Qualitative Cleaning: Data Repairing

Data Cleaning & Integration
CompSci 590.01 Spring 2017

Some contents were based on: Cong et al.’s VLDB 2017 slides
From profiling to repairing

• Last lecture: use data profiling to discover constraints on data

• This lecture: given constraints, “repair” data to make it conform to the constraints
Focus of today

• Cong et al. “Improving data quality: consistency and accuracy.” VLDB 2007
  • Data repairing under CFD
  • Batch vs. incremental
  • Sampling + human in the loop

• Overview of other repairing work
  • Other data repairing objectives
  • Repairing constraints + data jointly
  • Repairing under different types of constraints
A cost-based problem formulation

• Input: set $\Sigma$ of constraints + database instance $D$
• Output: a “repair” $D'$ such that
  • $D' \models \Sigma$; i.e., $\Sigma$ holds on $D'$
  • $\text{cost}(D, D')$ is minimized
    • What’s a trivial repair without considering this criteria?

• A possible cost($\cdot, \cdot$) formulation
  • Each cell has a weight in $[0,1]$: larger means higher penalty for repairing this cell
  • Further define a distance function for each value domain (e.g., string edit distance): larger distance between $v$ and $v'$ means higher penalty for changing $v$ to $v'$

• The problem is NP-complete even for a fixed set of FDs; it remains intractable for CFDs
  • We will go for heuristics
Warm-up: repairing under FDs

Suppose $\Sigma$ contains FDs only

- Given a pair of tuples violating a FD, you can fix the violation by setting their RHS attributes to be equal
- Doing so may cause more violations; just rinse and repeat

- Idea: use equivalence classes to track what cells must be set to the same value
  - Defer the decision of which specific value each equivalent class should take
“Chase” in action

[ CC, AC ] → [ City ]

<table>
<thead>
<tr>
<th>Name</th>
<th>CC</th>
<th>AC</th>
<th>City</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Ben</td>
<td>1</td>
<td>215</td>
<td>PHI 19132</td>
</tr>
<tr>
<td>t2</td>
<td>Joe</td>
<td>1</td>
<td>215</td>
<td>PHI 19132</td>
</tr>
<tr>
<td>t3</td>
<td>Paul</td>
<td>1</td>
<td>215</td>
<td>PHI 60132</td>
</tr>
<tr>
<td>t4</td>
<td>John</td>
<td>1</td>
<td>312</td>
<td>CHI 60132</td>
</tr>
</tbody>
</table>

Merge equivalence classes as needed

E3 = E1 ∪ E2

[ CC, AC ] → [ City ]

[ ZIP ] → [ City ]

<table>
<thead>
<tr>
<th>Name</th>
<th>CC</th>
<th>AC</th>
<th>City</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Ben</td>
<td>1</td>
<td>215</td>
<td>PHI 19132</td>
</tr>
<tr>
<td>t2</td>
<td>Joe</td>
<td>1</td>
<td>215</td>
<td>PHI 19132</td>
</tr>
<tr>
<td>t3</td>
<td>Paul</td>
<td>1</td>
<td>215</td>
<td>PHI 60132</td>
</tr>
<tr>
<td>t4</td>
<td>John</td>
<td>1</td>
<td>312</td>
<td>CHI 60132</td>
</tr>
</tbody>
</table>
How CFDs ruin the game

First CFD says E3 should take value PHI

But second CFD says E3 should take value CHI
CFD-resolve

Say $t.A$ violates a CFD

- Consider changing not only RHS attributes, but also LHS attributes
  - Don’t “invent” values: choose from active domain
  - If there is no suitable value, put “null”

- To resolve violations
  - Merge equivalence classes (as with FDs)
  - “Upgrade” equivalence classes
    - Assign each class “_” (unset) $\rightarrow$ some specific value $\rightarrow$ null
More details

Say $t. A$ violates a CFD $X \rightarrow A$, $t_p$

- **Case 1:** $t$ doesn’t match $t_p[A] = a$
  - If $\text{eq}(t. A)$, $t. A$’s equivalence class, is currently unset, just upgrade it to $a$
  - Otherwise, we try to change the LHS, i.e., $\text{eq}(t. B)$ for some $B \in X$

- **Case 2:** $t$ violates the CFD with another tuple $t'$
  - If at least one of $\text{eq}(t. A)$ and $\text{eq}(t. A')$ is “_”, merge the two equivalence classes
  - If $\text{eq}(t. A)$ and $\text{eq}(t. A')$ are assigned two different constants, try changing LHS attributes
### Example

Target value of equivalence class $E$

$$\text{targ}(E) = \text{not fixed} \Rightarrow \text{fixed} : \text{upgrade}$$

<table>
<thead>
<tr>
<th>Name</th>
<th>CC</th>
<th>AC</th>
<th>City</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Ben</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t2</td>
<td>Joe</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t3</td>
<td>Paul</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t4</td>
<td>John</td>
<td>1</td>
<td>312</td>
<td>CHI</td>
</tr>
</tbody>
</table>

#### E1: PHI Fixed

- $E1: \text{PHI}$

#### E2: Not Fixed

- $E2: \text{Not Fixed}$

**CC, AC → City**

- 1
- 215
- PHI

**ZIP → City**

- 60132
- CHI
### Example cont’d

#### Table 1: CC, AC → City

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t2</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t3</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t4</td>
<td>1</td>
<td>312</td>
<td>CHI</td>
</tr>
</tbody>
</table>

#### Table 2: ZIP → City

<table>
<thead>
<tr>
<th></th>
<th>ZIP</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60132</td>
<td>CHI</td>
</tr>
</tbody>
</table>

---

When picking a new value, choose “closest” from “related” tuples (t1, t2 in this case, which share the same City)

- E1: PHI Fixed
- E2: Not Fixed
Overall algorithm

• Start with each cell being in its own equivalent class; identify violating tuples for each CFD

• While there remains violations, pick the one whose repair costs the least
  • Repair leads to merging/upgrading equivalence classes, and possibly new violations
  • The process terminates because each step reduces the # of equivalence classes or increases the number of upgraded classes
From batch to incremental repairs

• Input: a clean input: set $\Sigma$ of constraints + clean database instance $D$ + new changes $\Delta D$

• Output: a repair of $D + \Delta D$

• Idea: repair one new tuple at a time, starting with individual attributes and considering more if needed
Example of repairing one tuple

<table>
<thead>
<tr>
<th>Name</th>
<th>CC</th>
<th>AC</th>
<th>City</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Mark</td>
<td>1</td>
<td>215</td>
<td>PHI</td>
</tr>
<tr>
<td>t2</td>
<td>Peter</td>
<td>44</td>
<td>131</td>
<td>EDI</td>
</tr>
<tr>
<td>t3</td>
<td>Eric</td>
<td>1</td>
<td>215</td>
<td>CHI</td>
</tr>
</tbody>
</table>

Single-attribute repair isn’t enough
Tuple ordering strategies

• Linear (no particular order): bad

• Weight-based: good
  • Repair tuples with higher accuracy first

• Violation-based: good
  • Repair tuples with fewer violations first
Consistent, but accurate?

- It’s better to have domain experts inspect repairs, but inspecting everything is too expensive

Sampling comes to rescue
- Inspect a small sample
- Edit both sample data and CFDs if necessary
- Make repairs automatically
- Rinse and repeat

Use stratified sampling
- Tuples involved in more violations will be sampled with higher probability
Experiments

• Sales data scraped from Web
• 7 CFDs with 300-5,000 pattern tuples each
• Initial dataset is clean
• Dirty data (with error rate 1-10%) generated by adding random noise to cells
  • Either a new value close to the original, or a value taken from some other random tuple
Accuracy of using CFDs vs. FDs

![Graph showing accuracy comparison between CFDs and FDs](image-url)
Quality vs. amount of error

![Graphs showing precision and recall vs. percentage of errors](image-url)

- Precision graph:
  - BatchRepair
  - V-IncRepair
  - W-IncRepair
  - L-IncRepair

- Recall graph:
  - BatchRepair
  - V-IncRepair
  - W-IncRepair
  - L-IncRepair

These graphs illustrate the performance of different repair algorithms based on the percentage of errors. Precision and recall metrics are plotted against the percentage of errors to evaluate the effectiveness of each method in maintaining data quality.
Summary and thoughts

• Seminal work on automatic repair involving constraints beyond traditional FDs

• Good ideas that were expanded later
  • Iterative cleaning with humans in the loop
  • Change constraints, not just data
  • Sampling
Other notions of “best” repair

Short of a cost function $\text{cost}(D, D')$, you can go for

- **Cardinality-minimal repair**: minimize $|\Delta(D, D')|$, where $\Delta(D, D') = \text{cells whose values differ in } D, D'$

- **Cardinality-set-minimal repair**: $D'$ is set minimal iff there exists no $D''$ such that $\Delta(D, D'') \subset \Delta(D, D')$
  - i.e., can’t find a repair that changes a subset of the cells

- **Set-minimal repair**: $D''$ is set minimal iff there exists no $D''$ such that $\Delta(D, D'') \subset \Delta(D, D')$ and $D''$ sets every cell in $\Delta(D, D'')$ to the same value as $D'$
  - i.e., can’t find a repair that leaves some cells intact instead
What about both data and rules?

• Given a violation, should you fix data or constraint?

• Instead of using arbitrary cost function, apply the **MDL** (minimum description length) principle
  • Minimize the total complexity of encoding the model (rules) + instance given the model
  • Chang & Miller, *ICDE* 2011
Beyond one type of constraints

• If we have constraints of multiple types, it’s better to consider them holistically instead of one type at a time.

• One problem formulation is conflict hypergraph.
  • Nodes: cells
  • Hyper-edges: set of cells involved in a violation
  • Find a minimum vertex cover
  • Rinse and repeat, since new violations may arise

Chu, Ilyas, Papotti. ICDE 2013
Project warm-up on Thursday

Come prepared!

• Everybody speaks
  • Teams already formed or being developed: pitch your idea, and recruit more members
  • Individuals looking for teams: state your background and interest

• Submit by Friday night on Piazza a list of members and briefly proposal