Fast Detection of Denial Constraint Violations

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

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

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

ABSTRACT

The detection of constraint-based errors is a critical task in many data cleaning solutions. Previous works perform the task either using traditional data management systems or using specialized systems that speed up error detection. Unfortunately, both approaches may fail to execute in a reasonable time or even exhaust the available memory in the attempt. To address the main drawbacks of previous approaches, we present the *FAst Constraint-based Error DeTector* (FACET) to detect violations of denial constraints (DCs). FACET uses column sketch information to organize a pipeline of special operators for DC predicates and it implements these operators using a set of efficient algorithms and data structures that adapt to different data characteristics and predicate structures. We evaluate our system on a diverse array of datasets and constraints, showing its robustness and performance gains compared to different types of DBMSs and to a specialized system.

PVLDB Reference Format:
Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. Fast Detection of Denial Constraint Violations. PVLDB, 15(4): 859 - 871, 2022.
doi:10.14778/3503585.3503595

1 INTRODUCTION

The detection of data errors is a routine task in data cleaning. Consider the data shown in Table 1. We can use a few data quality rules that help us maintain or improve the quality of the data. For example, (1) each employee has a unique value of ID; (2) employees cannot supervise their own supervisors (SID); or (3) for any two employees from the same department (Dept), the employee with the most years of service should not earn the lowest salary.

Table 1: The Employee table.

	ID	Name	Dept	StartDate	Salary	SID
t_1	100	C. Gardner	Sales	2012	3000	100
t_2	101	R. Geller	Research	2014	8000	102
t_3	102	D. Brown	Research	2014	6000	101
t_4	103	H. McCoy	Research	2015	8000	101

A traditional way to model data quality rules is by defining integrity constraints, say, a unique constraint for rule (1) and two denial constraints (DCs) for rules (2) and (3), respectively. Recent works on constraint-based data cleaning have used DCs as their constraint language, because of the generality and high expressive power of the formalism [20, 32, 35]. This paper also focuses on DCs. Our goal is to detect data errors as the subsets of tuples in conflict with the semantics of the DCs. Previous works have used SQL queries with a relational DBMS for such detection [15, 16, 19, 32]. In our example, the following SQL query returns the pairs of employees in conflict with the seniority semantics of the third rule:

```
SELECT  t.ID, u.ID
FROM    Employee t, Employee u
WHERE   t.Dept = u.Dept
AND     t.StartDate < u.StartDate
AND     t.Salary < u.Salary
```

Constraint-based queries like the one above require a DBMS to handle predicates of various sorts and costs. For instance, complex inequality predicates on pairwise relationships of tuples are commonplace in data quality rules and translate to expensive non-equi-joins. In these cases, the number of intermediates can quickly become much larger than the actual query result, so the use of inappropriate materialization strategies or inefficient algorithms to process intermediates might result in disastrous performance.

An effective error detection system must provide good performance for the wide range of constraints found in production. As DCs can be easily translated into SQL queries, using the strategies of query optimization and query execution of a DBMS to detect DC violations seems reasonable. In practice, however, the same DBMS that succeed to efficiently evaluate one DC may fail to do so for another, either due to very long runtime or due to excessive memory usage [24, 31]. In such settings, performance is difficult to predict and it can become the bottleneck of a data cleaning pipeline.

There has been prior work on DC-based error detection [6, 31]. In [6], the authors propose predicate evaluation algorithms that, for efficiency, are customized according to the class of each DC predicate. Those ideas inspired our prior work [31], which eliminates the preprocessing overheads found in [6] and presents additional predicate evaluation algorithms. Although these works have shown better performance than DBMSs for error detection, there remain critical aspects that warrant further development.

Three main drawbacks limit the performance and robustness of prior works. First, existing systems use the same low-level representation of intermediate results (e.g., offsets) across their entire processing pipeline. That approach neglects the computation patterns of predicate evaluation and may be suboptimal for some types of predicates. Second, an analysis of the main algorithms for predicate evaluation reveals that each may face great performance degradation depending on the predicate structure or input dataset. Third, the order of predicate evaluations is selected solely based on predicate selectivity: the fraction of tuples that satisfy a predicate. The goal is to reduce the number of processed intermediates so the

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503595

most selective predicates are evaluated first. However, the selectivities are drawn from table samples and may carry large estimation errors, which, in turn, may lead to slower evaluation plans [10, 26].

In response, we develop the FAst Constraint-based Error DeTector (FACET), a robust system to detect all violations of a set of DCs defined for an input dataset. FACET follows the framework of [6, 31], mapping each DC into a pipeline of *refinements*, logical operators whose goal is to efficiently evaluate DC predicates. There are various predicate structures, hence various refinement algorithms. In this paper, we redesign previous algorithms and propose new ones. These algorithms enable the use of hybrid data structures that adapt their storage mechanism to better suit the code pattern of each algorithm. With an extended set of algorithms, FACET has more options to plan error detection and, thus, avoid performance degradation traps. In addition, we propose a novel heuristic optimization algorithm that uses column sketches to select predicate evaluation order. Compared to the sampling approach of previous works, our algorithm offers far better accuracy, which enables FACET to organize pipelines in a more robust manner. Finally, FACET offers different modes of error detection when multiple DCs are given as input. If the DCs share common predicates, FACET can achieve deep reuse of both predicate evaluation and intermediate materialization. In summary, our main contributions are:

- A system that provides fast and robust detection of data errors that are DC violations (Section 4);
- A set of algorithms and optimizations to evaluate the various types of complex predicates of DCs (Section 5);
- A heuristic optimization algorithm to plan predicate evaluation based on column sketches (Section 6);
- An extensive experimental evaluation using synthetic and real-world data, a wide array of constraints, and comparisons with related work and with DBMS-based approaches, showing that our system is consistently faster than the competitors—up to orders of magnitude in many cases (Section 7).

We discuss related work in Section 2, provide background in Section 3, and summarize our conclusions in Section 8.

2 RELATED WORK

Generally speaking, constraint-based data cleaning involves two major steps: error detection and error correction [11]. The latter has been extensively studied in the past two decades, with a large body of work inspired by the seminal paper of Arenas et al. on consistent query answers and data repairing [1]. See recent works on tractability and implementation of error correction [8, 9, 21, 28]; as well as comprehensive discussions on the subject [3, 4].

Several works have used relational DBMSs to detect constraint violations [15, 16, 19, 32]. SQL-based techniques and a commercial DBMS have been used to detect violations of conditional functional dependencies in [15]. HoloClean is a well-known data cleaning system that runs DC-based queries on PostgreSQL for error detection [32]. PostgreSQL has also been used in the Llunatic system to detect violations (as well as to compute repairs) [19]. The same DBMS was used more recently in [16], where entity enhancing rules are translated into SQL queries and user-defined functions in an approach that detects constraint violations and duplicates in a unified process. Given the adoption of DBMS by related tools, we compare FACET to the DBMS approach. To mitigate limitations of any specific DBMS, our evaluation study compares FACET directly to several different DBMSs—each with a different engine (storage, optimizer, physical planner, execution). Our study additionally includes DCs, a dataset, and a DBMS which have not been used in the evaluation of prior works. These assets help us to place the performance of FACET into a wider perspective.

FACET is inspired by several ideas from [6, 31]. Hydra is an algorithm for the discovery of DCs [6]. Although its final goal is not error detection, it requires efficient detection to properly work. For efficiency, Hydra uses a set of refinement algorithms and compact data structures—we discuss these subjects in detail in Section 3. However, Hydra contains expensive preprocessing steps that hinder its performance. In prior work, we propose VioFinder [31], a system that uses the basic framework of Hydra, but introduces new refinement algorithms and eliminates the expensive preprocessing steps. In our experiments, VioFinder serves as our specialized tool baseline, since it is generally much faster than Hydra. As discussed above, there are main differences from FACET to these previous works, namely: hybrid data structures to handle intermediates; new refinement algorithms; a novel scheme to plan predicate evaluation; and different modes of multi-constraint error detection.

3 BACKGROUND

3.1 Representation of constraints

We express constraints using the denial constraint (DC) formalism, as it has a high expressive power, and it generalizes several other relevant types of constraints, including unique constraints, functional dependencies, and order dependencies [4, 14]. The basic idea of DCs is to identify conflicting relationships of combinations of column values with sets of predicates. We consider predicates of the form $\mathtt{p}\colon \mathtt{t.A}\ \mathtt{o}\ \mathtt{t'.B}$, where A, B are columns of a table r with schema R and n tuples; t, t′ is a pair of distinct tuples of r; and $\mathtt{o} \in \{=, \neq, <, \leq, >, \geq\}$ is a set of comparison operators. We can formulate a DC φ with the following notation:

$$\varphi\colon \forall \mathtt{t}, \mathtt{t'} \in \mathtt{r}, \neg(\mathtt{p}_1 \wedge \ldots \wedge \mathtt{p}_m)$$

Each DC specifies a conjunction of predicates that cannot be true for any pair of tuples. In other words, a pair of tuples t, t′ satisfies a DC φ if it evaluates to false for at least one of the predicates of φ. Otherwise, the pair of tuples t and t′ jointly violate the DC φ, which means that the table r is inconsistent with respect to φ.

We can express the data quality rules defined on the Employee table as the three respective DCs (the identifiers t, t′ are omitted from now on):

$$\varphi_1\colon \neg(\mathtt{t.ID} = \mathtt{t'.ID})$$

$$\varphi_2\colon \neg(\mathtt{t.ID} = \mathtt{t'.SID} \wedge \mathtt{t.SID} = \mathtt{t'.ID})$$

$$\varphi_3\colon \neg(\mathtt{t.Dept} = \mathtt{t'.Dept} \wedge \mathtt{t.StartDate} < \mathtt{t'.StartDate} \wedge \mathtt{t.Salary} < \mathtt{t'.Salary})$$

DC violations expose the data errors of a table with respect to the semantics of the data quality rules. For example, we find that tuples t_3 and t_4 jointly violate the DC φ_3 on Employee. The employees in both tuples work in the same department. Since Mr. D. Brown (tuple t_3) was hired first, he should not have a salary lower than that of

Mr. H. McCoy (tuple t_4). As a result, the pair of tuples (t_3, t_4) puts the Employee table in an inconsistent state.

The first step in constraint-based data cleaning is to detect constraint violations [13, 25]. A common next step is to build a conflict representation from the violation set, usually in the form of a hypergraph. The conflict representations support several other tasks, for example, data quality assessment and data repairing [21, 32]. As FACET is precisely designed to output the constraint violations in a dataset, it serves as a natural component for any framework using DCs, or of course, any constraint subsumed by DCs.

3.2 Violation detection using refinements

FACET follows the design of prior works and uses a special operator called *refinement* to evaluate DC predicates [6, 31]. Refinements operate on compact representations of pairs of tuples and they use algorithms that are custom-designed for the different predicate structures. These two properties enable refinements to process different classes of predicates fast while avoiding large intermediates.

Representation of intermediates. Most DCs used in production express pairwise relationships of tuples, so most DC predicates naturally have low selectivity. Processing each pair of tuples individually incur a high interpretation overhead: too many function calls and too many memory allocations to process all the pairs. An efficient alternative is to process compact representations of pairs of tuples [6]. Let tids denote a set of tuple identifiers, or simply tuples when the context is clear. The set of all tuples of a table r with n tuples is given by $\texttt{tids}_\texttt{r} = \{t_1, \ldots, t_n\}$. Ordered pairs $(\texttt{tids}_1, \texttt{tids}_2)$ represent sets of tuples pairs $(\texttt{t}, \texttt{t}')$, such that $\texttt{t} \in \texttt{tids}_1$, $\texttt{t}' \in \texttt{tids}_2$ and $\texttt{t} \neq \texttt{t}'$. For example, the pair of tids $(\{t_1, t_5\}, \{t_1, t_2, t_3\})$ represents the pairs of tuples $\{(t_1, t_2), (t_1, t_3), (t_5, t_1), (t_5, t_2), (t_5, t_3)\}$. In this work, we refer to pairs of tids where $\texttt{tids}_1 = \texttt{tids}_2$ as *reflexive*. The set of all distinct tuple pairs of a table can be expressed as $(\texttt{tids}_\texttt{r}, \texttt{tids}_\texttt{r})$.

Refinement operator. The refinement operator takes as input pairs $(\texttt{tids}_1, \texttt{tids}_2)$ and a predicate p, and it returns the set of pairs $(\texttt{tids}'_1, \texttt{tids}'_2)$ that represent all subsets of pairs of tuples of $(\texttt{tids}_1, \texttt{tids}_2)$ that satisfy p. For clarity, we defer the description of our refinement algorithms to Section 5.

Refinement pipeline. Consider a table r and the refinement of a predicate p_1 followed by the refinement of a predicate p_2. As predicate p_1 is the first in the pipeline, its refinement consumes the pair $(\texttt{tids}_\texttt{r}, \texttt{tids}_\texttt{r})$ to build auxiliary data structures. From these structures, it finds the pairs of tuples that satisfy p_1 and incrementally builds pairs of tids that serve as the input for the next stage (i.e., predicate p_2). Next, in the stage of predicate p_2, the refinement incrementally consumes the two sides of each pair of tids of the previous refinement and builds new auxiliary data structures. The process of finding qualifying tuple pairs for the current predicate is the same as above. Notice that the pair of tids produced at this stage represent tuple pairs that satisfy predicates p_1 and p_2 at the same time, and thus, represent the violations of the DC $\varphi\colon \neg(p_1 \wedge p_2)$. Consider the Employee table and a pipeline with predicates $p_1\colon \texttt{t.Dept} = \texttt{t}'\texttt{.Dept}$ and $p_2\colon \texttt{t.Salary} < \texttt{t}'\texttt{.Salary}$, in this order. The refinement of predicate p_1 produces the pair of tids $(\{t_2, t_3, t_4\}, \{t_2, t_3, t_4\})$. In turn, the refinement of predicate p_2 consumes this pair and produces a new pair of tids $(\{t_3\}, \{t_2, t_4\})$.

Observe that the sides of predicates and the sides of pairs of tids relate to each other. Given a predicate $\texttt{p}\colon \texttt{t.A} \circ \texttt{t}'\texttt{.B}$ and a pair $(\texttt{tids}_1, \texttt{tids}_2)$, the left-hand side t.A of p is for $\texttt{tids}_1$, and the right-hand side t.B of p is for $\texttt{tids}_2$.

4 THE DESIGN OF FACET

Fast(er) refinements. The structure of DC predicates, i.e., the number of columns and the comparison operator, play an important role in the refinement performance. Due to the nature of DC predicates, using an all-purpose refinement algorithm, say a nested loop approach, is inefficient. To avoid such a bottleneck, prior works separate predicates into classes (equalities, non-equalities, and inequalities) and provide a refinement algorithm for each class [6, 31]. However, in a deeper analysis, we have observed several design choices that limit the performance of the existing algorithms. Next, we discuss these choices, as they have guided the design of our refinement algorithms—described in detail in Section 5.

To represent intermediates (i.e., sets of tids), HYDRA uses arrays of integers [6], whereas VioFinder uses compressed bitmaps [31]. The problem is that each work uses the same, fixed representation scheme in all its refinement algorithms. We argue, and demonstrate in our experiments, that the fastest representation actually depends on the computation pattern of each algorithm. The refinements of non-equalities and inequalities need to compute many unions or differences of sets of tids. In this case, compressed bitmaps deliver great performance as they naturally benefit from fast bitwise operations [37]. The refinements of equalities, on the other hand, need to only store and read tids. As such, a simple array of integers avoids the space and decompression overhead generally incurred by compressed bitmaps [36]. For these reasons, FACET uses a hybrid approach, where its refinement algorithms can switch the type of tids representation depending on their computation pattern.

All existing refinement algorithms have a similar two-phase structure: they fetch column values to build auxiliary data structures and then iterate these structures to emit the results. In most algorithms, these structures are simple hash tables, and refinement performance is mostly determined by the time spent in the building phase. For the particular case of equalities and non-equalities on pairs of different columns, existing algorithms build two hash tables, one for each side of the input pair of tids. Then they iterate the entries of one hash table to look up the other one for matches. Our algorithms, on the other hand, follow a traditional hash-join-like approach that requires a single hash table plus a probing routine to emit results. By avoiding the building of the additional hash table, our algorithms perform generally better than prior solutions.

Inequalities are often the most computationally expensive part of the refinement pipeline, so we need to treat them carefully. Both prior works rely on only one algorithm each to handle inequalities [6, 31]. Yet both algorithms have weaknesses that limit their use or slow down execution depending on the input. We propose a third (new) algorithm to help in cases where previous algorithms underperform. We also propose to dynamically choose among the three algorithms based on information from the input (i.e, column cardinalities). We describe all algorithms for inequality refinement in Section 5, but defer the discussion on algorithm selection to

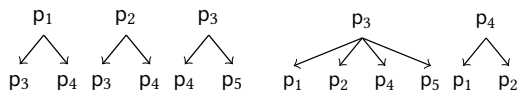
Section 6, where we elaborate on the relationship between column cardinality and predicate evaluation costs.

Robust evaluation plans. In principle, given a DC with m predicates, we could build a refinement pipeline using any of its $m!$ predicate permutations; the errors detected would remain the same. However, different predicates have very different evaluations costs to each other. In addition, FACET offers three algorithms for the refinement of inequalities, each may have different performance depending on the input. Thus, selecting a low-cost evaluation plan is critical, but challenging, as it involves selecting a good predicate order and appropriate algorithms.

The costs of our refinements algorithms (Section 5) are strongly related to predicate structure as well as the number of distinct values for a given set of columns. Unfortunately, prior approaches rely solely on (sampled) predicate selectivity to decide predicate order and do not explore the fact that different predicates have different evaluation costs [6, 31]. In Section 6, we present a novel algorithm that explores the high accuracy of column sketches to select refinement order and algorithms. Our approach mitigates poor evaluation plans, which would have higher chances of being picked if one would use estimates from samples.

Multi-constraint execution. Most previous works perform error detection one constraint at a time [16, 31, 32, 39]. If the input is a set of DCs with any pair of DCs sharing at least one predicate, this strategy results in repeated computations. However, it is possible to use a trie-based scheme to check multiple DCs at a time and alleviate the computation waste [6]. FACET follows such idea to organize the refinements in a trie, in such a way that if two or more DCs share a common refinement path, the intermediate materialization and the predicate processing for that path are also shared.

FACET supports two organizations of predicate tries. Given a set of DCs Φ, it can order the predicates of each DC by (i) the cost of the predicate in each DC (see Section 6), or by (ii) the frequency count of each predicate with respect to all predicates of Φ. Figure 1 illustrates the two organizations. The cost-based approach favors fast predicate processing, whereas the frequency-based approach favors processing reuse. While previous work supports only single-thread executions [6], FACET can evaluate the predicate tries independently of each other and benefit from parallel execution. In this way, FACET can speed up multi-constraint execution even when the input contains DCs having no common predicates.



(a) Cost-based **(b) Frequency-based**

Figure 1: Example of pipeline organizations for a set of DCs $\Phi = \{\neg(p_1 \wedge p_3), \neg(p_1 \wedge p_4), \neg(p_2 \wedge p_3), \neg(p_2 \wedge p_4), \neg(p_3 \wedge p_4), \neg(p_3 \wedge p_5)\}$ and predicates with ascending costs: p_1, p_2, p_3, p_4, p_5.

5 REFINEMENT ALGORITHMS

5.1 Equalities

Equalities are predicates of the form $\mathtt{p}$: $\mathtt{t.A} = \mathtt{t'.B}$. Assume column A as the build side, and column B as the probe side. We iterate the tuples on the left-hand side of the input pair of tids and fetch their values of column A. Since refinement operators produce pairs of tids, we build a hash table that maps each unique value of column A fetched into a pair $(\mathtt{tids}_1, \mathtt{tids}_2)$, where $\mathtt{tids}_1$ contains the set of tuples having that value and $\mathtt{tids}_2$ is empty. In the probing phase, we iterate each tuple on the right-hand side of the input pair of tids and we probe the hash table with the value of column B of that tuple. If we find an entry, we add that tuple into the right-hand side of the pair of tids of that entry ($\mathtt{tids}_2$). Finally, for each entry in the hash table having a pair of tids with at least one pair of distinct tuples, we push that pair to the next pipeline stage. Consider the refinement of the predicate $\mathtt{p}$: $\mathtt{t.SID} = \mathtt{t'.ID}$ and a reflexive pair of tids with all pairs of tuples of Employee. After the probing phase, the hash table contains the entries: $\langle 100, (\{t_1\}, \{t_1\})\rangle$, $\langle 101, (\{t_3, t_4\}, \{t_2\})\rangle$, and $\langle 102, (\{t_2\}, \{t_3\})\rangle$. The first entry can be ignored since it does not represent any pair of distinct tuples.

To decide about the build and probe side, we estimate the cardinality (number of distinct values) of the columns using the fast HyperLogLog sketches [17]. We use the column that has the lowest estimated cardinality as the build side, so we have the fewest possible number of table entries. Notice that if we use column B as the build side, we must reverse the order of the pair of tids.

Avoiding scans in reflexive pairs of tids. Some DCs have predicates or predicate sequences that enable further refinement optimizations. Equalities having the same column on both sides are a good example. Assume a predicate of the form $\mathtt{p}$: $\mathtt{t.A} = \mathtt{t'.A}$ to be the first in the pipeline. We build a hash table that maps each unique value of column A in the table into a reflexive pair $(\mathtt{tids}_1, \mathtt{tids}_1)$, where $\mathtt{tids}_1$ contains the tuples having that value. There is no need for probing the hash table in these cases. We can simply iterate each entry, if it has a pair of tids with at least one pair of distinct tuples (i.e., $\mathtt{tids}_1$ has more than one tuple), we push that pair into the next stage. Also, we use this strategy on sequences of predicates $\mathtt{p}$: $\mathtt{t.A} = \mathtt{t'.A}$ that are right at the beginning of the pipeline. For example, the pair of tids $(\{t_2, t_3, t_4\}, \{t_2, t_3, t_4\})$ represents pairs of employees of the same department and the pair of tids $(\{t_2, t_4\}, \{t_2, t_4\})$ represents pairs of employees of the same department that have the same salary. As the pairs of tids are always reflexive, we can avoid one entire scan of tids. Notice that we can perform this optimization for all the refinements of single-column predicates on reflexive pairs of tids.

5.2 Non-equalities

We refer to non-equalities as predicates of the form $\mathtt{p}$: $\mathtt{t.A} \neq \mathtt{t'.B}$. Similar to the refinement of equalities, we follow a hash-based approach, but with a key difference in how we build the output. Assume column A as the build side once more. Given a pair of tids $(\mathtt{tids}_1, \mathtt{tids}_2)$ as input, we build a hash table the same way we do for equalities on a pair of different columns. Now, assume each entry of this hash table to have a key k associated with a pair of tids $(\mathtt{tids}'_1, \mathtt{tids}'_2)$. An empty $\mathtt{tids}'_2$ means that none of the tuples in

$\texttt{tids}_2$ have a value k of column B matching that value k of column A of the tuples of $\texttt{tids}'_1$. In other words, all tuples of $\texttt{tids}'_1$ have a value of column A that is different from every value of column B of the tuples in $\texttt{tids}_2$. So we simply push the pair $(\texttt{tids}'_1, \texttt{tids}_2)$ to the next refinement. On the other hand, a non-empty $\texttt{tids}'_2$ means that all tuples of $\texttt{tids}'_1$ have a value k of column A that is equal to the k value of column B of the tuples of $\texttt{tids}'_2$. In this case, we calculate the set difference $\texttt{tids}''_2 = \texttt{tids}_2 \setminus \texttt{tids}'_2$. Then, we push the pair $(\texttt{tids}'_1, \texttt{tids}''_2)$ to the next refinement.

Hybrid tids in action. Functional dependencies exemplify a predicate structure that can greatly benefit from our refinement optimizations. Assume a DC φ: $\neg(\texttt{t.StartDate} = \texttt{t}'\texttt{.StartDate} \wedge \texttt{t.Salary} \neq \texttt{t}'\texttt{.Salary})$, or StartDate $\rightarrow$ Salary in functional dependency notation. We first perform the refinement of the equality on StartDate and obtain a reflexive pair $(\texttt{tids}_1, \texttt{tids}_1)$, where $\texttt{tids}_1 = \{\texttt{t}_2, \texttt{t}_3\}$. Through this stage, we store tids as simple arrays of integers, because the current refinement does not need to perform any bitwise operations. The next stage is the refinement of the non-equality on Salary, which benefits from tids reflexivity. In this stage, we fetch the values of salary of the integer-based tuples of $\texttt{tids}_1$ and build a hash table with entries $\langle 6000, \{\texttt{t3}\}\rangle$ and $\langle 8000, \{\texttt{t}_2\}\rangle$. At this stage, however, we use compressed bitmaps to store and operate tids, because of the non-equality requirement for logical operations. Each entry of the non-equality hash table is a value v of salary associated with a subset of $\texttt{tids}_1$: a set $\texttt{tids}'_1$ of tuples that have that value v of salary. Notice that the tuples of $\texttt{tids}'_1$ have a salary value different from the salary values of the tuples of the set difference $\texttt{tids}''_1 = \texttt{tids}_1 \setminus \texttt{tids}'_1$. This operation is implemented with a simple AND NOT. Finally, we push pairs $(\texttt{tids}'_1, \texttt{tids}''_1)$ into the output. For the non-equality on salaries, we first push the pair $(\{\texttt{t}_3\}, \{\texttt{t}_2\})$, and then the pair $(\{\texttt{t}_2\}, \{\texttt{t}_3\})$.

5.3 Inequalities

We refer to inequalities as predicates of the form $\texttt{p}$: $\texttt{t.A}$ o $\texttt{t}'\texttt{.B}$, where operator $\texttt{o} \in \{<, \leq, >, \geq\}$.

IEJoin. This algorithm was proposed in [24] and used in [6] to process the inequality predicates of DCs. The algorithm is designed, and limited, to process two inequalities at a time. First, it builds sorted versions of each side of the input pair of tids. Next, it computes permutation arrays and offset arrays whose contents are relative to the sorted tids. Guided by these arrays, it visits the sorted tuples from the right-hand side of the input pair and marks a bitmap for pairs of tuples that satisfy the second predicate. Finally, it iterates parts of the offset array regarding the left-hand side of the input and checks the marked bits in the bitmap to determine if the current pair of tuples also satisfy the first predicate. To produce pairs of tids we use the following strategy from [6]: while checking the bitmap, we incrementally insert the satisfying pair of tuples into the output pair of tids. If we find tuples producing different matching structures, either in the left-hand side or in the right-hand side of the output, we can push the current pair of tids and start a new iteration for a new output pair. Such strategy reduces the number of pairs of tids IEJoin pushes to further refinements.

The cost of the IEJoin algorithm is usually dominated by the sorting phase for instances that produce a relatively low output. For instances with many qualifying results, however, the dominant part of the algorithm becomes the iteration of offset arrays as well as bitmap scanning. As a result, the algorithm may severely underperform for predicates of low selectivity.

Hash-Sort-Merge (HSM). We refer to HSM as the algorithm for inequalities used in VioFinder [31]. The algorithm follows a sort-merge approach that uses the distinct values of each column as sentinels. Like our other refinement algorithms, it first builds a hash table for each column of the input predicate using the respective side of the input pair of tids. Then, it constructs a sorted set from the keys of each hash table and removes from the tails and heads of these sorted sets those values that cannot form any pair of tids that satisfy the inequality. In the merging phase, the algorithm performs interleaved linear scans of the two sorted sets to find pairs of values that satisfy the predicate. Then, it collects the tids associated with those values into the output pairs of tids. Each new result is built incrementally using the pairs of tids from previous iterations and logical union (OR) operations. Instead of pushing a pair of tids for each pair of matching values, the algorithm checks the pair from the previous iteration and tries to keep the tids associated with the new matching values in that same pair whenever the matching value structure allows it. This strategy not only reduces the number of pairs of tids moving in the pipeline, but also helps to produce denser tids that better benefit from bitmap compression.

The HSM algorithm builds the output incrementally from previous iterations, which alleviates performance slowdowns from predicates of low selectivity. However, the algorithm needs to map each distinct column value into its respective tids. This becomes an issue for predicates on high-cardinality columns. Although bitmap compression reduces the storage and operation complexity of tids [27], it does not reduce the number of logical operations required to incrementally build the resulting pair of tids.

Binning-Hash-Sort-Merge (BHSM). We propose the novel BHSM algorithm to extend HSM to use bitmaps with binning. Binning is a technique that partitions the column values into a number of ranges and uses a bitmap to represent each range. Instead of building hash tables, BHSM builds range maps that map ranges of a column domain into the tids (bitmaps) whose values of that column lies within the ranges. For simplicity of implementation, we use equal-width binning: we retrieve the minimum and maximum value of each column, and we divide that range into equally spaced ranges according to a fixed number of bins. The main flow of BHSM is similar to the HSM case, but the merging phase uses interleaved linear scans of sorted sets of column ranges rather than sorted sets of column values. For each pair of matching ranges, BHSM produces the associated pair of tids. In BHSM, these pairs are usually very dense, as they contain a large fraction of pairs of tuples that satisfy the inequality. This fact helps the algorithm achieve high throughput. In addition, BHSM needs to perform a *candidate check* for the tids related to the matching ranges. To do so, the algorithm simply runs the original HSM algorithm on these tids. In general, the number of tuples the HSM algorithm processes at this point is rather small compared to the total number of tuples in the original input.

As an example, consider all pairs of tuples of the Employee table and the refinement of the predicate $\texttt{p}$: $\texttt{t.StartDate} < \texttt{t}'\texttt{.StartDate}$

using BHSM with two bins. First, we build a range map with two entries: $\langle$[2011,2013), $\{t_1\}\rangle$ and $\langle$[2013,2015], $\{t_2, t_3, t_4\}\rangle$. Since the range [2011,2013) is lower than the range [2013,2015], the interleaved scan of ranges results in the pair $(\{t_1\}, \{t_2, t_3, t_4\})$. Next, we perform the candidate checks for each tids of the range map. Running HSM having $(\{t_1\}, \{t_1\})$ and p as input returns empty, whereas running HSM for $(\{t_2, t_3, t_4\}, \{t_2, t_3, t_4\})$ returns the pair of tids $(\{t_2, t_3\}, \{t_4\})$. Notice that the pairs of tids $(\{t_1\}, \{t_2, t_3, t_4\})$ and $(\{t_2, t_3\}, \{t_4\})$ represent all pairs that satisfy the inequality.

Compared to the HSM algorithm, the BHSM algorithm greatly improves the evaluation of predicates on high cardinality columns. In the first phase of the algorithm, the number of logical operations is always limited due to binning. As for the candidate check phase, it runs over only a part of the domain space. As a result, the column cardinality perceived at this point is smaller than the column cardinality perceived for the original column, so the issue with high cardinality columns is mitigated.

Optimizations. In the case of the refinement of single-column inequalities that consume reflexive pairs of tids, the algorithms iterate only one side of the pairs to build their auxiliary data structures. A second optimization is possible for refinements subsequent to inequality refinements implemented with either HSM or BHSM. During the "merge" phase of these algorithms, many of the output pairs of tids are formed incrementally from previous pairs. As an example, observe the right-hand side of the pair of tids for the refinement of p: t.Salary > t′.Salary with HSM: $(\{t_3\}, \{t_1\}), (\{t_2, t_4\}, \{t_1, t_3\})$. A refinement receiving pairs of tids of this form can cache the data structures built for the right-hand side of each pair. While the relative difference of the right-hand side of previous and current pairs is non-empty, we can build the supporting data structures incrementally. In the example, a refinement receiving the pairs of the refinement on p would build a structure for tuple t_1, then, in the next iteration, incrementally update that structure for tuple t_3.

6 REFINEMENT PLANNING

This section describes a low-overhead heuristic algorithm to select predicate evaluation orders whose expected costs are minimized. It also describes how to select the inequality algorithms that best fit the DC structure and input dataset.

6.1 A cost measure for refinements

A basic cost in most refinement algorithms regards iterating the hash tables entries that maintain tids. As a result, the number of distinct values of a column, hence the number of table entries, impacts the overall refinement performance. Furthermore, a key performance dimension for building and probing hash tables is their spatial size [34]. Large hash tables are less cache-effective due to higher data movement and perform worse than small hash tables that better fit in the cache [7, 40]. To illustrate this intuition, we ran FACET for a single equality p: t.A = t′.A using the two tids storage types: arrays of integers, and compressed bitmaps. We set the number of records to $N = 2^{20}$ for all executions and generated values following a uniform distribution. We increased the number of distinct values of column A in powers of two, and for each factor, we generated one hundred datasets. Figure 2 reports the average refinement runtime and the average size of the hash table built for each factor. As expected, there is a correlation between column cardinalities, hash table sizes, and runtime. Regarding tids representation, performance degradation is more severe for bitmaps, as they incur overhead that soon becomes non-negligible.

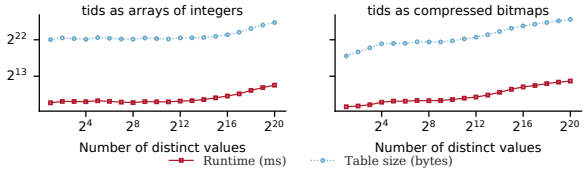


Figure 2: The impact of column cardinality on hash table sizes and equality refinement runtime.

Hash table partitioning is a well-studied way to reduce the penalties of hash tables with a large number of entries [40]. Unfortunately, the partitioning itself may also incur non-negligible overhead and slow down execution [2, 5, 40]. There has been a long debate on hashing schemes [5, 29, 33, 34], which is out of the scope of this paper. But even if we could find the fastest hashing scheme, the code pattern in the refinements remains the same. The higher the column cardinality, the higher the number of entries we iterate to produce the pairs of tids, and hence the higher the amount of computation. For example, in the HSM algorithm for inequalities, the number of logical ORs performed is a direct function of column cardinality. Thus, we use column cardinalities as a proxy for the amount of work a refinement is expected to perform.

Column cardinalities are usually not given, so we estimate them using HyperLogLog sketches [17]. Compared to other estimators, these sketches provide a good trade-off between fast estimation, accuracy, and stability for all cardinality ranges [22]. As we discuss next, we also need to calculate the cardinality of pairs of columns in some cases. We use the estimation framework proposed in [18] to correct estimates on pair of columns from small samples using HyperLogLog sketches on individual columns. As it has been shown in [18], the approach is able to produce highly accurate estimates while keeping a very low estimation overhead.

6.2 Organizing refinements

First, we divide the DC predicates into the classes described in Section 5: equalities, non-equalities, and inequalities. The intuition here is to evaluate the classes of predicates of higher selectivity first. Compared to other predicate classes, equalities usually incur low evaluation costs and have a higher selectivity, so they come first. Consider a predicate p: t.A = t′.A and $f(k)$ as the frequency of a column value k in the domain A, denoted $dom(A)$. For simplicity, assume $1 < f(k) < n$, that is, every value k appears more than once and A is not a single value column. To refine this predicate, we build a hash table for column A and then iterate a total of $|dom(A)|$ entries. This iteration results in $\sum_k f(k)^2$ pairs of tuples. Now, consider we change the operator of p to form a new predicate p′: t.A ≠ t′.A. We can intuitively see the increase in computations, as for each value k we need to perform one set difference computation. By the same token, we see the huge increase in intermediate size compared

to the equality counterpart, as we now have $\sum_k f(k) \cdot (n - f(k))$ pairs of tuples. We use the above intuition to evaluate predicates according to the selectivity signature of their classes: equalities first; then inequalities; and finally non-equalities.

Next, we order predicates within each class. At this point, we use column cardinality information to push down predicates whose refinement is likely faster to compute. The intuition is that predicates within the same class usually have selectivities within the same order of magnitude. Even if we favor predicates of lower selectivity, the amount of computation saved should pay off in the end. We follow general rules within each predicate class. When the predicates of a class consume reflexive pairs of `tids`, single-column predicates have the priority. In this way, more refinements save the scanning of one side of the input pair of `tids`. Otherwise, the priority is predicates on the column with the least estimated cardinality. As a result, more refinements work on smaller hash tables.

Single-column equalities. The higher the cardinality of a column, the more likely single tuples with a unique value for that column are to appear. We might be tempted to push the single-column equalities on high cardinality columns down in an attempt to reduce intermediate sizes—since tuples with unique column values cannot form satisfying pairs of tuples. However, the columns of DC predicates are highly correlated, which might lead to pairs of tuples being carried over a large portion of the pipeline. Furthermore, the strategy above favors evaluating more expensive predicates first. That being said, we still use the ascending order of column cardinalities to order a pair of single-column equalities. However, for more than two single-column equalities, we propose a greedy approach, called GreedyHLL, based on intermediate size and evaluation costs.

Consider a set of predicates $P = \{p_1, \ldots, p_m\}$, where each predicate p_i is a single-column equality on the column A_i. Let $|A_i|$ be the estimated number of distinct values of column A_i. Similarly, let $|A_i, A_j|$ be the estimated number of distinct value combinations of column pairs (A_i, A_j). We use $intermediates = \frac{n - |A_i, A_j|}{n}$ as the ratio of tuples expected to remain after the evaluation of the predicate pair (p_i, p_j). The higher the column pair cardinality, the lesser the number of tuples expected for subsequent refinements. Assuming that $|A_i| \leq |A_j|$, we use $cost = \sqrt[10]{|A_i| + |A_i, A_j|}$ to capture the expected costs of refining predicates p_i then p_j, in this order. As discussed in Section 6.1, the refinement costs increase with the magnitude of the column cardinalities. Thus, for each distinct pair of predicates $p_i, p_j \in P$, $i \neq j$, we compute $rank = intermediates \cdot cost$. The lower the *rank* values, the more likely a refinement is to evaluate cheaper pairs of predicates and output intermediates of smaller size. With the expected costs in place, we select pairs of predicates in ascending order of *rank*. For each selected pair, we place the predicate on the column with the least cardinality first. The algorithm finishes when no predicate pair is left to visit or when all predicates have been included in the predicate order.

Inequalities. We need to choose both the evaluation order and the refinement algorithms for inequalities—we first discuss the latter.

The `IEJoin` algorithm is designed and limited to two predicates at a time. Thus, for a single inequality, the decision stays between `HSM` and `BHSM`. We ran an experiment with `HSM`, as well as `BHSM` (with various numbers of bins), to perform the refinement of an inequality p: $t.A > t'.A$. We used uniformly distributed values and increasing values of column cardinality. Figure 3 (on the left) shows the results of the experiment for a table with 1M rows. We observe that `HSM` processes predicates on columns of low cardinality faster than `BHSM`, which means that the candidate checking overhead of `BHSM` does not pay off for those cases. The situation clearly reverts for predicates on higher cardinality columns. We observe that a threshold of 10,000 distinct values works well for switching between `HSM` and `BHSM`. Also, `BHSM` is stable with a number of bins between 100 and 1,000— similar ranges have been identified in studies on bitmap indexes [37, 38]. We set `BHSM` with 500 bins, as this setting consistently produced most of the fastest executions. We also performed the previous experiment with different table sizes and data following other data distributions, e.g., different Zipfian distributions, and we observed the same trends as above.

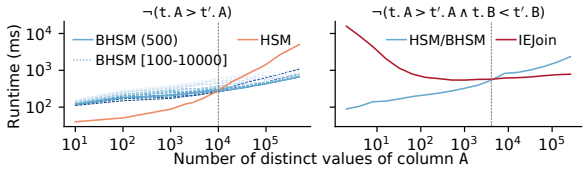


Figure 3: The different impacts of column cardinality on the runtime of `HSM`, `BHSM` and `IEJoin`.

For any pair of inequalities, we can either use a pair of refinements (with either `HSM` or `BHSM`) or a single refinement with `IEJoin`. It turns out that the different approaches complement each other well. We use a DC φ: $\neg(t.A > t'.A \wedge t.B < t'.B)$ to illustrate this behavior. We generate data in a similar way we did for the previous experiment for a single inequality. But now, for each distinct value v of column `A`, we generate a set of values for column `B` such that φ is fully satisfied. We set the cardinality of column `B` as twice the size of the cardinality of column `A`. To add violations into the datasets, we randomly selected 1% of the tuples and increased their values of column `B`. The results on the right of Figure 3 are for a table with 1M rows and shows that `IEJoin` is more stable than `HSM/BHSM` for higher values of column cardinality. In these cases, the sorting-first strategy of `IEJoin` is more efficient than the incremental strategy of `HSM/BHSM`. For smaller column cardinalities, however, the performance of the algorithm is more impaired than the performance of `HSM/BHSM`. In these cases, the dominant part of `IEJoin` is no longer sorting, but offset array iteration and bitmap scanning. The tipping point between `HSM/BHSM` and `IEJoin` occurs around 2^{13} distinct values. We varied the experiment above, with larger dataset scales, and observed that the absolute tipping points discussed for `HSM/BHSM` and `IEJoin` work well in different scenarios.

Given the above, we organize the inequalities of each DC based on the ascending order of the column cardinalities of each inequality, and then we select the refinement algorithms as follows. If the DC contains only a single pair of inequalities, then we check whether `IEJoin` is a viable option. We use `IEJoin` if the predicates of the pair of inequalities include only columns with a cardinality of 2^{13} or higher, and thus, avoid the performance degradation from `IEJoin` on low-cardinality columns. For other cases, we check each inequality to choose between `HSM` or `BHSM` (again, depending on

column cardinality) and build a sequence of HSM or BHSM executions that can straightforwardly benefit from the caching scheme discussed in Section 5. This strategy helps, for example, when the sequence of inequalities has columns of different ranges of cardinality values. The first refinements are usually fast to perform due to their lower-cardinality structure, whereas the further refinements benefit from cached hash tables and, possibly, binning.

We can improve algorithm selection even further when the evaluation pipeline also includes equalities. It turns out that the refinement of the equalities may produce intermediate pairs of tids whose structures enable us to switch the inequality algorithms for better performance. For a moment, consider the DC φ: $\neg(t.A = t'.A \wedge t.B < t'.B \wedge t.C < t'.C)$ and assume that B is a low-cardinality column. Also, assume that the predicate evaluation follows the predicate order as shown. In principle, we would choose HSM for the inequality on column B. However, the actual cardinality perceived by HSM depends on the joint cardinality |A, B|. The higher this value is, the higher the number of entries HSM needs to incrementally iterate to produce the results. Therefore, when equalities are present, we check the estimated cardinality |A, B| to choose between HSM/BHSM and IEJoin. Now, assume C as a high-cardinality column whose predicate would be assigned to BHSM. In this case, we check if the actual number of tuples that reach that far in the pipeline could actually benefit from binning. In our experiments, we observed that the number of tuples for these cases are smaller than the number of bins itself. Thus, we disable binning for these cases as it would incur unnecessary overheads.

7 EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental evaluation. We describe our experimental settings in Section 7.1. Then, we compare FACET to various other systems in Section 7.2. Finally, we evaluate the design decisions of FACET in more depth in Section 7.3.

7.1 Experimental Settings

Table 2 lists the twelve DCs and the four datasets used in our experiments. Most of the DCs have been used in related work [20, 31]. To better investigate our design decisions, we included in our experiments the Flights[1] dataset (containing monthly domestic flight records from 1990 to 2009) and the DCs φ_8, φ_9 and φ_{13}. For example, DC φ_8 helps investigate our algorithm for equality refinement as well as for the hybrid intermediate representation. The set of DCs contains both approximate DCs (with violations) and exact DCs (without violations), ranging from simple key constraints to complex data quality rules. This set represents forms of DCs that are usually seen in production (defined by experts or discovered from data). The datasets contain a mix of numeric, date, and categorical columns, which we classify depending on their domain size, i.e., cardinality values, with intervals [2,1000), [1000,10000), and [10000,|r|] as low, medium, and high, respectively.

We compared FACET to a leading commercial DBMS (DBMS-X) and to three distinct open source DBMSs: PostgreSQL represents the tuple-at-a-time model; MonetDB implements the column-at-a-time model; and DuckDB implements the vectorized model. Both MonetDB and DuckDB run in main memory after the dataset is loaded. Once a dataset was loaded into a DBMS, we indexed all columns mentioned in the DCs. We built B-tree indices with PostgreSQL and DBMS-X; block range indices and adaptive radix tree indices with DuckDB; and column imprints with MonetDB. To improve the query plans, we ran the respective commands of each DBMS to update the statistics on the tables before the executions.

To compare our system to a system that has also been designed for DCs, we used our prior work VioFinder [31]. Notice that VioFinder also uses the refinement pipeline, the pair of tids representation, and specialized algorithms for different predicate classes. Since our evaluation focuses on error-detection performance, we used a SELECT COUNT(*) clause in each DBMS query to return only the number of violations and avoid the materialization costs. By the same token, we set both FACET and VioFinder to count and return the number of violations detected.

FACET and VioFinder were implemented as Java programs that run in main memory after a dataset is loaded. The planning phase of VioFinder uses table samples of 1% and a linear factor of 20 (as suggested in [6]). In FACET, we use table samples of 1% to correct the cardinality estimates on pair of columns (as it can already produce good estimates [18]). Setting sample sizes as above yielded equivalent times for the planning phase of both FACET and VioFinder, which did not disturb the order of the remaining results.

We ran all experiments on an Intel Core i7-7700HQ (2.8 GHz, 4 physical cores/8 logical cores, 32 KB for L1, 256 KB for L2, and 6 MB for shared L3); 16 GB RAM; 256 GB SSD; Ubuntu 20.04; OpenJDK 64-Bit Server JVM 11.0.11; and JVM heap space limited to 8 GB. We terminated any execution that exceeded a four-hour time limit. Unless stated otherwise, we report runtime as the sum of all (but data loading) times involved in the executions, for all systems. For the DBMS approaches, we report indexing time and query execution time separately. Notice that the specialized systems (VioFinder and FACET) do not require traditional column indexing; they build their auxiliary structures on the fly. With the exception of one experiment in Section 7.3, we ran all executions on a single core. Finally, we report the average of five independent executions.

7.2 Comparing to other systems

Figure 4 shows the runtime of running FACET and the DBMS-based approaches for all the evaluated DCs, on three dataset scales (notice the log-scale). The dashed areas in each bar reflect the detection times alone. The results show that FACET is able to perform and scale well for all datasets and DCs. Also, they show that FACET finishes error detection much faster than the DBMSs most of the time (even if we consider only detection times). The speedups are of orders of magnitude for DCs that include inequalities: where FACET takes a few seconds to finish, the DBMSs often take a few hours or reach the time limit. In some cases, the index construction in the DBMSs takes longer than the detection itself. As we increase the number of rows, several DBMS's executions start showing a quadratic increase in runtime. On the other hand, FACET scales linearly with the input size for most DCs.

By contrast, the DBMS baselines produce mixed results. MonetDB was often unable to finish execution because its materialization model drastically increases memory consumption for larger inputs. PostgreSQL was always among the slowest executions (often by

[1] http://www.bts.gov/, last accessed on 12/12/21

Table 2: Datasets summary and denial constraints used in our experiments.

Dataset	Number of rows	Column Cardinalities	DC number	Denial constraint
Tax	10M	Low, High	φ_4	¬(t.AreaCode = t′.AreaCode ∧ t.Phone = t′.Phone)
Tax	10M	Medium, High	φ_5	¬(t.ZipCode = t′.ZipCode ∧ t.City ≠ t′.City)
Tax	10M	Low	φ_6	¬(t.State = t′.State ∧ t.HasChild = t′.HasChild ∧ t.ChildExemp ≠ t′.ChildExemp)
Tax	10M	Low, Medium, High	φ_7	¬(t.State = t′.State ∧ t.Salary > t′.Salary ∧ t.Rate < t′.Rate)
Flights	3.6M	Low, Medium	φ_8	¬(t.Origin = t′.Dest ∧ t.Dest = t′.Origin ∧ t.Distance ≠ t′.Distance)
Flights	3.6M	Low, Medium, High	φ_9	¬(t.Orgin = t′.Origin ∧ t.Dest = t′.Dest ∧ t.Flights > t′.Flights ∧ t.Passengers < t′.Passengers)
TPC-H	6M	Medium, High	φ_{10}	¬(t.Customer = t′.Supplier ∧ t.Supplier = t′.Customer)
TPC-H	6M	Medium	φ_{11}	¬(t.Receiptdate ≥ t′.Shipdate ∧ t.Shipdate ≤ t′.Receiptdate)
TPC-H	6M	Low, High	φ_{12}	¬(t.ExtPrice > t′.ExtPrice ∧ t.Discount < t′.Discount)
TPC-H	6M	Low, High	φ_{13}	¬(t.Qty = t′.Qty ∧ t.Tax = t′.Tax ∧ t.ExtPrice > t′.ExtPrice ∧ t.Discount < t′.Discount)
IMDB	2.5M	Low, High	φ_{14}	¬(t.Title = t′.Title ∧ t.ProductionYear = t′.ProductionYear ∧ t.Kind ≠ t′.Kind)
IMDB	5.8M	Low, High	φ_{15}	¬(t.Title = t′.Title ∧ t.Name = t′.Name ∧ t.CharName = t′.CharName ∧ t.Role = t′.Role)

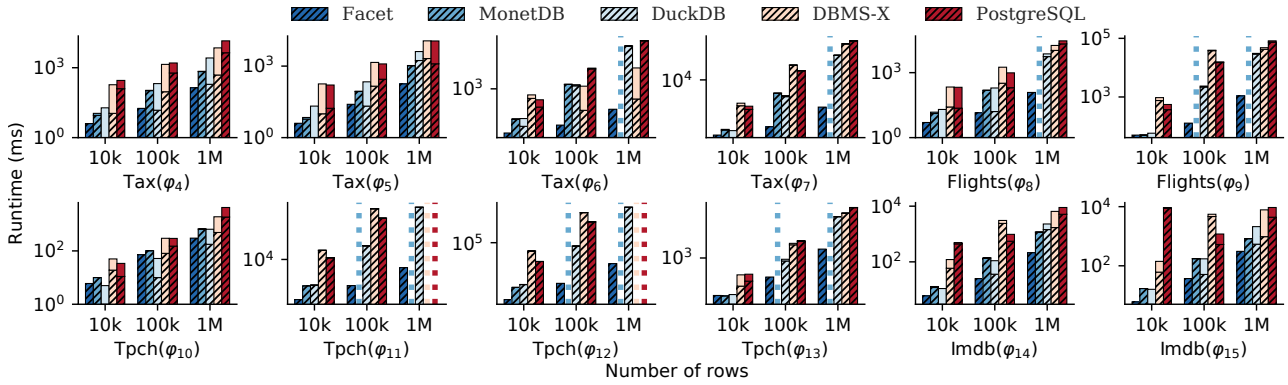


Figure 4: Runtime comparison of FACET to four distinct DBMSs using SQL queries. The dashed areas in each bar represent the violation detection time, whereas the solid areas represent the indexing construction time (only for the DBMSs). The dotted lines are cases where the respective DBMS either exceeded the time limit of four hours or ran out of memory.

orders of magnitude). This might be a side effect of the tuple-at-a-time model on low selectivity predicates, leading to many function calls and high interpretation overheads. DuckDB was the only DBMS able to complete the execution for all DCs within the time limit, and it was often the fastest DBMS option. The fact that DuckDB processes intermediates vector-at-a-time supports the design decision in FACET of processing pairs of tids at a time.

With the exception of PostgreSQL, the DBMSs performed reasonably well for DCs including only equalities. However, the performance of all DBMSs drastically worsens for DCs including non-equalities or inequalities. In broad terms, we observed the following: For equalities and non-equalities, DuckDB and MonetDB (the in-memory DBMSs) use hash join, whereas PostgreSQL and DBMS-X chose between merge join and hash join. All DBMSs fall back to using nested loops for inequalities but push down equalities whenever they are available. This latter decision is a clear strategy to decrease the number of intermediates before evaluating more expensive predicates—which is also FACET's strategy. With regard to indexing, the optimizers of MonetDB and DuckDB never chose indices, while PostgreSQL and DBMS-X used them only seldomly. The quality of the query plans varied not only between DBMSs, but also between database scales. For DCs φ_9, φ_{14} and φ_{15}, DBMS-X opted for different join algorithms between the 100K and the 1M datasets with great impact on the relative response time. Similarly, PostgreSQL used different plans (and varied materialization schemes) for DC φ_{15}, which resulted in poor performance even for the smallest dataset.

In the next experiment, we used GROUP BY clauses, instead of self-joins, to detect violations of functional dependencies (FDs). The approach is similar to [15], but our setting does not require handling constant values. Given an FD φ: $A_1, \ldots, A_m \rightarrow B$ defined for a table r, we obtain the values of columns $A_1, \ldots, A_m$ responsible for the violations of φ with queries of the form:

```
SELECT   DISTINCT t.A1,...,t.Am
FROM     r t
GROUP BY t.A1,...,t.Am
HAVING   COUNT(DISTINCT t.B)>1
```

Then, we use a subquery and the EXISTS operator to obtain the complete tuples that participate in the constraint violations. Figure 5 shows the runtimes of the above approach compared to the runtimes of FACET, for the three FDs φ_5, φ_6 and φ_{14}, on datasets with 1M rows. FACET is fastest, both in terms of detection time and total time. Interestingly, we observe that the performance gain of

FACET over the other systems is higher for DC φ_{14}, which has a few violations (DCs φ_5 and φ_6 have no violations). The GROUP BY approach for FDs may be faster than the self-join approach in some cases, as their respective queries have different structures and, thus, generate different query plans. However, the GROUP BY approach produces only the tuples that participate in the constraint violations; that is, it produces a different result granularity than the self-join approach. We would still need to process the output to generate the pair of tuples used in related data cleaning tools (e.g., [35]).

The query plans for the GROUP BY approach, DuckDB and MonetDB did not use indices and DBMS-X used them only for φ_5. Their query plans are very similar in terms of operations, aggregates and join algorithms. The runtime difference, however, comes from the execution model and the materialization of intermediates that are more efficient in DuckDB and DBMS-X than in MonetDB. In PostgreSQL, all query plans used bitmap index scans. However, the query cost exploded for the DISTINCT clause that is needed to de-duplicate the result from the inner query comparison.

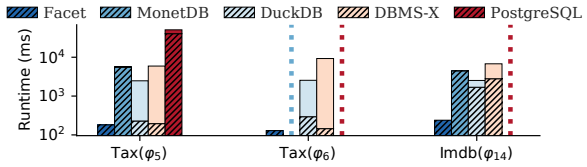


Figure 5: Runtime comparison of FACET to four distinct DBMSs using the GROUP BY approach. The filling patterns of the bars are the same as in Figure 4.

This next experiment compares the performance of FACET with that of VioFinder [31], by varying the number of rows in each table as we show in Figure 6. Unlike the DBMS-based approaches, both systems were able to finish for the full tables, most often in a matter of a few seconds. This fact indicates that the general design of these systems can mitigate many of the issues raised in the DBMS-based approach. Overall, the results show that FACET generally scales better than VioFinder with the number of rows—the main reason varies across DCs. For functional dependencies and keys (φ_4, φ_5, φ_6, φ_{14}, φ_{15}), the hybrid storage of tids and the predicate organization of FACET contribute to speedup factors from 1.31 to 2.17, for the maximum number of rows in each table. For DCs that include two-column equalities (φ_8, φ_{10}), the speedup factors (of up to 2.3) are additionally associated with the more efficient hash-join approach of FACET. Both FACET and VioFinder choose HSM to evaluate the inequalities of the DCs φ_7, φ_{11}, and φ_{13}. The former system is sightly faster for DC φ_7 due to its approach for the equality on that DC, but it renders no relevant improvement for DCs φ_{11}, and φ_{13}. Switching HSM for IEJoin to evaluate the pair of inequalities of the DC φ_9 and enabling BHSM for the ExtPrice inequality of the DC φ_{12} enabled FACET to achieve speedups, for the maximum number of rows in each table, of 3.64 and 14.49, respectively.

7.3 Evaluation of the design decisions of FACET

Given the above results, the next set of experiments are devoted to better investigate the design decisions in FACET and shed light on why it performs better than the other systems.

Figure 7 shows the performance changes from different types of tids storage—the DCs in the plot are representatives that illustrate the different performance impacts on inequalities, non-equalities, and equalities (left to right). The hybrid storage of FACET uses the power of bitmap compression or the simplicity of an array of integers dynamically, and its related runtime is always among the fastest. The compressed bitmaps support much faster executions than arrays of integers when predicate evaluation relies on logical operations (as in DCs φ_7 and φ_8). Otherwise, the additional costs in regard to bitmaps do not pay off, as in the case of DC φ_{15}.

In Figure 8 we compare the use of static inequality algorithms (as used in previous works) to the the adaptivity approach of FACET. The performance of all three inequality algorithms greatly vary depending on the input. All algorithms perform well for DC φ_7, but HSM is the fastest due to the cardinality settings. The joint cardinality of the columns in DC φ_9 is high, so IEJoin is the best option. Finally, DC φ_{12} includes predicates that severely hinders the performance of both IEJoin (due to the low-cardinality Discount column) and HSM (due to the high-cardinality ExtPrice column). In this case, BHSM is the best option, and it becomes orders of magnitude faster than the other two options as the number of rows grows. The adaptivity approach of FACET avoids the worst-case configuration by taking the values of column cardinality and the size of the pairs of tids into account. As a result, its runtime results closely follow that of the best algorithmic option for each input.

Next, we investigate the evaluation planning strategies of FACET. Table 3 illustrates the difference between naive evaluations of predicate pairs in the order shown (the baseline) vs. evaluations in the order chosen by FACET—which is, for these examples, the reverse order of what is shown. Clearly, the largest speedups occur for pairs of predicates that contain an equality combined with either an inequality or a non-equality. Pushing equalities down incurs much smaller intermediates, hence the larger speedups.

Recall that FACET considers predicate selectivity only to sort between different predicate classes and that it uses column sketches to sort within a predicate class. It turns out that predicate selectivity might be too sensitive for predicates within the same class. For example, while the number of intermediates (in pairs of tuples) from the evaluation of $\texttt{t.Salary} > \texttt{t}'\texttt{.Salary}$ is 1.07 times larger than the number of intermediates from the evaluation of $\texttt{t.Rate} < \texttt{t}'\texttt{.Rate}$, the column cardinality of Salary is about twelve times that of Rate.

Previous works use sampling and predicate selectivity to decide the order of predicates within the same class. As a result, errors in selectivity estimation could lead these system to choose poor predicate orders. For example, from a set of ten runs, sampling chose the higher-cost predicate order of the third entry of Table 3 twice. Our sketch-based approach, in turn, consistently chose the cheaper predicate order. To investigate this matter even further and to contrast the effectiveness of sampling ([6, 31]) vs. column sketches, we generated tables with $1M$ rows and a varied number of columns whose cardinality classes are chosen at random—one hundred tables per column number. For each table, we used a DC with inequalities of the form $\texttt{t.A}_i < \texttt{t}'\texttt{.A}_i$, where i varied with the number of columns. We used the true column cardinalities as the ground truth and check it against the order chosen by the estimators. The results in Table 4 show the superiority of the sketch-based approach ("HLL"): it is not impacted by the number of columns,

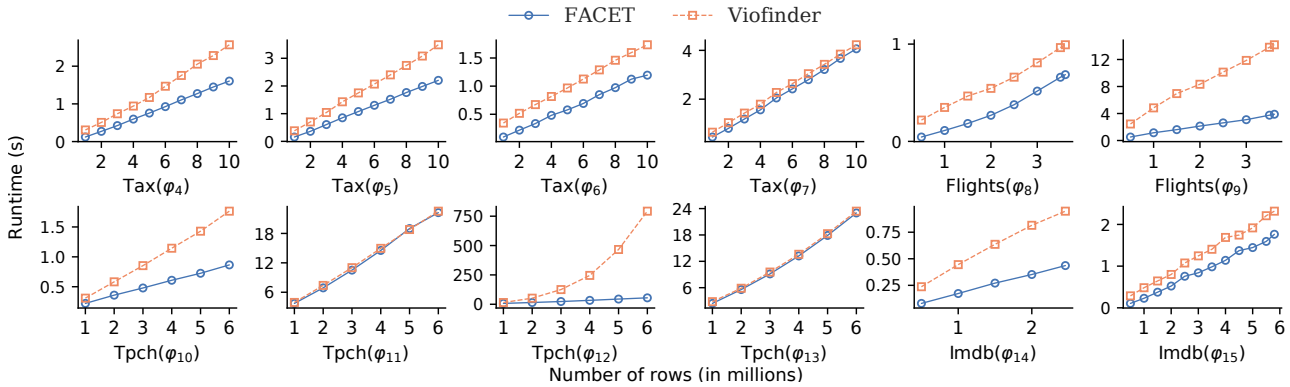


Figure 6: Runtime comparison of FACET to VioFinder.

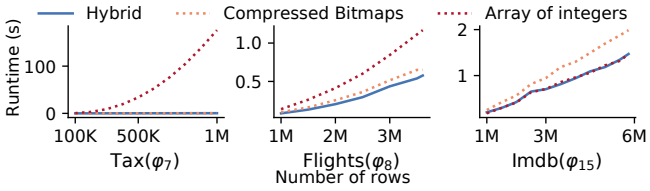


Figure 7: Impact of the types of tids storage on runtime.

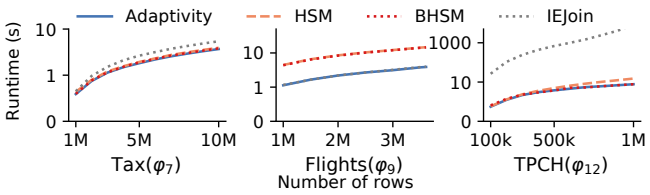


Figure 8: Impact of adaptivity vs. non-adaptivity strategies on the runtime for DCs including inequalities.

and it is fast to compute. Sampling, on the other hand, produces unsatisfactory accuracy even with large samples and for a small number of columns.

Table 3: Runtime speedups of predicate evaluations in the order shown relative to evaluations in the order by FACET.

Predicate evaluation order	Speedup	
	As shown	FACET
t.Phone = t′.Phone ∧ t.AreaCode = t′.AreaCode	1.00	**1.19**
t.Passengers < t′.Passengers ∧ t.Flights > t′.Flights	1.00	**1.22**
t.Salary > t′.Salary ∧ t.Rate < t′.Rate	1.00	**2.71**
t.ExtPrice > t′.ExtPrice ∧ t.Discount < t′.Discount	1.00	**17.02**
t.Flights > t′.Flights ∧ t.Origin = t′.Origin	1.00	**25.78**
t.Flights ≠ t′.Flights ∧ t.Origin = t′.Origin	1.00	**41.36**

The next experiment evaluates our GreedyHLL approach. We focus on uniqueness constraints as they are composed of single-column equalities solely. We ran the DC discovery algorithm of [30] on Tax and extracted all uniqueness constraints from a sample of 100k rows so they would likely be violated when checked over the

Table 4: Accuracy and estimation time comparison of the sketch-based approach vs. the sample-based one.

Number of Columns	Accuracy				Avg. time per table (ms)			
	Sampling			HLL	Sampling			HLL
	0.1%	1%	10%		0.1%	1%	10%	
2	0.48	0.64	0.65	1.0	4	36	356	39
3	0.24	0.29	0.32	1.0	5	48	477	55
4	0.07	0.14	0.11	1.0	6	65	636	75
5	0.03	0.03	0.06	1.0	8	78	758	92

entire table. The DCs have between two to five predicates each. In Figure 9 we show the DCs φ_4, φ_{15}, and the set of discovered constraints (DCs $\varphi_{16} - \varphi_{31}$) as we compare the runtime of FACET using the predicate order (or plans) from GreedyHLL, sampling ([6, 31]), and all other predicate permutations. Notice that runtime is not as affected by estimation errors as it is in settings containing inequalities. Still, the results illustrate how GreedyHLL is robust for constraints having a variety of predicate numbers. For the DCs having five predicates (120 possible plans), the worst plans produce executions that are nearly three times slower than the executions with GreedyHLL. The plans from GreedyHLL consistently produced the fastest (or close to the fastest) executions, whereas the plans from sampling produced the slowest executions for a few DCs.

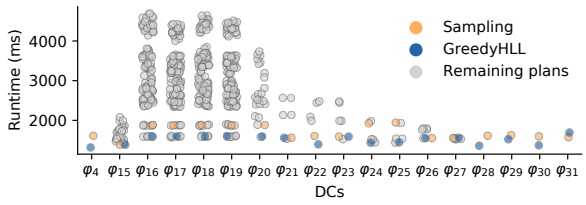


Figure 9: Runtime of FACET using the plans of GreedyHLL vs. runtime of FACET using all other possible plans.

The next experiment compares the execution options that FACET offers when multiple DCs are given as input. We used the uniqueness constraints described previously, which share many predicates among each other. Also, we used a subset of the DCs discovered

on Tax (sample with 100k rows). Since the number of discovered DCs was too large for human verification, we ranked the results with the scoring system described in [12] and picked the top-20 results—these DCs share just a few predicates. FACET can check each DC sequentially ("Sequential"), or each DC on a different thread ("Parallel"—up to eight threads in our environment). Alternatively, it can use two trie-based schemes, where predicates are sorted by cost ("CostTrie") or frequency ("FreqTrie"). Both trie-based modes can run in parallel (each root on a thread).

Figure 10 shows the results for a varied number of DCs. We ran each setting ten times using DCs taken at random for each DC number and report the average runtime of the runs—we used tables with 1M rows in this experiment. Clearly, enabling the multi-DC execution can improve the overall runtime even further. For example, Sequential was nearly three times slower than CostTrie Parallel for the set of uniqueness constraints. The performance of FreqTrie and CostTrie were generally similar: the former pushes down frequent, but likely more expensive predicates, whereas the latter follows the cheaper predicates first rule, which might decrease predicate evaluation reuse. The trie-based approaches achieve higher speedups for the set of uniqueness constraints—since they found many shared predicates, they reused many more predicate evaluations. For the top-20 DCs, on the other hand, there was not much space for reusing predicate evaluations and the speedups come from the independent (parallel) evaluation of each DC (or path in the trie-based approaches). Currently, FACET supports only inter-DC parallelism. Extending the refinement algorithms in Section 5 to benefit from intra-DC parallelism is a topic of future work.

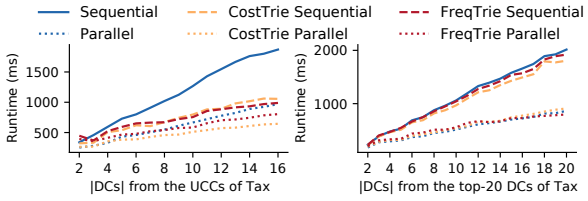


Figure 10: Performance of the multi-DC execution modes.

Figure 11 shows the runtime breakdown of FACET for each DC, on 1M rows. The time spent in each major part of the execution is tracked and plotted as a percentage of the entire runtime. Loading refers to the time spent to load the data file into main memory; planning regards the time spent on cardinality estimation, sampling, and predicate organization; and execution refers to the time spent in detecting violations. Mind that each DC reaches as many stages as its predicate number and that the pair of inequalities of DC φ_9 is handled in a single stage by the IEJoin algorithm. In general, the biggest execution costs were either data loading or evaluating expensive predicates. An interesting execution contrast from FACET planning can be seen, for example, between the stage bars of DCs φ_4 and φ_{15}. For φ_4, FACET pushes down the predicate on the low-cardinality column which leaves the "heavy" work for the second stage. In turn, the system opts for pushing down the equality pair with the highest selectivity in DC φ_{15}. In this case, the initial stages handle high-cardinality columns, eliminate most of the intermediates in the way, and thus, they greatly reduce the work for further stages.

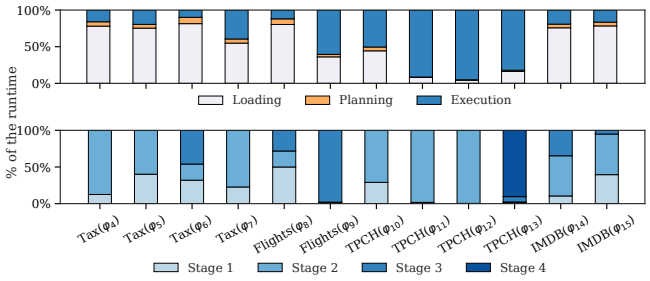


Figure 11: Runtime breakdown of FACET.

Our final experiment shows the heap size required to successfully execute FACET for each DC, on 1M records. To obtain this requirement, we ran FACET with a maximum heap size of 32 MB and continuously double this value until the system is able to process the given DC. We also measured the amount of memory used to store the dataset in memory. The results in Figure 12 show that FACET generally requires a low amount of memory to succeed. The requirement increases for DCs having a very high number of violations, e.g., φ_{11} and φ_{12}, or for DCs over columns of large domains and wide column values (e.g., the strings from φ_{15}). Finally, we observe that binning, as in φ_{12}, and that equality pushdown, as in φ_{13}, have a great potential to reduce memory footprint in general.

Figure 12: Memory requirement of FACET.

8 CONCLUDING REMARKS

In this paper, we presented the FACET system to address the challenges of detecting data errors as DC violations. We showed how column sketches can be a powerful tool in the organization of predicate evaluation and minimize the amount and cost of intermediate processing. We introduced new refinement algorithms and showed that switching between the available inequality algorithms can avoid occasions of suboptimal performance. Also, we proposed hybrid data structures as a better fit for the different patterns in each refinement algorithm. All these ideas are integrated into a system that is able to process multiple DCs at a time with robust and fast error detection performance.

FACET naturally integrates with constraint-based data cleaning pipelines. An extension for future work is the integration of FACET with other use-cases, for example: in query processing by adding cleaning operators into the query plans to track and repair DC violations on demand [20]; or in top-k query answering by counting the DC violations to rank the answering tuples [23].

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)*. 68–79. https://doi.org/10.1145/303976.303983

[2] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition,or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. https://doi.org/10.1145/3448016.3452831

[3] Leopoldo Bertossi. 2011. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers.

[4] Leopoldo Bertossi. 2019. Database Repairs and Consistent Query Answering: Origins and Further Developments. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 48–58. https://doi.org/10.1145/3294052.3322190

[5] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 37–48. https://doi.org/10.1145/1989323.1989328

[6] 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

[7] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for in-Memory Data-Intensive Computing. In *Proceedings of the USENIX Annual Technical Conference*. 281–294.

[8] Marco Calautti, Marco Console, and Andreas Pieris. 2019. Counting Database Repairs under Primary Keys Revisited. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 104–118. https://doi.org/10.1145/3294052.3319703

[9] Marco Calautti, Leonid Libkin, and Andreas Pieris. 2018. An Operational Approach to Consistent Query Answering. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 239–251. https://doi.org/10.1145/3196959.3196966

[10] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 2000. Towards Estimation Error Guarantees for Distinct Values. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 268–279. https://doi.org/10.1145/335168.335230

[11] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2201–2206. https://doi.org/10.1145/2882903.2912574

[12] 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

[13] 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)*. 458–469. https://doi.org/10.1109/ICDE.2013.6544847

[14] Wenfei Fan. 2015. Data Quality: From Theory to Practice. *SIGMOD Record* 44, 3 (2015), 7–18. https://doi.org/10.1145/2854006.2854008

[15] 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), 6:1–6:48. https://doi.org/10.1145/1366102.1366103

[16] 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

[17] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* (2007), 137–156.

[18] Michael J. Freitag and Thomas Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[19] 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

[20] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 805–815. https://doi.org/10.1145/3318464.3389775

[21] Amir Gilad, Daniel Deutch, and Sudeepa Roy. 2020. On Multiple Semantics for Declarative Database Repairs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 817–831. https://doi.org/10.1145/3318464.3389721

[22] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *PVLDB* 11, 4 (2017), 499–512. https://doi.org/10.1145/3186728.3164145

[23] Ousmane Issa, Angela Bonifati, and Farouk Toumani. 2020. Evaluating Top-k Queries with Inconsistency Degrees. *PVLDB* 13, 12 (2020), 2146–2158. https://doi.org/10.14778/3407790.3407815

[24] 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

[25] Solmaz Kolahi and Laks V. S. Lakshmanan. 2009. On Approximating Optimum Repairs for Functional Dependency Violations. In *Proceedings of the International Conference on Database Theory (ICDT)*. ACM, 53–62. https://doi.org/10.1145/1514894.1514901

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[27] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently Faster and Smaller Compressed Bitmaps with Roaring. *Softw. Pract. Exper.* 46, 11 (2016), 1547–1569. https://doi.org/10.1002/spe.2402

[28] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *PVLDB* 13, 11 (2020), 1948–1961. https://doi.org/10.14778/3407790.3407801

[29] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1123–1136. https://doi.org/10.1145/2723372.2747644

[30] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278. https://doi.org/10.14778/3368289.3368293

[31] Eduardo H. M. Pena, Edson Ramiro Lucas Filho, Eduardo Cunha 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

[32] 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

[33] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *PVLDB* 9, 3 (2015), 96–107. https://doi.org/10.14778/2850583.2850585

[34] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1961–1976. https://doi.org/10.1145/2882903.2882917

[35] Yu Sun and Shaoxu Song. 2021. From Minimum Change to Maximum Density: On S-Repair under Integrity Constraints. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 1943–1948. https://doi.org/10.1109/ICDE51399.2021.00181

[36] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 993–1008. https://doi.org/10.1145/3035918.3064007

[37] Kesheng Wu, Ekow Otoo, and Arie Shoshani. 2004. On the Performance of Bitmap Indices for High Cardinality Attributes. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 24–35. https://doi.org/10.1016/B978-012088469-8.50006-1

[38] Kesheng Wu, Kurt Stockinger, and Arie Shoshani. 2008. Breaking the Curse of Cardinality on Bitmap Indexes. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 348–365. https://doi.org/10.1007/978-3-540-69497-7_23

[39] Jing Nathan Yan, Oliver Schulte, MoHan Zhang, Jiannan Wang, and Reynold Cheng. 2020. SCODED: Statistical Constraint Oriented Data Error Detection. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 845–860. https://doi.org/10.1145/3318464.3380568

[40] Marcin Zukowski, Sándor Héman, and Peter Boncz. 2006. Architecture-Conscious Hashing. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*. ACM, 6–es. https://doi.org/10.1145/1140402.1140410