Literature Review for Fast Detection of Denial Constraint Violations

Team Members:

Name	Hawk ID
Patil Sagar Shekhargouda	spatil59@hawk.iit.edu
Pandya Marut	mpandya6@hawk.iit.edu
Nikam Yuvraj R.	ynikam@hawk.iit.edu

Abstract:

The detection of constraint-based violations is a very important task in many data cleaning solutions. There are many works which were proposed to perform this task but failed to execute within a reasonable time or due to memory restrictions. To address this, FACET is introduced which uses a method called “Column Sketching” to organize which helps to check the Denial Constraints (DCs) quickly and efficiently. In addition, it also makes use of different algorithms and data structures which makes FACET very effective in finding the DCs in various situations. The author evaluated FACET on a variety of datasets and constraints to demonstrate its robustness and performance gains compared to others.

1. INTRODUCTION

The author here takes an example where the author defines constraints.

1). Each employee has a unique value of ID.

2). Employees cannot supervise their own supervisors (SID)

3). For any two employees from the same department (Dept), the employee with the most years of service should not earn the lowest salary.

Then the author explains the traditional way where we can achieve 1). by making use of the unique constraint rule, 2). and 3). by defining the one DC for each. Then the author takes an example of SQL query which returns the pairs of employees in conflict with rule 3).

```
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 queries like the above may involve many sorts and might become costly in the case if the data is more or if there are many non-equi-joins. And in these cases, the intermediates can quickly become larger than the actual query result, and making use of these kinds of inefficient algorithms will badly degrade the performance.

The author then defines an effective error detection system as one which provides good performance for a wide range of constraints. This can be come by translating the DCs into the SQL queries using query optimization, but DBMS may fail on one of the DCs due to longer runtime or due to excessive memory usage. All these causes difficulty in predicting the performance of data cleaning pipelines.

Three drawbacks of previously proposed systems:
1). Existing systems use the same low-level representation of intermediate results.
2). Analysis of the main algorithm reveals that it may face severe performance degradation depending on the dataset and predicate structure.
3). The order of predicate evaluations i.e the most selective predicates are evaluated first but the selectivity of these most selective predicates is done from the sample table samples. This may lead to slower evaluations plans and estimation errors.

To overcome all these, the author introduces FACET which follows a framework of mapping each DC into a pipeline of refinements, logical operators which make sure the DCs are evaluated in an efficient manner. The author proposes various new algorithms which make use of the hybrid data structures that adapt their storage mechanism. With these new algorithms, FACET has more options to plan error detection and avoid performance degradation traps.

In addition, the author also proposes a new algorithm that makes use of column sketches which helps FACET to determine the predicate evaluation order. FACET also provides different modes of error detection when multiple DCs are given as input i.e. if the DCs share common predicates then FACET can achieve reuse of both predicate evaluation and intermediate materialization.

2. RELATED WORK

The author talks about different approaches which were used to detect DC violations.

Constraint based Data Cleaning involves two steps: i). Error Detection

ii). Error Correction

Many works have made use of relational DBMSs to detect constraint violations.
HOLOCLEAN was used to run DC-based queries on POSTGRESQL for error detection. PostgreSQL is also used in the Lunatic system to detect violations.
The author compares the FACET to the DBMS approach which is more commonly used, and author compares the evaluation where author compares FACET directly with different DBMSs (each with a different engine).

HYDRA:

The author of FACET was inspired by many ideas like HYDRA (an algorithm for discovery of DCs). Although Hydra's final goal is not error detection, it requires efficient detection to properly work.
For efficient detection, Hydra uses a set of algos and data structures, but Hydra contains expensive pre-processing which brings down its performance.

VioFInder: It's a tool previously designed by author, and it is much faster as compared to HYDRA.

3. BACKGROUND

3.1 Representation of Constraints

The basic idea of DCs is to identify conflicting relationships of combinations of column values with the set of predicates.
p: t.A o t'.B
A and B- columns of table r
n- tuples
t,t'- distinct tuples of the relation r
o- set of comparison operators (=, <,>,<=,>=,!=)

DC psi is formulated as: psi: ForAll t,t' E r, not(p1/\...../\pm)
Each DCs specifies a conjunction of predicates which should not be true for any pair of tuples. In other words, a pair of tuples t,t' satisfies a DC if it evaluates to false for one of the predicates otherwise the tuples t,t' violate the DC which means the table r is inconsistent with respect to DC Psi.

Author then describes for the Steps of Constraint based data cleaning:
Step 1: Detect the constraint violations.
Step2: Build a conflict representation from the violation set (in the form of hypergraph)

The conflict representation even supports other tasks like data quality assessment and data repairing.

The author also then explains that FACET is precisely designed to o/p the DC violations in a dataset, and it can also be used as a component for any framework which is using DCs.

3.2 Violation detection using refinements.

FACET follows the design of previous proposed works and uses a special operator called **refinement** to evaluate the DC predicates.
Refinements operate on representations of the pairs of tuples, and they also make use of an algorithm (custom designed) for different predicate structures. These help refinements to process different classes of predicates in a faster way by avoiding the large intermediates.

Representation of intermediates
Most DCs in production express pairwise relationships of tuples. (low selectivity).
Processing each pair of tuples individually will incur overhead. (too many function calls, memory allocations)
An efficient way to do the above is to process the compact representation of pairs of tuples.
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 are given by tidsr = {t1, . . . , tn}.
Ordered pairs (tids1, tids2) represent sets of tuples pairs (t, t'),
such that t ∈ tids1, t' ∈ tids2 and t ≠ t'. For example, the
pair of tids ({t1, t5}, {t1, t2, t3}) represents the pairs of tuples.
{(t1, t2), (t1, t3), (t5, t1), (t5, t2), (t5, t3)}. In this work, we refer
to pairs of tids where tids1 = tids2 as **reflexive.**

Refinement operator: The refinement operator is the one which takes i/p pairs (tids1, tids2) and a predicate p, and it returns the set of pairs (tids'1, tids'2) and these returned pairs represent all subsets of pairs of tuples (tids1, tids2) which satisfy p.

Refinement pipeline: Here the author explains how the refinement operator works when there are multiple predicates which are in the pipeline.

Consider table r and the refinement of a predicate p1 followed by the refinement of a predicate p2. As predicate p1 is the first in the pipeline, its refinement consumes the pair (tidsr, tidsr) to build auxiliary data structures. From these structures, it finds the pairs of tuples that satisfy p1

and incrementally builds pairs of tids that serve as the input for the next stage (i.e., predicate p2). Next, in the stage of predicate p2, 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 p1 and p2 at the same time, and thus, represent the violations of the DC φ : ¬(p1 ∧p2). Consider the Employee table and a pipeline with predicates p1 : t.Dept = t ′ .Dept and p2 : t.Salary < t ′ .Salary, in this order. The refinement of predicate p1 produces the pair of tids ({t2, t3, t4}, {t2, t3, t4}). In turn, the refinement of predicate p2 consumes this pair and produces a new pair of tids ({t3}, {t2, t4})

4. THE DESIGN OF FACET

Fast(er) refinements: Here the author talks about how to improve refinement performance.
The author mentions that using an all-purpose refinement algorithm for all the scenarios makes it inefficient.
To avoid this bottleneck, the author says that previous works separate predicates into different classes i). Equalities

ii). Non-equalities

iii). Inequalities

And each class has its own refinement algorithm.

The author also takes an example of HYDRA which uses arrays of integers to represent the intermediates and VioFinder uses compressed bitmaps to represent intermediates. The author says that in the above two examples each of them uses only array or compressed bitmaps to represent intermediates which would result in performance degradation.
The author further did in-depth analysis and found that the refinements of equalities need to only store and read tids. So, in this case arrays would be enough.
On the other hand, in the case of refinement of non-equalities and inequalities it is necessary to compute many unions or differences of sets of tids and in this case compressed bitmaps are faster and a good choice.
For these reasons, the author tells us that FACET uses a hybrid approach where its refinement algorithm can switch the type of tids representation depending on their computation pattern.

The author has further discussed the two-phase structure of the refinement algorithms.
In the first phase, the refinement algorithms fetch column values to build the auxiliary data structures, then in the second phase iterate the data structures to emit the results.

The author says that most of the refinement algorithms make use of hash tables as data structure and the performance of these refinement algorithms is dependent on the time spent on building phase.
The author gives an example of the case of equality and non-equalities on the pairs of different columns. Here the existing algorithms creates two separate hash tables for both and then compare both hash table and then emit the results whereas FACET makes refinement algorithms which makes use of hash-join-like approach where all the information is put into the single hash table then find the using a different technique and by doing this it saves time and improves performance.
Further Author explains that inequalities are the most computationally expensive part of the refinement pipeline. Here the author says that previous works rely only on one type of algorithm which makes them inefficient. The author also proposes a new third algorithm and mentions that the FACET chooses the most efficient among the three algorithms for the refinement purpose. The author also mentions that this is discussed in more detail in Section 6.
Robust evaluation plans: Here the author explains that selecting the order of predicates is very important. The author gives an example where DC with m predicates is present then we can choose the order of refinement pipeline in m! Ways. In addition, the author also states that FACET offers 3 algorithms for the refinement of the inequalities and each of them may have different performance depending on the input. Thus, choosing a low-cost evaluation plan is very important and it's also challenging as it depends on the order in which the predicates are refined.
The author mentions that the previous works rely on predicate selectivity to determine the predicate order, but they don't explore that different predicates have different evaluation costs. The author then proposes a novel algorithm that explores the high accuracy of the column sketches to select the refinement order and algorithms.

Multi-constraint execution: Author here tells us that the previous works perform error detection one constraint at a time. If the two DCs share one predicate and even in this case both are computed separately which relates to repeated computations. To overcome this problem, the author makes a trie-based scheme in FACET to check multiple DCs at the same time.
FACET supports two organizations of predicate tries: i). Based on cost of predicate in each DC. It favors fast processing of predicates.
ii). Based on frequency count of each predicate w.r.t all other predicates. It favors processing resumes.
FACET can evaluate predicate tries independently of each other which in turn benefits from parallel execution. This helps FACET to speed the multi-constraint execution even when the input contains DCs which have no common predicates.

5: REFINEMENT ALGORITHMS

Equalities:

In this section, the author explains how the FACET system handles equalities, which are comparisons between columns in a database. It describes a process where one column (column A) is used as the "build side," and another column (column B) is the "probe side." The system iterates through the data, creating a hash table that links unique values from column A to sets of tuples. In the probing phase, it checks values in column B and updates the sets of tuples accordingly. If a pair of distinct tuples is found, it is included in the results. The choice of which column to use as the build side is based on estimating the number of unique values in each column. The one with the lowest estimated number is chosen to minimize the number of hash table entries. If column B is selected as the build side, the order of the pair of tuples is reversed. The Author explains the example as using the predicate t.SID = t'.ID. After the probing phase, the hash table contains entries with values of column B (e.g., ⟨100, ({t1}, {t1})⟩, ⟨101, ({t3, t4}, {t2})⟩, and ⟨102, ({t2}, {t3})⟩.

Authors also analyzed the method to quickly find and process pairs of related data items, like employees in the same department, without doing unnecessary work. It helps them save time and do things more efficiently. They want to avoid doing unnecessary scans (searching or examining data) when they are dealing with pairs of data items called TIDs (table IDs). The Author has used predicates along with a hash table in such a way that the table uses reflexive pairs with the same columns on both sides (for example, {t2, t3, t4}, {t2, t3, t4}).

Non-equalities:

Like equalities, authors have analyzed non-equalities in such a way that hash based approach will be used. The only difference was the how to build output. In the non-equality the predicate explains that no two columns will be equal i.e. predicate p: t.A is not equal to t'.B.

For empty TIDs, if there's an empty tids'2 associated with a key, it means that none of the data items in tids2 have a value in column B that matches the value in column A of the data items in tids'. In simple terms, it means that the values in column A and column B are completely different. They then move this pair of data sets (tids'1, tids2) to the next step.

For nonempty TIDs, if there's data in tids'2 associated with a key, it means that all data items in tids'1 have a value in column A that matches the value in column B of the data items in tids'2. In this case, they calculate a set difference, which means they find the data items in tids2 that are not in tids'2. These data items go into a new set called tids''2. Then, they move this pair of data sets (tids'1, tids''2) to the next step for further processing.

Authors have researched a **Hybrid TIDs approach** to process data based on functional dependencies. They give an example of a functional dependency: ¬ (t.StartDate = t'.StartDate ∧ t.Salary ≠ t'.Salary), which essentially means that if two data items have the same start date, their salaries must be the same.

They first focus on the equality part, where they want to check if data items have the same start date. This results in a pair of data sets (tids1, tids1), where tids1 contains the data items that have the same start date. They store these data items in simple arrays of integers.

The next step involves checking if the salaries of the data items are not equal. They use the fact that tids1 is reflexive (meaning it has the same data items on both sides) to optimize the process. They look at the salaries of the data items in tids1 and create a hash table that associates salary values with subsets of data items that have that salary. They also use compressed bitmaps to store and process TIDs because they need to perform logical operations for non-equality.

Afterwards, authors have created pairs based on non-equality conditions and finally they have created pairs like {t3}, {t2}), and then the pair ({t2}, {t3}).

Inequalities:

In this context, authors refer to inequalities as conditions where two sets of data are compared using operators like "<" (less than), "≤" (less than or equal), ">" (greater than), or "≥" (greater than or equal). They have researched different algorithms on different sets of data.

a. IEJoin: IEJoin algorithm is a specific algorithm designed to handle inequality conditions in data. It's used to work with two inequality conditions at a time. It includes various steps as follows:

i. Sorting Data: First, it sorts the data on both sides of the inequality condition.

ii. Creating Arrays: It then creates arrays that describe the relationships between the sorted data. These arrays help keep track of the data's positions relative to the sorted order.

iii. Marking Pairs: Using these arrays, the algorithm goes through the sorted data on one side of the inequality and marks pairs of data items that meet the criteria of the second inequality condition.

iv. Checking Pairs: Next, the algorithm goes through the arrays for the other side of the inequality and checks the marked pairs to see if they also meet the criteria of the first inequality condition.

v. Output: The algorithm uses a strategy where it gradually adds pairs of data items to the output. If it encounters data items that create different patterns or don't match, it separates them and starts a new iteration to process them separately.

IEJoin algorithm’s performance is majorly taken by sorting phase which results low output.

b. Hash-Sort-Merge (HSM):

Authors have researched the HSM algorithm as part of further research on inequality conditions. HSM is a specific algorithm designed for dealing with inequality conditions. It's used in a tool called VioFinder. It includes various steps as follows:

i. Sorting & Merging: HSM follows a sort-merge approach, which means it organizes and combines data sets based on the distinct values in each column.

ii. Hashtable creation: Like other data processing algorithms, HSM starts by creating a hash table for each column in the input data. These hash tables help organize the data based on each column's values.

iii. Sort sets: After building the hash tables, HSM creates sorted sets of unique values from the keys in these hash tables. It then removes values from the beginning and end of these sorted sets if they can't form pairs of data items that satisfy the inequality condition.

iv. Merge Data: In the merging phase, the algorithm scans through the two sorted sets of values in an interleaved (alternating) manner to find pairs of values that satisfy the inequality condition.

v. Collecting TIDs: For each pair of matching values, the algorithm collects the table IDs (TIDs) associated with those values and creates pairs of TIDs in the output. It incrementally builds these results by using logical union (OR) operations.

According to authors, HSM is designed to work well with conditions of low selectivity, which means when there are not many matching results. However, it faces challenges with high-cardinality columns, where there are many distinct values in a column. Although bitmap compression helps reduce storage and complexity, it doesn't reduce the number of logical operations needed to build the resulting pairs of TIDs.

c. Binning-Hash-Sort-Merge" (BHSM):

Authors have extended their research in inequalities using BHSM that extends the capabilities of the existing HSM algorithm for handling inequality conditions in data. BHSM is a new algorithm that builds upon the HSM (Hash-Sort-Merge) algorithm to improve its performance when dealing with inequalities.

BHSM uses a binning method to partition the values in a column into ranges and represent these ranges using bitmaps. Instead of creating hash tables, BHSM creates "range maps" that map these column ranges to sets of data items (tids) based on the values within those ranges. For simplicity, BHSM uses equal-width binning. It finds the minimum and maximum values in each column, divides this range into equally spaced sub-ranges (bins), and creates bitmaps for each bin. BHSM follows a similar flow to HSM, with a sorting and merging phase. However, in the merging phase, BHSM uses interleaved linear scans of sorted sets of column ranges, rather than individual column values.

For each pair of matching column ranges, BHSM produces pairs of table IDs (TIDs) associated with those ranges. These pairs are typically very dense, meaning they contain many pairs of data items that meet the inequality condition. This makes the algorithm efficient. BHSM also performs a "**candidate check**" for the TIDs related to the matching ranges. To do this, it runs the original HSM algorithm on these TIDs. This check ensures that the pairs indeed satisfy the inequality condition.

The authors showed an example which falls under the BHSM approach which provides how BHSM works for a predicate that compares start dates in an “Employee” table. BHSM uses two bins to create a range map, then performs candidate checks. The result is pairs of TIDs that represent all pairs of data items that meet the inequality condition. BHSM is especially useful for handling high-cardinality columns, where there are many distinct values. It limits the number of logical operations in the first phase due to binning and focuses the candidate check phase on a smaller portion of the domain space. This helps mitigate the challenges associated with high-cardinality columns.

d. Optimizations:

Authors discuss optimizations applied to the refinement of single-column inequalities when working with reflexive pairs of table IDs (TIDs). When refining single-column inequalities and using algorithms that work with reflexive pairs of TIDs, the algorithms only need to iterate through one side of the TID pairs to create auxiliary data structures. This helps make the process more efficient by reducing the need to process both sides of the TID pairs.

A second optimization opportunity arises when dealing with refinements that follow inequality refinements, especially those implemented with algorithms like HSM or BHSM. These algorithms often create output pairs of TIDs incrementally from previous pairs.

The optimization involves caching the data structures built for the right-hand side of each pair of TIDs. So, when a new pair of TIDs arrives for refinement, the algorithm can make use of the data structures already built for the right-hand side of the previous pair.

Instead of building these data structures from scratch for each new pair, the algorithm can incrementally update the existing structures if there is some difference between the right-hand side of the previous pair and the current one.

Authors have discussed an example of the refinement of the inequality "t.Salary > t’.Salary" using HSM. The pairs of TIDs produced may look like this: "({t3}, {t1}), ({t2, t4}, {t1, t3})." A refinement receiving these pairs can use the data structures already built for the right-hand side of each pair (e.g., the data structures for t1, t3). It only needs to update these structures incrementally as new pairs arrive.

6. Refinement Planning

The author 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 in this section.

6.1

The author basically has emphasized the cost which is incurred from iterating through hash table entries. This section also states that large hash entries are less cache -effective reason being higher data movement & perform worse than small hash tables which better fit the cache. To illustrate this author illustrated using an experiment, ran FACET for single equality. Using the two storage types, array of integers & compressed bitmaps. Shown in figure.

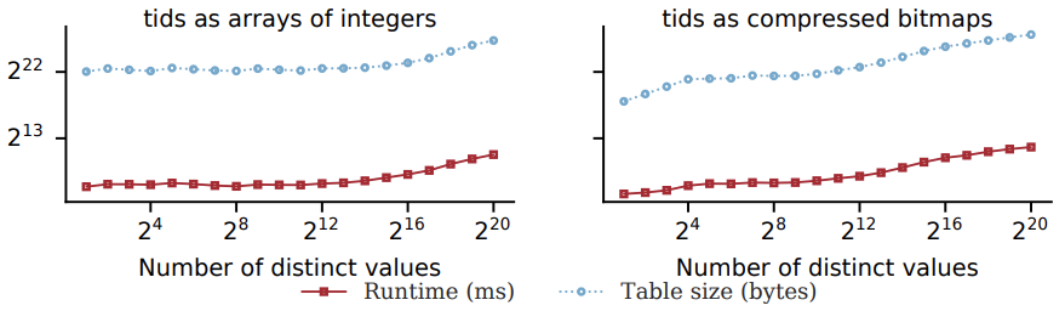

Figure 2: The impact of column cardinality on hash table sizes and equality refinement runtime.

Through this experiment the author established the The number of distinct values in a column (cardinality) impacts the overall performance of refinement operations.

6.2 Organizing Refinements:

The author divides the DC predicates into the class described in Section 5: equalities, non-equalities, and inequalities. Here, the intuition is to evaluate the classes of predicate of higher selectivity first.

Compared to other predicate classes, equalities usually incur low evaluation costs & have higher selectivity so, they come first. The author provides insights into ordering predicates within each class, primarily based on column cardinality information, the priority is predicates on the column with the least estimated cardinality. Author talks about how to prioritize single-column equalities, considering the likelihood of unique values in high-cardinality columns and for multiple single-column equalities, it introduces a greedy approach, GreedyHLL that considers intermediate size and evaluation costs.

Inequalities - The IEJoin algorithm is designed & limited to two predicates at time., single equality, thus for single inequality, the decision stays between HSM & BHSM.

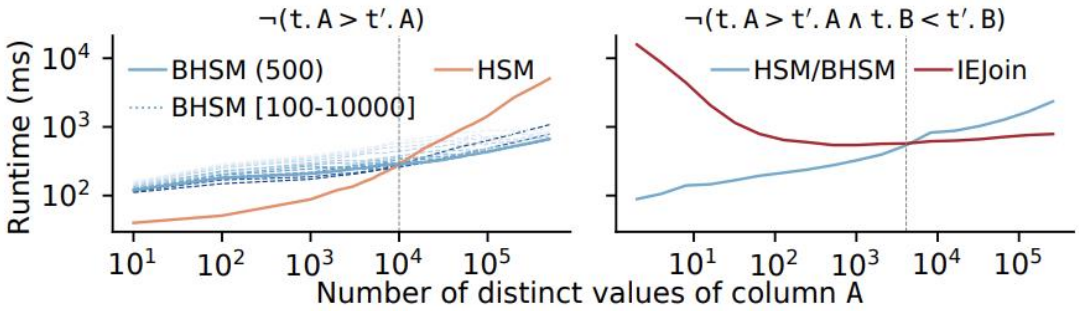

Figure 3: The different impacts of column cardinality on the runtime of HSM, BHSM and IEJoin.

For any pair, Author mentions we can either use a pair of refinements or single refinement with IEJoin. It turns out that different approaches complement each other well.

7. Experimental Evaluation:

In this section, Author reports the results of experimentation evaluation.
In 7.1, the author talks about experimental settings & comparison of FACET with various other systems in 7.2 and 7.3 Evaluation of Design decisions of FACET in more Depth.

7.1 - Experimental settings

Table - 2 lists 12 DCs and 4 Datasets used in the experiments. The author compared FACET to a leading commercial DBMS (DBMS-X) and three distinct Open-source DBMS, postgresql, MonteDB & DuckDB.

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	$\neg$(t.AreaCode = t′.AreaCode $\wedge$ t.Phone = t′.Phone)
Tax	10M	Medium, High	φ_5	$\neg$(t.ZipCode = t′.ZipCode $\wedge$ t.City $\neq$ t′.City)
Tax	10M	Low	φ_6	$\neg$(t.State = t′.State $\wedge$ t.HasChild = t′.HasChild $\wedge$ t.ChildExemp $\neq$ t′.ChildExemp)
Tax	10M	Low, Medium, High	φ_7	$\neg$(t.State = t′.State $\wedge$ t.Salary > t′.Salary $\wedge$ t.Rate < t′.Rate)
Flights	3.6M	Low, Medium	φ_8	$\neg$(t.Origin = t′.Dest $\wedge$ t.Dest = t′.Origin $\wedge$ t.Distance $\neq$ t′.Distance)
Flights	3.6M	Low, Medium, High	φ_9	$\neg$(t.Orgin = t′.Origin $\wedge$ t.Dest = t′.Dest $\wedge$ t.Flights > t′.Flights $\wedge$ t.Passengers < t′.Passengers)
TPC-H	6M	Medium, High	φ_{10}	$\neg$(t.Customer = t′.Supplier $\wedge$ t.Supplier = t′.Customer)
TPC-H	6M	Medium	φ_{11}	$\neg$(t.Receiptdate $\geq$ t′.Shipdate $\wedge$ t.Shipdate $\leq$ t′.Receiptdate)
TPC-H	6M	Low, High	φ_{12}	$\neg$(t.ExtPrice > t′.ExtPrice $\wedge$ t.Discount < t′.Discount)
TPC-H	6M	Low, High	φ_{13}	$\neg$(t.Qty = t′.Qty $\wedge$ t.Tax = t′.Tax $\wedge$ t.ExtPrice > t′.ExtPrice $\wedge$ t.Discount < t′.Discount)
IMDB	2.5M	Low, High	φ_{14}	$\neg$(t.Title = t′.Title $\wedge$ t.ProductionYear = t′.ProductionYear $\wedge$ t.Kind $\neq$ t′.Kind)
IMDB	5.8M	Low, High	φ_{15}	$\neg$(t.Title = t′.Title $\wedge$ t.Name = t′.Name $\wedge$ t.CharName = t′.CharName $\wedge$ t.Role = t′.Role)

7.2 - Compared to other systems.

Fig-4 shows the runtime of running FACET and the DBMS based approaches for all the evaluated DCs, on three data scales. The results show that FACET can perform and scale well for all datasets & DCs, it also shows that FACET finishes error detection much faster than the DBMSs, majorly.

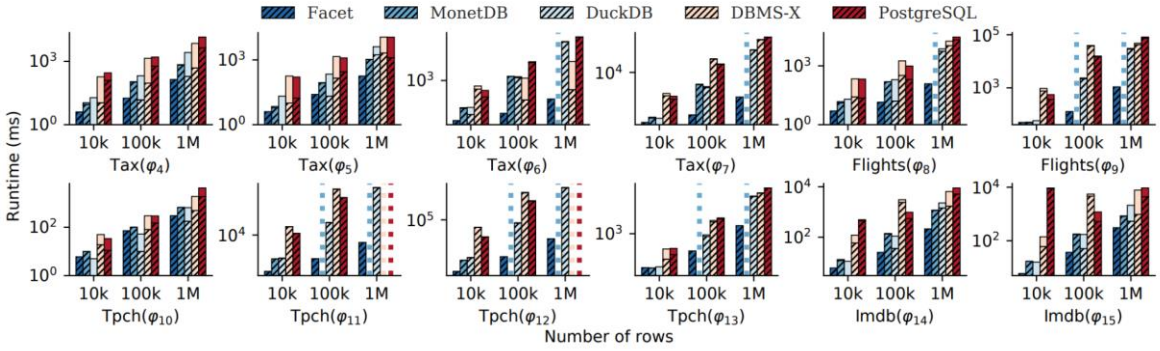

Figure 4: Runtime comparison of FACET to four distinct DBMSs using SQL queries. The dashed areas in each bar represent the violation detection time, whereas the solid areas represent the indexing construction time (only for the DBMSs). The dotted lines are cases where the respective DBMS either exceeded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

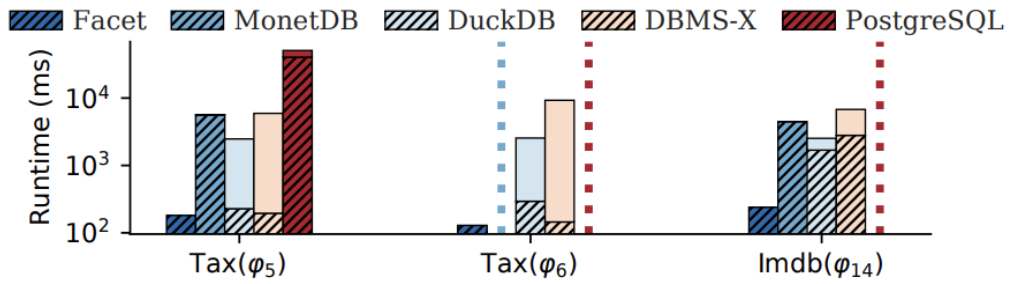

Figure 5: Runtime comparison of FACET to four distinct DBMSs using the GROUP BY approach. The filling patterns of the bars are the same as in Figure 4.

7.3 - Evaluation of the design decisions of FACET.

In this section the author does another set of experiments to better investigate the design decisions in FACET and talks about why it performs better. It includes various experiments.

Figure 7 illustrates the performance differences resulting from various types of tids storage.

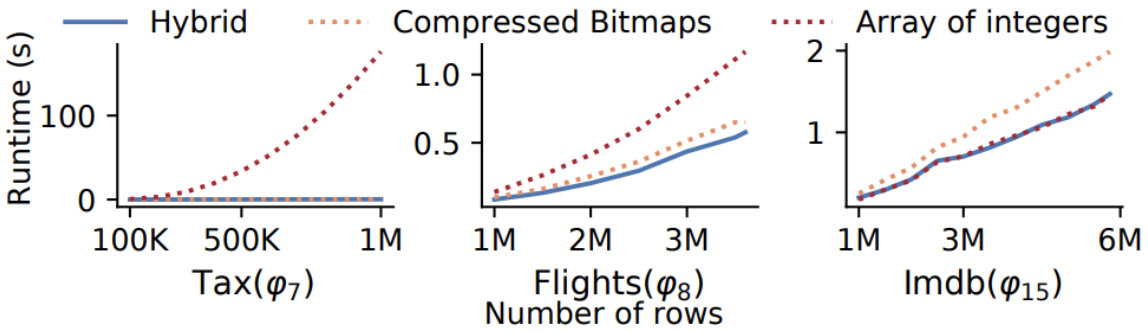

Figure 7: Impact of the types of `tids` storage on runtime.

Figure 8 compares the use of static inequality algorithms to FACET's adaptive approach.

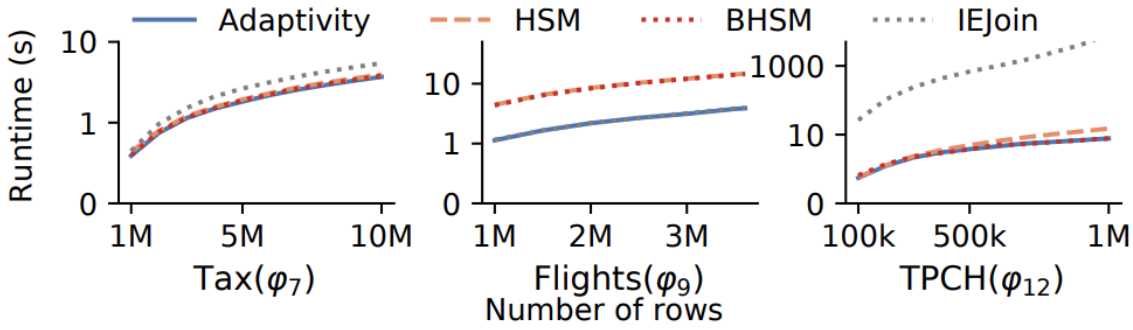

Figure 8: Impact of adaptivity vs. non-adaptivity strategies on the runtime for DCs including inequalities.

Table 3 compares the baseline evaluation order (naive) with FACET's chosen order, which is the reverse of the baseline order for the examples presented.

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. The author focusses on uniqueness constraints as they are composed of single column equalities solely.

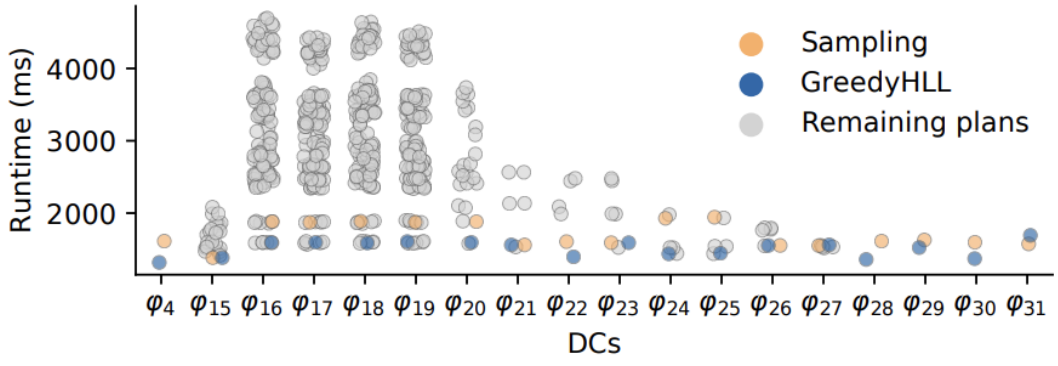

Figure 9: Runtime of FACET using the plans of GreedyHLL vs. runtime of FACET using all other possible plans.

The next experiment compares the execution options that FACET.

offers when multiple DCs are given as input. The author used the uniqueness constraints described previous sections, which share many predicates. among each other.

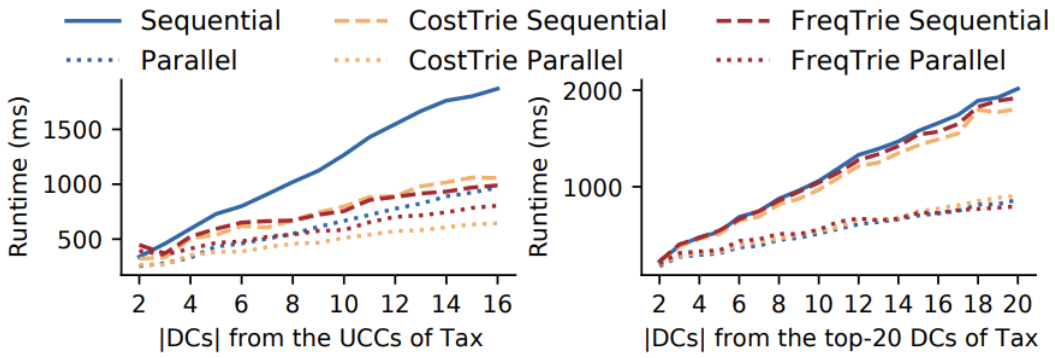

Figure 10: Performance of the multi-DC execution modes.

In the final experiment, Author shows the heap size required to successfully execute facet for each DC on 1M records.

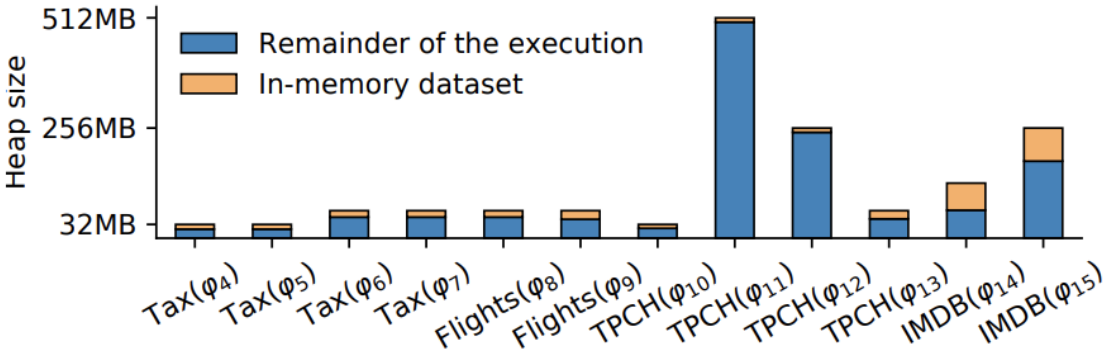

Figure 12: Memory requirement of FACET.

8. Concluding Remarks

In this section, Author concludes by talking about the FACET system presented to address challenges of detecting data errors as DC violations.
The author talks about the significance of column sketches as a powerful tool in the organization of predicate evaluation and minimizing the resource drain. The author introduces new algorithms to achieve the right performance. Author talks about building one single system built upon combining the various ideas and how FACET seamlessly integrates with constraint-based data cleaning pipelines and finally talks about how this work can be extended to solve other use cases like in query processing by adding cleaning operators into the query plans to track and repair DC violation on Demand.