Incremental Detection of Denial Constraint Violations

Youri Kaminsky
Hasso Plattner Institute
University of Potsdam, Germany
youri.kaminsky@hpi.de

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

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

ABSTRACT

Denial constraints (DCs) are well-known to express business rules on data. They subsume other integrity constraints (ICs), such as key constraints or functional dependencies. One can use traditional DBMS or specialized algorithms to validate such dependencies on a dataset. However, no known approach exists to detect DC violations *incrementally*. Data typically changes over time, and recomputing the entire violation set after every update is wasteful. Alerting data practitioners of data quality issues immediately, enables them to take measures earlier and can help prevent follow-up issues.

We present Weever, the first incremental approach to detect all violations of a given set of DCs. It uses a novel index structure to process inequality predicates and a new method to plan the execution order of predicates depending on their selectivity, reducing redundant computations when handling multiple DCs. Our evaluation shows that Weever outperforms a DBMS-based baseline by up to two orders of magnitude. And in the same time that a state-of-the-art static approach takes to analyze an entire dataset, Weever processes up to 200 000 insertions.

PVLDB Reference Format:
Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. Incremental Detection of Denial Constraint Violations. PVLDB, 18(4): 1000 - 1012, 2024.
doi:10.14778/3717755.3717761

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HPI-Information-Systems/Weever.

1 INTRODUCTION

Integrity constraints (ICs) are a cornerstone in enhancing data quality by ensuring consistency across database records. Most DBMSs support key constraints such as primary and foreign keys. However, supporting more complex constraints is needed for higher data quality standards, as production data usually contains much more complex data relationships. Consider the product shipping records in Table 1. It is fair to assume that shipping operation companies would like to maintain such records consistent with data quality ICs, for instance (i) package codes must be unique; (ii) two records must have equal distances between pairs of tracked packages where the origin of one package matches the destination of the other, and vice versa; or (iii) for any two packages traveling the same distance, the postage cost for the smaller package should not be higher than that for the larger package.

Table 1: Example relation of tracked packages

	package code	origin	destination	distance [km]	volume [m^3]	postage [$]
	0	London	Cape Town	9700	8	100
	1	Cape Town	Lima	9700	18	50
	2	London	New Delhi	6600	30	25
	3	Cape Town	London	9700	45	200
	4	New Delhi	London	6700	45	50
insert	5	Mexico City	Monaco	9700	18	10
delete	4	New Delhi	London	6700	45	50

Constraint (i) is easily captured as a primary key, but Constraints (ii) and (iii) require a more powerful formalism. As shown in [3, 13], denial constraints (DCs) are a practical way to model these and many other types of constraints. Typically, DCs are expressed as a conjunction of relational predicates over a relation. A relation instance is inconsistent (dirty) if it contains any set of tuples that satisfy the predicate conjunction, i.e., all the predicates of the DC simultaneously. These sets of tuples are *DC violations* and point to problematic combinations of values, i.e., data errors.

For the tracked packages example, the three constraints can be captured with the following DCs:

$\varphi_1 : \forall t, t' : \neg(t.\text{id} = t'.\text{id})$

$\varphi_2 : \forall t, t' : \neg(t.\text{origin} = t'.\text{destination} \wedge t.\text{destination} = t'.\text{origin} \wedge t.\text{distance} \neq t'.\text{distance})$

$\varphi_3 : \forall t, t' : \neg(t.\text{distance} = t'.\text{distance} \wedge t.\text{volume} < t'.\text{volume} \wedge t.\text{postage} > t'.\text{postage})$

Note that for example, tuples t_2 and t_4 violate DC φ_2: the distance between London and New Delhi varies. Such violations can occur during data ingestion or disappear when data is deleted, as illustrated by the operations marked in the example. We defer the more formal definition of DCs to Section 3.

Several data cleaning solutions have been proposed to identify and (potentially) repair DC violations and other data errors [6, 11, 21]. However, most existing solutions treat data cleaning as a one-time, offline task. Unfortunately, this approach is not practical in many scenarios: data changes rapidly, and since data-cleaning algorithms are computationally demanding, rerunning them at each update becomes cost-prohibitive.

Also, it is often impossible to fix violations immediately since there are many possible data repairs [13], and experts who can check each repair are usually not readily available. Nonetheless, tracking violations as soon as they occur can serve users with an immediate alert system that helps prevent cascading problems caused by unnoticed data errors. While the database might contain

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717761

errors, data reporting can leverage (and avoid) these errors to return, for instance, consistent query answers [1, 3].

The need for dynamic computation in scenarios where computation is expensive, and data changes frequently have long been acknowledged [20]. In data cleaning systems, the error detector serves as a critical bottleneck. A few studies have explored the challenges and applications of dynamic violation detection [7, 9], focusing on conditional functional dependencies, a type of constraint that is subsumed by DCs. For DC violation detection, in particular, this component has been implemented, for example, using SQL queries [21]. Recent work has shown performance problems with this approach [17]: some DBMS can take days to process the queries for typical DCs. Alternative solutions have been investigated, but so far, they have considered only static datasets, requiring rerunning for dynamic data.

In a quest towards dynamic data cleaning, our main contribution is the *first efficient incremental DC violation detection approach*: Weever. It adapts static DC violation detection concepts to the dynamic paradigm (Section 3). We introduce a novel data structure, the *LT-tree*, to efficiently process inequality predicates incrementally (Section 4). Additionally, Weever *schedules* the predicates of the given DCs optimally and maximizes the computational reusage when processing multiple DCs (Section 5). We conduct a comprehensive evaluation of Weever, comparing it against state-of-the-art static approaches and a baseline across various scenarios (Section 6).

2 RELATED WORK

Generally speaking, *error detection* is a primary stage in data cleaning [13]. Several data cleaning tools implement this stage by triggering SQL queries to a DBMS [8, 10, 11, 21]. For example, Fan et al. devised a set of SQL-based techniques for commercial DBMS to detect violations of conditional functional dependencies [8]. PostgreSQL has also been used in [11] and [21] to detect errors. The system in [10] also uses entity enhancement rules that were translated into SQL queries and user-defined functions.

As experimentally shown in [17], DBMS are not equipped to handle certain workloads, in particular, some types of DCs with complex predicates. Thus, efficient DC violation detection has been studied before [17, 18]. Additionally, some discovery methods for DCs share the sub-problem of validating possible DC candidates on the entire dataset [4]. These three approaches are closely related, building upon each other. All use the concept of refining a compact candidate representation with specialized algorithms for each operator. Weever adapts this concept for the incremental scenario, because it does not need to process pairs of tuples for inserts, but rather compares the newly inserted tuple to the existing data.

Similarly to Weever, [18] uses compressed bitsets to represent the candidate set. While [17] argues that switching between different representations of candidate sets improves performance, our approach benefits from fast set operations in each step. Thus, we avoid the overhead of transforming the representation. Finally, [17] shows the importance of the processing order of predicates in a DC. The authors propose a sketch-based method to better predict the selectivity of predicates. Since Weever creates an inverted index for every predicate up front, there is no need for cardinality estimation, and we can order predicates differently.

A different perspective is taken in [16]: Rapidash focuses on DC verification, i.e., finding the first violation in a dataset. Weever cannot be easily adapted to this scenario, as it always processes the predicates for all tuples. We could abort execution after identifying the first violation, but the performance would not be comparable to [16]. Nonetheless, the system also allows detecting all violations in a dataset. Its specialized index structure for inequality predicates is inspired by orthogonal range search, and processes multiple inequality predicates simultaneously. We develop our own index structure for inequality predicates, the Less-Than-tree (LT-tree), which is geared to the incremental scenario and can be used to validate multiple DCs simultaneously, in contrast to [16].

The incremental DC *discovery* problem is studied in [19]. However, that approach can handle only insertions, while ours can additionally process deletions (and updates as a deletion plus insertion). Thus, the discovered DCs might become non-minimal after a deletion or update. Since the predicate space in DC-discovery is larger, the approach only indexes one predicate per DC and loops over all tuple pairs generated by the index to validate DCs. As the DCs are predefined in our use case, we can keep an index for every predicate. The authors of [19] propose a new index structure for inequality predicates, which we discuss in detail in Section 4.1 and evaluate in Section 6.3.1.

3 DENIAL CONSTRAINTS

We use the denial constraint (DC) formalism to define business rules to ensure data quality. DCs subsume other known integrity constraints, such as unique column constraints (UCCs), functional dependencies (FDs) and order dependencies (ODs), and can express even more complex rules. We use the notation from [5] and formally define a DC as follows.

Definition 3.1. Given a relation R and a relational instance r, let $A, B \in R$ be possibly equal attributes, $t, t' \in r$ two tuples, and θ an operator. A *predicate* p_i is defined as $t.A\,\theta\,t'.B$. A DC φ is of the form $\varphi : \forall t, t' \neg(p_1 \wedge ... \wedge p_m)$.

We restrict the types of DCs considered in this work in accordance to previous works [4, 5, 17]: (1) inequality predicates ($<, \leq, >, \geq$) for text data are not supported, as the order typically has no semantic link to other ordered columns; and (2) the number of tuples participating in a DC is set to two. On the one hand, the definition above can be extended to use more tuples in different predicates, but most interesting dependencies can be found when comparing tuple pairs. On the other hand, we might compare a single tuple to a constant. However, in the context of DC violation detection, these predicates work as a simple filter, as they do not require other tuples to validate the predicate. Since all further DCs use the all quantifier $\forall t, t'$, we omit it from now on to improve visual clarity. Our algorithm Weever supports the comparison operators $\theta \in \{=, <, \leq, >, \geq, \neq\}$. While we refer to $=$ as *equality* and $\neq$ as *non-equality*, the other operations are generalized as *inequalities*.

3.1 DC violations

To *violate* a given DC, a tuple pair t, t' must fulfill all its predicates. As a DC is a negated conjunction of predicates, a single unfulfilled predicate prevents a violation. We can use this property to efficiently validate DCs, by considering only those tuple pairs that

already fulfilled all previously regarded predicates. Therefore, the predicates of a DC naturally resemble a pipeline. Each predicate receives the candidate set of still possible violating tuple pairs from the previous predicate and removes those tuple pairs that do not fulfill it. The candidate set is referred to as *intermediate* and the process of reducing it as *refinement*. Like related work, our refinement uses specialized algorithms for each operator [4, 17].

We continue our running example and examine the refinement pipeline for DC φ_2 as shown below. Initially, the candidate set contains all tuple pairs. The intermediate is refined by each predicate, until it contains all DC violations at the end, namely $\{(t_2, t_4), (t_4, t_2)\}$. In most approaches, the intermediate does not contain the actual tuple pairs, but uses a compact representation.

Step	Processed predicate	Intermediate
1	t.origin = t'.destination	$\{(t_0, t_3), (t_0, t_4), (t_1, t_0), (t_2, t_3),$ $(t_2, t_4), (t_3, t_0), (t_4, t_2)\}$
2	t.destination = t'.origin	$\{(t_0, t_3), (t_2, t_4), (t_3, t_0), (t_4, t_2)\}$
3	t.distance ≠ t'.distance	$\{(t_2, t_4), (t_4, t_2)\}$

3.2 Incremental DC violation detection

In static DC violation detection, the intermediate consists of tuple pairs – in an incremental scenario, we use a different structure. Because every new *insertion* can create new violations with the existing tuples, our initial candidate set contains all existing tuples. Implicitly, the new tuple forms a possible violating pair with all tuples in the intermediate set. As a DC is defined on tuple pairs, the inserted tuple can either be t or t' for validation purposes. Let $newT$ be the newly inserted tuple and T the set of all tuples. We create two independent intermediates, namely I and I' with $I = \{t \in T \mid (newT, t) \not\models \varphi\}$ and $I' = \{t \in T \mid (t, newT) \not\models \varphi\}$. However, we do not need a second intermediate for *reflexive* DCs. The two intermediates are always the same if there is neither a predicate that uses two attributes nor an inequality predicate. For example, φ_1 is reflexive, but φ_2 and φ_3 are not. Furthermore, a predicate can refine the two intermediates simultaneously if both A and B refer to the same attribute.

For our example DC φ_3, let us insert tuple t_5. We show the refinement pipeline below. Initially, the two intermediates contain all other tuples. As each predicate contains only one column, we can simultaneously refine both intermediates. As one intermediate is empty in the end, only (t_0, t_5) violates φ_3. In Section 5.4, we show how Weever processes inserted tuples in detail.

Step	Processed predicate	I	I'
1	t.distance = t'.distance	$\{t_0, t_1, t_3\}$	$\{t_0, t_1, t_3\}$
2	t.volume < t'.volume	$\{t_3\}$	$\{t_0\}$
3	t.postage > t'.postage	$\emptyset$	$\{t_0\}$

As *deletions* cannot create new violations, we do not need to process the predicate pipeline. Instead, deletions can only eliminate existing violations, so we have to efficiently keep track of them. For example, deleting t_4 removes both violating tuple pairs $(t_2, t_4), (t_4, t_2)$ from our previously discovered result for φ_2. In Section 5.6, we detail our handling of existing violations under deletions. Finally, *updates* can efficiently be processed by modelling them as a deletion and an insertion of the updated tuple.

4 LESS-THAN-TREE (LT-TREE)

We present a novel index structure, the LT-tree to efficiently validate inequality predicates (i.e., $<, \leq, >, \geq$), as these are the most expensive predicate class. As inequality predicates rely on ordering, hashing can speed up neither insertion nor querying. Therefore, our index structure extends a traditional Red-black tree[12] and represents an inverted index of the attribute values to the tuple ids that have that value. Additionally, tree nodes store a Less-Than-set (LT-set), which is the set of all tuple ids in the left subtree below them, i.e., the tuple ids that have a smaller value in the indexed column. Thus, the LT-tree has similarities to a segment tree [2]. The LT-sets allow our solution to achieve logarithmic time complexity for insertion, deletion, and both querying a single value and all smaller values. By retrieving all smaller values, we can also handle other inequality predicates by using the complement set.

4.1 Related approaches

Traditional DC validation approaches propose multiple algorithms to handle the different kinds of inequality predicates. However, none of them can be applied in our scenario. FACET [17] proposes three algorithms. The first, Hash-Sort-Merge (HSM), uses hash maps to deduplicate the attribute values and sorts the keys in a second step. Afterward, it generates the result refinements while iterating through the sorted key lists of the involved columns. While we could use the idea of hash maps for deduplication, we would need to re-sort the keys for every inserted value. Binning-Hash-Sort-Merge (BHSM) [17] is an improvement to efficiently handle cases with many distinct values. Here, the key space is partitioned into fixed buckets and HSM is run within these buckets. The key space in an incremental scenario is constantly changing, so the bucket boundaries shift often. Finally, [17] also employs the IEJoin [14] whenever two inequality predicates with many distinct values need to be processed. The method sorts the affected columns and uses a permutation array to store references to the other columns. Thus, it linearly scans the data only once after sorting. However, as the data structures used to optimize the access are fixed size arrays, they cannot be efficiently updated, which inhibits performance in our update-heavy scenario.

Qian et al. present an alternative way to process *two* inequality predicates jointly in an incremental setting [19]. They store ordered value pairs of both predicates in a skip list and traverse it to find the first violation. Then, they iterate through the rest of the list, as these values violate the predicates. However, the performance quickly degrades. First, the list must be sorted individually according to both pair values. Therefore, they must create entirely new lists when there are out-of-order values for even one predicate. For each new insertion, every list has to be processed separately. Second, the final iteration takes linear time if there are many errors because they must handle every violating tuple individually. Thus, their solution "Fetch" outperforms the LT-tree in an ideal setting, but for more realistic workloads it performs worse, as we show in Section 6.3.1.

4.2 What is stored?

The LT-tree is based on a Red-black tree [12]. According to the balancing rules of [12], the longest path from the root to a leaf is no more than twice as long as the shortest path from the root to

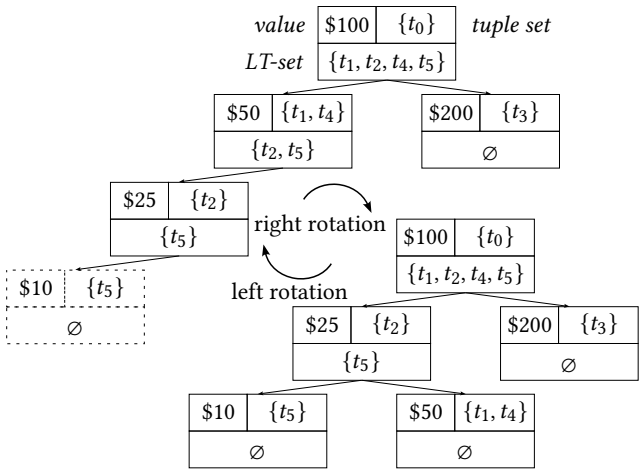


Figure 1: Exemplary state of the LT-tree; including the insert of t_5 (dashed node) and a necessary rebalance operation.

another leaf. An overview of the structures in memory are provided in Figure 1. Each node contains three components: (1) a distinct attribute **value** as its key. The value sorts the data. Like FACET [17], we allow only numeric attributes in inequality predicates, so there is a natural ordering between the values; (2) the **tuple set** of tuple ids that contain that attribute as a value; and (3) the **LT-set**, which is a union of all tuple sets in the left subtree below the node, i.e., all nodes with a smaller value. Thereby, we save approximately half of all tuple ids in every level of the tree.

Theorem 4.1. *Let n be the number of tuples that are uniformly distributed over d distinct values. The space complexity of a LT-tree is $O(log(d) \cdot n + d)$.*

Proof. There exist d nodes in the tree and each *tuple set* is on average of size n/d, since every distinct value occurs on average n/d times. Let h be the height of the tree. On each level i of the tree from 0 (root level) to $h-1$ (leaf level), there are up to 2^i nodes. Additionally, for each node in each level, we store the union of the *tuple sets* of their left subtree in the *LT-set*. The number of nodes in the left subtree for a node on level i is $2^{h-i-1}-1$. Thus, the size of the *LT-sets* in each level i are $2^i \cdot (2^{h-i-1}-1) \cdot n/d$. We combine the size of *tuple sets* and *LT-sets* for each level i.

$$\frac{n}{d} \cdot 2^i + \frac{n}{d} \cdot 2^i \cdot (2^{h-i-1}-1) = \frac{n}{d} \cdot 2^{h-1}$$

Therefore, every level of the tree stores the same amount of tuple ids. Since $h = log(d)$ for a balanced tree, we hold $log(d) \cdot 2^{log(d)-1} \cdot n/d = log(d) \cdot d/2 \cdot n/d = log(d) \cdot n/2$ tuple ids in the LT-tree. To also store the d distinct values, the overall space complexity is $O(log(d) \cdot n + d)$. □

Compared to a standard tree with a space complexity in $O(n+d)$, the additional overhead is feasible. The real-world space implications are even less severe: tuple id sets are represented as compressed bit sets, so the actual memory consumption is rather low, as shown in Section 6.3.1.

4.3 How is it accessed?

The LT-tree is traversed as a traditional binary tree for a search operation. For each visited node, the query value is compared to the current node's value. If the query value matches the value of the current node, we return the current node. Else, the left (smaller) or right (larger) child is selected as appropriate. However, we must extend the insertion and deletion procedures to maintain the LT-sets. We show that we can do so with no complexity overhead and how we query the tree for all tuples that have a smaller value than a given query value in logarithmic time complexity.

Insertion. Inserting into an LT-tree follows the same pattern of a traditional binary tree. However, we need to additionally update the LT-sets. We show the LT-tree after the insert of t_5 in Figure 1.

The insert procedure works as follows. Let t be the newly inserted tuple, v the value of t in the index attribute and t_i the tuple id of t.

(1) Traverse the tree to find the node that has v as its value.
(2) Whenever the smaller child is selected, add t_i to the LT-set of currently visited node.
(3) If v does not yet exist in the tree, create a new leaf node and initialize it with a value of v, a tuple set containing only t_i and an empty LT-set. Otherwise, simply insert t_i into the tuple set of the node with the value of v.
(4) If necessary, rebalance the tree (described below).

Theorem 4.2. *Let d be the number of distinct values. The time complexity of inserting in a balanced LT-tree is $O(log^2(d))$.*

Proof. To insert a value in a balanced LT-tree, we have to visit $O(log(d))$ nodes, as in a traditional binary tree. The additional insert in the LT-sets is a logarithmic operation in the bit set [15]. □

Deletion. The deletion procedure closely resembles the insertion. There are only two differences to the procedure described above. In Step 2, we *remove* the tuple id from the LT-set instead of inserting it. And in Step 3, we *remove* the node if the tuple set is empty after removing t_i from it. Thus, the time complexity is also $O(log^2(d))$.

Rebalance. To ensure the worst-case guarantees, we might need to rebalance the tree. Rebalancing occurs according to the rules of a traditional Red-black tree [12], rotating the necessary subtrees either left or right. Here, we need to also maintain the LT-sets. We can describe which LT-sets need to be updated based on the example in Figure 1. In the following paragraph, we identify the tree nodes by their values.

First, there is no need to update any LT-sets above \$25 or \$50, as only the order of these nodes changes. If they were the right child of an upper node, this also holds. Second, \$25's LT-set does not need to be updated, as the left subtree of \$25 always remains \$10. Third, \$50's LT-set needs to be updated because the left subtree changes from \$25 ∪ \$10 to ∅ or vice versa. If \$25 has a right subtree, it would become \$50's new left subtree and, thus, part of its LT-set. We can reuse the previous LT-sets to speed up the update. For a right rotation, we subtract \$25's tuple set and LT-set from \$50's LT-set. For a left rotation, we build \$50's LT-set by the union of \$25's tuple set and LT-set.

As shown in [12], we need at most two rotations for an insertion and three rotations for a deletion. Therefore, both procedures modify at most three LT-sets. However, these sets can hold up to all

tuple ids, so the time complexity of the set operations can become linear in the worst-case. The real-world impact is mitigated by our usage of a compressed bit set, which offers fast set operations [15].

Search. The main use case for the LT-tree is to efficiently find all tuples that have smaller values than a given query value. The tree supports those queries by leveraging the LT-sets. We show how to use the index for $>$ and $\geq$ in Section 5.1. When we traverse the tree and select the larger child, we add the LT-set to our result as shown in Algorithm 1. By adding the tuple id set of the matching node (if it exists) to the result, the method can be extended to $\leq$-queries.

Algorithm 1 Searching the LT-tree for all ids of those tuples that have a indexed value that is smaller than the search value

```
   Data: searchValue v
   Return smallerTuples
1: function FindAllSmaller
2:     x ← rootNode, smallerTuples ← ∅
3:     while v ≠ x.value do
4:         if v > x.value then
5:             smallerTuples ← ∪ x.tuple set ∪ x.LT-set
6:             x ← x.rightChild
7:         else x ← x.leftChild
8:     return smallerTuples
```

Theorem 4.3. *Let n be the number of tuples and d the number of distinct values. The number of nodes to visit to search all smaller values in a LT-tree is $O(log(d))$. The worst-case time complexity to generate the set of all tuple ids of the found tuples is $O(n)$.*

Proof. Algorithm 1 visits $O(log(d))$ nodes to search for a specific value because it traverses the tree at most once from the root to a leaf. Notice that the procedure generates the correct result, let x be a node of a LT-tree and v the search value. We start with x as the root of the LT-tree and distinguish three cases.

- $v =$ **value of** x**:** By construction of the tree, all values in the left subtree are smaller than v. Therefore, all tuple ids in the tuple sets of the subtree belong to the result, i.e., the LT-set of x. We union x's LT-set to the result. Additionally, all values in the right subtree are larger than v. Thus, we can abort the search at this point.
- $v <$ **value of** x**:** By construction of the tree, all values in the right subtree are larger than v. Thus, the tuple sets of those nodes and x do not belong in the result. Some values in the left subtree might still be larger than v, so we set x to its left child.
- $v >$ **value of** x**:** By construction, all values in the left subtree are smaller than v. Thus, all tuple ids in the tuple sets of the subtree belong to the result, i.e., the LT-set of x. We union x's LT-set and x's tuple set to the result. Some values in the right subtree might still be smaller than v, so we set x to its right child.

In the worst-case, v is larger than all values in the LT-tree making all tuple ids part of the result. However, every tuple id is added to the result set at most once. By construction, the ids in the tuple sets cannot appear in any sets of the subtree below it. While tuple ids in a LT-set can appear again in the LT-set of its left child, we only add the LT-set if we select the right child. □

5 DETECTION ALGORITHM: WEEVER

The Weever algorithm detects violations of a given set of business rules expressed as DCs. It creates specific indices for each attribute in the DCs. For every change, indices and result sets are updated. The output of Weever consists of all tuple pairs that violate the given DC set. Alternatively, it can restrict the output to all *new* violations for insertions or all *removed* violations for deletions.

Since deletions cannot produce any new violations, we first detail the more complex insert processing. We introduce our method to process a single predicate in Section 5.1, followed by our approach to order multiple predicates in Section 5.2, and handle multiple DCs efficiently in Section 5.3. We present the overall insertion routine in Section 5.4 and a batch-based variant in Section 5.5. Finally, we show how Weever handles deletions in Section 5.6.

5.1 Predicate processing

Processing a single predicate generally requires two steps. First, we access the appropriate index to obtain the operand. Second, the intermediates of all affected DCs are refined using the operand.The *operand* is a set of tuple ids that is used to refine the *intermediate*, i.e., the candidate set of possible violations, as described in Section 3.

To efficiently obtain operands, we create an index for each column present in a predicate: if a column is part of an inequality predicate, we use an LT-tree, and a simple hash map otherwise. The specific operand needed depends on the predicate type. Let $t.A\,\theta\,t'.B$ be a predicate and $newT$ the newly inserted tuple. For equality and non-equality predicates, we select tuples with the same value as $newT$. For inequality predicates, we obtain the tuples with a smaller value than $newT$. This operation is natively supported by the LT-tree (see Section 4.3). For predicates of type $t.A > newT.B$ and $t.A \leq newT.B$, we build the union of the tuple set of smaller tuples and the tuple set of equal tuples.

The intermediate and the operand are internally stored as a compressed bit set [15]. Thus, they allow fast set operations, but also save on space in practice. The operand bit set is independently created for each predicate. By generating a DC-agnostic operand, we can apply this operand to all DCs that share a predicate.

Weever applies the operands to the intermediate based on the predicate type. On the one hand, it builds the intersection of the intermediate and the operand for predicates of the types

$$t.A = newT.B \qquad t.A < newT.B \qquad t.A \leq newT.B$$

For these predicates, the operand contains those tuples that fulfill the predicate. By intersecting with the intermediate, we remove those tuples that cannot form a violation anymore. For example, the predicate $t.distance = t'.distance$ in φ_3 retains only $\{t_0\}$ in the intermediate. On the other hand, Weever performs a set difference between the intermediate and the operand for the predicate types

$$t.A \neq newT.B \qquad t.A > newT.B \qquad t.A \geq newT.B$$

Here, the operand contains exactly those tuples that do *not* fulfill the predicate. Thus, we remove them from the intermediate. For example, the predicate $t.distance \neq t'.distance$ in φ_2 *removes* $\{t_0\}$ from the intermediate in contrast to its equality counterpart.

For single-column predicates, i.e., those of the form $t.A\,\theta\,t'.A$, we can reuse the operand to refine both intermediates. Equality and non-equality predicates are reflexive, so we do not even need

to reapply the operand as long as both intermediates were equal before the predicate. While inequalities require the re-application of the operand, we do not need to access the index again. Every inequality predicate can be mirrored as $newT$ becomes t and t', e.g., the counterpart of $t.A < newT.A$ is $t'.A > newT.A$. Thus, only the application of the operand differs, but not the operand itself.

5.2 Scheduling predicates of an individual DC

The order in which predicates are executed plays an important role in determining performance. In particular, we can abort a predicate pipeline if the intermediate is empty. However, we cannot accurately predict the chance of the intermediate being empty after a predicate. Therefore, we classify the predicates of a DC into the following six groups and devise rules to handle them individually. We execute the classes in the following order.

(1) uni-column equality: $t.A = t'.A$
(2) bi-column equality: $t.A = t'.B$
(3) uni-column non-equality: $t.A \neq t'.A$
(4) bi-column non-equality: $t.A \neq t'.B$
(5) uni-column inequality: $t.A < t'.A$
(6) bi-column inequality: $t.A < t'.B$

Processing inequalities is costlier than all other predicates, as they require more set operations. Thus, predicates (5) and (6) are executed last. Additionally, we take the number of involved columns of a predicate into account. Uni-column predicates (1), (3), and (5) can simultaneously refine both intermediates. Finally, we separate equalities (1) and (2) from non-equalities (3) and (4) for two reasons. First, equalities are more selective for any column with cardinality > 2. Second, the initial intermediate construction is faster for equalities because we can simply copy the operand. For non-equalities, we need to copy all present tuple ids and build the set difference with the operand.

Within each class, the actual predicates are sorted based on two methods. For classes (1) – (4), we use selectivity to sort the predicates. As the index access does not depend on the cardinality of the column, we use the selectivity as a proxy for the chance of the intermediate being empty. Given an equality predicate, the intermediate is empty afterward, iff either the query value is unique or none of the previous members of the intermediate have an equal value to the query value. The more selective a predicate, the more likely the query value is unique or the smaller the operand to compare to the intermediate. Thus, we sort descending by selectivity. In contrast, FACET sorts by selectivity ascending because it results in smaller index sizes [17]. However, we index the column fully, so we are not concerned about the index size. We can reason similarly for non-equality predicates, but they are more selective for lower cardinalities. Since we separated equalities and non-equalities into different classes, we do not have to calculate the selectivity and can use the column cardinalities, which we already track for the index.

For classes (5) and (6), half of all other tuples are smaller on average, independent of the cardinality of the involved columns. Therefore, we sort by cardinality ascending, as the index access is logarithmic in the number of unique values.

Since the indices are updated for every insert, we would need to sort the predicates within each class after each insert. However, two strategies limit the number of times we need to sort. First, we sort only if a change *could* have occurred, i.e., if the cardinality of two columns change their order. Since the cardinality can only change by 1 per inserted tuple, we save the minimum cardinality difference for two adjacent predicates. While it might be low initially, usually column cardinalities drift further apart as more values are inserted. Second, we iterate the predicates before sorting to check whether a change in order really occurred. We observe that cardinalities change order frequently for smaller data sets, but over time we see fewer changes and reduce sort overhead for larger datasets.

5.3 Scheduling multiple DCs

Weever efficiently processes multiple DCs at the same time using the following four steps.

(1) Create an optimal schedule for each DC individually.
(2) Determine an execution order of the DCs.
(3) Identify common subsequences between DCs.
(4) Merge and schedule common prefixes.

First, Weever creates an optimal schedule for each DC as shown in the previous section. We do not aim for a single, optimal schedule of all used predicates, because the priority of a predicate depends on the state of the affected DCs. By keeping the individual schedules, we do not need to reorder the schedule if a DC finishes. We simply execute the next DC. As shown in Section 5.1, the operands to refine the intermediates are independent of the previous state. Thus, they can be applied to an arbitrary number of DCs. If we encounter an already processed predicate in another DC, it is skipped. For the example DCs in Table 2, the individual schedules are depicted in each row of the table.

Table 2: Example of multiple DCs with shared predicates

DC	predicates					
φ_4		p_2		p_4	p_5	p_6
φ_5		p_2	p_3	p_4	p_5	
φ_6	p_1	p_2	p_3			

Second, after creating an optimal schedule *within* each DC, Weever determines the execution order of the DCs as a global ordering of all predicates based on the rules of the previous section. We do not alter the order based on the frequency of a predicate. Neither sorting by frequency nor by the quotient of frequency and selectivity performs better, as we experimentally show in Section 6.3.2. The lower refinement power of less selective predicates offsets the potential gain of refining more DCs at the same time. Given the global order of the predicates, we sort the DCs descending by their start position, i.e., the position of their first predicate in the global order. In our example in Table 2, we execute the DCs in the following order: $\varphi_4, \varphi_5, \varphi_6$. While we execute globally lower ranked predicates first, they are at the beginning of a DC and thus optimal for that DC. In return, we can potentially refine other DCs already. Thereby, we minimize the number of overall processed predicates. For φ_6, it would be optimal to execute p_1 first. However, if we execute p_2 first, which is still optimal for φ_5 and φ_6, we can simultaneously refine all three DCs. Thus, we potentially never have to execute p_1.

Third, Weever identifies common subsequences of predicates in the DCs. As these subsequences represent the same refinement pipeline, we want to avoid the redundant computation. We propose a new method for handling common subsequences because our state-independent operands allow for more flexible schedules than the traditional technique of prefix trees used in [4, 17]. As we do not want to lose optimality of the DC schedules, the common subsequences need to start at the beginning of one DC. However, we neither restrict the position nor forbid other predicates in the other DCs. Simply put, we check if the prefix of a DC is the *subset* of any other DC. Therefore, we call them *common prefixes*. In our example in Table 2, $p_2 \wedge p_4 \wedge p_5$ of φ_4 is part of φ_5, and $p_2 \wedge p_3$ of φ_5 is part of φ_6. A prefix tree only finds an overlap of p_2 in φ_4, φ_5.

Fourth, Weever decides which common prefix to schedule. To share the intermediate of a common prefix, it can be copied to the other DCs after executing its predicates. To allow copying, the intermediate must only be refined by exactly the predicates in the common prefix. Therefore, if we decide to use the start of a DC as a common prefix, other common prefixes cannot refine that DC anymore. Therefore, we define two measures and compare them to decide. First, we calculate the *benefit* of a common prefix, which is defined as the number of intermediate refinements it saves. Second, we count the number of predicates of the respective DC, which can be part of other common prefixes and call that measure the *optimization potential* of a DC. In the example of Table 2, the benefit of the prefix of φ_4 is 2 and that of φ_5 is 1. The optimization potential of φ_4 is 0, while it is 2 for φ_5. Therefore, we schedule only the common prefix of p_2, p_4, p_5 of φ_4.

The decision for one common prefix changes the parameters for other common prefixes, because the respective DC cannot be optimized anymore. Therefore, we sort the potential common prefixes ascending by their start position. The intuition is the following: a potential common prefix at the front of a DC optimizes other DCs, while the back of the respective DC is optimized by other common prefixes. Thus, the benefit of a common prefix cannot be changed by later common prefixes. Then, for each common prefix, we decide whether we want to execute it and update the following common prefixes accordingly. While this entails a quadratic time complexity, the number of potential common prefixes is at most the number of DCs – typically, it is even lower (≤ 5).

In the end, the schedule of Weever to process multiple DCs contains the individual DC schedules in the execution order of the DCs and the common prefixes that shall be executed.

5.4 Insert processing

When inserting a tuple, we need to re-validate all given DCs and detect any new violations. We show the procedure in Algorithm 2. First, we preprocess the inserted tuple to obtain a unique identifier (Line 2) and a unified numeric representation (Line 3). We require a primary key for the change data to uniquely identify tuples. Weever uses a simple hashmap to store the mapping of the key to the id. Additionally, the tuple is transformed to a purely numeric representation to allow for easier handling. To this end, we also use a hashmap to map unique strings to an id. As we do not allow inequalities, we do not need to consider the content of the string and can simply use its id afterward. Second, the validation procedure itself

Algorithm 2 Inserting a tuple

```
   Data: insertedTuple, allIds, allViolations, schedule
   Return: violations
1: function Insert
2:     tId ← CreateNewKey(insertedTuple)
3:     tuple ← CreateNumericTuple(insertedTuple)
4:     newViolations ← ∅
5:     for prefix ← schedule.prefixes do
6:         intermediate ← ProcessAllPredicates(prefix, tuple)
7:         if intermediate = ∅ then
8:             MarkAsFinished(prefix.belongsToDC)
9:         else SetIntermediate(prefix.belongsToDC, intermediate)
10:    for DC ← schedule.DCs do
11:        intermediate ← ProcessAllPredicates(DC, tuple)
12:        if intermediate ≠ ∅ then
13:            allViolations[DC][tId] ← intermediate
14:            newViolations ← ∪ {intermediate}
15:    InsertIntoIndices(tuple)
16:    allIds ← ∪ {tId}
17:    UpdateSchedule(schedule)
18:    return newViolations
```

is executed (Lines 4 – 14). It is composed of the steps detailed in the previous sections. Given the schedule, we execute the prefixes first and transfer their state to the affected DCs, if the intermediate is not already empty. Afterward, the DCs are refined by processing the remaining predicates. In the end, the new violations are collected and stored to allow deleting them later. Third, the values of the inserted tuple are inserted into the indices (Line 15) and the tuple id is added to the set of all ids (Line 16). Fourth, the schedule is updated to ensure the next tuple can be inserted efficiently (Line 17). Finally, any new violations are returned. Alternatively, Weever can also return *all* violations.

5.5 Batch processing

Under batch-processing, our input is not a single update or delete, but rather a set of such changes, affording the potential to more efficiently process them. While batch processing does not change the handling of equality and non-equality predicates using hashmaps, we adapt the procedure for inequality predicates. Recall that the LT-tree stores a set of all tuples with smaller values in each node. Thus, we only need to union a logarithmic number of LT-sets. However, these operations are executed for every inserted tuple. By intelligently sorting the tuples in the indexed column, we can reduce the number of union operations.

Let t_1, t_2 be two tuples that are inserted consecutively, i is the indexed column and s_1 is the set of tuples with a smaller value than t_1, i.e., $s_1 = \{t' \mid t'[i] < t_1[i]\}$, and s_2 is defined similarly for t_2. Now, if $t_1[i] < t_2[i]$, then $s1 \subset s2$. Thus, we need to add only those tuples t' to the operand of t_2, where $t_1[i] < t'[i] < t_2[i]$. Weever sorts the batch by the first inequality predicate, according to the order described in Section 5.2. While we could extend this procedure to all inequality predicates, we would need to store *all* intermediates of the entire batch simultaneously. Thus, we only use the first predicate, as it is also the most likely to be executed.

5.6 Delete processing

Algorithm 3 Deleting a tuple

```
   Data: deletedTuple, indices
   Return: allViolations
1: function DELETE
2:     tId ← GETKEY(deletedTuple)
3:     for violationMap ← allViolations do
4:         DELETEFROMVIOLATIONMAP(violationMap, tId)
5:     DELETEFROMINDICES(CREATENUMERICTUPLE(deletedTuple))
6:     allIds ← − {tId}
7:     schedule ← UPDATESCHEDULE(schedule)
8:     return allViolations
```

Algorithm 3 shows the procedure of processing a deleted tuple in Weever. The handling of deletions does not need a predicate schedule, as it cannot find any new violations. The preprocessing step is equal to the insert handling in Algorithm 2. After obtaining the unique identifier, the violations are removed for each DC (Line 3). Next, the tuple values are deleted from the indices and the id is removed from the set of all ids. As the removal of values can change column cardinalities, we need to check if the schedule needs to be updated. Thereby, we guarantee that the next insert can be efficiently processed. While this is overhead, usually we can skip the expensive scheduling with the strategies described in Section 5.2. Finally, the set of all violations is returned.

5.7 Complexity analysis

The time complexity of the different steps (as depicted in Table 3) in processing a change are clearly dominated by enumerating the violations. In theory, a tuple can form a violating tuple pair with every other tuple, so the number of violations scales linearly and therefore also the enumeration. We mitigate this effect by our usage of a compressed bit set [15], making use of native bitwise operations. The tuple insertion and deletion time complexity is dictated by the LT-tree. At most, a tuple is added or removed from a logarithmic number of nodes. Each update to the compressed bit set [15] has logarithmic complexity, too, because they use a sorted data structure internally. The predicate scheduling does not occur for every change to the dataset, but its worst-case complexity is the sorting of the predicates. The DC scheduling is dominated by the updates of all other common prefixes after the decision for one common prefix. Since there are at most as many prefixes as there are DCs, the worst-case complexity is quadratic in the number of DCs.

Table 3: Worst case complexity for all operations for input size n, number of unique predicates p and constraints c.

Operation	Complexity	Example input sizes Tax dataset (Section 6.2.1)
Violation enumeration	$O(n)$	1 000 000
Tuple insertion or deletion	$O(log^2(n))$	400
Predicate scheduling	$O(p * log(p))$	234
DC scheduling	$O(c^2)$	400

6 EXPERIMENTAL EVALUATION

In this section, we evaluate Weever. We compare it to a DBMS-based solution, Facet and Rapidash, described in Section 6.2. Then, we analyze the index and both the individual DC and multi-DC scheduling techniques in Section 6.3.

6.1 Experimental Setup

All experiments were run on an Ubuntu-based (20.04 LTS) server, equipped with an Intel Xeon E5-2650 processor and 256 GB of RAM. All algorithms are single-threaded, running on Java 11 and reading their input data from CSV files. We set an explicit memory limit of 50 GB. Moreover, we set a time limit of 4 hours, aborting executions that exceeded that time threshold. Unless stated otherwise, all experiments were run on three randomly shuffled permutations of the datasets, and we present the mean of the runs. In a separate experiment, we observed superior runtime for Weever when the dataset was sorted by the columns used in the DC (up to an order of magnitude). The main reason is the improved performance of the compressed bitmap. However, in the incremental scenario, we do not have prior knowledge of the tuples, e.g., a known value range, nor can expect a sorted input.

We use the three datasets of [17] to evaluate Weever, and a subset of the DCs presented in [17]. We excluded some DCs because almost all tuple pairs produce a violation. Nevertheless, the DCs represent a broad set of different characteristics, including all types of predicates and both highly and low selective predicates. A short summary of our datasets and the DCs, as in [17], is shown in Table 4.

We compare Weever with three approaches. First, we created a DBMS-based baseline with PostgreSQL 12.18. We translated each DC into a SQL-based trigger function that counts violations before inserting a new tuple. Thus, the approach also operates on a tuple-by-tuple basis and keeps a violation count for every tuple at the time of insertion. Since the baseline does not maintain the actual violating tuples, it does not support deletes. To improve performance, we create an index for every column used in the respective DC: B+-tree indices for columns appearing in inequality predicates, and hash indices for columns in other predicates. To minimize unnecessary index maintenance, we remove all other indices.

Second, we choose Facet [17] as one traditional static DC violation detection algorithm using the authors' implementation. It does not output the violating tuple pairs, but also counts the number of violations. Third, we use Rapidash, as it outperforms Facet in some cases. While it's main focus is finding the *first* violation for a given DC, it can also detect all violations. As Facet, Rapidash also outputs the number of violations only. We obtained the authors' implementation only for the Tax dataset, as every DC is hard-coded as its own function and the dataset needs specific preprocessing. As both Facet and Rapidash do not support incremental changes, we simply run them for the different dataset sizes.

6.2 Runtime evaluation

In this section, we evaluate Weever on the presented DCs and compare it to the three baselines. The overall results for processing single DCs are presented in Figure 2. For Weever and the DBMS-based baseline, we show the runtime needed to process 1000 individual inserts (vertical axis) for a given number of tuples already

Table 4: Summary of dataset and DC characteristics

	Dataset	Tuples	denial constraint	# Violations
φ_7	Tax	6M	$\neg(t.\text{AreaCode} = t'.\text{AreaCode} \wedge t.\text{Phone} = t'.\text{Phone})$	0
φ_8	Tax	6M	$\neg(t.\text{ZipCode} = t'.\text{ZipCode} \wedge t.\text{City} \neq t'.\text{City})$	0
φ_9	Tax	6M	$\neg(t.\text{State} = t'.\text{State} \wedge t.\text{HasChild} = t'.\text{HasChild} \wedge t.\text{ChildExemp} \neq t'.\text{ChildExemp})$	0
φ_{10}	Tax	6M	$\neg(t.\text{State} = t'.\text{State} \wedge t.\text{Salary} > t'.\text{Salary} \wedge t.\text{Rate} < t'.\text{Rate})$	1438
φ_{11}	Flights	3.6M	$\neg(t.\text{Origin} = t'.\text{Dest} \wedge t.\text{Dest} = t'.\text{Origin} \wedge t.\text{Distance} \neq t'.\text{Distance})$	141 844 328
φ_{12}	Flights	3.6M	$\neg(t.\text{Orgin} = t'.\text{Origin} \wedge t.\text{Dest} = t'.\text{Dest} \wedge t.\text{Flights} > t'.\text{Flights} \wedge t.\text{Passengers} < t'.\text{Passengers})$	193 512 571
φ_{13}	TPC-H	6M	$\neg(t.\text{Customer} = t'.\text{Supplier} \wedge t.\text{Supplier} = t'.\text{Customer})$	1544
φ_{14}	IMDB	2.5M	$\neg(t.\text{Title} = t'.\text{Title} \wedge t.\text{ProductionYear} = t'.\text{ProductionYear} \wedge t.\text{Kind} \neq t'.\text{Kind})$	87 252

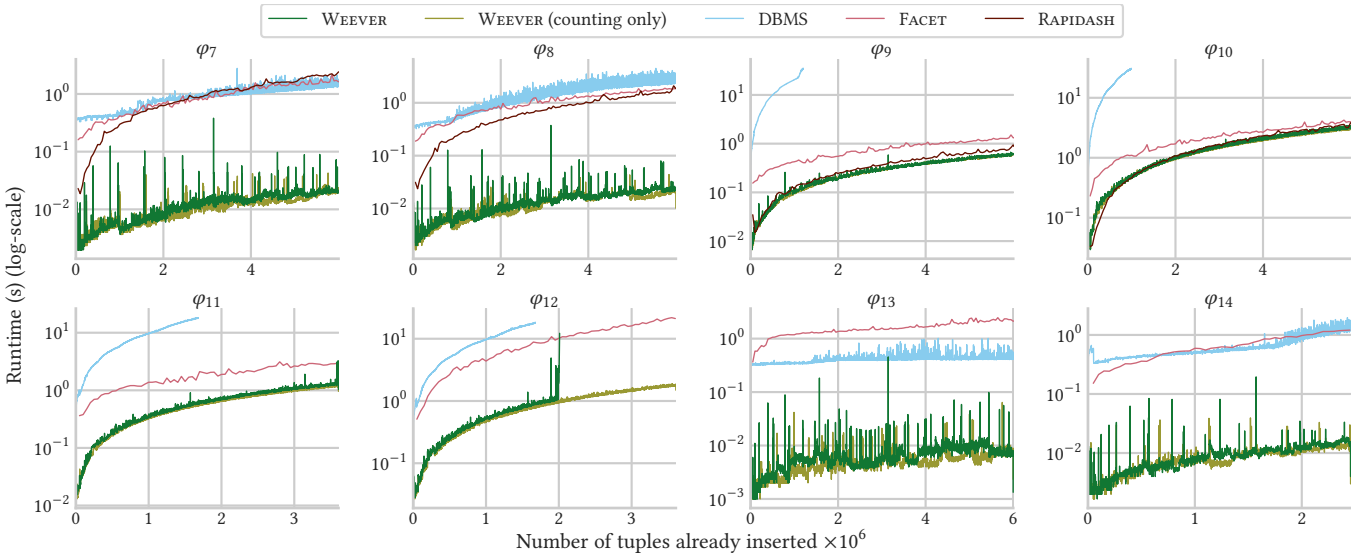


Figure 2: Comparison of the DBMS-based baseline, Facet and Rapidash to Weever

inserted (horizontal axis). For Facet and Rapidash, we simply depict the runtime to process a dataset of the given dataset size. We compare the runtime of the static approaches to the insertion-only performance because processing insertions is costlier. We show that Weever outperforms the other approaches even in this scenario and evaluate the independent deletion performance in Section 6.3.5.

6.2.1 Weever runtime behavior. Before a comparison with competitors, we examine the runtime behavior of Weever itself. First, there are noticeable spikes in the runtime (Figure 2). These are prominent for DCs, whose structure typically leads to quick processing, such as reflexive and highly selective predicates, but occur in every experiment. They arise for two reasons. First, Java garbage collection led to occasional increases in runtime. We tried manually initiating garbage collection after every 1000 inserts for a small sample of the data. While most spikes were thus eliminated, the overall runtime increased vastly. Therefore, we retained the default Java garbage collector for all experiments. Second, other spikes can be attributed to the overhead of occasionally enlarging hash maps. The largest spikes occur when the key map doubles in size. As our incremental method cannot know how many tuples shall be inserted, we cannot size our hash maps from the beginning.

Apart from the occasional spikes, we observe two kinds of scaling trends. For DCs φ_7, φ_8, φ_{13}, and φ_{14}, the runtime grows very slowly and stays low for the entire experiment. In contrast, it grows faster for DCs φ_9, φ_{10}, φ_{11}, and φ_{12}. The main reason for the difference is the size of the operands and the *intermediate*. As presented in Section 4.3, the set operations can comprise all existing tuples. Thus, the time spent on these set operations can grow linearly. We observe that this mostly happens whenever inequality predicates or low-cardinality columns (e.g., binary flags) are part of the DC. Interestingly, the node accesses in the LT-tree present a negligible overhead compared to the set operations. Similarly, the index accesses for the hash indices are always fast, but processing large operands and consequently intermediates due to the low selectivity leads to the linear growth.

Weever cannot finish computing φ_{12} because just storing the violations exceeds the main memory. The large runtime spikes at the end of its runtime are caused by the heavy interference from the garbage collector, as it tries to free memory frequently to allow processing to continue. This presents an extreme case: individual violations do not have a semantic meaning anymore. To process φ_{12}, we ran Weever without storing the violations. Thus, we lose the ability to process deletes and simply count violations as

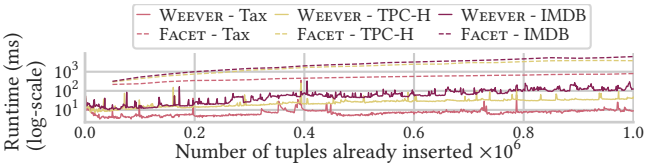
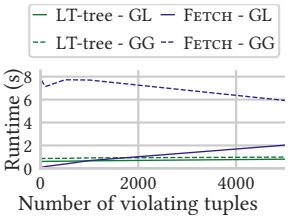


Figure 3: Comparison of Facet and Weever for the Top-20 DCs (ranked by [5]) in the Tax, TPC-H, and IMDB dataset



DC	column	LT-set cardinality	LT-set size
φ_{10}	Salary	5.49 x	3.04 x
φ_{10}	Rate	3.31 x	1.87 x
φ_{12}	Flights	5.41 x	1.64 x
φ_{12}	Passengers	8.65 x	2.73 x
	unique set	9.23 x	2.14 x

Figure 4: Index behavior for increasing number of violations and LT-tree storage consumption overhead, comparing the LT-set to the traditional tuple set for 1 000 000 tuples

the baselines do. We observe no difference in runtime for all DCs, showing the effectiveness of Weever's method to handle violations. Since we do not need to re-identify tuples, we can omit the key mapping when we do not store violations. Consequently, the large runtime spikes of the map resizing were eliminated.

Weever can also process multiple DCs at the same time. Therefore, we created DC sets for the Tax, TPC-H, IMDB dataset. Inspired by [17], we first discovered DCs on a sample of 10 000 tuples. Since there are thousands of DCs, we then ranked them using the method from [5] and finally picked the top 20 DCs. We filtered the DC sets, so Facet can also process them [17]. As depicted in Figure 3, Weever can process these DCs for all three datasets. We observe similar spikes as in the single DC experiment. The runtime scales linearly with the input size for all datasets, but more prominently for the TPC-H and IMDB datasets. This is a result of the picked DCs, as they require more processing time for these datasets.

6.2.2 Comparison to other approaches. In comparison to the other approaches, we observe that Weever is the fastest approach for all DCs (Figure 2). However, the runtime difference varies between DCs. The gap to the runtime of Facet and Rapidash depends on the scaling behavior of Weever. While a sizeable difference exists for the DCs for which Weever's runtime scales slowly, the other DCs are closer in runtime and exhibit a more similar scaling behavior. We can explain the difference by looking at the predicate selectivity. While Weever benefits from processing highly selective predicates because they quickly reduce the intermediate size, Facet benefits from processing less selective predicates. Since Facet can group tuples based on their predicate values, less selective predicates lead to fewer groups, which speeds up computation. Rapidash also benefits from less selective predicates because its internal index structure grows with the number of distinct values.

Additionally, the processing of inequality predicates in a static scenario can make use of the order to save set operations. The overall asymptotic complexity for sorting is the same as querying the LT-tree, but Facet traverses the sorted list and re-uses operands from smaller values to quickly refine the intermediate of larger values. Rapidash also exploits the sort-order and especially, in a low violation scenario like φ_{10}, its index structure works best. It outperforms Facet consistently and for some dataset sizes, it even processes the dataset faster than Weever processes 1000 inserts.

We can also compute the tipping point between Weever and the static approaches by counting the number of inserts Weever can handle within the runtime of a static approach. Simply put, if the violations should be updated more often than the tipping point, it is beneficial to use Weever. In our examples, the tipping point ranges between ~950 (φ_{10}) and ~200 000 (φ_{13}) inserts.

The DBMS baseline shares a closer resemblance to Weever. Both approaches validate each insert individually and compute the violations with the existing dataset. In turn, their scaling behavior is also quite comparable. Especially for DCs with highly selective predicates, the runtime of both Weever and the baseline stay almost constant. While we see fewer spikes in the baseline's runtime, the overall runtime is much higher. We observe that Weever is 20 to 200 times faster, for φ_{11} and φ_{13}, respectively. Additionally, some runs of the DBMS baseline hit our time limit and could not be fully computed. Since it cannot use a specialized index structure for inequality predicates, those DCs are particularly time-consuming. Only Facet supports processing multiple DCs natively. The other competitors can only sequentially validate the DCs. In comparison to Weever, we observe a difference of around two orders of magnitude. The tipping point for all datasets is around 50 000 inserts.

6.3 Analysis of Weever's design

In this section, we evaluate the design decisions of Weever, namely the LT-tree, the predicate ordering for a single DC, the scheduling of multiple DCs together, and the deletion handling.

6.3.1 LT-tree. First, we examine the storage and maintenance cost of the LT-tree. We compare the cardinality and storage size of the LT-sets to the traditional tuple sets, i.e., the usual index content, in Figure 4. The usual cardinality increase factor is below the theoretical value of $O(log(n)) = 20$. For a unique set of values, we roughly match the expected value of 10. However, we also observe the efficiency of the compressed bit sets, as the actual space penalty is lower than the cardinality difference. In practice, the LT-tree for the unique set takes 60 MB (accounting for the values, tuple sets, and LT-sets) in memory. Additionally, we measured the insertion time for the unique set from Figure 4. It scales linearly, as both the tuple set and LT-set do, but the overall performance remains fast. The average insert in the dataset of a million tuples takes ~4.5μs.

Second, we compare our novel LT-tree to the index structure from [19] (see Section 4), called Fetch. We compare to Fetch because it is the only available index structure to incrementally validate inequality predicates. To highlight the strengths and weaknesses of both index structures, we evaluate them on an artificial dataset that consists of two columns of integers. As Fetch is designed to handle *exactly two* inequality predicates at the same time, we use both indices in two scenarios: first, a DC consisting of two greater-than predicates (abbreviated as *GG*), i.e., $\varphi_{GG} = \neg$(t.A > t'.A $\wedge$ t.B > t'.B); second, a DC consisting of one greater-than and one

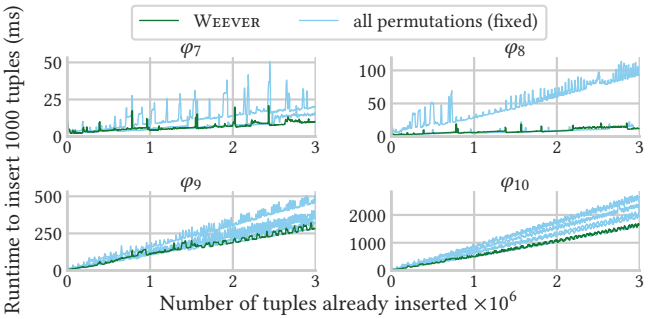


Figure 5: Comparison of the predicate order chosen by Weever to all possible permutations

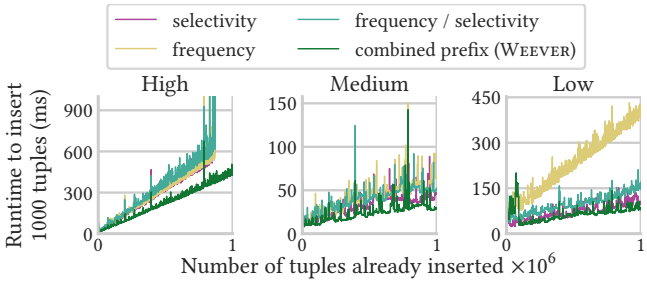


Figure 6: Comparison of the different methods to schedule multiple DCs for different predicate overlap levels

less-than predicate (*GL*), i.e., $\varphi_{GL} = \neg$(t.A > t'.A ∧ t.B < t'.B). A DC consisting of two less-than predicates behaves like the *GG* case.

In our initial setup, both columns contain 10 000 distinct values and the tuples are constructed such that there are no violations. We achieve this by generating the columns as two ascending sequences in the *GL* case and an ascending and a descending sequence in the *GG* case. After constructing the correct tuples, we shuffle the dataset to avoid any side effects due to a sorted input. We investigate the performance of both indices for increasing violating tuples by modifying a fixed number of tuples. For each violating tuple, we change the value in column B to a randomly chosen smaller value in the *GL* case, and a larger value in the *GG* case. Thus, the exact number of violating tuple pairs is random, as in real life scenarios, but every modified tuple produces at least one violating tuple pair. We vary the number of violating tuples between 0 and 5000, i.e., half of the data. Even the high number of violating tuples presents a realistic scenario because these inequality predicates are usually accompanied by different predicates that significantly reduce the number of DC violations. For example, while the number of violations of φ_{10} on a random sample of 10 000 tuples is 0, the number of violating tuples in the inequality predicates is 4800.

We present the overall runtime to process 10 000 inserts for different numbers of pre-existing violations in Figure 4. We observe a very large difference between the *GL* and *GG* scenarios for the Fetch algorithm. It is caused by the requirement to sort both columns at the same time. As this is not possible for the violation-free *GG* DC, it is necessary to create a linear amount of sublists, and, thus, the query time complexity is linear. In contrast, the violation-free *GL* DC requires just one sorted sublist and is very fast to process. However, for increasing violating tuples, we also observe a linear growth, as the number of required sublists increases linearly with the number of violations. Thus, the LT-tree is faster in both scenarios if there are more than 1000 violating tuples. Overall, the runtime of the LT-tree is very stable and is almost constant regardless of the number of violating tuples. The *GL* case is slightly faster because we do not need to perform set operations with the *allIds* set. In general, the set operations make up about 70% of processing time in the LT-tree. Only the remainder of time is spent on traversing the nodes and updating the tree.

6.3.2 Predicate Ordering. To evaluate our approach of ordering the predicates of a single DC, we compare it to all possible permutations. We choose φ_7 to φ_{10}, as they exhibit all the different characteristics: on the one hand, φ_7 represents a key and φ_8 a selective FD, so they are very fast to discover. On the other hand, φ_9 is a less selective FD and φ_{10} contains an inequality predicate, resulting in a slower runtime. The DCs have a low number of violations, so there are opportunities to abort the refinement pipeline early.

We present their runtime for 3 000 000 tuples in Figure 5. The plot shows the rolling average for 10 000 inserts, as the variance for fixed permutations is high. Weever can adapt its predicate order based on the current state to choose the best order for different states. Overall, Weever is either faster than or very close to the fastest permutation in all cases. The experiment indicates that there are sizeable differences between the different permutations. The gap between the permutations is lowest in φ_7, because both equality predicates are efficient to process. Nonetheless, there is a better order and Weever consistently chooses it. Thus, sorting by their selectivity descending really determines the faster schedule. Nearing the end of the experiment, the difference between the two permutations is lower, but Weever outperforms both. In these cases, it can switch between schedules to even further improve performance. For φ_8, we observe a larger discrepancy between the two permutations. The faster schedule processes the equality predicate first. If the non-equality is processed first, the runtime scales worse. This effect is caused by the initial intermediate construction being costlier for non-equalities. Our rule of always prioritizing equality predicates over non-equalities proofs useful here.

The runtime of all predicate permutations of φ_9 and φ_{10} scales linearly. There still are superior permutations for both DCs. The gap between different permutations is larger for φ_{10}, because of the expensive inequality predicates. For φ_9, we observe some points, e.g., at a dataset size of 900 000 tuples, where the fastest permutation outperforms Weever. At these points, the scheduling method causes some overhead, but overall, the scheduling overhead is negligible.

6.3.3 Scheduling multiple DCs. In this experiment, we evaluate our approach to schedule multiple DCs. We compare our method (Section 5.3) to sorting based on different criteria: descending by selectivity only, by frequency only, and by frequency divided by selectivity, as in [4]. Inspired by [17], we evaluate the different approaches on three sets of DCs on the *Tax* dataset with different numbers of shared predicates. First, we handpicked a set of FD candidates that share their left-hand side, i.e., high predicate overlap. Second, we extracted all UCCs on a sample of 10 000 tuples and

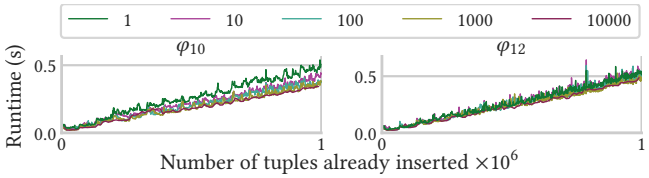


Figure 7: Comparison of the runtime for different batch sizes

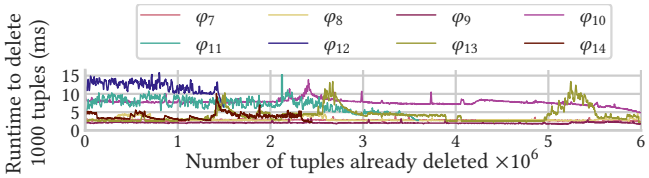


Figure 8: Comparison of the deletion runtime for all DCs

transformed them to DCs, i.e., medium predicate overlap. They share some predicates, as high cardinality columns are typically part of multiple DCs. Third, we use the Top-20 DCs from Section 6.2, i.e., low predicate overlap. They share only a few predicates because these DCs cover a wider variety of predicates.

We present the results in Figure 6 and observe that our method performs best in all three scenarios, with varying improvement. Intuitively, the higher the overlap, the better combining prefixes works. Furthermore, Weever can process more tuples in the high overlap scenario. Due to the otherwise redundant intermediate in multiple FDs, the combined prefix reduces the memory consumption considerably. Additionally, the saved intermediate refinements also yield a performance improvement. In the medium overlap scenario, the gap to other methods is closer, but we outperform them still consistently. In contrast, for a small existing dataset size in the low overlap scenario, the traditional methods perform better. This is to be expected, as there is some overhead to find and combine prefixes. Nevertheless, our method performs best or close to best for larger dataset sizes, even in the low overlap scenario. Except for the low overlap scenario, all other scheduling methods perform quite similar. In the low overlap scenario, the frequency method is by far the slowest. Overall, sorting based only on the selectivity performs best. While we can re-use the operand of a predicate to refine all affected DCs, the overhead for processing *worse* predicates earlier is not offset by the larger number of DCs that can be refined. Other methods, such as [4, 17], can only share refinements if they are at the front predicates of all involved DCs. Therefore, it is worthwhile to favor more common predicates for these methods. Our method can focus on generating the best schedule for each DC and still benefit from shared predicates.

6.3.4 Batch processing. Since batch processing speeds up only inequality predicates, we compare the performance of different batch sizes for 1 000 000 tuples on φ_{10} and φ_{12}. The performance gain depends on the specific DC. For φ_{10}, the performance improves for all batch sizes, while only larger batch sizes benefit in φ_{12}. To improve the throughput and still update the violations more often than the static baselines, we would choose a batch size of 100 for φ_{10}. It improves the runtime by about 25%. Thus, the tipping point from Section 6.2 would improve to ~1200 insertions.

6.3.5 Delete Handling. This final experiment evaluates our method for handling deletions. We delete randomly chosen tuples one by one and report the required processing time for 1000 deletions. We start by deleting from the full dataset and continue until we have removed the entire dataset. The results for all DCs are presented in Figure 8. To improve visual clarity, we show the rolling average of 10 000 tuples, as the variance of the low runtimes is large.

In general, the deletion runtimes are orders of magnitude smaller than the time needed to process inserts. Since deleting tuples cannot introduce new violations, we do not need to consider the predicates and can simply update the index structures. We observe a difference in runtime for DCs that contain inequality predicates. The runtime of both φ_{10} and φ_{12} is worse than the other DCs on the same dataset. For these DCs, we see a decrease in runtime for smaller dataset sizes ($\leq$ 1 500 000 tuples), whereas other DCs present a more constant runtime. The decrease does not span the entire dataset because the number of nodes in the LT-tree depends on the number of distincts.

Besides the general scaling behavior, we observe some runtime spikes. They partially occur for the same reason as in the insertion scenario, namely garbage collection and key map resizing (this time shrinking). However, there are larger runtime spikes in DCs that contain violations, i.e., φ_{10}, φ_{11}, φ_{12}, φ_{13}, and φ_{14}. They are caused by the deletion of actual violations. First, the garbage collector needs to process more data as more objects become irrelevant, namely deleted violations. Second, whenever we delete a tuple that is part of violating tuple pairs, we need to update the data structure for the other tuple. As φ_{11} has large clusters of violations, the processing takes longer. Nevertheless, the absolute runtime is still very low.

7 CONCLUSION

This work presented the *first approach to detect DC violations incrementally*: Weever. Thus, it is no longer necessary to recompute all violations for every dataset change. We introduced the novel LT-tree data structure to process complex inequality predicates. It extends traditional RB-trees by storing additional LT-sets and efficiently allows retrieving all tuples that are smaller than a query value. Weever employs the LT-tree and hash maps to detect violations quickly. In contrast to existing approaches, it outputs the explicit violation tuple id pairs. We propose new scheduling methods for single DCs and introduce the notion of state-independent operands to refine multiple DCs simultaneously. Our evaluation shows that our system efficiently handles both insertions and deletions. Compared to a traditional static approach, Weever updates the violations faster than rediscovering from scratch and outperforms an DBMS-based baseline by a factor of up to 200.

While Weever's performance suffices for most use-cases, it could be enhanced by employing multiple threads or nodes. In our initial experiments, we noticed that incrementally refining the intermediate outperforms simultaneously retrieving operands and combining them all at once. Thus, it is not trivial to use multiple workers. Future research shall extend Weever by adding support for disk storage and specialized index structures to minimize I/O operations. In addition, approaches that relax the limitation of two tuples per DC would be a worthwile and challenging research topic.

REFERENCES

[1] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent query answers in inconsistent databases. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. ACM, New York, NY, USA, 68–79. https://doi.org/10.1145/303976.303983

[2] Jon L. Bentley. 1977. *Algorithms for Klee's rectangle problems.* Technical Report. Carnegie Mellon University.

[3] Leopoldo Bertossi. 2011. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers. https://doi.org/10.1007/978-3-031-01883-1

[4] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient denial constraint discovery with Hydra. *PVLDB* 11, 3 (2017), 311–323. https://doi.org/10.14778/3157794.3157800

[5] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *PVLDB* 6, 13 (2013), 1498–1509. https://doi.org/10.14778/2536258.2536262

[6] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 458–469. https://doi.org/10.1109/ICDE.2013.6544847

[7] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving data quality: consistency and accuracy. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. VLDB Endowment, 315–326.

[8] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)* 33, 2 (2008). https://doi.org/10.1145/1366102.1366103

[9] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. 2012. Incremental Detection of Inconsistencies in Distributed Data. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 318–329. https://doi.org/10.1109/ICDE.2012.82

[10] Wenfei Fan, Chao Tian, Yanghao Wang, and Qiang Yin. 2021. Parallel Discrepancy Detection and Incremental Detection. *PVLDB* 14, 8 (2021), 1351–1364. https://doi.org/10.14778/3457390.3457400

[11] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with Llunatic. *VLDB Journal* 29, 4 (2020), 867–892. https://doi.org/10.1007/s00778-019-00586-5

[12] Leo J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *Proceedings of the Symposium on Foundations of Computer Science (SFCS)*. IEEE, 8–21. https://doi.org/10.1109/SFCS.1978.3

[13] Ihab F. Ilyas and Xu Chu. 2015. Trends in Cleaning Relational Data: Consistency and Deduplication. *Foundations and Trends in Databases* 5, 4 (2015), 281–393. https://doi.org/10.1561/1900000045

[14] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *PVLDB* 8, 13 (2015), 2074–2085. https://doi.org/10.14778/2831360.2831362

[15] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. *Software: Practice and Experience* 46 (2016), 1547 – 1569. https://doi.org/10.1002/spe.2402

[16] Zifan Liu, Shaleen Deep, Anna Fariha, Fotis Psallidas, Ashish Tiwari, and Avrilia Floratou. 2024. Rapidash: Efficient Detection of Constraint Violations. *PVLDB* 17, 8 (2024), 2009–2021. https://doi.org/10.14778/3659437.3659454

[17] Eduardo Pena, Eduardo Almeida, and Felix Naumann. 2022. Fast detection of denial constraint violations. *PVLDB* 15, 4 (2022), 859–871. https://doi.org/10.14778/3503585.3503595

[18] Eduardo H. M. Pena, Edson Ramiro Lucas Filho, Eduardo C. de Almeida, and Felix Naumann. 2020. Efficient Detection of Data Dependency Violations. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. ACM, 1235–1244. https://doi.org/10.1145/3340531.3412062

[19] Chaoqin Qian, Menglu Li, Zijing Tan, Ai Ran, and Shuai Ma. 2023. Incremental discovery of denial constraints. *The VLDB Journal* 32, 6 (2023), 1289–1313. https://doi.org/10.1007/s00778-023-00788-y

[20] G. Ramalingam and Thomas Reps. 1993. A categorized bibliography on incremental computation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 502–510. https://doi.org/10.1145/158511.158710

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