Synthesizing Linked Data Under Cardinality and Integrity Constraints

Amir Gilad*
Duke University
agilad@cs.duke.edu

Shweta Patwa*
Duke University
sjpatwa@cs.duke.edu

Ashwin Machanavajjhala
Duke University
ashwin@cs.duke.edu

ABSTRACT

The generation of synthetic data is useful in multiple aspects, from testing applications to benchmarking to privacy preservation. Generating the *links* between relations, subject to *cardinality constraints* (CCs) and *integrity constraints* (ICs) is an important aspect of this problem. Given instances of two relations, where one has a foreign key dependence on the other and is missing its foreign key (*FK*) values, and two types of constraints: (1) CCs that apply to the join view and (2) ICs that apply to the table with missing *FK* values, our goal is to impute the missing *FK* values such that the constraints are satisfied. We provide a novel framework for the problem based on declarative CCs and ICs. We further show that the problem is NP-hard and propose a novel two-phase solution that guarantees the satisfaction of the ICs. Phase I yields an intermediate solution accounting for the CCs alone, and relies on a hybrid approach based on CC types. For one type, the problem is modeled as an Integer Linear Program. For the others, we describe an efficient and accurate solution. We then combine the two solutions. Phase II augments this solution by incorporating the ICs and uses a coloring of the conflict hypergraph to infer the values of the *FK* column. Our extensive experimental study shows that our solution scales well when the data and number of constraints increases. We further show that our solution maintains low error rates for the CCs.

ACM Reference Format:
Amir Gilad, Shweta Patwa, and Ashwin Machanavajjhala. 2021. Synthesizing Linked Data Under Cardinality and Integrity Constraints. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457242

1 INTRODUCTION

In recent years, we have witnessed an increase in data-centric applications that call for efficient testing over reliable databases with certain desired qualities [11, 32]. Existing benchmarks such as TPC-H [41, 52] may not possess the desired characteristics for testing a specific application as they may not have the needed statistical qualities or the correct Integrity Constraints (ICs). The field of *data generation* [5, 9, 19, 22, 25, 35, 42, 44] has proven effective in this respect. Two prominent challenges in this field are: (1) the generation of links between different tables, i.e., aligning foreign keys with primary keys based on Cardinality Constraints (CCs) [5], and (2) ensuring that the data will satisfy a set of expected ICs [48].

In particular, when the real data is sensitive and access to it is heavily regulated, users often need to wait months or years to get access to the real data before they can even start writing data analysis programs. One solution is to generate realistic synthetic data that satisfies some CCs and ICs so that users can: (a) start writing code to analyse the data, (b) test it locally, and (c) evaluate whether access to the data would be useful for their purposes even before they get access to the real data. However, current methods for generating synthetic data under privacy constraints (especially state-of-the-art standards like differential privacy [17]) do not handle data with a combination of CCs or statistical constraints and ICs. Most, (e.g., [24, 47, 56]), only handle statistical constraints.

Furthermore, there has been a lot of recent work on answering count queries under differential privacy (e.g., Matrix mechanism [30], HDMM [36]) and in particular over relational databases [28]. A key challenge when answering queries especially over relational databases is that of *consistency* – are the answers outputted by a differentially private algorithm consistent with some underlying database? While there is work on using inference to enforce consistency when all the count queries are over a single view of the underlying database [23], these techniques do not extend to the case when: (a) the underlying database is relational and query answers are over several joined views of the relations, and (b) when the underlying database needs to satisfy some ICs. One solution to this problem is to find a database that is consistent with the query answers and the ICs, and answer queries from it. While techniques for finding such a consistent database are known for single tables without ICs [7, 23, 29], no such techniques are known when there are multiple tables in a relational database with ICs.

Moreover, DBMS testing and other applications may require databases that conform to both CCs and ICs to make them more realistic [5, 48]. For instance, consider a table with the attributes A and B. A query grouping over attributes A and B could return as many tuples as the cross product of the active domains of A and B. However, if there is a Functional Dependency $A \rightarrow B$, then the output size of the group-by query is only the maximum of the active domains of the two attributes. Thus, the presence of ICs can significantly impact the performance characteristics of queries.

In this paper, we investigate the problem of generating the links between database tables based on a set of linear CCs and a set of ICs.

Formally, we consider two relations, R_1 and R_2, where R_1 has a foreign key dependence on R_2 and is missing all values in its foreign key column *FK*. The goal is to impute *FK* in R_1 based on the given CCs and ICs. Importantly, this problem and our solutions can be

*Both authors contributed equally to this research.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457242

Persons (rel. R_1)

p_{id}	*Age*	*Rel*	*Multi-ling*	h_{id}
1	75	*Owner*	0	?
2	75	*Owner*	1	?
3	25	*Owner*	0	?
4	25	*Owner*	1	?
5	24	*Spouse*	0	?
6	10	*Child*	1	?
7	10	*Child*	1	?
8	30	*Owner*	0	?
9	30	*Owner*	1	?

Housing (rel. R_2)

h_{id}	*Area*
1	*Chicago*
2	*Chicago*
3	*Chicago*
4	*Chicago*
5	*NYC*
6	*NYC*

Figure 1: Database $\mathcal{D}$ with FK h_{id} missing from R_1

$$DC_{O,O}: \forall t_1, t_2.\ \neg(t_1.Rel = t_2.Rel = Owner \wedge t_1.h_{id} = t_2.h_{id})$$
$$DC_{O,S,low}: \forall t_1, t_2.\ \neg(t_1.Rel = Owner \wedge t_2.Rel = Spouse \wedge t_2.Age < t_1.Age - 50 \wedge t_1.h_{id} = t_2.h_{id})$$
$$DC_{O,S,up}: \forall t_1, t_2.\ \neg(t_1.Rel = Owner \wedge t_2.Rel = Spouse \wedge t_2.Age > t_1.Age + 50 \wedge t_1.h_{id} = t_2.h_{id})$$
$$DC_{O,C,low}: \forall t_1, t_2.\ \neg(t_1.Rel = Owner \wedge t_1.Multi\text{-}ling = 1 \wedge t_2.Rel = Child \wedge t_2.Age < t_1.Age - 50 \wedge t_1.h_{id} = t_2.h_{id})$$
$$DC_{O,C,up}: \forall t_1, t_2.\ \neg(t_1.Rel = Owner \wedge t_1.Multi\text{-}ling = 1 \wedge t_2.Rel = Child \wedge t_2.Age > t_1.Age - 12 \wedge t_1.h_{id} = t_2.h_{id})$$

(a) Denial Constraints: $DC_{O,O}$ enforces that no two homeowners can reside in the same home, $DC_{O,S,low}$ and $DC_{O,S,up}$ together specify the permissible age range of a spouse in any home, and $DC_{O,C,low}$ and $DC_{O,C,up}$ give the age range for a child living with a multi-lingual homeowner

$$CC_1: |\sigma_{Rel=Owner,Area=Chicago}(R_1 \bowtie R_2)| = 4$$
$$CC_2: |\sigma_{Rel=Owner,Area=NYC}(R_1 \bowtie R_2)| = 2$$
$$CC_3: |\sigma_{Age\leq 24,Area=Chicago}(R_1 \bowtie R_2)| = 3$$
$$CC_4: |\sigma_{Multi\text{-}ling=1,Area=Chicago}(R_1 \bowtie R_2)| = 4$$

(b) Cardinality Constraints: CC_1 and CC_2 give the number of homeowners in Chicago and NYC, resp., CC_3 gives the number of people younger than 25 who live in Chicago, and CC_4 gives the number of multi-lingual individuals in Chicago.

Figure 2: Set of DCs and set of CCs

extended to relational databases with a snowflake schema [13], by focusing on pairs of relations linked by foreign key joins.

Example 1.1. *Consider the relations in Figure 1 based on the Census database. R_1 describes people through attributes such as age, relationship to a household (e.g. owner or spouse), whether they speak more than 1 language and a (missing) household id, whereas R_2 shows the area for each household. In addition, we are given the set of ICs and CCs in Figures 2a and 2b, respectively. The goal is to impute values in the h_{id} column in R_1 so that the ICs and CCs are satisfied.*

We believe that the problem we focus on is a key building block for the general problem of synthesizing data consistent with CCs and ICs for all three use-cases mentioned above. In particular, we believe that one can use the wealth of existing literature to synthesize individual relations consistent with CCs without the key relationships and then use our technique to fill-in the foreign keys.

Our Contributions

We model the problem, give a theoretical analysis, and provide a solution for the generation of foreign keys for existing database relations while ensuring the satisfaction of a set of ICs and reducing the error of a set of CCs. Next, we give our main contributions.

Model and Theoretical Results: We define the problem of C-Extension whose input is a relation R_1 with an unknown foreign key dependence on a relation R_2, i.e., the *FK* column in R_1 is missing, and a set of CCs and ICs. For the CCs, we define and use linear CCs that apply to $R_1 \bowtie R_2$, based on [5]. For the ICs, we define a type of Denial Constraints (DCs) [14, 16], called Foreign Key DCs, that applies to R_1 and forbids tuples from having the same *FK* value under specified conditions. We then show that C-Extension is NP-hard in data complexity. This result leads us to a two-phase heuristic solution that still ensures the satisfaction of all DCs, while tolerating possible errors in the CC counts.

Solution: Our solution can be split into two phases: (1) first phase (Section 4) is designed for the completion of a view V_{Join} based on CCs, where V_{Join} represents $R_1 \bowtie R_2$ and is initialized with a copy of R_1 (without the *FK* column) along with an empty column per non-key column in R_2 (due to foreign key dependence, $|R_1| = |V_{Join}|$), and (2) second phase (Section 5) uses the generated view V_{Join} to complete the *FK* column in R_1 so that the DCs are satisfied.

Phase I: We provide a novel description of CC relationships that allows for V_{Join} to be completed efficiently and precisely under specific conditions (presented in Section 3.1). We further devise algorithms for this case and the general case:

- For the general case, we devise an algorithm that models the CCs and the tuples in V_{Join} as an Integer Linear Program (inspired by [5]). From its solution, we greedily infer the values in V_{Join} for the attributes that come from R_2.
- For the special case, we devise a novel algorithm based on relationships between the CCs. We show that if the CCs have containment or disjointness relationships between them (defined in Section 4.2), then we can find an exact completion of V_{Join} without any errors, provided one exists.

Our approach is a hybrid of these two solutions that employs the first solution for the subset of CCs that does not fit the special case, and employs the second solution for the subset of CCs that does.

Another novelty in our solution exploits the fact that the all-way marginals for R_1, i.e., counts of tuples with different combinations of values in R_1's non-key columns, have the same counts in V_{Join}. Thus, we augment the input set of CCs to improve accuracy.

Phase II: For the second phase, we employ the concept of a *conflict hypergraph* [16] and use a novel algorithm based on hypergraph coloring. We model the tuples in R_1 as vertices and connect by an edge every set of tuples that will violate a DC if assigned the same foreign key. Thus, colors represent the values that the foreign keys can take in R_1, and a proper coloring represents a mapping of tuples to foreign keys that does not violate any DC. Due to the previous stage that considered $R_1 \bowtie R_2$, tuples in R_1 have a certain list of permitted colors. This version of the graph coloring problem is called *List Coloring* [2] and is known to be NP-hard. To color the graph, we use a greedy coloring algorithm that considers vertices in descending order by degrees. The algorithm skips vertices whose list of permitted colors is subsumed by the colors assigned to their neighbors. We ensure a proper coloring by adding the least number of new colors for the skipped vertices. Adding colors beyond the permitted lists corresponds to artificially adding tuples in R_2.

Experimental Evaluation We have implemented our solution and performed a comprehensive set of experiments on a dataset derived from the 2010 U.S. Decennial Census [45]. We have evaluated

our solution in terms of accuracy and scalability in various scenarios, several of which were used for comparison with a baseline based on [5]. We further examined the runtime breakdown of our approach, presenting the runtimes of phases I and II in our solution. Our results indicate that our solution incurs relatively small error for CCs and no error for DCs (as guaranteed by our theoretical analysis). Moreover, our algorithms scale well for large data sizes, and large and complex sets of CCs and DCs. For increasing data scales, our approach was 17 times faster on average across different cases than the baseline we compare to.

2 PRELIMINARIES AND MODEL

We now define the basic concepts used throughout the paper, and the C-Extension problem.

Relations in a Database: Let R_1 and R_2 be relations over the schema attributes $(K_1, A_1, \ldots, A_p, FK)$ and $(K_2, B_1, \ldots, B_q)$, respectively. An attribute A_j of R_i may also be called a column and is denoted by $R_i.A_j$. $t \in R_i$ denotes a tuple in R_i and $t.A_j$ denotes the *cell* of column A_j in tuple t. The last column in R_1 (FK) is a foreign key column that gets its values from the key column K_2 in R_2. The view $V_{Join} = R_1 \bowtie_{FK=K_2} R_2$ denotes the join of the two relations. If all values of a column A_i are missing, it is called a *missing column*.

Example 2.1. *Consider a database $\mathcal{D}$ with two relations R_1 and R_2 as shown in Figure 1. $R_1.h_{id}$ is a missing column. The first row in R_1 says that $t_1.Age$ is 75, $t_1.Rel$ is Owner and $t_1.Multi\text{-}ling$ is 0.*

Foreign Key Denial Constraints: DCs [14] are a general form of constraints that can be written as a negated First Order Logic statement. DCs can express several types of integrity constraints like functional dependencies and conditional functional dependencies [10]. In this paper, we restrict our attention to DCs that contain a condition of the form $t_1.FK = \ldots = t_k.FK$.

Definition 2.2 (Foreign Key DC). *A Foreign Key DC on a relation $R(K_1, A_1, \ldots, A_p, FK)$ is defined as the following FOL statement:*

$$\forall t_1, t_2, \ldots, t_k. \quad \neg(p_1 \wedge \ldots \wedge p_n)$$

where $p_q = t_i.A_l \circ t_j.A_l$ or $p_q = t_i.A_l \circ c$, for $t_i, t_j \in R$, $p \geq 2$, $\circ \in \{=, <, >, \neq\}$, c and k are constants, and $p_n = (t_1.FK = \ldots = t_k.FK)$.

We use the terms Foreign Key DC and DC interchangeably.

Example 2.3. *$DC_{O,O}$ (Figure 2a), which states that two homeowners cannot be in the same home, can be formulated as follows:*

$$\forall t_1, t_2 \in R_1. \ \neg(t_1.Rel = t_2.Rel = Owner \wedge t_1.h_{id} = t_2.h_{id})$$

Note that the restriction to Foreign Key DCs means that all constraints are on people that are in the same household.

Linear Cardinality Constraints: CCs form the second class of constraints that allows for the specification of the number of tuples that should posses a certain set of attribute values, which can be expressed as a selection condition. As standard in previous work [5, 37], we restrict our attention to *linear* CCs.

Definition 2.4 (Linear CC, adapted from [5]). *A linear CC over a database $\mathcal{D}$ consisting of relations $R_1(K_1, A_1, \ldots, A_p, FK)$ and $R_2(K_2, B_1, \ldots, B_q)$ is defined as follows:*

$$|\sigma_\varphi(R_1 \bowtie_{FK=K_2} R_2)| = k$$

where φ is a Boolean selection predicate over a subset of (non-key) attributes in $\mathcal{D}$, and $k \in \mathbb{N}$.

In the rest of the paper, we only refer to conjunctive selection predicates with conjuncts of the form $A_i \circ c$, where $\circ \in \{=, <, >, \leq, \geq\}$ and c is in the domain of column A_i, though our algorithms can be extended to conditions that contain disjunction as well.

Example 2.5. *CC_1 (Figure 2b), which states that the number of homeowners (Rel = Owner) living in Area = Chicago must equal 4, can be written as: $|\sigma_{Rel=Owner,Area=Chicago}\ R_1 \bowtie R_2| = 4$.*

We denote by $R \vDash \sigma$ the fact that relation R meets constraint σ.

Problem Definition: We now formally define the C-Extension problem and discuss its intractability.

Definition 2.6 (C-Extension). *Let $R_1(K_1, A_1, \ldots, A_p, FK)$ and $R_2(K_2, B_1, \ldots, B_q)$ be two relations, where $R_1.FK$ is a foreign key mapped from $R_2.K_2$ and is empty . Let S_{DC} denote the set of DCs over R_1 and let S_{CC} denote the set of linear CCs over the foreign key join between R_1 and R_2. C-Extension is the problem of completing all the values in $R_1.FK$ to create $\hat{R_1}$ so that (1) $\forall \sigma \in S_{DC}$, $\hat{R_1} \vDash \sigma$, (2) $\forall \sigma \in S_{CC}$, $\hat{R_1} \bowtie_{FK=K_2} R_2 \vDash \sigma$.*

Example 2.7. *Reconsider relations R_1 and R_2 in Figure 1, and DCs and CCs in Figure 2. A solution $\hat{R_1}$ for the C-Extension problem as defined by these relations and constraints is shown in Figure 3.*

Persons (rel. R_1)

p_{id}	Age	Rel	$Multi\text{-}ling$	h_{id}
1	75	$Owner$	0	2
2	75	$Owner$	1	1
3	25	$Owner$	0	3
4	25	$Owner$	1	4
5	24	$Spouse$	0	2
6	10	$Child$	1	2
7	10	$Child$	1	2
8	30	$Owner$	0	5
9	30	$Owner$	1	6

Figure 3: Relation R_1 from Figure 1 with FK h_{id} filled-in to satisfy DCs and CCs given in Figure 2

The decision version of C-Extension is given by the same setting as in Definition 2.6. The output is 1 if there exists a completion of $R_1.FK$ such that all DCs and CCs are satisfied, and 0 otherwise.

Proposition 2.8. *The decision problem version of C-Extension is NP-hard in data complexity.*

Proof Sketch. We describe a reduction from NAE-3SAT to C-Extension. In the NAE-3SAT problem, we are given a 3-CNF formula φ and asked whether there is a satisfying assignment to φ with every clause having at least one literal with the value False. Given a 3-CNF formula $\varphi = C_1 \wedge \ldots \wedge C_n$, where $x_1, \ldots, x_m$ are the propositional variables in φ, construct a relation $R_1(Var, \alpha, Cls, Chosen)$, where $Chosen$ is missing all values, and Var, α, Cls columns take values:

(1) $(x_i, 1, C_j, ?)$ if making x_i *True* makes C_j *True*
(2) $(x_i, 0, C_j, ?)$ if making x_i *False* makes C_j *True*

We define S_{DC} to be the set with the following two DCs:

(1) $\forall t_1, t_2. \neg(t_1.Var{=}t_2.Var \wedge t_1.\alpha \neq t_2.\alpha \wedge t_1.Chosen{=}t_2.Chosen)$
(2) $\forall t_1, t_2, t_3. \neg(t_1.Cls{=}t_2.Cls{=}t_3.Cls \wedge t_1.Chosen{=}t_2.Chosen{=}t_3.Chosen)$

CCs are not needed in the reduction. The goal is to complete the missing column $Chosen$ in R_1. We define R_2 as containing two columns: a primary key column $Chosen$, and another column E. R_2 contains the tuples $(0, a)$ and $(1, b)$, i.e., the domain for $Chosen$ is

$\{0, 1\}$. Intuitively, *Chosen* encodes the satisfying assignment for φ by assigning values to each tuple, where $t.Chosen$=1 iff the assignment should be $t.Var$=$t.\alpha$. □

The full proofs are detailed in the full version [21].

3 SOLUTION OVERVIEW

Our solution proceeds in two phases as seen in Figure 4. In phase I, we consider the view V_{Join} representing the join of the two relations R_1 and R_2, where R_1 has a foreign key dependence on R_2, and initialize it with (non FK) columns from R_1 and an empty column per non-key column from R_2. We infer these values based on the CCs by a hybrid approach that uses both ILP [5] and a more efficient and accurate procedure for special cases. In phase II, we impute $R_1.FK$ by modeling the problem as a conflict hypergraph using the DCs, and coloring it based on the inferred values in V_{Join}.

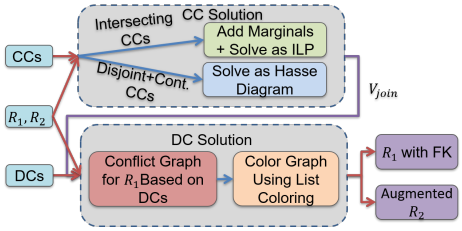


Figure 4: Solution Overview

3.1 Overview of the First Phase

Due to the foreign key dependence (Definition 2.6), we define V_{Join} over the columns $K_1, A_1, \ldots, A_p, B_1, \ldots, B_q$ such that $t \in R_1$ implies that there is a single $t' \in V_{Join}$ with $t.K_1$=$t'.K_1$ and $\forall 1 \leq i \leq p$. $t.A_i = t'.A_i$ with additional $B_1, \ldots, B_q$ entries that are initially all empty because FK is missing in R_1. Therefore, $|V_{Join}| = |R_1|$. Our goal is to complete these columns based on the CCs.

Example 3.1. *Reconsider R_1 and R_2 shown in Figure 1 and the CCs in Figure 2b. The join view V_{Join} is R_1 as it appears in Figure 1 (without h_{id}) with an empty Area column (as this is the schema of $R_1 \bowtie R_2$). Due to the foreign key dependency, we have $|V_{Join}| = |R_1|$, and V_{Join} contains a tuple for each R_1 tuple with the same values as in R_1 and an empty Area value. The reason is that the FK values are missing in R_1. Our goal is to fill-in V_{Join} so that the CCs are satisfied.*

We give a short description of our solution for completing V_{Join}.

Solution as an ILP (Section 4.1, green box in Figure 4): Given a set of CCs on V_{Join}, we model the problem of completing the missing columns as a system of linear equations with variables accounting for counts of different tuples needed in V_{Join} to satisfy the CCs. Thus, the variables must take non-negative integer values. We artificially add to S_{CC} all-way marginals (using the idea of *intervalization* from [5] that is explained in Section 4) from R_1 to enhance the accuracy of the solution. For example, based on CCs given in Example 1.1, $|\sigma_{Age\leq 24, Rel=Spouse, Multi\text{-}ling=0}| = 1$ gets added to S_{CC}. We then assign $B_1, \ldots, B_q$ values to the tuples in V_{Join} based on the solution returned by an ILP solver.

Using CC Relationships for Special Cases (Section 4.2, blue box in Figure 4): We give a novel description of the relationships between CCs based on their selection conditions, defining CC containment, disjointness and intersection. In the case where there are no intersecting CCs and no disjunctions, we give an algorithm to complete V_{Join} that models the containment and disjointness of CCs as a *Hasse diagram* [53] that it recurses on bottom-up to fill-in V_{Join}. Any leftover V_{Join} tuples without $B_1, \ldots, B_q$ values are randomly assigned a combination that cannot cause a new contribution towards the target count of any CC. However, if no such combinations are available, then the leftover V_{Join} tuples cannot be completed. We refer to these as invalid tuples.

Hybrid Approach (Section 4.3): In the absence of intersecting CCs, the solution decomposes cleanly as seen above. This motivates the hybrid approach that combines ideas from both cases to achieve better runtime and accuracy when some CCs intersect. We start by labeling each pair of CCs as disjoint, contained or intersecting. For all CCs that do not intersect or contain any intersecting CCs, we use the approach from Section 4.2, and for the rest, we use the ILP approach from Section 4.1. Lastly, as seen above in the special case, we may end up with some invalid tuples.

3.2 Overview of the Second Phase

After filling-in the columns of V_{Join} that originate in R_2 ($B_1 \ldots, B_q$), we turn to reverse-engineering R_1 from V_{Join}. This phase uses *conflict hypergraphs* [16] to represent possible DC violations.

Conflict Hypergraph (Section 5.1, red box in Figure 4): We use the notion of conflict hypergraph for the tuples of R_1 based on the DCs. Given a DC, we construct an edge for all the sets of tuples that cannot get the same foreign key value due to that DC.

Example 3.2. *Consider the relation R_1 depicted in Figure 1 and the first DC in Figure 2a. Suppose the first two tuples are assigned the same Area value in V_{Join}. Thus, the conflict hypergraph will have an edge containing the tuples with $p_{id} = 1$ and $p_{id} = 2$ since they are both owners and cannot be in the same household (the h_{id} value). The conflict hypergraph of our running example is depicted in Figure 7.*

List Coloring (Section 5.1, orange box in Figure 4): Proper coloring of the hypergraph ensures that there must be at least two vertices in each edge with distinct colors. Thus, modeling each FK value as a color and each tuple as a vertex allows us to prove that a proper coloring results in an assignment of FK values that satisfies the DCs. The values in V_{Join} filled-in by the previous phase induce a list of possible FK values, and thus colors, for R_1 tuples. Finding a proper coloring such that each vertex assumes a color from its predefined list is called List Coloring [2] and is NP-hard. We thus propose a greedy coloring algorithm based on vertex degree.

Algorithm for Satisfying the DCs (Section 5.2): The size of the conflict hypergraph can be very large and thus may cause a significant slowdown in practice. Therefore, we partition R_1 into smaller sets with the same $B_1 \ldots, B_q$ values and construct a conflict hypergraph for each set separately. For each non-invalid tuple, V_{Join} contains $B_1 \ldots, B_q$ values, so we can use our greedy coloring algorithm to find a coloring for them. We color invalid tuples at the end using all FK values as candidates. This phase may result in the addition of extra tuples to R_2 (the second output in Figure 4).

4 FIRST PHASE: SOLVING CCS

In this section, we focus on the first phase. Given two relations $R_1(K_1, A_1, \ldots, A_p, FK)$ and $R_2(K_2, B_1, \ldots, B_q)$, we wish to satisfy a set S_{CC} of CCs over the join view $V_{Join} = R_1 \bowtie_{FK=K_2} R_2$.

4.1 Solution as an ILP

We give a two-part solution in Algorithm 1 where we: (1) model the CCs as a system of linear equations and solve it using an ILP solver, and (2) greedily fill-in $B_1, \ldots, B_q$ values for each tuple in V_{Join}. The first part (lines 3–15) is inspired by [5]. Each variable represents the number of tuples with a specific combination of $A_1, \ldots, A_p, B_1, \ldots, B_q$ values in V_{Join}. Each CC is written as a sum of the variables whose associated tuples satisfy its selection condition. We now introduce the notion of *intervalization* [5].

Intervalization: Creating a variable for every combination of values in the cross product of the full domains of all the $p+q$ (non-key) columns in V_{Join} would give a very large ILP. We augment the notion of intervalization [5] so that it will not only assist in reducing the number of variables based on the intervals of values in S_{CC}, but also use only the combinations of $A_1, \ldots, A_p$ values already in R_1. We call this *binning* the distinct $(A_1, \ldots, A_p)$ values in R_1.

In the system of equations $Ax = b$, row r_i (in A) corresponds to CC_i and row b_i (in b) stores CC_i's target count. We create the vector x of variables by putting bins with the same $B_1, \ldots, B_q$ values as contiguous elements (see Example 4.1). Since input CCs are linear, each element in A is 0 or 1. The goal is to solve for an x with non-negative integer entries (line 15). Such a solution can be obtained if there exists a solution to C-Extension where $R_1 \bowtie_{FK=K_2} R_2$ satisfies S_{CC}. In the second part (lines 17–19), we fill-in the $B_1, \ldots, B_q$ values greedily. For each assignment $x_i = v_i$, we find at most v_i tuples (with empty $B_1, \ldots, B_q$ cells) in V_{Join} that satisfy x_i's selection condition on R_1, and fill-in their $B_1, \ldots, B_q$ values as encoded by x_i.

Example 4.1. *Reconsider relations R_1 and R_2 in Figure 1, CCs in Figure 2b and V_{Join} described in Example 3.1. Intervalization splits Age into $[0, 24]$ and $[25, 114]$ due to CC_3 (all other columns are categorical). Even though R_1 contains multiple tuples for multi-lingual homeowners with age greater than 24, it suffices to look at those with Age in $[0, 24]$ and $[25, 114]$. Importantly, for the given instance, we only need to keep track of the following tuple types: (1) Age $\in [25, 114]$, Rel = Owner, Multi-ling = 0, (2) Age $\in [0, 24]$, Rel = Spouse, Multi-ling = 0, (3) Age $\in [0, 24]$, Rel = Child, Multi-ling = 1, and (4) Age $\in [25, 114]$, Rel = Owner, Multi-ling = 1. Here, vector x uses a copy of these four bins with Area = Chicago in x_1 to x_4 and Area = NYC in x_5 to x_8. Without the idea of binning, we would need 16 variables because Area can take 2 distinct values and R_1 contains 8 unique tuples. Finally, we iterate through each $CC_i \in S_{CC}$ and add rows r_i and b_i in A and b, resp. For CC_1, $r_i = [1, 0, 0, 1, 0, 0, 0, 0]$ and $b_i = 4$ because only x_1 and x_4 match the selection conditions in CC_1; similarly for other CCs. Hence, $Ax = b$ has a solution given by $x_1 = 2, x_2 = 1, x_3 = 2, x_4 = 2, x_5 = 1, x_6 = 0, x_7 = 0$ and $x_8 = 1$. Finally, we iterate through x_i's to find V_{Join} tuples which satisfy its selection condition and assign the matching Area value that gives the view in Figure 5. E.g., we find two tuples in V_{Join} with Age $\in [25, 114]$, Rel = Owner and Multi-ling = 0 for x_1 and assign Area = Chicago.*

Augmenting with All-Way Marginals: When A is sparse, some x_i values in the solution may not match the true counts. Despite such discrepancies, we can complete several tuples in V_{Join} because we update at most as many tuples as the value of x_i in the solution. The order of updates may also impact which subset of V_{Join} tuples gets specific $B_1, \ldots, B_q$ values. For example, another solution to the ILP in Example 4.1 is given by $x_1 = 0, x_2 = 3, x_3 = 0, x_4 = 4, x_5 = x_6 = x_7 = 0, x_8 = 2$. This assigns Area = Chicago to tuples with $p_{id} = 2, 4, 5, 9$ in V_{Join}. However, the remaining tuples do not get any Area value and no CC in S_{CC} gets satisfied in V_{Join}. We overcome this issue by using both S_{CC} and all all-way marginals over $A_1, \ldots, A_p$ from R_1 when solving the ILP (see the discussion about the baseline's CC accuracy in Section 6). The solution reported in Example 4.1 was computed with all all-way marginals.

p_{id}	Age	Rel	Multi-ling	Area
1	75	Owner	0	Chicago
2	75	Owner	1	Chicago
3	25	Owner	0	Chicago
4	25	Owner	1	Chicago
5	24	Spouse	0	Chicago
6	10	Child	1	Chicago
7	10	Child	1	Chicago
8	30	Owner	0	NYC
9	30	Owner	1	NYC

Figure 5: Join view $V_{Join} = (R_1 \bowtie_{FK=K_2} R_2)$ of R_1 and R_2 from Figure 1 with filled-in *Area* values

Algorithm 1: Complete V_{Join} - Intersecting CCs

Input : Relations $R_1(K_1, A_1, \ldots, A_p)$ and $R_2(K_2, B_1, \ldots, B_q)$, S_{CC} - set of linear CCs with target counts

Output: V_{Join} - $B_1, \ldots, B_q$ values filled-in

1 /* model CCs as integer program and solve */
2 View $V_{Join}(K_1, A_1, \ldots, A_p, B_1, \ldots, B_q) \leftarrow$ copy of R_1 with empty $B_1, \ldots, B_q$ columns;
3 $n_{V_{Join}} \leftarrow$ number of bins in which distinct tuples of R_1 are grouped using binning;
4 $\forall i \in [q],\ n_i \leftarrow$ number of distinct B_i values in R_2;
5 $n \leftarrow n_{V_{Join}} \times \prod_{i=1}^{q} n_i$;
6 /*A will be a $(n_{V_{Join}} + |S_{CC}|) \times n$ matrix for CCs*/
7 $b \leftarrow$ empty $(n_{V_{Join}} + |S_{CC}|) \times 1$ vector for target counts;
8 $x \leftarrow n \times 1$ vector for non-negative integer variables;
9 **for** *each tuple type t_i accounted for by $n_{V_{Join}}$* **do**
10 Add row in A /* 0's except 1 for relevant variables in x*/;
11 $b[i] \leftarrow$ number of copies of t_i in R_1;
12 **for** *each $CC_i \in S_{CC}$* **do**
13 Add row in A /* 0's except 1 for relevant variables in x*/;
14 $b[i] \leftarrow CC_i.target$;
15 Compute x by solving $Ax = b$;
16 /* fill values in $B_1, \ldots, B_q$ greedily */
17 **for** *each $x_i \in x$ with value c_i* **do**
18 Find (at most) c_i tuples satisfying x_i's condition in V_{Join};
19 Update $B_1, \ldots, B_q$ values encoded by x_i;
20 **return** V_{Join}

Complexity: The complexity of Algorithm 1 is $O(|S'_{CC}| \cdot m + S)$, where S'_{CC} contains CCs from S_{CC} along with the marginals, and m is the number of variables that is upper-bounded by the number of tuples in R_1 times the product of the sizes of the active domains of $B_1, \ldots, B_q$ in R_2. Lastly, S is the time complexity of the ILP solver.

4.2 Efficient Algorithm for Special CC Types

In practice, Algorithm 1 may incur slow runtimes as generating and solving the system of equations is time consuming, even with state-of-the-art ILP solvers (as shown in Section 6). Thus, we describe a model for relationships between the CCs in S_{CC} and devise an algorithm to better tackle V_{Join} completion in specific scenarios.

$$CC_1 : |\sigma_{Age\in[10,14],Area=Chicago}(R_1 \bowtie R_2)| = 20$$
$$CC_2 : |\sigma_{Age\in[50,60],Multi\text{-}ling=0,Area=NYC}(R_1 \bowtie R_2)| = 25$$
$$CC_3 : |\sigma_{Age\in[13,64],Area=Chicago} R_1 \bowtie R_2| = 100$$
$$CC_4 : |\sigma_{Age\in[18,24],Multi\text{-}ling=0,Area=Chicago} R_1 \bowtie R_2| = 16$$

Figure 6: CC relationships. $CC_1 \cap CC_2 = \emptyset$, **and** $CC_4 \subseteq CC_3$

Definition 4.2. *$CC_i, CC_j \in S_{CC}$ are disjoint either if their selection conditions on the R_1 attributes are disjoint, or if their selection conditions on R_1 are identical and the conditions on R_2 are disjoint. We denote this by $CC_i \cap CC_j = \emptyset$.*

Note that we also consider pairs of CCs with the same R_1, but disjoint R_2 selection conditions as disjoint. For a pair (CC_i, CC_j) of such CCs, assigning $B_1, \ldots, B_q$ values in tuples that contribute to the count of CC_i should not limit the set of tuples available for CC_j, if a solution exists. We label such pairs similarly to a pair of disjoint CCs. Next, we define the notion of CC containment.

Definition 4.3. *Let $CC_i, CC_j \in S_{CC}$ such that $CC_i : |\sigma_{\varphi_i}(R)| = k_i$ and $CC_j : |\sigma_{\varphi_j}(R)| = k_j$. CC_i is contained in CC_j, denoted $CC_i \subseteq CC_j$, if φ_i uses a (non-strict) superset of attributes in φ_j and for each common attribute, the values in CC_i are a subset of the corresponding values in CC_j.*

Intuitively, if CC_i is contained in CC_j, then CC_i is more restrictive than CC_j, and assigning a tuple $t \in R_1$ values in $B_1, \ldots, B_q$ that satisfy the selection condition in CC_i will also satisfy the selection condition in CC_j. This observation defines a partial order on S_{CC} which we utilize later to find a solution for CCs.

Definition 4.4. *$CC_i, CC_j \in S_{CC}$ are said to be intersecting if they are neither disjoint nor does one contain the other. We denote this by $CC_i \cap CC_j \neq \emptyset$.*

Example 4.5. *Assume R_1 (or V_{Join}) contains 10 tuples with Age $\in [10, 30)$, 20 with Age $\in [30, 50)$ and 50 with Age $\in [50, 70]$. Let:*

$$CC_1 : |\sigma_{Age\in[10,50),Area=Chicago} R_1 \bowtie R_2| = 30$$
$$CC_2 : |\sigma_{Age\in[30,70],Area=NYC} R_1 \bowtie R_2| = 30$$

If all tuples with Age $\in [30, 50)$ get assigned Area = NYC, CC_1 cannot be satisfied. Even when Area = Chicago in CC_2, it is unclear how many tuples with age in $[30, 50)$ can be assigned Area = Chicago.

Solution Without Intersecting CCs: Now, we focus on the setting where there are no intersecting CCs present and describe Algorithm 2 that outputs an exact solution.

We use the notion of a Hasse diagram [53], denoted by $\mathcal{H} = (V, E)$, to encode the containment relationships between the CCs in S_{CC}. We refer to each connected component in the undirected version of $\mathcal{H}$ as a diagram. Within each diagram, the CC that is not contained in any other CC is referred to as the *maximal element.*

Algorithm 2 is given the join view V_{Join} with missing $B_1, \ldots, B_q$ columns, S_{CC} and the Hasse diagram $\mathcal{H}$ describing the containment relations in S_{CC}. We denote by $\mathcal{V}(\mathcal{H})$ and $\mathcal{E}(\mathcal{H})$ the collective set of all nodes and edges of the diagrams in $\mathcal{H}$. The algorithm operates recursively with a single base case – if all the CCs in S_{CC} are disjoint, i.e., $\mathcal{E}(\mathcal{H})$ is empty (line 2), then it simply chooses k_i tuples that can contribute to each $CC_i \in S_{CC}$ and completes their $B_1, \ldots, B_q$ values given by CC_i. When the base case is not met, for each $H \in \mathcal{H}$, the algorithm makes a recursive call on each child of the maximal element m in H (lines 9–11) to get the resulting view of the sub-diagram and then finds the remaining number of tuples that will get CC_m to its target count (lines 12–13). Finally, in the loop in line 15, the algorithm completes any missing values in the tuples while ensuring that these values do not add to the count of any $CC \in S_{CC}$ by finding combinations that are not specified in S_{CC}.

Algorithm 2: Complete V_{Join} - Non-intersecting CCs

Input : V_{Join} - View to complete, S_{CC} - Set of CCs, $\mathcal{H}$ - Set of diagrams encoding CC containment
Output : V_{Join} - $B_1, \ldots, B_q$ values filled-in

1 $\forall i \in \mathcal{V}(\mathcal{H})$. $\sigma_i, k_i \leftarrow$ selection condition on R_1, count;
2 **if** $\mathcal{E}(\mathcal{H}) = \emptyset$ **then**
3 **foreach** $i \in \mathcal{V}(\mathcal{H})$ **do**
4 Find k_i tuples in V_{Join} (without $B_1, \ldots, B_q$ values) that satisfy σ_i;
5 Assign $B_1, \ldots, B_q$ values;
6 **return** V_{Join};
7 **foreach** $H \in \mathcal{H}$ **do**
8 $m \leftarrow$ maximal elem. in H;
9 **foreach** $c \in children(m)$ **do**
10 $H_c \leftarrow$ sub-diagram with maximal elem. c;
11 V_{Join} = Algorithm 2$(V_{Join}, S_{CC}, \{H_c\})$;
12 Find $k_m - \sum_{c\in children(m)} k_c$ tuples in V_{Join} that satisfy $\sigma_{\bigwedge_{c\in children(m)} \neg\sigma_c}$;
13 Assign $B_1, \ldots, B_q$ values from CC_m;
14 $combo_{unused} \leftarrow$ list of combinations in R_2 columns that are not relevant to S_{CC};
15 **foreach** $t \in V_{Join}$ **do**
16 **if** $t.B_i, \ldots, t.B_q$ *values are missing* **then**
17 Assign a combination of values from $combo_{unused}$;
18 **return** V_{Join};

Example 4.6. *Reconsider CCs 1–4 in Figure 6. The set $\mathcal{H}$ is $\{H_1, H_2, H_3\}$, where H_1 and H_2 contain only CC_1 and CC_2, respectively, and H_3 is a diagram composed of one edge from CC_3 to CC_4. Algorithm 2 gets $\mathcal{H}$ along with V_{Join} and S_{CC} as input. It assigns CC_i's selection condition on R_1 and target count to σ_i and k_i, for all i. Then, it checks the condition in line 2, which does not hold as we have the edge (CC_3, CC_4). Thus, it goes to the loop in line 7 to iterate over the three diagrams. For H_3, the maximal element is CC_3, so Algorithm 2 recursively calls itself for the sub-diagram containing only CC_4 (line 11) and finds 16 tuples such that Age $\in [18, 24]$ and Multi-ling = 0, and assigns to them Area = Chicago (lines 3–5). It then returns from the recursive call to find $100 - 16 = 84$ tuples with Age $\in$*

$[13, 64] \setminus [18, 24]$ *and Multi-ling* $\neq 0$ *, and assigns to them Area = Chicago (lines 12–13). For* H_1 *(*H_2*) the maximal element is* CC_1 *(*CC_2*), the algorithm then performs a recursive call to itself with an empty diagram, and returns from the call to select* 20 *(*25*) tuples that have Age* $\in [10, 14]$ *(Age* $\in [50, 60]$ *and Multi-ling* $= 0$*) and assign to them Area = Chicago (Area = NYC). Here,* $combo_{unused}$ *contains values from Area's domain except Chicago and NYC which get used in* S_{CC}*. If there are any tuples in* V_{Join} *without an assignment (see loop on line 15), we assign to each a value chosen from* $combo_{unused}$*.*

At the end of the algorithm, any tuple in V_{Join} without $B_1, \ldots, B_q$ values is randomly assigned a combination of values that is not used in S_{CC} (line 17). We refer to these tuples as *invalid tuples* if no such combination is available. Observe that the matching tuples in R_1 do not have an FK to K_2 mapping , i.e., if there is a tuple in V_{Join} that is missing an assignment, also called an invalid tuple, then V_{Join} does not give a set of candidate K_2 values that could be assigned in its FK cell. We will handle such tuples in Section 5.2.

Proposition 4.7. *If* S_{CC} *does not contain intersecting CCs and there exists a join view* V_{Join} *that satisfies all CCs in* S_{CC}*, then Algorithm 2 finds such a view.*

Complexity: The complexity of Algorithm 2 is $O(|S_{CC}|^2 \cdot d_1 + |S_{CC}| \cdot (\max_i |dom_a(B_i)|)^{d_2} + |S_{CC}| \cdot |V_{Join}|)$, where d_1, d_2 are the number of columns in V_{Join} and R_2, $dom_a(B_i)$ is the active domain of $R_2.B_i$. The first term is for computing the relationships between CCs and recursing on the Hasse diagrams (lines 7–11), second term is for constructing $combo_{unused}$ (line 14) and third term is for lines 1–6, 12–13 and choosing a random value per tuple in lines 15–17. In practice, we only consider columns used in S_{CC} instead of d_2.

4.3 Hybrid Approach

In many cases, S_{CC} contains a combination of disjoint, contained, and intersecting CCs, so we combine Algorithms 1 and 2.

We start by constructing Hasse diagram based on containment relationship between pairs of CCs in S_{CC}. Next, we iterate through each diagram $H \in \mathcal{H}$, and discard H if it contains intersecting CCs. Note that the absence of an edge in the Hasse diagram does not guarantee the lack of intersection at the beginning of phase I (demonstrated by Example 4.5 where the Hasse diagram starts out as two nodes without an edge, but the CCs represented by these nodes do intersect). Therefore, we keep track of which CCs intersect to then discard the affected diagrams (set S_2) and run Algorithm 2 on the remaining diagrams (set S_1). In particular, $\forall CC_i \in S_1, CC_j \in S_2$. $CC_i \cap CC_j = \emptyset$, $CC_i \not\subseteq CC_j$ and $CC_j \not\subseteq CC_i$. We then run Algorithm 2 for CCs in S_1, and Algorithm 1 for those in S_2.

As seen above, it is possible that some tuples may not have a $B_1, \ldots, B_q$ assignment in V_{Join}. Let S_3 be the set of these tuples that are dealt with using $combo_{unused}$ as described in Example 4.6. If $|combo_{unused}| = \emptyset$, then all tuples in S_3 are invalid tuples.

Augmenting with Modified Marginals Our approach guarantees that the partial solution returned by Algorithm 2 satisfies S_1 exactly. In comparison to how we augment S_{CC} with marginals in Section 4.1 before solving the ILP, we now want the scope of the marginals being added to be limited to the tuples that are relevant for the CCs in S_2. For example, let $S_{CC} = \{CC_1, CC_3\}$ from Figure 2b. We add CCs with the following selection predicates: (1) *Age* <= 24, *Rel* = *Owner*, *Multi-ling* = 0, and (2) *Age* <= 24, *Rel* = *Owner*, *Multi-ling* = 1. It may still happen that the matrix A is sparse and some x_i's do not match the true counts causing some CC errors.

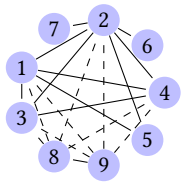


Figure 7: Conflict graph for the tuples in R_1 from Figure 1 based on V_{Join} from Figure 5. Two tuples are connected by a solid edge if their *Area* values match but they cannot be assigned the same h_{id} value. Dashed edges show DC violations between tuples with different *Area* values. Partitioning V_{Join} by *Area* values addresses such violations because the set of household ids is disjoint for different *Area* values

5 SECOND PHASE: ADDING DCS

We start by presenting our model for conflict hypergraph for FK DCs and then use it to describe the solution for DCs. In short, our approach is to reverse-engineer R_1 from V_{Join} so that joining it with R_2 recovers V_{Join}, and R_1 satisfies all DCs in S_{DC}.

5.1 Conflict Hypergraphs and List Coloring

We slightly augment the notion of conflict hypergraphs to illustrate possible Foreign Key DC violations caused by subsets of R_1 tuples.

Definition 5.1 (Conflict Hypergraph for Foreign Key DCs). *A conflict hypergraph for* R_1 *and* S_{DC} *is defined as* $G = (V, E)$ *where* V *is the set of tuples in* R_1 *and* $e = \{t_1, \ldots, t_k\} \in E$ *if there is a Foreign Key DC of the form* $\neg(\varphi(t_1, \ldots, t_k) \wedge t_1.FK = \ldots = t_k.FK)$ *such that* $\varphi(t_1, \ldots, t_k)$ *evaluates to True.*

It suffices to consider only $\varphi(t_1, \ldots, t_k)$ in the DCs when adding edges because FK is initially missing. Abusing notation, we denote a set of tuples $\mathcal{T}$ violating $\varphi_i(t_1, \ldots, t_k)$ in a given DC σ_i by $\mathcal{T} \not\models_{\varphi_i} \sigma_i$ (we will use this notation in Algorithm 4).

Next, we give the connection between conflict hypergraph coloring and FK assignment in R_1, so a proper coloring satisfies DCs.

Proposition 5.2. *Given an instance of C-Extension, a coloring of the conflict hypergraph gives an assignment to all cells of the missing FK column in* R_1 *such that all DCs are satisfied.*

We now turn to the problem of inferring FK values in R_1 from the completed V_{Join}. Each tuple in R_1 can have multiple options for foreign key values that lead to V_{Join} obtained in phase I. This establishes a list of possible colors, also referred to as *candidate colors*, for each vertex in the conflict graph. This problem is called List Coloring [2, 26]. It is a generalization of k-coloring, and is thus NP-hard. Hence, we use a heuristic approach, described by Algorithm 3, to color the vertices in a non-increasing order by degree, coloring as many vertices as possible. In Section 5.2, we describe how to color the vertices that remain uncolored by Algorithm 3.

Algorithm 3 takes as input the conflict hypergraph G_c, a mapping c from vertices to colors (initially empty) and a list of candidate colors L. It can be called on a graph with a partial color assignment (used in Algorithm 4 in Section 5.2). Initially, s is an empty list

that is used to store skipped vertices, and l is the list of uncolored vertices sorted in non-increasing order by degree in G_c (lines 2–3). In lines 4–12, we find a list of permissible colors for each $v \in l$, i.e., those vertices in G_c that have not been given a color in the input color map c. If vertex v belongs to an edge e where all vertices other than v in e have the same color c, then c is a forbidden color for v. Next, the algorithm assigns the "smallest" available color to v in line 10. Otherwise, v gets added to s and remains uncolored (line 12). Finally, color map c and list of skipped vertices s are returned.

Algorithm 3: Largest-first list coloring

Input : G_c - conflict hypergraph with color choices per vertex, c - a map from vertices to colors so far, L - list of candidate colors

Output: c - updated coloring that builds on the input coloring, s - list of skipped vertices

```
1  Function ColoringLF(G_c, c, L):
2      s ← ∅;
3      l ← sortDescendingDeg({v ∈ V[G_c] | v ∉ c});
4      for v ∈ l do
5          forbidden ← ∅;
6          for e s.t. v ∈ e do
7              if ∃c ∀u ≠ v ∈ e. c[u] = c then
8                  forbidden.add(c);
9          if L \ forbidden ≠ ∅ then
10             c[v] ← min(L \ forbidden);
11         else
12             s ← s ∪ {v};
13     return c, s
```

Example 5.3. *Reconsider the view V_{Join} and DCs shown in Figure 5 and 2a, respectively. Figure 7 (including the dashed edges) gives the resulting conflict graph G_c. For example, there is an edge between vertices 1 and 2 because $t_1.Rel = t2.Rel = Owner$, so assigning them the same FK value would violate $DC_{O,O}$. Here, $l = [2, 1, 3, 4, 8, 9, 5, 6, 7]$. Thus, Algorithm 3 returns: $c[1] = 2, c[2] = 1, c[3] = 3, c[4] = 4, c[5] = 3, c[6] = 2, c[7] = 2, c[8] = 5$ and $c[9] = 6$.*

Complexity: The complexity of Algorithm 3 is $O(|V| \cdot \log |V| + |V| \cdot |E|)$ since, the algorithm sorts all vertices by degree (line 3) and then traverses all edges adjacent to each vertex.

5.2 Algorithm for DCs

We describe Algorithm 4 as the last step in solving C-Extension by completing $R_1.FK$. In Section 4, we showed how to complete V_{Join} by assigning values in the columns that came from R_2. For a tuple $t \in V_{Join}$ with values $t.B_i = b_i, 1 \le i \le q$, the candidate FK values for the corresponding tuple in R_1 are given by $\pi_{K_2}\sigma_{B_1=b_1,\ldots,B_q=b_q}R_2$. We begin with an optimization that we employ in the algorithm.

Optimization: Working with a single conflict hypergraph when R_1 contains a large number of tuples would not scale since the hypergraph can form one clique in the worst-case. However, observe that we can partition the filled-in V_{Join} and R_2 by $B_1, \ldots, B_q$ values into sets, and only consider conflict hypergraphs within each set because the candidate FK values are disjoint across sets.

Example 5.4. *Reconsider relations R_1 and R_2, and view V_{Join} shown in Figures 1 and 5, respectively. In V_{Join}, the tuples have $Area = Chicago$ or $Area = NYC$. Note that the set of candidate keys for tuples with $Area = Chicago$ comprises of values in $\pi_{h_{id}}\sigma_{Area=Chicago}R_2$ that is disjoint from those in $\pi_{h_{id}}\sigma_{Area=NYC}R_2$. This eliminates edges that would have been added to the conflict graph if we were to consider all vertices at once (shown as dashed edges in Figure 7). After partitioning V_{Join} by $B_1, \ldots, B_q$ values and using the DCs in Figure 2a, we get two conflict graphs: (1) with vertices for tuples $t_1, \ldots, t_7$, and (2) with vertices for tuples t_8 and t_9. There is an edge between a* **pair** *of vertices when the corresponding tuples* **would** *violate a DC if assigned the same h_{id} value, and these are shown as solid edges in Figure 7.*

Algorithm 4: Complete $R_1.FK$ column using V_{Join}

Input : View $V_{Join}(K_1, A_1, \ldots, A_p, B_1, \ldots, B_q)$, Relations $R_1(K_1, A_1, \ldots, A_p, FK)$ and $R_2(K_2, B_1, \ldots, B_q)$, S_{DC} - set of DCs on R_1

Output: $\hat{R_1}$ - copy of R_1 with FK column filled-in, $\hat{R_2}$ - updated copy of $R_2(K_2, B_1, \ldots, B_q)$

```
1  c_all ← ∅, R̂1 ← copy of R1, R̂2 ← copy of R2;
2  for v = (b1, ..., bq) ∈ π_{B1,...,Bq} V_Join do
3      P_v = {t ∈ V_Join | ∀1 ≤ i ≤ q. t.B_i = b_i};
4      V ← ∅, E ← ∅, c ← ∅;
5      L = π_{K2} σ_{B1=b1,...,Bq=bq} R̂2;
6      for t_j ∈ P_v do
7          V ← V ∪ {v_j};
8      for T ⊆ P_v s.t. ∃σ ∈ S_DC. T ⊭_φ σ do
9          E ← E ∪ {T};
10     c, s ← ColoringLF(G_c = (V, E), c, L);
11     L_new ← |s| number of new colors;
12     c, s ← ColoringLF(G_c = (V, E), c, L_new);
13     for color c_new in L_new that gets used do
14         Add tuple t_new in R̂2 with t_new.K2 = c_new and
           t_new.B_i = b_i (1 ≤ i ≤ q);
15     c_all ← c_all ∪ c;
16 c_all ← solveInvalidTuples(V_Join, S_DC, S_CC, R̂2) ;
17 ∀t_j ∈ V_Join, t' ∈ R̂1. t_j.K1 = t'.K1 set t'.FK = c_all[v_j];
18 return R̂1, R̂2;
```

Algorithm 4 gets as input the view V_{Join} outputted by the algorithm described in Section 4.3, relations R_1 (with missing FK values) and R_2, and set S_{DC}. It outputs $\hat{R_1}$, i.e., R_1 with values in the FK column and $\hat{R_2}$, i.e., R_2 with possible additional tuples (as described next). The algorithm can be divided into three parts: (1) coloring the tuples that were assigned $B_1, \ldots, B_q$ values in V_{Join}, (2) coloring the invalid tuples, i.e., tuples that were *not* assigned $B_1, \ldots, B_q$ values in V_{Join}, and (3) coloring any skipped tuples (defined in Section 5.1). Algorithm 4 maintains a map from tuples to their list of colors in c and tracks the overall coloring in c_{all}. Eventually, c_{all} has a color for every vertex that is used to complete FK in $\hat{R_1}$ (lines 17, 18).

In lines 2–15, the algorithm iterates over each set of tuples with the same $B_1, \ldots, B_q$ value. Given a vector $v = (b_1, \ldots, b_q)$ of q constants, it iterates over tuples in sets given by $P_v = \{t \in V_{Join} \mid \forall 1 \leq i \leq q.\ t.B_i = b_i\}$. For each P_v, it generates the conflict hypergraph G_c as follows: a node v_j per tuple t_j, a list L of candidate colors given by the keys from tuples in $\hat{R}_2$ with values $v_j.B_1, \ldots, v_j.B_q$, and an edge per set of tuples in P_v that violates φ in some DC. Next, G_c, c and L are inputted to Algorithm 3, which outputs a partial coloring c and a list of skipped vertices s. Vertices in s are colored using at most $|s|$ new colors (lines 11-14), resulting in insertion of tuples in $\hat{R}_2$ because colors correspond to primary keys in $\hat{R}_2$.

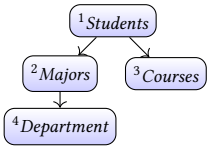
Procedure *solveInvalidTuples* (line 16) handles the invalid tuples (defined in Section 4.2). Since these do not have $B_1, \ldots, B_q$ values in V_{Join}, the corresponding $\hat{R}_1$ tuples are missing FK values because we have not yet considered them in any conflict hypergraphs. We construct a hypergraph for tuples in V_{Join} with edges incident to only the vertices for invalid tuples. However, the set s outputted by Algorithm 3 may contain invalid tuples that had to be skipped for a lack of available colors. Our strategy for coloring these is to assign to each a combination of $B_1, \ldots, B_q$ values that minimizes the error stemming from the CCs (defined in Section 6), and generate a tuple in $\hat{R}_2$ with a fresh key and the chosen $B_1, \ldots, B_q$ values. Finally, $\hat{R}_1.FK$ values are assigned based on the coloring c_{all} (line 17).

Proposition 5.5. *Given V_{Join}, R_1, R_2, S_{DC}, Algorithm 4 outputs relations $\hat{R}_1, \hat{R}_2$ such that $\hat{R}_2$ is a copy of R_2, possibly with more tuples, and $\hat{R}_1$ is a copy of R_1 with all the values in the FK column completed such that $\forall \sigma \in S_{DC},\ \hat{R}_1 \vDash \sigma$, and $\hat{R}_1 \bowtie_{FK=K_2} \hat{R}_2 = V_{Join}$.*

Complexity: The complexity of Algorithm 4 is $O(n \cdot |S_{DC}| \cdot \binom{n}{T})$, where $|R_1| = n$, and T is the number of tuples involved in the largest DC (assumed to be a constant), since the number of edges of each vertex can be at most $\binom{n}{T}$. The n component stands for the possible need to iterate over all tuples in V_{Join} in line 2, the $|S_{DC}|$ component stands for the possible need to iterate over all DCs when checking the condition in line 8, and the $\binom{n}{T}$ component is added due to the need to iterate over all subsets of P_v that may satisfy a DC in lines 8–9. Since Algorithm 3 (lines 10, 12) has a complexity of $O(n \cdot \binom{n}{T})$, and the loop (line 13) has complexity of $O(n)$, they are not presented in the overall complexity of the algorithm. Note that $\biguplus_{j=1}^{m} P_v^j = V_{Join}$, where m is the number of iterations and P_v^j is the set P_v^j generated in iteration j.

Extending the solution to snowflake schemas: Our solution can be generalized to snowflake schemas in a manner similar to [5]. The idea is to start from the fact table (the central table) as R_1 and a table connected to it as R_2, i.e. going from the inside out in a Breadth-First Search manner. In every step, we include the previously completed tables in R_1, allowing CCs that span over the join view of multiple tables. This ensures that tuples are (possibly) added to the relation in the role of R_2 only once, since in the next step it would be considered as R_1 and thus maintain the foreign key dependency from the previous steps.

Example 5.6. *Consider a central Students table with two foreign key dependencies of a Majors table and a Courses table, and another foreign key connection to a Departments table through Majors:*



The steps of the algorithm are as follows:

Step	R_1	R_2
1	$Students$	$Majors$
2	$Students \bowtie Majors$	$Courses$
3	$(Students \bowtie Majors) \bowtie Courses$	$Departments$

At each step we can therefore consider CCs over all tables we have considered so far. For example, in step 2, we can consider CCs over $((Students \bowtie Majors) \bowtie Courses)$ and not just over $Students \bowtie Courses$. Note that in the first step, we might have added artificial tuples to the Majors table. These are added without an FK value that connects them to the Departments table so we account for them in the last step of connecting the Majors and Departments tables, making sure that the DCs that apply to the Majors table are satisfied.

6 EXPERIMENTS

We analyze the performance of our (hybrid) approach, and compare it with a baseline algorithm (based on [5]) in these terms:

(1) Accuracy and runtime comparison between the baseline and our approach as data grows for fixed S_{DC} and two settings of S_{CC} (based on Section 4.2): (i) S_{CC} with no intersecting CCs, and (ii) S_{CC} with intersecting CCs.
(2) Accuracy and runtime comparison between the baseline and our approach for fixed data and combinations of good and bad S_{DC} and S_{CC}. Good S_{DC} creates zero cliques in conflict graphs and good S_{CC} contains zero intersecting CCs.
(3) Runtime performance of our approach when data and S_{DC} are kept fixed but the size of good S_{CC} and bad S_{CC} varies.
(4) Runtime performance of our approach for fixed data, and good S_{DC} and S_{CC} as the number of columns in R_2 grows.

We implemented our solution and baseline in Python 3.6.9 and Pandas DataFrame interface [39] on Tensor TXR231-1000R D126 Intel(R) Xeon(R) CPU E5-2640 v4 2.40GHz CPU with 512 GB (40 cores) of RAM. We use the standard PuLP [38] and NumPy Libraries for the ILP, and NetworkX [6] to construct and color conflict graphs.

A summary of our findings:

(1) Our approach satisfies all CCs in the absence of intersecting CCs with no error. Additionally, our approach satisfies all DCs (as guaranteed by our theoretical analysis), whereas the baseline does not (Figures 8-10). Overall, our approach has the shortest runtime (Figure 11a) and achieves better accuracy for CCs and DCs together. Additionally, augmenting the input set of CCs with marginals over the non-key attributes in R_1 improves accuracy for CCs. We also find that the time spent by the baseline on the ILP solver alone is comparable to the total time taken by our approach for larger data scales.
(2) At a fixed data scale and for good and bad settings of DCs and CCs, where good DCs do not create cliques in conflict graphs and good CCs do not intersect, our approach has the shortest runtime. Its best performance is for good DCs and good CCs. In comparison, using bad DCs is slower because conflict graphs become denser, and using bad CCs is even slower because of the ILP solver.

(3) Keeping the data and S_{DC} fixed, we find that increasing the size of and/or intersections in S_{CC} slows down V_{Join} completion, increasing the runtime of our approach (Figure 13).
(4) We find that the time spent on coloring grows faster than that for recursing on Hasse diagrams as the number of columns in R_2 grows when good S_{DC} and S_{CC} are used.

6.1 Setup

We now describe the experimental setup (summarized in Table 3) and define the error measures that are used to evaluate accuracy. We vary the database size (Table 1), DCs and CCs (see [21]) to examine the scalability and accuracy of our solution. In Section 6.2, the errors and runtimes are averaged over 3 independent runs.

Data. We perform experiments on a dataset that is derived from the 2010 U.S. Decennial Census [45] comprising of two relations $Persons(p_{id}, Rel, Age, Multi\text{-}ling, h_{id})$ and $Housing(h_{id}, Tenure, Area)$, with $Persons$ (R_1) missing all values in its foreign key column h_{id}. The different data scales are given in Table 1. By construction, V_{Join} and $Persons$ contain the same number of tuples. We also consider up to 10 (non-key) columns in $Housing$, where we go from $(Tenure, Area)$ to $(Tenure, County, Area, St)$, add (Div, Reg) and then add binary attributes $(Water, Bath)$ followed by $(Fridge, Stove)$. Note that values in Div and Reg are determined by the St value.

Table 1: Data scales given by the number of tuples

Scale	Persons table	Housing table	V_{Join}
1×	25,099	9,820	25,099
2×	50,039	19,640	50,039
5×	124,746	49,100	124,746
10×	249,259	98,200	249,259
40×	1,015,686	392,800	1,015,686
80×	2,043,975	785,600	2,043,975
120×	3,064,328	1,178,400	3,064,328
160×	4,097,471	1,571,200	4,097,471

Denial Constraints. S_{DC}^{all} is the set of DCs (see [21]) that not only gives the permissible age gap between a homeowner (Rel = Owner) and other members in the same home, but also limits the number of homeowners, spouses and unmarried partners per home. S_{DC}^{good} contains first 8 DCs, none of which create cliques in conflict graphs.

Cardinality Constraints. We use two sets of CCs, S_{CC}^{good} and S_{CC}^{bad}, with 1001 CCs each (see [21]). We assume that each input CC specifies a condition on an attribute from both R_1 and R_2. S_{CC}^{bad} contains CCs with intersecting Age intervals, but S_{CC}^{good} does not.

Error Measures. We measure *relative CC error* err_i as $\frac{|\hat{c_i}-c_i|}{\max(10,c_i)}$, where $\hat{c_i}$ and c_i are CC_i's (in S_{CC}) counts in the solution and input. We use a threshold of 10 in the denominator because some CCs have a target count of 0 for small data scales. We report the median relative CC errors in Figures 8-10, where $y = 1$ represents 100% error. We measure *DC error* as the fraction of tuples in $\hat{R_1}$ that violate a $DC \in S_{DC}$. E.g., if h_{id} value in the first two tuples in $Persons$ relation in Figure 3 was 2, then the DC error would be 2/9.

Baseline. Arasu et al. [5] focuses on the generation of synthetic databases with snowflake schema, where all joins are foreign key joins. This work considers CCs alone (no DCs) and imputes FK using V_{Join}. Motivated by this work, we establish the two baseline versions given below (Section 7 surveys more related works).

(1) **Baseline:** First, we use Algorithm 1 (without the for loop on line 9) to fill-in tuples in V_{Join}. Any V_{Join} tuple without an assignment is completed by randomly assigning values in $B_1, \ldots, B_q$. In phase II, we randomly assign a value from the candidate FK values given by V_{Join} for each tuple in R_1.
(2) **Baseline with marginals:** We also study the impact of augmenting S_{CC} with all Age–Rel–$Multi\text{-}ling$ (all-way) marginals from $Persons$, where domains of numerical attributes are broken using intervalization [5] on S_{CC}. Note that the marginals have equal target counts in $Persons$ and V_{Join} by construction. They ensure that each variable participates in the ILP, and is thus assigned a value in the solution. We find that this fills in all V_{Join} tuples. We refer to this algorithm as *baseline with marginals* that uses Algorithm 1 for phase I, followed by random assignment in FK using V_{Join} for phase II. Hence, it falls in-between the baseline and our approach (Section 4.3).

Table 2: Datasets used in experiments, with details about the data scales, DCs and CCs given in Table 1 and in [21]

Dataset no.	Data Scale	DCs	CCs
1-5	1× to 40×	S_{DC}^{all}	S_{CC}^{good}
6-10	1× to 40×	S_{DC}^{all}	S_{CC}^{bad}
11	10×	S_{DC}^{good}	S_{CC}^{good}
12	10×	S_{DC}^{good}	S_{CC}^{bad}
13 – 17	10×	S_{DC}^{all}	(500, 600, 700, 800, 900) S_{CC}^{good}
18 – 22	10×	S_{DC}^{all}	(500, 600, 700, 800, 900) S_{CC}^{bad}
23 – 26	40× to 160×	S_{DC}^{good}	S_{CC}^{good}
27 – 30	40× to 160×	S_{DC}^{good}	S_{CC}^{bad}
31 – 34	10× (4, 6, 8, 10 non-key R_2 columns)	S_{DC}^{good}	S_{CC}^{good}

Table 3: Experimental settings for Figures 8-13. Table 2 contains details about the input datasets

Experiment	Figure	Algorithm	Input datasets
Accuracy Exp.	8a	Baselines vs Hybrid	1-5
	8b	Baselines vs Hybrid	6-10
	9	Baselines vs Hybrid	10
	10	Baselines vs Hybrid	11, 12, 4, 9
Scalability Exp.	11a	Baselines vs Hybrid	9, 10
	11b	Hybrid	11, 23 – 26, 12, 27 – 30
	12	Hybrid	11, 31 – 34
	13	Hybrid	17, 22

6.2 Experimental Findings

We discuss results and address aspects raised at the start of Section 6.

Our approach vs baselines - Accuracy. We consider the experimental setup from Table 3 for the accuracy experiments, and detail our results in Figures 8-10. Our approach always satisfies all DCs, and all CCs in S_{CC}^{good}. For S_{CC}^{bad} (Table 8b), the median CC error is 0 but the smallest and largest average errors are 0.048 and 0.093 due to limitations in augmenting S_{CC} (Section 4.3). In contrast, the baseline gives median CC and DC errors between 0.233-0.580 and 0.228-0.373, whereas baseline with marginals satisfies all CCs but gives DC errors between 0.402-0.510. We take a closer look at the relative CC errors for data Scale 40× and S_{CC}^{bad} in Figure 9. Note that DCs are used only after V_{Join} is partitioned by $B_1, \ldots, B_q$ values, so CCs affect the quality of the solution given by Algorithm 4.

Next, we look at combinations of good and bad cases of DCs and CCs for data at Scale 10×. Again, our approach satisfies all DCs and gives a median CC error of 0. Here, half the CCs were passed into Algorithm 2 that satisfies CCs exactly. The remaining CCs are augmented (Section 4.3) to improve accuracy for Algorithm 1. We

Scale	CC error			DC error		
	Baseline	Baseline (marginals)	Hybrid	Baseline	Baseline (marginals)	Hybrid
1×	0.300	0	0	0.218	0.445	0
2×	0.367	0	0	0.245	0.465	0
5×	0.526	0	0	0.274	0.446	0
10×	0.604	0	0	0.303	0.489	0
40×	0.559	0	0	0.371	0.520	0

(a) S^{all}_{DC}, S^{good}_{CC}

Scale	CC error			DC error		
	Baseline	Baseline (marginals)	Hybrid	Baseline	Baseline (marginals)	Hybrid
1×	0.233	0	0	0.228	0.435	0
2×	0.300	0	0	0.246	0.434	0
5×	0.467	0	0	0.279	0.402	0
10×	0.537	0	0	0.305	0.510	0
40×	0.580	0	0	0.373	0.489	0

(b) S^{all}_{DC}, S^{bad}_{CC}

Figure 8: Error rate comparison between the baseline, baseline with marginals and hybrid as data grows from Scale 1×–40× and: (a) S^{all}_{DC} (12 DCs) and S^{good}_{CC} (1001 CCs) are used, and (b) S^{all}_{DC} (12 DCs) and S^{bad}_{CC} (1001 CCs) are used

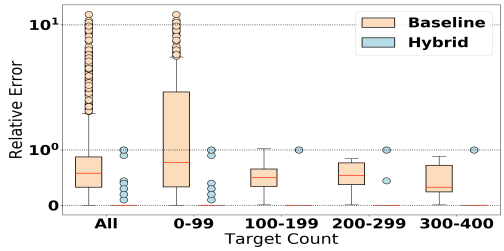
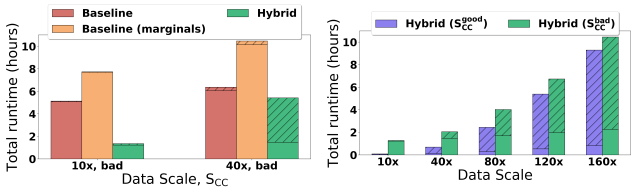


Figure 9: Relative CC error incurred by the baseline and hybrid for data Scale 40×, S^{all}_{DC} (12 DCs) and S^{bad}_{CC} (1001 CCs). We omit baseline with marginals as it satisfies all CCs

Data-set	CC error			DC error		
	Baseline (no aug)	Baseline (with aug)	Hybrid	Baseline (no aug)	Baseline (with aug)	Hybrid
11	0.618	0	0	0.081	0.009	0
12	0.573	0	0	0.079	0.004	0
4	0.604	0	0	0.303	0.489	0
9	0.537	0	0	0.305	0.510	0

Figure 10: CC and DC error comparison between baseline, baseline with marginals and hybrid for combinations of good and bad cases of DCs and CCs at data Scale 10×



(a) S^{all}_{DC} (12 DCs), S^{bad}_{CC} (b) S^{good}_{DC} (8 DCs), S^{good}_{CC} or S^{bad}_{CC}

Figure 11: Shaded area depicts phase II in: (a) Runtime comparison between baseline and hybrid for S^{all}_{DC}, S^{bad}_{CC} (1001 CCs) and data Scale 10× or 40×, (b) Runtime of hybrid for S^{good}_{DC} and data Scales 10×–160× with S^{good}_{CC} or S^{bad}_{CC} (1001 CCs)

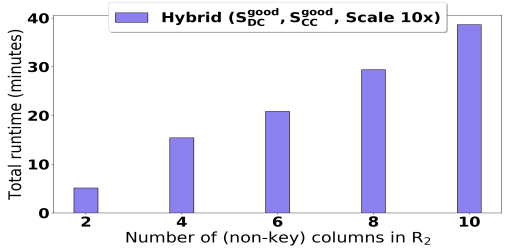


Figure 12: Runtime of hybrid for S^{good}_{DC} (8 DCs), S^{good}_{CC} (1001 CCs) and data Scale 10× as number of columns in R_2 grows

find that most CCs have a relative error of 0. For S^{bad}_{CC}, the average CC error given by our approach is 0.0735. In contrast, baseline gives CC errors between 0.537-0.618 and DC errors between 0.079-0.305, whereas baseline with marginals satisfies all CCs but gives DC errors between 0.004-0.510 due to random assignment in $R_1.FK$.

Our approach vs baselines - Runtime. We consider the experimental setup as given in Table 3 for the scalability experiments. The total runtimes at data Scales 10× and 40× are given in Figure 11a. Observe that the time spent on phase II by the baseline is minimal because it randomly assigns FK values, whereas our approach colors conflict graphs to satisfy all DC exactly. In our approach, Algorithm 1 and 4 are the bottlenecks taking 25.23 % and 72.74 % of the total runtime for S^{all}_{DC} and S^{bad}_{CC} at data Scale 40×.

At data Scale 40×, the runtimes for completing V_{Join} for S^{good}_{CC} and S^{bad}_{CC} are as follows: (1) baseline takes 5.88 hours and 6.07 hours, (2) baseline with marginals takes 9.75 hours and 10.15 hours, and (3) our approach takes 7.79 minutes and 1.48 hours. The corresponding total runtimes are: (1) 6.19 hours and 6.38 hours, (2) 10.04 hours and 10.49 hours, and (3) 1.06 hours and 5.43 hours hours. Our approach has the shortest runtime in completing V_{Join} because we take advantage of the relationships between the CCs to separate out intersecting CCs from S_{CC} which reduces the time to solve the ILP. In contrast, the baseline creates one large ILP with all CCs (with or without all-way marginals). In addition, our approach does not need the ILP solver for S^{good}_{CC}, further improving the runtime.

Note that the baselines do not take into account the DCs and randomly assign FK values based on filled-in V_{Join}, so solving the ILP dominates the time to populate h_{id} in $Persons$ (R_1). Our approach has the shortest runtime for (S^{good}_{DC}, S^{good}_{CC}) (at 5.17 minutes), followed by (S^{all}_{DC}, S^{good}_{CC}), (S^{good}_{DC}, S^{bad}_{CC}) and (S^{all}_{DC}, S^{bad}_{CC}) (at 1.36 hours). Intuitively, for fixed data, completing V_{Join} is faster for S^{good}_{CC} and conflict graphs are more likely to have fewer edges for S^{good}_{DC}. For (S^{good}_{DC}, S^{good}_{CC}) and (S^{all}_{DC}, S^{bad}_{CC}): (1) baseline took 4.84-5.14 hours, and (2) baseline with marginals took close to 8 hours. Baseline ran faster because it runs the ILP solver without the marginals.

Larger data scales - Runtime. We examine our solution's runtime for larger data scales when S^{good}_{DC} is used with S^{good}_{CC} vs S^{bad}_{CC} (see Figure 11b). We find that our solution scales well, taking a total of 9.3 hours for S^{good}_{CC} and 10.46 hours for S^{bad}_{CC} at data Scale 160×.

Increasing the number of R_2 columns - Runtime. We study the effect of increasing the number of R_2 columns on the runtime of our approach for S^{good}_{DC}, S^{good}_{CC} and data Scale 10× (see Figure 12). We describe in Section 6.1 how the number of columns in R_2 grows from

2 to 10. Our approach takes a total of 5.17 minutes for 2 columns and 38.66 minutes for 10 columns because we only consider the columns that are used in S_{CC} for $combo_{unused}$ (Algorithm 2).

Increasing the number of CCs - Runtime and accuracy. Here we study the effect of the size of S_{CC} on the runtime and error in our approach (the last row in Table 3). The breakdown of runtimes for 900 CCs chosen from S_{CC}^{good} and S_{CC}^{bad} is given in Figure 13.

	900 CCs from S_{CC}^{good}		900 CCs from S_{CC}^{bad}	
	Time	%	Time	%
Pairwise Comparison	4.48s	1.12	4.24s	0.10
Recursion	1.70m	25.64	1.29m	1.76
ILP solver	–	–	1.06h	86.21
Coloring	4.87m	73.24	8.77m	11.93

Figure 13: Runtime breakdown of the hybrid approach for data Scale 10× with S_{DC}^{all} (12 DCs) and 900 CCs from S_{CC}^{good} or S_{CC}^{bad} (overall we have 1001 CCs for both)

As more CCs are used, the time spent on labeling pairs of CCs as disjoint, contained or intersecting increases. Since S_{CC}^{good} contains no intersecting CCs, the ILP solver is not used and the runtime is faster. Algorithm 2 takes 1.42 minutes for 500 CCs and 1.78 minutes for 900 CCs. More CCs not only cause more updates to V_{Join}, but may also add CCs where $Area$ is used without $Tenure$, creating tuples with a partial assignment in V_{Join} that are completed in line 17 of the algorithm. For 900 CCs, the total runtime is 6.65 minutes, of which 4.87 minutes are spent in filling-in R_1 (Algorithm 4).

When more CCs are chosen from S_{CC}^{bad}, we see an increase in the number of CCs passed to Algorithm 1 that makes the ILP solver slower. Algorithm 2 takes 1.21 minutes for 500 CCs and 1.36 minutes for 900 CCs, whereas Algorithm 1 takes 25.99 minutes for 500 CCs and 1.06 hours for 900 CCs. For 900 CCs, the total runtime is 1.23 hours, of which 8.77 minutes are spent in completing R_1.

Our approach satisfies all DCs, and CCs in S_{CC}^{good}. The median and average CC error rates are 0 and 0.034-0.092, resp. We give an optimization for (bottleneck) coloring step in the full version [21].

7 RELATED WORK

Data generation has been the focus of multiple works, e.g., [4, 5, 9, 11, 12, 19, 22, 25, 32, 35, 42, 44, 46, 51]. *The main novelty of this paper is the generation of foreign keys for existing database relations while reducing the error of a set of CCs and ensuring the satisfaction of a set of DCs that relate to the foreign key attribute.*

A prominent line of work uses CCs to define the desired parameters of the generated data [5, 9, 44]. QAGen [9] was among the first system that focused on data generation in a *query-aware* fashion. The target application was to test the performance of a database management system (DBMS) when given a database schema, one parametric Conjunctive Query and a collection of constraints on each operator. MyBenchmark [32] extends [9] by generating a *set* of database instances that approximately satisfies the cardinality expectations from a set of query results. HYDRA [44] uses a *declarative approach* that allows for the generation of a database summary that can be used for dynamically generating data for query execution. Arasu et. al. [5] proposed a framework that supports multiple CCs and generates data using a graphical model that converts the CCs to equations, using the concept of intervalization for efficient computations. Indeed, we have drawn on this work for Algorithm 1. These approaches allow for complex CCs, whereas our approach allows for DCs as well. A recent work [48] has proposed a solution for generating *multiple data samples* using a seed sample of the data (generated by previous work [49]), statistical constraints and data validity constraints specified in OCL [1]. UpSizeR [51] has focused on scaling the database while maintaining foreign key constraints. Data generation from the database schema and statistical information has also been studied [42, 46].

The field of data privacy [17, 28, 30, 34, 36, 50, 54] typically gives mechanisms that generate query answers that do not expose features of the underlying private data, rather than generate the data itself. Some works [23, 56] focus on providing consistent query answers, but none, to our knowledge, consider queries over linked data that guarantee the satisfaction of a set of ICs. Yahalom et. al. [55] developed a framework for converting production data into test data by modeling it as a constrained satisfaction problem (CSP) using specific constraints that can be expressed as part of the CSP.

Finally, DCs (without CCs) have been mainly explored in relation with data cleaning [3, 8, 14, 16, 18, 20, 27, 43]. Previous work on the subject has focused on two main approaches: (1) repairing attribute values in cells [8, 16, 43] and (2) tuple deletion [14, 20, 33]. In this context, there has been previous work on automatically discovering DCs from the complete data [15, 31, 40]. We consider DCs based on the FK column, which is missing. In many scenarios, as is the premise in many data cleaning works (e.g., [14, 16, 20, 43]), such DCs can be naturally inferred from the schema or from domain knowledge. As in data cleaning, the constraints can be formulated by the users as logical statements [43] or as SQL queries [20].

8 CONCLUSIONS AND LIMITATIONS

We have defined the problem of generating links between database relations using linear CCs and foreign key DCs, and proved that it is intractable. Therefore, we have shown a novel two-phase heuristic solution. Our solution first considers the CCs, with a hybrid approach that combines an ILP-based solution and a solution based on specific relationships between the CCs. Second, our approach utilizes a version of conflict graph coloring in order to find a completion of the tuples that satisfies all DCs. Our experimental results show that our solution is both accurate and scalable.

There are many intriguing directions for future work. First, our solution focuses on linear CCs and a subset of DCs. Finding a solution when the constraints include non-linear CCs (e.g., CCs on the number of rows that share the same foreign key) and general DCs (e.g., DCs on tuples that do not share a foreign key) is an important extension of our approach. Second, in phase I, we assume foreign key dependence that induces a one-to-one mapping between the tuples of R_1 and the tuples of V_{Join}. Examining other join dependencies that do not have this property is an interesting direction of exploration. Third, in phase II, tuples may be artificially added to R_2 due to the coloring algorithm. Some scenarios may not allow such augmentation and thus require different solutions. Finally, the extension of our solution to non-relational databases, such as graph databases and wide-column store is another subject of future study.

Acknowledgements. This work was supported by the National Science Foundation under grants 1408982 and 1703431; and by DARPA and SPAWAR under contract N66001-15-C-4067.

REFERENCES

[1] 2017. Object Constraint Language 2.4 Specification. https://www.omg.org/spec/OCL/About-OCL/.

[2] Dimitris Achlioptas and Michael Molloy. 1997. The Analysis of a List-Coloring Algorithm on a Random Graph (extended abstract. *IEEE* (1997). https://users.soe.ucsc.edu/~optas/papers/list-coloring.pdf

[3] Foto N. Afrati and Phokion G. Kolaitis. 2009. Repair Checking in Inconsistent Databases: Algorithms and Complexity. In *ICDT*. 31–41.

[4] Shaukat Ali, Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. 2013. Generating Test Data from OCL Constraints with Search Techniques. *IEEE Trans. Software Eng.* 39, 10 (2013), 1376–1402.

[5] Arvind Arasu, Raghav Kaushik, and Jian Li. 2011. Data Generation Using Declarative Constraints. In *SIGMOD*. 685–696.

[6] Daniel A. Schult Aric A. Hagberg and Pieter J. Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. SciPy.

[7] Boaz Barak, Kamalika Chaudhuri, Cynthia Dwork, Satyen Kale, Frank McSherry, and Kunal Talwar. 2007. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *SIGACT-SIGMOD-SIGART*. 273–282.

[8] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. 2013. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Theory Comput. Syst.* 52, 3 (2013), 441–482.

[9] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: generating query-aware test databases. In *SIGMOD*. 341–352.

[10] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional functional dependencies for data cleaning. In *ICDE*. 746–755.

[11] Nicolas Bruno and Surajit Chaudhuri. 2005. Flexible Database Generators. In *Proc. VLDB Endow.* 1097–1107.

[12] Teodora Sandra Buda, Thomas Cerqueus, John Murphy, and Morten Kristiansen. 2013. VFDS: Very fast database sampling system. In *IRI*. 153–160.

[13] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.* 26, 1 (March 1997), 65–74.

[14] Jan Chomicki and Jerzy Marcinkowski. 2005. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.* 197, 1-2 (2005), 90–121.

[15] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *PVLDB* 6, 13 (2013), 1498–1509.

[16] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. 458–469.

[17] Cynthia Dwork. 2006. Differential Privacy. In *ICALP*, Vol. 4052. 1–12.

[18] Ronald Fagin, Benny Kimelfeld, and Phokion G. Kolaitis. 2015. Dichotomies in the Complexity of Preferred Repairs. In *PODS*. 3–15.

[19] Bálint Fazekas and Attila Kiss. 2018. Statistical Data Generation Using Sample Data. In *New Trends in Databases and Information Systems*, Vol. 909. 29–36.

[20] Amir Gilad, Daniel Deutch, and Sudeepa Roy. 2020. On Multiple Semantics for Declarative Database Repairs. In *SIGMOD*. 817–831.

[21] Amir Gilad, Shweta Patwa, and Ashwin Machanavajjhala. 2021. Synthesizing Linked Data Under Cardinality and Integrity Constraints. https://arxiv.org/abs/2103.14435. *CoRR* abs/2103.14435 (2021).

[22] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD*. 243–252.

[23] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. 2010. Boosting the Accuracy of Differentially Private Histograms Through Consistency. *Proc. VLDB Endow.* 3, 1 (2010), 1021–1032.

[24] Xi He, Ashwin Machanavajjhala, and Bolin Ding. 2014. Blowfish privacy: tuning privacy-utility trade-offs using policies. In *SIGMOD*. ACM, 1447–1458.

[25] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and Realistic Data Generation. In *Proc. VLDB Endow.* 1243–1246.

[26] Tommy R Jensen and Bjarne Toft. 2011. *Graph coloring problems*. Vol. 39. John Wiley & Sons.

[27] Solmaz Kolahi and Laks V. S. Lakshmanan. 2009. On approximating optimum repairs for functional dependency violations. In *ICDT*. 53–62.

[28] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: A Differentially Private SQL Query Engine. *Proc. VLDB Endow.* 12, 11 (2019), 1371–1384.

[29] Chao Li, Michael Hay, Gerome Miklau, and Yue Wang. 2014. A Data- and Workload-Aware Query Answering Algorithm for Range Queries Under Differential Privacy. *Proc. VLDB Endow.* 7, 5 (2014), 341–352.

[30] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. 2015. The matrix mechanism: optimizing linear counting queries under differential privacy. *VLDB J.* 24, 6 (2015), 757–781.

[31] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *Proc. VLDB Endow.* 13, 10 (2020), 1682–1695.

[32] Eric Lo, Nick Cheng, and Wing-Kai Hon. 2010. Generating Databases for Query Workloads. *Proc. VLDB Endow.* 3, 1 (2010), 848–859.

[33] Andrei Lopatenko and Leopoldo E. Bertossi. 2007. Complexity of Consistent Query Answering in Databases Under Cardinality-Based and Incremental Repair Semantics. In *ICDT*. 179–193.

[34] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. 2007. *L*-diversity: Privacy beyond *k*-anonymity. *ACM Trans. Knowl. Discov. Data* 1, 1 (2007), 3.

[35] Heikki Mannila and Kari-Jouko Räihä. 1989. Automatic Generation of Test Data for Relational Queries. *J. Comput. Syst. Sci.* 38, 2 (1989), 240–258.

[36] Ryan McKenna, Gerome Miklau, Michael Hay, and Ashwin Machanavajjhala. 2018. Optimizing error of high-dimensional statistical queries under differential privacy. *Proc. VLDB Endow.* 11, 10 (2018), 1206–1219.

[37] Ryan McKenna, Daniel Sheldon, and Gerome Miklau. 2019. Graphical-model based estimation and inference for differential privacy. In *ICML*, Vol. 97. 4435–4444.

[38] Stuart Mitchell, Stuart Mitchell Consulting, and Iain Dunning. 2011. PuLP: A Linear Programming Toolkit for Python. http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf.

[39] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[40] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *Proc. VLDB Endow.* 13, 3 (2019), 266–278.

[41] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* 29, 4 (2000), 64–71.

[42] Tilmann Rabl, Manuel Danisch, Michael Frank, Sebastian Schindler, and Hans-Arno Jacobsen. 2015. Just can't get enough: Synthesizing Big Data. In *SIGMOD*. 1457–1462.

[43] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.

[44] Anupam Sanghi, Raghav Sood, Jayant R. Haritsa, and Srikanta Tirthapura. 2018. Scalable and Dynamic Regeneration of Big Data Volumes. In *EDBT*. 301–312.

[45] William Sexton, John M. Abowd, Ian M. Schmutte, and Lars Vilhuber. 2017. Synthetic population housing and person records for the United States. https://www.openicpsr.org/openicpsr/project/100274/version/V1/view

[46] Entong Shen and Lyublena Antova. 2013. Reversing statistics for scalable test databases generation. In *DBTest*. 7:1–7:6.

[47] Joshua Snoke and Aleksandra B. Slavkovic. 2018. pMSE Mechanism: Differentially Private Synthetic Data with Maximal Distributional Similarity. In *PSD*, Vol. 11126. 138–159.

[48] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. 2017. Synthetic data generation for statistical testing. In *ASE*. 872–882.

[49] Ghanem Soltana, Nicolas Sannier, Mehrdad Sabetzadeh, and Lionel C. Briand. 2018. Model-based simulation of legal policies: framework, tool support, and validation. *Software and Systems Modeling* 17, 3 (2018), 851–883.

[50] Latanya Sweeney. 2002. k-Anonymity: A Model for Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* 10, 5 (2002), 557–570.

[51] Y. C. Tay, Bing Tian Dai, Daniel T. Wang, Eldora Y. Sun, Yong Lin, and Yuting Lin. 2013. UpSizeR: Synthetically scaling an empirical relational database. *Inf. Syst.* 38, 8 (2013), 1168–1183.

[52] TPC. 2020. TPC-H benchmark. http://www.tpc.org/tpch/.

[53] Stanley Gill Williamson. 2002. *Combinatorics for computer science*. Courier Corporation.

[54] William E. Winkler. 2004. Masking and Re-identification Methods for Public-Use Microdata: Overview and Research Problems. In *PSD*, Vol. 3050. 231–246.

[55] Ran Yahalom, Erez Shmueli, and Tomer Zrihen. 2010. Constrained Anonymization of Production Data: A Constraint Satisfaction Problem Approach. In *Secure Data Management, 7th VLDB Workshop*, Vol. 6358. 41–53.

[56] Jun Zhang, Graham Cormode, Cecilia M. Procopiuc, Divesh Srivastava, and Xiaokui Xiao. 2017. PrivBayes: Private Data Release via Bayesian Networks. *ACM Trans. Database Syst.* 42, 4 (2017), 25:1–25:41.