RULEMINER: Data Quality Rules Discovery

Xu Chu[1⋆] Ihab F. Ilyas[1⋆] Paolo Papotti[2] Yin Ye[2]

[1] University of Waterloo, Canada
{x4chu,ilyas}@uwaterloo.ca
[2] Qatar Computing Research Institute (QCRI), Qatar
{ppapotti,yye}@qf.org.qa

***Abstract*—Integrity constraints (ICs) are valuables tools for enforcing correct application semantics. However, manually designing ICs require experts and time, hence the need for automatic discovery. Previous automatic ICs discovery suffer from (1) limited ICs language expressiveness; and (2) time-consuming manual verification of discovered ICs. We introduce RULEMINER, a system for discovering data quality rules that addresses the limitations of existing solutions.**

I. INTRODUCTION

As businesses generate and consume data more than ever, enforcing and maintaining the quality of their data assets become critical tasks. One in three managers does not trust the information used to make decisions [5], since establishing trust in data becomes a challenge as the variety and the number of sources grow. Therefore, data cleaning is an urgent task towards improving data quality. Integrity constraints (ICs), originally designed to improve the quality of a database schema, have been recently repurposed towards improving the quality of data, either through checking the validity of the data at points of entry, or by cleaning the dirty data at various points during the processing pipeline [4], [6], [7]. ICs are often referred to as data quality rules when used for data cleaning.

Manually designing data quality rules often requires domain experts and engineers to express those rules in some formal language for automatic enforcement. As data owners are often not data quality rules experts, automatic rules discovery is necessary. Unfortunately, existing literature on automatic data quality rules discovery suffers from two main drawbacks. First, existing discovery algorithms are usually designed for a single rule language, thus not able to uncover many useful quality rules. Second, there are large amount of generated rules, and not all of them are useful due to overfitting and errors in the data. Manual evaluation of the output is a tedius process and requires expertise of the language at hand.

We introduce RULEMINER, a system for discovering data quality rules that addresses the limitations of existing solutions. The system discovers rules expressed with Denial constraints (DCs), which subsume many rule languages, including functional dependencies (FDs), conditional functional dependencies (CFDs), and check constraints (Section II). RULEMINER efficiently mines data (Section III) and employs a novel interestingness function to rank the discovered rules (Section IV).

In the demonstration we show how rules are discovered by our system from multiple real-world scenarios such as Freebase and enterprise data (Section V). Rules are exposed by means of compact positive and negative data examples for effective user validation and can be explored with adjustable ranking based on different properties of the rules. We also show how to use the discovered DCs by automatically generating violation detection and repairing java procedural code, which can be further refined by expert users.

⋆ Work done while at QCRI.

II. DATA QUALITY RULES

In the next three sections we introduce the language used to express quality rules in our approach, the algorithm to discover them, and the functions to rank discovered rules, respectively. For further details we refer the reader to [3]. We start by introducing the language with an example.

Example 1: Consider the US tax records table in Table I. Each record describes an individual address and tax information with 14 attributes: first and last name (FN, LN), gender (GD), area code (AC), mobile phone number (PH), city (CT), state (ST), zip code (ZIP), marital status (MS), has children (CH), salary (SAL), tax rate (TR), tax exemption amount if single (STX) or married (MTX).

Suppose that the following constraints hold: (1) area code and phone identify a person; (2) two persons with the same zip code live in the same state; (3) a person who lives in Denver lives in Colorado; (4) within a state, a person earning a lower salary cannot have a higher tax rate; and (5) a salary is always higher than the corresponding single tax exemption.

Constraints (1), (2), and (3) can be expressed as a key constraint, an FD, and a CFD, respectively.

$(1) : Key\{AC, PH\}$
$(2) : ZIP \rightarrow ST$
$(3) : [CT = \text{'Denver'}] \rightarrow [ST = \text{'CO'}]$

Since Constraints (4) and (5) involve order predicates $(>, <)$, and (5) compares different attributes in the same predicate, they cannot be expressed by FDs and CFDs. However, they can be expressed in first-order logic.

$$c_4 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$$
$$c_5 : \forall t_\alpha \in R, \neg(t_\alpha.SAL < t_\alpha.STX)$$

Since first-order logic is more expressive, Constraints (1)-(3) can also be expressed as follows:

$$c_1 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$$
$$c_2 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$$
$$c_3 : \forall t_\alpha \in R, \neg(t_\alpha.CT = \text{'Denver'} \wedge t_\alpha.ST \neq \text{'CO'})$$

Denial Constraints (DCs) [7], a universally quantified first order logic formalism, can express all constraints in

TID	FN	LN	GD	AC	PH	CT	ST	ZIP	MS	CH	SAL	TR	STX	MTX
t_1	Mark	Ballin	M	304	232-7667	Anthony	WV	25813	S	Y	5000	3	2000	0
t_2	Chunho	Black	M	719	154-4816	Denver	CO	80290	M	N	N	60000	4.63	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
t_8	Marcelino	Nuth	F	304	540-4707	Kyle	WV	25813	M	N	10000	4	0	0

TABLE I. TAX DATA RECORDS.

Example 1 as they are more expressive than FDs, CFDs, and check/domain constraints. Consider a database schema of the form $\mathbb{S} = (\mathbb{U}, \mathbb{R}, \mathbb{B})$, where $\mathbb{U}$ is a set of database domains, $\mathbb{R}$ is a set of database predicates or relations, and $\mathbb{B} = \{=, <, >, \neq, \leq, \geq\}$. We use a notation for DCs of the form $\varphi : \forall t_\alpha, t_\beta, t_\gamma, \ldots \in R, \neg(P_1 \wedge \ldots \wedge P_m)$, where P_i is of the form $v_1 \phi v_2$ or $v_1 \phi c$ with $v_1, v_2 \in t_x.A, x \in \{\alpha, \beta, \gamma, \ldots\}, A \in R$, and c is a constant. For simplicity, we assume there is only one relation R in $\mathbb{R}$. A DC states that all the predicates cannot be true at the same time, otherwise, there is a violation.

The more expressive power an IC language has, the harder it is to exploit it, for example, in automated data cleaning algorithms, or in its static analysis. There is an infinite space of business rules up to ad-hoc programs for enforcing correct application semantics. In RULEMINER we rely on DCs as they achieve a balance between expressiveness and complexity: (1) they can be expressed as SQL queries for consistency checking; (2) they have been proven to be a useful language for data cleaning in many aspects [4]; and (3) there are sound inference rules and a linear implication testing algorithm for them that enable efficient use, as we show next.

III. DISCOVERY ALGORITHM

Given a relational schema R and an instance I, our goal is to find all valid minimal DCs that hold on I.

Consider traditional FDs discovery on R with $|R| = m$. Taking an attribute as the right hand side of an FD, any subset of remaining $m-1$ attributes could serve as the left hand side. Thus, the space to be explored for FDs discovery is $m * 2^{m-1}$. Consider now discovering DCs involving at most two tuples without constants. A predicate space needs to be defined, upon which the space of DCs is built. The structure of a predicate consists of two different attributes and one operator. Given two tuples, we have $2m$ distinct cells; and we allow six operators $(=, \neq, >, \leq, <, \geq)$. Thus the size of the predicate space $\mathbf{P}$ is: $|\mathbf{P}| = 6 * 2m * (2m-1)$. Any subset of the predicate space could constitute a DC. Therefore, the search space for DCs discovery is of size $2^{|\mathbf{P}|}$. Checking if each candidate DC is a valid minimal DC is not feasible because of the exponential number of candidate DCs. Therefore, we turn to an instance-driven approach that strongly relies on DCs static analysis.

Figure 1 shows the flow of RULEMINER and the main steps for the mining algorithm. In the `Evidence Generation`, we build a data structure Evi_I to collect evidence from I. Evi_I is built by comparing each pair of tuples in I to find the set of satisfied predicates in $\mathbf{P}$. The discovery problem is then reduced to finding all minimal set covers on Evi_I with a depth first search (DFS) visit in the `Discovery Algorithm`.

To speed up the search for minimal set covers, we rely on properties of DCs. Specifically, the `Inference System`

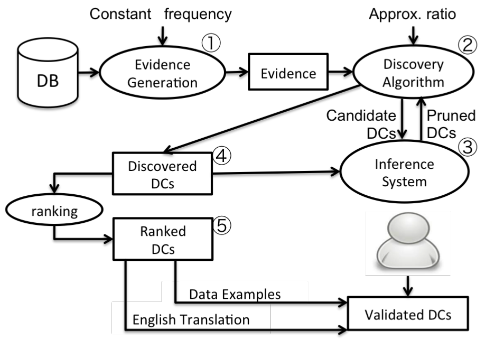


Fig. 1. System Architecture for RULEMINER.

$\mathcal{I}$ is composed of three inference rules: *triviality* provides a syntactic way of checking if a DC is trivial or not, while *augmentation* and *transitivity* state how a DC is implied by other DCs. The discovery algorithm provides candidate DCs to $\mathcal{I}$, which checks if they are implied or not. Implied DCs can be pruned to early terminate branches of the search tree.

As per discussed in [3], the complexity of our algorithm is $O(|\mathbf{P}| * n^2 + |\mathbf{P}| * (1 + w_P) * K_P)$, which is quadratic in the number of tuples n, thus successfully avoiding the exponential checking of all DCs, which has a complexity of $O(2^{|\mathbf{P}|} * n^2)$.

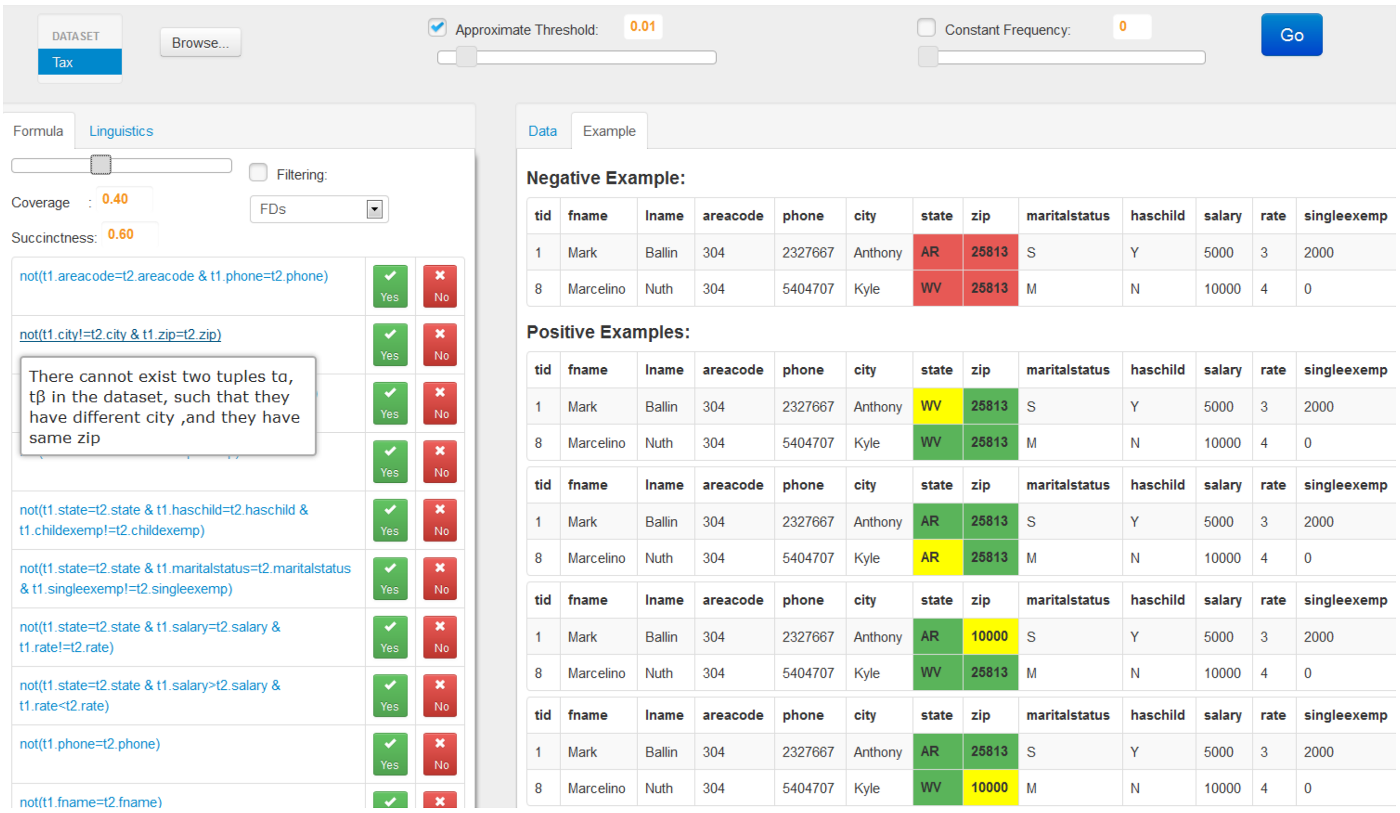
RULEMINER provides two parameters that users can set depending on the scenario. The *approximation ratio* represents the maximum number of violations allowed in the instance. This allows the discovery of DCs when there are errors in the input dataset. In addition, to discover rules involving constants such as c_3, the *constant frequency* parameter sets the minimum number of tuple pairs that a predicate with constants must satisfy in order to be considered evidence.

IV. RANKING FUNCTIONS

Though our discovery algorithm is able to prune trivial, non-minimal, and implied DCs, the number of DCs returned can still be large. To tackle this problem, RULEMINER employs a scoring function to rank rules based on their size and their support from the data. Given a DC φ, we denote by $Inter(\varphi)$ its *interestingness* score.

We recognize two dimensions that influence $Inter(\varphi)$: *succinctness* and *coverage* of φ, which are both defined on a scale between 0 and 1. Each score represents a different yet important intuitive dimension to rank discovered DCs.

Succinctness is motivated by the Occam's razor principle, which suggests that, among competing hypotheses, the one that makes fewer assumptions is preferred. It is also recognized that overfitting occurs when a model is excessively complex [2]. Minimum description length (MDL), which measures the code length needed to compress the data, is a formalism to realize



DATASET: Tax

Approximate Threshold: 0.01 ✓

Constant Frequency: 0

Coverage : 0.40

Succinctness: 0.60

Filtering: FDs

not(t1.areacode=t2.areacode & t1.phone=t2.phone)	Yes	No
not(t1.city!=t2.city & t1.zip=t2.zip)	Yes	No
	Yes	No
	Yes	No
not(t1.state=t2.state & t1.haschild=t2.haschild & t1.childexemp!=t2.childexemp)	Yes	No
not(t1.state=t2.state & t1.maritalstatus=t2.maritalstatus & t1.singleexemp!=t2.singleexemp)	Yes	No
not(t1.state=t2.state & t1.salary=t2.salary & t1.rate!=t2.rate)	Yes	No
not(t1.state=t2.state & t1.salary>t2.salary & t1.rate<t2.rate)	Yes	No
not(t1.phone=t2.phone)	Yes	No
not(t1.fname=t2.fname)		

There cannot exist two tuples tα, tβ in the dataset, such that they have different city ,and they have same zip

Negative Example:

tid	fname	lname	areacode	phone	city	state	zip	maritalstatus	haschild	salary	rate	singleexemp
1	Mark	Ballin	304	2327667	Anthony	AR	25813	S	Y	5000	3	2000
8	Marcelino	Nuth	304	5404707	Kyle	WV	25813	M	N	10000	4	0

Positive Examples:

tid	fname	lname	areacode	phone	city	state	zip	maritalstatus	haschild	salary	rate	singleexemp
1	Mark	Ballin	304	2327667	Anthony	WV	25813	S	Y	5000	3	2000
8	Marcelino	Nuth	304	5404707	Kyle	WV	25813	M	N	10000	4	0

tid	fname	lname	areacode	phone	city	state	zip	maritalstatus	haschild	salary	rate	singleexemp
1	Mark	Ballin	304	2327667	Anthony	AR	25813	S	Y	5000	3	2000
8	Marcelino	Nuth	304	5404707	Kyle	AR	25813	M	N	10000	4	0

tid	fname	lname	areacode	phone	city	state	zip	maritalstatus	haschild	salary	rate	singleexemp
1	Mark	Ballin	304	2327667	Anthony	AR	10000	S	Y	5000	3	2000
8	Marcelino	Nuth	304	5404707	Kyle	WV	25813	M	N	10000	4	0

tid	fname	lname	areacode	phone	city	state	zip	maritalstatus	haschild	salary	rate	singleexemp
1	Mark	Ballin	304	2327667	Anthony	AR	25813	S	Y	5000	3	2000
8	Marcelino	Nuth	304	5404707	Kyle	WV	10000	M	N	10000	4	0

Fig. 2. RULEMINER front end interface with Negative-Positive (NE-PE) pair for c_2 generated from tuple pair $\langle t_1, t_8 \rangle$.

the Occams razor principle. We first define the alphabet of DCs as $\mathbb{A} = \{t_\alpha, t_\beta, \mathbb{U}, \mathbb{B}, Cons\}$, where $\mathbb{U}$ is the set of all attributes, $\mathbb{B}$ is the set of all operators, and $Cons$ are constants. We define the length of a DC to be the number of symbols in $\mathbb{A}$ that appear in φ. We define succinctness of a DC φ $Succ(\varphi)$ to be the minimal possible length of any DC divided by the length of φ. The minimal length is 4, as in c_5, which has $Succ(c_5) = \frac{4}{4} = 1$. Other examples are $Succ(c_1) = \frac{4}{5} = 0.8$ and $Succ(c_4) = \frac{4}{8} = 0.5$.

Coverage is also a general principle in data mining to rank results [1]. These scoring functions measure the statistical significance of the mining targets in the input data. In frequent itemset mining, the support of an itemset is defined as the proportion of transactions in the data that contain the itemset. Only if the support of an itemset is above a threshold, it is considered to be frequent. The coverage of φ $Coverage(\varphi)$ depends on the support from I. Each tuple pair gives a support to φ depending on the number of satisfied predicates. For example, in Table I, tuple pair $\langle t_1, t_8 \rangle$ satisfies one predicate $t_\alpha.ZIP = t_\beta.ZIP$ in c_2 and gives more support than $\langle t_1, t_2 \rangle$, which satisfies no predicate in c_2. The $Coverage(\varphi)$ is defined by aggregating all different supports from all tuple pairs divided by the total number of tuple pairs.

Given a DC φ, we define the interestingness score as a linear weighted combination of the two dimensions $Inter(\varphi) = a \times Coverage(\varphi) + (1 - a) \times Succ(\varphi)$.

V. DEMONSTRATION OUTLINE

We demonstrate various aspects of RULEMINER system: (1) the front end of RULEMINER, including how the users set the parameters and how discovered rules are presented and validated; (2) a back end view of how RULEMINER generates DCs internally; and (3) how expert users can apply a discovered rule to see analytics on cleaning metadata and further fine-tune the rule with procedural java code.

1. Front End Interface. The front end interface for RULEMINER has the parameter specification, rules ranking, and their presentation and validation.

The upper panel in Figure 2 shows the input to RULEMINER: (1) Data source selection; (2) The approximation ratio specifies the maximum amount of errors for a valid rule; (3) The constant frequency defines how often a constant must appear in order to be considered for a rule; (4) A filtering option lets the user select a subset of the rules depending on the formalism of interest; and (5) A filtering option lets the user select a subset of the rules depending on the interested columns of the input table.

Discovered rules are presented ranked according to their *Inter* score as shown in the left panel in Figure 2.

In order to verify if a rule indeed applies, users need to validate. English translation and data examples for each rule assist the users in understanding the discovered rules.

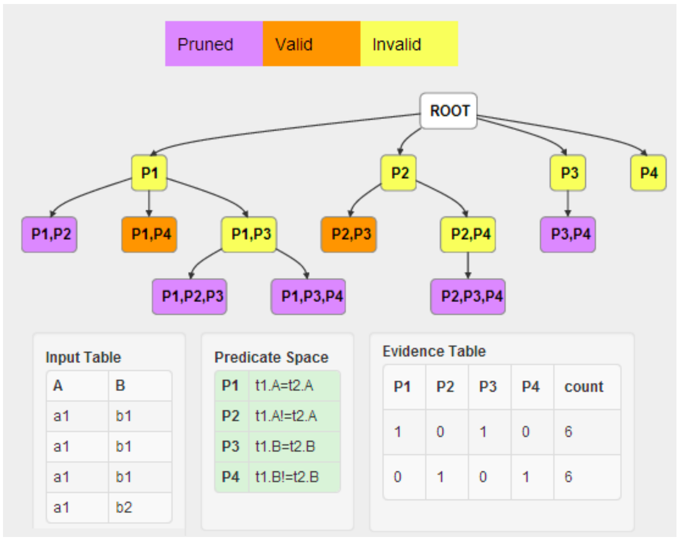
Each DC is translated into a meaningful English sentence. For example, c_1 is translated into "there cannot exist two tuples t_α, t_β in Tax data records, such that they have same area code, and they have same mobile phone number". While c_4 is translated into "there cannot exist two tuples t_α, t_β in Tax data records, such that they have same state, the salary of tuple t_α is less than that of tuple t_β, and the tax rate of tuple t_α is greater than that of t_β".

Data examples further assist the users in understanding the rules. Given a DC φ, with $\varphi.Pres$ that denotes its set of predicates:

- A *Negative Example (NE)* of a DC φ w.r.t. instance I is a pair of tuples $\langle t_x, t_y \rangle$, s.t. $\forall P \in \varphi.Pres, \langle t_x, t_y \rangle \models P$.
- A *Positive Example (PE)* of a DC φ w.r.t. instance I is a pair of tuples $\langle t_x, t_y \rangle$, s.t. $\exists P \in \varphi.Pres, \langle t_x, t_y \rangle \not\models P$.
- A *NE-PE pair* of a DC φ w.r.t. instance I is two pairs of tuples, such that the first pair is a NE, the second pair is a PE, and NE and PE differ in exactly one cell.

A NE of a DC φ is a pair of tuples that satisfies all predicates in φ, i.e., a violation of I for φ. For a discovered rule φ, we present to the users one NE and x corresponding PEs, where x equals to the number of cells involved in φ. If there is a violation in I, we use it as NE and change it to generate the PEs [4]. If there is no violation in I, we pick a PE that has as many predicates in φ as possible, and modifies it to have a NE. For example, the right panel in Figure 2 presents one NE and four corresponding PEs for c_2.

A NE-PE pair helps the user understand what is wrong in the NE by looking at the difference between the NE and the PE. If the user agrees NE is wrong data and PE is correct data, then the NE-PE pair is correct and the rule is valid. Otherwise, if the user disagrees because either NE is correct or PE is incorrect, then the NE-PE pair is incorrect and the rule is dropped.



Input Table

A	B
a1	b1
a1	b1
a1	b1
a1	b2

Predicate Space

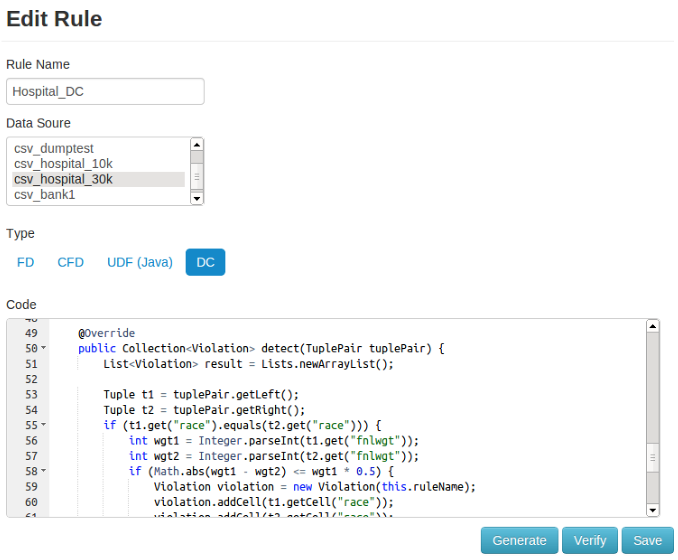
P1	t1.A=t2.A
P2	t1.A!=t2.A
P3	t1.B=t2.B
P4	t1.B!=t2.B

Evidence Table

P1	P2	P3	P4	count
1	0	1	0	6
0	1	0	1	6

Fig. 3. RULEMINER back end internals with predicate space, evidence set, and a snapshot of the DFS tree in the discovery algorithm.

2. Back End Internals. The back end shows how RULEMINER discovers DCs. Figure 3 shows the predicate space with four predicates we have built for a input table with two columns A and B. We also visualize the evidence as a table with P_1, P_2, P_3, P_4, and *count*; each row of the evidence table represents a satisfied predicate set for some tuple pairs; *count* is the number of tuple pairs that have the same satisfied predicate set. The DFS tree in Figure 3 is a snapshot of the search procedure for minimal set covers of the evidence set. The user can follow the search procedure by dragging the slider at the bottom.



Rule Name: Hospital_DC

Data Soure:
csv_dumptest
csv_hospital_10k
csv_hospital_30k
csv_bank1

Type: DC

Code

```
    @Override
    public Collection<Violation> detect(TuplePair tuplePair) {
        List<Violation> result = Lists.newArrayList();

        Tuple t1 = tuplePair.getLeft();
        Tuple t2 = tuplePair.getRight();
        if (t1.get("race").equals(t2.get("race"))) {
            int wgt1 = Integer.parseInt(t1.get("fnlwgt"));
            int wgt2 = Integer.parseInt(t2.get("fnlwgt"));
            if (Math.abs(wgt1 - wgt2) <= wgt1 * 0.5) {
                Violation violation = new Violation(this.ruleName);
                violation.addCell(t1.getCell("race"));
                violation.addCell(t2.getCell("race"));
```

Fig. 4. Procedural java code for violation detection and repairing.

3. Rules for Violation Detection and Repairing. Finally, we demonstrate how to automatically generate violation detection and repairing java procedural code for discovered DCs by feeding discovered DCs to *Nadeef* data cleaning system [6]. The generated java procedural code can be used by any third-party data cleaning system. Furthermore, expert users can even refine the generated code to go beyond the limitations of DCs language, as shown in Figure 4.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[3] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13), 2013.

[4] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[5] D. Deroos, C. Eaton, G. Lapis, P. Zikopoulos, and T. Deutsch. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill, 2011.

[6] A. Ebaid, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, J.-A. Quiane-Ruiz, N. Tang, and S. Yin. Nadeef: A generalized data cleaning system. *PVLDB*, 6(12), 2013.

[7] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, 2012.