Data-driven Schema Normalization

Thorsten Papenbrock
Hasso Plattner Institute (HPI)
14482 Potsdam, Germany
thorsten.papenbrock@hpi.de

Felix Naumann
Hasso Plattner Institute (HPI)
14482 Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Ensuring Boyce-Codd Normal Form (BCNF) is the most popular way to remove redundancy and anomalies from datasets. Normalization to BCNF forces functional dependencies (FDs) into keys and foreign keys, which eliminates duplicate values and makes data constraints explicit. Despite being well researched in theory, converting the schema of an existing dataset into BCNF is still a complex, manual task, especially because the number of functional dependencies is huge and deriving keys and foreign keys is NP-hard.

In this paper, we present a novel normalization algorithm called Normalize, which uses discovered functional dependencies to normalize relational datasets into BCNF. Normalize runs entirely data-driven, which means that redundancy is removed only where it can be observed, and it is (semi-)automatic, which means that a user may or may not interfere with the normalization process. The algorithm introduces an efficient method for calculating the closure over sets of functional dependencies and novel features for choosing appropriate constraints. Our evaluation shows that Normalize can process millions of FDs within a few minutes and that the constraint selection techniques support the construction of meaningful relations during normalization.

1. FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a statement of the form $X \rightarrow A$ with X being a set of attributes and A being a single attribute from the same relation R. We say that the left-hand-side (Lhs) X functionally determines the right-hand-side (Rhs) A. This means that whenever two records in an instance r of R agree on all their X values, they must also agree on their A value [7]. More formally, an FD $X \rightarrow A$ holds in r, iff $\forall t_1, t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1[A] = t_2[A]$. In the following, we consider only *non-trivial* FDs, which are FDs with $A \notin X$.

Table 1 depicts an example address dataset for which the two functional dependencies Postcode→City and Postcode→Mayor hold. Because both FDs have the same Lhs, we can aggregate them to the notation Postcode→City,Mayor. The presence of this FD introduces anomalies in the dataset, because the values `Potsdam`, `Frankfurt`, `Jakobs`, and `Feldmann` are stored redundantly and updating these values might cause inconsistencies. So if, for instance, some Mr. Schmidt was elected as the new mayor of Potsdam, we must correctly change all three occurrences of `Jakobs` to `Schmidt`.

Table 1: Example address dataset

First	Last	Postcode	City	Mayor
Thomas	Miller	14482	Potsdam	Jakobs
Sarah	Miller	14482	Potsdam	Jakobs
Peter	Smith	60329	Frankfurt	Feldmann
Jasmine	Cone	01069	Dresden	Orosz
Mike	Cone	14482	Potsdam	Jakobs
Thomas	Moore	60329	Frankfurt	Feldmann

Such anomalies can be avoided by normalizing relations into the Boyce-Codd Normal Form (BCNF). A relational schema R is in BCNF, iff for all FDs $X \rightarrow A$ in R the Lhs X is either a key or superkey [7]. Because Postcode is neither a key nor a superkey in the example dataset, this relation does not meet the BCNF condition. To bring all relations of a schema into BCNF, one has to perform six steps, which are explained in more detail later: (1) discover all FDs, (2) extend the FDs, (3) derive all necessary keys from the extended FDs, (4) identify the BCNF-violating FDs, (5) select a violating FD for decomposition (6) split the relation according to the chosen violating FD. The steps (3) to (5) repeat until step (4) finds no more violating FDs and the resulting schema is BCNF-conform. We find several FD discovery algorithms, such as Tane [14] and HyFD [19], that serve step (1), but there are, thus far, no algorithms available to efficiently and automatically solve the steps (2) to (6).

For the example dataset, an FD discovery algorithm would find twelve valid FDs in step (1). These FDs must be aggregated and transitively extended in step (2) so that we find, inter alia, First,Last→Postcode,City,Mayor and Postcode→City,Mayor. In step (3), the former FD lets us derive the key {First, Last}, because these two attributes functionally determine all other attributes of the relation. Step (4), then, determines that the second FD violates the BCNF condition, because its Lhs Postcode is neither a key nor superkey. If we assume that step (5) is able to automatically select the second FD for decomposition, step (6) decomposes the example relation into R_1(First, Last, Postcode) and R_2(Postcode, City, Mayor) with {First, Last} and {Postcode} being primary keys and R_1.Postcode→R_2.Postcode a foreign key constraint. Table 2 shows this result. When again checking for violating FDs, we do not find any and stop the nor-

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Table 2: Normalized example address dataset

First	Last	Postcode
Thomas	Miller	14482
Sarah	Miller	14482
Peter	Smith	60329
Jasmine	Cone	01069
Mike	Cone	14482
Thomas	Moore	60329

Postcode	City	Mayor
14482	Potsdam	Jakobs
60329	Frankfurt	Feldmann
01069	Dresden	Orosz

malization process with a BCNF-conform result. Note that the redundancy in City and Mayor has been removed and the total size of the dataset was reduced from 36 to 27 values.

Because memory became a lot cheeper in the last years, there is a trend of not normalizing datasets for performance reasons. Normalization is, hence, today often claimed to be obsolete. This claim is false and ignoring normalization is dangerous for the following reasons [8]:

1. Normalization removes redundancy and, in this way, decreases error susceptibility and memory consumption. While memory might be relatively cheep, data errors can have serious and expensive consequences and should be avoided at all costs.

2. Normalization does not necessarily decrease query performance; in fact, it can even increase the performance. Some queries might need some additional joins after normalization, but others can read the smaller relations much faster. Also, more focused locks can be set, increasing parallel access to the data, if the data has to be changed. So the performance impact of normalization is not determined by the normalized dataset but by the application that uses it.

3. Normalization increases the understanding of the schema and of queries against this schema: Relations become smaller and closer to the entities they describe; their complexity decreases making them easier to maintain and extend. Furthermore, queries against the relations become easier to formulate and many mistakes are easier to avoid. For instance, aggregations over columns with redundant values are hard to formulate correctly.

In summary, normalization should be the default and denormalization a conscious decision, i.e., "we should denormalize only at a last resort [and] back off from a fully normalized design only if all other strategies for improving performance have failed, somehow, to meet requiremnts", C. J. Date, p. 88 [8].

The objective of this work is to normalize a given relational instance into Boyce-Codd Normal Form. Note that we do not aim to recover a certain schema nor do we aim to design a new schema using business logic. To solve the normalization task, we propose a data-driven, (semi-)automatic normalization algorithm that removes all FD-related redundancy while still providing full information recoverability. Being data-driven means that all FDs used in the normalization process are extracted directly from the data and that all decomposition proposals are based solely on data-characteristics. In other words, we consider only redundancy that can actually be observed in a given relational instance.

The advantage of a data-driven normalization approach over state-of-the-art schema-driven approaches is that it can use the data to expose all syntactically valid normalization options, i.e., functional dependencies with evidence in the data, so that the algorithm (or the user) must only decide for a normalization path and not find one. The number of FDs can, indeed, become large, but we show that an algorithm can effectively propose the semantically most appropriate options. Furthermore, knowing all FDs allows for a more efficient normalization algorithm as opposed to having only a subset of FDs.

Research challenges. In contrast to the vast amount of research on normalization in the past decades, we do not assume that the FDs are given, because this is almost never the case in practice. We also do not assume that a human data expert is able to manually identify them, because the search is difficult by nature and the actual FDs are often not obvious. The FD Postcode→City from our example, for instance, is commonly believed to be true although it is usually violated by exceptions where two cities share the same postcode; the FD Atmosphere→Rings, on the other hand, is difficult to discover for a human but in fact holds on various datasets about planets. For this reason, we automatically discover all (minimal) FDs. This introduces a new challenge, because we now deal with much larger, often spurious, but complete sets of FDs.

Using all FDs of a particular relational instance in the normalization process further introduces the challenge of selecting appropriate keys and foreign keys from the FDs (see Step (5)), because most of the FDs are coincidental, i.e., they are syntactically true but semantically false. This means that when the data changes these semantically invalid FDs could be violated and, hence, no longer work as a constraint. So we introduce features to automatically identify (and choose) reliable constraints from the set of FDs, which is usually too large for a human to manually examine.

Even if all FDs are semantically correct, selecting appropriate keys and foreign keys is still difficult. The decisions made here define which decompositions are executed, because decomposition options are often mutually exclusive: If, for instance, two violating FDs overlap, one split can make the other split infeasible. This happens, because BCNF normalization is not dependency preserving [12]. In all these constellations, however, some violating FDs are semantically better choices than others, which is why violating FDs must not only be filtered but also ranked by such quality features.

Another challenge, besides guiding the normalization process in the right direction, is the computational complexity of the normalization. Beeri and Bernstein have proven that the question "Given a set of FDs and a relational schema that embodies it, does the schema violate BCNF?" is NP-complete in the number of attributes [3]. To test this, we need to check that the Lhs of each of these FDs is a key or a super key, i.e., if each Lhs determines all other attributes. This is trivial if all FDs are transitively fully extended, i.e., they are transitively closed. For this reason, the complexity lies in calculating these closures (see Step (2)). Because no current algorithm is able to solve the closure calculation efficiently, we propose novel techniques for this sub-task of schema normalization.

Overall, the number of functional dependencies in datasets is typically much greater than a human expert can manually cope with [18]. A normalization algorithm must, therefore, be able to handle such very large inputs automatically.

Contributions. We propose a novel, instance-based schema normalization algorithm called NORMALIZE that can perform the normalization of a relational dataset automatically or supervised by an expert. Being able to put a human in the loop enables the algorithm to combine its analytical strengths with the domain knowledge of an expert. With NORMALIZE and this paper, we make the following contributions:

a) *Schema normalization.* We show how the entire schema normalization process can be implemented as one algorithm, which no previous work has done before. We discuss each component of this algorithm in detail. The main contribution of our (semi-)automatic approach is to incrementally weed out semantically false FDs by focusing on those FDs that are most likely true.

b) *Closure calculation.* We present two efficient closure algorithms, one for general FD result sets and one for complete result sets. Their core innovations include a more focused extension procedure, the use of efficient index-structures, and parallelization. These algorithms are not only useful in the normalization context, but also for many other FD-related tasks, such as query optimization, data cleansing, or schema reverse-engineering.

c) *Violation detection.* We propose a compact data structure, i.e., a prefix tree, to efficiently detect FDs that violate BCNF. This is the first approach to algorithmically improve this step. We also discuss how this step can be changed to discover violating FDs for normal forms other than BCNF.

d) *Constraint selection.* We contribute several features to rate the probability of key and foreign key candidates for actually being constraints. With the results, the candidates can be ranked, filtered, and selected as constraints during the normalization process. The selection can be done by either an expert or by the algorithm itself. Because all previous works on schema normalization assumed all input FDs to be correct, this is the first solution for a problem that has been ignored until now.

e) *Evaluation.* We evaluate our algorithms on several datasets demonstrating the efficiency of the closure calculation on complete, real-world FD result sets and the feasibility of (semi-)automatic schema normalization.

The remainder of this paper is structured as follows: First, we discuss related work in Section 2. Then, we introduce the schema normalization algorithm NORMALIZE in Section 3. The following sections go into more detail explaining the closure calculation in Section 4, the key derivation in Section 5, and the violation detection in Section 6. Section 7, then, introduces assessment techniques for key and foreign key candidates. The normalization algorithm is finally evaluated in Section 8 and we conclude in Section 9.

2. RELATED WORK

Normal forms for relational data have been extensively studied since the proposal of the relational data model itself [6]. For this reason, many normal forms have been proposed. Instead of giving a survey on normal forms here, we refer the interested reader to [10]. The Boyce-Codd Normal Form (BCNF) [7] is the most popular normal form, because it removes most kinds of redundancy from relational schemata. This is why we focus on this particular normal form in this paper. Most of the proposed techniques can, however, likewise be used to create other normal forms. The idea for our normalization algorithm follows the BCNF decomposition algorithm proposed in [12] and many other text books on database systems. The algorithm eliminates all anomalies related to functional dependencies while still guaranteeing full information recoverability via natural joins.

Schema normalization and especially the normalization into BCNF are well studied problems [3, 5, 16]. Bernstein presents a complete procedure for performing schema synthesis based on functional dependencies [4]. In particular, he shows that calculating the closure over a set of FDs is a crucial step in the normalization process. He also lays the theoretical foundation for our paper. But like most other works on schema normalization, Bernstein takes the functional dependencies and their semantic validity as a given – an assumption that hardly applies, because FDs are usually hidden in the data and must be discovered. For this reason, existing works on schema normalization greatly underestimate the number of valid FDs in non-normalized datasets and they also ignore the task of filtering the syntactically correct FDs for semantically meaningful ones. These reasons make those normalization approaches inapplicable in practice. In this paper, we propose a normalization system that covers the entire process from FD discovery over constraint selection up to the final relation decomposition. We show the feasibility of this approach in practical experiments.

There are other works on schema normalization, such as the work of Diederich and Milton [9], who understood that calculating the transitive closure over the FDs is a computational complex task that becomes infeasible facing real-world FD sets. As a solution, they propose to remove so called *extraneous* attributes from the FDs before calculating the closure, which reduces the calculation costs significantly. However, if all FDs are minimal, which is the case in our normalization process, then no *extraneous* attributes exist, and the proposed pruning strategy is futile.

One important difference between traditional normalization approaches and our algorithm is that we retrieve *all minimal FDs* from a given relational instance to exploit them for closure calculation (syntactic step) and constraint selection (semantic step). The latter has received little attention in previous research. In [2], Andritsos et al. proposed to rank the FDs used for normalization by the entropy of their attribute sets: The more duplication an FD removes, the better it is. The problem with this approach is that it weights the FDs only for effectiveness and not for semantic relevance. Entropy is also expensive to calculate, which is why we use different features. In fact, we use techniques inspired by [20], who extracted foreign keys from inclusion dependencies.

Schema normalization is a sub-task in schema design and evolution. There are numerous database administration tools, such as Navicat[1], Toad[2], and MySQL Workbench[3], that support these overall tasks. Most of them transform a given schema into an ER-diagram that a user can manipulate. All manipulations are then translated back to the schema and its data. Such tools are partly able to support normalization processes, but none of them can automatically propose normalizations with FDs retrieved from the data.

[1]https://www.navicat.com/
[2]http://www.toadworld.com/
[3]http://www.mysql.com/products/workbench/

In [3], the authors propose an efficient algorithm for the membership problem, i.e., the problem of testing whether one given FD is in the cover or not. This algorithm does not solve the closure calculation problem, but the authors propose some improvements in that algorithm that our improved closure algorithm uses as well, e.g., testing only for missing attributes on the Rhs. They also propose derivation trees as a model for FD derivations, i.e., deriving further FDs from a set of known FDs using Armstrong's inference rules. Because no algorithm is given for their model, we cannot compare our solution against it.

As stated above, the discovery of functional dependencies from relational data is a prerequisite for schema normalization. Fortunately, FD discovery is a well researched problem and we find various algorithms to solve it. In this work, we utilize the HyFD algorithm, which is the most efficient FD discovery algorithm at the time [19]. This algorithm discovers – like almost all FD discovery algorithms – the complete set of all minimal, syntactically valid FDs in a given relational dataset. We exploit these properties, i.e., minimality and completeness in our closure algorithm.

3. SCHEMA NORMALIZATION

To normalize a schema into Boyce-Codd Normal Form (BCNF), we implement the straightforward BCNF decomposition algorithm shown in most textbooks on database systems, such as [12]. The BCNF-conform schema produced by this algorithm is always a tree-shaped *snowflake schema*, i.e., the foreign key structure is hierarchical and cycle-free. For this reason, our normalization algorithm is not designed to (re-)construct arbitrary non-snowflake schemata. It, however, removes all redundancy related to functional dependencies from the relations. If other schema design decisions that lead to alternative schema topologies are necessary, the user must (and can!) interactively choose different decompositions other than the ones our algorithm can propose.

In the following, we propose a normalization process that takes an arbitrary relational instance as input and returns a BCNF-conform schema for it. The input dataset can contain one or more relations, and no other metadata than the dataset's schema is required. This schema, which is incrementally changed during the normalization process, is globally known to all algorithmic components. We refer to a dataset's *schema* as its set of relations, specifying attributes, tables, and key/foreign key constraints. For instance, the schema of our example dataset in Table 2 is $\{R_1(\underline{\text{First}}, \underline{\text{Last}}, \text{Postcode}), R_2(\underline{\text{Postcode}}, \text{City}, \text{Mayor})\}$. Underlined attributes represent keys and same attribute names represent foreign keys.

Figure 1 gives an overview of the normalization algorithm, which we call Normalize. In contrast to other normalization algorithms, such as those proposed in [4] or [9], Normalize does not have any components responsible for minimizing FDs or removing extraneous FDs. This is because the set of FDs on which we operate, is not arbitrary; it contains only minimal and, hence, no extraneous FDs due to the FD discovery step. We now introduce the components step by step and discuss the entire normalization process.

(1) FD Discovery. Given a relational dataset, the first component is responsible for discovering all minimal functional dependencies. Any known FD discovery algorithm, such as Tane [14] or Dfd [1], can be used, because all these algorithms are able to discover the complete set of minimal

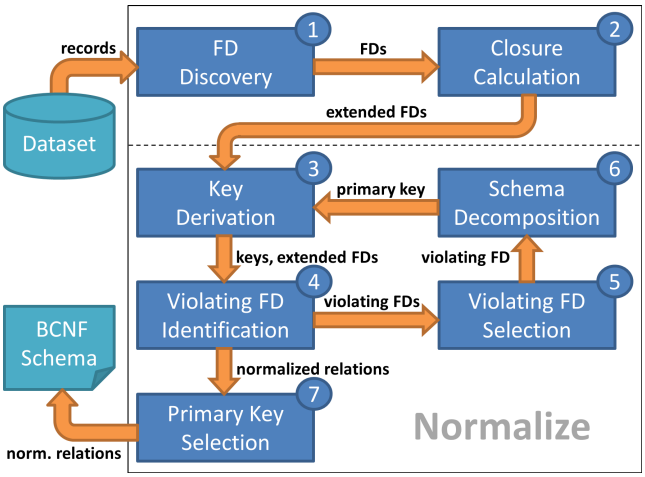

Figure 1: "Normalize" and its components.

FDs in relational datasets. We make use of our HyFD [19] algorithm here, because it is the most efficient algorithm for this task and it offers special pruning capabilities that we can exploit later in the normalization process. In summary, the first component reads the data, discovers all FDs, and sends them to the second component. For more details on this discovery step, we refer to [19].

(2) Closure Calculation. The second component calculates the closure over the given FDs. The closure is needed by subsequent components to infer keys and normal form violations. Formally, the *closure* X_F^+ *over a set of attributes* X given the FDs F is defined as the set of attributes X plus all additional attributes Y that we can add to X using F and Armstrong's transitivity axiom [9]. If, for example, $X = \{A, B\}$ and $F = \{A \rightarrow C, C \rightarrow D\}$, then $X_F^+ = \{A, B, C, D\}$. We now define the *closure* F^+ *over a set of FDs* F as a set of extended FDs: The Rhs Y of each FD $X \rightarrow Y \in F$ is extended such that $X \cup Y = X_F^+$. In other words, each FD in F is maximized using Armstrong's transitivity axiom. Because, as Beeri et al. have shown [3], this is an NP-hard task with respect to the number of attributes in the input relation, we shall propose an efficient FD extension algorithm that finds transitive dependencies via prefix tree lookups. This algorithm iterates the set of FDs only once and is able to parallelize its work. It exploits the fact that the given FDs are minimal and complete (Section 4).

(3) Key Derivation. The key derivation component collects those keys from the extended FDs that the algorithm requires for schema normalization. Such a key X is a set of attributes for which $X \rightarrow Y \in F^+$ and $X \cup Y = R_i$ with R_i being *all* attributes of relation i. In other words, if X determines all other attributes, it is a key for its relation. Once discovered, these keys are passed to the next component. Our method of deriving keys from extended functional dependencies does not reveal all existing keys in the schema, but we prove in Section 5 that only the derived keys are needed for BCNF normalization.

(4) Violating FD Identification. Given the extended FDs and the set of keys, the violation detection component checks all relations for being BCNF-conform. Recall that a relation R is BCNF-conform, iff for all FDs $X \rightarrow A$ in

that relation the Lhs X is either a key or superkey. So Normalize checks the Lhs of each FD for having a (sub)set in the set of keys; if no such (sub)set can be found, the FD is reported as a BCNF violation. Note that one could setup other normalization criteria in this component to accomplish 3NF or other normal forms. If FD violations were identified, these are reported to the next component; otherwise, the schema is BCNF-conform and can be sent to the primary key selection. We propose an efficient technique to find all violating FDs in Section 6.

(5) Violating FD Selection. The violating FD selection component is called with a set of violating FDs, if some relations are not yet in BCNF. In this case, the component scores all violating FDs for being good foreign key constraints. With these scores, the algorithm creates a ranking of violating FDs for each non-BCNF relation. From each ranking, a user picks the most suitable violating FD for normalization; if no user is present, the algorithm automatically picks the top ranked FD. Note that the user, if present, can also decide to pick none of the FDs, which ends the normalization process for the current relation. This is reasonable if all presented FDs are obviously semantically incorrect, i.e., the FDs hold on the given data accidentally but have no real meaning. Such FDs are presented with a relatively low score at the end of the ranking. Eventually, the iterative process automatically weeds out most of the semantically incorrect FDs by selecting only semantically reliable FDs in each step. We discuss the violating FD selection together with the key selection in Section 7.

(6) Schema Decomposition. Knowing the violating FDs, the actual schema decomposition is a straight-forward task: Each relation R, for which a violating FD $X \rightarrow Y$ is given, is split into two parts – one part without the redundant attributes $R_1 = R \backslash Y$ and one part with the FD's attributes $R_2 = X \cup Y$. Now X automatically becomes the new primary key in R_2 and a foreign key in R_1. With these new relations, the algorithm goes back into step (3), the key selection, because new keys might have appeared in R_2, namely those keys Z for which $Z \rightarrow X$ holds. Because the decomposition itself is straightforward, we do not go into more detail for this component in this paper.

(7) Primary Key Selection. The primary key selection is the last component in the normalization process. It makes sure that every BCNF-conform relation has a primary key constraint. Because the decomposition component already assigns keys and foreign keys when splitting relations, most relations already have a primary key. Only those relations that had no primary key at the beginning of the normalization process are processed by this component. For them, the algorithm assigns a primary key in a (semi-)automated way: All keys of the respective relation are scored for being a good primary key; then the keys are ranked by their score and either a human picks a primary key from this ranking, or the algorithm automatically picks the highest ranked key as the relation's primary key. Section 7 describes the scoring and selection of keys in more detail.

Once the closure of all FDs is calculated, the components (3) to (6) form a loop: This loop drives the normalization process until component (4) finds the schema to be BCNF-conform. Overall, the proposed components can be grouped into two classes: The first class includes the components (1), (2), (3), (4), and (6) and operates on a syntactic level; the results in this class are well defined and the focus is set on performance optimization. The second class includes the components (5) and (7) and operates on a semantic level; the computations here are easy to execute but the choices are difficult and the quality of the result matters.

4. CLOSURE CALCULATION

Armstrong formulated the following three axioms for functional dependencies on attribute sets X, Y, and Z [3]:

1. *Reflexivity*: If $Y \subseteq X$, then $X \rightarrow Y$.
2. *Augmentation*: If $X \rightarrow Y$, then $X \cup Z \rightarrow Y \cup Z$.
3. *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

For schema normalization, we are given a set of FDs F and need to find a cover F^+ that maximizes the right hand side of each FD in F. The maximization of FDs is important to identify keys and to decompose relations correctly. In our running example, for instance, we might be given Postcode→City and City→Mayor. A correct decomposition with foreign key Postcode requires Postcode→City,Mayor; otherwise we would lose City→Mayor, because the attributes City and Mayor would end up in different relations. Therefore, we apply Armstrong's transitivity axiom on F to calculate its cover F^+.

The closure F^+ extends each FD using Armstrong's reflexivity and transitivity axioms. Augmentation need not be used, because this rule generates new, non-minimal FDs instead of extending existing ones. The decomposition steps require the FDs' Lhs to be minimal, i.e., removing any attribute from X would invalidate $X \rightarrow Y$, because X should become a *minimal* key after decomposition.

The reflexivity axiom adds all Lhs attributes to an FD's Rhs. To reduce memory consumption, we make this extension only implicit: We assume that Lhs attributes always also belong to an FD's Rhs without explicitly storing them on that side. For this reason, we apply the transitivity axiom for attribute sets W, X, Y, and Z as follows: If $W \rightarrow X$, $Y \rightarrow Z$, and $Y \subseteq W \cup X$, then $W \rightarrow Z$. So if, for instance, the FD First,Last→Mayor is given, we can extend the FD First,Postcode→Last with the Rhs attribute Mayor, because {First, Last} ⊆ {First, Postcode} ∪ {Last}.

In the following, we discuss three algorithms for calculating F^+ from F: A naive algorithm, an improved algorithm for arbitrary sets of FDs, and an optimized algorithm for complete sets of minimal FDs. While the second algorithm might be useful for closure calculation in other contexts, such as query optimization or data cleansing, we recommend the third algorithm for our normalization system. All three algorithms store F, which is transformed into F^+, in the variable *fds*.

4.1 Naive closure algorithm

The naive closure algorithm, which was already introduced as such in [9], is given as Algorithm 1. For each functional dependency in *fds* (Line 3), the algorithm iterates all other FDs (Line 4) and tests if these extend the current FD (Line 5). If an extension is possible, the current FD is updated (Line 6). These updates might enable further updates for already tested FDs. For this reason, the naive algorithm iterates the FDs until an entire pass has not added any further extensions (Line 8).

Algorithm 1: Naive Closure Calculation

```
Data: fds
Result: fds
1 do
2     somethingChanged ← false;
3     foreach fd ∈ fds do
4         foreach otherFd ∈ fds do
5             if otherFd.lhs ⊆ fd.lhs ∪ fd.rhs then
6                 fd.rhs ← fd.rhs ∪ otherFd.rhs;
7                 somethingChanged ← true;
8 while somethingChanged ;
9 return fds;
```

4.2 Improved closure algorithm

There are several ways to improve the naive closure algorithm, some of which have already been proposed in similar form in [9] and [3]. We now present an improved closure algorithm that solves the following three issues: First, the algorithm should not check all other FDs when extending one specific FD, but only those that can possibly link to a missing Rhs attribute. Second, when looking for a missing Rhs attribute, the algorithm should not check all other FDs that can provide it, but only those that have a subset-relation with the current FD, i.e., those that are relevant for extensions. Third, the change-loop should not iterate the entire FD set, because some FDs must be extended more often than others so that many extension tests are executed superfluously.

Algorithm 2 shows our improved version. First, we remove the nested loop over all other FDs and replace it with index lookups. The index structure we propose is a set of prefix-trees, aka. tries. Each trie stores all FD Lhss that have the same, trie-specific Rhs attribute. Having an index for each Rhs attribute allows the algorithm to check only those other FDs that can deliver a link to a Rhs attribute that a current FD is actually missing (Line 8).

The *lhsTries* are constructed before the algorithm starts extending the given FDs (Lines 1 to 4). Each index-lookup must then not iterate all FDs referencing the missing Rhs attribute; it instead performs a subset search in the according prefix tree, because the algorithm is specifically looking for an FD whose Lhs is contained in the current FD's Rhs attributes (Line 9). The subset search is much more effective than iterating all possible extension candidates and has already been proposed for FD generalization lookups in [11].

As a third optimization, we propose to move the change-loop inside the FD-loop (Line 6). Now, a single FD that requires many transitive extensions in subsequent iterations does not trigger the same number of iterations over all FDs, which mostly are already fully extended.

4.3 Optimized closure algorithm

Algorithm 2 works well for all sets of FDs, but we can further optimize the algorithm with the assumption that these sets contain *all minimal FDs*. Algorithm 3 shows this more efficient version for complete sets of minimal FDs.

Like Algorithm 2, the optimized closure algorithm also uses the Lhs tries for efficient FD extensions, but it does not require a change-loop so that it iterates the missing Rhs attributes of an FD only once. The algorithm also checks

Algorithm 2: Improved Closure Calculation

```
Data: fds
Result: fds
1  array lhsTries size | schema.attributes | as trie;
2  foreach fd ∈ fds do
3      foreach rhsAttr ∈ fd.rhs do
4          lhsTries[rhsAttr].insert(fd.lhs);
5  foreach fd ∈ fds do
6      do
7          somethingChanged ← false;
8          foreach attr ∉ fd.rhs ∪ fd.lhs do
9              if fd.lhs ∪ fd.rhs ⊇ lhsTries[attr] then
10                 fd.rhs ← fd.rhs ∪ attr;
11                 somethingChanged ← true;
12     while somethingChanged ;
13 return fds;
```

only the Lhs attributes of an FD for subsets and not all attributes of a current FD (Line 7). These two optimizations are possible, because the set of FDs is complete and minimal so that we always find a subset-FD for any valid extension attribute. The following lemma states this formally:

Lemma 1. *Let F be a complete set of minimal FDs. If $X \to Y \in F$ and $X \to A$ with $A \notin Y$ is valid, then there must exist an $X' \subset X$ so that $X' \to A \in F$.*

Proof. If $X \to A$ and $X \to A \notin F$, then $X \to A$ is not minimal and a minimal FD $X' \to A$ with $X' \subset X$ must exist. If $X' \to A \notin F$, then F is not a complete set of minimal FDs, which contradicts the premise that F is complete. □

The fact that all minimal FDs are required for Algorithm 3 to work correctly has the disadvantage that complete sets of FDs are usually much larger than sets of FDs that have already been reduced to meaningful FDs. Reducing a set of FDs to meaningful ones is, on the contrary, a difficult and use-case specific task that becomes more accurate if the FDs' closure is known. For this reason, we perform the closure calculation before the FD selection and accept the increased processing time and memory consumption.

The increased processing time is hardly an issue, because the performance gain of Algorithm 3 over Algorithm 2 on same sized inputs is so significant that larger sets of FDs can still easily be processed. We show this in Section 8. The

Algorithm 3: Optimized Closure Calculation

```
Data: fds
Result: fds
1  array lhsTries size | schema.attributes | as trie;
2  foreach fd ∈ fds do
3      foreach rhsAttr ∈ fd.rhs do
4          lhsTries[rhsAttr].insert(fd.lhs);
5  foreach fd ∈ fds do
6      foreach attr ∉ fd.rhs ∪ fd.lhs do
7          if fd.lhs ⊇ lhsTries[attr] then
8              fd.rhs ← fd.rhs ∪ attr;
9  return fds;
```

increased memory consumption, on the other hand, becomes a problem if the complete set of minimal FDs is too large to be held in memory or maybe even too large to be held on disk. We then need to prune FDs, but which FDs can be pruned so that Algorithm 3 still computes a correct closure on the remainder? To fully extend an FD $X \rightarrow Y$, the algorithm requires all subset-FDs $X' \rightarrow Z$ with $X' \subset X$ to be available. So if we prune all superset-FDs with larger Lhs than $|X|$, the calculated closure for $X \rightarrow Y$ and all its subset-FDs $X' \rightarrow Z$ would still be correct. In general, we can define a maximum Lhs size and prune all FDs with a larger Lhs size while still being able to compute the complete and correct closure for the remaining FDs with Algorithm 3. This pruning fits our normalization use-case well, because FDs with shorter Lhs are semantically better candidates for key and foreign key constraints as we argue in Section 7. Normalize achieves the maximum Lhs size pruning for free, because it is already implemented in the HyFD algorithm that we proposed using for the FD discovery.

All three closure algorithms can easily be parallelized by splitting the FD-loops (Lines 3, 2, and 5 respectively) to different worker threads. This is possible, because each worker changes only its own FD and changes made to other FDs can, but do not have to be seen by this worker.

Considering the complexity of the three algorithms with respect to the number of input FDs, the naive algorithm is in $\mathcal{O}(|fds|^3)$, the improved in $\mathcal{O}(|fds|^2)$ and the optimized in $\mathcal{O}(|fds|)$. But because the number of FDs potentially increases exponentially with the number of attributes, all three algorithms are NP-complete in the number of attributes. We compare the algorithms experimentally in Section 8.

5. KEY DERIVATION

Keys are important in normalization processes, because they do not contain any redundancy due to their uniqueness. Hence, they do not cause anomalies in the data. Keys basically indicate normalized schema elements that do not need to be decomposed, i.e., decomposing them would not remove any redundancy in the given relational instance. In this section, we first discuss how keys can be derived from extended FDs. Then, we prove that the set of derived keys is sufficient for BCNF schema normalization.

Deriving keys from extended FDs. By definition, a key is any attribute or attribute combination whose values uniquely define all other records [6]. In other words, the attributes of a key X functionally determine all other attributes Y of a relation R. So given the extended FDs, the keys can easily be found by checking each FD $X \rightarrow Y$ for $X \cup Y = R$.

The set of keys that we can directly derive from the extended FDs does, however, not necessarily contain *all* minimal keys of a given relation. Consider here, for instance, the relations Professor(name, department, salary), Teaches(name, label), and Class(label, room, date) with Teaches being a join table for the n:m-relationship between Professor and Class. When we *denormalize* this schema by calculating R = Professor $\bowtie$ Teaches $\bowtie$ Class, we get R(name, label, department, salary, room, date) with primary key {name, label}. This key *cannot* directly be derived from the minimal FDs, because name,label$\rightarrow A$ is not a minimal FD for any $A \in R_i$; the two minimal FDs are name$\rightarrow$department,salary and label$\rightarrow$room,date.

Skipping missing keys. The discovery of missing keys is an expensive task, especially when we consider the number of FDs that can be huge for non-normalized datasets. The BCNF-normalization, however, only requires those keys that we can directly derive from the extended FDs. We can basically ignore the missing keys, because the algorithm checks normal form violations only with keys that are *subsets* of an FD's Lhs (see Section 6) and all such keys can directly be derived. The following lemma states this more formally:

Lemma 2. *If X' is a key and $X \rightarrow Y \in F^+$ is an FD with $X' \subseteq X$, then X' can directly be derived from F^+.*

Proof. Let X' be a key of relation R and let $X \rightarrow Y \in F^+$ be an FD with $X' \subseteq X$. To directly derive the key X' from F^+, we must prove the existence of an FD $X' \rightarrow Z \in F^+$ with $Z = R \setminus X'$.

X must be a minimal Lhs in some FD $X \rightarrow Y'$ with $Y' \subseteq Y$, because $X \rightarrow Y \in F^+$ and F is the set of all minimal FDs. Now consider the precondition $X' \subseteq X$: If $X' \subset X$, then $X \rightarrow Y \notin F^+$, because X is a key and, hence, it determines any attribute A that X could contain more than X'. Therefore, $X = X'$ must be true. At this point, we have that $X \rightarrow Y' \in F^+$ and $X = X'$. So $X' \rightarrow Y' \in F^+$ must be true as well, which also shows that $Y' = Y = Z$, because X' is a key. □

The key derivation component in Normalize in fact discovers only those keys that are relevant for the normalization process by checking $X \cup Y = R$ for each FD $X \rightarrow Y$. The primary key selection component in the end of the normalization process must, however, discover all keys for those relations that did not receive a primary key from any previous decomposition operation. For this task, we use the DUCC algorithm by Heise et al. [13], which is specialized in key discovery. The key discovery is an NP complete problem, but because the normalized relations are much smaller than the non-normalized starting relations, it is a fast operation at this stage of the algorithm.

6. VIOLATION DETECTION

Given the extended *fds* and the *keys*, detecting BCNF violations is straightforward: Each FD whose Lhs is neither a key nor a super-key must be classified as a violation. Algorithm 4 shows how this can be efficiently done again using a prefix tree for subset searches.

At first, the violation detection algorithm inserts all given keys into a trie (Lines 1 to 3). Then, it iterates the *fds* and, for each FD, it checks if the FD's Lhs contains a `null` value $\perp$. Such FDs do not need to be considered for decompositions, because the Lhs becomes a primary key constraint in the new, split off relation and SQL prohibits `null` values in key constraints. Note that there is work on *possible/certain key constraints* that permit $\perp$ values in keys [15], but we continue with the standard for now. If the Lhs contains no `null` values, the algorithm queries the *keyTrie* for subsets of the FD's Lhs (Line 8). If a subset is found, the FD does not violate BCNF and we continue with the next FD; otherwise, the FD violates BCNF.

To preserve existing constraints, we remove all primary key attributes from a violating FD's Rhs, if a primary key is present (Line 11). Not removing the primary key attributes from the FD's Rhs could cause the decomposition step to break the primary key apart. Some key attributes would

Algorithm 4: Violation Detection

```
Data: fds, keys
Result: violatingFds
1  keyTrie ← new trie;
2  foreach key ∈ keys do
3      keyTrie.insert(key);
4  violatingFds ← ∅;
5  foreach fd ∈ fds do
6      if ⊥ ∈ fd.lhs then
7          continue;
8      if fd.lhs ⊇ keyTrie then
9          continue;
10     if currentSchema.primaryKey ≠ null then
11         fd.rhs ← fd.rhs − currentSchema.primaryKey;
12     if ∃ fk ∈ currentSchema.foreignKeys:
13         (fk ∩ fd.rhs ≠ ∅) ∧ (fk ⊈ fd.lhs ∪ fd.rhs) then
14         continue;
15     violatingFds ← violatingFds ∪ fd;
16 return violatingFds;
```

then be moved into another relation breaking the primary key constraint and possible foreign key constraints referencing this primary key. Because the current schema might also contain foreign key constraints, we test if the violating FD preserves all such constraints when used for decomposition: Each foreign key *fk* must stay intact in either of the two new relations or otherwise we do not use the violating FD for normalization (Line 12). The algorithm finally adds each constraint preserving violating FD to the *violatingFds* result set (Line 15). In Section 7 we propose a method to select one of them for decomposition.

When a violating FD $X \rightarrow Y$ is used to decompose a relation R, we obtain two new relations, which are $R_1(R \backslash Y \cup X)$ and $R_2(X \cup Y)$. Due to this split of attributes, not all previous FDs hold in R_1 and R_2. It is obvious that the FDs in R_1 are exactly those FDs $V \rightarrow W$ for which $V \cup W \subseteq R_1$ and $V \rightarrow W' \in F^+$ with $W \subseteq W'$, because the records for $V \rightarrow W$ are still the same in R_1; R_1 just lost some attributes that are irrelevant for all $V \rightarrow W$. The same observation holds for R_2 although the number of records has been reduced:

LEMMA 3. *The relation $R_2(X \cup Y)$ produced by a decomposition on FD $X \rightarrow Y$ retains exactly all FDs $V \rightarrow W$, for which $V \cup W \subseteq R_2$ and $V \rightarrow W$ is valid in R.*

PROOF. (1) Any valid $V \rightarrow W$ of R is still valid in R_2: Assume that $V \rightarrow W$ is valid in R but invalid in R_2. Then R_2 must contain at least two records violating $V \rightarrow W$. Because the decomposition only *removes* records in $V \cup W$ and $V \cup W \subseteq R_2 \subseteq R$, these violating records must also exist in R. But such records cannot exist in R, because $V \rightarrow W$ is valid in R; hence, the FD must also be valid in R_2.

(2) No valid $V \rightarrow W$ of R_2 can be invalid in R: Assume $V \rightarrow W$ is valid in R_2 but invalid in R. Then R must contain at least two records violating $V \rightarrow W$. Because these two records are not completely equal in their $V \cup W$ values and $V \cup W \subseteq R_2$, the decomposition does not remove them and they also exist in R_2. So $V \rightarrow W$ must also be invalid in R_2. Therefore, there can be no FD valid in R_2 but invalid in R. □

Assume that, instead of BCNF, we would aim to assure 3NF, which is slightly less strict than BCNF: In contrast to BCNF, 3NF does not remove all FD-related redundancy, but it is dependency preserving. Consequently, no decomposition may split an FD other than the violating FD [4]. To calculate 3NF instead of BCNF, we could additionally remove all those groups of violating FDs from the result of Algorithm 4 that are mutually exclusive, i.e., any FD that would split the LHS of some other FD. To calculate stricter normal forms than BCNF, we would need to have detected other kinds of dependencies. For example, constructing 4NF requires all multi-valued dependencies (MVDs) and, hence, an algorithm that discovers MVDs. The normalization algorithm, then, would work in the same manner.

7. CONSTRAINT SELECTION

During schema normalization, we need to define key and foreign key constraints. Syntactically, all keys are equally correct and all violating FDs form correct foreign keys, but semantically the choice of primary keys and violating FDs makes a difference. Judging the relevance of keys and FDs from a semantic point of view is a difficult task for an algorithm – and in many cases for humans as well – but in the following, we define some quality features that serve to automatically score keys and FDs for being "good" constraints, i.e., constraints that are not only valid on the given instance but are true for its schema.

The two selection components of NORMALIZE use these features to score the key and foreign-key candidates, respectively. Then, they sort the candidates by their score. The most reasonable candidates are presented at the top of the list and likely accidental candidates appear at the end. By default, NORMALIZE uses the top-ranked candidate and proceeds; if a user is involved, she can choose the constraint or stop the process. The candidate list can, of course, become too large for a full manual inspection, but (1) the user always needs to pick only one element, i.e., she does not need to classify all elements in the list as either true or false, (2) the candidate list becomes shorter in every step of the algorithm as many options are implicitly weeded out, and (3) the problem of finding a split candidate in a ranked enumeration of options is easier than finding a split without any ordering, as it would be the case without our method.

7.1 Primary key selection

If a relation has no primary key, we must assign one from the relation's set of keys. To find the semantically best key, NORMALIZE scores all keys X using the following features:

(1) Length score: $\frac{1}{|X|}$
Semantically correct keys are usually shorter than random keys (in their number of attributes $|X|$), because schema designers tend to use short keys: Short keys can more efficiently be indexed and they are easier to understand.

(2) Value score: $\frac{1}{max(1,|max(X)|-7)}$
The values in primary keys are typically short, because they serve to identify records and usually do not contain much business logic. Most relational database management systems (RDBMS) also restrict the maximum length of values in primary key attributes, because primary keys are indexed by default and indices with too long values are more difficult to manage. So we downgrade keys with values longer

than 8 characters using the function *max(X)* that returns the longest value in attribute (combination) X; for multiple attributes, *max(X)* concatenates their values.

(3) Position score: $\frac{1}{2}(\frac{1}{|left(X)|+1} + \frac{1}{|between(X)|+1})$
When considering the order of attributes in their relations, key attributes are typically located left and without non-key attributes between them. This is intuitive, because humans tend to place keys first and logically coherent attributes together. The position score exploits this by assigning decreasing score values to keys depending on the number of non-key attributes left *left(X)* and between *between(X)* key attributes X.

The formulas we propose for the ranking reflect only our intuition. The list of features is most likely also not complete, but the proposed features produce good results for key scoring in our experiments. For the final *key score*, we simply calculate the mean of the individual scores. The perfect key with one attribute, a maximum value length of 8 characters and position one in the relation, then, has a key score of 1; less perfect keys have lower scores.

After scoring, NORMALIZE ranks the keys by their score and lets the user choose a primary key amongst the top ranked keys; if no user interaction is desired (or possible), the algorithm automatically selects the top-ranked key.

7.2 Violating FD selection

During normalization, we need to select some violating FDs for the schema decompositions. Because the selected FDs become foreign key constraints after the decompositions, the violating FD selection problem is similar to the foreign key selection problem [20], which scores inclusion dependencies (INDs) for being good foreign keys. The viewpoints are, however, different: Selecting foreign keys from INDs aims to identify semantically correct links between existing tables; selecting foreign keys from FDs, on the other hand, is about forming redundancy-free tables with appropriate keys.

Recall that selecting semantically correct violating FDs is crucial, because some decompositions are mutually exclusive. If possible, a user should also discard violating FDs that hold only accidentally in the given relational instance. Otherwise, NORMALIZE might drive the normalization a bit too far by splitting attribute sets – in particular sparsely populated attributes – into separate relations.

In the following, we discuss our features for scoring violating FDs $X \rightarrow Y$ as good foreign key constraints:

(1) Length score: $\frac{1}{2}(\frac{1}{|X|} + \frac{1}{|Y|\cdot(|R|-2)})$
Because the LHS X of a violating FD becomes a primary key for the LHS attributes after decomposition, it should be short in length. The RHS Y, on the contrary, should be long so that we create large new relations: Large right-hand sides not only raise the confidence of the FD to be semantically correct, they also make the decomposition more effective. Because the RHS can be at most $|R| - 2$ attributes long in relation R (one attribute must be X and one must not depend on X so that X is not a key in R), we weight the RHS's length by this factor.

(2) Value score: $\frac{1}{max(1,|max(X)|-7)}$
The value score for a violating FD is the same as the value score for a primary key X, because X becomes a primary key after decomposition.

(3) Position score: $\frac{1}{2}(\frac{1}{|between(X)|+1} + \frac{1}{|between(Y)|+1})$
The attributes of a semantically correct FD are most likely placed close to one another due to their common context. We expect this to hold for both the FD's LHS and RHS. The space between LHS and RHS attributes, however, is only a very weak indicator, and we ignore it. For this reason, we weight the violating FD anti-proportionally to the number of attributes *between* LHS attributes and *between* RHS attributes.

(4) Duplication score: $\frac{1}{2}(2 - \frac{|uniques(X)|}{|values(X)|} - \frac{|uniques(Y)|}{|values(Y)|})$
A violating FD is well suited for normalization if both LHS X and RHS Y contain possibly many duplicate values and, hence, much redundancy. The decomposition can, then, remove many of these redundant values. As for most scoring features, a high duplication score in the LHS values reduces the probability that the FD holds by coincidence, because only duplicate values in an FD's LHS can invalidate the FD and having many duplicate values in LHS X without any violation is a good indicator for its semantic correctness. For scoring, we estimate the number of unique values in X and Y with $|uniques()|$; because exactly calculating this number is computationally expensive, we create a Bloom-filter for each attribute and use their false positive probabilities to efficiently estimate the number of unique values.

We calculate the final *violating FD score* as the mean of the individual scores. In this way, the most promising violating FD is one that has a single LHS attribute determining almost the entire relation with short and few distinct values. Like for the key scoring, the proposed features reflect our intuitions and observations; they might not be optimal or complete, but they produce reasonable results for a difficult selection problem: In our experiments the top-ranked violating FDs usually indicate the semantically best decomposition points.

After choosing a violating FD for becoming a foreign key constraint, we could in principle decide to remove indovidual attributes from the FD's RHS. One reason might be that these attributes also appear in another FD's RHS and can be used in a subsequent decomposition. So when a user guides the normalization process, we present all RHS attributes that are also contained in other violating FDs. He/she can then decide to remove such attributes. If no user is present, nothing is removed.

8. EVALUATION

In this section, we evaluate the efficiency and effectiveness of our normalization algorithm NORMALIZE. At first, we introduce our experimental setup. Then, we evaluate the performance of NORMALIZE and in particular its closure calculation component. In the end, we assess the quality of the normalization output.

8.1 Experimental setup

NORMALIZE has been implemented using the *Metanome* data profiling framework (www.metanome.de), which defines standard interfaces for different kinds of profiling algorithms [17]. In particular, Metanome provided the implementation of the HYFD FD discovery algorithm. Common tasks, such as input parsing, result formatting, and performance measurement, are standardized by the framework and decoupled from the algorithm itself.

Table 3: The datasets, their characteristics, and their processing times

Name	Size	Attr.	Records	FDs	FD-Keys	FD Disc.	Closure$_{impr}$	Closure$_{opt}$	Key Der.	Viol. Iden.
Horse	25.5 kB	27	368	128,727	40	4,157 ms	1,765 ms	486 ms	40 ms	246 ms
Plista	588.8 kB	63	1000	178,152	1	9,847 ms	6,652 ms	857 ms	49 ms	55 ms
Amalgam1	61.6 kB	87	50	450,020	2,737	3,462 ms	745 ms	333 ms	7 ms	25 ms
Flight	582.2 kB	109	1000	982,631	25,260	20,921 ms	132,085 ms	1,662 ms	77 ms	93 ms
MusicBrainz	1.2 GB	106	1,000,000	12,358,548	0	2,132 min	215.5 min	1.4 min	331 ms	26 ms
TPC-H	6.7 GB	52	6,001,215	13,262,106	347,805	3,651 min	3.8 min	0.5 min	163 ms	4093 ms

Hardware. We ran all our experiments on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB DDR3 RAM. The server runs on CentOS 6.7 and uses OpenJDK 64-Bit 1.8.0_71 as Java environment.

Datasets. We primarily use the synthetic *TPC-H*[4] dataset (scale factor one), which models generic business data, and the *MusicBrainz*[5] dataset, which is a user-maintained encyclopedia on music and artists. To evaluate the effectiveness of Normalize, we denormalized the two datasets by joining all their relations into a single, universal relation. In this way, we can compare the normalization result to the original datasets. For MusicBrainz, we had to restrict this join to eleven selected core tables, because the number of tables in this dataset is huge. We also limited the number of records for the denormalized MusicBrainz dataset, because the associative tables produce an enormous amount of records when used for complete joins.

For the efficiency evaluation, we use four additional datasets, namely Horse, Plista, Amalgam1, and Flight. We provide these datasets and more detailed descriptions on our web-page[6]. In our evaluation, each dataset consists of one relation with the characteristics shown in Table 3; the input of Normalize can, in general, consist of multiple relations.

8.2 Efficiency analysis

Table 3 lists six datasets with different properties. The amount of minimal functional dependencies in these datasets is between 128 thousand and 13 million, and thus too great to manually select meaningful ones. The column *FD-Keys* counts all those keys that we can directly derive from the FDs. Their number does not depend on the number of FDs but on the structure of the data: Amalgam1 and TPC-H have a snow-flake schema while, for instance, MusicBrainz has a more complex link structure in its schema.

We executed Normalize on each of these datasets and measured the execution time for the components (1) FD Discovery, (2) Closure Calculation, (3) Key Derivation, and (4) Violating FD Identification. The first two components are parallelized so that they fully use all 32 cores of our evaluation machine. The necessary discovery of the complete FD set still requires 36 and 61 hours on the two larger datasets, respectively.

First of all, we notice that the key derivation and violating FD identification steps are much faster than the FD discovery and closure calculation steps; they usually finish in less than a second. This is important, because the two components are executed multiple times in the normalization process and a user might be in the loop interacting with the system at the same time. In Table 3, we show only the execution times for the first call of these components; subsequent calls can be handled even faster, because their input sizes shrink continuously. The time needed to determine the violating FDs depends primarily on the number of FD-keys, because the search for Lhs generalizations in the trie of keys is the most expensive operation. This explains the long execution time of 4 seconds for the TPC-H dataset.

For the closure calculation, Table 3 shows the execution times of the improved (*impr*) and optimized (*opt*) algorithm. The naive algorithm already took 13 seconds for the Amalgam1 dataset (compared to less than 1 s for both *impr* and *opt*), 23 minutes for Horse (<2 s and <1 s for *impr* and *opt*, respectively), and 41 minutes for Plista (<7 s and <1 s). These runtimes are so much worse than the improved and optimized algorithm versions that we stopped testing it. The optimized closure algorithm, then, outperforms the improved version by factors of 2 (Amalgam1) to 159 (MusicBrainz), because it can exploit the completeness of the given FD set. The more extensions of right-hand sides the algorithm must perform, the higher this advantage becomes. The average Rhs size for Amalgam1 FDs, for instance, increases from 32 to 56, whereas the average Rhs size for MusicBrainz FDs increases from 3 to 40. For TPC-H, the average Rhs size increases from 10 to 23. The runtimes of the optimized closure calculation are, overall, acceptable when compared to the FD discovery time. Therefore, it is not necessary to filter FDs prior to the closure calculation.

Because closure calculation is not only important for normalization but for many other use cases as well, Figure 2 analyses the scalability of this step in more detail. The graphs show the execution times of the improved and the optimized algorithm for an increasing number of input FDs. The experiment takes these input FDs randomly from the 12 million MusicBrainz FDs; the number of attributes is kept constant to 106. We again omit the naive algorithm, because it is orders of magnitude slower than both other approaches.

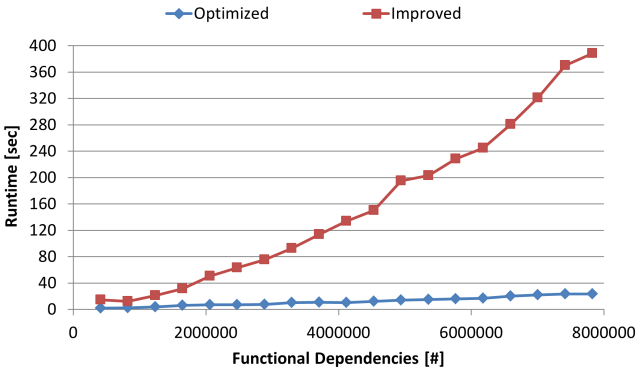

Figure 2: Scaling the number of input FDs for closure calculation.

[4]http://tpc.org/tpch
[5]https://musicbrainz.org
[6]https://hpi.de/naumann/projects/repeatability

Figure 3: Relations after normalizing TPC-H.

Both runtimes in Figure 2 appear to scale almost linearly with the number of FDs, because the extension costs for each single FD are low due to the efficient index lookups. Nevertheless, the index lookups become more expensive with an increasing number of FDs in the indexes (and they would also become more numerous, if we would increase the number of attributes as well). Because the improved algorithm performs the index lookups more often than the optimized version (i.e. changed loop) and with larger search keys (i.e. Lhs and Rhs), the optimized version is faster and scales better with the number of FDs: It is from 4 to 16 times faster in this experiment.

8.3 Effectiveness analysis

For a fair effectiveness analysis, we perform the normalization automatically, i.e., without human interaction. Under human supervision, better (but possibly also worse) schemata than presented below can be produced. For the following experiments, we focus on TPC-H and MusicBrainz, because we denormalized these datasets before so that we can use their original schemata as gold standards for their normalization results.

Figure 3 shows the BCNF normalized TPC-H dataset. The color coding indicates the original relations of the different attributes. So we first notice that Normalize almost perfectly restored the original schema: We can identify all original relations in the normalized result. The automatically selected constraints, i.e., keys and foreign keys are all correct w.r.t. the original schema, which is possible because the original schema was snow-flake shaped.

Nevertheless, we also observe two interesting flaws in the automatically normalized schema: First, Normalize decomposed the LINEITEM relation a bit too far; syntactically, the result is correct and perfectly BCNF-conform, but semantically, the splits with only one dependent and more than three foreign key attributes are not reasonable. Second, the attribute shippriority originally belongs to the ORDERS relation but was placed into the REGION relation. This is syntactically a good decision, because the region also determines the shipping priority and putting the attribute into this relation removes more redundant values than putting it into the ORDERS relation.

Figure 4 shows the BCNF-normalized MusicBrainz dataset. Although MusicBrainz has originally no snow-flake schema, Normalize was still able to reconstruct almost all original relations. Only ARTIST_CREDIT_NAME was not reconstructed and its attributes now lie in the semantically

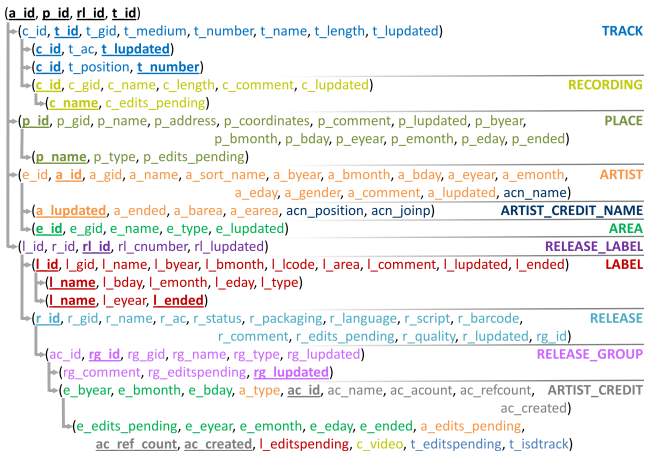

Figure 4: Relations after normalizing MusicBrainz.

related ARTIST relation. Because MusicBrainz is originally not snow-flake shaped, the normalization produced a new top-level relation that represents all many-to-many relationships between artists, places, release labels, and tracks. This top-level relation can be likened to a fact table.

Most mistakes are made for the ARTIST_CREDIT relation, which was the first proposed split. This split took away some attributes from other relations, because these attributes do not contain many values and assigning them to the ARTIST_CREDIT relation makes syntactically sense. A human expert, if involved, would have likely avoided that, because Normalize does report to the user that these attributes are also dependent on other violating FDs Lhs attributes. Overall, however, the normalization result is quite satisfactory, keeping in mind that no human was involved in creating it.

We also tested Normalize on various other datasets with similar findings: If datasets have been de-normalized before, we can find the original tables in the proposed schema; if sparsely populated columns exist, these are often moved into smaller relations; and if no human is in the loop, some decompositions become detailed. All results were BCNF-conform and semantically understandable.

9. CONCLUSION

We proposed Normalize, an instance-driven, (semi-) automatic algorithm for schema normalization. The algorithm has shown that functional dependency profiling results of any size can efficiently be used for the specific task of schema normalization. We also presented techniques for guiding the BCNF decomposition algorithm in order to produce semantically good normalization results that also conform to changes of the data.

Our implementation is publicly available at http://hpi.de/naumann/projects/repeatability. It is currently console-based, offering only basic user interaction. Future work shall concentrate on emphasizing the user-in-the-loop, for instance, by employing graphical previews of normalized relations and their connections. We also suggest research on other features for the key and foreign key selection that may yield even better results. Another open research question is how normalization processes should handle dynamic data and errors in the data.

10. REFERENCES

[1] Z. Abedjan, P. Schulze, and F. Naumann. DFD: Efficient functional dependency discovery. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 949–958, 2014.

[2] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 731–742, 2004.

[3] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)*, 4(1):30–59, 1979.

[4] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems (TODS)*, 1(4):277–298, 1976.

[5] S. Ceri and G. Gottlob. Normalization of relations and prolog. *Communications of the ACM*, 29(6):524–544, 1986.

[6] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[7] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[8] C. J. Date. *Database Design & Relational Theory*. O'Reilly Media, 2012.

[9] J. Diederich and J. Milton. New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems (TODS)*, 13(3):339–365, 1988.

[10] R. Fagin. Normal forms and relational database operators. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 153–160, 1979.

[11] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[12] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[13] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment*, 7(4):301–312, 2013.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[15] H. Köhler, S. Link, and X. Zhou. Possible and certain SQL key. *Proceedings of the VLDB Endowment*, 8(11):1118–1129, 2015.

[16] H. Mannila and K.-J. Räihä. Dependency inference. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 155–158, 1987.

[17] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome. *Proceedings of the VLDB Endowment*, 8(12):1860–1871, 2015.

[18] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[19] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.

[20] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*, 2009.