On Repairing Referential Integrity Constraints in Relational Databases

Raji Ghawi(✉)

Bavarian School of Public Policy, Technical University of Munich,
Richard-Wagner-Straße. 1, 80333 Munich, Germany
raji.ghawi@tum.de

Abstract. Integrity constraints (ICs) are semantic conditions that a database should satisfy in order to be in a consistent state. Typically, ICs are declared with the database schema and enforced by the database management system (DBMS). However, in practice, ICs may not be specified to the DBMS along with the schema, this is considered a bad database design and may lead to many problems such as inconsistency and anomalies. In this paper, we present a method to identify and repair missing referential integrity constraints (foreign keys). Our method comprises three steps of verification of candidate foreign keys: data-based, model-based, and brute-force.

Keywords: Relational databases · Integrity constraints · SQL · Validation · Verification

1 Introduction

Integrity constraints capture an important aspect of every database application. They are semantic conditions that a database should satisfy in order to be an appropriate model of reality [6]. There are several types of integrity constraints including: primary keys, functional dependencies, and referential integrity constraints (known as foreign keys). These constraints are derived from the semantics of the data and of the miniworld it represents. It is the responsibility of the database designers to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced [13]. A common assumption in data management is that databases can be kept consistent, that is, satisfying certain desirable integrity constraints. In practice, and for many reasons[1], a database may not satisfy those integrity constraints, and for that reason it is said to be *inconsistent*.

Inconsistency is an undesirable property for a database. Therefore, as a database is subject to updates, it should be kept consistent. This goal can be achieved in several ways. One of them consists in declaring the ICs together with the schema, thus, the DBMS will take care of keeping the database consistent,

[1] For example, when merging data from different sources.

S. Kozielski et al. (Eds.): BDAS 2019, CCIS 1018, pp. 82–96, 2019.
https://doi.org/10.1007/978-3-030-19093-4_7

by rejecting transactions that may lead to a violation of the ICs [6]. Another possibility is the use of triggers or active rules that are created by the user and stored in the database. They react to updates of the database by notifying a violation of an IC, rejecting a violating update, or compensating the update with additional updates that restore consistency. Another common alternative consists of keeping the ICs satisfied through the application programs that access and modify the database, i.e., from the transactional side. However, the correctness of triggers or application programs with respect to ensuring database consistency is not guaranteed by the DBMS.

In practice, for many reasons, integrity constraints *are not specified* to the DBMS along with the database schema. In particular, foreign keys could be missing because of (1) lack of support for checking foreign key constraints in the host system, (2) fear that checking such constraints would impede database performance, or (3) lack of database knowledge within the development team. Absence of integrity constraints specification could be considered a bad database design and may lead to many problems such as inconsistency, and anomalies. Anomalies may cause redundancy (during insertion or modification), accidental loss of information (during deletion), waste of storage space, and generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship.

In this paper, we address the problem of *absence of referential integrity constraints* specification, that is, when integrity constraints are not specified to the DBMS. We present a method to identify and repair missing foreign keys in a relational database. In this method, we first identify candidate foreign keys, then we conduct a thorough validation process of those candidates. The validation process comprises three types of verification, namely: data-based, model-based and brute-force verification. The objective of this process is to find valid foreign keys such that they are then specified to, and enforced by the DBMS.

The paper is organized as follows. Section 2 reviews related works. We present some preliminaries and the problem definition in Sect. 3. Then, we give an overview of the solution in Sect. 4 where we discuss identification and validation of candidate foreign keys. Then, we present the three verification steps: data-based (Sect. 5), brute-force (Sect. 6), and model-based (Sect. 7).

2 Related Work

A considerable amount of research has been done in the area of repairing inconsistent databases. Some works focused on *data cleaning* techniques to cleanse the database from data that participates in the violation of the ICs (see [20] for an overview). Other works have addressed the problem of *Consistent Query Answering* (CQA), that is, computing consistent answers over inconsistent database (see [2,5,9,10,22]). Such works rely on the notion of *database repair*, which is a new database instance that is consistent with respect to the ICs, and minimally differs from the inconsistent database at hand. Some of these works are implemented in prototypes systems, such as Hippo [12], and ConQuer [14]. However, these works

assume that the integrity constraints are correctly specified; and the problem is in the data. In contrast, in our work, we assume that the integrity constraints are not correctly specified to the DBMS. We focus on identifying and repairing missing referential integrity constraints.

Another related area is the discovery of inclusion dependencies in a given database. Many papers have been addressed approximate and exact discovery of inclusion dependencies [15,21,23], and different discovery strategies have been proposed, based on inverted indices [17], sort-merge joins [4], and distributed data aggregation [16]. Research has also devised algorithms for exact discovery of n-ary inclusion dependencies, such as Mind [17] and Binder [19], and for approximate discovery, such as Faida [15]. These works assume data to be complete or consistent, hence proposed discovery methods are mainly based on data.

What distinguish our present work is that we address a two-fold problem. First, foreign keys are missing, therefore their discovery is needed. Second, data itself is not assumed to be consistent or complete, therefore repairing the database instance is also needed. Thus, our proposed solution combines repairing the database schema (specifying valid foreign keys), and repairing the database instance (removing dangling values, when necessary).

In literature, various kinds of repair semantics have been proposed, based on database operations used, and the type of constraints/dependencies. For inclusion dependencies, three types of repairs are possible in general:

1. Tuple-insertion-based repairs [8] – New tuples are inserted in order to satisfy violated constraints. This repair semantics is applied when the database at hand is considered to be incomplete and is then completed via additional tuple insertions. Repairing inclusion dependencies with tuple-insertion requires that values have to be invented for them. This leads to possibly infinitely many repairs. Moreover, value inventions are in general non-deterministic, and complex to handle [6]; and they can lead to the undecidability of consistent query answering [8].
2. Tuple-deletion-based repairs [11] – Tuples that violate constraints are deleted. This class of repairs assume that the database instance at hand is closed, and no insertions of new tuples are accepted [6], therefore integrity-restoration actions are limited to tuple deletions. A good reason for adopting this kind of repair semantics is that, when we insert tuples to enforce inclusion dependencies, we may have to invent data values for the inserted tuples.
3. Null-insertion based repairs [7,18] – Under this repair semantics, inclusion dependencies are repaired by insertions of null values to restore consistency. Null values can also be used for value invention as required by tuple-insertion-based repairs of referential ICs.

In this present paper, we adopt tuple-deletion-based repairs in the first place, but we also consider null-insertion-based repairs are possible and valid in our case study. However, we do not address tuple-insertion repairs as we assume the database instance is closed, in the sense that no insertions of new tuples are allowed, because they require value invention.

3 Preliminaries and Problem Definition

In relational databases, referential integrity is a property of data stating that references within it are valid. It requires every value of one attribute of a relation to exist as a value of another attribute in a different (or the same) relation. Formally, referential integrity constraints are expressed in terms of inclusion dependencies [1]. Let R be a relation schema and $X = A_1, \cdots, A_n$ a sequence of attributes (possibly with repeats) from R. For an instance I of R, the projection of I onto the sequence X, denoted $I[X]$, is the n-ary relation $I[X] = \{\langle t(A_1), \cdots, t(A_n)\rangle \mid t \in I\}$.

Let $\boldsymbol{R}$ be a relational schema. An inclusion dependency (IND) over $\boldsymbol{R}$ is an expression of the form $\sigma \doteq R[A_1, ..., A_m] \subseteq S[B_1, ..., B_m]$ where: R, and S are (possibly identical) relation names in $\boldsymbol{R}$, $A_1, ..., A_m$ is a sequence of distinct attributes of R, and $B_1, ..., B_m$ is a sequence of distinct attributes of S. An instance $\boldsymbol{I}$ of $\boldsymbol{R}$ satisfies σ, denoted $I \models \sigma$, if $\boldsymbol{I}(R)[A_1, ..., A_m] \subseteq \boldsymbol{I}(S)[B_1, ..., B_m]$. The left-hand side of an IND is referred to as *dependent* attribute(s) and the right-hand side as *referenced* attribute(s) [19]. Both of these attribute sequences must be of the same size m. An IND is said to be unary if m = 1, otherwise it is n-ary. Notice that we do not consider any semantics for null values as they do not contribute to INDs: we simply ignore them.

In this study, we have a relational database that has a large number of tables. This database is poorly designed and suffers from several design problems. However, it is running and a complex application operates on top of it, thus a redesign from scratch is not possible. Our task is to repair the database and get rid of the design issues. Given a relational database that is accessed by an application program, and has a large number of tables; assume the following: (1) all tables have primary keys (declared with the schema), (2) most of tables do not have declared foreign keys, (3) some tables have data records while others do not, (4) all data records are identified using *universally unique identifier*s (UUID); the problem is to identify missing foreign keys and declare them in the database schema such that the integrity constraints are enforced by the DBMS.

Actually, in our case-study, the reasons of inconsistency were mainly the lack of database knowledge within the development team, and inappropriate use of ORM (Object-Relational Mapping) technique [3]. As consequence, integrity constraints were enforced by the application code only, without DBMS support; hence when some records are deleted via the application, their referencing records are not removed, leaving the database in an inconsistent state.

4 Solution Outlines

Identification of Foreign Keys. Our legacy database consists of a large number of tables. All the tables have primary keys (PKs). However, only a subset of the foreign keys are defined at the database level, while the majority of the relationships among the tables are not defined in terms of foreign keys. Thus, our major objective is to identify the foreign keys and define them at the database

level in order to enforce the referential integrity constraints. Our method goes as follows. First, we conduct an exhaustive identification of candidate foreign keys. Then, we validate candidate foreign keys using both the data and the application code. Finally, we define valid foreign keys explicitly in the database, and correct invalid foreign keys.

The first step is to obtain an exhaustive list of candidate foreign keys. The result is a table of four columns: table, column, reference table, and reference columns. Table 1 shows a sample of such candidate foreign keys. The identification is performed manually by first investigating the columns names. According to the naming convention used in our database, if a column name contains a name of primary key of another table, it is probably a foreign key that references this primary key. Also, if the column name contains 'parent' this indicates that the column is probably a foreign key that references (the primary key of) its own table. When necessary, the application code is also examined to confirm the candidacy of each foreign key. By the end of this process, we obtain a list of candidate foreign keys.

Table 1. Examples of candidate foreign keys

Table	Column	Reference table	Reference column
`department`	`department_branch_id`	`branch`	`branch_id`
`department`	`department_parent`	`department`	`department_id`
`department_section`	`department_section_department_id`	`department`	`department_id`

Validation of Candidate Foreign Keys. Identified candidate foreign keys need to be validated. Such a validation is conducted over several steps. First, if the owner table of a foreign key has data records ($\boldsymbol{I}(R) \neq \emptyset$), then we use *data-based verification* where data records are used to verify the validity of the foreign key. That is, we check whether the current instance satisfy the constraint or not. Candidate foreign keys that fail in the data-based verification step are subject to another verification step called *brute-force verification*, to find their potential reference tables and/or their known values that have no reference (dangling values.) However, if the owner table is empty ($\boldsymbol{I}(R) = \emptyset$), we use a *model-based verification* where the application program associated with database is used to validate the foreign key (Sect. 7). In any case, candidate foreign keys that successfully pass the data-based verification or model-based verification are considered valid, and thus are declared explicitly in the database schema. Foreign keys that are verified using brute-force step are corrected manually, and dangling values are removed as we will see in Sect. 6. Figure 1 illustrates the overall process of validation of candidate foreign keys.

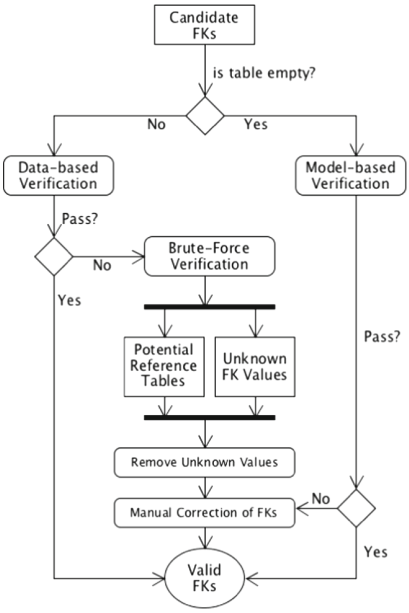


Fig. 1. Process of validation of candidate foreign keys

5 Data-Based Verification

The idea behind data-based verification is the following: a candidate foreign key is valid if the database instance satisfies this foreign key constraint. Formally, let $\boldsymbol{R}$ be a database schema with an instance $\boldsymbol{I}$, and let $\sigma \doteq R[A] \subseteq S[B]$ be a candidate foreign key over $\boldsymbol{R}$, this candidate foreign key σ is considered valid if the current instance $\boldsymbol{I}$ satisfies it ($I \models \sigma$.) In order to verify whether a candidate foreign key $\sigma \doteq R[A] \subseteq S[B]$ is satisfied, the sets of distinct values of the referencing column $R[A]$ and the referenced column $S[B]$ (denoted $\boldsymbol{I}(R)[A]$ and $\boldsymbol{I}(S)[B]$, respectively) are extracted from the database instance $\boldsymbol{I}$. Based on the definition of inclusion dependency, if $\boldsymbol{I}(R)[A] \subseteq \boldsymbol{I}(S)[B]$, then the referential integrity constraint holds, and $\sigma \doteq R[A] \subseteq S[B]$ is indeed a valid foreign key.

The successfully validated foreign keys, are then defined explicitly in the database in order to enforce the referential integrity constraint. This could be done using the following SQL script:

```
ALTER TABLE R ADD FOREIGN KEY (A) REFERENCES S(B);
```

Once a foreign key is defined with the schema, it will be enforced by the DBMS; that is, if a user attempts to insert a tuple that violates the referential IC (e.g., a tuple with a value $t[A] \notin \boldsymbol{I}(S)[B]$), the DBMS will reject this insertion, keeping the constraint satisfied and the database consistent.

Example 1. Consider a database schema with three relations $P_1(A, B)$, $P_2(C, D)$ and $P_3(E)$, and two candidate foreign keys $\sigma_1 \doteq P_1[B] \subseteq P_2[C]$ and $\sigma_2 \doteq$

$P_2[D] \subseteq P_3[E]$. Consider an instance I' such that $I'(P_1) = \{(a,b),(c,d)\}$, $I'(P_2) = \{(b,e),(d,f),(g,h)\}$ and $I'(P_3) = \{(e),(h),(i)\}$. Then we have:

$$\{b,d\} = I'(P_1)[B] \subseteq I'(P_2)[C] = \{b,d,g\}$$
$$\{e,f,h\} = I'(P_2)[D] \nsubseteq I'(P_3)[E] = \{e,h,i\}$$

This means that the instance I' *satisfies* the constraint $\sigma_1 \doteq P_1[B] \subseteq P_2[C]$, but not the constraint $\sigma_2 \doteq P_2[D] \subseteq P_3[E]$. In this example, the first candidate foreign key is *valid* and therefore is declared with the schema, whereas the second candidate is *invalid* and should be verified again with brute-force verification.

6 Brute-Force Verification

Candidate foreign keys that do not successfully pass the data-based verification (i.e., do not satisfy the condition $\boldsymbol{I}(R)[A] \subseteq \boldsymbol{I}(S)[B]$) are subject to another inspection called brute-force verification. The idea behind this test is the following: the values of a candidate foreign key are compared with the values of the primary keys of all tables in the database. When those values *match* (possibly partially) the values of a primary key of a table, this table is considered a *potential reference table* for the candidate foreign key. If there is no match with any table, then such unmatched values are considered unknown and should be removed to keep the consistency of the database. In this section, we present brute-force verification as an algorithm which takes a candidate foreign key as input, and returns a set of potential referenced tables and a set of unknown values as output. According to the outcomes of the algorithm, we present possible solutions for different cases.

When the condition $\boldsymbol{I}(R)[A] \subseteq \boldsymbol{I}(S)[B]$ is not satisfied, this means that there is at least an element $\widehat{a} \in \boldsymbol{I}(R)[A]$ such that $\widehat{a} \notin \boldsymbol{I}(S)[B]$. Let us denote the set of such elements $\widehat{A}$, then we have:

$$\widehat{A} = \{a \mid a \in \boldsymbol{I}(R)[A] \wedge a \notin \boldsymbol{I}(S)[B]\} = \boldsymbol{I}(R)[A] - \boldsymbol{I}(S)[B]$$

In some sense, this set contains incorrect tuples of $I(R)[A]$, i.e., tuples that violate the integrity constraint; thus we want to find their correct references if any. However, besides those incorrect tuples, $\boldsymbol{I}(R)[A]$ could contain other correct tuples that reference $S[B]$. Let us denote the set of correct tuples as A^*, then:

$$A^* = \{a \mid a \in \boldsymbol{I}(R)[A] \wedge a \in \boldsymbol{I}(S)[B]\} = \boldsymbol{I}(R)[A] \cap \boldsymbol{I}(S)[B]$$

Note that $\boldsymbol{I}(R)[A] = \widehat{A} \cup A^*$. The set A^* could be empty, and in this case there is no data-based evidence that S is a potential referenced table for the candidate foreign key. However, if this set A^* is not empty, then the table S remains a potential reference table for $R[A]$.

Let T be a relation schema, and let K_T be the primary key of T, the set of values of this primary key in the database instance is: $\boldsymbol{I}(T)[K_T]$.

The brute-force verification is depicted in Algorithm 1, that takes an input a candidate foreign key $\sigma \doteq R[A] \subseteq S[B]$, and returns a set $\widehat{A}$ of dangling values of $R[A]$, and a set Z of potential referenced tables.

First, the set Z is initially empty, whereas the set A^* equals the intersection $\boldsymbol{I}(R)[A] \cap \boldsymbol{I}(S)[B]$. If this set is not empty, then S is indeed a potential reference table, thus it is added to Z. Then, the set of dangling values $\widehat{A}$ initially equals the difference $\boldsymbol{I}(R)[A] - \boldsymbol{I}(S)[B]$. For every table T in the database $\boldsymbol{R}$, we extract the set of values of T's primary key: $\boldsymbol{I}(T)[K_T]$, and find Q the intersection of this set with $\widehat{A}$. If this intersection Q is empty, the loop continues. But if it is not empty, this means that we found a potential referenced table T for the candidate foreign key $R[A]$, therefore, we append this table T to the set of potential referenced tables Z, and we remove the set Q from $\widehat{A}$, because this subset is not considered dangling any more, since we found its originating table T. Finally, the algorithm returns the set of potential referenced tables Z that have been found, and the set of remaining dangling values $\widehat{A}$.

Algorithm 1. Brute-Force Verification

Input: a candidate foreign key ($\sigma \doteq R[A] \subseteq S[B]$)
Output: a set of potential referenced tables Z and a set of dangling values $\widehat{A}$

1: $Z \leftarrow \emptyset$
2: $A^* \leftarrow \boldsymbol{I}(R)[A] \cap \boldsymbol{I}(S)[B]$ ▷ correctly referenced values
3: **if** $A^* \neq \emptyset$ **then**
4: $\quad Z \leftarrow Z \cup \{S\}$
5: $\widehat{A} \leftarrow \boldsymbol{I}(R)[A] - \boldsymbol{I}(S)[B]$ ▷ dangling values
6: **for** $T \in \boldsymbol{R}$ **do**
7: $\quad$ **find** $\boldsymbol{I}(T)[K_T]$ ▷ extract the values of T's pk
8: $\quad Q \leftarrow \widehat{A} \cap \boldsymbol{I}(T)[K_T]$
9: $\quad$ **if** $Q \neq \emptyset$ **then**
10: $\quad\quad \widehat{A} \leftarrow \widehat{A} - Q$
11: $\quad\quad Z \leftarrow Z \cup \{T\}$
12: **return** Z, $\widehat{A}$

Algorithm 1 is executed for every candidate foreign key $\sigma \doteq R[A] \subseteq S[B]$, that has failed in the data-based verification. Based on the outcomes of the algorithm, we distinguish three cases:

- **Case 1.** $Z = \emptyset$, the FK has no potential referenced tables at all.
- **Case 2.** $|Z| = 1$, the FK has one potential referenced table.
- **Case 3.** $|Z| > 1$, the FK has more than one potential referenced tables.

6.1 Case 1. No Potential Referenced Tables

The first case is the easiest one to solve. It means that we are unable to find an alternative table that could be referenced by $R[A]$. This proves that the

originally candidate referenced table S should be actually the correct referenced table, even if there is no data-based evidence. Anyway, the problem in this case is only with dangling values $\widehat{A}$. Therefore, the solution is simply to remove the dangling values $\widehat{A}$, and declare the foreign key with the schema. The solution can be expressed in SQL as shown in Listing 1.1.

Listing 1.1. Solution of Case 1

```
-- Remove dangling values
DELETE FROM R WHERE A IN Â;

-- Define the foreign key
ALTER TABLE R
ADD FOREIGN KEY (A) REFERENCES S(B);
```

Example 2. Consider a schema with three relations $P_1(A, B)$, $P_2(C)$ and $P_3(D)$ and a candidate foreign key $\sigma \doteq P_1[B] \subseteq P_2[C]$. Consider $I'(P_1) = \{(a, b)\}$, $I'(P_2) = \{(c)\}$ and $I'(P_3) = \{(d)\}$. Clearly, the instance I' does not satisfy σ and the candidate foreign key fails the data-based verification. With brute-force verification, we have $Z = \emptyset$ and $\widehat{B} = \{b\}$. Therefore, the solution is to admit the candidate foreign key σ as valid and declare it with the schema, and to remove the dangling value b, either by removing the entire tuple (a, b), or setting b to null, $I'(P_1) = \{(a, NULL)\}$.

6.2 Case 2. One Potential Referenced Table

The second case is also easy to solve, because the only potential referenced table found in Z must be the correct referenced table that we are looking for. This found table could be the same candidate referenced table S, or another table. Anyway, let us denote it U. The solution is simply to correct the foreign key to be: $\sigma' \doteq R[A] \subseteq U[K_U]$ where K_U is the primary key of U. This implies the removal of the dangling values $\widehat{A}$, and defining the correct foreign key explicitly:

Listing 1.2. Solution of Case 2

```
-- Remove dangling values
DELETE FROM R WHERE A IN Â;

-- Define the correct foreign key
ALTER TABLE R
ADD FOREIGN KEY (A) REFERENCES U(K_U);
```

Example 3. Consider a schema with three relations $P_1(A, B)$, $P_2(C)$ and $P_3(D)$ and a candidate foreign key $\sigma \doteq P_1[B] \subseteq P_2[C]$. Consider $I'(P_1) = \{(a, b), (c, d)\}$, $I'(P_2) = \{(e), (f)\}$ and $I'(P_3) = \{(b), (d)\}$. Clearly, $I' \not\models \sigma$ and the candidate foreign key fails the data-based verification. With brute-force verification, we have $Z = \{P_3\}$ and $\widehat{B} = \emptyset$. The solution here is to correct the candidate foreign key to be: $\sigma' \doteq P_1[B] \subseteq P_3[D]$.

6.3 Case 3. Many Potential Referenced Tables

The third case is tricky, as there are many different potential referenced tables $Z = \{U_1, U_2, \cdots, U_n\}$ that are referenced by $R[A]$.

Example 4. Consider a schema with three relations $P_1(A, B)$, $P_2(C)$ and $P_3(D)$ and a candidate foreign key $\sigma \doteq P_1[B] \subseteq P_2[C]$. Consider $I'(P_1) = \{(a, b), (c, d)\}$, $I'(P_2) = \{(b), (e)\}$ and $I'(P_3) = \{(d), (f)\}$ (Fig. 2). Clearly, $I' \not\models \sigma$ and the candidate foreign key fails the data-based verification. With brute-force verification, we have $Z = \{P_2, P_3\}$ and $\widehat{B} = \emptyset$.

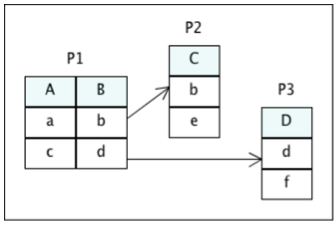


Fig. 2. Database schema of Example 4

This case requires a careful inspection of the database schema and the application code by the database designer in order to know exactly the reason of this situation, and to figure out an appropriate solution. For instance, we could distinguish two cases:

Case 3-1. For instance, it could be the case that only one of these tables is the correct one, and the others are not. Here, the solution is similar to the solution of Case 2 above, but with an additional removal of erroneous values. That is, let us consider that U_1 is the correct table, and the other tables $U_2, \cdots, U_n$ should not be referenced by $R[A]$.

In this case, the only valid foreign key is: $\sigma_1 \doteq R[A] \subseteq U_1[K_{U_1}]$ which should be declared in the schema. In addition to removing dangling values $\widehat{A}$ from $R[A]$, we need also to remove the erroneous values that reference any of $U_2, \cdots, U_n$ tables. This solution can be expressed in SQL as shown in Listing 1.3.

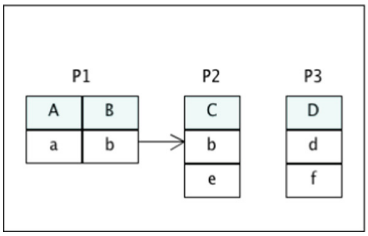
Example 5. In the previous example (Example 4), we found two potential referenced tables $Z = \{P_2, P_3\}$. If we consider P_2 is the correct reference table, then the valid foreign key is $\sigma \doteq P_1[B] \subseteq P_2[C]$ (Fig. 3-a), and the erroneous data record is (c, d). But if we consider P_3 is the correct reference table, then the valid foreign key is $\sigma' \doteq P_1[B] \subseteq P_3[D]$ (Fig. 3-b), and the erroneous data record is (a, b).

Listing 1.3. Solution of Case 3-1

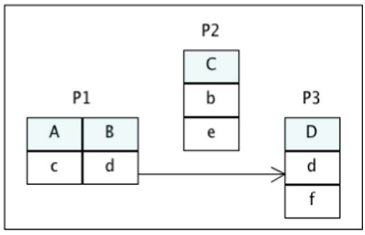
```
-- Remove dangling values
DELETE FROM R WHERE A IN Â;

-- Remove erroneous values
DELETE FROM R
WHERE A IN (SELECT K_U2 FROM U2);
...
DELETE FROM R
WHERE A IN (SELECT K_Un FROM Un);

-- Define the correct foreign key
ALTER TABLE R
ADD FOREIGN KEY (A) REFERENCES U1(K_U1);
```



(a) P_2 is the correct reference table



(b) P_3 is the correct reference table

Fig. 3. Solutions for Case 3-1 (Example 5)

Case 3-2. Another possibility, is that all the potential tables $Z = \{U_1, U_2, \cdots, U_n\}$ are considered correct and should be indeed referenced by $R[A]$. This case contradicts with the basic principles of database design, and must be solved radically. Based on the semantics of the relations, two solutions are possible:

Case 3-2, Solution 1. Replace the column A in R, by n columns $A_1, A_2, \cdots, A_n$ that reference $U_1, U_2, \cdots, U_n$, respectively. This means, we replace the candidate foreign key $\sigma \doteq R[A] \subseteq S[B]$ by the following foreign keys:

$$\begin{gathered}\sigma_1 \doteq R[A_1] \subseteq U_1[K_{U_1}] \\ \sigma_2 \doteq R[A_2] \subseteq U_2[K_{U_2}] \\ \cdots \\ \sigma_n \doteq R[A_n] \subseteq U_n[K_{U_n}]\end{gathered}$$

This solution implies that we first delete dangling values, then we add new columns $A_1, A_2, \cdots, A_n$ to R. Now, we should update R such that each column A_i contains the values of A that reference $U_i[K_{U_i}]$, for $i = 1, \cdots, n$. Finally, we drop the column A, and define the foreign keys. This solution can be expressed using SQL as shown in Listing 1.4.

Example 6. In Example 4, we found two potential referenced tables $Z = \{P_2, P_3\}$. If both tables are considered correct and should be referenced by $P_1[B]$, then the solution presented above is to replace the column B by two columns B_1 and B_2, such that we have two foreign keys instead of one, namely: $P_1[B_1] \subseteq P_2[C]$, and $P_1[B_2] \subseteq P_3[D]$ (Fig. 4).

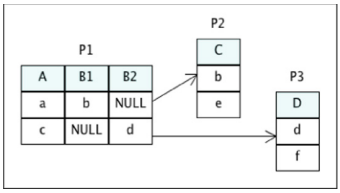


Fig. 4. Solution 1 of Case 3-2 (Example 6)

Listing 1.4. Solution 1 of Case 3-2

```
-- Remove dangling values
DELETE FROM R WHERE A IN Â;

-- Add new columns to R
ALTER TABLE R
   ADD COLUMN A_1 ...,
   ...
   ADD COLUMN A_n ...;

-- Migrate data values from A to the new columns
UPDATE R SET A_1 = A
WHERE A IN (SELECT K_U1 FROM U_1);
...
UPDATE R SET A_n = A
WHERE A IN (SELECT K_Un FROM U_n);

-- Drop column A
ALTER TABLE R DROP COLUMN A;

-- Finally, define the correct foreign keys
ALTER TABLE R
  ADD FOREIGN KEY (A_1) REFERENCES U_1(K_U1),
  ...
  ADD FOREIGN KEY (A_n) REFERENCES U_n(K_Un);
```

Case 3-2, Solution 2. Replace the whole table R by n new tables, $R_1, R_2, \cdots, R_n$, each of which will be referencing one of the referenced tables $U_1, U_2, \cdots, U_n$. In this case, each one of those tables R_i will contain a version of the column A_i that references $U_i[K_{U_i}]$. This means that we also replace the candidate foreign key $\sigma = R[A] \subseteq S[B]$ by the following foreign keys:

$$\begin{aligned} \sigma_1 &\doteq R_1[A_1] \subseteq U_1[K_{U_1}] \\ \sigma_2 &\doteq R_2[A_2] \subseteq U_2[K_{U_2}] \\ &\cdots \\ \sigma_n &\doteq R_n[A_n] \subseteq U_n[K_{U_n}] \end{aligned}$$

This solution implies that we first delete dangling values, then we create new tables $R_1, R_2, \cdots, R_n$, including the definition of foreign keys. Then, data records should be migrated from R to those new tables such that each new table R_i contains the records that reference his corresponding reference table U_i for $i = 1, \cdots, n$. Finally, table R can be dropped safely. This solution can be expressed using SQL code as shown in Listing 1.5.

Listing 1.5. Solution 2 of Case 3-2

```
-- Remove dangling values
DELETE FROM R WHERE A IN Â;

-- Create new tables, including defining correct foreign keys
CREATE TABLE R1 (
  ..., A1, ...
  FOREIGN KEY A1 REFERENCES U1(K_U1)
);
...
CREATE TABLE Rn (
  ..., An, ...
  FOREIGN KEY An REFERENCES Un(K_Un)
);

-- Migrate data from R to the new tables
INSERT INTO R1 (..., A1, ...)
SELECT ..., A, ... FROM R
WHERE A IN (SELECT K_U1 FROM U1);
...
INSERT INTO Rn (..., An, ...)
SELECT ..., A, ... FROM R
WHERE A IN (SELECT K_Un FROM Un);

-- Finally, drop table R
DROP TABLE R;
```

Example 7. In Example 4, we found two potential referenced tables $Z = \{P_2, P_3\}$. If both tables are considered correct and should be referenced by $P_1[B]$, then the solution presented above is to replace the table $P_1(A, B)$ by two tables $P_{1,1}(A_1, B_1)$ and $P_{1,2}(A_2, B_2)$, such that we have two foreign keys: $P_{1,1}[B_1] \subseteq P_2[C]$, and $P_{1,2}[B_2] \subseteq P_3[D]$, as shown in Fig. 5.

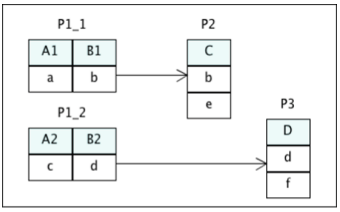


Fig. 5. Solution 2 of Case 3-2 (Example 7)

7 Model-Based Verification

This type of verification is applied for candidate foreign keys whose tables do not have data records, and it is based on the application program. In our case study, each database table has an associated model (Java bean). Model-based verification is based on the following idea: we inspect the model associated with owner table of the candidate foreign key. If this model has a reference to the model associated with referenced table of the foreign key, then we consider that the foreign key is valid. Formally, given a candidate foreign key $\sigma = R[A] \subseteq S[B]$, we inspect the models M_R and M_S associated with the tables R and S, respectively. If the model M_R contains a field a of type M_S, this means that the table R has a column C_a which is probably a foreign key references the table S associated with the model M_S.

8 Conclusion

In this paper, we have presented a method for repairing referential integrity constraints in relational databases. This method is applied when the constraints are not correctly specified in terms of foreign keys in the database schema. The method starts with identifying candidate foreign keys. Then, these candidates are verified using three types of verification: model-based, data-based, and brute-force verification. Valid foreign keys are declared with the database schema and, thus, are enforced by the DBMS to keep the database consistent.

Our method has been applied on a real-world database composed of 167 tables. We have identified 393 candidate foreign keys. Among them, there are 246 foreign keys whose tables have records, hence they are subject to data-based validation; whereas the remaining 147 foreign keys have empty tables, thus they are subject to model-based verification. Among the 246 foreign keys that have been verified using data-based validation, there are 232 that passed, while 14 foreign keys only have failed, thus they are subject to brute-force validation. As a result of brute-force validation, 4 FKs have no potential reference tables (case 1), while 9 FKs have one potential reference table (case 2), and only one FK has three potential reference tables (case 3). For all the 14 foreign keys there were dangling records in the range of 1 to 7 records per table (in total: 31 records). Among the 147 foreign keys that have been verified using model-based validation, only 4 FKs have failed the test and hence have been manually corrected.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Boston (1995). http://webdam.inria.fr/Alice/
2. Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proceedings of the 18th Symposium on Principles of Database Systems, PODS 1999, pp. 68–79. ACM, New York (1999)
3. Barry, D., Stanienda, T.: Solving the Java object storage problem. Computer **31**(11), 33–40 (1998)

4. Bauckmann, J., Leser, U., Naumann, F., Tietz, V.: Efficiently detecting inclusion dependencies. In: ICDE, pp. 1448–1450. IEEE Computer Society (2007)
5. Bertossi, L.: Consistent query answering in databases. SIGMOD Rec. **35**(2), 68–76 (2006)
6. Bertossi, L.: Database Repairing and Consistent Query Answering. Morgan & Claypool Publishers (2011)
7. Bravo, L., Bertossi, L.: Semantically correct query answers in the presence of null values. In: Grust, T., et al. (eds.) EDBT 2006. LNCS, vol. 4254, pp. 336–357. Springer, Heidelberg (2006). https://doi.org/10.1007/11896548_27
8. Calì, A., Lembo, D., Rosati, R.: On the decidability and complexity of query answering over inconsistent and incomplete databases. In: Proceedings of the 22nd Symposium on Principles of Database Systems, PODS 2003, pp. 260–271. ACM, New York (2003)
9. Chomicki, J.: Consistent query answering: opportunities and limitations. In: 17th International Workshop on Database and Expert Systems Applications (DEXA 2006), 4–8 September 2006, Krakow, Poland, pp. 527–531 (2006)
10. Chomicki, J.: Consistent query answering: five easy pieces. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 1–17. Springer, Heidelberg (2006). https://doi.org/10.1007/11965893_1
11. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Inf. Comput. **197**(1–2), 90–121 (2005)
12. Chomicki, J., Marcinkowski, J., Staworko, S.: Hippo: a system for computing consistent answers to a class of SQL queries. In: Bertino, E., et al. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 841–844. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24741-8_53
13. Elmasri, R., Navathe, S.: Fundamentals of Database Systems, 6th edn. Addison-Wesley Publishing Company, Boston (2010)
14. Fuxman, A., Fazli, E., Miller, R.J.: ConQuer: efficient management of inconsistent databases. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 155–166. ACM, New York (2005)
15. Kruse, S., et al.: Fast approximate discovery of inclusion dependencies. In: Datenbanksysteme für Business, Technologie und Web (BTW 2017) (2017)
16. Kruse, S., Papenbrock, T., Naumann, F.: Scaling out the discovery of inclusion dependencies. In: Datenbanksysteme für Business. Technologie und Web (BTW 2015), pp. 445–454. Gesellschaft für Informatik e.V, Bonn (2015)
17. Marchi, F.D., Lopes, S., Petit, J.M.: Unary and N-ary inclusion dependency discovery in relational databases. J. Intell. Inf. Syst. **32**(1), 53–73 (2009)
18. Molinaro, C., Greco, S.: Polynomial time queries over inconsistent databases with functional dependencies and foreign keys. Data Knowl. Eng. **69**(7), 709–722 (2010)
19. Papenbrock, T., Kruse, S., Quiané-Ruiz, J.A., Naumann, F.: Divide & conquer-based inclusion dependency discovery. VLDB Endow. **8**(7), 774–785 (2015)
20. Rahm, E., Do, H.H.: Data cleaning: problems and current approaches. IEEE Data Eng. Bull. **23**, 2000 (2000)
21. Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., Leser, U.: A machine learning approach to foreign key discovery. In: 12th International Workshop on the Web and Databases, WebDB 2009, Rhode Island, USA (2009)
22. Wijsen, J.: Consistent query answering under primary keys: a characterization of tractable queries. In: Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, 23–25 March 2009, Proceedings, pp. 42–52 (2009)
23. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: On multi-column foreign key discovery. VLDB Endow. **3**(1–2), 805–814 (2010)