Descriptive and Prescriptive Data Cleaning

Anup Chalamalla[1*] Ihab F. Ilyas[1*] Mourad Ouzzani[2] Paolo Papotti[2]

[1] University of Waterloo, Canada [2] Qatar Computing Research Institute (QCRI), Qatar

ABSTRACT

Data cleaning techniques usually rely on some quality rules to identify violating tuples, and then fix these violations using some repair algorithms. Oftentimes, the rules, which are related to the business logic, can only be defined on some target report generated by transformations over multiple data sources. This creates a situation where the violations detected in the report are decoupled in space and time from the actual source of errors. In addition, applying the repair on the report would need to be repeated whenever the data sources change. Finally, even if repairing the report is possible and affordable, this would be of little help towards identifying and analyzing the actual sources of errors for future prevention of violations at the target. In this paper, we propose a system to address this decoupling. The system takes quality rules defined over the output of a transformation and computes explanations of the errors seen on the output. This is performed both at the target level to *describe* these errors and at the source level to *prescribe* actions to solve them. We present scalable techniques to detect, propagate, and explain errors. We also study the effectiveness and efficiency of our techniques using the TPC-H Benchmark for different scenarios and classes of quality rules.

1. INTRODUCTION

A common approach to address the problem of dirty data [8] is to apply a set of data quality rules or constraints over a target database, to "detect" and to eventually "repair" erroneous data [1, 7, 10, 3, 15]. Tuples or cells (attribute-value of a tuple) in a database D that are inconsistent *w.r.t.* a set of rules Σ are considered to be in violation, thus possibly "dirty". A repairing step tries to "clean" these violations by producing a set of updates over D leading to a new database D' that satisfies Σ. Unfortunately, in many real life scenarios [13], the picture is different, and data and rules are decoupled in space and time; constraints are often declared not on the original data but rather on reports or views, and at a later stage in the data processing life cycle.

Example 1. *Consider the report T (see Figure 1) about shops for an international franchise. The HR department enforces a set of policies in the franchise workforce and identifies two problems in T. The first violation (t_a and t_b, in bold) comes from a rule stating that, in the same shop, the average salary of the managers (*Grd*=2) should be higher than that of the staff (*Grd*=1). The second violation (t_b and t_d, in italic) comes from a rule stating that a bigger shop cannot have a smaller staff.*

T	Shop	Size	Grd	AvgSal	#Emps	Region
t_a	**NY1**	46 ft^2	**2**	**99** $	1	US
t_b	**NY1**	*46* ft^2	***1***	**100** $	*3*	US
t_c	NY2	62 ft^2	2	96 $	2	US
t_d	NY2	*62* ft^2	*1*	90 $	*2*	US
t_e	LA1	35 ft^2	2	105 $	2	US
t_f	LND	38 ft^2	1	65 £	2	EU

Emps	EId	Name	Dept	Sal	Grd	SId	JoinYr
t_1	e4	John	S	91	1	NY1	2012
t_2	e5	Anne	D	99	2	NY1	2012
t_3	e7	Mark	S	93	1	NY1	2012
t_4	e8	Claire	S	116	1	NY1	2012
t_5	e11	Ian	R	89	1	NY2	2012
t_6	e13	Laure	R	94	2	NY2	2012
t_7	e14	Mary	E	91	1	NY2	2012
t_8	e18	Bill	D	98	2	NY2	2012
t_9	e14	Mike	R	94	2	LA1	2011
t_{10}	e18	Claire	E	116	2	LA1	2011

Shops	SId	City	State	Size	Started
t_{11}	NY1	New York	NY	46	2011
t_{12}	NY2	New York	NY	62	2012
t_{13}	LA1	Los Angeles	CA	35	2011

Figure 1: A report T on data sources Emps & Shops.

To explain these errors, we adopt an approach that summarizes the violations in terms of predicates on the database. In the example, since the problematic tuples have Attribute *Region* set to US, we describe (explain) the violations in the example as $[T.Region = US]$. Note that this explanation summarizes all tuples that are involved in a violation, and not necessarily the erroneous tuples; in many cases, updating only one tuple in a violation (set of tuple) is enough to bring the database into a consistent state. For example, a repairing algorithm would identify t_b.Grd as a possible error in the report. Hence, by updating t_b.Grd, the two violations would be removed. Limiting the erroneous tuples can guide us to a more precise explanation. In the example, the explanation $[T.Region = US \wedge T.Shop = NY1]$ is more specific, if we believe that t_b.Grd is the erroneous cell. The process of explaining data errors is two-fold: identifying a set of potential erroneous tuples (cells); and finding concise descriptions

[*] Work partially done while at QCRI.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610520.

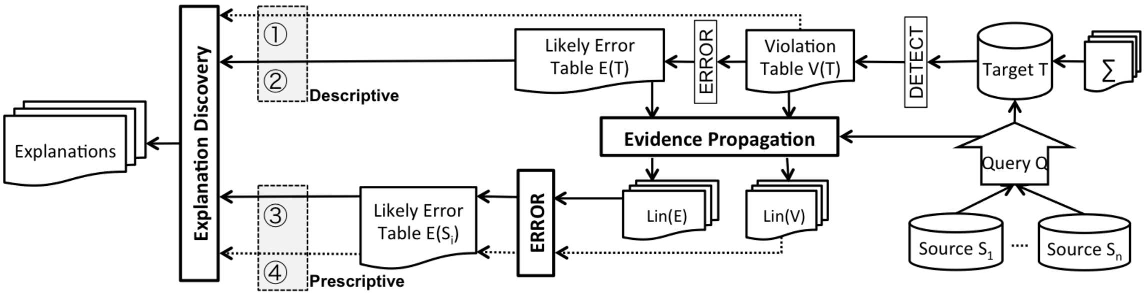


Figure 2: System Architecture.

that summarize these errors and that can be consumed by users or by other analytics layers.

We highlight the problem of explaining errors when errors are identified in a different space and at a later stage than when errors were digitally born. Consider the following query that generated Table T in Example 1. Since violations detected in the report are actually caused by errors that crept in at an earlier stage, i.e., from the sources, propagating these errors from a higher level in the transformation to the underlying sources can help in identifying the source of the errors and in prescribing actions to correct them.

Example 2. *Let us further assume that the previous report T is the result of a union of queries over multiple shops of the same franchise. We focus on the query over source relations Emps and Shops for the US region (Figure 1).*

```
Q: SELECT SId as Shop, Size, Grd, AVG(Sal) as
   AvgSal, COUNT(EId) as #Emps,'US' as Region
   FROM US.Emps JOIN US.Shops ON SId
   GROUP BY SId, Size, Grd
```

We want to trace back the tuples that contributed to the problems in the target. Tuples $t_a - t_d$ are in violation in T and their lineage is $\{t_1 - t_8\}$ and $\{t_{11} - t_{12}\}$ over Tables *Emps* and *Shops*. By removing these tuples from any of the sources, the violation is removed. Two possible explanations of the problems are therefore $[Emps.JoinYr = 2012]$ On Table *Emps*, and $[Shops.State = NY]$ on Table *Shops*.

As we mentioned earlier, t_b is the erroneous tuple that was identified by the repairing algorithm. Its lineage is $\{t_1, t_3, t_4\}$ and $\{t_{11}\}$ over Tables *Emps* and *Shops*, respectively. By focusing on this tuple, we can compute more precise explanations on the sources, such as $[Emps.Dept = S]$. Drilling even further, an analysis on the lineage of t_b may identify t_4 as the most likely source of error since by removing t_4, the average salary goes down enough to clear the violation. Therefore, the most precise explanation is $[Emps.EId = e8]$. The example shows that computing likely errors enables the discovery of better explanations. At the source level, this leads to the identification of **actions** to solve the problem. In the example, the employee with id *e8* seems to be the cause of the problem.

We propose Database Prescription (DBRx for short) (Figure 2), a system to support *descriptive* and *prescriptive* data cleaning. DBRx takes quality rules defined over the output of a transformation and computes explanations of the errors. Given a transformation scenario (sources S_i, $1 < i < n$, and query Q) and a set of quality rules Σ, DBRx computes a violation table VT of tuples not complying with Σ. VT is mined to discover a descriptive explanation ① in Figure 2 such as $[T.Region = US]$. The lineage of the violation table over the sources enables the computation of a prescriptive explanation ④ such as $[Emps.JoinYr = 2012]$ and $[Shops.State = NY]$ on the source tables. When applicable, a repair is computed over the target, thus allowing the possibility of a more precise description ② such as $[T.Region = US \wedge T.Shop = NY1]$, and a more precise prescriptive explanation ③ based on propagating errors to the sources such as $[Emps.Dept = S]$ and $[Emps.EId = e8]$.

Building DBRx raises several challenges: **First**, propagating the evidence about violating tuples from the target to the sources can lead to a lineage that covers a large number of source tuples. For example, an aggregate query would clump together several source tuples, with only few containing actual errors. Simply partitioning the source tuples as dirty and clean is insufficient; tuples do not contribute to violations in equal measure. **Second**, we need a mechanism to accumulate evidences on tuples across multiple constraints and violations and hence identify the most likely tuples with errors. For the target side, there are several repairing algorithms that we can use. But for the source side, a new algorithm, which relaxes the requirements of repair semantics, is needed. **Third**, after identifying the likely errors, mining the explanations involves two issues that we need to deal with: (1) what are the explanations that accurately cover all and only the identified erroneous tuples?; and (2) how to generate explanations concise enough in order to be consumable by humans?

We summarize our **contributions** in this paper as follows:

a. We introduce the problem of descriptive and prescriptive data cleaning (Section 2). We define the notion of explanation, and formulate the problem of discovering explanations over the annotated evidence of errors at the sources (Section 3).
b. We develop a novel weight-based approach to annotate the lineage of target violations in source tuples (Section 4).
c. We present an algorithm to compute the most likely errors in presence of violations that involve large number of tuples with multiple errors (Section 5).
d. We combine multi-dimensional mining techniques with approximation algorithms to efficiently solve the explanation mining problem (Section 6).

We perform an extensive experimental analysis using the TPC-H Benchmark and real-world datasets (Section 7). We conclude the paper with a discussion of related work (Section 8) and of future direction of research (Section 9).

2. PROBLEM STATEMENT

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be the set of schemas of n source relations, where each source schema S_i has d_i attributes $A_1^{S_i}, \ldots, A_{d_i}^{S_i}$ with domains $dom(A_1^{S_i}), \ldots, dom(A_{d_i}^{S_i})$. Let R

be the schema of a target view generated from $\mathcal{S}$. Without loss of generality, we assume that every schema has a special attribute representing the tuple id. A *transformation* is a union of SPJA queries on an instance I of $\mathcal{S}$ that produces a unique instance T of R with t attributes $A_1^T, \ldots, A_t^T$.

Any instance T of a target view is required to comply with a set of data quality rules Σ. We clarify the rules supported in our system in the next Section. For now, we characterize them with the two following functions:

- DETECT(T) identifies cells in T that do not satisfy a rule $r \in \Sigma$, and store them in a violation table $V(T)$.
- ERROR ($V(T)$) returns the most likely erroneous cells for the violations in $V(T)$ and store them in an error table $E(T)$.

While DETECT has a clear semantics, ERROR needs some clarifications. At the target side, we consider the most likely erroneous cells as simply those cells that a given repair algorithm decides to update in order to produce a clean data instance, i.e., an instance that is consistent *w.r.t.* the input rules. Our approach can use any of the available alternative repair algorithms, e.g., [15], (Section 3.3). At the source, we need to deal with the lineage of problematic cells instead of the problematic cells themselves, to produce the most likely erroneous cells. Existing repair algorithms were not meant to handle such a scenario; we show in Section 5 our own approach to produce these cells.

Our goal is to describe problematic data with concise explanations. Explanations are composed of queries over the relations in the database as follows.

Definition 1. *An* explanation *is a set $\mathcal{E}$ of conjunctive queries where $q \in \mathcal{E}$ is a query of k selection predicates $(A_{l_1}^{S_i} = v_{l_1}) \wedge \cdots \wedge (A_{l_k}^{S_i} = v_{l_k})$ over a table S_i with d_i attributes, $1 \leq k \leq d_i$, and v_{l_j} $(1 \leq j \leq k)$ are constant values from the domain of the corresponding attributes. We denote with size($\mathcal{E}$) the number of queries in $\mathcal{E}$.*

There are three requirements for an explanation: (i) coverage - covers error tuples, (ii) conciseness - has small number of queries, and (iii) accuracy - covers mostly error tuples.

Example 3. *Consider again relation Emps. Let us assume that t_1, t_3, t_4, and t_7 are error tuples. There are alternative explanations that cover them. The most concise is exp$_7$:(Emps.Grd=1), but one clean tuple is also covered (t_5). Explanation exp$_8$:(Emps.eid=e$_4$) $\vee$ (Emps.eid=e$_7$) $\vee$(Emps.eid=e$_8$) $\vee$ (Emps.eid=e$_{14}$) has a larger size, but it is more accurate since no clean tuples are covered.*

We define *cover* of a query q as the set of tuples retrieved by q. The cover of an explanation $\mathcal{E}$ is the union of $cover(q_1), \ldots, cover(q_n)$, $q_i \in \mathcal{E}$. For a relation R with a violation table $V(R)$ computed with DETECT, we denote with $\mathcal{C}$ the clean tuples $R\backslash$ ERROR ($V(R)$). We now state the *exact Descriptive and Prescriptive Data Cleaning* (DPDC) problem:

Definition 2 (EXACT DPDC). *Given a relation R, a corresponding violation table $V(R)$, and an* ERROR *function for $V(R)$, a solution for the* exact DPDC problem *is an explanation $\mathcal{E}_{opt}$ s.t.*

$$\mathcal{E}_{opt} = \underset{size(\mathcal{E})}{\operatorname{argmin}}(\mathcal{E} | (cover(\mathcal{E}) = E(R)))$$

If function ERROR over the target is not available (①), the problem is defined on V(R) instead of E(R).

Unfortunately, since all errors must be covered and no clean tuples are allowed in the cover, the exact solution in the worst case does not exist. In other cases, it may be a set of queries s.t. each query covers exactly one tuple. The number of queries in the explanation equals the number of errors (as for exp_8 in Example 3), making the explanation hard to consume.

To allow more flexibility, we drop the strict requirement over the precision of the solution and allow it to cover some clean tuples. We argue that explanations such as exp_7 can better highlight problems over the sources and are easier to consume. More specifically, we introduce a weight function for a query q, namely $w(q)$, that depends on the number of clean and erroneous tuples that it covers:

$$w(q) = |E(R) \setminus cover(q)| + \lambda * |cover(q) \cap \mathcal{C}|$$

where $\mathcal{C}$ is the set of clean tuples, $w(\mathcal{E})$ is the sum $w(q_1) + \ldots + w(q_n)$, $q_i \in \mathcal{E}$ that we want to minimize, and the constant λ has a value in [0,1]. The weight has two roles. First, it favors queries that cover many errors (first part of the weight function) to minimize the number of queries to obtain full coverage in $\mathcal{E}$. Second, it favors queries that cover few clean tuples (second part). Constant λ weighs the relative importance of clean tuples *w.r.t.* errors. In fact, if clean and erroneous tuples are weighted equally, selective queries with $|cover(q) \cap \mathcal{C}| = \emptyset$ are favored, since they are more precise, but they lead to larger size for $\mathcal{E}$. On the contrary, obtaining a smaller explanation justifies the compromise of covering some clean tuples. We set parameter λ to the error rate for the scenario, we shall describe in Section 6 how it is computed. We now state the relaxed version of the problem.

Definition 3 (RELAXED DPDC). *Given a relation R, a corresponding violation table $V(R)$, an* ERROR *function for $V(R)$, and a weight function $w(\mathcal{E})$, a solution for the* relaxed DPDC problem *is an explanation $\mathcal{E}_{opt}$ s.t.*

$$\mathcal{E}_{opt} = \underset{w(\mathcal{E})}{\operatorname{argmin}}(cover(\mathcal{E}) \supseteq E(R))$$

When the DPDC problem is solved over the target (resp. sources), it computes descriptive (resp. prescriptive) explanations. We can map this problem to the well-known weighted set cover problem, which is proven to be an NP-Complete problem [4], where the universe are the errors in $E(R)$ and the sets are all the possible queries over R.

3. VIOLATIONS AND ERRORS

While many solutions are available for the standard data cleaning setting, i.e., a database with a set of constraints, we show in this section how the two levels in our framework, namely target and source, make the problem much harder.

3.1 Data Quality Rules

Quality rules can be usually expressed either using known formalisms or more generally through arbitrary code. We thus distinguish between two classes of quality rules over relational databases.

Examples for the first class are conditional functional dependencies (CFDs) and check constraints (CCs). Since rules in these formalisms can be expressed with the more general class of denial constraints (DCs), we will refer to this language in the following and denote such rules with Σ^D.[1]

[1] Our repair model focuses on detecting problems on the existing data with the big portion of business rules supported

Consider a set of finite built-in operators $\mathbb{B} = \{=, <, >, \neq, \leq, \geq\}$. A DC has the general 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 $v_1 \phi$ *const* with v_1, v_2 of the form $t_x.A, x \in \{\alpha, \beta, \gamma, \ldots\}, A \in R$, and *const* is a constant. For simplicity, we use DCs with only one relation S in $\mathcal{S}$.

Example 4. *The rules in the running example correspond to the following DCs (for simplicity, we omit the universal quantifiers):*
$c_1 : \neg(t_\alpha.Shop = t_\beta.Shop \wedge t_\alpha.Avgsal > t_\beta.Avgsal \wedge t_\alpha.Grd < t_\beta.Grd)$
$c_2 : \neg(t_\alpha.Size > t_\beta.Size \wedge t_\alpha.\#Emps < t_\beta.\#Emps)$

The second class includes rules expressed with arbitrary declarative languages (such as SQL) and procedural code (such as Java programs) [7]. These are alternatives to the traditional rules in Σ^D. We denote these more general rules with Σ^P. Thus, $\Sigma = \Sigma^D \cup \Sigma^P$.

Example 5. *A rule expressed in Java could use an external web service to validate if the ratio of the size of the staff and the size of the shop comply with a local legal policy.*

3.2 Target Violation Detection

Given a set of rules Σ, we require that any rule $r \in \Sigma$ has a function DETECT that identifies groups of cells (or tuples) that together do not satisfy r. We call such set of cells a *violation*. We collect all such violations over T *w.r.t.* Σ in a *violation table* with the schema (vid, r, tid, att, val), where *vid* represents the violation id, r is the rule, *tid* is the tuple id, *att* is the attribute name of the cell, and *val* is the value *tid.att* of that cell. We denote the violation table of a target view T as $V(T)$. We mine $V(T)$ for explanations in case ①.

For DCs in Σ^D, DETECT can be easily obtained. A DC states that all the predicates cannot be true at the same time, otherwise, we have a violation. Given a database instance I of schema $\mathcal{S}$ and a DC φ, if I satisfies φ, we write $I \models \varphi$, and we say that φ is a *valid DC*. If we have a set of DC Σ, $I \models \Sigma$ if and only if $\forall \varphi \in \Sigma, I \models \varphi$.

For rules in Σ^P, the output emitted by the code when applied on the data can be used to extract the output required by DETECT. In Example 5, in case of non compliance with the policy, the cells *Size*, *#Emps* and *Region* will be considered as one violation.

3.3 Target Errors Detection

As we mentioned in the introduction (Example 2), the ability to identify actual errors can improve the performance of the system (case ②). We can rely on the literature on data repairing as a tool to identify the errors in a database. If a cell needs to be changed to make the instance consistent, then that cell is considered as an error.

Repair refers to the process of correcting detected violations. Several algorithms have been proposed for repairing inconsistent data, mainly based on declarative data quality rules (such as in Σ^D) [1, 10, 3]. These rules naturally have a static semantics for violation detection (as described above) and a dynamic semantics to remove them. This can be modeled with a repair function. Given a violation for a certain rule, this function outputs an update to the database to satisfy the violations identified by the corresponding DETECT.

For rules in Σ^P, the repair function must be provided [7]. If such a function is not available (as in many cases), our explanations will be limited to violations and their lineage, cases ① and ④, respectively. For a DC in Σ^D, computing its repair function is straightforward: the repair function is the union of the inverse for each predicate in it.

Example 6. *Given the rules in the running example, a repair function would compute the following updates:*
repair(c_1): $(t_\alpha.Shop \neq t_\beta.Shop) \vee (t_\alpha.Avgsal \leq t_\beta.Avgsal) \vee (t_\alpha.Grd \geq t_\beta.Grd)$
repair(c_2) : $(t_\alpha.Size \leq t_\beta.Size) \vee (t_\alpha.\#Emps \geq t_\beta.\#Emps)$

The repair problem (even with FDs only) is known to have NP complexity [15]. Heuristic algorithms to compute repairs identify the minimal number of cells to change to obtain an instance that conforms to the rules. More precisely, for a violation table $V(T)$ and the repair functions $F = f_1, \ldots, f_n$ for n rules in Σ, REPAIR$(V(T),F)$ computes a set of cell updates on the database s.t. it satisfies Σ. While we are not interested in the actual updates to get a repair, we consider the cells to be updated by the repair algorithm to be the likely errors, therefore ERROR coincides with REPAIR.

3.4 From Target to Sources

We have introduced how violations and errors can be detected over the target. Unfortunately, a target rule can be rewritten at the sources only in limited cases. This is not possible for the rules expressed as Java code in Σ^P as we treat them as black-boxes. For rules in Σ^D, the rewriting depends on the SQL script in the transformation. Rules may involve target attributes whose lineage is spread across multiple relations (as in Example 1), thus the transformation is needed in order to apply them. An alternative approach is to propagate the violations from the target to source at the instance level. However, going from the target to the sources introduces new challenges.

T	Shop	avgHrs
t_a	NY1	23
t_b	NY2	25

Shifts	Sid	Hours	Week	Clerk
t_1	NY1	20	11	John
t_2	NY1	20	11	Anne
t_3	NY1	30	12	Anne
t_4	NY1	30	12	John
t_5	NY1	22	13	John
t_6	NY1	22	13	John
t_7	NY1	17	14	John
t_8	NY2	20	11	Laure
t_9	NY2	30	11	Bill

Figure 3: Average Hours by Shop.

Example 7. *Consider a source relation* Shifts *and a target relation T (Figure 3) obtained with the following query:*

```
SELECT SId as Shop, AVG(Hours) as avgHrs
FROM Shifts where SID like 'NY%'
GROUP BY SId
```

Given the check constraint $\neg(avgHrs < 25)$ over T, tuple t_a is a violation. By removing its lineage ($t_1 - t_7$), the violation is removed. However, we are interested in identifying most likely errors and considering the entire lineage may not be necessary. In fact, it is possible to remove the violation by just removing a subgroup of the lineage. In particular, all the subsets of size 1 to 4 involving t_1, t_2, t_5, t_6, t_7 are possible alternatives, whose removal removes the violation on t_a.

It is easy to see that the lineage of the violation leads to the problematic tuples over the source. Computing a repair

by DCs. However, more complex repair models for missing tuples, such as [11], can be supported with extensions.

on the source requires a new repair algorithm such that by updating some source tuples, the results of the query change and satisfy the constraints. This is always possible, for example by removing the entire lineage. However, similarly to the target level, the traditional concept of minimality can still guide the process of identifying the source tuples that need to change. There are two motivations for this choice. First, treating the entire lineage as errors is far from the reality for a query involving a large number of tuples. Second, considering the entire lineage for explanation discovery will not help in finding meaningful explanations. Unfortunately, it is known that computing all the possible subsets of such lineage is a NP problem even in simpler settings with one SPJU query [5]. We can easily see from the example how the number of subsets can explode.

The above problem shows the impossibility of computing a minimal repair for the target violations over the sources. However, we are interested in identifying the source error tuples in order to discover explanations, not in computing a target repair. Thus, the source Error module will use the minimality principle, without the need to compute a target repair. In Section 4, we introduce scoring functions to quantify the importance of source tuples *w.r.t.* target violations. We then use these scores in two algorithms that return the most likely error source tuples (Section 5).

4. EVIDENCE PROPAGATION

The evidence propagation module involves two tasks. The first task is to trace the lineage of tuples in violations at the target to source tuples. To this end, we implemented inverse query transformation techniques proposed by Cui et al. [6]. The second task is to determine how to propagate violations as evidence over the source.

For each tuple t in a violation $v \in V(T)$, we denote the cells in t that are involved in v as *problematic cells.* These cells are in turn computed from some source cells, also labeled as problematic. To solve a violation, we consider the delete operation over the sources. However, as discussed above, we do not want to identify the minimal groups of problematic source tuples that need to be removed. On the contrary, we take a practical approach. We look at tuples individually by using two scores that quantify the effect of source tuples and source cells in the lineage of each violation.

Cells Contribution. Given a violation v, we want to measure how much the value in each problematic source cell contributes to v. In fact, not all problematic source cells contribute equally to v.

Example 8. *The first violation in Example 1 covers problematic tuples t_a and t_b and the problematic cells over attributes Shop, Grd, AvgSal. The cells are in turn computed from t_{11}.Sid, t_1-t_4.Grd, t_1-t_4.SId, and t_1-t_4.Sal. One of the predicates that trigger the violation is t_b.AvgSal>t_a.AvgSal. Tuple t_b.AvgSal is computed from t_1.Sal, t_3.Sal and t_4.Sal. Among them, the high value of t_4.Sal is a more likely cause for the violation than t_1.Sal or t_3.Sal.*

Tuples Removal. Wrongly joined tuples can trigger an extra tuple in the result of a query, thus causing a violation in the target. We want to measure how much removing a problematic source tuple removes v.

Example 9. *Let us assume that the correct value for t_1.SId is a shop different from NY1, say NY2. Erasing t_1 removes the violation for the second rule in Example 1 (the two stores would have the same number of employees), even though NY1 as a value is not involved in the violation.*

We derive from sensitivity analysis [14] our definitions of *contribution* and *removal* scores. The intuition is that we want to compute the sensitivity of a model to its input. In general, given a function, the influence is defined by how much the output changes given a change in one of the input variables. In our context, the models are the operators in the SQL query, which take a set of source tuples as input and output the problematic tuples in the view.

Definition 4. *A* contribution score *$cs_v(c)$ of a problematic source cell c* w.r.t. *a target violation v is defined as the difference between the original result and the updated output after removing c divided by the number of cells that satisfy the SQL operator.*

A removal score *$rs_v(t)$ of a problematic source tuple t* w.r.t. *a target violation v is 1 if by removing c, v is removed, 0 otherwise.*

A score vector CSV of a cell for contribution scores (RSV of a tuple for removal scores) is a vector $[cs_1, \ldots, cs_m]$ ($[rs_1, \ldots, rs_m]$), where m is the number of violations and $cs_1, \ldots, cs_m \in \mathbb{R}$ ($rs_1, \ldots, rs_m \in \mathbb{B}$). If a problematic cell or tuple does not contribute to a certain violation, we put an empty field in the vector. We will omit the subscript if there is no confusion.

We assume that the transformation is a SPJAU query. We compute CSVs and RSVs using the query tree. For an SPJAU query, every node in the tree is one of the following five operators: (1) selection (S), (2) projection (P), (3) join (J), (4) aggregation (A), and (5) union (U).

Example 10. *Figure 5 shows the query tree for our running example. It has three operators: (1) the $\bowtie$ operator, (2) the* aggregation *operator with the* group by*, and (3) the* projection *on columns Sid, Size, Grd, Region, Sal, and Eid (not shown in the figure for the sake of space).*

4.1 Computing CSVs

We compute CSVs for cells in a top-down fashion over the operator tree. Each leaf of the tree is a source tuple, with its problematic cells annotated with a CSV. Let v be a violation in $V(T)$ on a rule $r \in \sum$. We initialize, cs of each problematic cell in target T to 1. Let I^l be an *intermediate result* relation computed by an operator $O^l \in \{S, P, J, A, U\}$ at level l of the tree, whose input is a non-empty set of intermediate source relations $Inp(O^l) = I_1^{l-1}, I_2^{l-1}, \ldots$. In our rewriting, we compute the scores for problematic cells of $Inp(O^l)$ from the cell scores of I^l.

Let c^l be a problematic cell in I^l, $cs(c^l)$ its contribution score, $val(c^l)$ its value, and $Lin(c^l, I^{l-1})$ its lineage. Procedure 1 computes cs for intermediate cells.

Procedure 1. (Intermediate Cell CS)*: Let I_k^{l-1} be an intermediate relation* contributing *to cell c^l. We have two cases for computing $cs(c^{l-1})$, $c^{l-1} \in Lin(c^l, I_k^{l-1})$:*

(a) If $O^l = A$ (c^l is an aggregate cell) and $r \in \Sigma^D$, then $cs(c^{l-1})$ depends on the aggregate operator op *and on the constraint predicate $P \in r$ being violated, $P : val(ci^l) \phi val(c_0^l)$ with $\phi \in \{<, >, \leq, \geq\}$:*

- *if* op $\in$ $\{avg, sum\}$, *then* $cs(c^{l-1})$ *is* $\frac{val(c^{l-1})}{\sum val(g_i), g_i \in Lin(c^l, I^{l-1})}$ *if* $\phi \in \{<, \leq\}$, *and* $cs(c^l) \cdot (1 - \frac{val(c^{l-1})}{\sum val(g_i), g_i \in Lin(c^l, I^{l-1})})$ *if* $\phi \in \{>, \geq\}$;

I_1^1	Sid [CSV]	Size [CSV]	Grd [CSV]	Sal [CSV]	Eid [CSV]
i_1^1	**NY1** [$\frac{1}{3}$,“”]	*46* ft^2 [“”,$\frac{1}{3}$]	***1*** [$\frac{1}{3}$, $\frac{1}{3}$]	**91** [$\frac{91}{300}$, “”]	*e4* [“”, $\frac{1}{3}$]
i_2^1	**NY1** [1,“”]	46 ft^2	**2** [1,“”]	**99** [0, “”]	e5
i_3^1	**NY1** [$\frac{1}{3}$,]	*46* ft^2 [“”,$\frac{1}{3}$]	***1*** [$\frac{1}{3}$, $\frac{1}{3}$]	**93** [$\frac{93}{300}$, “”]	*e7* [“”,$\frac{1}{3}$]
i_4^1	**NY1** [$\frac{1}{3}$, “”]	*46* ft^2 [“”, $\frac{1}{3}$]	***1*** [$\frac{1}{3}$, $\frac{1}{3}$]	**116** [$\frac{116}{300}$,“”]	*e8* [“”,$\frac{1}{3}$]
i_5^1	NY2	*62* ft^2 [“”,$\frac{1}{2}$]	*1* [“”,$\frac{1}{2}$]	89	*e11* [“”,$\frac{1}{2}$]
i_6^1	NY2	62 ft^2	2	94	e13
i_7^1	NY2	*62* ft^2 [“”,$\frac{1}{2}$]	*1* [“”,$\frac{1}{2}$]	91	*e14* [“”,$\frac{1}{2}$]
i_8^1	NY2	62 ft^2	2	98	e18
i_9^1	LA1	35 ft^2	2	94 $	e19
i_{10}^1	LA1	35 ft^2	2	116 $	e20

Figure 4: Procedure 1 Applied on Intermediate Source I_1^1.

Emps	EId [CSV]	Sal [CSV]	Grd [CSV]	SId[CSV]		[RSV]
t_1	e4 [“”,$\frac{1}{3}$]	91 [$\frac{91}{300}$,“”]	1 [$\frac{1}{3}$, $\frac{1}{3}$]	NY1 [$\frac{1}{3}$,“”]		[0,1]
t_2	e5	99 [0,“”]	2 [1,“”]	NY1 [1,“”]		[1,“”]
t_3	e7 [“”,$\frac{1}{3}$]	93 [$\frac{93}{300}$,“”]	1 [$\frac{1}{3}$, $\frac{1}{3}$]	NY1 [$\frac{1}{3}$,“”]		[0,1]
t_4	e8 [“”,$\frac{1}{3}$]	116 [$\frac{116}{300}$,“”]	1 [$\frac{1}{3}$, $\frac{1}{3}$]	NY1 [$\frac{1}{3}$,“”]		[1,1]
t_5	e11 [“”,$\frac{1}{2}$]	89	1 [“”,$\frac{1}{2}$]	NY2		[“”,0]
t_6	e13	94	2	NY2		[]
t_7	e14 [“”,$\frac{1}{2}$]	91	1 [“”,$\frac{1}{2}$]	NY2		[“”,0]
t_8	e18	98	2	NY2		[]
t_9	e14	94	2	LA1		[]
t_{10}	e18	116	2	LA1		[]

Figure 6: Procedures 1 and 2 Applied on Emps.

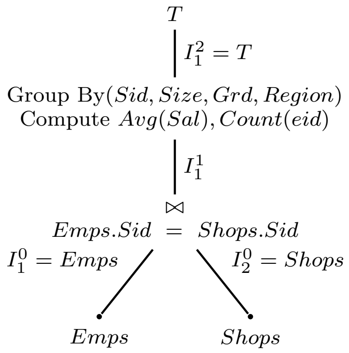


Figure 5: Query Tree.

Shops	SId [CSV]	Size [CSV]		[RSV]
t_{12}	NY1 [2,“”]	46 [“”,1]		[1,1]
t_{13}	NY2	62 [“”,1]		[“”,1]
t_{14}	LA1	35		[]

Figure 7: Procedures 1 and 2 Applied on Shops.

- *if* op $\in$ $\{max, min\}$, *let* $Lin^{\neg P}(c^l, I_k^{l-1}) \subseteq Lin(c^l, I_k^{l-1})$ *be the subset of cells that violate* P, $cs(c^{l-1})$ *is* $\frac{1}{|Lin^{\neg P}(c^l, I_k^{l-1})|}$ *for* $c^{l-1} \in Lin^{\neg P}(c^l, I_k^{l-1})$, *and 0 for all other cells.*

(b) else, $cs(c^{l-1})$ *is* $cs(c^l) \cdot \frac{1}{|Lin(c^l, I_k^{l-1})|}$

Example 11. *Figure 4 reports the CSVs of problematic cells in the intermediate relation* I_1^1. *These are computed by rewriting* I_1^2, *which is* T, *as shown in Figure 5. For example,* $t_b.Grd$ *is computed from cells* $i_1^1.Grd$, $i_3^1.Grd$, *and* $i_4^1.Grd$. *By case (b) these cells get a score of* $\frac{1}{3}$.

Similarly, $t_b.AvgSal$ *is aggregated from* $i_1^1.Sal$, $i_3^1.Sal$, *and* $i_4^1.Sal$, *and* $t_a.AvgSal$ *from* $i_2^1.Sal$. *By case (a) the scores of* $i_1^1.Sal$, $i_2^1.Sal$, $i_3^1.Sal$, *and* $i_4^1.Sal$ *are based on the values of the cells, as shown in Fig. 4. Score of* $i_2^1.Sal$ *is computed as 0 using the first part of case (a).*

Procedure 1 has two cases depending on the query operators and Σ. In case (a), where an aggregate is involved in a violation because of the operator of a rule, we have additional information with regards to the role of source cells in a violation. In case (b), which involves only SPJU operators where the source values are not changed in the target, we uniformly distribute the scores of the problematic cells across the contributing cells. Notice that case (a) applies for $\sum^D$ only, since the actual test in $\sum^P$ is not known. However, case (b) applies for both types of rules.

An intermediate source cell may be in the lineage of several intermediate problematic cells. In this case, their cell scores are *accumulated* by summation following Procedure 2.

Procedure 2. (Intermediate Cell Accumulation)*: Let* $\mathbf{O}^l = O(c^{l-1}, I^l)$ *denote the set of all cells computed from cell* $c^{l-1} \in I_k^{l-1}$ *in the intermediate result relation* I^l *by operator* O, $cs(c^{l-1}) = \sum_{c^l \in \mathbf{O}^l} cs(c^{l-1}, c^l)$.

Algorithm 1: ComputeCSV(T, $V(T)$, $\mathcal{S}$)

```
$O^T \leftarrow$ Operator that generated $T$
$h \leftarrow$ Highest level of query tree
$Inp(O^T) \leftarrow I_1^{h-1}, \dots, I_{r_h}^{h-1}$
$rstate \leftarrow (T, O^T, Inp(O^T))$
$stateStack \leftarrow$new $Stack()$
$stateStack.push(rstate)$
for each violation $v \in V(T)$ do
    while !$stateStack.empty()$ do
        $nextState \leftarrow stateStack.pop()$
        if $nextState[1]$ is $T$ then
            $pcells \leftarrow v^c(T)$ {Problematic cells at T}
        else
            $pcells \leftarrow Lin(v^c(T), nextState(1))$ {Problematic
            cells at an intermediate relation}
        for each cell $c \in pcells$ do
            $computeScores(c, v, l, nextState)$
        for each intermediate relation $I^{l-1} \in nextState[3]$
        do
            Apply Procedure 2 on problematic cells of $I^{l-1}$
            $O^{l-1} \leftarrow$ operator that generated $I^{l-1}$
            $newState \leftarrow (I^{l-1}, O^{l-1}, Inp(O^{l-1}))$
            $stateStack.push(newState)$

function computeScores($c$, $v$, $l$, $nstate$)
for each intermediate relation $I^{l-1} \in nstate[3]$ do
    Apply Procedure 1 on $c, nstate[2], I^{l-1}$
```

Example 12. *In Fig. 7, CSVs of* $t_{12}.SId$ *for the violation between* t_a *and* t_b *are computed from 4 cells in the intermediate relation* I_1^1 *in Figure 4. Cells* $i_1^1.SId$, $i_3^1.SId$, $i_4^1.Sid$ *have a score of* $\frac{1}{3}$ *and* $i_2^1.Sid$ *has a score 1. Procedure 2 computes* $cs(t_{12}.Sid)$ *=2* w.r.t. *this violation.*

Given a target relation T, its violation table $V(T)$ and source relations $\mathcal{S}$, Algorithm 1 computes CSVs of the problematic cells. The algorithm defines a *state* as a triple $(I^l, O^l, Inp(O^l))$. It initializes the root state $(T, O^T, Inp(O^T))$ (line 4), where O^T is the top operator in the tree that computed T. For each violation v and for each problematic cell c, we compute the scores of problematic cells (lineage of c) in all relations in $Inp(O^T)$ (Lines 10-13) using Procedure 1 (Line 24). For each intermediate relation in $Inp(O^T)$, we use Procedure 2 to accumulate the cs scores of each problematic cell and compute its final cs score *w.r.t.* the violation v (Lines 16-17). We then add new states to the stack for each relation in $Inp(O^T)$. The algorithm computes scores all the way up to source relations until the stack is empty, terminating when all the generated states have been visited. Examples of CSVs are shown in Figures 6 and 7.

Once CSVs are computed for cells, we compute them for tuples by summing up the cell scores along the same violation while ignoring non contributing cells.

4.2 Computing RSVs

In contrast to contribution scores, removal scores are directly computed on tuples and are Boolean. If a violation can be eliminated by removing a source tuple, independently of the other tuples, then such a tuple is important. This heuristics allow us to identify minimal subsets of tuples in the lineage of a violation that can solve it through removal. Instead of computing all subsets, checking for each source tuple allows fast computation.

We use a simple bottom-up algorithm to compute RSVs. It starts with the source tuples in the lineage of a violation. For each source relation S and for each problematic tuple $s \in S$, it removes both s and the tuples computed from it in the intermediate relations in the path from S to T in the query tree. If the violation is removed, we assign a score 1 to s_i, 0 otherwise. RSVs for the source relations in the running example are shown in Figures 6 and 7.

5. LIKELY ERRORS DISCOVERY

Given the target violations, we use our scoring methods to identify the most likely errors at the source (scenarios ③ and ④). Since the goal is to correctly separate the potential error tuples from non-error tuples, we use the intuition that most likely errors are expected to have higher scores. A top-k analysis of the tuples' scores for each violation can identify potential errors. However, there is no k that works for all scenarios.

We present two approaches to solve this problem. In the first approach, we design an outlier function to separate high and low scoring tuples for each violation. In the second approach, we show a reduction from the facility location problem and apply a polynomial time $\log n$-approximation algorithm to compute the likely source errors [12].

5.1 Distance Based Local Error Separation

In several cases (such as queries with aggregates), source tuples in the lineage of a violation consist of a subset of tuples that have high scores based on our scoring model, while the remaining have low scores. To precisely measure the distance between tuples, we define it as follows.

Definition 5. *Given two source tuples s_1 and s_2 in v, we define their* distance *as:*

$$D(s_1, s_2) = |(cs_v(s_1) + rs_v(s_1)) - (cs_v(s_2) + rs_v(s_2))|$$

Two tuples with high scores are expected to have a smaller distance between them than the distance between a high-scoring tuple and a low-scoring one. Our goal is to obtain an optimal separation between high- and low-scoring tuples.

Let H_v be the set of high-scoring tuples and L_v the set of low-scoring ones. Intuitively, a separation is preferable to another one if by adding a tuple $s \in L_v$ to H_v, the difference between the sum of pair-wise distances among all tuples of $H_v \cup \{s\}$ and the sum of their scores becomes smaller. The intuition is clarified in the following gain function.

Definition 6. *Let the score of a tuple s_i for violation v be $c_v(s_i) = (cs_v(s_i) + rs_v(s_i))$. Let $Lin(v, S)$ consists of the lineage tuples of v in S and L_v be a subset of $Lin(v, S)$. We define the* separation gain *of L_v as:*

$$SG(L_v) = \sum_{s \in L_v} (c_v(s)) - \sum_{1 \leq j < |L_v|} \sum_{j < k \leq |L_v|} D(s_j, s_k)$$

We define an optimal separation as the one that maximizes this function for H_v.

Example 13. *Consider six source tuples for a violation v having scores $\{s_1$:0.67, s_2:0.54, s_3:0.47, s_4:0.08, s_5:0.06, s_6:0.05$\}$. The sum of pair-wise distances for $H_v = \{s_1, s_2, s_3\}$ is 0.24, while the sum of scores is 1.68, thus $SG(H_v)$=1.44. If we add s_4 to H_v, the pair-wise distances of $H'_v : \{s_1, s_2, s_3, s_4\}$ raises to 1.67 and the sum of scores to 1.76. Clearly, this is not a good separation, and this is reflected by the low gain $SG(H'_v)$=0.08. Similarly, if we remove s_3 from H_v the new SG also decreases to 1.14.*

As it is exponential in the number of subsets to obtain an optimal separation, we provide a greedy heuristic to compute its approximation using ideas from the nearest neighbor chain algorithm for agglomerative clustering [18]. We first order all the tuples in $Lin(v, S)$ in the descending order of their scores, and designate each tuple as its own cluster. We start with the highest scoring tuple's cluster, and keep adding to it the next tuple in the order, while computing the separation gain at each step. We terminate after reaching a separation where the gain attains a local maximum. We generate two clusters, the cluster that is being extended and the subset of tuples that are not in this cluster. In Example 13, the gain after adding s_1, s_2, and s_3 is 0.6, 1.14, and 1.44, respectively. After adding s_4, the gain becomes 0.08 and therefore we stop at s_3. From each violation v, its H_v is added to the set of most likely source error tuples. The algorithm requires a linear space and quadratic time (pair-wise distances) in the number of tuples.

5.2 Global Error Separation

Since we have multiple violations, instead of looking at scores locally per violation, we introduce an alternative approach that looks for the most likely error tuples globally. We accumulate evidences coming from multiple violations as in the following example.

Example 14. *Consider two violations v_1 and v_2, and four source tuples s_1–s_4. Let the scores of the tuples be v_1: (s_1[0.8], s_2[0.1], s_3[0.1]), v_2: (s_3[0.5], s_4[0.5]). Here, s_1 is* *the most likely error tuple for v_1 and s_3 is the one for v_2 as it is the one that contributes most over the two violations.*

The goal is to select a subset of tuples that globally contribute the most to the violations. We can formulate this problem using the known NP-Hard uncapacitated facility location problem (FLP) [17]. The uncapacitated facility location problem is described as follows.

a. a set $\mathcal{Q} = \{1, \ldots, n\}$ of potential sites for locating facilities,

b. a set $\mathcal{D} = \{1, \ldots, m\}$ of clients whose demands need to be served by the facilities,

c. a profit c_{qd} for each $q \in \mathcal{Q}$ and $d \in \mathcal{D}$ made by serving the demand of client d from the facility at q,

d. a non-negative cost f_q for each $q \in \mathcal{Q}$ associated with opening the facility at site q.

The objective is to select a subset $Q \subseteq \mathcal{Q}$ of sites to open facilities and to assign each client to exactly one facility s.t. the difference of the sum of maximum profit for serving each client and the sum of facility costs is maximized, i.e.,

$$\underset{Q \subseteq \mathcal{Q}}{\operatorname{argmax}}\left(\sum_{d \in \mathcal{D}} \max_{q \in Q}(c_{qd}) - \sum_{q \in Q} f_q\right)$$

We obtain a reduction from the FLP to the problem of computing most likely errors in PTIME in the number of violations and in the number of source tuples. For each client d, we associate a violation $v_j \in V(T)$. Let $Lin(V(T), S) = \cup_{v_j \in V(T)} Lin(v_j, S)$, $n = |Lin(V(T), S)|$, and $m = |V(T)|$. For each site q, we associate a source tuple in $Lin(V(T), S)$. For each tuple s in $Lin(v_j, S)$, we associate the cost c_{qd} between site q (s) and client d (v_j) with the score $(cs_j(s) + rs_j(s))$. We assume the fixed cost f_q of covering a source tuple to be 1. A solution to our problem is optimal if and only if a solution to the facility location problem is optimal. We present a greedy heuristic [17] for this problem as follows.

We start with an empty set Q of tuples, and at each step we add to Q a tuple $s \in Lin(V(T), S) \setminus Q$ that yields the maximum improvement in the objective function:

$$f(Q) = \sum_{d \in \mathcal{D}} \max_{q \in Q}(c_{qd}) - \sum_{q \in Q} f_q$$

For a tuple $s \in Lin(V(T), S)) \setminus Q$, let $\Delta_s(Q) = f(Q \cup \{s\}) - f(Q)$ denote the change in the function value. For a violation v_j, let $u_j(Q)$ be $\max_{s \in Q}(cs_j(s) + rs_j(s))$, and $u_j(\emptyset) = 0$. Let $\delta_{js}(Q) = cs_j(s) + rs_j(s) - u_j(Q)$. Then, we write $\Delta_s(Q)$ as follows:

$$\begin{aligned}\Delta_s(Q) &= f(Q \cup \{s\}) - f(Q) \\ &= \sum_{v_j \in V(T)} \left(\begin{cases} \delta_{js}(Q) & \text{if } \delta_{js}(Q) > 0 \\ 0 & \text{otherwise} \end{cases}\right) - 1\end{aligned} \tag{1}$$

The -1 corresponds to the cost of covering a tuple. In each iteration, of the heuristic, $\Delta_s(Q)$ is computed for each $s \in Lin(V(T), S)) \setminus Q$. We add a tuple s whose marginal cost $\Delta_s(Q)$ is maximum. The algorithm terminates if either there are no more tuples to add or if there is no such s with $\Delta_s(Q) > 0$.

The algorithm identifies tuples whose global (cumulative) contributions (to all violations) is significantly higher than others. This global information leads to higher precision compared to the distance based error separation, but to a lower recall if more than one tuple is involved in a violation. Favor precision *w.r.t.* to recall is desirable, as it is easier to discover explanations from fewer errors than discover them from a mix of error and clean tuples. This will become evident in the experiments.

6. EXPLANATION DISCOVERY

The problem of explanation discovery pertains to selecting an optimal explanation of the problematic tuples from a large set of candidate queries. An optimal explanation covers the most likely error tuples, while minimizing the number of clean tuples being covered and the size of the explanation.

We compute optimal explanations in two steps. We first determine candidate queries. We then use a greedy algorithm for the weighted set cover, with weights based on the function over query q defined in Section 2.

Candidate Queries Generation The goal is to generate the candidate queries for a source S with d dimensions. The algorithm first generates all queries with a single predicate for each attribute A^l of R, s.t. the queries cover at least one tuple in $E(R)$. A data structure $P[1..d]$ is used to store the queries of the respective attributes. The algorithm then has a recursive step in which queries of each attribute (A^{l_0}) are expanded in a depth-first manner by doing a conjunction with queries of attributes $A^l \ldots A^d$ where $l = l_0 + 1$. The results of the queries are added to a temporary storage P' and are expanded in the next recursive step.

Computing Optimal Explanations In the second stage, we compute the optimal explanation from the generated candidate queries. In Section 2, we defined the weight associated with each query as follows.

$$w(q) = |E(R) \setminus cover(q)| + \lambda * |cover(q) \cap \mathcal{C}|$$

Our goal is to cover in $\mathcal{E}$ the tuples in $E(R)$, while minimizing the sum of weights of the queries in $\mathcal{E}$. An explanation is optimal if and only if a solution to the weighted set cover is optimal. By using the greedy algorithm for weighted set cover [4], we can compute a $\log(|E(R)|)$-approximation to the optimal solution. The explanation is constructed incrementally by selecting one query at a time. Let the marginal cover of a new query q *w.r.t.* $\mathcal{E}$ be defined as the number of tuples from (R) that are in q and that are not already present in $\mathcal{E}$:

$$mcover(q) = (q \cap E(R)) \setminus (\mathcal{E} \cap E(R))$$

Algorithm 2: GreedyPDC(candidate queries $\mathcal{P}$ over R)

```
$\mathcal{E}_{opt} \leftarrow \{\}$
$bcover(\mathcal{E}_{opt}) \leftarrow \{\}$
while $bcover(\mathcal{E}_{opt}) < |E(R)|$ do
    $minCost \leftarrow \infty$
    $min\_q \leftarrow null$
    for each query $q \in \mathcal{P}$ do
        $cost(q) \leftarrow \frac{w(q)}{mcover(q)}$
        if cost($q$)$\leq minCost$ then
            if cost($q$)$= minCost$ and
            $bcover(q) < bcover(min\_q)$ then
                continue to next query
            $min\_q \leftarrow q$
            $minCost = cost(q)$
    Add $min\_q$ to $\mathcal{E}_{opt}$
    $bcover(\mathcal{E}_{opt}) \leftarrow bcover(\mathcal{E}_{opt}) \cup bcover(min\_q)$
```

At each step, Algorithm 2 adds to $\mathcal{E}$ the query that minimizes the weight and maximizes the marginal cover. Let $\text{bcover}(q) = E(R) \cap cover(q)$, similarly for $\text{bcover}(\mathcal{E}_{opt})$.

Parameter λ weighs the relative importance of the clean tuples *w.r.t.* errors. In practice, the number of errors in a database is a small percentage of the data. If clean and erroneous tuples are weighted equally in the weight function, selective queries that do not cover clean tuples are favored. This can lead to a large explanation size. We set the parameter λ to be the error rate, as it reflects the proportion between errors and clean tuples. If the error rate is very low, it is harder to get explanation with few clean tuples, thus we give them a lower weight in the function. If there are many errors, clean tuples should be considered more important in taking a decision. For mining at the source level (③ and ④ in Figure 2), we estimate the error rate by dividing the number of likely errors by the number of tuples in the lineage of the transformation (either violations or errors from the target).

7. EXPERIMENTS

An end-to-end evaluation of our system requires a setup with one or more source relations and a set of target schemas on which business rules can be defined. In Sec. 7.1, we test the quality of error and explanation discovery modules with datasets from the TPC-H benchmark. In Sec. 7.2, we compare the explanations computed by DBRx against two alternative systems on five real-world datasets.

7.1 Synthetic Dataset

The TPC-H Benchmark data generator defines a general schema typical of many businesses. We picked two representative queries as target schemas, namely Q3 and Q10[2], and defined three scenarios with the following rules in Σ^D:

Scenario S1. $c_{Q10} : \urcorner(t_\alpha.revenue > \delta_1)$.
Scenario S2. $c'_{Q10} : \urcorner(t_\alpha.name = t_\beta.name \wedge t_\alpha.c\_phone[1,2] \neq t_\beta.c\_phone[1,2])$.
Scenario S3. $c_{Q3} : \urcorner(t_\alpha.revenue > \delta_2)$,
$c'_{Q3} : \urcorner(t_\alpha.o\_orderdate = t_\beta.o\_orderdate \wedge t_\alpha.o\_shippriority \neq t_\beta.o\_shippriority)$.

Rules c_{Q10} and c_{Q3} are check constraints over one tuple, while c'_{Q10} and c'_{Q3} are FDs over pairs of tuples. In these scenarios, we focus on Σ^D to show the impact of (i) the repair computation over the target and of (ii) the role of ERROR in the source. We use Σ^P in the real-data study.

Error Induction on TPC-H. We generate instances and queries using *dbgen* and *qgen* tools, respectively. We assign values to parameters in the rules s.t. the reports have no violations. We then add errors in the source relations s.t. the reports have violations when recomputed.

Each experiment has a source instance D, a transformation Q, and target rules Σ. We identify candidate source attributes A based on the lineage of the attributes in the rule. Since our goal is to explain errors, we induce errors s.t. they happen on tuples covered by some pre-set explanations, or *ground explanations*. This allows us to test how good we are at recovering these explanations. We induce errors for a given ground explanation over the source, such as $\mathcal{E}_g = \{q_1 : (lineitem.l\_ship = R), q_2 : (lineitem.l\_ship = S)\}$, while enforcing that attributes in $\mathcal{E}_g$ are not in A.

We consider two parameters for inducing errors *w.r.t.* ground explanation $\mathcal{E}_g$: source error rate e (ratio of number of error tuples to $|Lin(V(T))|$), and explanation rate n (a fixed percentage of $e \cdot |Lin(V(T))|$). Error rate e corresponds to the total number of errors to induce, while n corresponds to the number of such error tuples that should satisfy the ground explanation. The remaining errors, i.e., $(1-n) \cdot e \cdot |Lin(V(T))|$, are induced on random tuples which do not match the ground explanation $(Lin(V(T)) \setminus \mathcal{E}_g)$.

For each $r \in \Sigma^D$ and for each predicate $P \in r$, we identify the corresponding attributes A_P and tuples in $\text{cover}(\mathcal{E}_g)$, and modify their values up to the budget of errors. We make sure that errors are detectable over the target schema. In scenarios S1 and S2, one error tuple at the source is sufficient to detect a target violation, while for S3 we need to introduce two or three errors to induce a target violation.

Metrics. We introduce two metrics to test DBRx. For each metric, we show how to compute precision(P) and recall (R).

Error Discovery Quality – evaluates the quality of the likely errors discovery and the scoring. We compare the errors computed by ERROR over the lineage versus the changes introduced in the errors induction step ($\mathcal{B}$).

$$P_{Err} = \frac{E(T) \cap \mathcal{B}}{E(T)} \quad R_{Err} = \frac{E(T) \cap \mathcal{B}}{\mathcal{B}}$$

Explanation Quality – evaluates the quality of the discovered explanations. We measure the quality of an explanation computed by DBRx by testing the tuples overlap with the ground explanations.

$$P_{\mathcal{E}} = \frac{cover(\mathcal{E}_{opt}) \cap cover(\mathcal{E}_g)}{cover(\mathcal{E}_{opt})} \quad R_{\mathcal{E}} = \frac{cover(\mathcal{E}_{opt}) \cap cover(\mathcal{E}_g)}{cover(\mathcal{E}_g)}$$

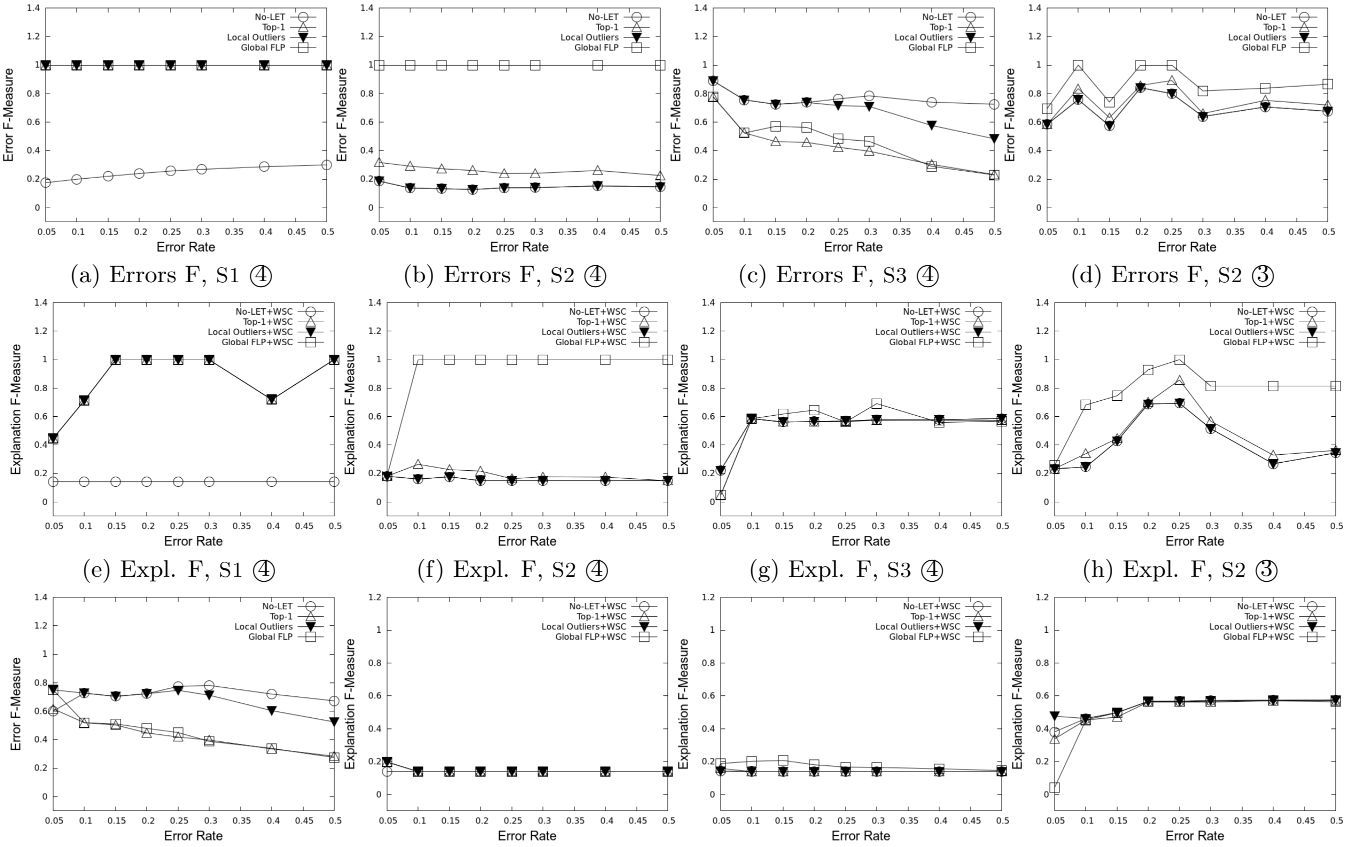
Algorithms. We implemented the algorithms introduced in the paper and baseline techniques to compare the results. For scoring, we use the technique based on outliers detection (LOCAL OUTLIERS) and the one based on the facility location problem (GLOBAL-FLP). As baselines, we consider all the tuples in the lineage with the same base score 1 (NO-LET), and the tuple(s) with the highest score for each violation (TOP-1). For explanation discovery, we implemented Algorithm 2.

Results. We discuss three experiments designed to measure the effectiveness and efficiency of the modules in DBRx. Since we can compute target repairs for these scenarios, we discuss mining on propagated target errors (③) and mining on propagated target violations (④). All measures refer to the relations where errors have been introduced.

Experiment A: Quality of Error Discovery. We start testing the quality of the alternative ERROR implementations. For space reason we report the error F-measure.

In **ExpA-1**, we fix the queries in the ground explanation and increase the error rate without any random errors ($n = 1$). We start by discussing the results for the case of the rewriting of the target violations (④). Figure 8a shows that all the methods perform well for S1, with the exception of NO-LET. This shows that computing errors is easy when there is only a simple check constraint over an aggregate value. Figure 8b shows that only GLOBAL-FLP obtains high F-measure with S2. This is because it uses the global information given by the many pair-wise violations. Figure 8c shows that for S3, which has multiple constraints and multiple errors, the methods obtain comparable results. However, a close analysis shows that, despite that having

[2] The TPC-H documentation [20] contains the SQL code.



(a) Errors F, S1 ④ (b) Errors F, S2 ④ (c) Errors F, S3 ④ (d) Errors F, S2 ③

(e) Expl. F, S1 ④ (f) Expl. F, S2 ④ (g) Expl. F, S3 ④ (h) Expl. F, S2 ③

(i) Err. F, Random, S3 ④ (j) Expl. F, Random, S1 ④ (k) Expl. F, Random, S2 ④ (l) Expl. F, Random, S3 ④

Figure 8: Experimental results for the prescriptive data cleaning problem.

multiple errors violate the hypothesis of Global-FLP, it still achieves the best precision, while the best recall is obtained by No-LET. Figure 8d shows again S2, but computed on the rewriting of the target repair (③). Compared to Figure 8b, Global-FLP does slightly worse, while all the others improve. This reflect the effect of the target repair at the source level: it gives less, but more precise, information to the error discovery. This provides less context to Global-FLP, which takes advantage of the larger amount of evidence in ④. Similar behavior is observed in S3.

In **ExpA-2**, we study how having 50% random errors affects the quality of error discovery. Figure 8i reports the error F-measure for S3. Despite the random errors, the results do not differ much from the simpler scenario in Figure 8c. We observed similar results for S1 and S2.

Experiment B: Quality of Explanations. We test the quality of the explanations with different Error modules.

In **ExpB-1**, we study how increasing the error rate affects the results. Figure 8e shows that all the errors detection methods have similar results for S1, with the exception of No-LET. This is not surprising, as in this scenario errors are easy to identify and the size of the aggregate is large. Figure 8f reflects the quality of the error detection of Global-FLP (as seen in Figure 8b) on the quality of the explanation for S2 8g shows that the precision in error detection has higher impact than the recall for the discovery of explanation. As discussed for Figure 8c, Global-FLP has the highest precision for S3, but the lowest recall. This shows that is better to identify fewer errors with higher confidence. Figure 8h shows the explanation F-measure for S2 on the rewriting of the target repair (③). Compared to Figure 8f, most of the methods improved their quality, while Global-FLP's quality decreased. This is a direct consequence of the detect error quality shown in Figure 8d. Examples of ground and discovered explanations are reported in Figure 9.

In **ExpB-2**, we study how having 50% random errors affects the quality of the discovered explanations. Figures 8j and 8k show the explanation's F-measure for scenarios S1 and S2, respectively. Despite the error discovery did not change much with the random noise, the discovery of explanation is affected in these two scenarios, as it is clear from the comparison with Figures 8e and 8f. This behaviour shows that the quality of the explanations is only partially related to the error detection and that a large amount of errors that cannot be explained can make hard the discovery of existing explanations. Fortunately, results on real data show that useful explanations can still be discovered. Moreover, Figure 8l shows consistent result for S3 *w.r.t.* the case without random errors (Figure 8g). This shows that the accumulation of errors from two different quality rules has a strong effect even in noisy scenarios.

Experiment C: Running Time. We measured the average running time for TPC-H data of size 10 and 100 MB. For the 100 MB dataset and S1, the average running time across different error rates is 100.29 seconds for rewriting the violations and computing their score. The average running time for the Error function is less than 2 seconds. The pattern mining including the candidate pattern generation

Exp.	Ground Explanation	No-LET	Top-1	Local Outliers	Global FLP
S1 ④	• $l\_shipmode$ = RAIL ∧ $l\_shipinstruct$ = TBR • $l\_shipmode$=SHIP ∧ $l\_shipinstruct$=DIP	• $l\_returnflag = R$	• $l\_shipmode = RAIL$ ∧ $l\_shipinstruct$ =TBR • $l\_shipmode = SHIP$ ∧ $l\_shipinstruct$ =DIP	• $l\_shipmode = RAIL$ ∧ $l\_shipinstruct$ =TBR • $l\_shipmode = SHIP$ ∧ $l\_shipinstruct$ =DIP	• $l\_shipmode = RAIL$ ∧ $l\_shipinstruct$ =TBR • $l\_shipmode = SHIP$ ∧ $l\_shipinstruct$ =DIP
S2 ③	• $c\_mktsegment = HOUSE \wedge c\_author = a_1$ • $c\_mktsegment = AUTO \wedge c\_author = a_2$	• $c\_nationkey = 3$ • $c\_nationkey = 20$ • $c\_nationkey = 16$	• $c\_nationkey = 3$ • $c\_nationkey = 20$ • $c\_nationkey = 16$	• $c\_nationkey = 3$ • $c\_nationkey = 20$ • $c\_nationkey = 16$	• $c\_mktsegment = HOUSE \wedge c\_author = a_1$ • $c\_mktsegment = AUTO \wedge c\_author = a_2$

Figure 9: Explanation output for scenarios S1 and S2.

took 52 seconds. The results for S2 and 100 MB vary only in the rewriting module, as it took 430 seconds because of the large number of pair-wise violations. The execution times for 10 MB are at least 10 times smaller with all modules.

7.2 Real Data

We run DBRx using the Global-FLP technique on five real-world scenarios with different types of data quality rules. In some of the scenarios, we also compare DBRx with Scorpion [21] and the technique on tracing data errors [16]. Since ground explanations are not available, we measure the output quality only in terms of precision of the explanations. We manually mark an explanation as correct based on our own knowledge of the data. The precision is the number of correct queries in the optimal explanation over the total number of queries in the explanation.

Datasets and Data Quality Rules on Target. The five scenarios we evaluate are described as follows:

t_sensors [21] The source consists of sensor data with 2.3M tuples over 7 attributes. The target is a transformation that averages temperatures grouped by hour over a selection of dates. The rule over the target is a check constraint ($avg(temperature) < 23$).

elections [21] The source contains 18 months campaign expenses from the 2012 US Presidential Election in a 14 attributes and 116K tuples table.The target reports total expenses of Barack Obama on each date, and the rule constrains this amount to be less than $10M on any date.

p_sensors [16] In this scenario, measurements of nine mobile phone sensors are recorded and classifiers are defined to determine five Boolean variables. The classifiers act as transformations and the measurements as source data. Classifiers that do not determine the variables correctly due to input errors are identified by comparing their output against a ground truth with a rule expressed as a Java code in Σ^P.

stocks We constructed two tables about stocks. The first, namely Q, contains daily stock quotes of $S\&P500$ companies for a two-month period from a trusted source with 30K tuples. The second table is built by extracting mentions of companies and their stock quotes from 45k Bloomberg articles during the same time period. On every page, we ran two regular expression based extractors (E1 and E2) and an induction based one (E3). We mapped the two sources to the target schema $[company, date, stock\_price, src]$, where the attribute *src* was either *extracted* or *master*. We cleaned the output of E2 to ensure that its tuples are correct. A target rule in Java code (Σ^P) states that two tuples are in violation if, given the same company and the same date, the difference among the price values is higher than 10% of the one coming from Q.

players In this scenario, we obtained data with information about soccer players from 6 web sites. The data has 7 attributes. The transformation is the union of the 6 sources with an attribute *source* that keeps track of the source relation. The target rule is $name \rightarrow birthdate$.

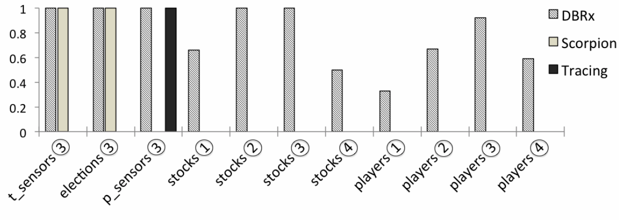


Figure 10: Precision of explanations with real datasets.

Algorithms and Results. Figure 10 shows the results we obtained by running DBRx on the five scenarios and for different cases. For *t_sensors* and *elections*, we compare the output of our mining at the source (case ③) with Scorpion [21]; we both achieve the same results. For *t_sensors*, the explanation responsible for high temperatures is *sensorid=15*. For *elections*, the explanation for high expenses is *recipient_st='DC'* ∧ *recipient_nm='GMMB INC.'*. For *p_sensors*, we could only obtain a fragment of the data used in [16]. This data induces errors in one of the observations from the sensor "gps". We were able to retrieve this pattern by applying our techniques with the same performance of the system in [16].

In scenarios *stocks* and *players*, we mine both the target and the source and present their results in Fig 10, thus showing all of cases targeted by DBRx. Alternative techniques [22, 21, 16] do not apply here, since we have arbitrary SQL in the transformation, and declarative constraints as target rules, thus the ground truth is not available.

For *stocks*, the system is able to compute explanations with perfect precision when mining on errors $E(T)$, both for cases ② (*src=extracted*) and ③ (*extractor=E1* ∧ *extractor=E3*), while some mistakes are made when mining the violation ($V(T)$), in both cases ① and ④. This reflects the impact of a repair of very high quality at the target, which is possible because a reliable source is available.

Things change slightly for the more complicated case of *players*. Here there are six sources that often disagree and may fail to form clear majorities over correct values for the birthdate of a player. This is reflected in a low precision in all cases, but again cases ② and ③ show the positive impact of the error computation at the target. Examples of correct explanations at the source level are *src=espn* ∧ *birthdate=0* and *src=footymania* ∧ *birthdate=0/-1/2000.*

8. RELATED WORK

Provenance. Several proposals tackled the problem of verifying the semantic correctness of data transformations by pointing at anomalies in the results. These have been mainly termed as "Why questions" [2] and "Why-Not questions" [11, 19]. In the first case, the system finds the origin of some tuples or cells in the results. Provenance can be useful to extend DBRx to transformations expressed as black boxes. In particular, a transformation system (e.g., a Java program) supporting the eager approach (aka bookkeeping) carries extra annotations that can be used to produce evidence tables. Our rewriting technique is a lazy approach to provenance that can be readily deployed on an existing system supporting SPJUA queries.

In the second case, we look for explanations about tuples that were expected but are missing from the results. Two models have been proposed for this case. One adjusts the query to provide the desired output [19]. Such model does not apply to DBRx because we trust the transformations. The other model explains a missing tuple t in terms of insertions to the database s.t. t appears in the result. We can extend DBRx with this model by allowing inclusion dependencies in Σ^D and implementing existing algorithms [11] in the evidence propagation module.

Causality. There have been proposals to discover explanations to problematic values in the results of an aggregate query [22, 21] or of a transformation process [16]. These share the same goals as our proposal. However, they have limitations that limit their applicability. Scorpion [22, 21] works on aggregate queries only, and target constraints are limited to one tuple check constraint with a variable and a constant. It lacks the support for arbitrary Σ and arbitrary SQL, which are contributions of our work. CARE [16] requires the availability of the ground truth in order to detect errors. This is not realistic in a data cleaning setting. Moreover, it requires lineage information and it does not tackle the problem of propagating the evidence to the sources.

Similar attempts over probabilistic databases (e.g., [14]) also rank "sensitive" individual tuples by interest. However, they do not construct explanations based on predicates.

Dependency Propagation. The problem of propagating dependencies is to determine, given a target view over the sources and their dependencies, if another dependency holds on the view. We address the inverse process with SPJAU views; this is not supported by existing approaches and leads to undecidability [9]. Moreover, our instance-driven rewriting allows extension for scenarios where the transformation is not a query, but a black box with provenance.

View Updates. In the view update problem, the goal is to minimize the number of tuples to delete in the sources, such that the desired change in the target is obtained and no other target tuples are modified. The side-effect free variant of this problem is related to our rewriting from target to source. Unfortunately, the problem is intractable even for views defined in terms of simple SPJU queries [5]. This intractability motivated our scoring scheme; we drop the requirement to solve the view update in an exact fashion and opt for scores that can be computed efficiently.

Data Cleaning. Data cleaning focuses on detecting and repairing errors on a database by using declarative constraints [3, 15, 7, 10, 1]. In our target Error module, we can use any of these algorithms. However, they rely on properties of the violations that do not apply when the violations are rewritten over the sources. Thus, they cannot be used at the source Error module. Extending them to compute a target repair through updates on the sources requires to solve the view update problem and is thus not tractable.

9. CONCLUSIONS

Given a view over sources and a set of quality rules over it to identify violations, we introduced explanations both at the target and at the source levels for the problematic data. To make these explanations easy to consume for users, we formulated a problem that minimise their size while guaranteeing coverage of the violations. The main intuitions behind this work are that (i) violations at the target level can be expressed as evidence of problems over the sources and (ii) summarising such evidence leads to meaningful explanations of the problems. We plan to extend this work by considering multi-level transformations, such as ETL processes, where at each step rules for data cleaning can be enforced.

10. REFERENCES

[1] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3(1):197–207, 2010.

[2] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

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

[4] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[5] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the complexity of view update analysis and its application to annotation propagation. *IEEE TKDE*, 24(3):506–519, 2012.

[6] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.

[7] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. Ilyas, M. Ouzzani, and N. Tang. Towards a commodity data cleaning system. In *SIGMOD*, 2013.

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

[9] W. Fan, S. Ma, Y. Hu, J. Liu, and Y. Wu. Propagating functional dependencies with conditions. *PVLDB*, 1(1):391–407, 2008.

[10] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The Llunatic Data-Cleaning Framework. *PVLDB*, 6(9):625–636, 2013.

[11] M. Herschel and M. A. Hernández. Explaining missing answers to spjua queries. *PVLDB*, 3(1):185–196, 2010.

[12] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical programming*, 22(1):148–162, 1982.

[13] W. H. Inmon. *Building the Data Warehouse*. John Wiley Publishers, 2005.

[14] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, pages 841–852, 2011.

[15] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.

[16] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *SIGMOD*, pages 505–516, 2011.

[17] P. B. Mirchandani and R. L. Francis. *Discrete location theory*. 1990.

[18] F. Murtagh. Clustering in massive data sets. In *Handbook of massive data sets*, pages 501–543. Springer, 2002.

[19] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[20] Transaction Processing Performance Council. The TPC Benchmark H 2.16.0. *http://www.tpc.org/tpch*, 2013.

[21] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[22] E. Wu, S. Madden, and M. Stonebraker. A demonstration of dbwipes: Clean as you query. *PVLDB*, 5(12):1894–1897, 2012.