Efficient Detection of Data Dependency Violations

Eduardo H. M. Pena
Federal University of Technology - Paraná, Brazil
eduardopena@utfpr.edu.br

Edson R. L. Filho
Federal University of Paraná, Brazil
erlfilho@inf.ufpr.br

Eduardo C. de Almeida
Federal University of Paraná, Brazil
eduardo@inf.ufpr.br

Felix Naumann
Hasso Plattner Institute, Germany
felix.naumann@hpi.de

ABSTRACT

Research on data dependencies has experienced a revival as dependency violations can reveal errors in data. Several data cleaning systems use a DBMS to detect such violations. While DBMSs are efficient for some kinds of data dependencies (e.g., unique constraints), it is likely to fall short of satisfactory performance for more complex ones, such as order dependencies.

We present a novel system to efficiently detect violations of denial constraints (DCs), a well-known formalism that generalizes many kinds of data dependencies. We describe its execution model, which operates on a compressed block of tuples at-a-time, and we present various algorithms that take advantage of the predicate form in the DCs to provide effective code patterns. Our experimental evaluation includes comparisons with DBMS-based and DC-specific approaches, real-world and synthetic data, and various kinds of DCs. It shows that our system is up to three orders-of-magnitude faster than the other solutions, especially for datasets with a large number of tuples and DCs that identify a large number of violations.

CCS CONCEPTS

• **Information systems** → **Inconsistent data**; **Data cleaning**.

KEYWORDS

Error Detection, Data Cleaning, Integrity Constraints

ACM Reference Format:
Eduardo H. M. Pena, Edson R. L. Filho, Eduardo C. de Almeida, and Felix Naumann. 2020. Efficient Detection of Data Dependency Violations. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340531.3412062

1 INTRODUCTION

A fundamental aspect of data quality is *data consistency*. Fan gives a concise definition: "Data consistency refers to the validity and integrity of data representing real-world entities" [6]. A natural way to capture data inconsistencies is to detect violations of *data dependencies* [6]. Data dependencies express semantic properties that data should satisfy to represent a valid state of the real-world. In this regard, a dependency violation is a combination of values from one or more records in the database that do not satisfy the value relationship imposed by that dependency. A database is consistent if it holds no violation of the dependencies defined for it.

There has been much research on reasoning, discovery, and use of data dependencies [4, 6, 7]. An important question is whether a dependency formalism is able to capture the inconsistencies commonly found in production data, i.e., its expressiveness. Early work has proposed to capture inconsistencies of traditional dependencies, such as functional dependencies and inclusion dependencies [3]; and extensions of such dependencies have been presented to overcome expressiveness limitations [7]. These extensions include partial, conditional, and approximate versions of the traditional dependencies, which can capture exceptions or errors in the data [7]. Recent work has proposed to detect (and possibly repair) violations of different types of dependencies at once [4, 16]. Denial constraints (DCs) align with such a holistic view naturally. The formalism is one of the most general forms of dependency discussed in the literature since it generalizes several different types of dependencies [4, 6, 14]. A DC expresses a set of relational predicates that specify constraints on the combination of column values. Any tuple, or set of tuples, that disagrees with these constraints is a DC violation that reflects inconsistencies in the database.

The detection of DC violations is an expensive operation [4, 16]. DC-based data cleaning either rely on DBMSs [16] or implement a module [4] for this task. As many legitimate DCs express constraints on pairs of tuples, detecting their violations exhibits a quadratic time complexity in the number of tuples [4]. This complexity is perhaps the reason the experimental evaluations of DC-based systems are limited to simple dependencies (mainly functional dependencies) or small datasets. In many real-world scenarios, however, data cleaning has to deal with large datasets and complex DCs.

In this work, we present our DC violation detector VioFinder. There are three central ideas in its design: (i) Specialized data structures that reduce memory overheads and enable the algorithms in VioFinder to perform fast operations; (ii) A customizable operator that enables us to use effective algorithms to deal with complex DC predicates; and, (iii) An execution model that avoids the materialization of large intermediates and allows optimizations inter operators. Our experimental evaluation shows that VioFinder delivers efficient performance for several different kinds of DCs.

The remainder of this paper is as follows. In Section 2, we discuss the background and related work. In Section 3, we introduce the design of VioFinder, and in Section 4, its several algorithms. Then,

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3412062

in Section 5, we present our experimental results: We compare VioFinder with a DC-based tool and several DBMSs and demonstrate that our tool is orders of magnitude faster than the competitors in many cases. In Section 6, we present our final thoughts.

2 BACKGROUND AND RELATED WORK

In this section, we first present the fundamentals to represent data dependencies and data inconsistencies. Then, we review baseline approaches for the detection of data inconsistencies.

2.1 Denial constraints

Denial constraints (DCs) generalize many kinds of dependencies, and can express complex business rules [6, 14]. In this work, we focus on DCs representing unique constraints, functional dependencies, order dependencies, and dependencies that include comparisons of two tuples across two different columns—these types of dependencies are some of the most common in practice [6].

DCs use relationships between predicates to specify inconsistent states of column values. Let r denote a relation with schema R and n tuples; t a tuple of r; A and B columns of R; and $O = \{=, \neq, <, \leq, >, \geq\}$ a set of comparison operators of the database. A predicate p has the form $t.A\ o\ t'.B$: $A, B \in R$; $t, t' \in r$; and $o \in O$. Predicates can compare two tuples for the same column, i.e., $(A = B)$; compare column values to constants, or use many tuples at a time. Nonetheless, we need only DCs containing predicates without constants, on two different tuples, as they suffice to express the types of dependencies in this work. We express DCs as follows:

$$\varphi\colon \forall t, t' \in r, \neg(p_1 \wedge \ldots \wedge p_m),$$

We can translate each dependency into a set of predicates that express their semantics. As an example, consider the relation *hours* in Table 1. We can represent an unique constraint of the column combination (EmpID, ProjID) on *hours* with the following DC (the identifiers t, t' are omitted from now on):

$$\varphi_1\colon \neg(t.\text{EmpID} = t'.\text{EmpID} \wedge t.\text{ProjID} = t'.\text{ProjID})$$

For a relation to be consistent with a DC φ, there cannot exist any pair of tuples such that the conjunction of the predicates of φ is true. Consider the following constraint: For any two employees with the same role, the one who has worked more hours should not receive a lower bonus than the other. This constraint is expressed as a DC as follows:

$$\varphi_2\colon \neg(t.\text{Role} = t'.\text{Role} \wedge t.\text{Hours} > t'.\text{Hours} \wedge t.\text{Bonus} < t'.\text{Bonus})$$

In Table 1, tuples t_1 and t_2 share the same value of Role. Between those two, tuple t_1 has the highest value of Hours, so it should not have the lowest value of Bonus. That means that the pair of tuples (t_1, t_2) is a violation of φ_2, and hence Table 1 is inconsistent.

2.2 Detection of DC violations

A naive approach to detect the violations of a DC is to simply evaluate its conjunction of predicates for each pair of tuples. If the evaluation is true, then we add that pair of tuples to the result. This approach exhibits a quadratic time complexity in the number of records, which can be computationally prohibitive for large relations. A straightforward alternative is to use SQL with the query processing capabilities of DBMSs. However, this might not eliminate the quadratic complexity either, as we discuss next.

Table 1: An instance of the relation *hours*.

	EmpID	ProjID	Role	Hours	Bonus
t_1	E1	P1	Developer	4	$2000
t_2	E2	P1	Developer	2	$3000
t_3	E3	P1	Developer	4	$4000
t_4	E1	P2	DBA	4	$4000

The predicates of DCs compare the values of columns between two tuples of the same table. Therefore, a simple self-join query using the predicates of the DC in the where clause exposes the violations. The following example shows a SQL query that finds the EmpID's of tuple pairs that violate the DC φ_2:

```
1  select t.EmpID, t'.EmpID
2  from   hours t, hours t'
3  where  t.Role = t'.Role
4  and    t.Hours > t'.Hours
5  and    t.Bonus < t'.Bonus;
```

Related work has reported that self-joins (and mainly inequality self-joins) have received little attention in commercial DBMSs [11]. Indeed, our experiments with three different DBMSs exposed two main issues: (i) excessive memory requirements; and (ii) use of ineffective join algorithms. Some DBMSs run out of memory or took more than one hour to execute queries for common functional dependencies on samples with $200K$ tuples. Besides, most DBMSs rely on nested-loop approaches for self-joins with range predicates, which may result in extremely long runtimes.

Indices might not help either: some conditions to detect violations require validating all the records with table scans. The DBMS may not use the indices in the query plans, and the few cases that indices are chosen do not pay off for the costs of index creation. One of the reasons for the poor performance of DBMSs is the expected cost to materialize self-joins, which is quadratic in the number of records in the worst case [1]. This cost is evident when DCs require high-cardinality predicates, such as a range predicate for an order dependency with many qualifying tuples. We refer to [15] for a study related to self-join cardinality estimation.

2.3 Related work

The underlying violation detection mechanism of several data cleaning tools is a traditional DBMS [5, 8, 16]. These tools inherit the performance issues discussed earlier, and their evaluation experiments used small datasets or only simple dependencies, such as functional dependencies. Implementing a dedicated DC violation module is an alternative, for instance, Chu et al. do so using pairwise comparisons [4]. However, their experimental evaluation also used only a small number of records (i.e., up to $100K$ tuples).

The issue of scalability in data cleaning is studied by Khayyat et al. [10]. The authors introduce a framework to perform violation detection and database repairing in distributed settings. The core idea is to translate data cleaning rules (expressed in UDF-based form) into jobs that are executed on top of parallel data processing

frameworks. Although our approach focuses on centralized environments, it is able to efficiently detect violations for very large datasets. Nonetheless, extending our approach for distributed data processing environments is an interesting topic for future work.

Closer to our work is the DC violation detection component of Hydra – a state-of-the-art algorithm for DC discovery [2]. Efficient detection of DC violations is a central part of the algorithm, so the authors have proposed novel techniques to handle the problem. There are two main ideas in this component: The use of specialized data structures; and the customization of algorithms for different predicate types. While these ideas have inspired our project, the way VioFinder organizes and operates on its data structures is different from Hydra. For example, Hydra uses the IEJoin algorithm, which has been shown to deliver efficient performance for self-joins based on range predicates [11]. Our system, in turn, uses a novel sort-merge approach that can be even faster than IEJoin. We also use different approaches for other types of predicate, as discussed later in this paper. We use Hydra and IEJoin as the main baselines in our experimental evaluation.

3 THE VIOFINDER SYSTEM

This section introduces principles that enable VioFinder to avoid performance issues, such as materialization overheads and nested-loop joins. As a result of these principles, VioFinder can deliver robust performance for different types of data dependencies.

3.1 Cluster, cluster pairs, and partitions

VioFinder uses specialized data structures to represent enumerations of pairs of tuples compactly. We define these structures first, as they are key to understanding how VioFinder works. A *cluster* c is a set of tuple identifiers (the tuple position within the table). A *cluster pair* is an ordered pair (c_1, c_2) that represents the set of all pairs of tuples (t, t'), such that $t \in c_1$, $t' \in c_2$ and $t \neq t'$. For instance, the cluster pair $(\{t_1\}, \{t_1, t_2, t_3\})$ represents the set of pairs of tuples (t_1, t_2), (t_1, t_3). A *partition* L is any set of cluster pairs. Partitions consume much less memory than exhaustive enumerations of pairs of tuples. For a relation r with n tuples, the cluster pair $(\{t_1, \ldots, t_n\}, \{t_1, \ldots, t_n\})$ represents the whole Cartesian product $r \times r$ using only $2n$ integers, whereas the equivalent enumeration of pairs of tuples requires $n(n-1)$ pairs of integers to do so.

3.2 Refinement of columns and partitions

A fundamental operation of VioFinder is the *refinement of columns*. A *column refiner* takes as input one predicate and returns partitions containing cluster pairs that represent every pair of tuples that is true for the input predicate. As an example, consider the refinement of columns for the predicate t.Role = t′.Role and the records in Table 1. The refinement gives us a partition with a single cluster pair: $[(\{t_1, t_2, t_3\}, \{t_1, t_2, t_3\})]$—the cluster pair $(\{t_4\}, \{t_4\})$ is discarded since it does not produce any pair of different tuples. The main primitive here is a full table scan for each column of the predicate. In Section 4, we describe how to implement the refinement of columns for the different comparison operators. For now, we assume column refiners to be "black-boxes." We assume a random sequence of refinements—we discuss the ordering of refinements in Section 3.5.

A second fundamental operation of VioFinder is the *refinement of partitions*. Each *partition refiner* takes as input a predicate and a partition and produces new partitions containing cluster pairs with every pair of tuples that is true for the input predicate, and of course, true for the predicates in the past refinements that produced the input partition. As an example, consider the partition from predicate t.Role = t′.Role described earlier: $[(\{t_1, t_2, t_3\}, \{t_1, t_2, t_3\})]$. Pushing this partition into the refinement of partitions for the predicate t.Hours > t′.Hours produces the partition: $[(\{t_1, t_3\}, \{t_2\})]$. If we push this last partition further into the refinement of partitions for the predicate t.Bonus < t′.Bonus, we obtain the partition $[(\{t_1\}, \{t_2\})]$. This partition represents the violations of DC φ_2.

The refinement of partitions is similar to the refinement of columns. However, the former requires fetching only the values of columns of the tuples in the partitions, instead of entire columns as the latter requires. Another difference is in the type of optimizations we can use within each kind of refinement—we describe these optimizations in Section 4.

3.3 Cluster indexes

A basic step in the refinement of columns is the creation of cluster indexes on the columns of predicates. Let V be the set of values in the domain of column A. For every value $v \in V$, we assign a cluster c with all tuples having v as the value in column A. The cluster index $\mathcal{H}_A$ is a hash map where each entry maps a value $v \in V$ into its cluster c. For instance, the cluster index $\mathcal{H}_{\text{Role}}$ is: [⟨"Developer", $\{t_1, t_2, t_3\}$⟩, ⟨"DBA", $\{t_4\}$⟩]. Similarly, the refinement of partitions requires the creation of conditioned cluster indexes $\mathcal{H}_{A,c}$. We fetch column values of the tuples in the cluster then create a hash map such that each distinct value fetched is mapped into a cluster with all tuples having that value. For example, the conditioned cluster index $\mathcal{H}_{\text{Hours},\{t_1,t_2,t_3\}}$ is: [⟨2, $\{t_2\}$⟩, ⟨4, $\{t_1, t_3\}$⟩].

We considered three facts to choose an implementation for clusters, which are essentially sets of integers. First, the size of cluster indexes grows linearly with the number of distinct values of a column since these values are mapped to one cluster each. Second, refinement algorithms produce partitions containing many cluster pairs. Third, these algorithms have to compute unions or differences of clusters. These facts led us to employ Roaring (compressed) bitmaps, a hybrid data structure that combines bitmaps with sorted arrays to achieve good compression rates [13]. As a result, we can store large numbers of clusters with many integers using less memory. Besides, Roaring bitmaps perform fast unions and differences as bitwise `OR` and `AND NOT` operations, which are, in many cases, even faster than non-compressed counterparts. For algorithmic details on Roaring bitmaps, we refer the reader to [13].

3.4 Partition pipelines

VioFinder assigns a refiner to each DC predicate based on the predicate's form and refiners connect through a partition pipeline. VioFinder keeps each column of the dataset as an in-memory array so that refiners can readily fetch the values of the columns in their predicates. Partition pipelines work as push-based iterations. Figure 1 illustrates a pipeline with three refiners. Each partition is linked to either a next refiner or to the output. In the former case, the current refiner produces a new partition and pushes it to the

next refiner, which immediately starts consuming the cluster pairs one by one. In the latter case, no more refinement is necessary, so partitions are pushed to the output. At this point, the concrete violations are materialized.

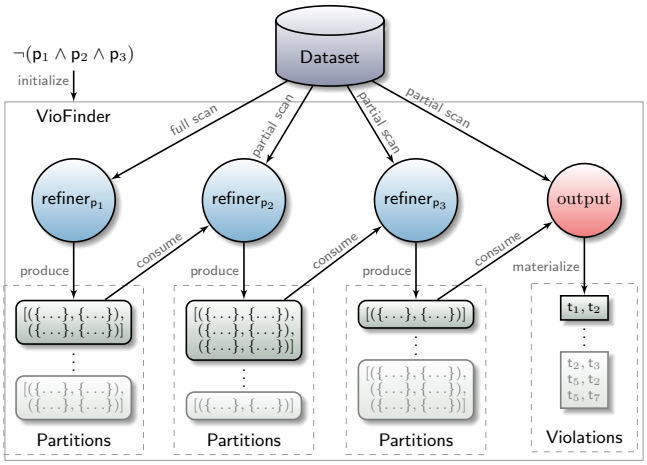

Figure 1: Example of a partition pipeline.

The partition pipeline has the following properties:

Customizable refinement. Conceptually, refiners implement a produce/consume interface so they can use different refinement implementations and optimizations at different stages of the pipeline. Instead of using a general-purpose refinement strategy (e.g., nested loop), VioFinder uses different refinement strategies depending on the form of each predicate.

Controlled intermediates. Some refinements might produce large intermediates. Thus, our refinement algorithms check the size of current partitions before pushing new tuples into the pipeline to avoid excessive resource utilization. As a result, refinements can use logical optimizations that work for multiple tuples at a time, while avoiding materializing large intermediates.

Late materialization. VioFinder does not need to materialize results until after the last refinement in the pipeline. Column refiners fetch only values of the columns in its predicates, and partition refiners do so only for tuples from previous refinements. Such a scheme maximizes the use of memory bandwidth: only the necessary parts of relevant tuples are fetched in each stage of the pipeline.

Cluster pair processing. The actual refinement is computed at the level of cluster pairs with four primary steps:

(a) Iteration over the tuples in each cluster—a tight loop suffices to iterate entire clusters fast, because they usually have far fewer tuples than the relation.
(b) Fetch of column values—as already mentioned, only the column values that are relevant for a refinement are fetched.
(c) Build of auxiliary data structures—the auxiliary data structures in refinements usually have a low memory footprint since they grow with the size of clusters.
(d) Refinement logic—some forms of partitions enable refinements to skip tuple fetches, which improves performance.

The steps above also apply to column refiners, with the difference that entire columns are fetched in Steps (a) and (b).

3.5 Order of refinements

The *order* of the DC predicates (and, therefore, refinements) has a significant impact on performance. Choosing a poor predicate order might produce large intermediate partitions, which causes significant overhead in intermediate refinements. We choose the order of predicates based on predicate selectivity. The selectivity of a predicate is the fraction of pairs of tuples in a relation that satisfy that predicate. We estimate approximate selectivities using a small random sample of pairs of tuples without replacement, and then we order the predicates from most selective to least selective. This technique is also used in Hydra [2]; however, the algorithm uses a larger sample. For every tuple in the dataset, Hydra samples a small number of other tuples to form pair of tuples. In our experiments, we use a small sample bounded to $1M$ elements, as it consistently produced the same predicate order as Hydra's samples, and it was faster to estimate. We refer to [9] for a discussion on selectivity estimation, such a discussion is beyond the scope of the paper.

4 REFINEMENT ALGORITHMS

DCs express predicates of different forms to support a wide range of data dependencies. These predicates include comparison within a single column or across two different columns and use different operators. In the following, we present refinement algorithms that take the predicate form into account for efficiency. For convenience, we present our algorithms as equijoins with the *equal to* operator =, antijoins with the *not equal to* operator ≠, and non-equijoins with *range* operators $\{<, \leq, >, \geq\}$. Most of the algorithms handle a single predicate at a time. For better performance, one of the algorithms handles multiple predicates at a time.

4.1 Equijoins

The most basic form of refinement is the refinement of columns for equality predicates on a single column, such as t.A = t′.A. The first step is to build a cluster index $\mathcal{H}_A$. Each cluster c of $\mathcal{H}_A$ is precisely a set of tuples having the same value v, so we can use cluster pairs in the form of reflexive relations (c, c) to represent all pairs of tuples that have the same value v in column A. Clusters with only one tuple are ignored, because they cannot produce pairs of distinct tuples. We insert each valid cluster pair (c, c) into the output partition L and check its size. If the number of cluster pairs in the partition L exceeds a threshold, we stop iterating the clusters in the cluster index $\mathcal{H}_A$, and push the partition L into the next level of the pipeline. The next call to the refiner skips the clusters pairs of $\mathcal{H}_A$ that have already been processed.

Some refinements might produce large intermediate results. After all, refinements are equivalent to the self-joins in the predicates of a DC, which often join non-key columns. Nonetheless, we can avoid the full materialization of large intermediates by controlling the size of partitions currently being processed, as in our first refinement algorithm. Refinements stop producing new cluster pairs as soon as the number of cluster pairs in a partition exceeds a threshold, or when it has no more pairs of tuples to compute. In the former case, the state of the refinement is saved so that the next call to it starts producing new cluster pairs from where it stopped earlier. For simplicity, we do not elaborate on these procedures for the remainder refinement algorithms.

Equality predicates on single columns are very common in DCs. For instance, DCs use them to represent unique constraints or the left-hand side of functional dependencies. Compared to other forms of predicates, equality predicates have higher selectivity so they usually come first in the refinement pipeline. Also, partitions can only reduce in size as they go through the pipeline stages for sets of predicates with this form. For instance, the partition for predicate t.Role = t′.Role is $[(\{t_1, t_2, t_3\}, \{t_1, t_2, t_3\})]$, and the partition for the conjunction of predicates t.Role = t′.Role ∧ t.Hours = t′.Hours is $[(\{t_1, t_3\}, \{t_1, t_3\})]$. We take advantage of this fact with a code pattern that reduces clusters as fast as possible and, therefore, reduces the materialization of intermediate partitions.

Algorithm 1 is a special case of refinement that handles multiple predicates at once, namely multiple equality predicates on single columns. In the initial call `refineCluster`(c_r, A_1, L) in Line 12 we build a cluster index with every tuple in the table. When we call `refineCluster`(c, A_i, L) for $i > 1$, every tuple in c has the same combination of values in columns $A_1, \ldots, A_{i-1}$. As a consequence, the tuples in the clusters c' of the conditioned cluster index $\mathcal{H}_{A_i,c}$ (Line 2) have the same combination of values in columns $A_1, \ldots, A_i$. The base case occurs when there are no further predicates to check, in which case we insert cluster pair (c, c) into the output partition L (Line 9).

Algorithm 1: Refinement of columns for predicate sequence of the form p_1: t.A_1 = t′.A_1, …, p_i: t.A_i = t′.A_i

```
1  Function refineCluster(c, A_i, L)
2      let H_{A_i,c} be a conditioned cluster index
3      let C′ be the set of clusters in H_{A_i,c}
4      foreach c′ ∈ C′ do
5          if c′.size > 1 then
6              if there exists a predicate p_{i+1} then
7                  refineCluster(c′,A_{i+1},L)
8              else
9                  Insert cluster pair (c′, c′) into L
10 initialize an empty partition L
11 initialize a cluster c_r with every tuple of table r
12 refineCluster(c_r, A_1, L)
13 return L
```

The refinement of columns for equality predicates on two *different* columns, t.A = t′.B, is similar to traditional hash joins. We first build cluster indexes $\mathcal{H}_A$ and $\mathcal{H}_B$. The cluster index $\mathcal{H}_A$ acts as the "build input", whereas cluster index $\mathcal{H}_B$ acts as the "probe input"—we assume column A to produce fewer entries than column B. We iterate the values v in the cluster index $\mathcal{H}_A$ and, for each of those, we probe cluster index $\mathcal{H}_B$. If cluster index $\mathcal{H}_B$ contains the value v, then we combine the cluster assigned to the value v in $\mathcal{H}_A$, denoted c_A, with the cluster assigned to the value v in $\mathcal{H}_B$, denoted c_B. The cluster pair (c_A, c_B) indicates that every tuple $t \in c_A$ have the same value in column A, which is equal to the value of every tuple $t' \in c_B$ in column B.

Algorithm 2 shows the refinement of partitions for an equality predicate on two (not necessarily different) columns. We iterate each cluster pair (c_1, c_2) in the input partition, for which we retrieve two conditioned cluster indexes: $\mathcal{H}_{A,c_1}$ and $\mathcal{H}_{B,c_2}$. The remainder of the algorithm is analogous to the refinement of columns for equality predicates on two different columns. The difference is that build-inputs are conditioned cluster indexes $\mathcal{H}_{A,c_1}$, whereas probe-inputs are conditioned cluster indexes $\mathcal{H}_{B,c_2}$.

Algorithm 2: Refinement of partition L_{in} for predicates of the form t.A = t′.B (A and B can be equal)

```
1  initialize an empty partition L_out
2  foreach cluster pair (c_1, c_2) ∈ L_in do
3      let H_{A,c_1} and H_{B,c_2} be conditioned cluster indexes
4      let V be the set of values in H_{A,c_1}
5      foreach v ∈ V do
6          c_B ← H_B(v)
7          if c_B is not null then
8              c_A ← H_A(v)
9              Insert cluster pair (c_A, c_B) into L_out
10 return L_out
```

The algorithms we have presented so far take linear time in the number of tuples. In short, we fetch column values, build cluster indexes using hashing, and iterate the entries in these clusters to emit partitions.

4.2 Antijoins

The following refinement of columns detects pairs of tuples having different values of a single column, i.e., predicates of the form t.A ≠ t′.A. We need to insert cluster pairs (c, c') into the result partition: Cluster c is each cluster of the cluster index $\mathcal{H}_A$; and cluster c' is the relative complement of cluster c in a cluster with all tuples in the table c_r— also termed set difference $c' \leftarrow c_r \setminus c$.

We do as follows to detect pairs of tuples having different values for two different columns, t.A ≠ t′.B. Given an entry $\langle v, c_A \rangle$ in the cluster index $\mathcal{H}_A$, we check whether there exists an entry $\langle v, c_B \rangle$ in the cluster index $\mathcal{H}_B$. If so, we insert a cluster pair (c_A, c') into the result partition, where $c' \leftarrow c_r \setminus c_B$. Otherwise, the value in column A of the tuples in c_A is different from the values in column B of every tuple in the table. In this case, we can insert a cluster (c_A, c_r) into the result.

The refinement of partitions for antijoin predicates is given as Algorithm 3. For each cluster pair (c_1, c_2) in the input partition L_{in}, we retrieve the conditioned cluster indexes $\mathcal{H}_{A,c_1}$ and $\mathcal{H}_{B,c_2}$. Then, for each value v (with assigned cluster c_A) of cluster index $\mathcal{H}_{A,c_1}$ we search for a cluster c_B in the cluster index $\mathcal{H}_{B,c_2}$. In a successful search, we use the relative complement of cluster c_B in the cluster c_2 to form the result cluster pair with c_A (Lines 8–10). Otherwise, cluster c_2 has no tuple whose value in B is v, so it can be directly combined with cluster c_A in Line 12.

About time complexity, building, and probing cluster indexes takes linear time in the number of tuples. Also, the algorithms for antijoin predicates have the additional cost of set difference operations (e.g., Line 9 in Algorithm 3). The selectivity of these types of predicates are usually low, so their respective refinements

Algorithm 3: Refinement of partition L_{in} for predicates of the form t.A ≠ t′.B (A and B can be equal)

```
1  initialize an empty partition L_out
2  foreach cluster pair (c1, c2) ∈ L_in do
3      let H_{A,c1} and H_{B,c2} be conditioned cluster indexes
4      let V be the set of values in H_{A,c1}
5      foreach v ∈ V do
6          c_A ← H_A(v)
7          c_B ← H_B(v)
8          if c_B is not null then
9              c′ ← c2 \ c_B
10             Insert cluster pair (c_A, c′) into L_out
11         else
12             Insert cluster pair (c_A, c2) into L_out
13 return L_out
```

might produce large intermediate partitions. In practice, these types of refinement come last in the pipeline, at a point where most pairs of tuples have already been filtered out.

4.3 Non-equijoins with range operators

Let us next consider the refinement of columns for range predicates of the form t.A > t′.A. We build the cluster index $\mathcal{H}_\mathsf{A}$ and sort its entries in ascending order according to the keys (the distinct values of the column). For clarity, we denote such sorted maps with $\vec{\mathcal{H}}_\mathsf{A}$. For the sorted entries $\langle v_1, c_1\rangle, \ldots, \langle v_i, c_i\rangle \in \vec{\mathcal{H}}_\mathsf{A}$ we have the following: Every tuple in the cluster c_i has a value v_i that is greater than the values v_j in the tuples of clusters c_j, for all $j < i$. For each cluster c_i, we form a cluster pair $(c_i, c_i{}')$ such that $c_i{}' = \bigcup_{j=1}^{i-1} c_j$. At each iteration i, we compute the cluster $c_i{}'$ using a copy of the last cluster $c_{i-1}{}'$ and only one union operation. Finally, we insert each cluster pair $(c_i, c_i{}')$ into the output partition L. For predicates of the form t.A >= t′.A we must include c_i into the right-hand side of the cluster pair, so we compute clusters $c_i{}' = \bigcup_{j=1}^{i} c_j$. The algorithm is symmetric for predicates of the form t.A < t′.A and t.A ≤ t′.A with the entries of the cluster index $\mathcal{H}_\mathsf{A}$ in descending order according to the keys. In the worst case, the values of column A are all distinct; thus, cluster indexes have n entries. In this case, the time complexity is dominated by the time spent to sort these n entries plus the time to perform n union operations.

The remaining of the refinement algorithms are based on the sort-merge paradigm. The general idea is to iterate sorted cluster indexes to find and build matching cluster pairs from previous iterations incrementally. Algorithm 4 shows the refinement of columns for a predicate on two different columns, such as t.A > t′.B. After building sorted cluster indexes $\vec{\mathcal{H}}_\mathsf{A}$ and $\vec{\mathcal{H}}_\mathsf{B}$ in Line 2, we filter their values out for those entries that cannot form cluster pairs that satisfy the predicate. That is, we remove from cluster index $\vec{\mathcal{H}}_\mathsf{A}$ the entries with values that are smaller than the smallest value of cluster index $\vec{\mathcal{H}}_\mathsf{B}$, and from $\vec{\mathcal{H}}_\mathsf{B}$ the entries with values that are greater than the greatest value of $\vec{\mathcal{H}}_\mathsf{A}$. If the cluster indexes $\vec{\mathcal{H}}_\mathsf{A}$ and $\vec{\mathcal{H}}_\mathsf{B}$ are empty at this point, there are no matching cluster pairs so the algorithm returns an empty partition. Otherwise, the first entry $\langle v_{\text{high}}, c_{\text{high}}\rangle$ of cluster index $\vec{\mathcal{H}}_\mathsf{A}$ has a value that is strictly greater than the value of the first entry $\langle v_{\text{low}}, c_{\text{low}}\rangle$ of cluster index $\vec{\mathcal{H}}_\mathsf{B}$, so we form the first cluster pair that satisfy the predicate (Lines 4– 7). Such cluster pairs are kept in variables *pair* that are updated as we find new matching clusters.

Algorithm 4: Refinement of columns for predicate of the form t.A > t′.B

```
1  initialize an empty partition L
2  let H⃗_A and H⃗_B be sorted cluster indexes
3  remove from H⃗_A and H⃗_B those entries that do not produce
       cluster pairs for t.A > t′.B
4  ⟨v_high, c_high⟩ ← H⃗_A.next()
5  ⟨v_low, c_low⟩ ← H⃗_B.next()
6  pair ← (c_high, c_low)
7  Insert pair into L
8  if H⃗_A.hasNext() or H⃗_B.hasNext() then
9      while H⃗_B.hasNext() do
10         ⟨v_low, c_low⟩ ← H⃗_B.next()
11         if v_high > v_low then
12             pair.rhs ← pair.rhs ∪ c_low
13         else
14             while H⃗_A.hasNext() do
15                 ⟨v_high, c_high⟩ ← H⃗_A.next()
16                 if v_high <= v_low then
17                     pair.lhs ← pair.lhs ∪ c_high
18                 else
19                     c_temp ← a copy of pair..rhs
20                     c_low ← c_temp ∪ c_low
21                     pair ← (c_high, c_low)
22                     Insert pair into L
23                     break
24     while H⃗_A.hasNext() do
25         ⟨v_high, c_high⟩ ← H⃗_A.next()
26         pair.lhs ← pair.lhs ∪ c_high
27 return L
```

The merging part of the algorithm begins in Line 9. We take the value v_{high} used to form the current *pair* and find matching entries $\langle v_{\text{low}}, c_{\text{low}}\rangle$ in the cluster index $\vec{\mathcal{H}}_\mathsf{B}$ that also satisfy the predicate. Then, we update the right-hand side of *pair* to include the tuples of clusters c_{low} (Lines 9-12). Whenever we find a non-matching entry, we update the left-hand side of *pair* (Lines 14–17). That is because there might be entries in $\vec{\mathcal{H}}_\mathsf{A}$ with values v_{high} that, despite being smaller than the current v_{low}, are greater than the values v_{low} previously used in Lines 9-12. By doing this, we keep as much tuples as possible within the the same cluster pair. We find the starting point of a new matching cluster pair whenever we find a new entry $\langle v_{\text{high}}, c_{\text{high}}\rangle$ in $\vec{\mathcal{H}}_\mathsf{A}$ with a value v_{high} greater than the current v_{low} (the else clause in Line 18). At this point, the left-hand

side of the new cluster pair is c_{high} and its right-hand side is the union of the tuples in the current c_{low} with all tuples in the c_{low} from previous iterations. In other words, the right-hand side of *pair* can only expand. We repeat the while loop in Line 9 until there is no entry in $\vec{\mathcal{H}}_{\text{B}}$ to visit. Finally, we perform a last update in the left-hand side of the last *pair* with any left entry of cluster index $\vec{\mathcal{H}}_{\text{A}}$ (Lines 24–26).

The time complexity for Algorithm 4 is given by the time spent to build and sort cluster indexes, plus the time spent in merging these clusters. While the merge loop runs in $O(2n)$ (assuming n entries in each cluster index), performing cluster unions and copies depends on the internal states of their bitmaps.

Algorithm 4 requires minor changes to work with operator $\geq$, and it is symmetric for operators in $\{<, \leq\}$, with cluster indexes $\vec{\mathcal{H}}_{\text{A}}$ and $\vec{\mathcal{H}}_{\text{B}}$ sorted in descending order of keys. The refinement of partitions for predicates with operators in $\{>, \geq, <, \leq\}$ and two (not necessarily different) columns also follows Algorithm 4 with minor changes. The starting point is building conditioned cluster indexes for each cluster pair in the input partition. The remainder of the algorithm is the same as described above.

4.4 Cached cluster indexes

The partitions produced by refinements of range predicates, with operators in $\{>, \geq, <, \leq\}$, have a great deal of redundancy across the right-hand sides of their cluster pairs. As an example, observe the output of the refinement of columns for predicate t.Bonus < t′.Bonus: $[(\{t_2\}, \{t_1\}), (\{t_3, t_4\}, \{t_1, t_2\})]$. If we were to compute conditioned cluster indexes for cluster $\{t_1\}$ and $\{t_1, t_2\}$ from scratch, we would require to fetch tuple t_1 twice instead of just once. For larger clusters, the waste would be high, and the running time would increase dramatically. To avoid unnecessary tuple fetches, VioFinder employs a simple but efficient cache mechanism.

The cache works for the refinement of partitions holding incremental redundancy on the right-hand side of their cluster pairs. Such partitions derive from refinements (of both columns or partitions) that use predicates with operators in $\{>, \geq, <, \leq\}$. VioFinder maintains a conditioned cluster index $\mathcal{H}_{\text{A},c_{cache}}$, where cluster c_{cache} is a set of tuples with its values of column A already fetched. Assume we are about to build a conditioned cluster index $\mathcal{H}_{\text{A},c}$. We compute the relative difference of c_{cache} in c, that is, $c_{diff} = c \setminus c_{cache}$. If this result is non-empty, then we already have a portion of the cluster index $\mathcal{H}_{\text{A},c}$ as the cluster index $\mathcal{H}_{\text{A},c_{cache}}$. In this case, we fetch the remaining values of column A we need: the tuples of c_{diff}. We use these values to update $\mathcal{H}_{\text{A},c_{cache}}$. At this point, the cluster index $\mathcal{H}_{\text{A},c_{cache}}$ holds the entries required for $\mathcal{H}_{\text{A},c}$, so that we can proceed with the remaining parts of the refinement. On the other hand, an empty result of the relative difference means that the sequence of redundant tuple has stopped, so we can no longer use the previous $\mathcal{H}_{\text{A},c_{cache}}$. In this case, we must build a new cluster index $\mathcal{H}_{\text{A},c_{cache}}$ from scratch.

5 EXPERIMENTAL EVALUATION

We ran several experiments with VioFinder, three DBMSs, and a system tailored for DCs. In this section, we compare the performance of these systems and analyze the design choices of VioFinder.

5.1 Experimental setup

Datasets and DCs. We used three datasets and eight DCs, as shown in Table 2. The Tax dataset is a synthetic compilation of tax-records of US individuals. We generated several Tax instances (with up to $100M$ records) using the data generator from [7]. The DCs φ_3–φ_5 are defined for the single table of the Tax dataset. The TPC-H dataset is extracted from the synthetic TPC-H benchmark[1]. We used a scale factor of ten to produce TPC-H instances with up to $60M$ records. The DC φ_6 is defined for the denormalization of tables *lineitem* and *orders*, and the DCs φ_7 and φ_8 are defined for the *lineitem* table alone. The IMDB dataset is extracted from the real-world movie dataset [2] described in [12]. The DC φ_9 is defined for the denormalization (with up to $2.5M$ records) of tables *title* and *kind_type*, and the DC φ_{10} is defined for the denormalization (with up to $5.8M$ records) of tables *cast_info*, *title*, *aka_name*, *name*, *role_type*, and *char_name*. These DCs were designed to cover various types of dependencies: Unique constraints (φ_3, and φ_{10}), functional dependencies (φ_4 and φ_9), order dependencies (φ_7), and other dependencies with complex relationships (φ_5, φ_6, and φ_8). Although some of them may not hold in production, they have complex predicate structures that challenge the performance of the evaluated systems.

Table 2: Datasets and denial constraints for experiments.

Dataset	DC
Tax	φ_3: ¬(t.AreaCode = t′.AreaCode ∧ t.Phone = t′.Phone)
Tax	φ_4: ¬(t.State = t′.State ∧ t.HasChild = t′.HasChild ∧t.ChildExemp ≠ t′.ChildExemp)
Tax	φ_5: ¬(t.State = t′.State ∧ t.Salary > t′.Salary ∧t.Rate < t′.Rate)
TPC-H	φ_6: ¬(t.Customer = t′.Supplier ∧ t.Supplier = t′.Customer)
TPC-H	φ_7: ¬(t.Extended_price > t′.Extended_price ∧t.Discount < t′.Discount)
TPC-H	φ_8: ¬(t.Receiptdate ≥ t′.Shipdate ∧ t.Shipdate ≤ t′.Receiptdate)
IMDB	φ_9: ¬(t.Title = t′.Title ∧ t.ProductionYear = t′.ProductionYear ∧t.Kind ≠ t′.Kind)
IMDB	φ_{10}: ¬(t.Title = t′.Title ∧ t.Role = t′.Role ∧t.Name = t′.Name ∧ t.CharName = t′.CharName)

Baselines. We compare VioFinder with the DC violation detection component described in [2], referred to here as Hydra–IEJoin. Besides, we compared our system with three DBMSs: PostgreSQL (v.12.1), MonetDB (v.11.35.3), and SQLServer (v.2019 CU3). These systems have different query processing models, with different impacts on the materialization of intermediate data. PostgreSQL implements the tuple-at-a-time model that moves entire tuples around the memory hierarchy. In contrast, the column-at-a-time processing model of MonetDB fetches only the columns in the SQL statement but keeps the intermediate data in memory during processing. Finally, SQLServer implements a middle ground with a vector-at-a-time model.

[1] http://www.tpc.org/tpch/
[2] https://homepages.cwi.nl/~boncz/job/

Implementation. We implemented VioFinder as a standalone tool in Java that runs in main-memory after dataset loading. We used the Roaring bitmap library to implement clusters[3]. Hydra–IEJoin is also a standalone tool running in main-memory. We used the Java implementation provided by the authors. To use the DBMSs, we translated each DC in Table 2 into a SQL query and executed it using the vanilla version of the three DBMSs. We created indexes on all predicate columns to investigate if and when the DBMSs improve their execution plans. We checked all implementations separately, and all return the same result. As we did not need to materialize the violations, we used `select count(*)` in each query to return only the number of violations, and we set the standalone tools to return a count with the number of violations they found.

Infrastructure and execution. We used a server running Debian 10 (buster) as the experimentation platform. The server is equipped with twelve sockets, each with an Intel(R) Xeon(R) CPU E7-8837 octa-core processor running at 2.67GHz, 756GB of RAM, and 2TB of disk. All executions were single-threaded. VioFinder and Hydra–IEJoin run on an Oracle's JDK 64-Bit Server VM 1.8.0 with the maximum heap size set to 32GB. The numbers in the reports are the average measurement of three independent runs. We used a default threshold of ten cluster pairs for VioFinder.

5.2 Performance evaluation

Comparison with baselines. We measured the runtime of all DC violation detectors on different datasets and DCs. To be able to run the SQL queries within a time limit of 3 hours, we used a sample with $200K$ records of each dataset. Runtimes are broken down into loading, preprocessing, and querying. For the DBMSs, these measures are, respectively, the time spent to load the raw files into the DBMS, create indexes, and execute the query. For Hydra-IEJoin, these measures are, respectively, the time spent to load the raw files into memory, map the input into integer domains plus the time to decide predicate order, and execute the algorithm. VioFinder's runtime composition is similar to Hydra-IEJoin's, except that it does not include the input mapping time.

Figure 2 depicts the measured runtimes of all five systems for all datasets and DCs of Table 2. In summary, the results in this experiment demonstrate that VioFinder performs best in every scenario and that it can be at times orders of magnitude faster than the DBMS approaches. For DCs φ_7 and φ_8, VioFinder finished in a matter of few seconds, PostgreSQL and SQLServer in a matter of few hours, and MonetDB did not finish due to memory limit exceptions. We can see speedups of 1625×, for example, when VioFinder is compared to SQLServer for DC φ_8. Moreover, VioFinder delivered between 3× and 17.5× faster executions than Hydra-IEJoin.

The execution plans and performance among the evaluated DBMSs varied considerably. For the keys in DCs φ_3 and φ_{10} and the functional dependency in DC φ_9, PostgreSQL used a sort-merge join approach slower than the hashjoins in SQLServer and MonetDB—we can see the performance impact from algorithm choice in the querying time. All systems used hashjoins for the relationship of mutual inclusion in DC φ_6 and reported fast querying. In contrast, we measured the worst runtimes for DCs that express relationships of order between columns (i.e., DCs φ_5, φ_7 and φ_8). MonetDB threw memory limit exceptions for DCs φ_7 and φ_8 and reported the slowest runtime for DC φ_5. The system used a thetajoin implementation based on a Cartesian product that produced large intermediates and impaired performance. PostgreSQL and SQLServer relied on nested loops for those three DCs and performed poorly considering the small number of tuples in the experiment.

Regarding index usage, the DBMSs used table scans for most of the executions due to the selectivity of the predicates. MonetDB and SQLServer used no indices, whereas PostgreSQL used index scans on column Extended_price for the DC φ_7 and on column Shipdate for the DC φ_8. The order of predicate evaluation also influenced performance: For DC φ_4, SQLServer used hashjoins to evaluate the equijoin predicates, then checked the non-equijoin as a residual predicate. The two other DBMSs also used hashjoins, but evaluated the non-equijoin filter first, yielding the worst runtime. For DC φ_5, all systems evaluated the equijoin predicate first, which helped reducing intermediates and improved performance.

The differences in the executions of the DBMSs were expected: after all, they differ from each other internally. These results support our design decisions with VioFinder, though. By processing partitions of limited size at-a-time, VioFinder bounds the materialization of intermediates. Choosing the order of DC predicates based on predicate selectivity leads VioFinder to process predicates that produce smaller intermediates first. Besides, VioFinder carefully selects refinement algorithms. Notice that the best results reported by the DBMSs use hash-based approaches. VioFinder mirrors this observation and uses hash-like approaches whenever possible. For range predicates, VioFinder uses algorithms that are more effective than the nested loop solutions in the DBMSs. We observe similar concerns with Hydra-IEJoin. However, VioFinder spends much less time than Hydra-IEJoin in preprocessing.

Scalability in the number of tuples. This experiment considers only querying times (i.e., execution times without loading, index creation, or preprocessing times), because it focuses on the algorithmic efficiency of each system. The previous experiment is a baseline comparison, so we used Hydra-IEJoin as its authors originally conceived it. However, Hydra-IEJoin has to map the input into an integer domain, because its implementation is based on integer comparisons. VioFinder does not need this step, and also uses a faster approach to decide predicate order. Thus, to eliminate the additional costs of Hydra-IEJoin, we integrated Hydra-IEJoin's algorithms into VioFinder's platform for this experiment.

Figure 3 shows the runtimes (only querying times) measured for the datasets with an increasing number of rows—mind that some plots have different scales. The plots show SQLServer as the only DBMS approach, over only DCs without range predicates: None of the DBMSs finished execution for DCs with range predicates in less than twenty-four hours or without throwing a memory exception. MonetDB faced the same issue executing functional dependencies, and, in the cases PostgresSQL finished, the observed runtimes were orders of magnitude higher than the other DBMSs. In practice, SQLServer was the fastest among the DBMSs for most DCs and datasets. The DBMS approach scaled better than VioFinder for DC φ_6. The columns in this DC are keys, for which DBMSs are well-optimized. In this case, VioFinder has less opportunity to use its

[3] https://github.com/RoaringBitmap/RoaringBitmap

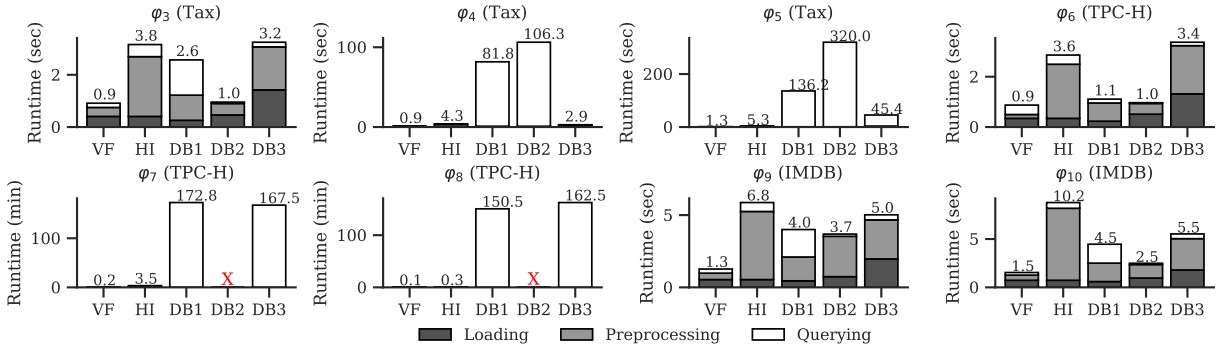

Figure 2: Runtime comparison between VioFinder (VF), Hydra-IEJoin (HI), PostgresSQL (DB1), MonetDB (DB2) and SQLServer (DB3). The datasets are table samples with $200K$ records each.

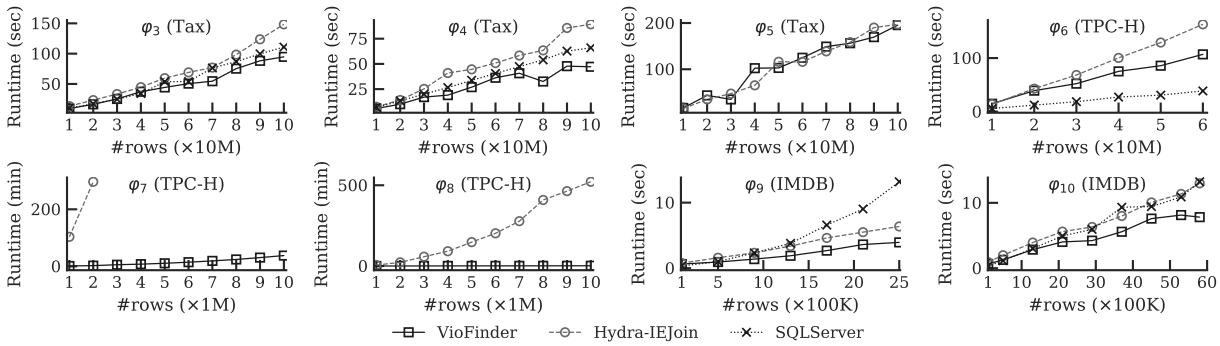

Figure 3: Scalability of VioFinder, Hydra-IEJoin, and SQLServer for an increasing number of rows.

optimizations (e.g., it does not use Algorithm 1). For all other DCs and datasets, VioFinder performs better than the DBMSs solutions.

Although both VioFinder and Hydra-IEJoin show characteristics of linear growth for DCs $\varphi_3, \varphi_4, \varphi_6, \varphi_9$ and φ_{10}, the relative performance difference consistently grows as the number of records grows. Both systems use hash-based approaches with such DCs, but differ in key implementation details. VioFinder deals with multiple equality predicates on single columns at once (with Algorithm 1), whereas Hydra-IEJoin does so one predicate at a time. As a result, Hydra-IEJoin requires larger partitions (with larger cluster pairs) to be moved through the pipeline, which may decrease performance. Moreover, VioFinder uses bitmaps with sorted arrays to implement set operations (e.g., set difference in different than predicates), whereas Hydra-IEJoin uses hash sets. The former approach has been shown to be consistently faster [13].

The performance of VioFinder and Hydra-IEJoin was roughly similar for DC φ_5, but significantly differed for DCs φ_7 and φ_8. For instance, VioFinder was on average 307× faster than Hydra-IEJoin for DC φ_8 on $10M$ rows. Notice that DCs φ_5, φ_7 and φ_8 are those with range predicates. VioFinder uses our proposed sort-merge approaches to process range predicates, whereas Hydra-IEJoin uses the IEJoin algorithm [11]. Both approaches include a phase that builds auxiliary data structures and a phase that uses those data structures to produce the results. The costs of the initial phase in the VioFinder's sort-merge algorithms consist of building cluster indexes and sorting its entries, and the costs to produce results consist of a merge loop that triggers logical operations and bitmap copying. In contrast, the IEJoin algorithm in Hydra-IEJoin evaluates two range predicates in a single pass. The initial costs of the algorithm involve the computing of auxiliary arrays based on sorted versions of column values. As for its second part, the basic idea is to iterate the relative positions of the auxiliary arrays; operate on a bitmap to mark positions of tuples that satisfy the first predicate; then find tuples that also satisfy the second predicate by iterating another auxiliary array and the marked bitmap. The primitives in the second phase of both approaches have a high impact on performance.

We broke down the executions and observed the following. For DC φ_5, the first phase occupied most of the execution time in both approaches; that is, they spent most of the time in sorting. The refinement of the equality predicate of DC φ_5 occupied only a small fraction of the execution time for both approaches. For DCs φ_7 and φ_8, however, both approaches spent most of the time in their second phase. IEJoin has to iterate auxiliary arrays to find and collect qualifying tuples. For DCs with a larger number of violations, as it is the case of DCs φ_7 and φ_8, this primitive is heavily penalized, because many tuples qualify. In contrast, the sort-merge approach builds the results incrementally from previous iterations with the copying of bitmaps. While the approach is also penalized for DCs with a large number of violations, its incremental processing saves plenty of computations and yields lower runtimes.

5.3 Additional evaluation of VioFinder

The next set of experiments focuses on VioFinder. We evaluated the effects that the cache mechanism has on runtime, maximum memory usage, and the number of tuple fetches. We used DC φ_8, because its execution exemplifies how caching can benefit performance. Figure 4 shows the measurements using a cache-disabled version of VioFinder relative to the measurements using the original—the Y-axis is on a log scale. The cache-disabled version has to perform dramatically more tuple fetches and runs considerably slower than its cache-enabled counterpart. The larger the number of tuples in the input, the higher the relative differences in tuple fetches and runtime. Although VioFinder consumed more memory using the cache mechanism for fewer tuples (i.e., less than $400K$), it stably consumed about the same amount of memory for larger inputs. This effect happened because the larger inputs produced clusters with a higher density that took better advantage of bitmap compression.

We evaluated the impact of varying cluster pair thresholds on runtime and maximum memory usage. We observed that performance and memory usage was relatively stable for small thresholds (i.e., less than 100). Partitions with more than one cluster pair benefited the performance of refinements dealing with a few of tuples at-a-time, because there was less interpretation overhead. We used a default threshold of 10, because it is the median value of those thresholds that produced the best runtimes for each DC. However, memory usage increased with larger thresholds as partitions are more likely to store more cluster pairs. Large partitions create long-living data objects in the heap that persist for long portions of the pipeline. This effect degrades runtime, because garbage collection needs to perform additional tracing and marking of long-living objects, consuming additional CPU time. Figure 5 illustrates such behavior for DC φ_8 by showing the memory usage and runtime with increasing tuple pair thresholds relative to these measures with the default threshold.

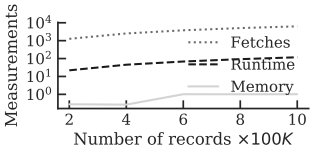

Figure 4: Relative impact of caching cluster indexes on DC φ_8.

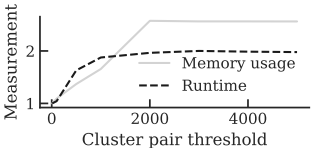

Figure 5: Relative impact of increasing cluster pair thresholds on DC φ_8.

For this last experiment, we measured the size of the in-memory data structures storing the datasets, and the maximum memory used by VioFinder during each execution. Figure 6 shows the results for four DCs—the plots also include the number of violations detected. For most DCs, the contributing factor to the linear increase in memory use is the number of tuples. Notice, however, that DC φ_8 has a huge number of violations. In that case, handling the large intermediates used to produce output consumed much more memory than the in-memory datasets. Nonetheless, these results show that VioFinder is not expensive in terms of memory use.

6 SUMMARY

In this paper, we introduced a DC violation detection system to handle a wide range of data dependencies, from unique constraints

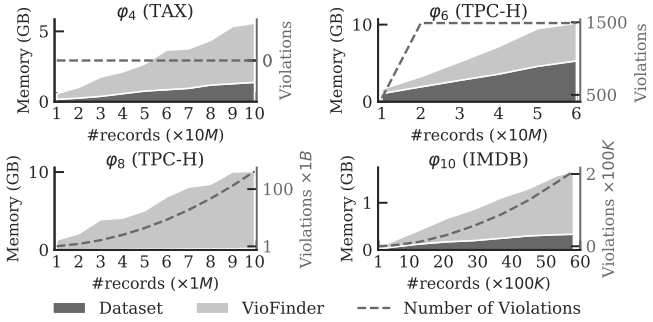

Figure 6: Maximum memory usage.

to other dependencies that express complex relationships between columns. VioFinder shows efficient performance through partition pipelines and effective refinement strategies. Even for larger inputs, or DCs that produce sizeable intermediates and results, our system's performance degrades much more gracefully than the performance of baselines. Our future directions include developing an optimizer for selecting different refinement strategies and leveraging VioFinder in distributed data processing platforms.

REFERENCES

[1] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking Join and Self-Join Sizes in Limited Storage. In *(PODS)*. 10–20.

[2] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *PVLDB* 11, 3 (2017), 311–323.

[3] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. 2005. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *SIGMOD*. 143–154.

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

[5] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*. 541–552.

[6] Wenfei Fan. 2015. Data Quality: From Theory to Practice. *SIGMOD Record* 44, 3 (2015), 7–18.

[7] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional Functional Dependencies for Capturing Data Inconsistencies. *TODS* 33, 2 (2008).

[8] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2014. That's All Folks!: Llunatic Goes Open Source. *PVLDB* (2014), 1565–1568.

[9] Joseph M. Hellerstein and Michael Stonebraker. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. *SIGMOD Rec.* 22, 2 (1993), 267–276.

[10] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD*. 1215–1230.

[11] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *PVLDB* 8, 13 (2015), 2074–2085.

[12] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDB Journal* 27, 5 (2018), 643–668.

[13] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently Faster and Smaller Compressed Bitmaps with Roaring. *Softw. Pract. Exper.* 46, 11 (2016), 1547–1569.

[14] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278.

[15] Davood Rafiei and Fan Deng. 2020. Similarity Join and Similarity Self-Join Size Estimation in a Streaming Environment. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 32, 4 (2020), 768–781.

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