Rapidash: Efficient Detection of Constraint Violations

Zifan Liu
University of Wisconsin-Madison
zifan@cs.wisc.edu

Shaleen Deep
Microsoft
shaleen.deep@microsoft.com

Anna Fariha
University of Utah
afariha@cs.utah.edu

Fotis Psallidas
Microsoft
Fotis.Psallidas@microsoft.com

Ashish Tiwari
Microsoft
ashish.tiwari@microsoft.com

Avrilia Floratou
Microsoft
Avrilia.Floratou@microsoft.com

ABSTRACT

Denial Constraint (DC) is a well-established formalism that captures a wide range of integrity constraints commonly encountered, including candidate keys, functional dependencies, and ordering constraints, among others. Given their significance, there has been considerable research interest in achieving fast detection of DC violations, especially to support activities related to data exploration and preparation. Despite the significant advancements in the field, prior work exhibits notable limitations when confronted with large-scale datasets: the current state-of-the-art algorithm demonstrates a quadratic (worst-case) time and space complexity relative to the dataset's number of rows. In this paper, we establish a connection between orthogonal range search and DC violation detection. We then introduce Rapidash, a novel algorithm that demonstrates near-linear time and space complexity, representing a theoretical improvement over prior work. To validate the effectiveness of our algorithm, we conduct comprehensive evaluations on both open-source and real-world production datasets, with our production datasets notably being an order of magnitude larger than the datasets employed in prior studies. Our results reveal that Rapidash achieves up to 84× faster performance compared to state-of-the-art approaches while also exhibiting superior scalability.

PVLDB Reference Format:
Zifan Liu, Shaleen Deep, Anna Fariha, Fotis Psallidas, Ashish Tiwari, and Avrilia Floratou. Rapidash: Efficient Detection of Constraint Violations. PVLDB, 17(1): XXX-XXX, 2024.

1 INTRODUCTION

Integrity constraints play a pivotal role in a wide range of data analysis tasks such as data exploration [3, 18], data cleaning and repair [21, 38], data synthesis [19], and query optimization [30]. By enforcing integrity constraints, organizations can ensure reliability, consistency, and accuracy of their data, enabling them to make informed decisions, derive meaningful insights, and extract maximum value from their datasets. One class of constraints that is of particular interest is *Denial Constraints* (DCs) [15]. DCs are appealing as they are expressive enough to capture many useful integrity constraints such as functional dependencies, ordering constraints, unique column combinations [4, 40], etc.

Example 1. *Table 1 shows a sample of a tax dataset that contains information about tax rates for people in different US states. The following two rules are true about this dataset:* C_1 : `SSN` *column is a candidate key;* C_2 : `Zip`→`State` *is a functional dependency. However, the rule* C_3 : *"for all people in the same state, if person A has an equal or larger salary than person B, then A should have an equal or larger tax rate than B"—is not true since the tuples* t_2 *and* t_3 *have a higher value for* `Salary` *than the tuple* t_4 *but have tax rates lower than 22% (15% and 20%, respectively). Each of these rules can be expressed as DCs as we will see in Section 2.*

Table 1: Tax rates for people in different states in the USA.

	SSN	Zip	Salary	FedTaxRate	State	StateCode
t_1	100	10108	3000	20%	New York	01
t_2	101	53703	5000	15%	Wisconsin	02
t_3	102	53703	6000	20%	Wisconsin	02
t_4	103	53703	4000	22%	Wisconsin	02

We focus on the detection of DC violations on a given dataset. The process involves verifying whether a given DC is satisfied on a specific dataset (we refer to this task as DC verification in Section 2), and is particularly valuable during data exploration, where analysts aim to quickly ascertain the presence or absence of specific patterns within the dataset. For example, a data analyst, while exploring a dataset for the first time, might want to quickly detect which of the constraints (rules) presented in Example 1 hold on the dataset. Additionally, in case a constraint does not hold on the dataset, the analyst might want to enumerate a few tuples that *violate* the constraint (see violation enumeration in Section 2). In Example 1, there are two violations with respect to C_3: (t_2, t_4) and (t_3, t_4). These violations can be fed into upstream data cleaning tools [17, 20, 38], which can be used to repair data inconsistencies. Thus, DC violation detection serves as a valuable tool in assessing dataset quality [18].

Limitations of existing work. In recent years, substantial advancements have happened in the field of DC violation detection [35, 36]. However, our practical experience in applying some of these approaches to real-world production datasets has unveiled noteworthy limitations of existing methods (refer to Section 5 for comprehensive details). In particular, the best-known algorithm, Facet [35], has a worst-case time and space complexity $\Omega(|\mathbf{R}|^2)$ on a given relation $\mathbf{R}$ with cardinality $|\mathbf{R}|$ (number of rows), which makes it infeasible for large-scale datasets.

Our approach. In this work, we make the connection between the problem of DC violation detection and *orthogonal range search* [10, 11]. Given a set of multidimensional points, orthogonal range search allows efficient searching of all points that lie within an axis-aligned, multidimensional rectangle. The key insight of our work is that

range search can be utilized by modeling the tuples in the relation as multidimensional points. We illustrate this with an example.

Example 2. *Consider the rule C_3 from Example 1 and the three tuples with* `State=Wisconsin` *(t_2, t_3, and t_4) from Table 1. The three tuples can be visualized as two-dimensional points when considering the columns* `Salary` *and* `FedTaxRate` *(Figure 1a), which are of interest for rule C_3. Let us fix our attention on tuple t_4 that has* `Salary` $= 4000$ *and* `FedTaxRate` $= 22$. *According to C_3, all tuples that have* `Salary` ≥ 4000, *must also have a* `FedTaxRate` ≥ 22. *In other words, if we are able to find a tuple that has* `Salary` ≥ 4000 *but a strictly smaller tax rate than 22, we will have found a pair that violates the rule C_3. The shaded rectangular area in Figure 1b encodes precisely this criteria: the shaded rectangular area represents the region with* `Salary` ≥ 4000 *but* `FedTaxRate` < 22. *In our example, both t_2 and t_3 lie in the shaded region and thus violate C_3 with respect to t_4.*

To leverage this observation, we would need to employ a data structure that allows fast searching of points for any given multidimensional axis-parallel rectangle. This is precisely the orthogonal range search problem [10, 11] for which there exist practical data structures. Making the connection between DC violation detection and orthogonal range search opens up new possibilities in addressing this challenge, potentially surpassing the capabilities of state-of-the-art methods in violation detection.

In this work, we take the first step in exploring the relationship between these two domains. We demonstrate that there are various challenges in realizing our vision. First, taking an arbitrary DC (formally defined in Section 2) and translating it into an instance of the orthogonal range search problem is not trivial. The complexity arises from having to support constraints with complex structures. Our work is the first one to show how different classes of denial constraints can be mapped into instances of the orthogonal range search problem. Furthermore, we show that directly applying vanilla range search data structures (i.e. without any of the optimizations proposed in this work) leads to poor performance. Figure 2 shows the total running time of verifying four practical DCs from [37] on the TPC-H dataset that has been used by several prior works related to DCs[1]. We find that using vanilla range search is 30× worse than Facet (the state-of-the-art) and 100× worse than Rapidash (our system).

In light of these issues, a principled study to design algorithms and optimizations that translate theoretical models for orthogonal range search into tangible performance improvements is imperative. In this work, we propose novel optimizations that provably reduce the time complexity over vanilla range search.

Our contributions. Our key contributions are the following:

(1) Making the connection between orthogonal range search and violation detection and taking the first step in exploring the interplay between these two domains.
(2) *A novel DC violation detection algorithm.* We perform a fine-grained classification of DCs and show how they can be mapped to the orthogonal range search problem. We present a near-optimal algorithm Rapidash for verifying a given DC on a dataset **R** by leveraging foundational research in the domain of orthogonal range search [10, 32, 42]. Our proposed algorithm has a time complexity of $O(|\mathbf{R}| \log^{f(\varphi)} |\mathbf{R}|)$, where $f(\varphi)$ is a parameter that is dependent only on the characteristics of the DC φ and not on the input dataset **R**. This represents a significant improvement over the best-known verification algorithm [35], which has a worst-case quadratic complexity (both time and space). We also show that in certain scenarios, Rapidash can run in linear space while still achieving provably sub-quadratic running time.
(3) *Experimental evaluation.* We conduct an extensive empirical evaluation over both open-source and production datasets, notably the latter being an order of magnitude larger than the datasets employed in prior studies. We show that Rapidash achieves up to 84× speedup over the state-of-the-art [35] for DC violation detection. Additionally, our findings show that Rapidash exhibits better scalability over previous work.

[1]See Section 5 for a more detailed analysis of the constraints.

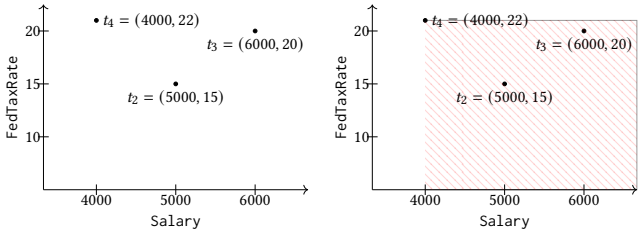


Figure 1: Geometric representation of the data for Example 2. (a) `Salary` and `FedTaxRate` for tuples with `State = Wisconsin` in table `Tax`. (b) The shaded area contains the tuples violating C_3 with `Salary` $\geq t_4$.`Salary` and `FedTaxRate` $< t_4$.`FedTaxRate`.

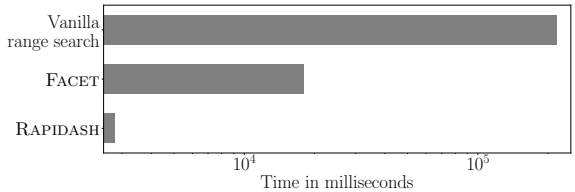


Figure 2: Total time for verification of four DCs on TPC-H dataset (see Table 3) using different approaches.

2 BACKGROUND AND PROBLEM STATEMENT

Relations. Let **R** be the input relation and vars(**R**) denote the finite set of attributes (i.e. the columns). We use $|\mathbf{R}|$ to denote the cardinality (number of tuples) of **R**. We will use A, B to denote attributes, s and t to denote tuples, and t.A to denote the value of attribute A of tuple t. Throughout the paper, we assume bag semantics where the relation can have the same tuple present multiple times.

Denial Constraints (DCs). DCs express predicate conjunctions to determine conflicting combinations of column values. They generalize other integrity constraints, including unique column combinations, functional dependencies, and order dependencies. We define a predicate p as the expression s.A op t.B where $s, t \in \mathbf{R}$, op $\in \{=, \neq, \geq, >, \leq, <\}$ and A, B $\in$ vars(**R**). We will refer to $\neq$ as disequality and $\geq, >, \leq, <$ as inequalities. All operators except equality

will be collectively referred to as non-equality operators. A DC φ is a conjunction of predicates of the following form:

$$\forall s, t \in R, s \neq t: \quad \neg(p_1 \wedge \cdots \wedge p_m)$$

A tuple pair (s, t) is said to be a violation if all predicates in φ evaluate to true. We will say that φ holds on $\mathbf{R}$ if there is no violation, i.e., the DC is *exact*. An exact DC is said to be minimal if no proper subset of its predicates forms another exact DC. A predicate is said to be *homogeneous* if it is of the form $s.\mathsf{A}$ op $t.\mathsf{A}$ or $s.\mathsf{A}$ op $s.\mathsf{B}$, i.e. it is either defined over a single column A or it is defined over a single tuple s, but two different columns; and *heterogeneous* if it is of the form $s.\mathsf{A}$ op $t.\mathsf{B}$. We will refer to $s.\mathsf{A}$ op $t.\mathsf{A}$ as *row-level* homogeneous predicate since it is comparing across two rows.

Since most DCs of interest contain only row-level homogeneous predicates (such as ordering dependencies [40], functional dependencies, candidate keys, etc.), for simplicity, we will use the term homogeneous DC to refer a DC that contains only row-level homogeneous predicates. A heterogeneous DC can contain all types of predicates. Without loss of generality, we will assume that each column of $\mathbf{R}$ participates in at most one predicate of a homogeneous DC. We will use $\mathsf{vars}_{\mathsf{op}}(\varphi)$ to denote the set of columns in a homogeneous DC that appear in some predicate with the operator op. A DC may also contain *column-level* homogeneous predicates of the form $s.\mathsf{A}$ op $s.\mathsf{B}$. However, in the interest of space, we do not discuss them in this paper given that they are not very common in practice[2] and we defer the interested reader to [2].

Example 3. *Continuing from Example 1, each constraint can be expressed using a DC as follows: (1)* $\varphi_1 : \neg(s.\mathtt{SSN} = t.\mathtt{SSN})$*; (2)* $\varphi_2 : \neg(s.\mathtt{Zip} = t.\mathtt{Zip} \wedge s.\mathtt{State} \neq t.\mathtt{State})$*; (3)* $\varphi_3 : \neg(s.\mathtt{State} = t.\mathtt{State} \wedge s.\mathtt{Salary} \leq t.\mathtt{Salary} \wedge s.\mathtt{FedTaxRate} > t.\mathtt{FedTaxRate})$*. The universal quantification is left implicit. Let us fix our attention to* φ_3*. Note that* $vars_{=}(\varphi_3) = \{\mathtt{State}\}$, $vars_{\leq}(\varphi_3) = \{\mathtt{Salary}\}$*, and* $vars_{>}(\varphi_3) = \{\mathtt{FedTaxRate}\}$*. All the DCs except* φ_3 *hold on the relation* Tax *defined in Table 1 and are minimal exact DCs.*

We are now ready to state the problem considered in the paper.

Problem Statement. Given a relation $\mathbf{R}$ and a DC φ, determine:
(1) whether φ holds on $\mathbf{R}$ (DC verification).
(2) enumerate all the tuple pairs that violate the constraint (violation enumeration).

All complexity results in this paper are based on the standard RAM model [24] of computation.

3 LIMITATIONS OF EXISTING SOLUTIONS

We now discuss the limitations of Facet for the problems we aim to address. In Section 5, we experimentally demonstrate some of these limitations using real-world datasets.

We begin by giving a brief description of the key ideas underlying Facet [35], the state-of-the-art system for DC violation detection. Let tids denote a set of tuple identifiers. All tuples in relation $\mathbf{R}$ can be represented as $\mathsf{tids}_{\mathbf{R}} = \{t_1, \ldots, t_{|\mathbf{R}|}\}$. An ordered pair $(\mathsf{tids}_1, \mathsf{tids}_2)$ represents all tuple pairs (s, t) such that $s \in \mathsf{tids}_1, t \in \mathsf{tids}_2, s \neq t$. Facet processes one predicate of the DC at a time, taking a set of ordered pairs $(\mathsf{tids}_1, \mathsf{tids}_2)$ as input and generating another set of ordered pairs $(\mathsf{tids}'_1, \mathsf{tids}'_2)$ that represent tuple pairs that satisfy the predicate as the output. This process is known as *refinement* and Facet refines each predicate using specialized algorithms for each operator. The output of a refinement is consumed as the input for refining the next predicate. At the end of processing all the predicates, we get all tuple pairs that satisfy all the predicates, and, thus, represent the violations.

Example 4. *Consider the DC* $\varphi_3 : \neg(s.\mathtt{State} = t.\mathtt{State} \wedge s.\mathtt{Salary} \leq t.\mathtt{Salary} \wedge s.\mathtt{FedTaxRate} > t.\mathtt{FedTaxRate})$*. The refinement of predicate* $p_1 : s.\mathtt{State} = t.\mathtt{State}$ *produces the set* $\{(\{t_2, t_3, t_4\}, \{t_2, t_3, t_4\})\}$ *with a single ordered pair. This ordered pair represents the set of tuple pairs:* (t_2, t_3), (t_2, t_4), (t_3, t_2), (t_3, t_4), (t_4, t_2), (t_4, t_3) *since each of them satisfy* p_1*. Next, this singleton set is provided as input to predicate* $p_2 : s.\mathtt{Salary} < t.\mathtt{Salary}$ *which produces a new set* $\{(\{t_4\}, \{t_2, t_3\}), (\{t_2\}, \{t_3\})\}$ *since the* Salary *for* t_4 *is smaller than both* t_2 *and* t_3 *but* Salary *for* t_2 *is smaller than* t_3 *only. Finally, we process predicate* $p_3 : (s.\mathtt{FedTaxRate} > t.\mathtt{FedTaxRate})$*. The output of the refinement by predicate* p_3 *would be* $\{(\{t_4\}, \{t_2, t_3\})\}$ *which represents the two violations of* φ_3*:* (t_4, t_2) *and* (t_4, t_3)*.*

Facet contains algorithms that are custom-designed for the different predicate structures. We now highlight the three key sources of inefficiency in Facet:

(1) **Complexity of IEJoin.** Facet uses IEJoin [29] as the algorithm for processing inequalities. The algorithm is designed to process two inequalities at a time and operates on two sets of tuple pairs simultaneously. The runtime complexity of IEJoin is $O(|R| \cdot |S|)$ for processing inequality joins between two relations R and S (although its space complexity is only $O(|R| + |S|)$). As noted in [35], IEJoin is severely under-performing for predicates of low selectivity.

(2) **Complexity of Hash-Sort-Merge.** Since IEJoin is designed for at least two predicates with inequality, Facet proposed two novel optimizations to process DCs with a single inequality predicate: **`Hash-Sort-Merge (HSM)`** and **`Binning-Hash-Sort-Merge (BHSM)`**. However, it can be shown that both HSM and BHSM still require a quadratic amount of running time and space in the worst-case. Similarly, processing of predicates containing disequality also requires quadratic time and space in the worst-case. In the experimental evaluation, we will demonstrate that this is not just a theoretical argument but manifests itself in reality even for constraints with as few as two predicates.

(3) Since Facet processes one predicate at a time, it needs to make at least one full pass over the dataset. As we will see later, this is not always necessary for DC verification. Even for enumeration, we will show there exists efficient ways to process the constraints.

4 RAPIDASH DESIGN AND ANALYSIS

In this section, we describe the general ideas underlying Rapidash followed by specific improvements and optimizations. Our algorithm builds appropriate data structures to store the input data (leveraging existing work on orthogonal range search), and issues appropriate queries to detect violations of a given DC. For ease of exposition and aiding readability, we will focus on the problem of DC verification throughout this section (item (1) in our problem statement in Section 2). At the end of the section, we point out how our main algorithm can be readily modified in a minimal way to

[2]None of the experimental evaluations in [34–37] contain a DC with column-level homogeneous predicates.

Algorithm 1: DC verification for homogeneous constraints with equality predicates

Input : Relation $\mathbf{R}$, Homogeneous DC φ with only equality predicates.
Output: True/False

$H \leftarrow$ empty hash table
foreach $t \in \mathbf{R}$ **do**
  /* Project tuple t on the columns participating in equality predicates */
  $v \leftarrow \pi_{\text{vars}_=(\varphi)}(t)$
  if $v \notin H$ **then**
    $H[v] \leftarrow 0$
  $H[v] \leftarrow H[v] + 1$
  if $H[v] > 1$ **then**
    /* Hash collision – violation detected */
    return False
return True

support violation enumeration (item (2) in the problem statement). We defer a detailed description to the Appendix in [2].

4.1 Foundation

Before we delve into the details of our proposed algorithm and its relationship to orthogonal range search, we will provide some intuition behind the design of the algorithm taking as an example the simplest scenario: homogeneous constraints with equality predicates only. The constraint $\varphi_1 : \neg(s.\texttt{SSN} = t.\texttt{SSN})$ presented in the previous section is a qualifying constraint. The verification algorithm should return `True` over a relation $\mathbf{R}$, when `SSN` is a candidate key. If at least one pair of tuples in $\mathbf{R}$ shares the same `SSN` value, then the algorithm should return `False`.

To evaluate whether such a tuple exists, we can incrementally populate a hash table tuple-by-tuple. The intuition is that if two tuples fall in the same hash partition then they have the same `SSN` value, and thus, we have identified a violation. The steps are presented in Algorithm 1. For each new tuple, we extract the `SSN` value and check whether it already exists in the hash table. If not, then we create a new entry with an associated count of 1. If there is already an entry then we increase the count by 1. If the count becomes greater than 1, we have identified two tuples with same `SSN` value and we return `False`. For example, in Table 1, the algorithm would create 4 hash table entries (one per `SSN` value), each with count 1, and thus, the constraint would evaluate to `True`. The algorithm works similarly with constraints having more than one equality predicates. This algorithm is straightforward and easy to understand, yet it grows more complex with the inclusion of non-equality predicates. It is at this juncture that orthogonal range search becomes relevant. Before we dive deeper into these scenarios, we will first provide some background on orthogonal range search.

4.2 Orthogonal Range Search

In this section, we present some background on orthogonal range search [10, 11]. Given a totally ordered domain $\mathbb{N}$, let $A \subseteq \mathbb{N}^k$, for some $k \geq 1$, be a set of size N. Let $\mathbf{L} = (\ell_1, \ldots, \ell_k)$ and $\mathbf{U} = (u_1, \ldots, u_k)$ be such that $\mathbf{L}, \mathbf{U} \in \mathbb{N}^k$ and $\ell_i \leq u_i$ for all $i \in [k]$.

Algorithm 2: DC verification for homogeneous constraints with inequality predicates

Input : Relation $\mathbf{R}$, Homogeneous DC φ with inequalities.
Output: True/False

$k \leftarrow$ #number of columns appearing in inequality predicates
$H \leftarrow$ empty hash table
foreach $t \in \mathbf{R}$ **do**
  /* Project tuple t on the columns participating in equality predicates */
  $v \leftarrow \pi_{\text{vars}_=(\varphi)}(t)$
  if $v \notin H$ **then**
    $H[v] \leftarrow$ new OrthogonalRangeSearchTree(k)
  /* Project tuple t on the columns participating in inequality predicates */
  $z \leftarrow \pi_{\text{vars}(\varphi) \setminus \text{vars}_=(\varphi)}(t)$
  if $\neg H[v]$.isEmpty() **then**
    /* Evaluate violations through two range search queries */
    $\mathbf{L}, \mathbf{U}, \mathbf{L}', \mathbf{U}' \leftarrow$ CreateRangeSearchQueries(z, ϕ)
    if $H[v]$.booleanRangeSearch($\mathbf{L}, \mathbf{U}$) $\vee$ $H[v]$.booleanRangeSearch($\mathbf{L}', \mathbf{U}'$) **then**
      /* Violation detected */
      return False
  /* Insert z into the range tree */
  $H[v]$.insert(z)
return True
procedure CreateRangeSearchQueries(t, φ)
  /* $\mathbf{L}$ and $\mathbf{U}$ are indexed by the non-equality predicates p_i. Both are of size k */
  $\mathbf{L} \leftarrow (-\infty, \ldots, -\infty), \mathbf{U} \leftarrow (\infty, \ldots, \infty)$
  /* Create range search query ($\mathbf{L}$, $\mathbf{U}$) */
  foreach *inequality predicate* $p_i \in \varphi$ **do**
    if p_i.op *is* $<$ *or* $\leq$ **then**
      $\mathbf{U}_i \leftarrow \pi_{p_i.\text{col}}(t)$
    if p_i.op *is* $>$ *or* $\geq$ **then**
      $\mathbf{L}_i \leftarrow \pi_{p_i.\text{col}}(t)$
  /* Create inverted range search query ($\mathbf{L}'$, $\mathbf{U}'$) */
  $\mathbf{U}' \leftarrow \mathbf{L}, \mathbf{L}' \leftarrow \mathbf{U}$
  flip $-\infty$ to ∞ and ∞ to $-\infty$ in $\mathbf{U}'$ and $\mathbf{L}'$ respectively
  return $\mathbf{L}, \mathbf{U}, \mathbf{L}', \mathbf{U}'$

Definition 1. *An orthogonal range search query is denoted by* $(\mathbf{L}, \mathbf{U})$*, and its evaluation over* A *consists of enumerating the set*

$$Q(A) = \{a \in A \mid \bigwedge_{i \in [k]} \ell_i \ \text{op}_1 \ a_i \ \text{op}_2 \ u_i\}$$

where $\text{op}_1, \text{op}_2 \in \{=, <, \leq\}$.

In other words, $\mathbf{L}$ and $\mathbf{U}$ form an axis-aligned hypercube in k dimensions, and $Q(A)$ reports all points in A that lie on/within the hypercube. The Boolean version of the orthogonal range search problem consists of determining if $Q(A)$ is empty or not.

Example 5. *Consider the Table* `Tax` *from Example 1. Let* A *be the set of two-dimensional points obtained by projecting* `Tax` *on* `Salary` *and* `FedTaxRate`*. Let* $\mathbf{L} = (3500, 5)$ *and* $\mathbf{U} = (4500, 25)$*. Then, the orthogonal range query* $(\mathbf{L}, \mathbf{U})$ *is asking for all points such that the* `Salary` *is between* 3500 *and* 4500*, and the* `FedTaxRate` *is between* 5 *and* 25*. In Table* `Tax`*, only* t_4 *satisfies the criteria (its values of* `Salary`

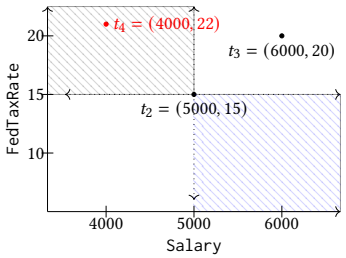


Figure 3: `Salary` and `FedTaxRate` for each tuple in `Tax`. The grey (upper left quadrant centered at t_2) and blue shaded areas (lower right quadrant centered at t_2) show the regions where the tuples that could form a violation with t_2 lie.

and `FedTaxRate` *are* 4000 *and* 22 *respectively). Thus, the result of the orthogonal range search query* $(\mathbf{L}, \mathbf{U})$ *is* $\{(4000, 22)\}$.

The two most celebrated data structures for orthogonal range search are range trees and kd-trees [10]. We will review their complexity and trade-offs when analyzing the complexity of our specific algorithm.

4.3 Verification Algorithm

In this section, we present our verification algorithm that leverages prior work on orthogonal range search. The algorithm builds on top of the ideas behind Algorithm 1 but extends them to cover for homogeneous constraints that contain both equalities and inequalities. Without loss of generality, we will assume that none of the predicates contain disequality. This assumption will be removed later. Finally, we assume that the categorical columns in $\mathbf{R}$ have been dictionary-encoded to integers, a standard assumption in line with prior work [34, 36].

The algorithm preserves the core concepts of Algorithm 1, namely the use of a hash table and early termination in case of violations. The primary modification involves incorporating orthogonal range search indexes to identify violations stemming from inequality predicates. The algorithm is presented in Algorithm 2.

We first compute the number k of columns in inequality predicates (we assume $k > 0$ as Algorithm 1 covers the case where $k = 0$). We then proceed similarly as before: project each tuple on the columns participating in equality predicates (v on line 4) and evaluate whether the resulting tuple has been seen before (line 5). If not, we create a new hash table entry whose value now, instead of being an integer counter, is a range search tree of k dimensions (line 6). This tree will be used to index the k-dimensional tuples containing the columns participating in inequality predicates and identify violations. Before we further delve into the pseudocode, we explain the main intuition through an example.

Example 6. *Consider the relation* `Tax` *from Example 1 and the DC* $\phi_3 : \neg(s.\texttt{State} = t.\texttt{State} \wedge s.\texttt{Salary} \leq t.\texttt{Salary} \wedge s.\texttt{FedTaxRate} > t.\texttt{FedTaxRate})$*, which contains one equality and two inequality predicates ($k = 2$). For simplicity, we will omit the details of how range search works in this example but instead present it later. Algorithm 2 will first start with the equality predicate, and place t_1 in a hash partition by hashing t_1.`State` = New York and initiatialize a 2-dimensional range search tree for that partition (line 6). Since this tree is empty, we do not perform any violation detection (line 8) and we insert* $(t_1.\texttt{Salary}, t_1.\texttt{FedTaxRate}) = (3000, 20)$ *in the tree (line 12).*

Next, we process t_2, which is placed in a different hash partition since t_2.`State` = Wisconsin and we initialize a new range tree for this partition. As in the previous step, we then insert $(t_1.\texttt{Salary}, t_1.\texttt{FedTaxRate}) = (5000, 15)$ *in the tree (call this step Ⓐ). When t_3 is processed, it is placed in the same partition as t_2 since they have the same* `State` *value.*

At this point, we have two tuples in the same hash partition, and thus we need to consider the inequality predicates in the DC to establish whether there is a DC violation. This is where the orthogonal range search tree is leveraged. Such a violation would occur in two scenarios: (1) if any tuple in the tree (t_2 in this case) has `Salary` *less than t_3.`Salary` = 6000 but* `FedTaxRate` *more than t_3.`FedTaxRate` = 20, or (2) if any tuple in the tree has* `Salary` *more than t_3.`Salary` but* `FedTaxRate` *less than t_3.`FedTaxRate`.*

To identify whether any of the two scenarios above is true, we perform two orthogonal range search queries using the values of t_3 to probe the index. More specifically, we perform a search with $\mathbf{L} = (-\infty, 20)$ *and* $\mathbf{U} = (6000, \infty)$ *(scenario 1). Then, we also search in the* inverted *range* $\mathbf{L}' = (6000, -\infty)$ *and* $\mathbf{U}' = (\infty, 20)$ *(scenario 2). Since t_2 does not lie in the desired range, both range searches return false. Figure 3 visualizes this result. Since there is no violation, we insert* $t_3(6000, 20)$ *in the tree (call this step Ⓑ)*

Finally, t_4 (highlighted in red in Figure 3) is processed and placed in the same partition as t_2 and t_3 because of same value in `State` *column. We again initiate two range search queries based on t_4* `Salary` = 4000 *and* `FedTaxRate` = 22*. The queries would be* $\mathbf{L} = (-\infty, 22), \mathbf{U} = (4000, \infty)$ *and* $\mathbf{L}' = (4000, -\infty), \mathbf{U}' = (\infty, 22)$*. Then, t_2 and t_3 form a violation wrt. t_4 since both the points represent a higher salary than* 4000 *but a lower tax rate than* 22*, and Line 11 returns False.*

Table 2: Data structure parameter on input of size n [32]. k is the number of dimensions of the points inserted in the tree.

DS	Insertion $I(n)$	Answering $T(n)$	Space $S(n)$
Range tree	$O(\log^k n)$	$O(\log^k n)$	$O(n \cdot \log^{k-1} n)$
kd-tree	$O(\log n)$	$O(n^{1-\frac{1}{k}})$	$O(n)$

Algorithm 2 formalizes the process described in the example above. When it comes to evaluating inequality predicates, we generate two appropriate range queries based on the values of the current tuple and the operator type in the inequality predicates (line 9) and then search the range tree (line 10). If any of the range search queries returns `True`, a DC violation is detected and the algorithm terminates (line 11). Otherwise, the tuple is inserted in the tree (line 12), and the algorithm continues with the next tuple in the relation.

Seminal work by Overmars [32] showed that using range trees and kd-trees, one can design an algorithm with the parameters as shown in Table 2. We now demonstrate an example of how range trees are used by Algorithm 2 when performing step Ⓐ and step Ⓑ in Example 6.

Example 7. *In Figure 4, we illustrate insertion and search in a range tree. The tree is two-dimensional and stores (*`Salary`, `FedTaxRate`*) points. We show the process of inserting t_2, t_3 (i.e. performing step Ⓐ*

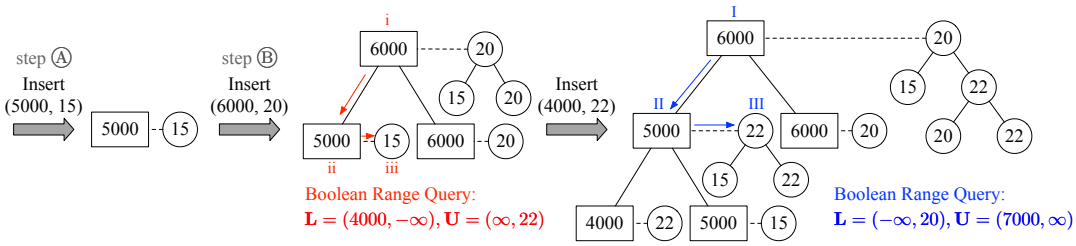


Figure 4: An illustration of insertion and search in a 2D range tree that stores (`Salary`, `FedTaxRate`). The primary tree with rectangle nodes stores `Salary`, and the secondary trees with circle nodes store `FedTaxRate`. The Roman numbers denote the visiting order during the search.

and step Ⓑ from Example 6) and a range search for points whose `Salary` ≥ 4000 *and* `FedTaxRate` < 22. *In the range tree, both the primary tree (which stores* `Salary`*) and the secondary tree (which stores* `FedTaxRate`*) are binary search trees. Leaf nodes store the inserted data, and each internal node stores the smallest value in its right subtree. Each node in the primary tree is linked to a secondary tree, which stores the* `FedTaxRate` *value of all the points present in the subtree rooted at this node. When we insert each point, we find the insert position in the primary tree, create a leaf node to store the inserted value, and an internal node to connect the new leaf node and the leaf node at the insert position. We also update all the secondary trees for nodes in the path from the root to the insert position. When we perform the range search, we look for nodes whose values lie in the range by preorder traversal. For instance, the search* $\mathbf{L} = (4000, -\infty), \mathbf{U} = (\infty, 22)$ *is performed by going from root node i to the left child node ii. Since the* `Salary` *value stored in node ii is in the range, we go to the secondary tree linked to it. Finally, we find node iii whose* `FedTaxRate` *value is less than* 22 *and return true for the query.*

As another example, suppose we also insert t_4 *in the tree and search for a point where* `Salary` ≤ 7000 *and* `FedTaxRate` > 20. *At node II, we go to its secondary tree directly instead of traversing its descendants since all the points stored in the subtree rooted at node II have* `Salary` ≤ 5000, *which are within the search range for* `Salary`. *At node III, we return true since the minimum* `FedTaxRate` *stored in its right subtree is* 22, *which is greater than* 20.

Due to space limitation, we establish the correctness of Algorithm 2 and show the analysis of time/space complexity in our tech report [2]. We state our main result as follows.

Theorem 1. *Algorithm 2 runs in time* $O(|\mathbf{R}| \cdot (I(|\mathbf{R}|) + T(|\mathbf{R}|)))$ *and uses space* $S(|\mathbf{R}|)$ *when using range tree or kd-tree with parameters as shown in Table 2.*

With range trees, the running time is $O(|\mathbf{R}| \cdot \log^k |\mathbf{R}|)$ and space usage is $O(|\mathbf{R}| \log^{k-1} |\mathbf{R}|)$; for kd-trees, the running time is $O(|\mathbf{R}|^{2-\frac{1}{k}})$ and space requirement is $O(|\mathbf{R}|)$.

Comparison with Facet. Our approach is superior to Facet in three respects. First, we use polynomially less space and time in the worst-case. Second, there exist instances where our algorithm saves a significant amount of time and space by early termination[3].

[3]In this paper, we use the term early termination to mean that the algorithm can avoid scanning the entire relation for some instances.

Proposition 1. *For every homogeneous DC* φ *with at least one non-equality predicate, there exists a relation* $\mathbf{R}$ *such that Algorithm 2 takes* $O(1)$ *time and Facet requires* $\Omega(|\mathbf{R}|)$ *time.*

Although Proposition 1 may seem like an obscure theoretical argument, the behavior is commonly observed in the real world. We empirically demonstrate this in our experiments. The third aspect concerns the space usage. It turns out that the space requirement of Facet is relation-dependent, with no way of predicting how much memory will actually be needed, an artifact of the lack of nontrivial provable bounds on its time and space requirement. If the machine has only limited amount of memory, Facet will be unable to complete the refinements and fail. On the other hand, our solution allows verification with linear space using kd-trees. This flexibility is important for resource-constrained production scenarios.

4.4 Heterogeneous Predicates

In this section, we present adjustments to Algorithm 2 to handle heterogeneous predicates.

Example 8. *Continuing our study of* `Tax` *from Table 1, consider the following constraint:* $\psi_2 : \neg(s.\texttt{Salary} < t.\texttt{FedTaxRate})$ *which says that all values of the* `Salary` *column must be greater than or equal to any value of* `FedTaxRate` *column.*

This is an example of a heterogeneous predicate that cannot be handled by Algorithm 2 (notice that both s and t are referenced and the columns are different). Along the same lines, heterogeneous comparison constraints over date-related columns have also been found to be useful in our production settings. For example, over the `TPC-H` schema, [37] identified the heterogeneous constraint $\neg(s.\texttt{Receiptdate} \geq t.\texttt{Shipdate} \wedge s.\texttt{Shipdate} \leq t.\texttt{Receiptdate})$ which represents the business logic that a new order is shipped only after all the previously shipped orders are received, i.e., the [`Shipdate`, `Receiptdate`] intervals of the orders never overlap. These real-world scenarios underscore the need for going beyond simple homogeneous constraints considered in the previous section.

We now discuss how our algorithm can be extended to handle heterogeneous predicates of the form $s.\mathsf{A}$ op $t.\mathsf{B}$, where op is $=, <, \leq, >$ or $\geq$. We note that an equality predicate $s.\mathsf{A} = t.\mathsf{B}$ is equivalent to $s.\mathsf{A} \leq t.\mathsf{B} \wedge s.\mathsf{A} \geq t.\mathsf{B}$ and by supporting inequalities, we can support heterogeneous equality predicates as well.

Now, let's look at how to handle heterogeneous inequality predicates. Suppose that φ has a predicate $s.C < t.D$. If $C = D$, then

Algorithm 3: Generalized range search query generation that covers both homogeneous and heterogeneous predicates with inequalities

Input : A tuple r from relation $\mathbf{R}$, DC φ
Output: Two range search queries (normal and inverted)

```
1  procedure CreateRangeSearchQueries(r, φ)
2      L, L' ← (−∞, ..., −∞), U, U' ← (∞, ..., ∞)   /* L, U, L', U' are indexed by attributes of R that appear in inequality predicates */
3      foreach inequality predicate s.C op t.D in φ do
4          if op is < or ≤ then
5              U.C ← r.D
6              L'.D ← r.C
7          if op is > or ≥ then
8              L.C ← r.D
9              U'.D ← r.C
10     return L, U, L', U'
```

the predicate is homogeneous and building a 1-dimensional range search data structure is enough. However, when $C \neq D$, we need to index in 2 dimensions (C and D). Additionally, we need to adjust our procedure for computing the two search range queries $(\mathbf{L}, \mathbf{U})$, $(\mathbf{L}', \mathbf{U}')$ to consider the different attributes present in the predicate. Let us look at an example.

Example 9. *Consider the DC ψ_2 from Example 8 that checks that all the* Salary *values must be greater than any* FedTaxRate *value. We will create one 2-dimensional range search data structures in which we will store value of (*Salary, FedTaxRate*). Suppose we are processing tuple t_2 with* Salary = 5000 *and* FedTaxRate = 15. *We first do a range search to check if there is a tuple with* Salary *that is less than t_2.*FedTaxRate *denoted as* $\mathbf{L} = (-\infty, -\infty), \mathbf{U} = (15, \infty)$. *Additionally, we check whether there is a tuple with* FedTaxRate *that is larger than t_2.*Salary *denoted as* $\mathbf{L}' = (-\infty, 5000), \mathbf{U}' = (\infty, \infty)$. *If any of the two range search queries return True, we have found a violation.*

Given this example, it becomes clear that we can easily extend Algorithm 2 to account for heterogeneous predicates by simply adjusting the procedure to create the two range search queries that are used to detect violations. Algorithm 3 shows the updated query generation procedure. The main idea is that if φ has a predicate $s.C < t.D$, then when we process a new tuple r, the upper-bound for attribute C is set to $r.D$ in the forward check, and the lower-bound for attribute D is set to $r.C$ in the inverted check (because we are comparing attribute C of s with attribute D of t in the predicate). It is worth noting that this algorithm can be applied for homogeneous constraints as well. Note that, when $C = D$, we recover our original procedure presented in Algorithm 2. Thus, we can safely handle both types of constraints by simply replacing the `CreateRangeSearchQueries` function in Algorithm 2 with the one presented here. Finally, we also note that the new generalization also extends our algorithm to handle the case when attributes occur in more than one predicate.

4.5 Supporting disequalities

So far, we have discussed how to support homogeneous and heterogeneous predicates with equalities or inequalities. In this section, we show that it is possible to apply orthogonal range search techniques to support disequalities as well. This type of operator is quite important as it is required for specifying functional dependencies. We demonstrate that with the below example.

Example 10. *Consider the constraint: $\psi_2 : \neg(s.\texttt{Zip} = t.\texttt{Zip} \wedge s.\texttt{StateCode} \neq t.\texttt{StateCode})$. This DC represents the functional dependency* Zip→StateCode. *The constraint contains a disequality predicate which cannot be handled by any of our previous algorithms.*

We now discuss how we can support verification of such predicates. Any predicate $s.\mathrm{A} \neq t.\mathrm{B}$ can be written as a union of two predicates: $(s.\mathrm{A} < t.\mathrm{B}) \vee (s.\mathrm{A} > t.\mathrm{B})$. Therefore, a DC containing ℓ predicates with op as $\neq$ can be equivalently written as a conjunction of 2^ℓ DCs containing no disequality operator.

If the original homogeneous DC contains no inequality predicate, then it is possible to reduce the number of equivalent DCs from 2^ℓ to $2^{\ell-1}$. The idea is that a violation (s, t) is symmetric (i.e. (t, s) is also a violation) if the DC contains only equality and disequality predicates. Therefore, when converting a DC to only have inequalities, it suffices to expand $(s.A \neq t.A)$ to just $(s.A < t.A)$ for one last disequality predicate instead of $(s.A < t.A) \vee (s.A > t.A)$.

Proposition 2. *Given a homogeneous DC φ with only equality and ℓ disequality predicates, there exists an equivalent conjunction of $2^{\ell-1}$ DCs that contain only equality and inequality predicates.*

4.6 Optimizations

In the previous sections, we have presented how we can support verification of both homogeneous and heterogeneous constraints with equalities, inequalities and disequalities. We now present optimizations for specific type of constraints that can improve the overall performance.

Optimization for a single inequality. If a DC has row homogeneous equality predicates and at most one predicate (homogeneous or heterogeneous) containing an inequality, then the verification can be done in linear time. The key idea is that for predicate containing inequality $s.A$ op $t.B$, it is enough to keep track of the running minimum and maximum values for values seen in columns A and B respectively as we process the relation. To illustrate the idea, we use an example.

Example 11. *Consider the functional dependency ψ_1 from Example 10 and the* Tax *table. Based on proposition (2), we can convert the disequality predicate into inequality to get the DC $\neg(s.\texttt{Zip} = t.\texttt{Zip} \wedge s.\texttt{StateCode} < t.\texttt{StateCode})$. Since both columns in the inequality predicate are the same (*StateCode*), we have $A = B$. Let us focus on rows of* Tax *with* Zipcode=53703. *We will keep track of the* min *and* max *value of* StateCode *for each of these rows. When t_2 is processed, we set* min = max = t_2.Zipcode = 02. *For t_3, we observe that since t_3 also has* StateCode=02, *the predicate t_2.*StateCode < t_3.StateCode *is false and thus no violation is found. The values of* min *and* max *remain unchanged. Finally, for t_4, we also do not find a violation since t_4.*StateCode = 02. *However, if there was a different row t_4' such that t_4'.*Zip = 53703 *and t_4'.*StateCode = 03, *t_2.*StateCode < t_4'.StateCode *would become true and thus, we would have found a violation since all predicates are true for a tuple pair.*

Algorithm 4: DC verification for DCs with row-homogeneous equality and one inequality predicate

Input : Relation $\mathbf{R}$, DC φ containing equality predicates of form $s.\mathrm{C} = t.\mathrm{C}$ (for some C) and **one** inequality predicate p of form $s.\mathrm{A}$ op $t.\mathrm{B}$

Output : True/False

```
H ← empty hash table
foreach t ∈ R do
    v ← π_vars=(φ)(t)
    if v ∉ H then
        H[v] ← (+∞, +∞, −∞, −∞)    /* four-tuple represents
          (min_A, min_B, max_A, max_B) for v */
    if (p.op ∈ {<, ≤} ∧ H[v].min_A op t.B) ∨ (p.op ∈ {>, ≥}
      ∧ H[v].max_A op t.B) then
        return false
    if (p.op ∈ {<, ≤} ∧ t.A op H[v].max_B) ∨ (p.op ∈ {>, ≥}
      ∧ t.A op H[v].min_B) then
        return false
    H[v].min_A ← min{H[v].min_A, t[A]}      /* modify min_A */
    H[v].min_B ← min{H[v].min_B, t[B]}      /* modify min_B */
    H[v].max_A ← max{H[v].max_A, t[A]}      /* modify max_A */
    H[v].max_B ← max{H[v].max_B, t[B]}      /* modify max_B */
return true
```

Algorithm 4 shows the algorithm. Like the previous algorithms, we begin by partitioning the input into a hash table based on the equality predicates. Let the inequality predicate be $s.\mathrm{A}$ op $t.\mathrm{B}$. The main idea is to maintain the running minimum and maximum values for the Columns A and B *for each partition of the input*. Since the comparison is one-dimensional, it is sufficient to compare against the minimum (or maximum) value. The algorithm makes only one pass over the entire dataset and the overall time complexity is $O(|\mathbf{R}|)$. While this optimization is simple, it has important implications. In particular, popular constraints such as functional dependencies (FD) are DCs that contain exactly one inequality predicate. Algorithm 4 recovers the standard linear time algorithm to verify FDs [25]. However, it is unclear that Facet, in its present form, can achieve the same provable guarantee.

4.7 Enumerating violations

In this section, we discuss violation enumeration ((2) in our problem statement). Recall that the Algorithm 2 with minor modifications already supports the enumeration of the violations. In this section, we describe an optimization that improves the time and space complexity of enumerating the DC violations. The first observation we make is that, unlike verification, enumeration usually requires examining every tuple of the relation. Therefore, we can make use of sort-based optimizations. We demonstrate with an example.

Example 12. *Consider* $\varphi_3 : \neg(s.\texttt{State} = t.\texttt{State} \wedge s.\texttt{Salary} \leq t.\texttt{Salary} \wedge s.\texttt{FedTaxRate} > t.\texttt{FedTaxRate})$ *from Example 3 and the rows in table* Tax *with* State=Wisconsin. *The first step is to sort the three rows in increasing order of* Salary, *which creates the ordering* t_4, t_2, t_3. *The key observation here is that once sorting has been performed, it is guaranteed that any tuple can only form a violation (wrt. the* Salary *predicate) with a tuple that appears* after *it in the sort order. At this point, the* Salary *attribute can be completely removed from consideration as the corresponding predicate will always be satisfied. We can now resume our usual processing by constructing a one dimensional binary search tree on* FedTaxRate *attribute. We first insert* t_4.FedTaxRate = 22 *into the tree. For the next tuple in the order* t_2, *which has* t_2.FedTaxRate = 15, *we ask the tree to enumerate all tuples that have value greater than* 15, *a standard operation on a binary search tree. We find the* t_4 *is such a tuple and thus report* (t_4, t_2) *as a violation to the user. Then, we insert* t_2.FedTaxRate *in the tree. Finally,* t_3 *is processed and we search for all values in the tree that with* FedTaxRate *greater than* 20 *and report* (t_4, t_3) *as a violation.*

A detailed description of this optimization and the corresponding enumeration algorithm is presented in [2]. We also have the following complexity result.

Theorem 2. *Let* k *be the number of columns that occur in predicates containing an inequality. Then, our enumeration algorithm runs in time* $O(|\mathbf{R}| \cdot (I(|\mathbf{R}|) + T(|\mathbf{R}|) + \log|\mathbf{R}|) + K)$ *to enumerate* K *violations and uses space* $S(|\mathbf{R}|)$ *when using range tree or kd-tree with parameters as shown in Table 2 with the number of dimensions as* $k-1$.

4.8 Discussion

So far, we saw how we can leverage orthogonal range search to support DC constraint verification and enumeration. Our approach relies on creating classes of constraints (homogeneous with equalities, homogeneous with both equalities and inequalities, heterogeneous) and devising specialized algorithms and optimizations for each class of constraints. An alternative approach would be to create a k-dimensional range search index, where k is the distinct number of column in *all* predicates (as opposed to inequality predicates only in Rapidash). We could then use this index to perform appropriate range search queries and detect violations. We call this approach *vanilla range search* and we experimentally compare it with Rapidash in Section 5.

The main advantage of our techniques over vanilla range search is that they reduce the dimensionality of the range search tree (value of k) which results in significant savings in performance and space. As shown in Table 2, although range search data structures in their most efficient form only add a $(\log|\mathbf{R}|)^k$ term in the time complexity, the factor is multiplicative. Thus, even for relatively small datasets containing $1M \sim 2^{20}$ rows and a DC with $k = 1$ homogeneous predicates (such as an FD), we will have $(\log_2 2^{20})^1 = 20$ as a multiplicative factor which leads to a blowup in both space usage and time in the worst-case. Thus, reducing the value of k whenever possible, can lead to improved performance and allow scaling to larger datasets. This intuition based on the theoretical properties of range search trees is further validated in our experiments.

Our approach categorizes constraints and comes up with approaches to reduce (or even eliminate) k. This is achieved by: 1) leveraging hash tables for homogeneous equality predicates and only pairing them with range trees when inequalities are present, 2) reducing dimensions in the presence of disequalities when possible (see Proposition (2)), completely eliminating range search trees for constraints with singe inequalities (such as FDs) and leveraging running min and max values instead, 3) reducing dimensionality through sorting for violation enumeration.

Table 3: List of denial constraints used in experiments for each dataset.

Dataset	Cardinality	#Columns	DC number	Denial constraint
Tax	1M	12	c_1	$\neg(s.\texttt{AreaCode} = t.\texttt{AreaCode} \wedge s.\texttt{Phone} = t.\texttt{Phone})$
Tax	1M	12	c_2	$\neg(s.\texttt{ZipCode} = t.\texttt{ZipCode} \wedge s.\texttt{City} \neq t.\texttt{City})$
Tax	1M	12	c_3	$\neg(s.\texttt{State} = t.\texttt{State} \wedge s.\texttt{HasChild} = t.\texttt{HasChild} \wedge s.\texttt{ChildExemp} \neq t.\texttt{ChildExemp})$
Tax	1M	12	c_4	$\neg(s.\texttt{State} = t.\texttt{State} \wedge s.\texttt{Salary} > t.\texttt{Salary} \wedge s.\texttt{Rate} < t.\texttt{Rate})$
TPC-H	1M	12	c_5	$\neg(s.\texttt{Customer} = t.\texttt{Supplier} \wedge s.\texttt{Supplier} = t.\texttt{Customer})$
TPC-H	1M	12	c_6	$\neg(s.\texttt{Receiptdate} \geq t.\texttt{Shipdate} \wedge s.\texttt{Shipdate} \leq t.\texttt{Receiptdate})$
TPC-H	1M	12	c_7	$\neg(s.\texttt{ExtPrice} > t.\texttt{ExtPrice} \wedge s.\texttt{Discount} < t.\texttt{Discount})$
TPC-H	1M	12	c_8	$\neg(s.\texttt{Qty} = t.\texttt{Qty} \wedge s.\texttt{Tax} = t.\texttt{Tax} \wedge s.\texttt{ExtPrice} > t.\texttt{ExtPrice} \wedge s.\texttt{Discount} < t.\texttt{Discount})$
NCVoter	1M	67	c_9	$\neg(s.\texttt{countyid} = t.\texttt{countyid} \wedge s.\texttt{countydesc} \neq t.\texttt{countydesc})$
NCVoter	1M	67	c_{10}	$\neg(s.\texttt{ageatyearend} > t.\texttt{birthyear})$
NCVoter	1M	67	c_{11}	$\neg(s.\texttt{statuscd} = t.\texttt{statuscd} \wedge s.\texttt{voterdesc} = t.\texttt{voterdesc} \wedge s.\texttt{reasoncd} \neq t.\texttt{reasoncd})$
NCVoter	1M	67	c_{12}	$\neg(s.\texttt{mailzipcode} = t.\texttt{zipcode} \wedge s.\texttt{statecd} \neq t.\texttt{mailstate})$
D_1	50M	28	$\varphi_{1,1}$	$\neg(s.A = t.A \wedge s.B = t.B \wedge s.C \neq t.C \wedge s.D \neq t.D)$
D_1	50M	28	$\varphi_{1,2}$	$\neg(s.C = t.C \wedge s.E = t.E \wedge s.F = t.F \wedge s.G \neq t.G \wedge s.H \neq t.H)$
D_1	50M	28	$\varphi_{1,3}$	$\neg(s.B = t.B \wedge s.I = t.I \wedge s.J = t.J \wedge s.K \neq t.K \wedge s.L \neq t.L)$
D_1	50M	28	$\varphi_{1,4}$	$\neg(s.A = t.A \wedge s.I = t.I \wedge s.M > t.M \wedge s.N \neq t.N)$
D_2	25M	28	$\varphi_{2,1}$	$\neg(s.A = t.A \wedge s.B = t.B \wedge s.C \geq t.C \wedge s.D \leq t.D \wedge s.E \leq t.E \wedge s.F \geq t.F \wedge s.G > t.G)$
D_2	25M	28	$\varphi_{2,2}$	$\neg(s.A \neq t.A \wedge s.B = t.B \wedge s.H \leq t.H \wedge s.F \geq t.F \wedge s.G \geq t.G)$
D_2	25M	28	$\varphi_{2,3}$	$\neg(s.A = t.A \wedge s.I \neq t.I \wedge s.D \leq t.D \wedge s.G \geq t.G \wedge s.J = t.J)$
D_2	25M	28	$\varphi_{2,4}$	$\neg(s.C \leq t.C \wedge s.D \leq t.D \wedge s.K = t.K)$

Multi-constraint execution. In this work, we focus on single-core and independent processing of DCs (i.e. no work sharing). Before we end the subsection, we briefly comment on the multi-constraint execution capabilities. Facet supports processing multiple DCs at the same time and parallel processing. Our framework could also be expanded to leverage common prefixes of two or more DCs. If the prefixes are equality predicates, the hash table can be shared. For tree-based data structures, the branching of the internal nodes of the tree can be carefully controlled to reuse the nodes that are common to multiple DCs. Similarly, although range search structures can use parallel processing capabilities [12] in theory, their behavior in practice needs further investigation. Further, since Facet performs processing in a *columnar* fashion but our algorithm is *row-oriented*, the impact of the different design choices needs to be explored rigorously.

5 EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental evaluation. In particular, we seek to answer the following questions:

- **(Q.1)** How does Rapidash, FACET and vanilla orthogonal range search compare in terms of performance?
- **(Q.2)** What is the performance improvement (time and space) of the Rapidash verification and enumeration algorithm compared to Facet on open-source datasets used in prior work [35]?
- **(Q.3)** What is the performance and scalability of Rapidash and Facet over large scale real-world production datasets with complex constraints?
- **(Q.4)** What is the impact of varying number of violations on Facet and both variants of Rapidash?

5.1 Experimental Setting

Datasets, DCs, and Hardware. We perform experiments on both open-source datasets and production datasets. For open-source datasets, we use the Tax, TPC-H, and NCVoter, with a total of 12 DCs that were identified by prior works [34–36] to be representative of what is usually seen in production settings (as defined by experts or discovered from data). We also use two production datasets (related to banking records and document shipping) from Microsoft customers interested in applying DC verification on their data. Each dataset contains a mix of categorical, numeric, and datetime columns. For both production datasets, we pick 3 DCs by taking a 10% sample of each dataset and discover DCs that are true over the sample. The fourth DC (denoted by $\varphi_{i,4}$ for dataset D_i) holds over the full dataset. Table 3 lists a total of 20 DCs over all datasets that we use in our experiments[4]. Note that constraints c_5, c_6, c_{10}, c_{12} are examples of heterogeneous constraints. We ran all experiments on an Intel(R) Xeon(R) W-2255 CPU @ 3.70GHz machine with 128GB RAM running Windows 10 Enterprise (version 22H2). All of our experiments are executed over a single core and in the main memory setting.

Evaluation Metrics. We perform experiments for both DC verification and enumeration ((1) and (2) in our problem statement respectively) using Rapidash and Facet and report the end-to-end running time and space consumption. For enumeration performance, we report the total time to count and return the number of all the violations (same approach as Facet [35]) to avoid output materialization cost and focus on understanding the intrinsic hardness. All reported running times are the trimmed mean of five independent executions after the dataset has been loaded in memory.

[4]The column names in the DCs have been omitted due to security and privacy concerns.

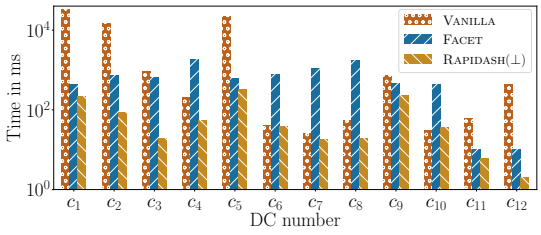


Figure 5: Running time for verification for vanilla range search tree, Facet, and Rapidash(⊥).

Implementation. Similar to prior work, Rapidash is implemented in Java. We use a standard implementation of orthogonal range trees (referred to as Rapidash(⊥)) and kd-trees (referred to as Rapidash(kd)). Since we were unable to obtain the original Facet source code from the authors of [35], we implemented Facet ourselves in Java using the Metanome infrastructure from [13] and [34], with all optimizations enabled as described in [35]. We manually verified that the performance of our implementation is in line with the numbers reported in [35] accounting for hardware differences[5]. Further, to ensure an apples-to-apples comparison, for DC verification, we ensure that Facet execution terminates as soon as the first violation is found.

5.2 Facet vs. Rapidash vs Vanilla Range Search

In this section, we answer **Q.1**, i.e., what is the performance of Rapidash vs Facet vs vanilla range search (as described in Section 4.8) for constraint verification. Results for constraint enumeration are included in [2] and follow similar trend as verification.

Figure 5 shows the running time of using a range tree directly (i.e. the vanilla approach)[6], Facet, and Rapidash(⊥) on all 12 DCs over the three open-source datasets. For 8 out of 12 DCs, directly applying range search was slower than Facet by up to 100×. While this may seem surprising at first glance, it is explained by the blowup due to logarithmic factors as described in Section 4.8, an observation that was also made in [41]. For the remaining 4 DCs, due to a large number of violations, both Rapidash(⊥) and vanilla range search tree terminated early and thus the running time is close. We note here that Rapidash(⊥) still has an advantage because our optimizations ensure that the data structure is created over a smaller number of dimensions. For all DCs, Rapidash(⊥) was the fastest out of all three. This result demonstrates that even after making the connection between DC processing and orthogonal range search, non-trivial optimizations are required to obtain competitive performance through range search algorithms. In the subsequent experiments, we exclude vanilla range search from the comparisons because Rapidash already demonstrates superior performance. We defer the interested reader to [2] for more experiments with vanilla range search. Instead, we concentrate on comparing Rapidash variants against the state-of-the-art algorithm, Facet.

[5]There is some deviation to be expected since the Tax dataset used in [35] has not been publicly released. We could only obtain a different version of the dataset (available at [1]) generated by a subset of the authors for a different publication.

[6]The results of using vanilla kd-trees is shown in the Appendix of [2] and is even slower compared to vanilla range trees.

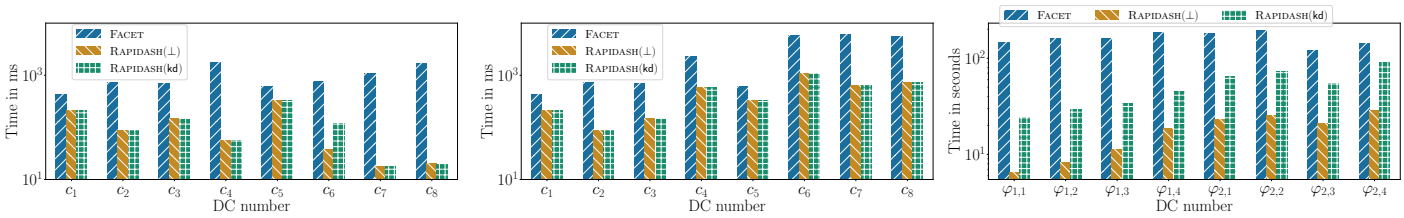
5.3 Evaluation on Open-Source Datasets

In this section, we answer **Q.2** by diving deeper into the tradeoffs between the two implementation of range trees and kd-trees. Figure 6 shows the running time for Facet and Rapidash. Let us fix our attention to Figure 6a. Our first observation is that for c_1 and c_5 DCs (which contain only equality and thus, both Facet and Rapidash take a provably linear amount of time in theory), Rapidash is faster by 2× for verification. This is because Facet requires cardinality estimation for all columns involved in the predicates, followed by creating the refinements which require iterating over the dataset again. Rapidash requires no statistics and iterates over the dataset only once. Constraints c_6, c_7, and c_8 all have a large number of violations (on the order of several hundred million). For these constraints, both versions of Rapidash are up to 84× faster than Facet. The performance gap is directly attributed to Proposition 1. Rapidash can find a violation after only looking at a few tuples in the dataset while Facet requires expensive computation. In fact, for c_6 and c_7, we observed that the size of all ordered pairs[7] after just the first refinement (which are inequality predicates) is 1.2B and 3.6B respectively.

The speedup improvement obtained by Rapidash also extends to the violation enumeration problem (Figure 6b). Although there is no early termination possible for counting, Rapidash still performs up to an order of magnitude better due to our improved algorithms. Facet performance, on the other hand, degrades further since the last refinement cannot be stopped early as Facet requires *all* refinements to be complete in order to begin counting. Note that both Rapidash(⊥) and Rapidash(kd) have the same performance numbers since all constraints contain at most two inequality predicates and thus, both range trees and kd-trees degenerate into a simple binary search tree. For the NCVoter dataset (Figure 7c), we also observe the same behavior. For all DCs, both variants of Rapidash are $2 - 200\times$ faster.

Figure 7 shows the space usage for both verification and violation enumeration. For Facet, space usage is calculated using the cardinality (in millions) of cluster pairs constructed and we use the number of nodes in the tree constructed for Rapidash(⊥) and Rapidash(kd). For every DC, Rapidash uses significantly lower space compared to Facet. The high space usage of Facet is directly attributed to the size of ordered pairs generated after refining predicates. The largest gap is observed for c_6, c_7, and c_8, which is expected since the DCs have a lot of violations, making refinement computation and storage expensive. On the other hand, for each constraint, Rapidash(kd) requires only (provably) linear amount of memory and this behavior can be directly observed in practice as well. For NCVoter, Facet memory use ranges from 256MB to 384MB (maximum memeroy was used by c_3 since it has 25B violations) and Rapidash memory use was between 8MB to 71MB. We note that for constraint c_{10}, an example of a heterogeneous constraint, Rapidash uses the inequality heterogenous predicate optimization from Section 4.6 and constructs a tree on only `birthyear` instead of both `ageatyearend` and `birthyear`, leading to a small memory footprint of only 8MB for both variants of Rapidash compared to 256MB for Facet.

[7]Given an ordered pair $(\text{tids}_1, \text{tids}_2)$, its size is defined as $|\text{tids}_1| + |\text{tids}_2|$

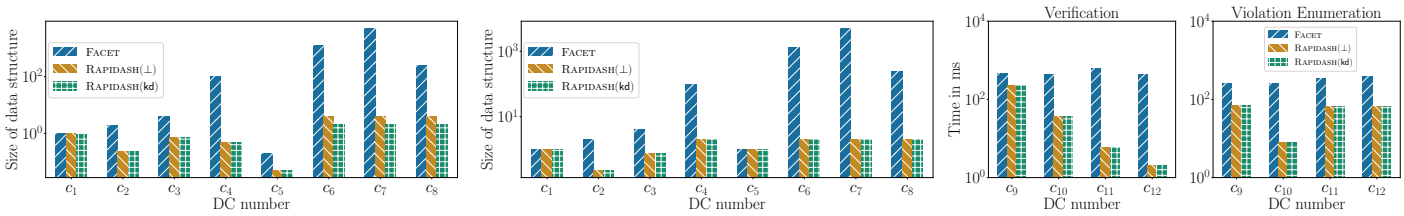


(a) Running time (in milliseconds) for DC verification on Tax and TPC-H.

(b) Running time (in milliseconds) for violation enumeration on Tax and TPC-H.

(c) Running time (in seconds) for DC verification on production datasets.

Figure 6: Running time for evaluation on different datasets.



(a) Space usage for DC verification on Tax and TPC-H.

(b) Space usage for violation enumeration on Tax and TPC-H.

(c) Running time (in milliseconds) for DC verification on NCVoter.

Figure 7: (Left and middle) Space requirement (cardinality of ordered pairs for FACET and number of nodes in the trees for RAPIDASH in millions) of different algorithms for DC verification on Tax and TPC-H; (Right) Running time (in milliseconds) for DC verification and violation enumeration on NCVoter.

Constraints with near worst-case behavior. To demonstrate that for FACET, even simple constraints on a small dataset can have poor performance in practice (as promised in Section 3), we tested the constraint $\phi = \neg(s.\text{Tax} \neq t.\text{Tax} \wedge s.\text{ExtPrice} \neq t.\text{ExtPrice})$ on TPC-H dataset. For this constraint, FACET runs for a few minutes and the program then crashes due to Java out-of-memory error due to creation of trillions of ordered pairs when refining the second predicate of the constraint. The materialization eventually consumes the entire main memory. However, RAPIDASH can count the number of violations within 5 seconds.

Table 4: Running time (in milliseconds) for violation enumeration on the TPC-H dataset with varying cardinality.

DC	Dataset size	Enumeration		
		FACET	RAPIDASH(⊥)	RAPIDASH(kd)
c_6	1M	5693	1075	1090
	2M	11713	1818	1835
	4M	24454	3739	3890
c_7	1M	6030	640	652
	2M	13653	1408	1423
	4M	34628	3053	3092
c_8	1M	5591	741	793
	2M	17136	1098	1124
	4M	34628	2772	2783

Scalability. Table 4 shows the behavior of the algorithms on the TPC-H dataset with varying cardinality. Let us take constraint c_8 (rightmost subfigure) as an example. As the cardinality increases, the speedup obtained increases from 7.5× for 1M to 13.86× when the dataset size is 4M. This suggests that the running time of FACET grows at a non-constant rate compared to RAPIDASH which grows in line with expectation.

Table 5: Running time (in milliseconds) of FACET and RAPIDASH (R(⊥) and R(kd)) on DCs for Tax for varying number of violations.

DC	# vio	% rows changed	Detection			Enumeration		
			FACET	R(⊥)	R(kd)	FACET	R(⊥)	R(kd)
c_1	138K	5%	347	1	1	812	294	294
	264K	10%	352	1	1	835	251	251
	482K	20%	373	1	1	904	262	262
	920K	50%	406	1	1	1042	295	295
c_2	8.4M	5%	469	2	2	1196	104	104
	15.7M	10%	566	1	1	1500	136	136
	27M	20%	504	1	1	1500	114	114
	43M	50%	492	2	2	1812	121	121
c_3	1.2B	5%	616	12	12	1014	160	160
	2.2B	10%	585	10	10	991	162	162
	3.6B	20%	615	4	4	1030	172	172
	5.4B	50%	589	4	4	1812	133	133
c_4	0.8B	5%	1781	16	26	9008	544	544
	1.5B	10%	1625	15	20	16656	630	630
	2.6B	20%	1792	9	29	22094	621	621
	4.1B	50%	1729	15	31	28280	705	705

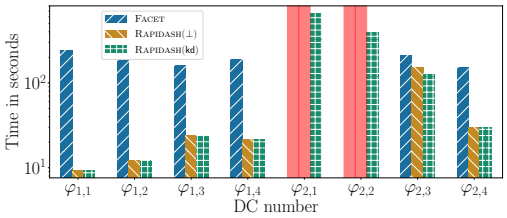


Figure 8: Running time (in seconds) for violation enumeration on production datasets. Red (solid) bars for $\varphi_{2,1}$ and $\varphi_{2,2}$ denote an out-of-memory error.

5.4 Evaluation on Production Datasets

In this section, we answer **Q.3**. Figure 6c shows the running time (in log scale) of Rapidash for all production datasets and DCs. The speedup obtained by our algorithm is close to an order of magnitude and up to 40×. Compared to Facet, both algorithms perform significantly better on all DCs. Rapidash(⊥) performs better than Rapidash(kd) on all DCs. This is not surprising since using kd-trees for orthogonal range search is more expensive as shown by their big-O time analysis. However, Rapidash(kd) is still faster than Facet by up to 20×.

The speedup obtained by Rapidash can be attributed to two reasons. First, Rapidash can terminate as soon as a violation is discovered, as opposed to Facet, which cannot do early termination in general (see Section 3). The second reason is that Rapidash does not require any expensive materialization as opposed to the ordered pair generation that is done by Facet. We also measured the space usage of all systems. For Facet, the space usage is the cardinality of the cluster pairs generated at each stage of the refinement pipelines. For Rapidash based algorithms, the space usage refers to the number of points inserted in the tree. For all DCs, Facet used 1.4−8× more space compared to Rapidash(⊥). Rapidash(kd) was a further order of magnitude lower in its space requirement compared to Rapidash(⊥). Interested reader can find the figure in our technical report. .

Figure 8 shows the running time for violation enumeration. For most constraints, we observe a similar trend in the running time as we saw for verification. The most interesting observation is that for constraints $\varphi_{2,1}$ and $\varphi_{2,2}$, both Facet and Rapidash(⊥) fail to complete due to Java out-of-memory error, as both constraints contain a large number of inequality predicates. However, Rapidash(kd) can finish the computation in about 10 minutes, thanks to its linear memory use guarantee. In terms of space usage, we observed very similar behavior as Figure 7. We also observed good scalability behavior on the production datasets. We omit the graphs due to space limitations and refer the interested reader to the tech report [2].

5.5 Varying the Number of Violations

We conclude this section by answering **Q.4**. We analyze the performance for a fixed set of DCs but modify the dataset to vary the number of violations. We chose the Tax dataset for the experiment since all four DCs have zero violations on the unperturbed data. To introduce violations, we take the original table and modify a certain fraction of the rows by replacing the values in the row with another value from the domain of the column. Table 5 shows the number of violations for each DC as we modify 5%, 10%, 20%, 50% of the dataset and running time. For detection, Rapidash is up to two orders of magnitude faster due to benefit of early termination. In the case of enumeration, all systems need to perform more computation. For Facet, we observe that as the number of violations increases, the running time also increases. This behavior is attributed to the cost of materializing cluster pairs which get large as the number of violations increase. The effect is starkly visible for c_4 where the running time increase by over three times for 9s to 28s. The performance of both variants of Rapidash is more robust to the change in number of violations since tree based structures allow efficient counting. The space usage of Rapidash was also observed to be an order of magnitude smaller than Facet, in line with our experimental results for other open-source and production datasets.

6 RELATED WORK

DC Violation Detection. DCs as an integrity constraint language was originally proposed by Chu et al. [15]. We refer the reader to [4, 5] for a general overview. To the best of our knowledge, Facet [35] is the state-of-the-art algorithm for DC violation detection. Our proposed algorithm has better worst-case time/space complexity than Facet, as well as methods that rely on DBMS to detection violations [17, 20, 38], resulting in significant performance improvements in practice.

DC Discovery. DC verification and enumeration are closely related to solutions for DC discovery (such as Hydra [13] and DCFinder [34]). Both of these systems rely on the two-step process of first building the evidence set, followed by enumerating the DCs. The two-step approach is also been successfully used for other dependency discovery algorithms [33, 39].

Range Searching. The connection between geometric algorithms and general join query processing has been made by several prior works [28, 31]. Specifically, range searching has been used for aggregate query processing [27] and CQs involving comparisons [41, 42]. Optimizations introduced in this paper could also be applied to certain subsets of queries considered in [41, 42] since DCs can be expressed as CQs with "short" comparisons but with the added twist that the query is a self-join. A further consideration in our work (but not elaborated in the prior works) is that we have a significant interest in making sure that our algorithms can run in linear space, a requirement for production implementation. Range trees and their variants have also been extensively used in geospatial information systems (see [7–9, 14, 16, 22, 26] for an overview) and indexes for database systems [23]. For an overview of the theoretical aspects of range searching, we refer the reader to [6].

7 CONCLUSIONS

In this paper, we studied the problem of DC violation detection. We presented Rapidash, a DC violation detection algorithm with near-linear time complexity with respect to the dataset size that leverages prior work on orthogonal range search. Through empirical evaluation, we demonstrated that our algorithm is faster than the state of the art by up to 84× on open-source datasets and large-scale production datasets.

REFERENCES

[1] 2023. https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-dc-algorithms.html. Accessed: 09/28/2023.

[2] 2024. Technical Report: Efficient Detection of Constraint Violations. https://aka.ms/rapidash.

[3] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment* 9, 12 (2016), 993–1004.

[4] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data Profiling: A Tutorial. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1747–1751. https://doi.org/10.1145/3035918.3054772

[5] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. Data Profiling. Morgan & Claypool Publishers. *Synthesis Lectures on Data Management* (2018).

[6] Pankaj K. Agarwal. 2004. Range Searching. In *Handbook of Discrete and Computational Geometry, Second Edition*, Jacob E. Goodman and Joseph O'Rourke (Eds.). Chapman and Hall/CRC, 809–837. https://doi.org/10.1201/9781420035315.ch36

[7] Lars Arge, Mark de Berg, Herman Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)* 4, 1 (2008), 1–30.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[9] Norbert Beckmann and Bernhard Seeger. 2009. A revised R*-tree in comparison with related index structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 799–812.

[10] Jon Louis Bentley and Jerome H Friedman. 1979. Data structures for range searching. *ACM Computing Surveys (CSUR)* 11, 4 (1979), 397–409.

[11] Jon Louis Bentley and James B Saxe. 1980. Decomposable searching problems I. Static-to-dynamic transformation. *Journal of Algorithms* 1, 4 (1980), 301–358.

[12] Pouya Bisadi and Bradford G Nickerson. 2011. Orthogonal Range Search using a Distributed Computing Model.. In *CCCG*.

[13] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient denial constraint discovery with hydra. *Proceedings of the VLDB Endowment* 11, 3 (2017), 311–323.

[14] King Lum Cheung and Ada Wai-Chee Fu. 1998. Enhanced nearest neighbour search on the R-tree. *ACM SIGMOD Record* 27, 3 (1998), 16–21.

[15] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *Proceedings of the VLDB Endowment* 6, 13 (2013), 1498–1509.

[16] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer. https://www.worldcat.org/oclc/227584184

[17] Wenfei Fan, Chao Tian, Yanghao Wang, and Qiang Yin. 2021. Parallel discrepancy detection and incremental detection. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1351–1364.

[18] Anna Fariha, Ashish Tiwari, Arjun Radhakrishna, Sumit Gulwani, and Alexandra Meliou. 2021. Conformance Constraint Discovery: Measuring Trust in Data-Driven Systems. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 499–512. https://doi.org/10.1145/3448016.3452795

[19] Chang Ge, Shubhankar Mohapatra, Xi He, and Ihab F. Ilyas. 2021. Kamino: Constraint-Aware Differentially Private Data Synthesis. *Proc. VLDB Endow.* 14, 10 (2021), 1886–1899. https://doi.org/10.14778/3467861.3467876

[20] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with Llunatic. *The VLDB Journal* 29 (2020), 867–892.

[21] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning denial constraint violations through relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 805–815.

[22] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*. ACM Press, 47–57. https://doi.org/10.1145/602259.602266

[23] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. *Generalized Search Trees for Database Systems*. Morgan Kaufmann. 562–573 pages. http://www.vldb.org/conf/1995/P562.PDF

[24] John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[25] Toshihide Ibaraki, Alexander Kogan, and Kazuhisa Makino. 1999. Functional dependencies in Horn theories. *Artificial Intelligence* 108, 1-2 (1999), 1–30.

[26] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. 2002. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*. ACM, 546–557. https://doi.org/10.1145/564691.564755

[27] Mahmoud Abo Khamis, Ryan R Curtin, Benjamin Moseley, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional aggregate queries with additive inequalities. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–41.

[28] Mahmoud Abo Khamis, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems (TODS)* 41, 4 (2016), 1–45.

[29] 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. *Proc. VLDB Endow.* 8, 13 (2015), 2074–2085. https://doi.org/10.14778/2831360.2831362

[30] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *The VLDB Journal* 31, 1 (2022), 1–22.

[31] Hung Q Ngo, Dung T Nguyen, Christopher Re, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 234–245.

[32] Mark H Overmars. 1983. *The design of dynamic data structures*. Vol. 156. Springer Science & Business Media.

[33] Thorsten Papenbrock and Felix Naumann. 2016. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*. 821–833.

[34] Eduardo HM Pena, Eduardo C de Almeida, and Felix Naumann. 2019. Discovery of approximate (and exact) denial constraints. *Proceedings of the VLDB Endowment* 13, 3 (2019), 266–278.

[35] Eduardo HM Pena, Eduardo C de Almeida, and Felix Naumann. 2021. Fast detection of denial constraint violations. *Proceedings of the VLDB Endowment* 15, 4 (2021), 859–871.

[36] Eduardo HM Pena, Edson R Lucas 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 & Knowledge Management*. 1235–1244.

[37] Eduardo HM Pena, Fabio Porto, and Felix Naumann. 2022. Fast Algorithms for Denial Constraint Discovery. *Proceedings of the VLDB Endowment* 16, 4 (2022), 684–696.

[38] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[39] Philipp Schirmer, Thorsten Papenbrock, Ioannis Koumarelas, and Felix Naumann. 2020. Efficient discovery of matching dependencies. *ACM Transactions on Database Systems (TODS)* 45, 3 (2020), 1–33.

[40] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of Order Dependencies. *Proc. VLDB Endow.* 5, 11 (2012), 1220–1231. https://doi.org/10.14778/2350229.2350241

[41] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data*. 108–121.

[42] Dan E Willard. 1996. Applications of range query theory to relational data base join and selection operations. *journal of computer and system sciences* 52, 1 (1996), 157–169.

Algorithm 5: DC Processing

Input : Relation $\mathbf{R}$, Homogeneous DC φ, $f \in \{\texttt{booleanRangeSearch}, \texttt{count}, \texttt{enumerate}\}$

```
1  k ← |vars(φ) \ vars_=(φ)|, c ← 0
2  H ← empty hash table
3  foreach t ∈ R do
4      v ← π_{vars_=(φ)}(t)
5      if v ∉ H then
6          if k ≠ 0 then
7              H[v] ← new OrthogonalRangeSearch()
8          else
9              H[v] ← ∅
10     if k ≠ 0 then
11         L, U ← SearchRange(t)                                   /* From Algorithm 2 */
12         L', U' ← InvertRange(L, U)
13         process(t, v, false)
14         H[v].insert(π_{vars(φ)\vars_=(φ)}(t))
15     else
16         process(t, v, true)
17         H[v] ← H[v] ∪ t
18 procedure process(t, v, isEqual)
19     if f = booleanRangeSearch then
20         if isEqual then
21             return true if |H[v]| > 1
22         else
23             return true if H[v].booleanRangeSearch(L, U) ∨ H[v].booleanRangeSearch(L', U')
24     if f = count then
25         if isEqual then
26             c ← c + |H[v]| − 1
27         else
28             c ← c + H[v].count(L, U) + H[v].count(L', U')
29     if f = enumerate then
30         if isEqual then
31             output (s, t) for each s ∈ H[v]
32         else
33             output (s, t) for each s ∈ H[v].enumerate(L, U) ∪ H[v].enumerate(L', U')
```

A GENERAL ALGORITHM

Algorithm 5 shows a simple generalization that is able to perform verification, counting, and enumeration. The main difference compared to Algorithm 2 is the different processing done in the process function depending on the function f.

B ENUMERATION FOR HETEROGENEOUS CONSTRAINTS

Recall that if the constraint contains at least one row homogeneous predicate with inequality, then we can choose the column referenced in the said predicate as the sort column C and apply Algorithm 6.

If there exists no row homogeneous predicate, we pick any heterogeneous predicate (say $s.A < t.B$). Then, we create a copy of the relation $\mathbf{R}$ (say $\mathbf{R'}$). We sort $\mathbf{R}$ on column A and $\mathbf{R'}$ on column B in ascending order, and initialize pointers p_1 for $\mathbf{R}$ and p_2 for $\mathbf{R'}$ by pointing them to the head of the sorted relations. We will use the notation $p \rightarrow A$ to mean the value of attribute A for the tuple pointed to be pointer p.

We perform a sort-merge style traversal of both relations and insert only points for $\mathbf{R}$ into the data structure. We take the projection of the tuple pointed to by p_1 on the columns that participate in predicates that mention s, insert it into the range search data structure, and advance p_1 as long as $p_1 \rightarrow A < p_2 \rightarrow B$. If p_1 cannot be advanced, then we take the tuple pointed to by p_2, construct the range search query, and query the data structure containing points from $\mathbf{R}$ that are guaranteed to have their value for A smaller than $p_2 \rightarrow B$. Then, pointer p_2 is advanced. We keep advancing p_2 until $p_1 \rightarrow A < p_2 \rightarrow B$ and go back to processing $\mathbf{R}$.

C MISSING PROOFS

C.1 Analysis of Theorem 1

Time and Space Complexity. We next establish the running time of the algorithm. First, observe that if $k = 0$, then the algorithm takes $O(|\mathbf{R}|)$ time since the for loop only performs a constant number of hash table operations. If $k \geq 1$, the algorithm performs one insertion and two Boolean orthogonal range search queries in each iteration of the for loop. Suppose the insertion time complexity, denoted by $I(n)$, is of the form[8] $\log^{\alpha} n$ and search time complexity is $T(n)$ when the data structure has n points in it. The running time can be bounded as:

$$\sum_{i=1}^{|\mathbf{R}|} \Big(\underbrace{\log^{\alpha} i}_{\text{insertion time}} + \underbrace{2 \cdot T(i)}_{\text{query time}} \Big) < \int_{1}^{|\mathbf{R}|+1} \log^{\alpha} i \, di + \int_{1}^{|\mathbf{R}|+1} 2 \cdot T(i) \, di$$

$$= O(|\mathbf{R}| \cdot \log^{\alpha} |\mathbf{R}|) + \int_{1}^{|\mathbf{R}|+1} 2 \cdot T(i) \, di$$

Seminal work by Overmars [32] showed that using range trees and k-d trees, one can design an algorithm with the parameters as shown in Table 2.

The integral in the second term in the equation above can be bounded by setting $T(i) = \log^{k} i$ or $T(i) = i^{1-1/k}$. In both cases, the second term evaluates to $O(|\mathbf{R}| \cdot T(|\mathbf{R}|))$. For space usage, note that the hash table takes a linear amount of space in the worst case. Thus, the space requirement of the tree data structure determines the space complexity. The main result can be stated as follows.

Lemma 1. *Algorithm 2 correctly determines whether a homogeneous DC φ is satisfied.*

Proof. We first show that Algorithm 2 is correct when φ only contains equality predicates. In this case, it is sufficient to determine whether there exist two distinct tuples t_1 and t_2 such that $\pi_{\mathsf{vars}_{=}(\varphi)}(t_1) = \pi_{\mathsf{vars}_{=}(\varphi)}(t_2)$. The hash table H stores a counter for each distinct $\pi_{\mathsf{vars}_{=}(\varphi)}(t)$ and increments it for each tuple $t \in R$ (Line 6). Thus, the algorithm will correctly return false as soon as some counter becomes greater than one and return true only if no such t_1, t_2 exists. Next, we consider the case when there exists at least one predicate with inequality. We show the proof for the case when all inequality predicate operators are $<$, i.e., all predicates in the DC are of the form $(s.A = t.A)$ or $(s.A < t.A)$. The proof for other operators is similar. We first state the following claim.

Claim 1. *Let w be the set of attributes that appear in the predicates with inequalities. Two tuples t_1 and t_2 in the same partition can form a violation iff $t_1(w) \prec t_2(w)$ or $t_2(w) \prec t_1(w)$, where the notation $t(w)$ denotes the projection, $\pi_w(t)$, of tuple t on attributes w.*

Here, $\prec$ is the standard coordinate-wise strict dominance checking operator. Claim 1 follows directly from the semantics of the operator under consideration and the definition of a violation. Suppose t is the tuple being inserted in the tree. Line 10 will query the range tree with $\mathbf{L} = (-\infty, \ldots, -\infty), \mathbf{U} = (t(v_1), \ldots, t(v_k))$ and $\mathbf{L}' = (t(v_1), \ldots, t(v_k)), \mathbf{U}' = (\infty, \ldots, \infty)$. In other words, the algorithm searches for a point in the tree such that t is strictly smaller or larger for each of the k coordinates. The existence of such a point would imply that there exists a pair that forms a violation.

If the orthogonal range search finds no point, Claim 1 tells us that t cannot form a violation with any tuple s already present in the range tree. In each iteration of the loop, we insert one tuple into the range tree. Therefore, if t_1 and t_2 form a violation, it will be discovered when one of them (say t_2) is already inserted in the range tree and t_1 is being processed in the loop. This completes the proof. □

C.2 Other missing proofs

Proof of Proposition 1. We sketch the proof for $\varphi : \forall s, t \in \mathbf{R},\ \neg(s.A = t.A \wedge s.B < t.B)$, which can be extended straightforwardly for other DCs of interest. We construct a unary relation $\mathbf{R}(A, B)$ of size N as follows: the first tuple t_1 is (a_1, b_1) and the remaining $N - 1$ tuples are (a_1, b_2) where $b_1 < b_2$. Note that t_1 forms a violation with every other tuple in the relation. Algorithm 2 initializes one range tree when processing t_1 (Line 9) and inserts t_1 in it (Line 12). Thereafter, when tuple t_2 is processed, the range search query (Line 10) will return true and the algorithm will terminate. Note that all the operations take $O(1)$ time since the tree only contains two tuples. However, FACET requires $\Omega(|\mathbf{R}|)$ time for processing the refinement of $s.A = t.A$ predicate.

Proof of Proposition 2. Consider the constraint $\varphi : \neg(\phi \wedge s.A \neq t.A)$, where ϕ is a conjunction of homogeneous equality and disequality predicates. Let (q, r) be a violation to φ. Without loss of generality, we assume that $q.A < r.A$, and then (q, r) is also a violation to $\varphi_1 : \neg(\phi \wedge s.A < t.A)$. Since ϕ only contains equality and disequality predicates, (r, q) also satisfies ϕ by symmetricity, and therefore (r, q) is a violation to φ and $\varphi_2 : \neg(\phi \wedge s.A > t.A)$. In fact, for any violation (r, q) to φ, one of (r, q) and (q, r) must violate φ_1 while the other violates φ_2. Thus, we only need to check φ_1 for violations, which contains $l - 1$ disequality predicates and can be written as a conjunction of 2^{l-1} DCs containing no disequality predicates by logical equivalence.

Allowing mixed homogeneous constraints. We now extend our verification algorithm to work also for mixed homogeneous constraints that can contain predicates of the form $s.A$ op $s.B$ as well as $s.A$ op $t.A$. Let $\forall s, t : \neg\phi(s, t)$ be a mixed homogeneous denial constraint. We first rewrite ϕ in the form $\phi_S(s) \wedge \phi_T(t) \wedge \phi_{ST}(s, t)$ where ϕ_S contains all predicates that mention only s (and not t), ϕ_T contains all predicates

[8] Throughout the paper, we use $\log^{k} N$ to mean $(\log N)^{k}$ and not iterated logarithms.

that mention only t (and not s), and ϕ_{ST} contains all predicates that mention both s and t. The constraint $\forall s, t : \neg\phi$ that we need to verify over a given $\mathbf{R}$ can be equivalently rewritten as follows:

$$\begin{aligned} \forall s, t : \neg\phi \quad &\Leftrightarrow \quad \forall s, t : \neg(\phi_S(s) \wedge \phi_T(t)) \vee \neg\phi_{ST}(s, t) \\ &\Leftrightarrow \quad \forall s, t : (\phi_S(s) \wedge \phi_T(t)) \Rightarrow \neg\phi_{ST}(s, t) \\ &\Leftrightarrow \quad \forall s \in \mathbf{S} : \forall t \in \mathbf{T} : \neg\phi_{ST}(s, t) \end{aligned}$$

where $\mathbf{S}$ is the set of all tuples in $\mathbf{R}$ s.t. ϕ_S is true, and $\mathbf{T}$ is the set of all tuples in $\mathbf{R}$ s.t. ϕ_T is true. Note that $\mathbf{S}$ and $\mathbf{T}$ can overlap.

Example 13. *We give an example of a mixed homogeneous constraint inspired by one of our production scenario. Consider the table* `Docship` *with columns* (`Shipcity, Receiptcity, Shipdate, Receiptdate`). *Consider the constraint* $\neg(s.\texttt{Shipcity} = s.\texttt{Receiptcity} \wedge t.\texttt{Shipcity} = t.\texttt{Receiptcity} \wedge s.\texttt{Shipcity} = t.\texttt{Shipcity} \wedge s.\texttt{Receiptdate} \geq t.\texttt{Shipdate} \wedge s.\texttt{Shipdate} \leq t.\texttt{Receiptdate})$. *The constraint is mixed-homogeneous since the first three predicates are homogeneous and the remaining predicates are heterogeneous. The constraint encodes the business logic that for intra-city document shipping, the* [`Shipdate, Receiptdate`] *intervals of orders in the same city never overlap, i.e, a new document is shipped only if a previously shipped document has been received.*

For verification, we maintain two range search data structures (same as the H in Algorithm 2) $H_{\mathbf{S}}$ and $H_{\mathbf{T}}$ for points in $\mathbf{S}$ and $\mathbf{T}$ respectively. For each tuple (aka point) $q \in \mathbf{R}$, we first check whether it belongs to $\mathbf{S}$ and $\mathbf{T}$.

- If $q \in \mathbf{S}$, we perform range search on $H_{\mathbf{T}}$ to find any point r such that $\phi_{ST}(q, r)$ is true. If there is no such point, we insert q into $H_{\mathbf{S}}$. Otherwise, the constraint does not hold and the algorithm terminates. This step checks whether there is any previously seen point r in $\mathbf{T}$ such that (q, r) forms a violation.
- Similarly, if $q \in \mathbf{T}$, we perform range search on $H_{\mathbf{S}}$ to find any point r such that $\phi_{ST}(r, q)$ is true. If there is no such point, we insert q into $H_{\mathbf{T}}$. Otherwise, we output false.

Example 14. *Continuing example 13, suppose the table has the following rows.*

	`Shipcity`	`Receiptcity`	`Shipdate`	`Receiptdate`
t_1	*Paris*	*Paris*	*24-01-2024*	*26-01-2024*
t_2	*Paris*	*Paris*	*27-01-2024*	*28-01-2024*

Observe that $\phi_S(s) = (s.\texttt{Shipcity} = s.\texttt{Receiptcity})$ *and* $\phi_T(t) = (t.\texttt{Shipcity} = t.\texttt{Receiptcity})$. *We build 2-D range search structures* H_S *and* H_T *over (*`Shipdate, Receiptdate`*).* H_S *and* H_T *both store points whose* `Shippingcity` *is the same as* `Receivingcity`. *Since* $t_1.\texttt{Shippingcity} = t_1.\texttt{Receivingcity}$, *we insert (24-01-2024, 26-01-2024) into* H_S *and* H_T. *When processing* t_2, *since* t_2 *satisfies* $\phi_S(s)$, *we query* H_T *to find a point whose* `Shipcity` *is Paris,* `Shipdate` *is smaller than or equal to 28-01-2024 and* `Receiptdate` *is greater than or equal to 27-01-2024, and get no point for this query. Similarly, since* t_2 *also satisfies* $\phi_T(t)$, *we query* H_S *to find a point whose* `Shipcity` *is Paris,* `Shipdate` *is smaller than or equal to 28-01-2024 and* `Receiptdate` *is greater than or equal to 27-01-2024, and also get no point. Therefore, the constraint holds on the table. Note that in this example since* H_S *and* H_T *store the same set of points, we only need to keep one of them.*

D ENUMERATING VIOLATIONS

Algorithm 6 shows the detailed steps for enumerating violations for a homogeneous DC. We begin by sorting the relation on a column that participates in some predicate that has an inequality operator (line 2). We can safely assume that the operator is either $<$ or $>$ since $\leq, \geq$ operator can be decomposed into two constraints: one containing only $=$ as the operator for the predicate and the other containing $<$ or $>$. The algorithm iterates over the dataset and uses the hash table H or range search data structure in a similar fashion to Algorithm 2. There are two important differences to note. First, we can omit the inverted range search. This is because the sort order guarantees that for any tuple t, a violation can be formed only with tuples that appear before t in the sort order. Second, since the dataset has already been sorted on one of the columns with inequality predicates, the number of dimensions of the point inserted in the range search data structure is reduced by one compared to Algorithm 2. The same idea can also be applied to any DC containing at least one row homogeneous predicate and using the column referenced in the predicate for sorting. For DCs containing only heterogeneous predicates, we use a sort-merge style approach.

Theorem 2 has important implications. By reducing the number of dimensions by one, we get a logarithmic factor improvement in running time and space requirement for range trees, and a polynomial improvement in running time when using k-d trees compared to using the enumeration variant of Algorithm 2. As an example, for the class of constraints that contain at most two inequality predicates, we can do the enumeration of violations in $O(|\mathbf{R}| \log |\mathbf{R}| + K)$ time to enumerate K violations. This class of constraints has been shown to be representative of constraints that are routinely observed in practice [35]. We conclude this section by noting that by using $\text{count}(\mathbf{L}, \mathbf{U})$ instead of $\text{enumerate}(\mathbf{L}, \mathbf{U})$ on line 22 of Algorithm 6, we can compute the number of violations as well.

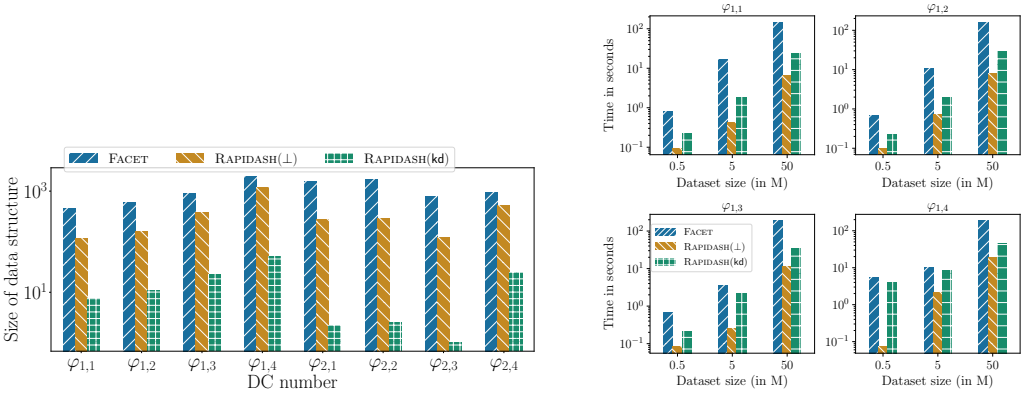
E ADDITIONAL EXPERIMENTS

Scalability on production datasets. To study the scalability of Rapidash, we used dataset D_1 and varied the number of rows to understand the impact of input size on the running time of the DCs. Figure 9b shows the results when varying the dataset size of D_1 from 0.5M to 50M. Both Rapidash($\perp$) and Rapidash(kd) scale almost linearly for the first three DCs. For $\varphi_{1,4}$, while Rapidash($\perp$) scales linearly, Rapidash(kd) has super linear scalability, which is in line with the expectation. The behavior of Rapidash on other datasets was also very similar. The

Algorithm 6: DC enumeration

Input : Relation $\mathbf{R}$, Homogeneous DC φ containing at least one row homogeneous inequality predicate
Output: DC violations

$H \leftarrow$ empty hash table
sort $\mathbf{R}$ on a column (say C) that participates in an inequality predicate in ascending order if the predicate operator is $<$ (and descending for $>$)
$\ell \leftarrow |\text{vars}(\varphi) \setminus \{C \cup \text{vars}_{=}(\varphi)\}|$
Temp $\leftarrow \emptyset$
foreach $t_i \in \mathbf{R}$ **do**
  $v \leftarrow \pi_{\text{vars}_{=}(\varphi)}(t_i)$
  if $v \notin H$ **then**
    if $\ell \neq 0$ **then**
      $H[v] \leftarrow$ new OrthogonalRangeSearch()
    else
      $H[v] \leftarrow \emptyset$
  if $\ell \neq 0$ **then**
    /* Two rows can only form a violation if they satisfy the predicate containing C. Therefore, no violation can be formed for tuples that have the same value for attribute C. */
    if $t_i.C = t_{i+1}.C$ **then**
      Temp $\leftarrow$ Temp $\cup\ t_i$
    else
      if Temp $\neq \emptyset$ **then**
        foreach $r \in$ Temp **do**
          $H[v]$.insert$(\pi_{\text{vars}(\varphi)\setminus\{C\cup\text{vars}_{=}(\varphi)\}}(r))$
        Temp $\leftarrow \emptyset$
      $H[v]$.insert$(\pi_{\text{vars}(\varphi)\setminus\{C\cup\text{vars}_{=}(\varphi)\}}(t_i))$
    $\mathbf{L}, \mathbf{U} \leftarrow$ SearchRange(t_i) /* From Algorithm 2 */
    $\mathcal{L} \leftarrow H[v]$.enumerate$(\mathbf{L}, \mathbf{U})$
    output (s, t) for each $s \in \mathcal{L}$
  else
    output (s, t) for each $s \in H[v]$
    $H[v] \leftarrow H[v] \cup t_i$



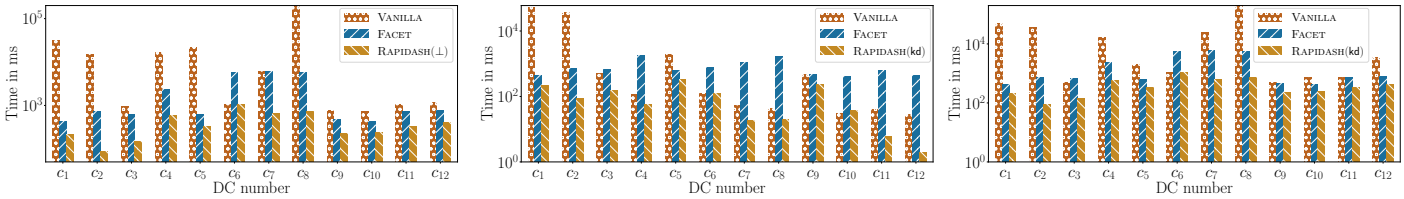
(a) Space requirement of different algorithms for DC verification on prod datasets

(b) Running time (in seconds) for DC verification on D_1 with varying cardinality.

Figure 9: Additional experiments on production datasets

performance gap between Facet and our solution narrows when the dataset size is small. This is expected since Facet performance depends on the sizes of cluster pairs generated by refinements.

Running time for vanilla. Figure 10 shows the running time of using range trees for DC enumeration and the experiments for running of using kd-trees for enumeration and verification. In all experiments, Rapidash was the best performing and in most cases, vanilla usage of range search algorithms was poor in performance. Facet performance was somewhere in the middle.



(a) Running time (in ms) for enumeration on all open source datasets.

(b) Running time (in ms) for enumeration on all open source datasets.

(c) Running time (in ms) for DC verification on all open source datasets.

Figure 10: Running time for vanilla range search for verification and enumeration.