



Holistic primary key and foreign key detection

Lan Jiang¹ · Felix Naumann¹

Received: 28 February 2019 / Revised: 20 May 2019 / Accepted: 21 May 2019 /

Published online: 10 June 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Primary keys (PKs) and foreign keys (FKs) are important elements of relational schemata in various applications, such as query optimization and data integration. However, in many cases, these constraints are unknown or not documented. Detecting them manually is time-consuming and even infeasible in large-scale datasets. We study the problem of discovering primary keys and foreign keys automatically and propose an algorithm to detect both, namely Holistic Primary Key and Foreign Key Detection (HoPF). PKs and FKs are subsets of the sets of unique column combinations (UCCs) and inclusion dependencies (INDs), respectively, for which efficient discovery algorithms are known. Using score functions, our approach is able to effectively extract the true PKs and FKs from the vast sets of valid UCCs and INDs. Several pruning rules are employed to speed up the procedure. We evaluate precision and recall on three benchmarks and two real-world datasets. The results show that our method is able to retrieve on average 88% of all primary keys, and 91% of all foreign keys. We compare the performance of HoPF with two baseline approaches that both assume the existence of primary keys.

Keywords Data profiling application · Primary key · Foreign key · Database management

1 Structuring schemata

Primary keys (a.k.a. *keys*) and *foreign keys* are two of the most important constraints for relational databases, indicating the entity integrity and referential integrity that databases need to follow. Both constraints are ubiquitously used in databases. In a perfect world, these constraints should be explicitly assigned by database designers. However, in many

✉ Lan Jiang
lan.jiang@hpi.de

Felix Naumann
felix.naumann@hpi.de

¹ Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3,
14482 Potsdam, Germany

real-world cases, primary keys and foreign keys are either incomplete or missing, which makes it hard to understand the structure of the schemata. The reasons for the lack of constraints are diverse. For example, primary keys might not be defined by schema designers for tables that are considered fact tables and usually have multi-column primary keys. Also, developers are reluctant to use primary key constraints for non-numeric columns that are not automatically incremented. Foreign keys, on the other hand, are sometimes not defined for efficiency reasons: they may affect the speed of inserting, updating, and deleting records within the DBMS. A common practice is to control referential integrity in the application layer. All constraint definitions may be lost when migrating databases, especially through database dumps and flat files.

Knowledge of primary keys and foreign keys is essential for applications, such as data cleansing, reverse engineering, query optimization, and data integration. Although for small-scale databases, missing keys and foreign keys can still be manually labeled by domain experts, it is extremely time-consuming or even infeasible to do so for larger schemata. A few efforts have already been made for foreign key detection (Roslin et al. 2009; Zhang et al. 2010), but they assume the presence of primary keys. Normally, primary keys exist for data stored in RDBMS platforms since these tools explicitly request users to specify a primary key for each table. However, this does not always apply to databases compiled from flat files or web sources, since such constraint information would need to be stored in a separate file. In our experience, such accompanying information is often not present. This lack motivates us to detect primary keys as well and design an approach to solve the two inter-dependent problems in a holistic fashion.

Primary keys and foreign keys, which in general can cover multiple attributes, are essentially the particular cases of the *Unique Column Combinations* (UCCs) and *Inclusion dependencies* (INDs). Fortunately, many algorithms to discover UCCs and INDs have been proposed in previous works (Abedjan et al. 2015), which makes these metadata easily accessible. However, those algorithms commonly generate huge amounts of UCCs and INDs – many more than the real primary keys and foreign keys. Therefore, the task is to distinguish true keys and foreign keys from spurious UCCs and INDs. By examining data, intuitive rules can be found to distinguish keys from non-keys, which makes the automatic detection of keys feasible. A foreign key must reference a primary key according to its definition. Therefore, the detected foreign keys depend on the primary keys that we found. In this work, we devise an algorithm to first discover primary keys for a schema and then foreign keys based on these primary keys. In turn, the predicted foreign keys help remove some incorrectly predicted primary keys, improving the quality of the overall result. Section 3 introduces the problem definition more formally, but with Fig. 1 we already give a motivating example of the difficulty of PK and FK detection.

The potentially very many unique column combinations for one table are all candidates for the primary key of that table. For example, the table *Trade* in Fig. 1 has two unique column combinations, namely *T.ID* and *T.CA.ID*. Although one may also consider *T.ID,T.CA.ID* as a unique column combination, we regard only minimal unique column combinations for simplicity (Details discussed in Section 3.2). According to the schema documentation, only the *T.ID* is the true primary key among these UCCs, whereas the other one is a spurious candidate. In real world cases, there might be much more spurious candidates, making it challenging to recognize only the true PKs.

Beside true foreign keys, we can expect many spurious inclusion dependencies. The IND *Trade.T.TT.ID* \subseteq *TradeType.TT.ID*, for example, is a true foreign key while the IND *Trade.T.ID* \subseteq *Company.CO.ID* becomes a foreign key candidate only because the

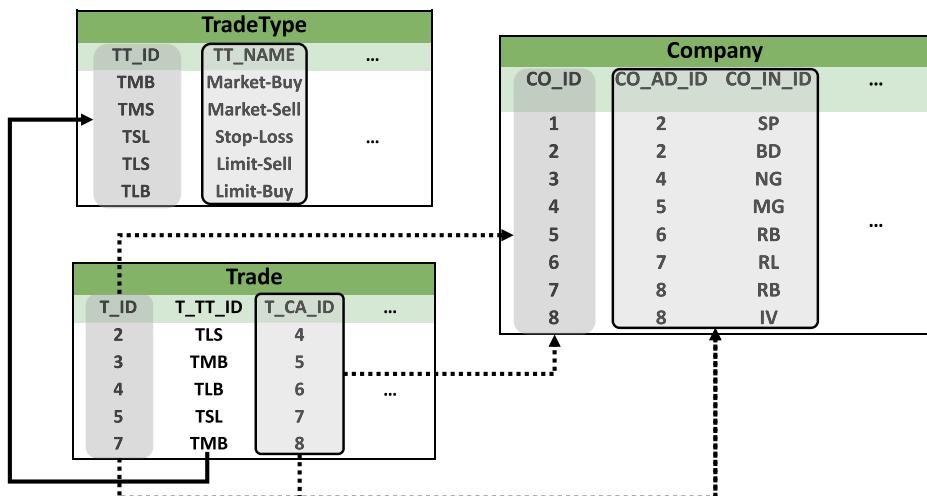


Fig. 1 A subset of tables of the *TPC-E* schema. For simplicity, some columns are omitted. True primary keys are indicated by unframed grey boxes, whereas spurious UCCs by framed grey boxes. True foreign keys are indicated by solid arrows, while spurious INDs by dashed arrows

containment is satisfied. As is shown in Fig. 1, there may have much more spurious INDs than true FKs.

In other cases, not shown in the figure, two column combinations might be included in one another. For example, the *TPC-E* columns $\{Trade.T_ID\}$ and $\{Settlement.SE_T_ID\}$ are primary keys of tables *Trade* and *Settlement*, respectively, and they happen to contain the same set of values. Consequently, both INDs $Trade.T_ID \subseteq Settlement.SE_T_ID$ and $Settlement.SE_T_ID \subseteq Trade.T_ID$ are considered as foreign key candidates, whereas only the latter one is a true foreign key.

As the results of key and foreign key discovery should be validated by an expert user, we consider both precision and recall measures in our evaluation. However, our proposed algorithm emphasizes recall, because we assume that it is easier for users to remove false positives from a small result than to discover false negatives in the large candidate set. The contributions of our approach are the following:

- We present the first combined primary key / foreign key discovery algorithm, removing the assumption that primary keys are known and present.
- We propose advanced pruning rules to filter out spurious unique column combinations and inclusion dependencies.
- Comparative experiments on five different datasets show the effectiveness of our holistic algorithm. Foreign keys are also generated without assigning primary keys to show that the absence of primary keys worsens the performance of foreign key detection.

We discuss related work in Section 2 and give the formal definition of the research problem in Section 3. Section 4 lists the features used for the primary key and foreign key detection algorithm, and Section 5 describes pruning strategies for primary and foreign key candidates. The overall algorithm is explained in Section 6. Experimental results are shown and compared to related work in Section 7. We conclude with Section 8.

2 Related work

The algorithm proposed in this paper requires unique column combinations (UCCs) and inclusion dependencies (INDs) as its input. While their discovery is not the focus of the paper, we briefly summarize the latest techniques, followed by a discussion of previous research on primary and foreign key discovery.

2.1 Metadata discovery

Efficiently discovering all UCCs and INDs for a given dataset is a challenge, due to the combinatorial explosion of column combinations. The problems of discovering all minimal UCCs and all maximal n-ary INDs are NP-complete (Lucchesi and Osborn 1978; Kantola et al. 1992). Fortunately, there are already quite a few algorithms designed to discover unique column combinations and inclusion dependencies (Abedjan et al. 2015), which are efficient in practice. However, their result sizes can be formidable. It is not unusual to discover hundreds or even thousands of UCCs and INDs even in tables with only tens of columns. The challenge is to find out true keys and true foreign keys from such large candidate sets.

2.2 Key constraints discovery

Surprisingly, not many efforts have been made for primary key detection. A set of simple heuristic features was proposed to distinguish true primary keys from spurious unique column combinations for the purpose of decomposing a relation into Boyce-Codd normal form (Papenbrock and Naumann 2017). The authors calculate and add up scores for their features for each UCC. Lacking a human expert to view the ranked results, the UCC with the top score for a table is assumed to be its primary key.

More efforts were made for discovering foreign keys (Lopes et al. 2002; Memari et al. 2015; Rostin et al. 2009; Zhang et al. 2010; Chen et al. 2014; Marchi et al. 2009). Here, we briefly introduce three representative works among them. Similar to the aforementioned primary key discovery idea, intuitive features can also be suggested for foreign key detection. Rostin et al. proposed ten features for machine learning methods to automatically detect foreign keys from various datasets (Rostin et al. 2009). However, the method detects only single column foreign keys. The authors list several situations in which the classifier makes mistakes, including transitive foreign keys and one-to-one relationships. A transitive foreign key represents the situation, where a primary key referenced by a foreign key is also a foreign key. One-to-one relationships are paraphrased as “ $PK \subseteq PK'$ in this paper and explained below. Their machine learning-based method is not able to solve these two situations.

Assuming that the data of a foreign key should well represent a sample from the key column it references, a state-of-art method introduces the *randomness* metric to discover both single-column and multi-column foreign keys (Zhang et al. 2010). The authors used the earth-mover distance (EMD) to measure the data distribution similarity between LHS and RHS of foreign key candidates: it is the minimum cost of transforming one distribution into the other by moving counts of values among buckets within a distribution (Rubner et al. 1998). The data distributions are created by choosing a fixed, same number of buckets for both LHS and RHS, and counting the number of corresponding values for each bucket. The closer the data distributions are, the smaller the distance is. It then ranks all foreign key candidates with regard to their earth-mover distance and reports the performance based on

top X% of the result. We experimentally show that our approach outperforms this work in both precision and recall in most cases.

Chen et al. proposed combining heuristic features with different pruning rules to detect foreign keys, which is most similar to our algorithm (Chen et al. 2014). However, they assume that each table pair can hold only one foreign key, which is a too strong restriction for real-world scenarios.

We note that *all* the aforementioned approaches assume that primary keys are already known and base their heuristics on them. We drop this assumption of prior knowledge and propose an approach that is better suited for many real-world scenarios.

3 Problem setting

Before introducing the problem, we briefly define the terms that we use in this paper. We denote a relational schema as \mathbb{R} , which can contain multiple relations. Each relation, denoted as R , is a named non-empty set of attributes. Attributes are denoted as $A \in R$ with their domain $\text{dom}(A)$. Each relation contains a set of tuples; the instance of R is denoted as r . A tuple in r is denoted as t . Based on the aforementioned definitions, a tuple t can be represented as $\bigcup_{A \in R} t[A]$, and for each attribute A the value $t[A] \in \text{dom}(A)$. We use $t[X]$ to denote the projection of t to the values of $X \subseteq R$. Let \mathbf{U} and \mathbf{I} be the set of unique column combinations and set of inclusion dependencies extracted from the corresponding discovering algorithms, respectively.

3.1 Types of dependencies

There are many different dependencies to describe the characteristics of a table or relationships between tables. We use only unique column combinations and inclusion dependencies since they are the prerequisites of primary keys and foreign keys, respectively. We give the definitions of these dependencies and their relationships to primary and foreign keys.

Definition 1 (Unique Column Combination) Given a relation R with its instance r , a *unique column combination* (UCC) is a set of attributes $X \subseteq R$ whose projection contains only unique, non-null value entries on r , i.e., $\forall t_i, t_j \in r, i \neq j : t_i[X] \neq t_j[X]$ and $\forall t \in R, A \in X : t[A] \neq \perp$.

Note that a column combination may contain one or multiple columns, as primary keys can be composed of one or more attributes. In fact, we have observed commercial schemata with up to 16 attributes in a primary key definition (Faust et al. 2014). A UCC is minimal, when there are no valid UCCs in its subset. In principle, each relation R should have one and only one primary key. While in practice, a primary key might not be definable for some tables, we assume that each table is duplicate free, and so at least one UCC exists.

Each table in a schema contains a number of UCCs while only one among each set is the true primary key. Picking one from each set, we can constitute a list of UCCs representing the primary keys of \mathbb{R} and denote it with $PK_{\mathbb{R}}$.

Definition 2 (Inclusion Dependency) Given two relations $R_i, R_j \in R, i \neq j$, an *inclusion dependency* (IND), denoted as $R_i[X] \subseteq R_j[Y]$, states that all the value entries in the column combination X are also contained in the column combination Y , i.e., $\forall t_i[X] \in r_i, \exists t_j[Y] \in r_j : t_i[X] = t_j[Y]$ for the relational instances r_i and r_j .

We call the dependent part $R_i[X]$ the left-hand side (LHS) and the referenced part $R_j[Y]$ the right-hand side (RHS). Note that the definition of IND implies $|X| = |Y|$. When $|X| = |Y| = 1$ we call the IND *unary*, otherwise *n-ary*. A foreign key must be an IND, because by definition, each value appearing in the LHS of a foreign key must be included in the value set of its RHS.

3.2 Problem definition

The formal problem statement of our task is:

PK/FK-Detection Problem: Given a database with schema \mathbb{R} , its set of minimal UCCs \mathbf{U} and its set of IND \mathbf{I} , find the primary key set and foreign key set, denoted as \mathbf{P} and \mathbf{F} , where $\mathbf{P} \subseteq \mathbf{U}$ and $\mathbf{F} \subseteq \mathbf{I}$.

The dependency between primary key and foreign key is obvious, because the right-hand side of a foreign key must be a primary key. Although each table may have multiple alternative keys, we assume only one of them is the true primary key. In this work, we use only minimal UCCs for two reasons: 1) the proposed PK features (see details in Section 4) always prefer the minimal UCCs than their non-minimal supersets; 2) the complete set of UCCs (including both minimal and non-minimal UCCs) could contain exponentially more UCCs than the minimal counterpart, because each column combination subsuming a UCC is also a valid UCC. However, we are aware of a disadvantage of this choice: when predicting primary keys on small relations, the true multi-column primary keys will not be added into the PK candidate sets, because a subset of such a PK set is already a minimal UCC. We will discuss it further in Section 7.

To stop suggesting too many false foreign keys, our algorithm stops foreign key detection when all relations in the schema are connected or no candidate foreign key remains. Different automatic choices of the primary keys in a schema can lead to different choices of foreign keys. To solve the problem holistically, we employ pruning rules on a score-based approach as explained in the following sections.

For simplicity, we assume only exact UCCs and INDs as input, while in reality data might be erroneous. Approximate detection algorithms address this issue and discover UCCs and INDs that might contain violations (Ilyas et al. 2004). Our approach is indifferent to such input variations and is still able to make good suggestions for keys and foreign keys.

4 Features for primary keys and foreign keys

Previous works have already proposed some useful features to identify either primary keys (Papenbrock and Naumann 2017) or foreign keys (Rostin et al. 2009; Zhang et al. 2010). We adopt some of them as well as define new ones to score each candidate of primary keys and foreign keys. In this section, we introduce each feature briefly before putting them to use in the next section.

4.1 Primary key features

Several useful heuristic features have been explored to distinguish true primary keys from spurious UCCs (Papenbrock and Naumann 2017), including a UCC's cardinality, average value length, and positions of its attributes in the schema. They have been proven by the authors to be effective in discovering primary keys for their application of normalization.

We reuse the score functions for these three features in our task. Based on our observation, in many cases, headers of primary key columns follow a different pattern than that of spurious candidates. Therefore, we introduce an additional name-based feature.

Cardinality It is easier to understand and maintain primary keys with few attributes. Here, the more columns are involved in a UCC, the lower its chance is to be a true primary key. We define the cardinality score as $\frac{1}{|X|}$, where X is the column set of the UCC.

Value length Columns used as primary keys are expected to have short value length, because they are usually identifiers with no semantic meaning. In addition, indexing long values of primary keys may impair efficiency. The score function used for this feature is $\frac{1}{\max(1, |\max(X)| - n)}$, where $|\max(X)|$ is the length-average of the longest values of each column in X , and n represents the parameter to penalize long values. It can be adjusted according to the datasets. In our experiments we choose $n = 8$, reflecting typical choices of data types for primary keys. For multiple column UCCs, we calculate the score for each column and use the average as its overall score. While using a constant in the score function affects our generality, the experimental result shows it is quite useful.

Position In principle, attributes are unordered in a relation, while in practice, there is an implicit order of attribute positions when defining the schema. In most cases, primary keys appear in the first positions of the column set. A similar observation was made for web tables (Venetis et al. 2011). In addition, for multi-column primary keys, we expect no (or only few) non-key columns in between the key columns. The column position score is calculated as $\frac{1}{2}(\frac{1}{|left(X)|+1} + \frac{1}{|between(X)|+1})$, where $left(X)$ and $between(X)$ represent the number of columns left of the first column of X and the number of non-key columns between the first and last columns of X , respectively.

Name suffix We notice in all datasets used for experiments, primary key columns are usually indicated by their column suffix name, e.g., “id” and “key”. Here we choose “id”, “key”, “nr”, and “no” as our suffix set. Clearly, this list can be extended, for instance, to include foreign language schemata. We apply the score function as $\frac{|suffix(X)|}{|X|}$, while $|suffix(X)|$ counts the number of columns in the UCC whose name contains either suffix mentioned above. We denote the average of the above feature scores as the primary key candidate scores. As we cannot expect all schemata to be so accommodating in their labels, we also perform experiments with this feature turned off.

4.2 Foreign key features

Rostin et al. proposed ten heuristic features to discover foreign keys among INDs (Rostin et al. 2009). Zhang et al. suggested a *randomness* feature for data value distribution to subsume a variety of those features except column names (Zhang et al. 2010). We validate the effectiveness of such coverage and use the data distribution as a measure as well. However, the profiling time is found to be quite long due to the construction of multi-dimensional histograms for n-ary foreign key candidates in practice. Therefore, we simply treat each dimension of n-ary foreign key candidates as single column candidates and average the scores. Column name similarity also plays an influential role, which is not covered by the data distribution features. Therefore, we employ two features for foreign keys discovery, which are *column name* and *data distribution*.

Column name In many cases, for the purpose of better understanding and database maintenance, database designers do not give arbitrary names to related columns. Given an IND $R.A \subseteq S.B$, it is more likely to represent a true foreign key if the labels of R and A are similar to those of S and B . We tested several string similarity functions and found the fuzzy similarity function initially proposed to solve the fuzzy matching between records (Chaudhuri et al. 2003), to be most suitable for our task. To calculate similarity using this metric, each string is tokenized first by delimiters, such as uppercase letters, and “_”. For each foreign key candidate, we combine the token sets of table name and column name for both LHS and RHS. For example, given an IND $\text{Trade.T_S_SYMB} \subseteq \text{LastTrade.LT_S_SYMB}$, the column names of LHS and RHS are turned into $T_L = \{\text{Trade}, \text{T}, \text{S}, \text{SYMB}\}$ and $T_R = \{\text{Last}, \text{Trade}, \text{LT}, \text{S}, \text{SYMB}\}$, respectively. When calculating the similarity of LHS and RHS, we map each token in T_R to the most similar unmapped token from T_L , as calculated by the Levenshtein distance. The similarity of $L \subseteq R$ is calculated according to the following formula (Chaudhuri et al. 2003):

$$\text{sim}(L, R) = \frac{\sum_{j=1}^n (\text{sim}_j \times \ln(\frac{1}{f_j}))}{\sum_{j=1}^n \ln \frac{1}{f_j}}$$

where sim_j is the token similarity between each token in T_R and the one most similar to it in T_L . If T_R contains fewer tokens than T_L , some tokens in T_R cannot find a map. For each such token j , we set the sim_j as zero. f represents the frequency of each token, and $\ln(\frac{1}{f_j})$ is the weight of each token in T_R , which measures the rarity of the token. Intuitively, a rare token is more useful to indicate the similarity between two strings, therefore should receive a higher weight. The similarity is asymmetric, which means given two strings L and R , the similarity of $L \subseteq R$ is different from that of $R \subseteq L$. This is useful to recognize a true foreign key when two column-sets are included in one another and thus their data distribution score is identical.

Data distribution The data distribution of the participating columns is a good indicator to distinguish foreign keys from spurious INDs. In Zhang et al. (2010) the authors assume that values of the dependent column(s) should be uniformly sampled from those of the referenced column(s) and propose the earth-mover distance as a cost measure. We verify the effectiveness of this assumption while observing the time overhead to construct this measure is large. Therefore, we propose a simpler yet (as our experiments show) effective histogram difference to represent the cost: Given an IND $R_i[X] \subseteq R_j[Y]$, we create a set of buckets according to the value range of each column Y_i in $R_j[Y]$ and put each value into the corresponding bucket. We choose twenty as the number of buckets by default, and in Section 7.2 we experimentally show that it is a good choice. The buckets form a histogram, which is denoted as $\text{Hist}(Y_i)$, and for each column X_i in X we place its values into the buckets created for Y_i . The overall data distribution score is the average of the histogram difference score in each dimension. While there are some other alternatives, such as the L_1 norm and histogram intersection, we use the Bhattacharyya coefficient (Bhattacharyya 1943) – a known good solution to measure the similarity between two histograms.

5 Pruning PK and FK candidates

Enumerating and scoring all combinations of UCCs and INDs in the search space is the most naive method to discover PKs and FKs. It is extremely time-consuming due to the

potentially exponential number of candidates. Consider a schema with only 20 tables, each of which contains only two UCCs and the whole database contains only 50 INDs. Then there are 2^{20} primary key candidate combinations and 2^{50} candidate combinations of foreign keys in the search space. And in practice, tables contain many more UCCs and typical databases contain thousands of INDs.

Foreign keys are defined to reference a primary key. This motivates us to combine the discovery of primary keys and foreign keys holistically. Specifically, a combination of UCCs is selected from the search space and considered as the predicted primary keys. Thereafter, we determine the foreign keys from the INDs whose RHS are among these primary keys. The PK's score is summed up with the FK's score for this selection. After enumerating all combinations of UCCs and their corresponding INDs, the selection of UCCs and INDs with the highest overall score is the output PKs and FKs.

In order to avoid checking every UCCs and INDs, we suggest a handful of filtering techniques for both primary key and foreign key discoveries in this chapter to help the algorithm skip unnecessary checks. These techniques include foreign key candidate prefiltering, $PK \subseteq PK$ filtering, primary key candidate pruning, and foreign key candidate pruning. We discuss each of them in detail in the rest of this chapter.

5.1 FK candidate prefiltering

The number of INDs typically grows quadratically with the number of tables and columns (Tschiirschnitz et al. 2017). However, not all of them are good candidates of foreign keys. The prefiltering step aims to keep only a small portion of the suitable INDs as foreign key candidates. We use the following rules to refine the original IND set:

RHS uniqueness A foreign key can reference only a primary key, which is by definition a UCC and does not contain null-values. Therefore, we prune all INDs whose RHS is not a UCC, and thus not a primary key candidate.

Non-null column combinations We observed that columns with only *null* values exist, especially in real-world datasets. In principle, these columns can be seen as included in any other column. However, for the purpose of deriving foreign keys, they are not useful and thus we ignore them.

Table 1 displays the number of INDs before and after prefiltering (we introduce the datasets in more detail in Section 7.1).

5.2 $PK \subseteq PK$ filtering

If the values of the LHS of an IND are a consecutive subsequence of the values of RHS, it may seem like a good foreign key candidate. However, in many cases, these are in fact

Table 1 INDs before and after FK candidate prefiltering

Dataset	Before	After
TPC-H	90	33
TPC-E	511	175
Adwork	19,602	2,047
SCOP	6,450	2,062
MusicBrainz	236,151	28,722

auto-incremented unary primary keys of the two tables. Given an IND $A \subseteq B$, we remove it if the ordered values in A form a consecutive subsequence of the ordered values of B – in such cases it is more likely that A is a key in its own right, and its inclusion in B is spurious.

In TPC-H for example, all tables have such an auto-incremented integer primary key. *REGION* contains only five tuples, whose primary key values are 1...5. On the other hand, *NATION* contains 24 tuples, whose primary key values are 1...24. Therefore, *REGION.REGIONKEY* \subseteq *NATION.NATIONKEY* seems to be a good foreign key candidate. However, such a $PK \subseteq PK$ IND is meaningless and thus should be filtered out.

Previous efforts did not deal well with such $PK \subseteq PK$ candidates in the predicted foreign keys (Rostin et al. 2009; Chen et al. 2014). Motivated by the fact that most tables contain auto-incremental integer primary keys, the authors of Zhang et al. (2010) suggest a consecutive prefix or suffix check between LHS and RHS as a naive approach to detect foreign keys. In our approach, we relax this restriction by considering any consecutive subsequence.

5.3 Primary key candidate pruning

Each table may possess multiple UCCs. To obtain the optimal result, we need to consider all valid UCC combinations, i.e., the Cartesian product of the UCCs by the tables they belong to. In practice, the search space is still quite large, even after we restrict only one out of all the UCCs for each table to be considered as the primary key. For example, we can see from Table 2 that for the TPC-H schema, there are more than 76 million different $PK_{\mathbb{R}}$ candidates, denoted as PK_{cc} , i.e. Primary key combination candidates.

To reduce the number of candidates, we score each UCC by the primary key features mentioned in Section 4.1. We rank them within each table in a descending order of their scores with the goal of pruning poor candidates.

Figure 2 shows the score for each primary key candidate of the table *Trade* in the TPC-E schema, as an example. The x-axis represents the rank of candidates with regard to their scores. We denote S_i as the score for the top- i th candidate. The *score-difference* is marked as $SD_i = S_i - S_{i+1}$ for each pair of neighboring candidates. In analogy to the notion of a *knee* for continuous curves, we define a cliff for our discrete case:

Definition 3 (Cliff) Given the sorted score list of the primary key candidates of a table $S = \{S_1, S_2, \dots, S_n\}$ and the corresponding SD list $SD = \{SD_1, SD_2, \dots, SD_{n-1}\}$, the *cliff* is the pair of neighboring candidates S_i and S_{i+1} with the largest SD score.

Table 2 Evaluation datasets with primary key candidate combinations *before* and *after* pruning Lower

Datasets	Tables	PKcc before	PKs U	PKs L	PKcc after
TPC-H	8	7.67×10^7	8	0	1
TPC-E	32	9.03×10^{13}	31	1	768
AdWork	27	6.23×10^{21}	27	0	256
SCOP	42	7.25×10^{11}	42	0	2
MusicBrainz	124	3.73×10^{25}	122	2	576

PKs U and L indicate the number of true primary keys falling into Upper or Lower. Thus, PKs L is the difference of the number of tables and PKs U

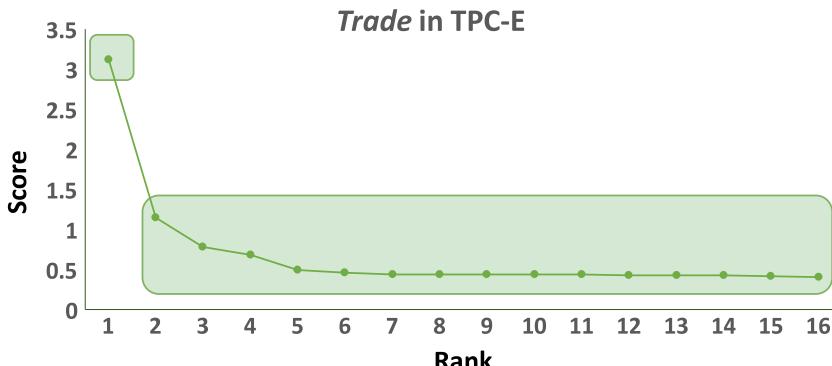


Fig. 2 Scores of primary key candidates of tables *Trade* in *TPC-E*. The *cliff* appears between the first and second candidates

The candidates of each table are separated into two subgroups, namely *Upper*, which includes all the candidates before the cliff (S_1, \dots, S_i), and *Lower*, which includes the remaining candidates (S_{i+1}, \dots, S_n). For example, the cliff of *Trade* is the pair of top two primary key candidates due to the largest *SD* score between them. In this case, the upper part contains only the top candidate, while the lower part contains the remaining 15 ones.

We calculated Upper and Lower for each table in all datasets to see into which part the true primary keys fall. Table 2 shows the result. For example, 31 out of 32 primary keys in the *TPC-E* fall into Upper, which means the pruning loses only very few true positives. The statistics in Table 2 supports this observation for also for the other datasets. To reduce the search space of PK_R , we simply drop Lower from contention for primary key, shrinking the search space dramatically. This step is implemented as *cliffPrune()* in Algorithm 2. For instance, the search space for SCOP is reduced to only two combinations of primary keys. On the other hand, only for TPC-E and MusicBrainz do we lose primary keys. We verified the effectiveness of filtering out poor candidates with cliff. By using this notion, there is no need to setup a parameter to control the number of filtered out candidates.

5.4 Foreign key candidate pruning

The IND discovery algorithms, which produce the foreign key candidates, regard only value containment between columns. However, there are two types of conflicts when searching for the true foreign keys among the candidates. Our approach needs to validate that no conflicts are created each time a foreign key is predicted. We explain the strategies employed by our algorithm and visualize them in Fig. 3.

Uniqueness of foreign keys Each foreign key can reference only one primary key in a schema, while the values of the LHS of an IND might be contained in multiple different RHSs. If one of these INDs is a true foreign key (shown as the solid arrow in Fig. 3a), it is clear that all others are spurious (shown as the dashed arrow).

Non-cyclic reference If a cyclic reference exists in the schema, all the involved column combinations contain the same values, which we deem as not semantically meaningful. Therefore, we do not predict a foreign key that causes a cyclic reference in the schema graph. For example, the dashed arrow in Fig. 3b introduces a cyclic reference and should

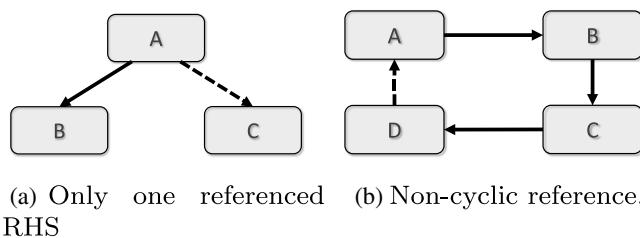


Fig. 3 Dashed INDs are invalid foreign keys

not be predicted as a foreign key if the other three solid arrows are already classified as foreign keys.

Schema connectivity After conducting the aforementioned processing and filtering, the number of remaining foreign key candidates shrinks dramatically. Our algorithm proceeds to score each candidate and choose a good set of foreign keys. To avoid specifying a score threshold, we make the assumption that the schema graph is in fact a spanning tree. The connectivity of all the tables is a good indicator that the majority of true foreign keys have been found, as shown by the recall in Fig. 5. To this end, HoPF checks whether the schema connectivity has been fulfilled each time a new foreign key is predicted.

6 Holistic algorithm HoPF

Figure 4 shows the overall process of our proposed algorithm HoPF (*Holistic Primary Key and Foreign Key Detection*). It refines the set of UCCs and INDs first, decreasing the

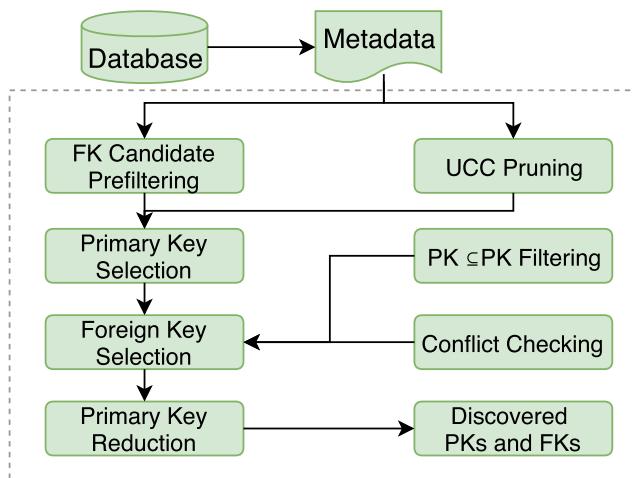


Fig. 4 Overview of the HoPF algorithm

number of primary key and foreign key candidates needed to be processed in the next step. For all surviving candidates, HoPF enumerates all primary key candidate sets and their corresponding INDs, scoring each of them with regard to the features proposed in Section 4 and selecting the proposed true primary keys and foreign keys. $PK \subseteq PK$ candidates are removed during this procedure. Conflict checking is integrated into the process of foreign key detection to remove further spurious INDs. Before producing the final result, a primary key reduction step simplifies the predicted primary keys and amends the corresponding foreign keys. Note that the used candidate pruning techniques are described in Chapter 5.

The general algorithm to holistically determine primary keys and foreign keys from UCCs and INDs is displayed in Algorithm 1, which in turn calls various methods that are shown separately.

In line 1.2, i.e., line 2 of Algorithm 1, the search space of primary key combinations is generated and saved in $PKcc$ (Primary Key combination candidates). The loop from 1.3 calculates the scores of these primary key combinations in a row along with their corresponding foreign keys, determining the one with the highest score as the predicted primary key set and foreign key set. In line 1.4 only the foreign key candidates that reference a predicted primary key are retained in $FKCands$. We restrict the minimal number of FKS to be $|PKc| - 1$ in line 1.5, otherwise it is certain to break the rule of schema connectivity. In line 1.6 the algorithm searches for the INDs to form the foreign keys given the detected primary keys PKc . Here, D represents the discarded inclusion dependencies that are used to restore some foreign keys later in line 1.7. The lines 1.8-1.12 simply select the primary keys and their corresponding foreign keys as the final prediction.

Algorithm 1 Holistic PK/FK detection (HoPF).

Input: UCC set U , IND set I
Output: predicted primary keys P , predicted foreign keys F

- 1 $P = \{\}, F = \{\}, Q = 0;$
- 2 $PKcc = GetPKCombinations(U);$
- 3 **forall** PKc of $PKcc$ **do**
- 4 $FKCands = I \setminus PruneCands(I, PKc);$
- 5 **if** $|FKCands| \geq |PKc| - 1$ **then**
- 6 $(FKCands, D) = GetFKCandidates(FKCands);$
- 7 $(PKc, FKS) = PKReduce(PKc, FKCands, D);$
- 8 $score = Score(PKc) + Score(FKS);$
- 9 **if** $Q < score$ **then**
- 10 $P = PKc;$
- 11 $F = FKS;$
- 12 $Q = score;$
- 13 Return (P, F)

Algorithm 2 displays the procedure of finding all suitable primary key combination candidates. It first groups UCCs by the tables they belong to, and filters out poor primary key candidates with the cliff-technique described in Section 5.3. To compose a primary key combination, one candidate from each table's leftover UCC set is selected.

Algorithm 2 GetPKCombinations().

Input: UCC set U , relations R
Output: primary key combinations $PKcc$

- 1 $PKcc = \{\}$;
- 2 **forall** T_i of R **do**
- 3 $g_i = \{u | u \in U \wedge T_u = T_i\}$;
- 4 $cliffPrune(g_i)$;
- 5 **forall** $PKc \in g_1 \times \dots \times g_{|R|}$ **do**
- 6 $PKcc = PKcc \cup \{PKc\}$;
- 7 **return** $PKcc$

The procedure of generating foreign keys given the primary keys is shown in Algorithm 3. All foreign key candidates in I are ranked by their score. In line 3.3 we initialize a graph to represent the tables of a schema and the predicted foreign keys among them. Each node represents a single table. An edge is added between the two corresponding nodes if there is a foreign key connecting two tables. From line 3.4 to line 3.15 qualified INDs are greedily added to the graph as predicted foreign keys from top to bottom w.r.t. their scores. Each time a new foreign key is predicted, the algorithm checks whether the schema connectivity is met and stops if the graph is connected. The circle-reference conflict is checked in line 3.6, followed by the $PK \subseteq PK$ check. Candidates failing the $PK \subseteq PK$ check are added into the discard set D and used in the primary key reduction phrase. Once a foreign key is predicted, all other candidates that share the same LHS are excluded (lines 3.14 and 3.15).

Algorithm 3 GetFKCCandidates().

Input: inclusion dependencies I
Output: foreign key candidates $FKCands$, discard set D

- 1 $FKCands = \{\}$; $D = \{\}$; $maxFKsScore = 0$;
- 2 $I' = SortByScoreDescending(I)$;
- 3 $G(I') = G(N=T, E=\{\})$;
- 4 **while** I' is not empty **do**
- 5 $FKCand = FirstElement(I')$;
- 6 **if** $FKCand$ causes no circle reference **then**
- 7 **if** $FKCand$ is not a $PK \subseteq PK$ **then**
- 8 $FKCands = FKCands \cup \{FKCand\}$;
- 9 $E = E \cup Edge(FKCand)$;
- 10 **if** $G(I')$ is connected **then**
- 11 | break;
- 12 **else**
- 13 | $D = D \cup \{FKCand\}$;
- 14 $FKCand^+ = \{FKCand' | LHS(FKCand') = LHS(FKCand) \wedge FKCand' \subseteq I'\}$;
- 15 $I' = I' \setminus FKCand^+$;
- 16 **return** ($FKCands$, D)

In principle, each table should define a primary key to keep entity integrity. In practice, however, primary keys are not always defined, especially for so-called join tables, which represent $m : n$ relationships. Although an indexed primary key helps to query more efficiently, it becomes an encumbrance under frequent data modifications, leading schema

designers to avoid defining primary keys. We observe that the two largest datasets used in our experiments, i.e., *SCOP* and *Musicbrainz* both contain a few join tables without primary key definitions. Because we are using only minimal UCCs as input to generate primary key candidates, a true multi-column primary key of a join table may not be included in the candidate set, if a subset of these columns is already a UCC. If our algorithm marks it as the primary key for such a table, it is definitely a false positive. A true foreign key, whose LHS is such a false positive primary key, may be marked as spurious IND with the $PK \subseteq PK$ filtering described above.

A naïve solution to avoid this loss of true foreign key is to use the set of full UCCs as input to produce primary keys. However, we found this approach has two disadvantages. First, a full set of UCCs contains exponentially more UCCs than its minimal set counterpart, because each column combination subsuming a UCC is also a valid UCC. Second, the primary key features selected for HoPF does not prefer non-minimal UCCs than minimal ones. To boost the recall of foreign keys, we propose primary key reduction as a post-process as shown in Algorithm 4. We observe that by removing these inappropriate primary keys, we can rediscover some true foreign keys that were discarded when iterating the FK candidates. For each predicted primary key, we exclude it from PKC and update the corresponding foreign key candidates. We assume that a predicted primary key is not true (and should be removed) if the overall score rises with its absence. This process consequently leads to two advantages. On the one hand, some false positive primary keys are ultimately removed, and previously discarded $PK \subseteq PK$ candidates referencing one of them are restored, because their LHSs are no longer primary keys. On the other hand, the false positive foreign keys referencing one of these removed primary keys are not conceived as foreign keys anymore.

Algorithm 4 PKReduce().

Input: primary key combination PKC , predicted foreign keys $FKCands$, discarded foreign key candidates D

Output: updated primary keys PKC , updated foreign keys FKs

```

1  $FKs = \{\}$ ;
2  $score = score_{PKC} + score_{FKCands}$ ;
3 for each  $PK$  in  $PKC$  do
4    $PKC' = PKC \setminus PK$ ;
5    $FKCands' = UpdateFKCands(PKC', FKCands, D)$ ;
6   if  $score < score_{PKC'} + score_{FKCands'}$  then
7      $PKC = PKC'$ ;
8      $FKs = FKCands'$ ;
9      $score = score_{PKC} + score_{FKs}$ ;
10     $PKC = PKC \setminus P$ ;
11 return  $PKC, FKs$ 

```

7 Experiments and analysis

We now evaluate the effectiveness of the proposed algorithm HoPF. After introducing the setup of the experiments, we first present the precision and recall that HoPF achieves on various datasets, followed by an analysis of incurred errors of three different types, i.e., incorrect primary key, empty LHS column, and $PK \subseteq PK$. Next, we explore and report the

Table 3 Datasets and their statistics

Name	Tables	Columns	UCCs	PKs	INDs	FKs
TPC-H	8	61	435	8	90	8
TPC-E	32	185	167	32	411	45
AdvWorks	27	321	1,434	27	4,300	45
SCOP	65	282	120	42	5,244	90
MusicBrainz	124	682	252	124	236,151	168

influence on F-1 score with different size of buckets used for data distribution feature. After that, we explore the performance of foreign key discovery without assigning primary keys, showing the necessity of detecting primary keys. Finally, we conclude this section with the performance comparison between HoPF and the state-of-art algorithm (Zhang et al. 2010).

7.1 Experimental setup

Because HoPF consumes UCCs and INDs as input, we assume that the datasets at hand have already been profiled so that these dependencies, along with basic statistics are available. Given many efficient profiling algorithms (Abedjan et al. 2015), we can readily acquire these dependencies, making this assumption reasonable. We extracted all these metadata with *Metanome* (www.metanome.de), a data profiling platform which provides a wide spectrum of interfaces for different profiling algorithms (Papenbrock et al. 2015).

We conducted our experiments on multiple datasets, including two benchmark datasets *TPC-H* and *TPC-E*,¹ a generated dataset *AdventureWorksDW*,² as well as the two real-world datasets *SCOP*³ and *MusicBrainz*.⁴ All datasets contain complete key and foreign key definitions, which we used as gold standard for precision and recall evaluation. Also, all datasets contain header names for each column. Details of the datasets are shown in Table 3. As our algorithm is data-driven, we removed empty tables, which appeared only in *MusicBrainz*, reducing the number of tables there from 206 to 124. Both *TPC-H* and *TPC-E* have several parameters to control their scales. To ease comparison with existing work, we set the parameters as in Zhang et al. (2010).

The number of inclusion dependencies includes both unary and n-ary ones. HoPF attempts to discover both single-column and multi-column foreign keys out of them, respectively. After double-checking the datasets, we notice that only 42 true primary keys are defined in *SCOP* although the schema contains 65 tables. We observed that all the tables without defined primary keys are join tables. We attempt to remove these from our result by employing primary key reduction, which is described in Section 7.2.

7.2 Precision and recall analysis

We executed HoPF on the entire schema of each dataset. The algorithm takes UCCs, INDs, and basic statistics as input, and outputs a subset of UCCs/INDs respectively as the predicted

¹<http://www.tpc.org>

²<https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks>

³<http://scop.berkeley.edu>

⁴<https://musicbrainz.org>

PKs/FKs. We compared the output sets with the gold standard of each dataset, respectively, and report precision and recall, i.e., the amount of predicted true foreign keys divided by the amount of predicted foreign keys, and the amount of predicted true foreign keys divided by the amount of foreign keys in the gold standard. Figure 5 illustrates the precision and recall of foreign key discovery for each dataset for top X predicted foreign keys. As we can see from the figure, recall increases gradually, whereas precision does not see an apparent drop. That is to say, for each dataset, the true foreign keys usually appear in the top part of the predicted list, which means they have higher scores than spurious foreign key candidates. And when checking more predicted entries to the bottom of the list, we see fewer foreign keys while more spurious INDs. This proves the effectiveness of the foreign key features we select for HoPF.

Table 5 lists the number of primary keys and foreign keys discovered by HoPF. It manages to discover most true primary keys for all datasets. After inspecting the sets of predicted primary keys, we found that errors occur for two reasons. First, the primary key features do not cover those erroneous cases well. For example, the algorithm failed to find the true primary key for the *Financial* table in *TPC-E*, because the table contains a three-column primary key and several spurious unary UCCs. This contrasts to our hypothesis that a primary key shall have small cardinality. In addition, the names of the true primary key columns do not follow the suffix name rule we employ. These two factors contribute to the failure of predicting the correct primary key for this relation. However, this error accounts for only a small portion. We do not find out other failures with the same reason, indicating that the features we employ for primary key discovery are in general effective. More errors stem from the primary key reduction step. We will discuss the influence of this strategy on the performance of both PK and FK discovery below. We also studied the actual false negative and false positive cases of predicted foreign keys that caused by HoPF, and found they fall into the following three categories:

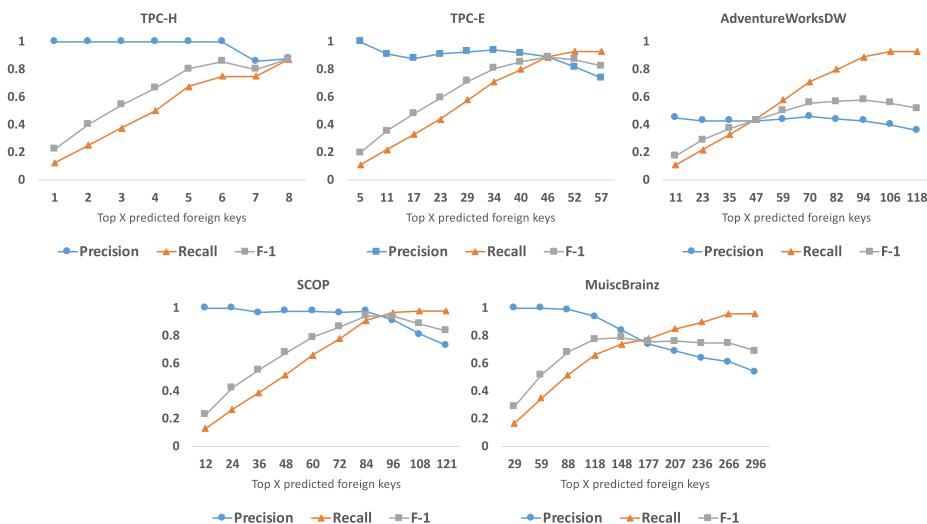


Fig. 5 Effectiveness of HoPF on the five datasets. The x-axis reflects choosing the top X predicted foreign keys

Table 4 True primary keys and true foreign keys predicted with and w/o primary key reduction

Dataset	# PKs w (w/o) reduction	# FKS w (w/o) reduction
TPC-H	8 (8)	7 (7)
TPC-E	27 (30)	42 (40)
AdvWorks	25 (25)	40 (42)
SCOP	35 (41)	88 (77)
MusicBrainz	93 (109)	161 (137)

Incorrect primary key An incorrectly predicted primary key could largely hurt the foreign key discovery results, because foreign keys must reference a particular primary key. Therefore, once HoPF obtains a false positive/negative primary key, it might possibly predict multiple false positive/negative foreign keys. In practice, however, we did not witness many errors caused by incorrect primary keys. After inspecting the results, we found out that the spurious primary key candidates usually have low scores and are thus filtered by conflict rules and the connectivity restriction.

Empty LHS column If the LHS of an IND is empty, all attributes in the LHS contain only null-values. An IND with an empty LHS can still be a valid foreign key, especially in real-world datasets. However, our data distribution feature is not able to establish an evaluation of such an inclusion dependency and we filter them out in an early stage. For example, in *MusicBrainz*, $\text{artist_type.parent} \subseteq \text{artist_type.id}$ is a true foreign key that cannot be detected, since the LHS is an empty column. We observed 24 other foreign keys with an empty LHS only in *MusicBrainz*, which reduces recall of largely.

PK \subseteq PK Although we employ the $\text{PK} \subseteq \text{PK}$ filter in an early stage to remove these candidates, some of them may be added back to the predicted result when the algorithm removes redundant primary keys with primary key reduction. In *MusicBrainz*, for example, $\text{place_alias.type.id} \subseteq \text{label_alias.type.id}$ is reconsidered as a foreign key after the algorithm no longer holds $\{\text{place_alias.type.id}\}$ as a primary key. Nevertheless, as we can see from Table 4, because primary key reduction works well in most cases, HoPF is still able to block most $\text{PK} \subseteq \text{PK}$ candidates.

By removing those primary keys that are referenced by some predicted foreign keys with a low score, the discovered foreign keys receive a higher overall score. Therefore, HoPF considers them as non-primary keys. Nevertheless, the loss of the performance of primary key discovery improves the performance of foreign key discovery, as displayed

Table 5 Gold standard and number of primary keys and foreign keys discovered by HoPF, marked as ‘true’ and ‘disc.’, respectively. ‘undoc.’ represents the undocumented foreign keys discovered by HoPF

Datasets	True PKs	Disc. PKs	True FKS	Disc. FKS	Undoc. FKS
TPC-H	8	8	8	7	0
TPC-E	32	27	45	42	0
AdvWorks	27	25	45	40	2
SCOP	42	35	90	88	2
MusicBrainz	124	101	168	161	0

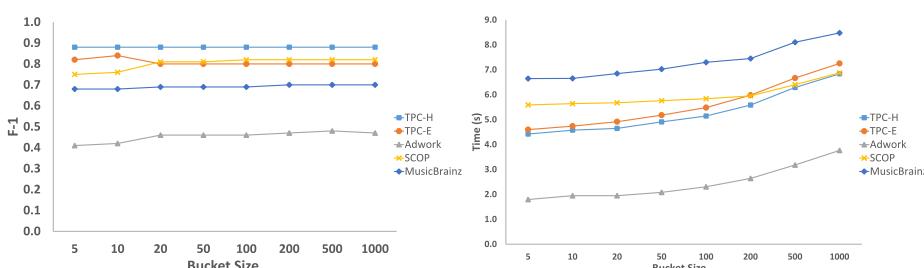
and explained below. It brings back a group of true foreign keys that were considered as $PK \subseteq PK$ in the previous step (Table 5).

Table 4 displays the number of true primary keys and foreign keys predicted by HoPF with and without primary key reduction. For example, HoPF retrieves 35 primary keys and 88 foreign keys for *SCOP* with primary key reduction. The respective numbers are 41 and 77 for the same dataset without primary key reduction. As we can see from the results, applying primary key reduction helps to predict more foreign keys at the price of losing some true primary keys for all datasets, except for *AdvWorks* where two more true foreign keys were removed from predictions after primary key reduction. After double-checking the results, we found that primary key reduction falsely removed the true primary key of a table, which has a unary primary key, which is referenced by the two missing foreign keys. Therefore, we suggest employing the primary key reduction strategy for datasets containing join tables without primary key definitions. In these cases, this strategy helps discover more foreign keys at the price of a small loss of true primary keys. Nevertheless, we believe it is easier for users to notice the missing primary keys compared to finding the missing foreign keys, which necessitates to collectively consider multiple tables. In our experiments, we employed primary key reduction for all datasets.

We also tested the influence of choosing different bucket numbers on our histogram difference feature. Figure 6a illustrates the F-1 score for each dataset under different bucket numbers. The scores fluctuate a little when we choose bucket size smaller than 20, but keep stable in general. Figure 6b shows the time consumption for each dataset. For easy reading, the time is unitized by second and displayed logarithmically. As we can see, the time overhead to construct this data structure increases significantly when using more buckets. For all participant datasets, the optimal bucket size is between 10 and 20, and we choose 20 as our default value.

Using only minimal UCCs as the input of HoPF to produce primary key candidates can possibly miss true n-ary primary keys, if a subset of a n-ary primary key is already a UCC. This problem and the problem it causes on join tables has been discussed early in the section, for which we proposed the primary key reduction strategy. In addition, the lack of complete UCC sets also causes a similar consequence for very small datasets, with only few records per relation. In these cases, relations can more readily have unary UCCs, thus shadowing larger UCCs that are the true PKs as we regard only minimal UCCs as our input.

To gauge this effect, we apply HoPF on a TPC-H instance of scale factor to 0.001. In this case, all relations in the dataset contain only dozens of records. The result confirms our suspicion: HoPF is not able to discover the true primary keys of table *lineitem* and *partsupp* who



(a) F-1 score under different bucket number. (b) Time consumption under different bucket number.

Fig. 6 The F-1 score under different bucket number and their corresponding time spent

have n-ary primary keys, because both of them also have unary UCCs that are part of the true PK. Unfortunately, our algorithm cannot solve this very-small-table dilemma without changing the prerequisite of only consuming minimal UCCs. As mentioned in Section 3.2, changing the prerequisite so as to use also non-minimal UCCs does not solve this dilemma, because the proposed PK features favors UCC candidates with small cardinality, and thus HoPF is more liable to refer to them as true primary keys.

7.3 Foreign key detection without primary keys

Previous works all assume that primary keys are at present. However, this assumption is too optimistic for many databases, especially for those stored as dumps and flat files, whose constraint definition might not be tightly attached to data themselves. In these cases, foreign key discovery algorithms based on this assumption might break or predict possibly many erroneous foreign keys. Here, we use HoPF to explore the influence on the quality of discovered foreign keys without the presence of primary keys. Fortunately, even without discovering the primary keys, our method can still rank the list of INDs by their foreign key score. Instead of predicting primary keys holistically with foreign keys, we let HoPF output a predicted foreign key, as long as its RHS is a UCC.

Table 6 displays the detected foreign keys with and without knowledge of primary keys. Consistent to our assumption, without the primary keys, HoPF obtains fewer true positive foreign keys, at the price of searching in even larger foreign key candidate sets. For example, if we ignore the primary keys and rank only the foreign key candidates by their score, we obtain 141 true foreign keys out of 1079 candidates in *Musicbrainz*, whereas we obtain 161 true foreign keys out of only 296 candidates if we detect the primary keys first and use them as the input for foreign key detection.

In general, we conceive only those inclusion dependencies whose RHS is among the primary key set into the foreign key candidate set. Without knowledge of primary keys, HoPF would consider each IND whose RHS is a UCC as a foreign key candidate. This explains why the number of foreign key candidates grows without considering primary keys; each table typically contains many UCCs, and each of them may contribute several valid INDs into the candidate set.

Restricted by the *uniqueness of foreign key* rule, a true foreign key sharing the same LHS with another foreign key candidate is excluded if the latter one has a higher score and is predicted as a foreign key. However, if an incorrect foreign key is predicted, adding a true foreign key to the final result may cause a circle-reference, and thus be rejected. This explains why fewer true positive foreign keys are acquired without the knowledge of primary keys.

Table 6 Detected foreign keys without and with knowledge of primary keys

Datasets	FKs	FKs disc. w/o PKs	Cand. FKs w/o PKs	FKs disc. with PKs	Cand. FKs with PKs
TPC-H	8	2	18	7	8
TPC-E	45	41	77	42	57
AdvWorks	45	39	369	40	118
SCOP	90	88	167	88	121
MusicBrainz	168	141	1,079	161	296

7.4 Undocumented foreign key discovery

In addition to discovering documented foreign keys, further potential foreign keys were found in *AdventureWorksDW* and *SCOP* in spite of their absence in the gold standard. These uncertain foreign keys fall into two categories: missing ones and erroneous ones. For instance, *pdb_release_author.pdb_author_id* \subseteq *pdb.author.id* in *SCOP* was counted as a false positive while we believe it is in fact a true foreign key. On the other hand, *cdd.release_id* \subseteq *pfam.release.id* is a documented foreign key while we believe it to be incorrectly documented; the correct foreign key should likely be *cdd.release_id* \subseteq *cdd.release.id*, which was predicted by HoPF.

We can imagine several reasons for these undocumented foreign keys, e.g., loss during data migration or removal by schema designers for query efficiency. The number of undocumented foreign keys predicted by HoPF for each dataset is shown in the last column in Table 5. Such a discovery may provide us with an insight for further database application such as data integration.

7.5 Comparison

We re-implemented the state-of-art algorithms of Zhang et al. (2010) and Chen et al. (2014), which we dub *Randomness* and *FastFK*, respectively, and compared their performances with HoPF. As explained in more detail in Section 2, the *Randomness* algorithm

Table 7 Comparison of foreign key detection among HoPF and other two previous work (Zhang et al. 2010; Chen et al. 2014). The bold number indicates the better performance of the algorithm against the counterparts

Data-set	True FKs	Algorithm	With column names				Without column names			
			P	R	F-1	Predicted FKs	P	R	F-1	Predicted FKs
TPC-H	8	FastFK	.56	.90	.69	16	.56	.90	.69	16
		Randomness	1	1	1	8	.21	1	.35	39
		HoPF	.88	.88	.88	8	.88	.88	.88	8
TPC-E	45	FastFK	.72	.95	.82	59	.59	.78	.67	59
		Randomness	1	.89	.94	45	.57	.82	.67	308
		HoPF	.72	.91	.80	57	.64	.82	.72	57
AdvWorks	45	FastFK	.32	.97	.49	131	.24	.72	.37	131
		Randomness	.90	.41	.56	122	.21	.58	.31	122
		HoPF	.31	.84	.46	118	.27	.72	.39	120
SCOP	90	FastFK	.57	.94	.71	149	.53	.87	.66	149
		Randomness	.36	.61	.45	151	.36	.61	.45	151
		HoPF	.70	.94	.80	121	.63	.82	.71	118
MusicBrainz	168	FastFK	.33	.74	.46	368	.28	.62	.39	367
		Randomness	.24	.49	.32	341	.24	.49	.32	341
		HoPF	.54	.96	.69	296	.28	.50	.36	289

In HoPF and Chen et al. (2014), column name similarity between LHS and RHS of a foreign key candidate is set as a foreign key feature while non-matching column names of LHS and RHS is used as a postprocessing (Zhang et al. 2010). We use four metrics to measure the performances: precision (P), recall (R), F-1 score, and the number of predicted foreign keys (Predicted FKs)

measures the data distribution between LHS and RHS with a so-called *randomness* measure. *FastFK*, which assumes that there exists only single-attribute foreign keys, employs a few features to score foreign key candidates, as well as some pruning rules to decrease the search space. For *Randomness*, we applied $\theta = 0.9$, and bottom 256 sketches, 256 and 16 quantiles for unary and n-ary foreign keys candidates, respectively, – the sweet spots determined by the original authors. As both previous works assume that primary keys are present and known, we provide them with true primary keys in this experiment setting.

To improve results, Randomness also considers column names, keeping only candidates with exactly matched names. The authors apply this technique only to TPC-H and TPC-E, relying on external documentation to guide a manual trimming of column labels before the matching. We compared their results with HoPF and *FastFK* when switching on the *column name* features. We also compared the result of the Randomness without conducting this postprocessing with the other two when switching off the *column name* features. Table 7 displays the details of this comparison.

As seen in the table, without knowledge of column names, all three algorithms experience a drop in precision and recall, proving that column names are indicative in recognizing foreign keys. While using the column names, the performances of the three algorithms are quite similar on synthetic datasets (*TPC-H*, *TPC-E*, and *AdventureWorks*), while HoPF outperforms the two baseline approaches in the real-world datasets (*SCOP* and *MusicBrainz*). We also notice that HoPF produces smaller predicted foreign key sets for all the datasets compared to the two baseline approaches, making any post-processing by a human expert easier.

8 Conclusions

Primary keys and foreign keys are important integrity constraints to keep databases consistent. However, data stored as flat files or dumps do not always carry these constraint definitions and in many cases, it is up to the user of the data to identify them and thus understand the data better and enforce its quality. As schemata can be quite large and complex, automatic discovery of primary and foreign keys is a relevant (and challenging) research topic.

Previous efforts were made to discover foreign keys and primary keys separately. In this work, we have proposed the HoPF algorithm to integrate primary key and foreign key detection holistically. We employ a set of carefully designed features to score and distinguish both the true primary keys and foreign keys from the spurious UCCs and INDs. We employ several useful pruning rules to effectively reduce the search spaces of both PKs and FKs.

In performance experiments on five diverse datasets, our algorithm reaches an average recall of 88% and 91% in primary keys and foreign keys discovery, respectively. We show with an experiment that without knowledge of primary keys (which is assumed in related work), the performance of foreign key discovery is much worse, indicating the necessity to discover primary keys in advance or simultaneously. We compared precision and recall with the state-of-art algorithms.

As future work we plan to mesh our work on choosing good keys and foreign keys into the lattice-based UCC and IND detection algorithms mentioned in Section 2.1, providing a powerful (yet inexact) pruning mechanism for their large spaces.

References

- Abedjan, Z., Golab, L., Naumann, F. (2015). Profiling relational data: a survey. *VLDB Journal*, 24(4), 557–581.
- Bhattacharyya, A. (1943). On a measure of divergence between two statistical populations defined by their probability distributions. *Bulletin of the Calcutta Mathematical Society*, 35, 99–109.
- Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R. (2003). Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the international conference on management of data (SIGMOD)* (pp. 313–324).
- Chen, Z., Narasayya, V.R., Chaudhuri, S. (2014). Fast foreign-key detection in microsoft SQL server powerpivot for excel. *Proceedings of the VLDB Endowment*, 7(13), 1417–1428.
- Faust, M., Schwalb, D., Plattner, H. (2014). Composite group-keys – space-efficient indexing of multiple columns for compressed in-memory column stores. In *Memory data management and analysis - first and second international workshops, revised selected papers* (pp. 139–150).
- Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A. (2004). CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the international conference on management of data (SIGMOD)* (pp. 647–658).
- Kantola, M., Mannila, H., Räihä, K., Siirtola, H. (1992). Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligence Systems*, 7(7), 591–607.
- Lopes, S., Petit, J., Toumani, F. (2002). Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems (IS)*, 27(1), 1–19.
- Lucchesi, C.L., & Osborn, S.L. (1978). Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2), 270–279.
- Marchi, F.D., Lopes, S., Petit, J. (2009). Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information System*, 32(1), 53–73.
- Memari, M., Link, S., Dobbie, G. (2015). SQL data profiling of foreign keys. In *Proceedings of the international conference on conceptual modeling (ER)* (pp. 229–243).
- Papenbrock, T., Bergmann, T., Finke, M., Zwiener, J., Naumann, F. (2015). Data profiling with metanome. *Proceedings of the VLDB Endowment*, 8(12), 1860–1863.
- Papenbrock, T., & Naumann, F. (2017). Data-driven schema normalization. In *Proceedings of the international conference on extending database technology (EDBT)* (pp. 342–353).
- Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., Leser, U. (2009). A machine learning approach to foreign key discovery. In *Proceedings of the ACM SIGMOD workshop on the web and databases (WebDB)*.
- Rubner, Y., Tomasi, C., Guibas, L.J. (1998). A metric for distributions with applications to image databases. In *Proceedings of the international conference on computer vision (ICCV)* (pp. 59–66).
- Tschirschnitz, F., Papenbrock, T., Naumann, F. (2017). Detecting inclusion dependencies on very many tables. *ACM Transactions on Database Systems (TODS)*, 42(3), 18:1–18,29.
- Venetis, P., Halevy, A.Y., Madhavan, J., Pasca, M., Shen, W., Wu, F., Miao, G., Wu, C. (2011). Recovering semantics of tables on the web. *Proceedings of the VLDB Endowment*, 4(9), 528–538.
- Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D. (2010). On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1–2), 805–814.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.