an entity type

Surprisingly, foreign keys are very frequent in hierarchical and network database schemas,

despite the fact that the DBMSs offer explicit constructs to represent relationship types. If the target entity type has been given an absolute identifier comprising attributes only, these foreign keys can be considered standard. However, if the target is identified relatively to one of its parent entity types, the foreign key must reference entities through their hierarchical identifiers (e.g., their *concatenated key* in IMS). Though their detection is delicate, their interpretation is straightforward.

Figure 17 expresses the fact that if two services have the same name, they belong in different departments. So, the department name and the service name of a definite service suffice to uniquely designate it. If expenses are associated with the services that incur them, each EXPENSE entity must include a reference (a foreign key) to one SERVICE entity. This foreign key is made of a department name (DptName) and a service name (ServName). It is interpreted as the relationship type *by*.

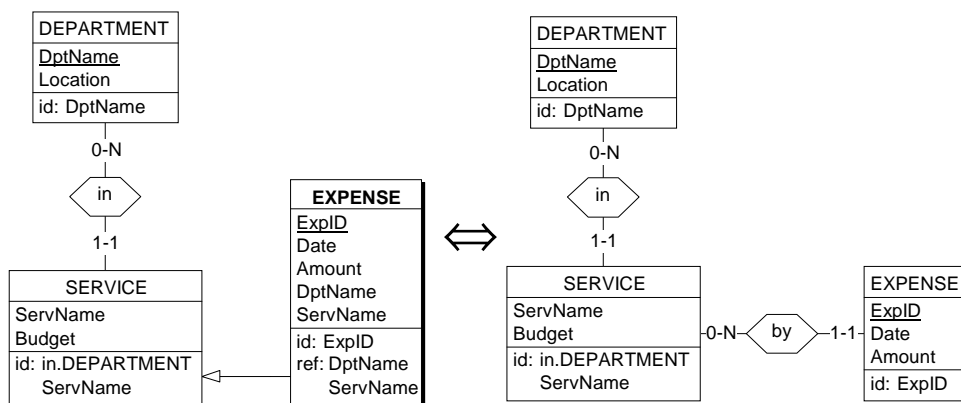


Figure 17: *Left:* each SERVICE entity references a SERVICE through its hierarchical identifier. *Right:* this hierarchical foreign key translates into a relationship type.

3.5 Hierarchical foreign key to a multivalued attribute

Record types as they appear in standard files often compensate the lack of explicit *inter-record* relationships by complex *intra-record* hierarchical field structures. In particular, multivalued compound fields, possibly at several levels, are popular structures to implement a hierarchy of entity types. In such a structure, some dependent entities can be represented by instances of multivalued fields, instead of by individual records. Referencing these entities from within other records consists in designating these values. Hence the concept of foreign keys referencing field values instead of records or rows.

Figure 18 (left) describes a typical example. An ORDER record represents a customer order that includes from 0 to 20 details. Each of these details mention a different Item in a certain quantity. This structure is represented by the ORDER record type which includes the multi-

valued field Detail. This field has distinct ItemCode values (this property is declared through an attribute identifier). To identify a unique Detail value, the programmer must supply a value of OrdID to locate the parent record and a value of ItemCode to identify the right field value. For each detail, some shipments can be made to the customer. Therefore, each shipment is associated with a detail. Each SHIPMENT record designates its parent Detail value through the hierarchical foreign key {OrdID,ItemCode}.

By transforming this multivalued attribute into an entity type, we get the source pattern of *Hierarchical foreign key to an entity type* (Section 3.4). Hence the immediate interpretation of Figure 18 (right).

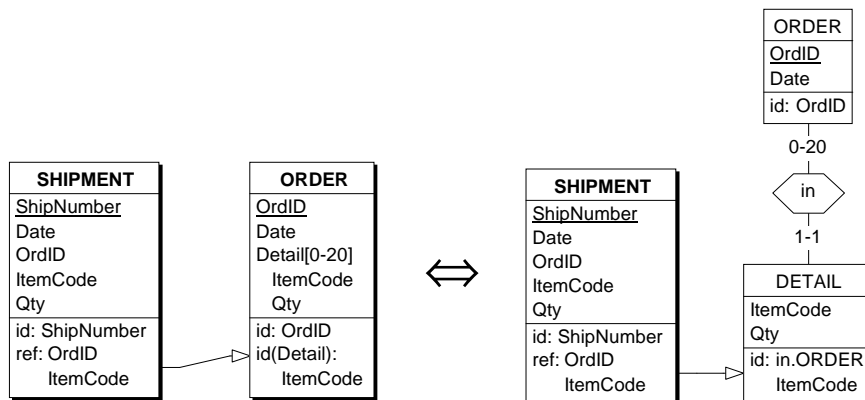


Figure 18: *Left:* each SHIPMENT entity references a definite value of the attribute Detail of an ORDER entity. *Right:* this reference is easy to interpret once the multivalued attribute has been transformed into an entity type.

3.6 Computed foreign key

A component of a *computed foreign key* is an indirect reference to the corresponding component of the target identifier. For example, a Date value can be used to denote a Year, or a Customer ID can be used to denote a City (the City of the designated customer). We will call (a bit inappropriately) *computed foreign keys* such patterns. There can be several ways to derive the explicit foreign key from the actual one. We will illustrate the most common ones through two examples: computation and table lookup.

The first example (Figure 19, left) expresses that each purchase is assigned to a fiscal year. However, the PURCHASE entity type does not include an explicit foreign key to FISCAL-YEAR. Instead, the stored value is {Date} from which the actual foreign key, denoted by $f(\text{Date})$, can be computed through time manipulation functions.

In the second example (Figure 19, right), purchases are liable to a definite tax rate, depending on the country of the customer and on the year of purchase. The actual foreign key should be

{Country,Year}, but is stored as {Customer,Year} instead. However, the actual value can be computed from the stored one by considering that the Country of a PURCHASE is the CountryName of the CITY of the CUSTOMER of the PURCHASE. Hence the foreign key notation $\{f(\text{Customer}),\text{Year}\}$.

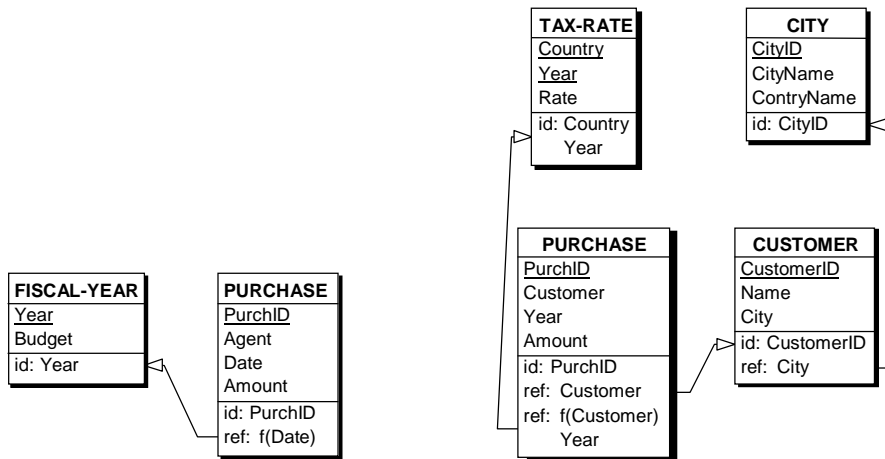


Figure 19: Two examples of computed foreign keys. *Left:* the value of PURCHASE.Date can be used to identify a fiscal year. *Right:* the value of PURCHASE.Customer allows us to get the CityName of the concerned customer; this value, combined with a Year value, is used to identify a tax rate.

The computed foreign keys are transformed into relationship types. However, the source arguments of the foreign key must often be kept, leading to some kind of redundancy which must be described by an explicit integrity constraint.

The first pattern can be processed in the ways described in Figure 20. In the right side translation, the attribute Date has been trimmed in order to eliminate the redundancy induced by its Year component.

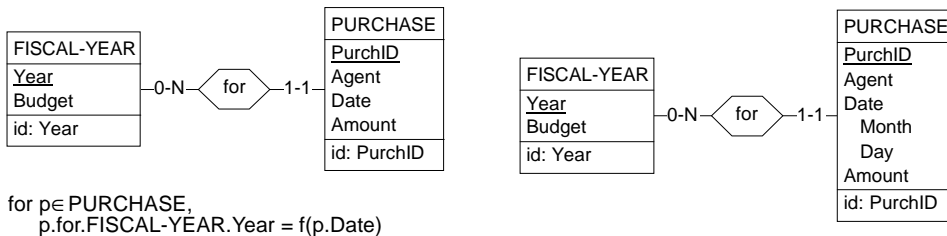


Figure 20: Two equivalent translations of the computed foreign key of Figure 19, left.

The second pattern can be translated as illustrated in Figure 21 (left). By extracting the common attribute Country in TAX-RATE and CountryName in CITY as a single autonomous entity type, we can propose the more expressive schema of Figure 21, right.

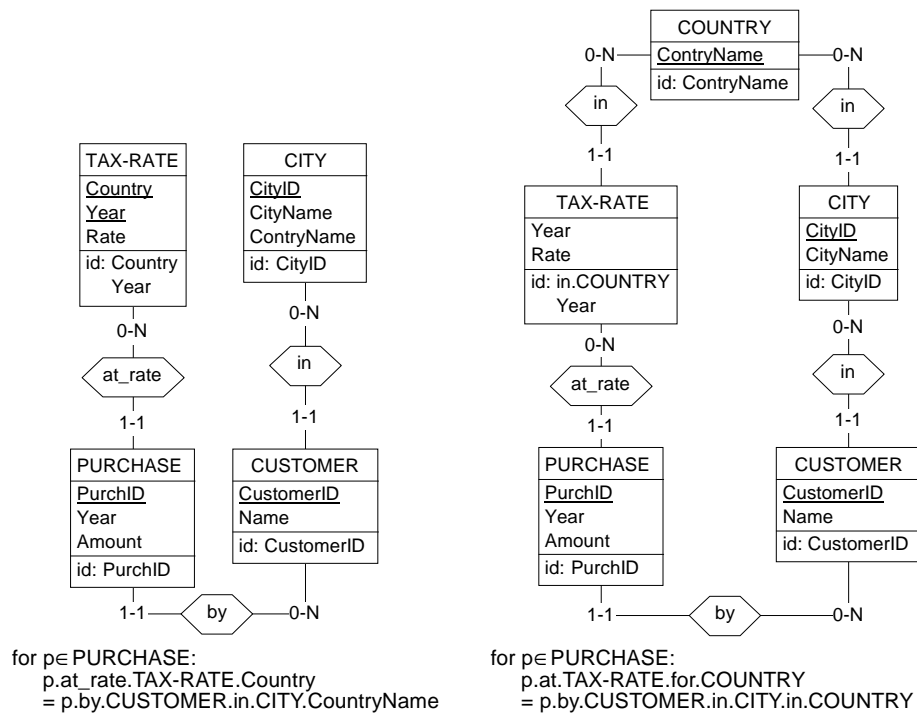


Figure 21: Interpreting the lookup computed foreign key of Figure 19, right.

3.7 Non-1NF foreign keys

A so-called *first normal form* (1NF) relation is defined on simple domains only. In more practical words, its attributes are atomic and single-valued. Many DMS provide more complex constructs that allow developers to define compound and/or multivalued attributes. They allow defining *non-1NF* structures. Such is the case of file managers, CODASYL, IMS and object managers. In addition, even 1NF schemas can include hidden, or implicit, non-1NF constructs, as shown in Section **XXXXXXXX**.

Quite naturally, some non-1NF attributes are, or include, foreign keys as well. For instance, a component of a compound attribute, or a multivalued attribute, can be used to reference entities. These attributes are called non-1NF foreign keys.

The examples of Figure 22 include respectively one level-2 foreign key (left) and two multi-

valued foreign keys (right), one of them being at the first level and the other one at the second level.

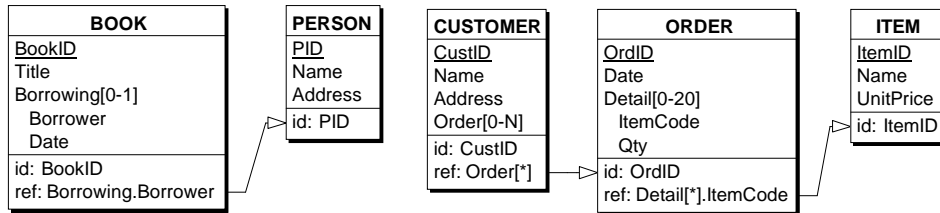


Figure 22: Three examples of non-1NF foreign keys.

We consider four main situations, from which all the other patterns can be solved easily.

1. The foreign key is a *component of a level-1, mandatory, single-valued, compound attribute*: disaggregate the parent attribute first (case not illustrated).
2. The foreign key is a *component of a level-1, optional, single-valued, compound attribute* (Figure 22, left; foreign key BOOK.Borrowing.Borrower): disaggregate the parent attribute (leading to Figure 23, a) or transform it into an entity type first (leading to Figure 23, b).
3. The foreign key is a *level-1 multivalued attribute* (Figure 22, right; foreign key CUSTOMER.Order[*]): interpret this foreign key as a many-to-many relationship type (leading to relationship type *place* in Figure 24).
4. The foreign key is a *component of a level-1, multivalued, compound attribute* (Figure 22, right; foreign key ORDER.Detail[*].ItemCode): transform the parent multivalued attribute into an entity type first (leading to entity type *Detail* and relationship types *of* and *ref* in Figure 24).

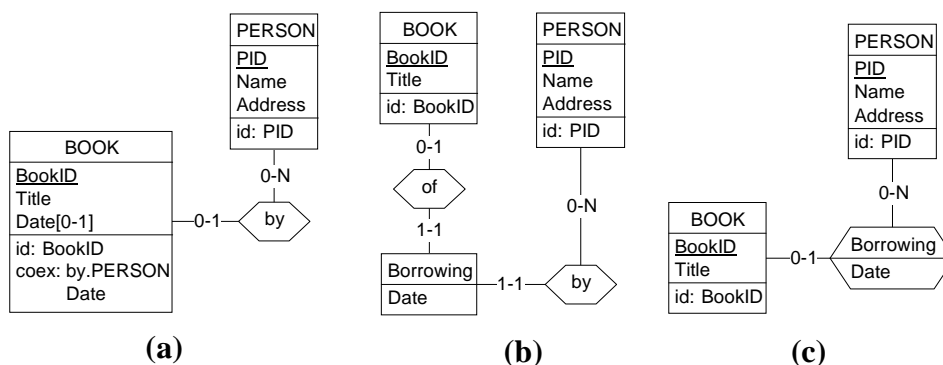


Figure 23: Three semantically equivalent interpretations of the first example, ordered by increasing expressivity. The schema (a) derives from an initial disaggregation of Borrowing, the schema (b) is

obtained from the transformation of the attribute Borrowing into an entity type, while the schema (c) refines the second one by transforming the new entity type into a relationship type.

These techniques must be iteratively applied until the resulting pattern can be solved through standard interpretation. It is important to observe that a multivalued foreign key basically represents a *many-to-many* relationship type (Figure 24), unless it is also declared an identifier of its entity type, in which case it translates into a *one-to-many* relationship type.

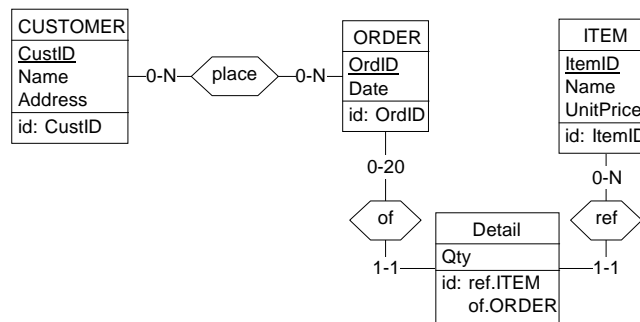


Figure 24: The two multivalued foreign keys of Figure 22 (right) have been interpreted. Note that *place* is *many-to-many*, since a given value of `ORDER.OrdID` can appear as `CUSTOMER.Order[*]` in more than one `CUSTOMER` entity.

4 Inclusion constraints

A foreign key is a structure that enforces a special case of inclusion constraint, where the target set is that of an identifier of the target entity type. Several authors include the study of some inclusion patterns into the foreign key domain [Petit, XX]. We will analyze two variants of this concept.

4.1 Inclusion constraint

The values of an attribute (or of a list of attributes) of an entity type are included into the set of values of an attribute (or of a list thereof) of another entity type. In the example of Figure 25, the entity type SUPPLY provides the conditions (quantity and price) at which each supplier can supply each item; a customer order can be assigned to a supplier only if this supplier can supply the item ordered.

Though true (i.e., non referential) inclusion constraints can be kept in the conceptual schema, it is best to try to express them into explicit constructs. The simplest approach consists in representing each target tuple by an explicit entity, therefore transforming the inclusion constraint into a referential constraint. Applying this technique to our example, we transform the couple of attributes SUPPLIER.(Supplier,Item) into an entity type through the value representation variant. The source couple ORDER.(Supplier,Item) automatically transforms into a standard foreign key that can be further processing as usual.

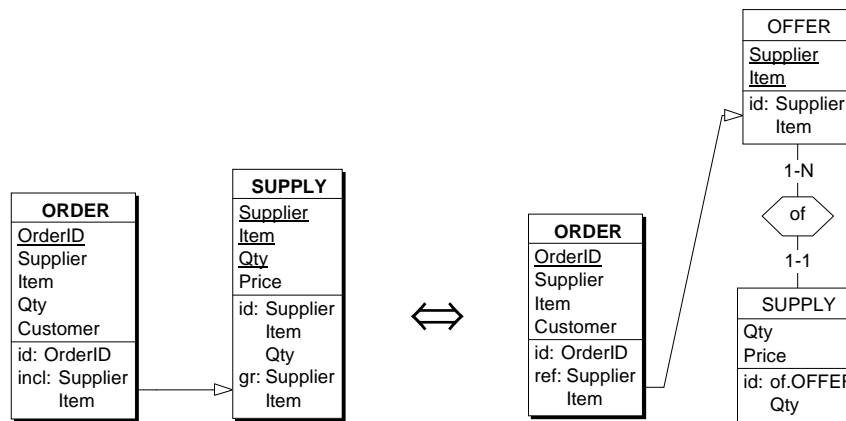


Figure 25: An inclusion constraint (left): the couple of values of (Supplier,Item) of each ORDER instance must appear in at least one SUPPLY entity. This constraint is replaced with a pure referential constraint (via a standard foreign key) through transforming the target attributes into an entity type.

4.2 Domain sharing

Two attributes share the same domain of values which happens to be particularly meaningful in the application domain, so that, at some time, they can take the same values. Most often, it has been found by program analysis, for instance as an SQL join-based query.

Figure 26 (left) illustrates the situation. An item is offered in all the shops of a chain at a definite price. The shops of a chain are located in towns, and have a given size. Obviously, the attributes OFFER.Chain and SHOP.Name can share common values, as testified by the following programming patterns found in several programs:

```
select *
from   OFFER O, SHOP S
where  O.Chain = S.Name
```

However, no stricter constraints (such as $\text{OFFER.Chain} \subseteq \text{SHOP.Name}$) can be stated.

A common interpretation can be described as follows: both attributes are extracted as entity types, which are then integrated into the unique entity type CHAIN (Figure 26, right). A strict equivalence would require an *at-least-one* constraint on CHAIN, since only chains that appear in OFFER or in SHOP or in both are represented in the left side schema. This constraint can generally be dropped.

For obvious reason, some authors consider this reasoning as the explicitation of hidden or *implicit objects* (here CHAIN).

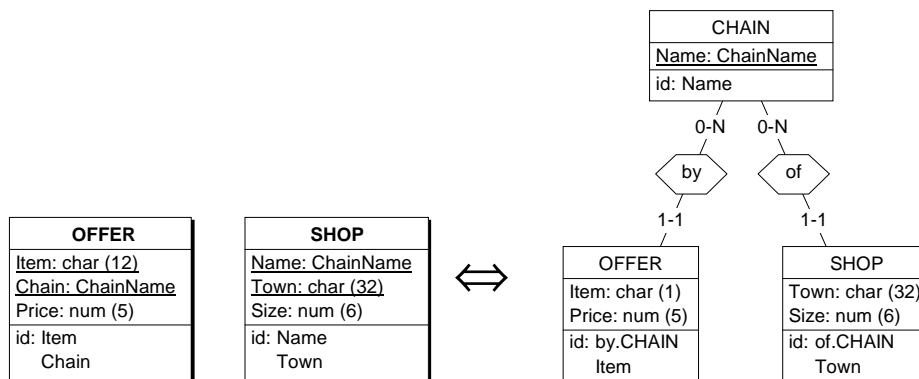
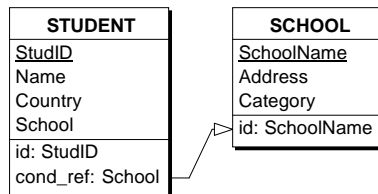


Figure 26: The entity types OFFER and CHAIN share the same domain ChainName through their attribute OFFER.Chain and SHOP.Name. This domain is represented by the explicit entity type CHAIN.

5 Complex foreign key patterns

This section will describe foreign key patterns that are more complex to understand and to conceptualize. They generally require initial transformations to make their semantics clearer.



for $s \in \text{STUDENT}$: $s.\text{Country} = \text{"Belgium"} \Rightarrow s.\text{School} \in \text{SCHOOL.SchoolName}$

Figure 27: The attribute `STUDENT.School` references a `SCHOOL` entity only if the `STUDENT` entity describes a Belgian student. Otherwise it gives the name of the unregistered school the non-Belgian student comes from.

5.1 Conditional foreign key

If a foreign key is conditional, it references entities only under a definite condition, otherwise, it is given another interpretation. The schema of Figure 27 illustrates the concept: for each student, the administration records the school from which s/he originates. If the student is Belgian, then the school must be one of the known Belgian institutions.

Analysis

Clearly, the attribute `STUDENT.School` encompasses two different semantics, depending on some filtering condition (*being Belgian or not*). We replace this attribute with two exclusive optional attributes `Belgian-School` and `Foreign-School`. For any `STUDENT` entity, there is either a `Belgian-School` value or a `Foreign-School` value. `BelgianSchool` has a not null value if and only if `Country` is set to "Belgium". The value of `BelgianSchool` is a foreign key to `School`. This expansion is illustrated in Figure 28 (left).

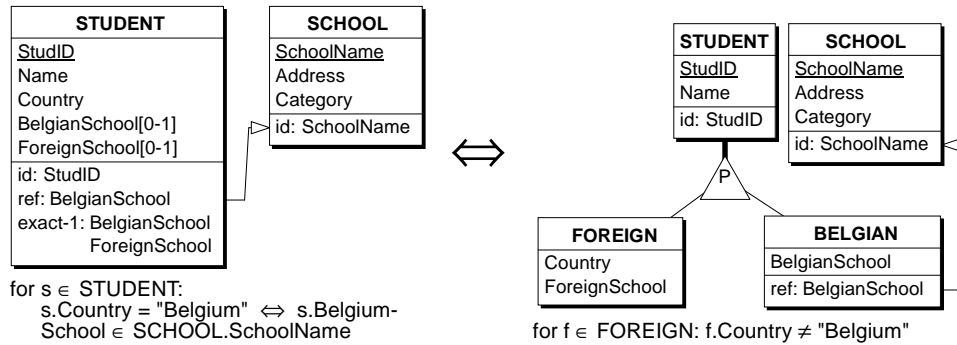


Figure 28: The dual semantics of the attribute `STUDENT.School` leads to defining two distinct attributes (left) or two subtypes of students (right).

This schema suggests partitioning the students into two categories, namely Belgian students and Foreign students (Figure 28, left). The foreign key can then be processed in the standard way.

5.2 Overlapping identifier - foreign key

The foreign key shares some attributes with an identifier of the entity type. We distinguish two patterns.

1. All the components of the foreign key also appear in the identifier.
2. Neither the foreign key nor the identifier include the other one.

The first pattern in which the foreign key is completely included into the identifier can be processed in the standard way. The components of the foreign key that appear in the identifier are replaced with the target role of the corresponding relationship type (see Figure 7 for a similar example; see also the synthesis of Section 8.2).

The second pattern (Figure 29) is more delicate. Indeed, the foreign key cannot be completely replaced with a relationship type as in the previous situation, since some (but not all) of its components belong to the identifier.

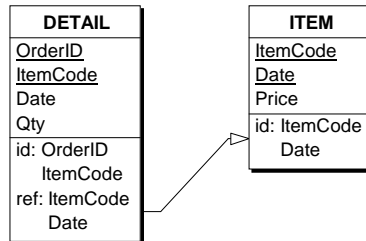


Figure 29: The identifier and the foreign key of the entity type DETAIL share the common attribute ItemCode. The foreign key cannot be replaced with a relationship type as in standard patterns.

Analysis

According to the usual approach, the foreign key should be replaced by a relationship type, which would be absurd since it would imply replacing, in the identifier, the common components with a part only of the relationship type. To solve the problem, we include the new attribute ItemCode_R that is a pure copy of the common attribute(s), so that we can separate the identifier components from those of the foreign key, which costs us an additional integrity constraint (Figure 30, left).

We can now transform the foreign key into a relationship type. Since the redundancy has not been removed but merely transformed, we must express it as an integrity constraint (Figure 30, right).

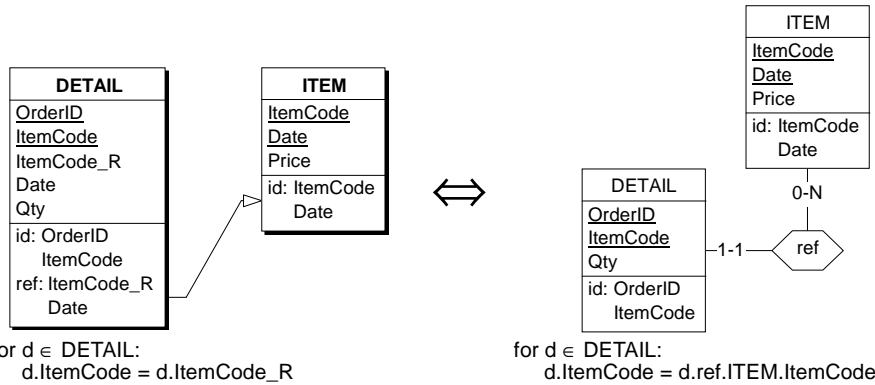


Figure 30: The identifier and the foreign key are separated thanks to the duplication of the common attributes. This redundancy remains in the conceptual schema.

In a similar, but more complex, pattern XXXXXXXXXXXXXXXXXXXXX

5.3 Overlapping foreign keys

Two foreign keys overlap if they share one or several attributes and if none is a subset of the other. In the example of Figure 31, each line of invoice belongs to an invoice and references a line of order. Both invoice and line of order reference the order they originate from.

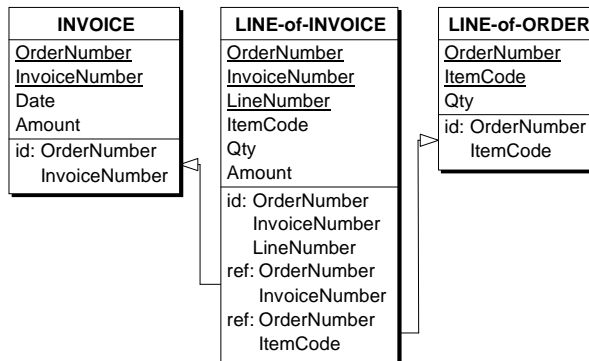


Figure 31: The overlapping foreign keys share the common attribute OrderNumber. Neither foreign key can be replaced with a relationship type without the other being destroyed.

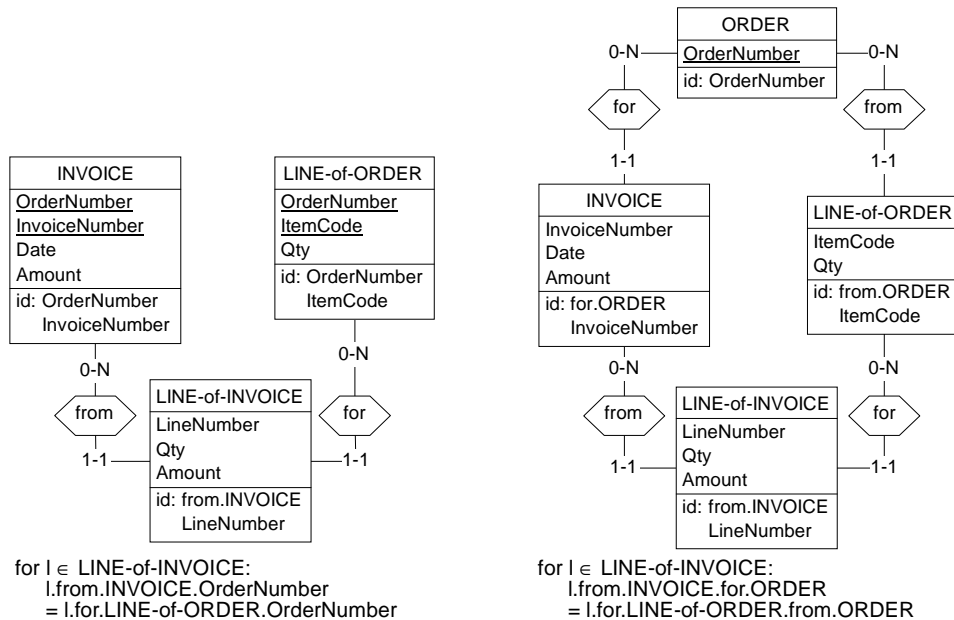


Figure 32: Two valid interpretation of overlapping foreign keys. Both include redundancies.

Analysis

Because of the common attribute `OrderNumber`, none of the foreign keys can be replaced completely with a relationship type. Following a reasoning that is close to that of Section 5.2, we duplicate the common attribute into `OrderNumber_L`, we express this redundancy by an integrity constraint, then we substitute the new attribute for the previous one in the second foreign key. We can now translate each independent foreign key into a relationship type (Figure 32, left). If the common attribute represents an important concept, it can be extracted as an autonomous entity type, namely `ORDER` in our example. The redundancy constraint is modified accordingly (Figure 32, right).

5.4 Non-minimal FK

This pattern does not concern the foreign key itself, but rather the referenced primary identifier. Indeed, the latter is a superset of another, minimal, identifier, and therefore is not minimal. It goes as follows (Figure 33).

1. A `LECTURE` entity represents the fact that a lecturer teaches a given subject. A lecturer is allowed to teach one subject only. This fact is notified by the secondary identifier `{Lecturer}`. A (trivial) primary identifier comprising `{Subject,Lecturer}` has been defined for technical reasons we will explain below.
2. A `REGISTRATION` entity states that a student is taught a subject by a lecturer. Making `{Subject,Lecturer}` a foreign key to `LECTURE` ensures that this lecturer actually is allowed to teach this subject.

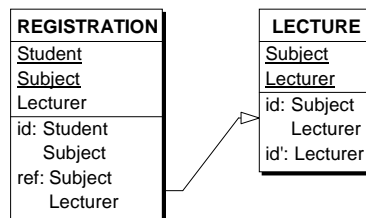


Figure 33: Unexpectedly, the foreign key references a non minimal primary identifier.

Analysis

This apparently disturbing pattern is a clever trick to implement 3NF schemas that are not in BCNF⁶. The basic reasoning can be sketched as follows.

We must first recall some basic facts.

6. A relation `R` is in third normal form (3NF) if no non-key attributes functionally depend on a strict subset of a candidate key, or on non-key attributes. It is in Boyce-Codd normal form (BCNF) if, for each functional dependencies that holds in `R`, the left hand side is a candidate key. All BCNF are in 3NF, but not conversely. See Chapter XXXXXX.

1. Standard RDBMS ensure two major integrity constraints only, namely unique keys (or identifiers) and foreign keys. Therefore, any constraint that can be reduced to these structures can be explicitly implemented into SQL-2. All the other constraints must be coded as CHECK predicates, TRIGGERS, STORED PROCEDURES or as procedural code sections scattered throughout the programs. This is the case for non-key functional dependencies, i.e., dependencies whose minimal left-hand side is not an identifier.
2. 3NF schemas include primary and foreign keys only, and can therefore be completely implemented in SQL-2.
3. Some relational schemas are in 3NF but not in BCNF, and therefore lead to data redundancy problems. Unfortunately, decomposing them in BCNF induces a new brand of problems: though all intra-relation FDs are key-based (each determinant is a full, minimal primary key), some FDs of the 3NF schema are lost because they were defined on attributes that are now distributed among several relations. The most popular example is the following.

```

registr(Student, Subject, Lecturer)
Lecturer  $\longrightarrow$  Subject
Student, Subject  $\longrightarrow$  Lecturer

```

The keys are {Student, Subject} and {Student, Lecturer}. The schema is trivially in 3NF since it has no non-key attributes. However, the determinant of one of the FDs is not a key. Therefore, the schema is not in BCNF.

4. Three solutions can be proposed to implement this schema.

- A.

```
registr(Student, Subject, Lecturer)
Lecturer  $\longrightarrow$  Subject
```
- B.

```
registr(Student, Subject, Lecturer)
lecture(Lecturer, Subject)
registr[Lecturer, Subject]  $\subseteq$  lecture
```
- C.

```
registr(Student, Lecturer)
lecture(Lecturer, Subject)
registr*lecture: Student, Subject  $\longrightarrow$  Lecturer
```

Each schema has its advantages and its drawbacks. However, one of them only, namely B, can be adapted in such a way that all the integrity constraints are translatable into pure SQL-2 constructs.

The revised version of B is obtained as follows: a new primary key comprising all the attributes of lecture is defined, while the original key is expressed as a mere candidate key (secondary id). The inclusion constraint can then be translated into a foreign key. Hence the following schema,

- B'.

```
registr(Student, Subject, Lecturer)
lecture(Lecturer, Subject)
primary-id(lecture): {Lecturer, Subject}
secondary-id(lecture): {Lecturer}
registr[Lecturer, Subject]  $\subseteq$  lecture
```

This schema is the exact relational interpretation of Figure 33.

Following this discussion, we can propose a transformation in which the foreign key `registr.(Lecturer,Subject)` is replaced with relationship type *for*. However, the attribute `Subject` cannot be removed since it participates in the primary identifier of `REGISTRATION`. This situation is a special case of *overlapping identifier - foreign key* (Section 5.2) and can be solved accordingly. The result is shown in Figure 34.

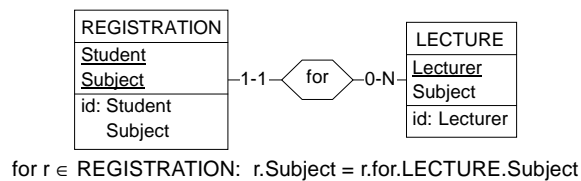


Figure 34: Conceptual expression of the schema of Figure 33. Deriving from a 3NF but non-BCNF schema, it includes a redundancy that must be expressed explicitly.

5.5 Partially reciprocal foreign keys

This pattern of interleaved foreign keys is a concise and elegant way to represent a bijective (one-to-one) relationship set included into another relationship set. When considering its conceptual equivalent, the source relational schema appears much more concise and free from complex additional integrity constraints. Unfortunately, the price to be paid for this conciseness is that its meaning is far from intuitive.

The schema of Figure 35 expresses that cities are located in countries (or states) and that one of the cities of each country is its capital. Several cities can have the same name (e.g., Paris, Venice), but not in the same country.

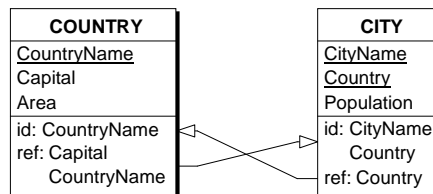


Figure 35: The foreign keys `COUNTRY.(Capital,CountryName)` and `CITY.Country` are partially reciprocal.

Analysis

A deeper analysis of the pattern shows that two important additional properties can be inferred from the declared structures.

First, we observe that the foreign key `COUNTRY.(Capital,CountryName)` is a non-minimal identifier, since it is a superset of the primary identifier of `COUNTRY`. Though this property

is derived and need not be specified, we represent it in the schema of Figure 36 to clarify the reasoning.

Secondly, the foreign key `CITY.Country` is total, and must be represented by the tag *equ* instead of *ref*. Indeed, interpreting the entity types of the schema of Figure 35 as relations, we can express the foreign keys as,

$$\begin{aligned} \text{COUNTRY}[\text{CountryName}, \text{Capital}] &\subseteq \text{CITY}[\text{Country}, \text{CityName}] \\ \text{CITY}[\text{Country}] &\subseteq \text{COUNTRY}[\text{CountryName}]. \end{aligned}$$

The first constraint also implies,

$$\text{COUNTRY}[\text{CountryName}] \subseteq \text{CITY}[\text{Country}]$$

so that,

$$\text{COUNTRY}[\text{CountryName}] = \text{CITY}[\text{Country}].$$

The schema of Figure 36 has been enriched with these properties.

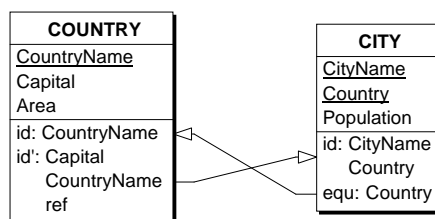


Figure 36: The *partially reciprocal foreign keys* pattern refined.

Transforming the foreign keys into relationship types must be carried out carefully, because each foreign key references an identifier that is involved in the other foreign key, so that both keys must be replaced simultaneously. In addition, not all the components of the keys can be replaced with the relationship types. Indeed, while `CITY.Country` and `COUNTRY.Capital` can be replaced, `COUNTRY.CountryName` must be kept, because it is the primary ID of `COUNTRY`. The attributes and relationship types of transformed structure are shown in Figure 37.

We have transformed the foreign keys into relationship types, except for the component `CountryName`, which is an unavoidable redundancy that still have to be declared. Examining the source schema, we observe that each `COUNTRY` entity references a `CITY` entity that precisely references it. Indeed, for any `COUNTRY` entity a , the referenced `CITY` entity b (its capital) has a `Country` value which is equal to $a.CountryName$. Consequently, in the interpreted schema of Figure 37, any instance (a,b) of *capital* is an instance of *in* as well. Hence the inclusion integrity constraint that translates the redundancy.

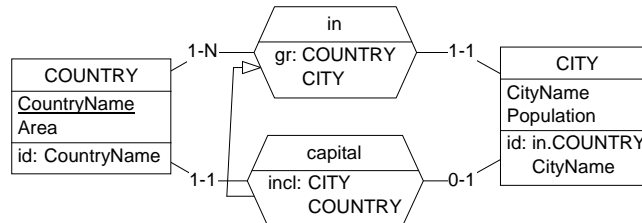


Figure 37: The conceptual interpretation of partially reciprocal foreign keys is less concise but clearer than its relational source of Figure 35.

5.6 Inverse foreign keys

Normally, in a relational database, a foreign key K from source table B to target table A is a construct that allows programmers to get an A row from a B row, and conversely. Provided both the foreign key and its matching candidate key are supported by indexes, the database engine can provide equally fast access in both directions.

This is not necessarily true in more primitive data managers, such as elementary file management systems. In this case, access from B to A is allowed, but there is no means to get B records from A quickly. The most obvious approach consists in defining an inverse foreign key from A to B . Such a key often is multivalued. In addition, object-oriented DBMS generally implement inter-object links by including an A -based attribute into object type B , or an B -based attribute in object type A , or both. In the latter case, these attributes act as inverse references. Some OO-DBMS even offer a way to declare the inverse property explicitly.

According to the example of Figure 38 (left), customers have placed orders (CUSTOMER.Orders) and each order has an owner (ORDER.Owner). The inverse constraint (with tag *inv*) specifies that any order o is one of the orders of the owner of o , i.e.,

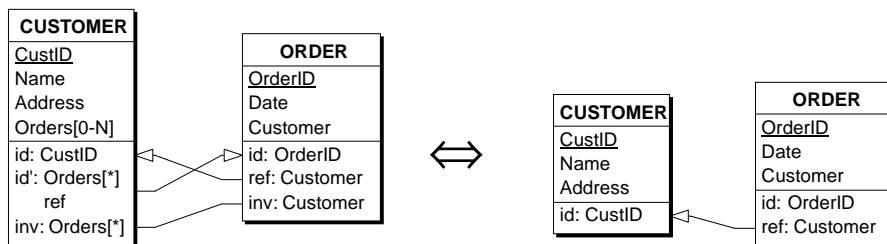
$$\text{for } o \in \text{ORDER}, c \in \text{CUSTOMER}, o.\text{Customer} = c.\text{CustID} \Leftrightarrow o.\text{OrderID} \in c.\text{Orders}$$


Figure 38: Two inverse foreign keys provide bi-directional navigation but induce data redundancy. Discarding one of these foreign keys removes this redundancy.

Analysis

A first observation will help us understand the various patterns: *a foreign key is single-valued iff its inverse is an identifier*. If the source schema does not comply with this property, then it is either inconsistent or insufficiently refined.

Since any of the foreign keys is a pure redundancy from the information point of view, it can be removed (Figure 38, right). To comply with standard practice, it is best to keep the single-valued foreign key, if any. In case of ambiguity, preferably keep the mandatory key. Through this cleaning operation, we get a pattern that has already been described, either a standard foreign key or a non-1NF foreign key. We will consider the three typical situations.

First case: both inverse foreign keys are single-valued. According to the property recalled above, each key is an identifier (a property generally left implicit⁷). We keep one of the foreign key, preferably that which is mandatory, if any (Figure 39). The result translate immediately into a *one-to-one* relationship type.

It is interesting to note that this pattern (with identifiers ignored) is proposed by some textbooks and CASE tools as the preferred implementation of *one-to-one* relationship types. Needless to say that this proposal is particularly awkward since the cleaned schema can be implemented in SQL-2 without any such trick.

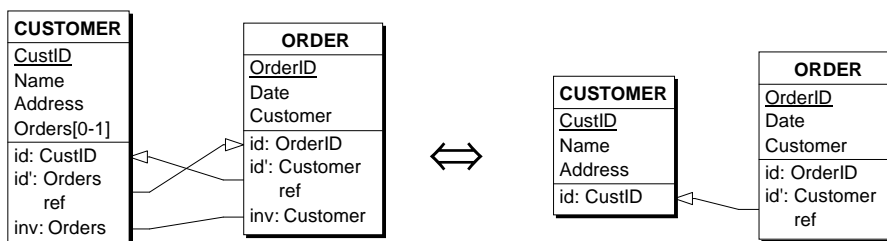


Figure 39: When both inverse foreign keys are single-valued, they must be identifiers as well.

Second case: one of the inverse foreign keys is multivalued. Since this foreign key is the inverse of a single-valued foreign key, it is an identifier as well. We keep the single-value key (Figure 38), which can be transformed into a *many-to-one* relationship type.

Third case: both foreign keys are multivalued. One of them is kept (Figure 40), which yields a *many-to-many* relationship type.

7. In which case the foreign keys produce two *many-to-one* relationship types. Since they are inverse of each other, they specialize into a *one-to-one* relationship type (if a function is the inverse of a function, it is a bijection).

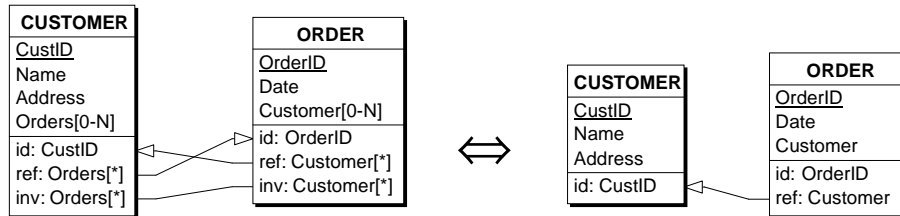


Figure 40: Both inverse foreign keys are multi-valued.

5.7 Meta foreign keys

A meta-FK is an attribute or of set of attributes, each value of which identifies a set of entity types.

<to develop>

This pattern was discovered in the database of Claroline, an e-learning platform: each course is represented by a set of MySQL (single-row) tables. As a consequence, the size of the schema is dependent on the number of rows in some tables.

6 Temporal foreign keys

<to develop as a special case of *Interval FK*>

This section will address a special case of computed foreign keys that is so common that it deserves a special treatment, namely the *temporal foreign keys*. Other special domains can be considered in the same way, but we will limit the discussion to this popular category.

Most databases include, in some way, a temporal dimension, in particular when the history of the real world facts and events has to be recorded. Temporal data can be more complex than current data, that merely record the current state of the world. When the data in one table reference data in another table, the temporal dimension leads to a new definition of the foreign key. Indeed, the referential constraint must be satisfied, not only for the current data, but also for the data considered at any time in the past. The domain of temporal databases is very rich and complex [Snodgrass, 2000], so that we will describe only some of the most basic aspects of temporal foreign keys.

Standard representation of temporal data

Though there are many ways to organize historical data, we will consider the most usual structure illustrated in the Figure 41. The table H_PROJECT contains the successive states of a set of projects (only the states of the project BIOTECH are shown). Two timestamp columns, namely *start* and *end*, indicate for each row the period during which the state described by the other columns remained (or remains) constant. This validity period is represented by a semi-open temporal interval $[start, end[$ in such a way that the state was valid from the instant *start* (included) and was finished at time *end* (not included). For instance, the row p2 indicates that, at instant 41, the project BIOTECH changed its theme (from *Biotechnology* to *Genetic engineering*) and its budget (from 180,000 to 160,000). This state remained unchanged until the instant preceding 47, at which time (row p3) the budget was reduced by 40,000. An *end* value of 9999 represents the far future, so that the corresponding state is the current state of the project. It is assumed that the history of a project is continuous and shows no gap, i.e., non extremal periods during which no information was recorded; in addition, no two states of the same project overlap.

H_PROJECT					
	TITLE	start	end	THEME	BUDGET

p1	BIOTECH	10	41	Biotechnology	180,000
p2	BIOTECH	41	47	Genetic engineering	160,000
p3	BIOTECH	47	84	Genetic engineering	120,000
p4	BIOTECH	84	135	Genetic engineering	140,000
p5	BIOTECH	135	9999	Biotechnology	140,000

Figure 41: Excerpts from the table H_PROJECT, recording the successive states of the project BIOTECH, among others. The timestamp values are abstract integers to simplify the discussion.

Temporal foreign key

Now, we want to record the successive states of a population of employees. The table of Figure 42 shows some rows describing the evolution of the employee M158. The column PROJECT aims at referencing the project on which this employee worked, or still is working, during each state. We can guess that this reference is not as simple as in standard databases, in which only the current states of projects and employees are recorded. Let us consider the row e2 of H_EMPLOYEE. It informs us that, from instants 40 to 65, the employee M158 worked on the project BIOTECH. Two observations:

1. this information is valid, since this project was active during this period: the life period of the project, namely [10,9999], encompasses the validity period [40,65[of the state e2 of the employee;
2. the row e2 in H_EMPLOYEE references three successive states of this project; indeed, it has a period [40,65[that overlap (i.e., shares at least one common instant with) three successive periods of H_PROJECT, namely [10,41[, [41,47[and [47,84[.

H_EMPLOYEE							
	CODE	start	end	NAME	STATUS	ADDRESS	PROJECT

e1	M158	15	40	Mercier	T	Paris	BIOTECH
e2	M158	40	65	Mercier	P	Paris	BIOTECH
e3	M158	65	108	Mercier	P	Paris	SURVEYOR
e4	M158	108	9999	Mercier	P	Paris	BIOTECH

Figure 42: The table H_EMPLOYEE records the history of employees. In particular, it informs (through the column PROJECT) on which project each employee was working on during each state.

These observations allow us to state the definition of the *temporal referential integrity*. The contents of the tables are valid, as far as temporal referential integrity is concerned *iff*,

$$\begin{aligned}
&\forall e \in H_EMPLOYEE, \\
&\quad \exists p1, p2 \in H_PROJECT, \\
&\quad\quad e.PROJECT = p1.TITLE = p2.TITLE \\
&\quad\quad \wedge p1.start \leq e.start < p1.end \\
&\quad\quad \wedge p2.start < e.start \leq p2.end
\end{aligned}$$

Note that this definition is valid for target tables that satisfy the no-gap, no-overlap hypothesis. Otherwise, the definition is more complex. The schema of these tables is shown in Figure 43. The tag *tref* is used to denote the temporal foreign key.

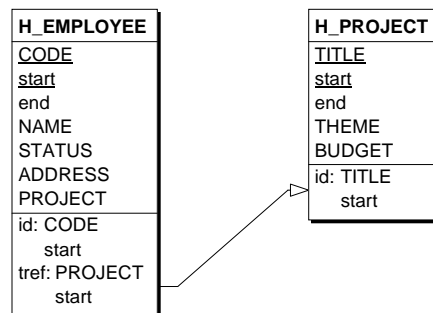


Figure 43: The column PROJECT is a temporal foreign key to the temporal table H_PROJECT.

Interpretation of a temporal foreign key

There is no standard definition nor representation to declare temporal structures at the conceptual level. We will use the graphical notation of the temporal ERA model defined in [Dettienne, 2001], shown in Figure 44, top. Alternately, we can use stereotypes to mark the constructs as temporal (Figure 44, bottom). These schemas indicate that the temporal foreign key is interpreted as a temporal relationship type. In this section, we have implicitly adopted the *valid time* interpretation, according to which the time period of a state is the set of instant at which the states was known to be valid in the real world⁸.

Degenerated forms of temporal foreign key

First of all, let us observe that the standard representation used in Figure 41 is redundant. Indeed, except for the current state of an entity, the value of *end* of a state is also the value of *start* of the next state. Therefore, the column *start* is sufficient to represent the history of an entity. The table ITEM_PRICE in Figure 45 stores the evolution of the prices of a set of items. The column Date indicates from which date the price of the item was applicable, until another price was assigned. The column Current is set to 0 for all the past prices and to 1 for the current price. Though such a column is not necessary, it is often added for performance

8. As opposed to the *transaction time*, that represents the period during which the state was recorded in the database. More of this in [Snodgrass, 2000] for instance.

reasons.

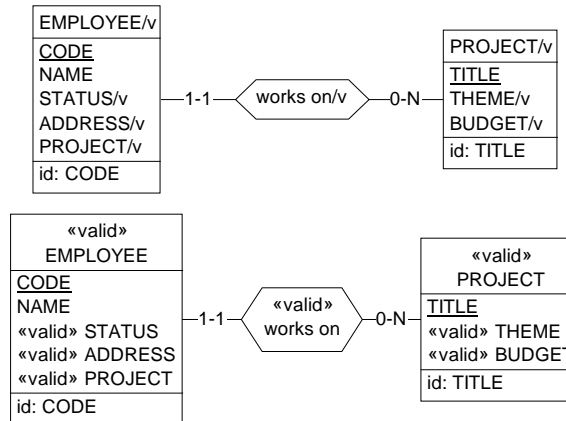


Figure 44: Two equivalent graphical representation of the conceptual structures derived from the schema of Figure 43. The objects marked with the symbols "/v" or "«valid »" are temporal.

In the standard representation (Figure 43), the timestamp columns *start* and *end* basically are technical data introduced to represent and process entity histories, and do not correspond to intrinsic properties of the entities. Many (if not all) schemas include temporal columns that represent natural events of the application domain. For instance, in the schema of Figure 45, the column *ORDER.Date* gives the date on which each order was placed. Such columns often are qualified by the term *user-defined time*, since their temporal semantics is known by the users of the data only. However, nothing prevents us to interpret a user-defined time column to participate in a temporal foreign key: thanks to the values of *ItemNum* and *Date* in the table *ORDER*, the matching row in *ITEM_PRICE* giving the price applicable can be identified easily. Hence the declaration of the temporal foreign key of the Figure 45.

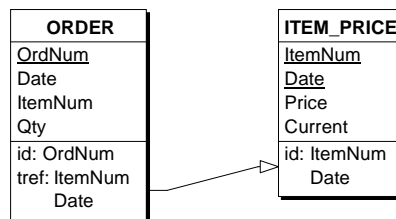


Figure 45: Both tables use a degenerated form of timestamping to represent historical data.

7 Pathological foreign keys

Though some of the foreign key patterns analyzed so far can be considered unusual, puzzling or tricky, each of them is a correct and logical technical answer to a standard or complex structural problem. Unfortunately, legacy logical schemas sometimes include awkward, or even wrong, foreign key patterns that can lead to erroneous conceptual schemas if processed carelessly.

In some situations, the problem can originate from two different causes. First, the flawed structure actually exists in the legacy database, as the result of an erroneous design or coding decision, and has been correctly reported in the logical schema we are conceptualizing. Secondly, the structure does not exist in the database, but has been introduced in the logical schema due to insufficient analysis during the Data Structure Extraction phase. It is important to identify the exact source of the problem, e.g., through more detailed data analysis.

7.1 Loosely-matching foreign key

The relational literature suggests (to say the least) that a foreign key and its corresponding candidate key be defined on the same domain. This rule is not always applied in practical databases, which often rely on the looser rule that both keys must be *comparable in some way*. The following correspondences have been found in COBOL and SQL data structures:

Foreign key	Target identifier	Evaluation
char(8)	char(8)	dom(FK) = dom(ID): standard pattern
char(8)	num(8)	dom(FK) \supseteq dom(ID): potential compatibility
num(8)	char(8)	dom(FK) \subseteq dom(ID): compatibility
char(12)	char(8)	dom(FK) \supseteq dom(ID): potential compatibility
char(8)	char(12)	dom(FK) \subseteq dom(ID): compatibility
char(10)	compound num(4) char(6)	dom(FK) \supseteq dom(ID): potential compatibility
compound num(4) char(6)	char(10)	dom(FK) \subseteq dom(ID): compatibility

Figure 46: Some frequent pattern of source and target domains. The expression dom(M) denotes the set of potential values of attribute(s) M.

Quite obviously, the problem is the elicitation of such loose foreign keys rather than their interpretation. Note that this problem can be considered as a *Computed foreign key* pattern (Section 3.6), and processed accordingly. In this case, the function is some kind of *casting*.

7.2 99% correct foreign key

Such a construct should be a foreign key, and actually *is* a foreign key most of the time. In other words, each key value is expected to reference an entity, but data analysis shows some exceptions, i.e., values which do not match any target entities. No explanation is given, except possible data errors. This pattern is fairly close to the conditional foreign key situation (Section 5.1).

In the example of Figure 47, most CUSTOMER entities reference a CATEGORY entity. Some of them however have been found to have Category values that fail to denote any known category.

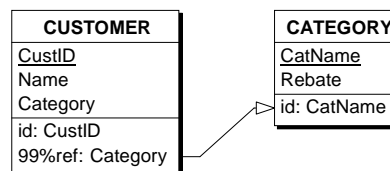


Figure 47: Most CUSTOMER entities have a Category value that references a CATEGORY entity.

Analysis

Two scenarios must be distinguished. According to the first one, the data errors can be ignored, and the construct is considered a plain foreign key and processed accordingly. This approach follows the idea that reverse engineering basically is a decisional process based on a collection of hints. The second scenario postulates that the exceptions must be taken into consideration, and represented explicitly. We address this second approach, which is imperative when data must be migrated⁹.

We replace the attribute Category with two exclusive optional attributes Category and Wrong_Category (Figure 48, left). For any CUSTOMER entity, there is either a Category value or a Wrong_Category value. The value of Category is a correct foreign key, which is interpreted in the usual way, while the value of Wrong_Category is erroneous, and is left uninterpreted (Figure 48, right).

If needed, the final schema can explicitly show the two kinds of CUSTOMER entities (Figure 49).

9. Identifying and discarding erroneous data in data migration is a process called *data cleaning*.

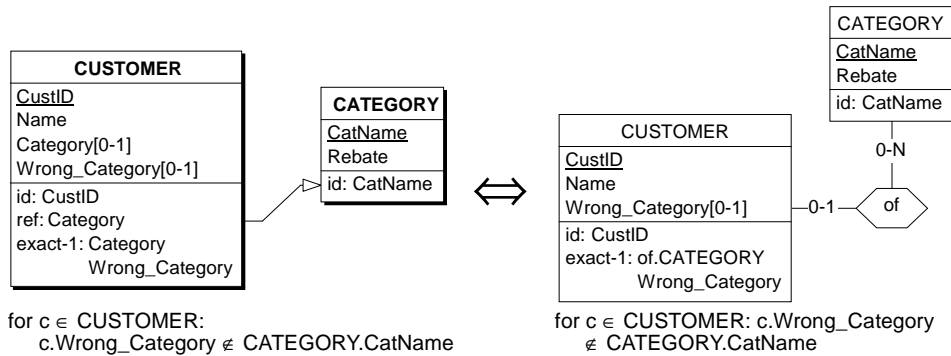


Figure 48: Each customer has a correct category reference, (in which case the latter translates into a relationship type), or it has a wrong category.

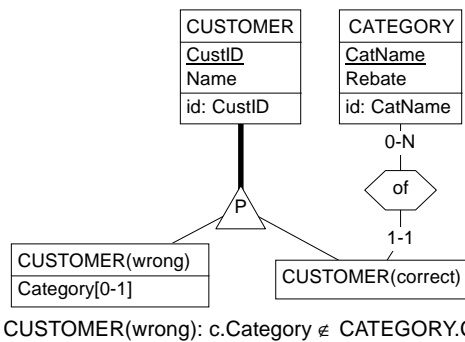


Figure 49: This schema clearly distinguishes the correct data from the wrong ones.

7.3 Transitive foreign key

A *transitive foreign key* is the composition of two or more other foreign keys. Being derived, such a foreign key can be removed. Depending on the relationship between these keys, the attributes forming the transitive foreign key can be removed or not.

Many transitive foreign keys have not been explicitly defined in the legacy database, but have been discovered through program and/or data analysis techniques.

Analysis

There are two different patterns, that require different processing. The first pattern is illustrated by the Figure 50, which models a situation in which invoices depend on orders and orders are placed by customers. Since the orders of each customer are identified by a unique

number, the identifier of the order is used to make invoice entities reference their orders. Consequently, each INVOICE entity includes the reference of the CUSTOMER entity, itself referenced by its ORDER entity. This reference is a transitive foreign key. It can be discarded without any information loss. However, the attribute that composes this foreign key must be preserved, since it belongs to another, basic, foreign key. Since there are no explicit redundant attributes, this pattern is called *non-redundant transitive foreign key*. It could have been explicitly declared by the developer, or, most probably, it was discovered during the Data Structure Extraction phase. This kind of transitive foreign key can be formally identified through the rules of Section 1.

In the example of the second pattern (Figure 51), each customer is in contact with an employee who is in charge of his/her problems. The employee depends on a given department. The developer gave the entity type CUSTOMER the attribute Department aimed at referencing the department of the employee of the customer. This foreign key too is transitive. However, its attribute itself is redundant and can be removed, hence the name *redundant transitive foreign key*. Since it is supported by a specific attribute, this transitive foreign key was intentional, most probably for performance reasons. The transitivity property cannot be formally identified and must be discovered through program/data analysis techniques.

After such cleaning, the schema can be processed through any other method described in the former sections.

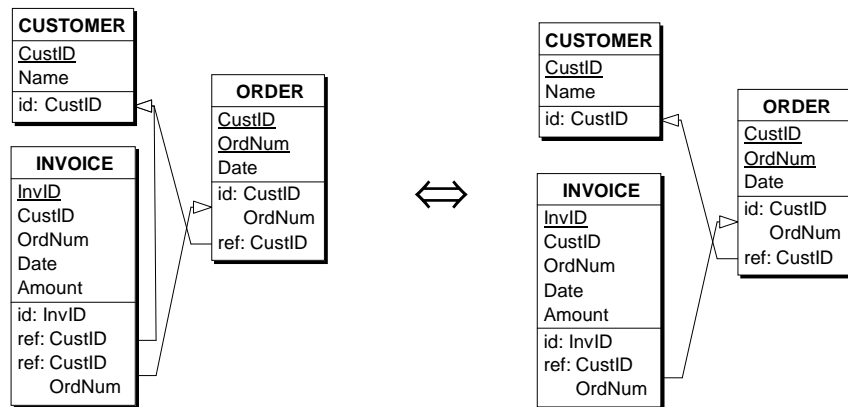


Figure 50: *Non-redundant transitive foreign key*: the customer of an invoice is the customer of the order of this invoice. Cleaning the schema consists in removing the transitive foreign key.

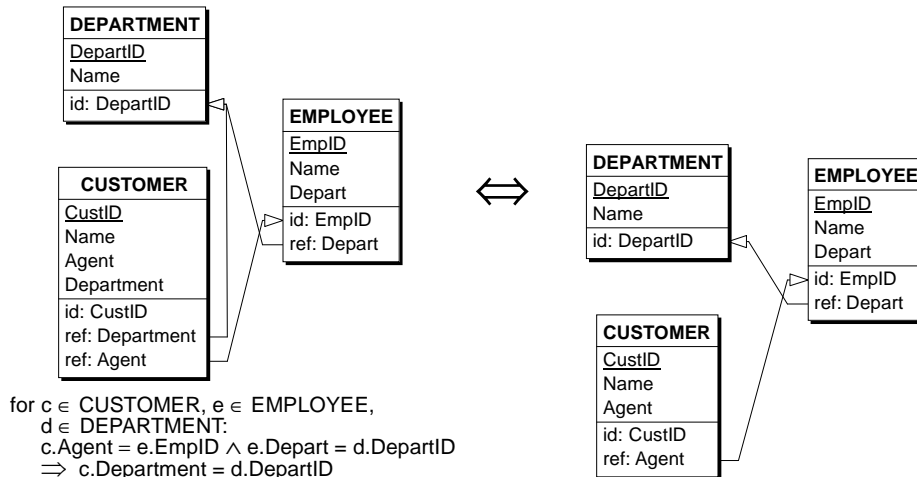


Figure 51: *Redundant transitive foreign key:* the department of a customer is the department of the employee in charge of this customer. Cleaning the schema consists in removing the transitive foreign key **and** its attribute components.

7.4 Partly optional foreign key

In a partly optional foreign key, some, but not all, components of a multiple-component foreign key are optional. Though this form is perfectly legal in SQL-2, it violates the principles of optional foreign keys. In particular, there exist no known ERA/SQL translation rules that can produce such a pattern.

As an example, we consider the schema of Figure 52, that describes a situation in which a collection of dissertation titles are proposed to last year students. A dissertation is identified by its title and the year it is being, or has been, proposed. Students are characterized by their name and the year they have to choose a dissertation subject. When they have made this choice, they are given the title of this dissertation. Technically speaking, when attribute Dissert of a STUDENT entity is null, then this entity references no DISSERTATION entity, while when Dissert is not null, then (Dissert, Year) references a DISSERTATION entity.

Analysis

The source schema is awkward. Indeed, the components of a standard foreign key must all be mandatory (not null) or all optional (nullable). In the latter case, the components are subject to a coexistence constraint (see Section 2.3). We must first clarify the schema by considering two kinds of STUDENT entities, namely those which have no Dissert values and which do not reference any DISSERTATION entity, and those which have a value for their attribute Dissert, and therefore reference a DISSERTATION entity. The latter students form the class

of last-year students.

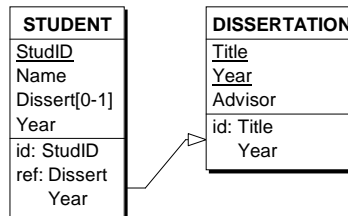


Figure 52: The foreign key (Dissert,Year) is made up of optional and mandatory attributes.

We define the subtype LAST-YEAR-STUDENT as the collection of STUDENT entities which have a not null Dissert value. The mandatory attributes Dissert and Year of this new entity type form a standard foreign key targeting DISSERTATION (Figure 53, left). Note that the attribute Year of LAST-YEAR-STUDENT is noted STUDENT.Year to indicate that it is inherited from STUDENT.

The cleaned schema now includes a standard foreign key that is easily transformed (Figure 53, right).

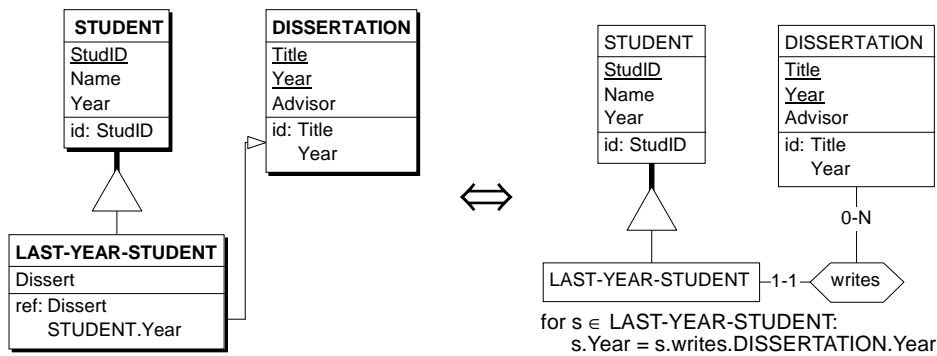


Figure 53: Cleaning a partly optional foreign key by making the subtype LAST-YEAR-STUDENT explicit.

7.5 Embedded foreign key

An embedded foreign key is made up of attributes that also are components of another foreign key. As we shall see it generally suggests both a transitive foreign key, and a missing foreign key. This pattern is frequent in database schemas where foreign keys have been elicited through program and data analysis. Except in badly designed schemas, the embedded foreign key has not been declared explicitly.

The schema of Figure 54 describes invoices that depend on orders and that are sent to cus-

tomers. We observe that the component of the foreign key `INVOICE.Customer` form a *proper subset* of the foreign key `INVOICE.(Customer,Order)`.

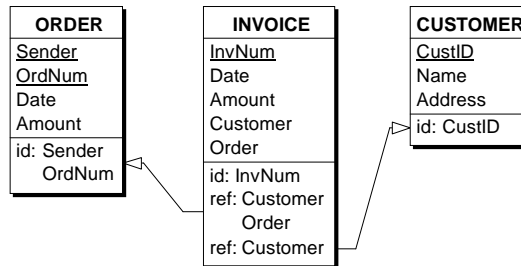


Figure 54: An embedded foreign key is an evidence of an incompletely refined physical schema.

Analysis

An idea emerges immediately: couldn't `ORDER.Sender` be a foreign key to `CUSTOMER`? Obviously, if we can prove that `ORDER.Sender` \longrightarrow `CUSTOMER`, then the foreign key `INVOICE.Customer` \longrightarrow `CUSTOMER` is transitive and can be removed (see Section 7.3). Let us write the set relations that express the foreign keys:

1. $INVOICE.[Customer,Order] \subseteq ORDER.[Sender,OrdNum]$
 $\Rightarrow INVOICE.[Customer] \subseteq ORDER.[Sender]$
2. $INVOICE.[Customer] \subseteq CUSTOMER.[CustID]$

Unfortunately, from these expressions, we cannot infer that `ORDER.Sender` \longrightarrow `CUSTOMER`. However, unless the population of `INVOICE` is empty, some values of `ORDER.Sender` (among them, those which appear in `INVOICE.Customer`) are `CUSTOMER.CustID` values as well. So, we can distinguish two kinds of `ORDER` entities: those that reference `CUSTOMER` entities (their `Sender` values are in `CUSTOMER.CustID`) and those which do not. The former are collected in the `CUST-ORDER` entity type (Figure 55, left). Now, the schema exhibits an explicit transitive foreign key which can be removed.

The conceptual interpretation of the modified schema is immediate (Figure 55, right).

The source pattern can also be interpreted in another way. Instead of sticking strictly to this schema, we use it as an evidence of the possible foreign key `ORDER.Sender` \longrightarrow `CUSTOMER`, which so far is a mere hypothesis. Should we succeed in proving that it is an implicit foreign key, we could add it to the schema, and remove the transitive foreign key `INVOICE.Customer` \longrightarrow `CUSTOMER` (Figure 56).

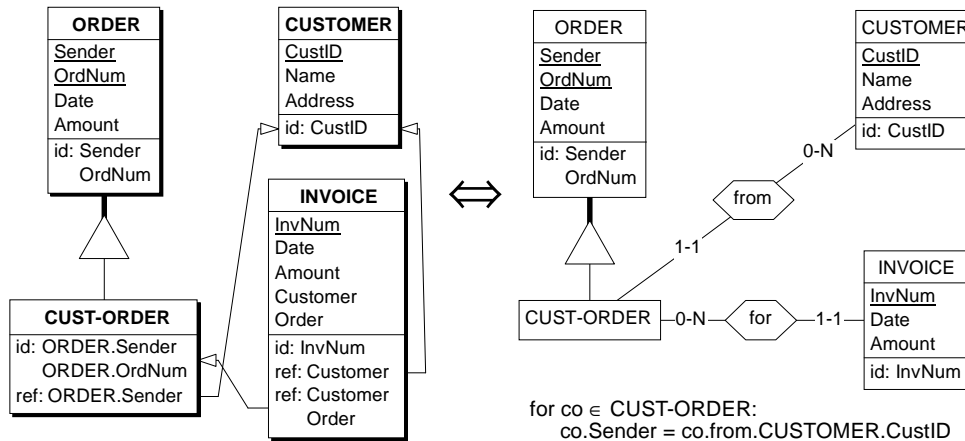


Figure 55: The missing foreign key has been added, leading to the known pattern of *non-redundant transitive foreign key*.

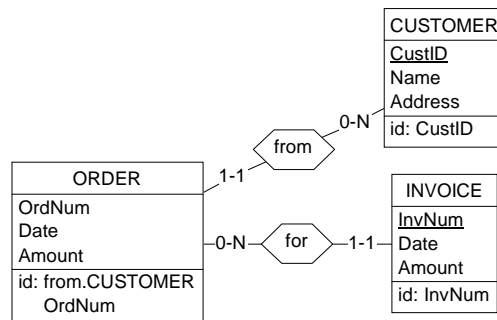


Figure 56: Turning the evidence into a decision: *ORDER.Sender* definitely is a foreign key to *CUSTOMER*.

7.6 Reflexive foreign key

This pattern includes a perfectly valid, though **strictly useless** foreign key. It is presented here for three reasons.

- This structure has been reported as an explicitly declared foreign key in an actual ORACLE application. Hopefully through code generation!
- It is a explicit implementation of the reflexivity property of foreign keys: $A[A1] \subseteq A[A1]$.
- We found it nice to close a rather serious section with a touch of humor¹⁰.

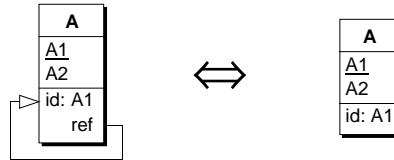


Figure 57: Art need not be useful.

Note that this pattern should not be confused with cyclic foreign keys through which it may happen that *some entities reference themselves*.

10. Quite frustratingly, humorous issues in the database realm often require some laborious comments before being enjoyed at their full potential. But most generally the reward is worth the journey.

8 Synthetic tables

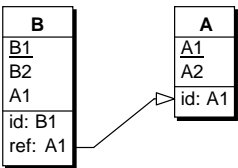
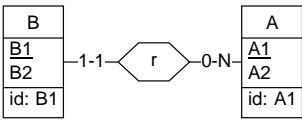
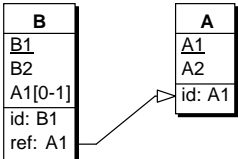
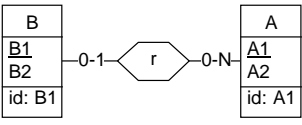
In this last section, we collect the most common foreign key variants into a series of tables, then we analyze the relationship between a foreign key and the local identifiers.

8.1 Summary of the most common foreign key patterns

Considering that a foreign key can be

- single-valued or multivalued,
- identifying or non-identifying,
- mandatory or optional,
- total (equality) or not,

we get the 16 basic combinations described in Tables 1 to 4 here below. We observe that the combinations of first two characteristics yield, respectively, *many-to-one*, *one-to-one*, *one-to-many* and *many-to-many* relationship types. This analysis is based on the most common cardinalities of both attributes and roles, namely [0-1], [1-1], [0-N] and [1-N]. Considering other cardinality patterns can be deduced without problems.

Foreign key pattern	Interpretation	Comment	see
		Mandatory foreign key. This standard pattern gives a many-to-one relationship type with a mandatory role and an optional role.	2.1
		The FK is optional, so that the role of r.B is optional too. The other role still is optional.	2.2

		<p>A total foreign key translates into a relationship type with two mandatory roles.</p>	<p>2.4</p>
		<p>Total, optional foreign key. Gives a mandatory role and an optional role.</p>	<p>2.2 2.4</p>

Table 1: Interpreting single-valued, non-identifying, foreign keys into **many-to-one** rel-types.

Foreign key pattern	Interpretation	Comment	see
		<p>A single-valued foreign key which is an identifier translates into a one-to-one relationship type.</p>	<p>2.5</p>
		<p>Same as above + Table 1.</p>	<p>2.5 2.2</p>
		<p>Same as above + Table 1.</p>	<p>2.5 2.4</p>
		<p>Same as above + Table 1.</p>	<p>2.5 2.2 2.4</p>

Table 2: Interpreting single-valued, identifying, foreign keys into **one-to-one** rel-types.

Foreign key pattern	Interpretation	Comment	see
		A multivalued foreign key which is an identifier translates into a one-to-many relationship type.	2.5
		Same as above + Table 1.	2.5
		Same as above + Table 1.	2.5
		Same as above + Table 1.	see 2.5

Table 3: Interpreting multivalued, identifying, foreign keys into **one-to-many** rel-types.

Foreign key pattern	Interpretation	Comment	see
		A multivalued foreign key which is not an identifier translates into a many-to-many relationship type.	2.5

		<p>Same as above + Table 1.</p>	<p>2.5</p>
		<p>Same as above + Table 1.</p>	<p>2.5</p>
		<p>Same as above + Table 1.</p>	<p>2.5</p>

Table 4: Interpreting multivalued, non identifying, foreign keys into **many-to-many** rel-types.

8.2 Relationship between a foreign key and the identifiers

Both an identifier and a foreign key comprise a set of attributes from the same entity type. It is quite natural to wonder whether the relation between these sets matters. Considering any two non empty sets E1 and E2, there are five basic relations:

- $E1 \cap E2 = \emptyset$
- $E1 \subset E2$
- $E1 = E2$
- $E1 \supset E2$
- $E1 \cap E2 \neq \emptyset \wedge E1 - E2 \neq \emptyset \wedge E2 - E1 \neq \emptyset$

Table 5 explores these five relations and links them with foreign key patterns described in the previous sections.

Foreign key pattern	Interpretation	Comment	see
		<p>The foreign key and the identifier <i>are disjoint</i>. There is no relationship between the identifier and the derived relationship type.</p>	<p>2.1</p>

		<p>The foreign key is a <i>proper subset of the identifier</i>. In the transformed schema, it is replaced with the relationship type in the identifier.</p>	2.4
		<p>The identifier and the foreign key <i>comprises the same attributes</i>. The derived relationship type is one-to-one and the identifier disappears.</p>	2.5
	<p>for $b \in B, b.B1 = b.r.A.A1$</p>	<p>The identifier is a <i>proper subset of the foreign key</i>, which therefore is a non-minimal identifier. It partially translates into a one-to-one relationship type. Since B1 cannot be removed, it is redundant, hence the constraint.</p>	5.5
	<p>for $b \in B, b.B2 = b.r.A.A1$</p>	<p>The identifier and the foreign key share a common attribute, but have each a proper attribute. The foreign key partially translates into a many-to-one relationship type. Since B2 cannot be removed, it is redundant, hence the constraint.</p>	5.2

Table 5: The five relationships between a foreign key and the local identifiers.

Exercises

Here follow some logical schemas including foreign keys. Unless suggested otherwise, the reader is invited to build, for each of them, an equivalent conceptual schema. Note that some schemas do not mimic exactly the patterns studied in this chapter. In such cases, the reader will design his/her own techniques following the frameworks suggested above.

Exercise 1. A first simple schema

This one is quite easy, since it includes standard foreign keys only (Ex-1). Nevertheless, some attention could help.

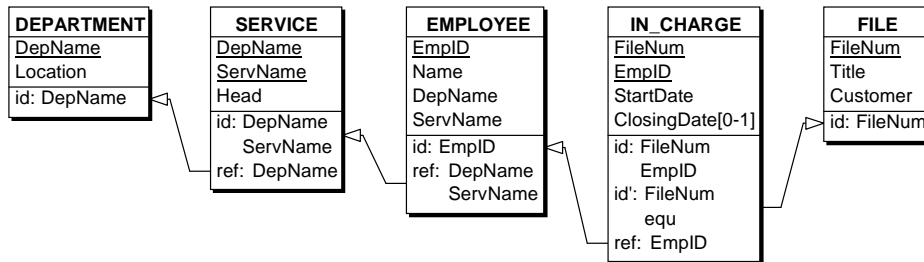


Figure Ex-1: A collection of standard foreign keys.

Exercise 2. Distributing water

A water distribution network is made up of nodes and pipes linking nodes in a directed tree structure. The fluid flows from the root node to the leaf nodes. In a pipe, it flows from the source node to the sink node. The pipes attached to a common source node are uniquely numbered. Among the outgoing pipes of each source node, one is considered its main pipe (Ex-2).

Derive a conceptual schema from the following relational schema that attempts to express this application domain.

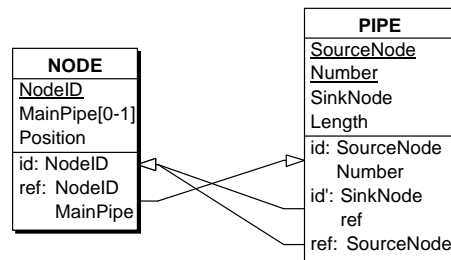


Figure Ex-2: Partly reciprocal foreign keys.

Exercise 3. Children and the social security

In a social security system, children depend on parents (who are members), and are associated with accounts, as expressed in the relational schema Ex-3. Derive a correct conceptual schema from this schema.

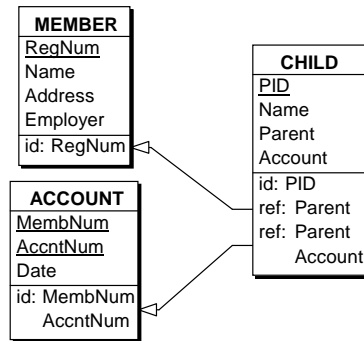


Figure Ex-3: Nothing missing?

Exercise 4. Offering products and assigning orders

Propose a conceptual schema equivalent to the following schema (Ex-4), recovered from a set of flat files.

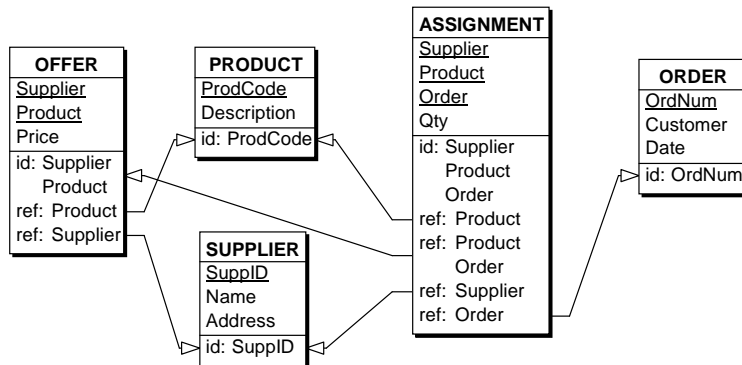


Figure Ex-4: Nothing in excess?

Exercise 5. Ordering in quantities

An order is correct if it corresponds to an available supply, that is, if there exists a supplier that can supply the item ordered in the quantity specified in the order. The table SUPPLY gives, for each supplier and each item it can ship, the prices for increasing quantities (e.g., for 1 unit, for (2 to) 5 units, for (6 to) 10 units, and so forth). The database that records these facts is represented in Ex-5. Derive an equivalent conceptual schema.

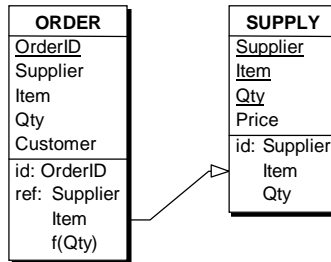


Figure Ex-5: Another computed foreign key.

Exercise 6. *Delivering . . .*

Give a correct interpretation of the inclusion constraint of schema Ex-6.

Exercise 7. . . . and shipping

The schema Ex-7 is an excerpts of the structure of a large relational database for spare part management that has been reengineered in a car manufacturing company.

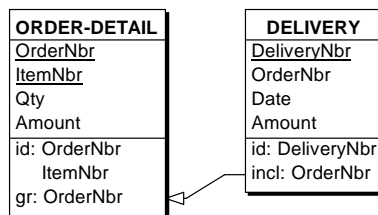


Figure Ex-6: An inclusion constraint.

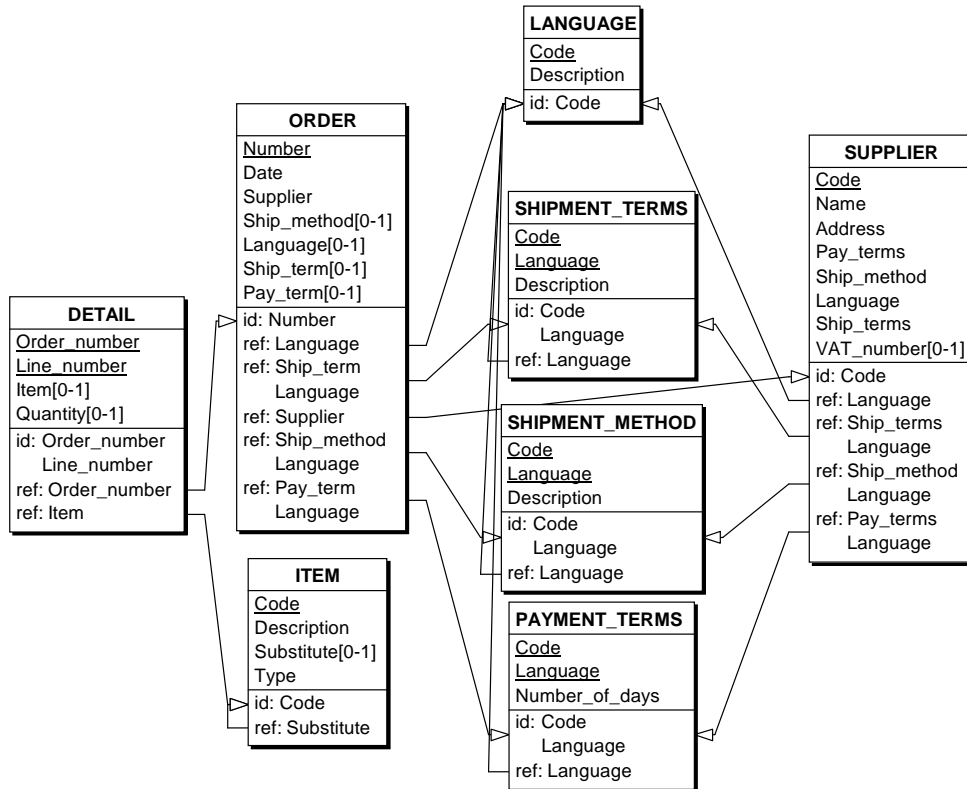


Figure Ex-7: Languages are everywhere.

List of the foreign keys:

DETAIL.Order_Number → ORDER	SHIPMENT_TERMS.Language → LANGUAGE
DETAIL.Item → ITEM	SHIPMENT_METHOD.Language → LANGUAGE
ITEM.Substitute → ITEM	PAYMENT_TERMS.Language → LANGUAGE
ORDER.Ship_term → SHIPMENT_TERMS	SUPPLIER.Ship_term → SHIPMENT_TERMS
ORDER.Ship_method → SHIPMENT_METHOD	SUPPLIER.Ship_method → SHIPMENT_METHOD
ORDER.Pay_term → PAYMENT_TERMS	SUPPLIER.Pay_term → PAYMENT_TERMS
ORDER.Language → LANGUAGE	SUPPLIER.Language → LANGUAGE
ORDER.Supplier → SUPPLIER	

Exercise 8. More about non-minimal foreign keys

Prove that the schemas Ex-8 are equivalent. *Hint:* the left side schema is that of Figure 34. In addition, refer also to the theory of normalization (notably, 3NF against BCNF).

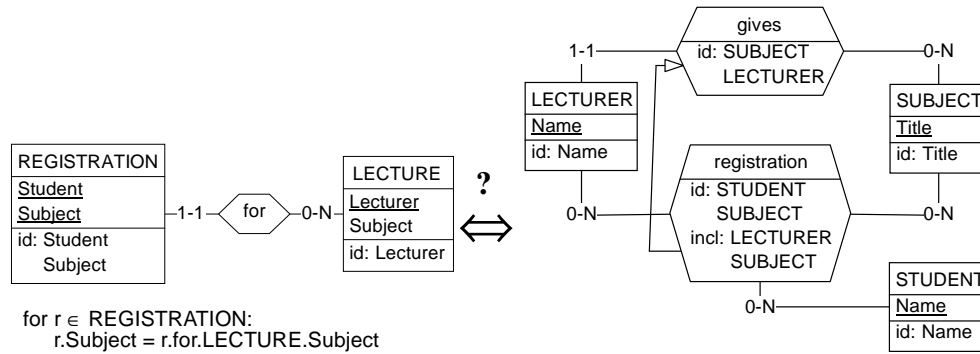


Figure Ex-8: Are these schemas equivalent?

Exercise 9. About partially reciprocal foreign keys

The schema Ex-9 is proposed as an interpretation of the schema of Figure 35. Evaluate the correctness of this interpretation.

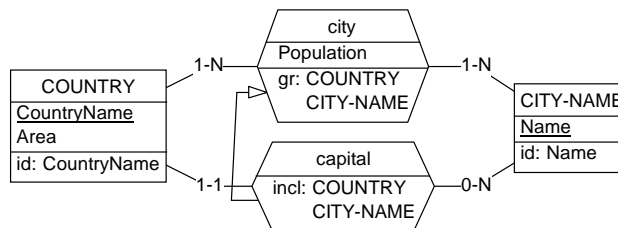


Figure Ex-9: Is this schema equivalent to the schema of Figure 35?

Exercise 10. Normalizing a legacy relational table

The Data Structure Extraction phase has extracted the table structure (Ex-10, left) accompanied with a set of functional dependencies.

Since the table is not in 3NF, it has been decomposed into normalized components that we want to conceptualize (Ex-10, right). However, the resulting schema is not quite equivalent to the source database. Indeed, the tables BOND_LEVEL and CUST_CONTACT include historical data about respectively the price evolution of the bonds and on the successive addresses of the customers. Therefore, the foreign keys ORDER.(Bond,DateSold) and ORDER.(Customer,DateSold) are *temporal foreign keys*. Considering this fact, and observing that the foreign keys share an attribute, propose a conceptual schema for this relational schema.

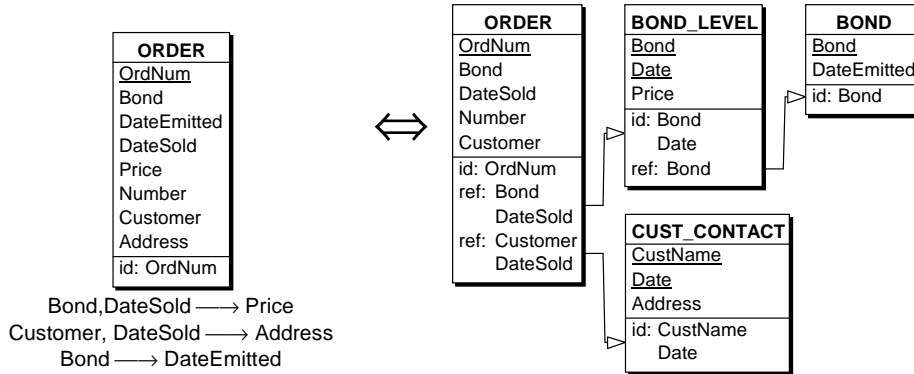


Figure Ex-10: Interpreting a normalized relational schema.

Exercise 11. Programmers write programs

The schema Ex-11 includes some nice non-standard and complex foreign keys.

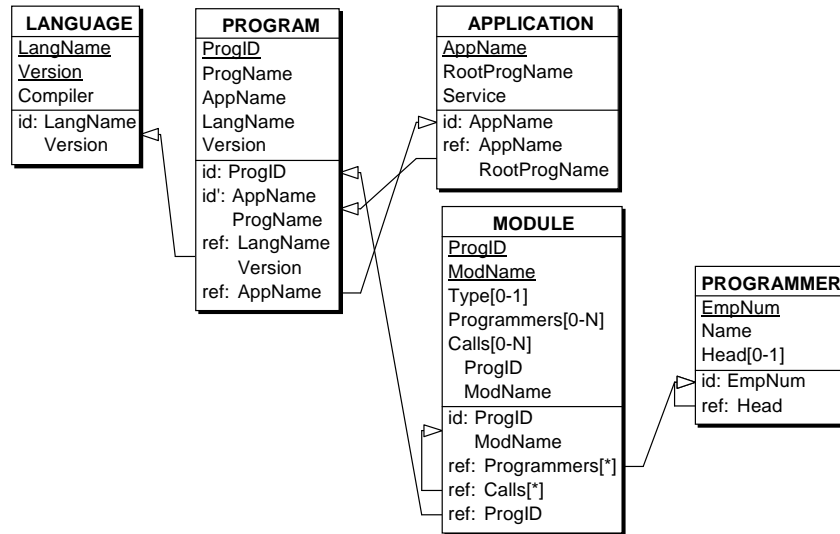


Figure Ex-11: A bunch of interesting foreign key patterns.

Exercise 12. Actual keys, at last!

This fragment is an excerpt from a key management application written in MS Access for a teaching and research department. To receive a copy of key (for an office, a library, a room, a desk, etc.) an employee must have access to an account. He can be attributed one copy only

for each key. Propose a conceptual schema for this database fragment.

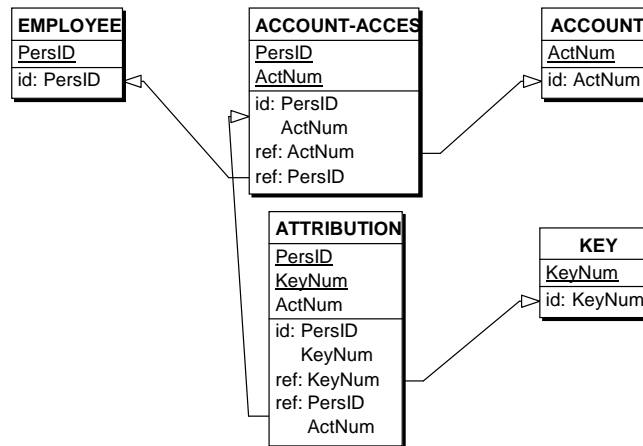


Figure Ex-12: Attributing office keys to employees

Exercise 13. Inclusion constraints in the country

In Figure Ex-13, a value of the field DESCRIPTION references a sequence of text lines, each in a different language, providing the description of a hotel. Same for ACCESS, giving access information. Find the underlying semantics of this schema.

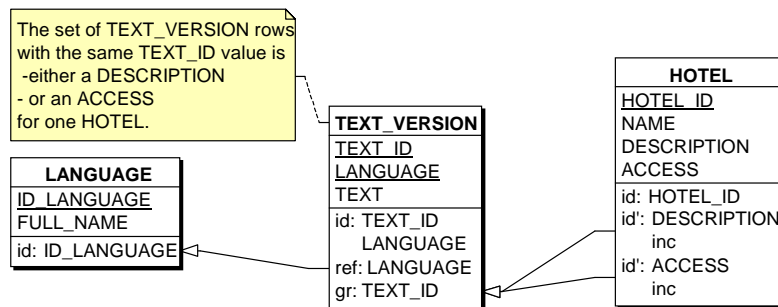


Figure Ex-13: Express the semantics of inclusion constraints.

Exercise 14. Families

The schema of Figure 14 is intended to describe evolving family relationships among persons. Derive from it a correct conceptual schema.

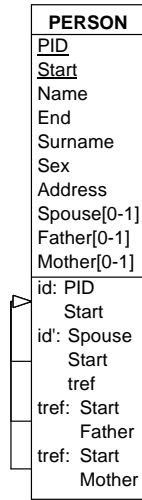


Figure Ex-14: A family structure

Exercise 15. Deriving generic rules

All the interpretation rules described in this chapter have been illustrated through actual examples. This approach, though attractive, could make it difficult to identify other similar problem patterns and to solve them. Hence the idea to provide abstract problem/solution patterns instead.

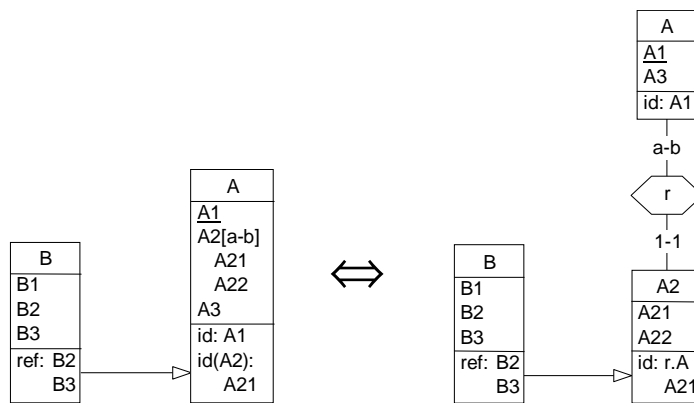


Figure Ex-15: Abstraction of the Hierarchical foreign key to a multivalued attribute rule of Figure 18. [a-b] denotes any valid cardinality range.

Figure Ex-15. shows how the interpretation of the concrete example of Figure 18 can be

generalized into an abstract rule that is easier to apply. Note that the constructs that play no role in the interpretation, such as the identifier of SHIPMENT, have been ignored. Propose a similar abstraction for each of the most common interpretation rules of this chapter.

