A Revival of Integrity Constraints for Data Cleaning

Wenfei Fan\textsuperscript{1,2} Floris Geerts\textsuperscript{1} Xibei Jia\textsuperscript{1}

\textsuperscript{1}School of Informatics, University of Edinburgh

\textsuperscript{2}Bell Laboratories, Alcatel-Lucent
Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
- Constraint-based methods for data cleaning
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues

Acknowledgments. Our thanks to our colleagues for their input:

Michael Benedikt (Oxford Univ), Philip Bohannon (Yahoo! Research), Jan Chomicki (SUNY at Buffalo), Anastasios Kementsietsidis (IBM Watson), Shuai Ma (Edinburgh), Ming Xiong (Bell Labs), David Richardson and Colin Adams (Commercialization, Edinburgh), ...
Real life encounter: It may happen to you

Mr. Smith, our database records indicate that you owe us an outstanding amount of £1,921.76 for council tax in 2007

<table>
<thead>
<tr>
<th>NI#</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC1234566</td>
<td>131</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>SC1234566</td>
<td>020</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Portland</td>
<td>LDN</td>
<td>W1B 1JL</td>
</tr>
</tbody>
</table>

A data quality problem: ▶ M. Smith moved from Edinburgh to London in 2006, and no longer lived in Edinburgh in 2007; ▶ The council database was not correctly updated: it retains both Smith's old address and his new address.
Real life encounter: It may happen to you

Mr. Smith, our database records indicate that you owe us an outstanding amount of £1,921.76 for council tax in 2007

<table>
<thead>
<tr>
<th>NI#</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC1234566</td>
<td>131</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>SC1234566</td>
<td>020</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Portland</td>
<td>LDN</td>
<td>W1B 1JL</td>
</tr>
</tbody>
</table>

A data quality problem:

- M. Smith moved from Edinburgh to London in 2006, and no longer lived in Edinburgh in 2007;
- The council database was not correctly updated: it retains both Smith’s old address and his new address.
Real-world data is often **dirty**

Dirty data: inconsistent, inaccurate, incomplete, stale, or deliberately falsified

- US: Pentagon asked 200+ dead officers to re-enlist
- UK: there are 81 million national insurance numbers but only 60 million people eligible
- Australia: 500,000 dead people retain active medicare cards
- In a database of 500,000 customers, 120,000 records become invalid within a year – death, divorce, marriage, move
- Typical data error rate in industry: 1% – 5%
- …
Real-world data is often dirty

Dirty data: inconsistent, inaccurate, incomplete, stale, or deliberately falsified

- US: Pentagon asked 200+ dead officers to re-enlist
- UK: there are 81 million national insurance numbers but only 60 million people eligible
- Australia: 500,000 dead people retain active medicare cards
- In a database of 500,000 customers, 120,000 records become invalid within a year – death, divorce, marriage, move
- Typical data error rate in industry: 1% – 5%
- ...

Errors and inconsistencies may be introduced during data gathering, storage, transmission, transformation, integration, ...

The prevalent use of Internet has been increasing the risks, in an unprecedent scale, of creating and propagating dirty data.
Dirty data is costly

Telecommunication services: dirty data routinely leads to failure to bill for services, delay in repairing network problems, unnecessary leasing of equipment ⇒ loss of revenue, credibility, customers
Dirty data is costly

Telecommunication services: dirty data routinely leads to failure to bill for services, delay in repairing network problems, unnecessary leasing of equipment ⇒ loss of revenue, credibility, customers

▷ Poor data costs US companies $600 billions annually;
▷ Erroneously priced data in retail databases costs US customers $2.5 billion each year;
▷ World-wide losses from payment card fraud reached $4.84 billion in 2006;
▷ 30% – 80% of the development time for data cleaning in a data integration project; and
▷ don’t forget “dirty data” about WMD in Iraq

The market for data quality tools is growing at 17% annually ⇒ the 7% average of other IT segments
Data quality: Criteria

- **Consistency**: whether the data contains errors or conflicts that emerge as violations of certain semantic rules.
  Example: \( \text{age} = 82 \) and \( \text{age} = 28 \) for the same patient

- **Accuracy**: how close a value representing a real-life entity is to the true value of the entity.
  Example: \( \text{age} \leq 200 \) vs. \( \text{age} = 45 \)

- **Completeness**: whether a given query can be answered given the information available.
  Example: \( \text{age} = \text{null} \) (missing value) in a patient record, or missing patient record (missing tuple)

- **Timeliness**: whether the data is too stale to answer a given query.
  Example: Council tax collection in 2007 based on an old address of 2005

...
Research activities

Statistics, management, and computer science

- **Error correction** (data imputation): to localize tuples that violate a given set of semantic rules, and fix erroneous values in the tuples that are identified as violations of the rules.

- **Object identification**: to identify tuples from one or more relations that refer to the same real-world object.

- **Profiling**: to infer and discover meta-data (constraints or semantic rules) from sample data.

- **Data integration**: to resolve conflicts in the sources via object identification; quality-driven query processing by explicitly taking into account the quality of data from various sources.

Approaches: probabilistic, empirical, and logic-based, ...
Detecting semantic errors: Integrity constraints

- **Syntactic errors**: when a value is not in the corresponding domain or range; e.g., name = 1.23, age = 250.
- **Semantic errors**: when a value representing a real-world entity is different from the true value of the entity; e.g., CIA found WMD in Iraq.

Semantic errors are hard to detect and fix.
Detecting semantic errors: Integrity constraints

- **Syntactic errors**: when a value is not in the corresponding domain or range; e.g., name = 1.23, age = 250.
- **Semantic errors**: when a value representing a real-world entity is different from the true value of the entity; e.g., CIA found WMD in Iraq.

Semantic errors are hard to detect and fix.

Integrity constraints: for specifying the semantics of data.

<table>
<thead>
<tr>
<th>NI#</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC1234566</td>
<td>131</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>SC1234566</td>
<td>020</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Portland</td>
<td>LDN</td>
<td>W1B 1JL</td>
</tr>
</tbody>
</table>

Functional dependency:  \( \text{NI#} \rightarrow \text{name, AC, phn, street, city, zip} \)

- **NI#** is a key: there is a unique record for each distinct NI#.
- For SC1234566, at least one of the records must be dirty.
Integrity constraints: Flashback

Integrity constraints (data dependencies): first-order logic sentences

$$\forall x_1 \ldots x_m (\phi(x_1, \ldots, x_m) \rightarrow \exists y_1 \ldots y_n \psi(z_1, \ldots, z_k))$$

defined in terms of conjunctions of relation atoms and variables.

- designed for improving the quality of schema
- almost as old as relational databases (Codd 1972)

Familiar constraints:

- Functional dependencies: $R(X \rightarrow Y)$
- Inclusion dependencies: $R_1[X] \subseteq R_2[Y]$
- Equality Generating Dependencies (EGDs; e.g., FDs)
- Tuple Generating dependencies (TGDs; e.g., INDs)
- Full Dependencies (e.g., FDs)
- ...
Constraint-based data cleaning: A principled approach

Constraints as data quality rules: detect errors and inconsistencies that emerge as violations of the constraints

- Specifying a fundamental part of the semantics of data.
- Reasoning techniques: inference systems, algorithms, ..., to remove redundant rules and check the consistency of the rules. Recall Armstrong’s Axioms, and algorithms for computing closures and minimal covers of FDs
- Constraint profiling: discovery of data-quality rules; e.g., TANE, FastFD, ...

Many data quality tools still heavily rely on manual effort, ad-hoc rules and low-level programs – difficult to write and maintain.

Constraints should logically become part of data quality tools.
References

Survey on traditional data dependencies:

Surveys on data quality

Sources of the statistics
The need for revising traditional constraints

One of the central technical problems is how to tell whether the data is dirty or clean

▶ Schema: country code (CC), area code (AC), phone (phn), ...

\[
\text{Cust}(\text{CC}: \text{int}, \text{AC}: \text{int}, \text{phn}: \text{int}, \text{name}: \text{string}, \text{street}: \text{string}, \text{city}: \text{string}, \text{zip}: \text{string})
\]

▶ Instance:

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t2</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t3</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>
The need for revising traditional constraints

Instance:

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t2</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t3</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

functional dependencies (FDs):

\[fd_1: [CC, AC, phn] \rightarrow [street, city, zip],\]
\[fd_2: [CC, AC] \rightarrow [city].\]

The database satisfies the FDs. But the data is NOT clean!

Traditional constraints were designed for improving the quality of schema

We need constraints for improving the quality of data
Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
  - Conditional functional dependencies (CFDs): for capturing inconsistencies in a single relation;
  - Conditional inclusion dependencies: for schema matching and for capturing inconsistencies across different relations;
  - Matching dependencies: for object identification
- Constraint-based methods for data cleaning
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues
Outline

- Data quality: An overview

- Revisions of constraints for improving the quality of data
  - Conditional functional dependencies (CFDs): for capturing inconsistencies in a single relation;
    - Syntax and semantics
    - Reasoning about CFDs: satisfiability, implication, inference system, propagation
    - An extension: adding disjunction and negation
  - Conditional inclusion dependencies: for schema matching and for capturing inconsistencies across different relations;
  - Matching dependencies: for object identification

- Constraint-based methods for data cleaning

- Demo: SemanDaQ, a constraint-based data cleaning tool

- Open research issues
Capturing inconsistencies in the data

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$:</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_2$:</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_3$:</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- In the **UK**, the zip code **uniquely** determines the street.

  \[\text{cfd}_1: ([\text{CC} = 44, \text{zip}] \rightarrow [\text{street}])\]

  - This constraint specifies a **semantic** property of the data.
  - It is **conditional**: it may not hold for other countries, e.g., USA
  - It can’t be expressed as standard FDs: constants + variables
Capturing inconsistencies in the data

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_2$</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_3$</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- In the **UK**, the zip code **uniquely** determines the street.

  \[\text{cfd}_1: ([\text{CC} = 44, \text{zip}] \rightarrow [\text{street}])\]

  - This constraint specifies a **semantic** property of the data.
  - It is **conditional**: it may **not** hold for other countries, e.g., USA
  - It **can’t** be expressed as standard FDs: constants + variables
  - The example database does not satisfy this constraint

The data is **not** clean after all, although it satisfies the FDs.
Patterns of semantically related data values

- In the **UK**, if the **area code** is **131**, then the **city** must be Edinburgh (**EDI**)
- In the **USA**, if the **area code** is **908**, then the **city** must be Murray Hill (**MH**)
- Refining the FD \( \text{fd}_1: [CC, AC, phn] \rightarrow [\text{street}, \text{city}, \text{zip}] \) by adding conditions (patterns of semantically related constants)
  
\[
\begin{align*}
\text{cfd}_2: \quad & ([CC = 44, AC = 131, phn] \rightarrow [\text{street}, \text{city} = \text{‘EDI’}, \text{zip}]) \\
\text{cfd}_3: \quad & ([CC = 01, AC = 908, phn] \rightarrow [\text{street}, \text{city} = \text{‘MH’}, \text{zip}])
\end{align*}
\]
Patterns of semantically related data values

- In the **UK**, if the **area code** is 131, then the **city** must be Edinburgh (**EDI**)
- In the **USA**, if the **area code** is 908, then the **city** must be Murray Hill (**MH**)
- Refining the FD $fd_1$: $[CC, AC, phn] \rightarrow [street, city, zip]$ by adding conditions (patterns of semantically related constants)

  $cfd_2$: $([CC = 44, AC = 131, phn] \rightarrow [street, city = 'EDI', zip])$
  $cfd_3$: $([CC = 01, AC = 908, phn] \rightarrow [street, city = 'MH', zip])$

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$:</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_2$:</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_3$:</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

None of the tuples in the example database is clean
The need for the revision: to improve the quality of data

\[
\text{cfd}_1: ([\text{CC} = 44, \text{zip}] \rightarrow [\text{street}])
\]
\[
\text{cfd}_2: ([\text{CC} = 44, \text{AC} = 131, \text{phn}] \rightarrow [\text{street}, \text{city} = '\text{EDI}', \text{zip}])
\]
\[
\text{cfd}_3: ([\text{CC} = 01, \text{AC} = 908, \text{phn}] \rightarrow [\text{street}, \text{city} = '\text{MH}', \text{zip}])
\]

- They capture inconsistencies that traditional FDs cannot detect – FDs were developed for schema design after all
- Data integration in real-life: source constraints
  - hold on a subset of sources
  - but only hold conditionally on the integrated data
- They are NOT expressible as traditional FDs
  - do not hold on the entire relation
  - contain constant data values, besides logical variables

Data quality rules: to determine whether the data is dirty or clean
Conditional Functional Dependencies (CFDs)

An **extension** of traditional functional dependencies:

- A CFD is defined to be a pair $\varphi = R(X \rightarrow Y, T_p)$, where
  - $X \rightarrow Y$ is a standard FD, embedded in $\varphi$;
  - $T_p$ is the **pattern tableau** consisting of tuples $t_p$ over $X \cup Y$;
  - In a pattern tuple $t_p$, each $t_p[A]$ is either a **constant** from $\text{dom}(A)$ or a **wildcard** ‘\_’ (unnamed variable) that draws values from $\text{dom}(A)$.

- A single CFD representing cfd$_2$, cfd$_3$ and FD fd$_1$:
  - $\text{Cust}([\text{CC, AC, phn}] \rightarrow [\text{street, city, zip}], T_p)$
  - pattern tableau $T_p$:

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44</td>
<td>131</td>
<td>_</td>
<td>_</td>
<td>EDI</td>
<td>_</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>908</td>
<td>_</td>
<td>_</td>
<td>MH</td>
<td>_</td>
</tr>
<tr>
<td>1</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

Each pattern tuple $t_p$ is a constraint
Special cases of CFDs

- Traditional FDs as a special case: expressing the FD $fd_1$ as
  - $\text{Cust}([\text{CC, AC, phn}] \rightarrow [\text{street, city, zip}], T_p)$
  - pattern tableau $T_p$:

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Special cases of CFDs

- Traditional FDs as a special case: expressing the FD $fd_1$ as
  - $Cust([CC, AC, phn] \rightarrow [\text{street, city, zip}], T_p)$
  - pattern tableau $T_p$:
    
    | CC | AC | phn | street | city | zip |
    |----|----|-----|--------|------|-----|
    |    |    |     |        |      |     |

- Constant CFDs:
  - $Cust([CC, AC] \rightarrow [\text{city}], T_p)$
  - pattern tableau $T_p$:
    
    | CC | AC | city |
    |----|----|------|
    | 44 | 131| EDI  |
    | 01 | 908| MH   |
The semantics of CFDs

- Operator \( \equiv \) for matching patterns:
  - a matches b (\( a \equiv b \))
    - either a or b is \(_\)
    - both a and b are constants and \( a = b \).
  - tuple \( t_1 \) matches tuple \( t_2 \) (\( t_1 \equiv t_2 \)): defined component-wise
    - \( (a, b) \equiv (a, \_ \) but \( (a, b) \not\equiv (a, c) \).

- Pattern tuples:
  - \( t_{p[X]} \): identifying a subset \( \{ u | u \in D, u[X] \equiv t_{p[X]} \} \);
  - \( u[Y] = v[Y] \equiv t_{p[Y]} \): enforcing the FD \( X \rightarrow Y \) and the pattern \( t_{p[Y]} \) to the subset.

- Conditional: \( t_{p} \) applies to the subset rather than to the entire D.
The semantics of CFDs

- Operator $\simeq$ for matching patterns:
  - $a$ matches $b$ ($a \simeq b$)
    - either $a$ or $b$ is $\_$
    - both $a$ and $b$ are constants and $a = b$.
  - tuple $t_1$ matches tuple $t_2$ ($t_1 \simeq t_2$): defined component-wise
    - $(a, b) \simeq (a, \_)$ but $(a, b) \not\simeq (a, c)$.

- A database $D$ satisfies a CFD $\varphi = R(X \rightarrow Y, T_p)$ iff for each pair of tuples $u, v \in D$ and for each pattern tuple $t_p \in T_p$,
  if $u[X] = v[X] \simeq t_p[X]$, then $u[Y] = v[Y] \simeq t_p[Y]$
The semantics of CFDs

- Operator $\simeq$ for matching patterns:
  - $a$ matches $b$ ($a \simeq b$)
    - either $a$ or $b$ is \_\_
    - both $a$ and $b$ are constants and $a = b$.
  - tuple $t_1$ matches tuple $t_2$ ($t_1 \simeq t_2$): defined component-wise
    - $(a, b) \simeq (a, \_\_)$ but $(a, b) \not\simeq (a, c)$.

- A database $D$ satisfies a CFD $\varphi = R(X \rightarrow Y, T_p)$ iff for each pair of tuples $u, v \in D$ and for each pattern tuple $t_p \in T_p$,
  
  if $u[X] = v[X] \simeq t_p[X]$, then $u[Y] = v[Y] \simeq t_p[Y]$

- Pattern tuples:
  - $t_p[X]$: identifying a subset $\{u \mid u \in D, u[X] \simeq t_p[X]\}$;

Conditional: $t_p$ applies to the subset rather than to the entire $D$
Violation of CFDs

- Cust([CC, AC, phn] → [street, city, zip], \( T_p \))

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_p ):</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>131</td>
<td>_</td>
<td>_</td>
<td>EDI</td>
<td>_</td>
</tr>
<tr>
<td>( t_p )</td>
<td>01</td>
<td>908</td>
<td>_</td>
<td>_</td>
<td>MH</td>
<td>_</td>
</tr>
</tbody>
</table>

- Tuple \( t_3 \) violates the CFD:
  - \( t_3[CC, AC, phn] = t_3[CC, AC, phn] \approx t_p[CC, AC, phn] \)
  - \( t_3[city] \not\approx t_p[city] \)
Violation of CFDs

- **Cust([CC, AC, phn] → [street, city, zip], \(T_p\))**

<table>
<thead>
<tr>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>_</td>
<td>_</td>
<td>EDI</td>
<td>_</td>
</tr>
<tr>
<td>01</td>
<td>908</td>
<td>_</td>
<td>_</td>
<td>MH</td>
<td>_</td>
</tr>
</tbody>
</table>

- **Tuple \(t_3\) violates the CFD:**
  - \(t_3[CC, AC, phn] = t_3[CC, AC, phn] \approx t_p[CC, AC, phn]\)
  - \(t_3[city] \not\approx t_p[city]\)

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1):</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>(t_2):</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>(t_3):</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

While it takes two tuples to violate an FD, a single tuple may violate a CFD
Static analyses: reasoning about constraints

Central technical problems associated with any constraint language:

▶ The satisfiability problem: whether a given set of constraints can be satisfied by some database at all?

▶ The implication problem: whether a constraint is a logical consequence of a given set of constraints?

▶ A stronger property – the finite axiomatizability: whether there exists a finite set of inference rules that are sound and complete for implication analysis (e.g., Armstrong’s Axioms)

A balance between the expressive power and complexity.

An issue important for data exchange and data integration:

▶ Propagation: whether constraints on sources hold on a view in a certain form?

Renewed interest in these decision problems, for data cleaning.
Satisfiability: “Dirty” constraints?

One can specify any traditional FDs without worrying about whether or not there are conflicts among them.
Satisfiability: “Dirty” constraints?

One can specify any traditional FDs without worrying about whether or not there are conflicts among them.

In contrast, a set of CFDs may have conflicts or inconsistencies:

\[ \varphi = R(A \rightarrow B, T_p), \text{ where } T_p = \begin{array}{|c|c|} \hline A & B \\ \hline \_ & b_1 \\ \hline \_ & b_2 \\ \hline \end{array} \]

- For any nonempty database \( D \) and for any tuple \( t \) in \( D \), \( \varphi \) says that \( t[B] \) must be both \( b_1 \) and \( b_2 \).
Satisfiability: “Dirty” constraints?

One can specify any traditional FDs without worrying about whether or not there are conflicts among them.

In contrast, a set of CFDs may have conflicts or inconsistencies:

\[ \varphi = R(A \rightarrow B, T_p) \],

where \( T_p = \begin{array}{c|c}
A & B \\
\_ & b_1 \\
\_ & b_2 \\
\end{array} \)

- For any nonempty database \( D \) and for any tuple \( t \) in \( D \), \( \varphi \) says that \( t[B] \) must be both \( b_1 \) and \( b_2 \).

The satisfiability problem is to determine, given a schema \( R \) and a set \( \Sigma \) of constraints defined on \( R \), whether or not there exists a nonempty instance \( D \) of \( R \) that satisfies all constraints \( \varphi \) in \( \Sigma \).

To decide whether or not data quality rules are dirty themselves
In the same setting as the classical dependency theory

Recall domain specification in a schema:

\[
\text{Cust}(\text{CC}: \text{int}, \text{AC}: \text{int}, \text{phn}: \text{int}, \text{name}: \text{string}, \text{street}: \text{string}, ...)\]

It is typically assumed that in each domain,

- there are at least two elements,
- there is no upper bound: possibly infinitely many
In the same setting as the classical dependency theory

Recall domain specification in a schema:

\[ \text{Cust(CC: int, AC: int, phn: int, name: string, street: string, ...)} \]

It is typically assumed that in each domain,

- there are at least two elements,
- there is no upper bound: possibly infinitely many

**Good news:** in this setting, CFDs do not make our lives much harder

**Theorem**

*For CFDs, the satisfiability problem is in quadratic time.*
In the same setting as the classical dependency theory

Recall domain specification in a schema:

\[
\text{Cust}(\text{CC}: \text{int}, \text{AC}: \text{int}, \text{phn}: \text{int}, \text{name}: \text{string}, \text{street}: \text{string}, \ldots)
\]

It is typically assumed that in each domain,
- there are at least two elements,
- there is no upper bound: possibly infinitely many

**Good news:** in this setting, CFDs do not make our lives much harder

**Theorem**

*For CFDs, the satisfiability problem is in quadratic time.*

However, in practice we have to consider a more general setting
In practice, it is common to find attributes with a finite domain: Boolean, date, ...

While the presence of attributes with a finite domain does not complicate the analyses of FDs, it does take a toll on CFDs.

Consider $\Sigma = \{\psi_1, \psi_2\}$, where

$\psi_1 = R(A \rightarrow B, T_1)$, and $\psi_2 = R(B \rightarrow A, T_2)$

$T_1 = \begin{array}{|c|c|}
\hline
A & B \\
\hline
\text{true} & b_1 \\
\text{false} & b_2 \\
\hline
\end{array}$

$T_2 = \begin{array}{|c|c|}
\hline
B & A \\
\hline
b_1 & \text{false} \\
b_2 & \text{true} \\
\hline
\end{array}$

If dom(A) is Boolean, then $\Sigma$ is not satisfiable!
The interaction between CFDs and domain constraints

In practice, it is common to find attributes with a finite domain: Boolean, date, ...

While the presence of attributes with a finite domain does not complicate the analyses of FDs, it does take a toll on CFDs

Consider $\Sigma = \{\psi_1, \psi_2\}$, where

$\psi_1 = R(A \rightarrow B, T_1)$, and $\psi_2 = R(B \rightarrow A, T_2)$

$T_1 = \begin{array}{|c|c|}
A & B \\
\hline
true & b_1 \\
false & b_2
\end{array}$

$T_2 = \begin{array}{|c|c|}
B & A \\
\hline
b_1 & false \\
b_2 & true
\end{array}$

If $\text{dom}(A)$ is Boolean, then $\Sigma$ is not satisfiable!
The interaction between CFDs and domain constraints

In practice, it is common to find attributes with a finite domain: Boolean, date, ...

While the presence of attributes with a finite domain does not complicate the analyses of FDs, it does take a toll on CFDs

Consider $\Sigma = \{\psi_1, \psi_2\}$, where

$\psi_1 = R(A \rightarrow B, T_1)$, and $\psi_2 = R(B \rightarrow A, T_2)$

$T_1 = \begin{array}{|c|c|} \hline A & B \\ \hline \text{true} & b_1 \\ \text{false} & b_2 \\ \hline \end{array}$

$T_2 = \begin{array}{|c|c|} \hline B & A \\ \hline b_1 & \text{false} \\ b_2 & \text{true} \\ \hline \end{array}$

If $\text{dom}(A)$ is Boolean, then $\Sigma$ is not satisfiable!
The interaction between CFDs and domain constraints

In practice, it is common to find attributes with a finite domain: Boolean, date, ...

While the presence of attributes with a finite domain does not complicate the analyses of FDs, it does take a toll on CFDs

Consider $\Sigma = \{\psi_1, \psi_2\}$, where

$\psi_1 = R(A \rightarrow B, T_1)$, and $\psi_2 = R(B \rightarrow A, T_2)$

$$
\begin{array}{c|c}
A & B \\
\hline
\text{true} & b_1 \\
\text{false} & b_2 \\
\end{array}
\quad
\begin{array}{c|c}
B & A \\
\hline
b_1 & \text{false} \\
\text{false} & b_2 \\
\text{true} & b_2 \\
\end{array}
$$

If dom(A) is Boolean, then $\Sigma$ is not satisfiable!

**Theorem**

*When attributes with a finite domain may be present, the satisfiability problem for CFDs is NP-complete.*
Implication: eliminating redundancies

The implication problem is to determine, given a schema $\mathcal{R}$, a set $\Sigma$ of constraints and a single constraint $\phi$ defined on $\mathcal{R}$, whether or not $\Sigma$ implies $\phi$, denoted by $\Sigma \models \phi$, i.e., whether for any instance $D$ of $\mathcal{R}$ that satisfies $\Sigma$, $D$ also satisfies $\phi$.

To remove redundant data quality rules

Example: $\phi$ is redundant provided that $\Sigma = \{\phi_1, \phi_2\}$ is given:

- $\phi = R(A \rightarrow C, T_p)$, $\phi_1 = R(A \rightarrow B, T_1)$, $\phi_2 = R(B \rightarrow C, T_2)$;
- $T_p = \begin{array}{cc} A & C \\ a & c \end{array}$, $T_1 = \begin{array}{cc} A & B \\ - & b \end{array}$, $T_2 = \begin{array}{cc} B & C \\ - & c \end{array}$;
The implication problem is to determine, given a schema $\mathcal{R}$, a set $\Sigma$ of constraints and a single constraint $\phi$ defined on $\mathcal{R}$, whether or not $\Sigma$ implies $\phi$, denoted by $\Sigma \models \phi$, i.e., whether for any instance $D$ of $\mathcal{R}$ that satisfies $\Sigma$, $D$ also satisfies $\phi$.

To remove redundant data quality rules

Example: $\varphi$ is redundant provided that $\Sigma = \{\phi_1, \phi_2\}$ is given:

- $\varphi = R(A \rightarrow C, T_p)$, $\phi_1 = R(A \rightarrow B, T_1)$, $\phi_2 = R(B \rightarrow C, T_2)$;
- $T_p = \begin{array}{cc} A & C \\ a & c \end{array}$, $T_1 = \begin{array}{cc} A & B \\ - & b \end{array}$, $T_2 = \begin{array}{cc} B & C \\ - & c \end{array}$

For traditional FDs, the implication problem is in linear-time.

**Theorem**

The implication problem for CFDs is in quadratic time in the absence of finite-domain attributes, and is coNP-complete in the general setting.
Finite axiomatizability of CFDs

Armstrong’s axioms for FDs:

Reflexivity: If \( Y \subseteq X \), then \( X \rightarrow Y \)

Augmentation: If \( X \rightarrow Y \), then \( XZ \rightarrow YZ \)

Transitivity: If \( X \rightarrow Y \) and \( Y \rightarrow Z \), then \( X \rightarrow Z \)

Sound and complete: \( \Sigma \models \phi \) iff \( \phi \) can be inferred from \( \Sigma \) using the axioms.
Finite axiomatizability of CFDs

Armstrong’s axioms for FDs:

- **Reflexivity**: If \( Y \subseteq X \), then \( X \rightarrow Y \)
- **Augmentation**: If \( X \rightarrow Y \), then \( XZ \rightarrow YZ \)
- **Transitivity**: If \( X \rightarrow Y \) and \( Y \rightarrow Z \), then \( X \rightarrow Z \)

**Sound and complete**: \( \Sigma \models \phi \) iff \( \phi \) can be inferred from \( \Sigma \) using the axioms.

**Theorem**

*There is a sound and complete inference system for CFDs.*

More involved than Armstrong’s axioms: dealing with patterns

- If \((X \rightarrow Y, t_p)\) and \((Y \rightarrow Z, t'_p)\), and
- if \( t_p[Y] \preceq t'_p[Y] \) (\( a \preceq _-, a \preceq a, _- \preceq _- \)),
- then \((X \rightarrow Z, (t_p[X] \parallel t'_p[Z]))\)
Static Analyses: CFDs vs. FDs

- In the absence of attributes with a finite domain:

<table>
<thead>
<tr>
<th></th>
<th>satisfiability</th>
<th>implication</th>
<th>finite axiomatizability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CFD</strong></td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>yes</td>
</tr>
<tr>
<td><strong>FD</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>yes</td>
</tr>
</tbody>
</table>

- General setting:

<table>
<thead>
<tr>
<th></th>
<th>satisfiability</th>
<th>implication</th>
<th>finite axiomatizability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CFD</strong></td>
<td>NP-complete</td>
<td>coNP-complete</td>
<td>yes</td>
</tr>
<tr>
<td><strong>FD</strong></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>yes</td>
</tr>
</tbody>
</table>

The interaction between domain constraints and CFDs.
Constraint propagation: The need

In data exchange or data integration, constraints that hold on sources may only hold conditionally on the target data.

- **Sources:** two relations for customers in the UK and USA
  \[ R_S(AC: \text{int}, \text{phn: int, name: string, street: string, city: string, zip: string}) \]

- **A traditional FD on** \( R_{UK} \): \( \text{zip} \rightarrow \text{street} \)

- **View definition:** \( (R_{UK} \times (\text{CC: 44})) \cup (R_{USA} \times (\text{CC: 01})) \)

- **The FD no longer** holds on the target data
In data exchange or data integration, constraints that hold on sources may only hold \textit{conditionally} on the target data.

- Sources: two relations for customers in the UK and USA
  \[
  R_S(\text{AC: int, phn: int, name: string, street: string, city: string, zip: string})
  \]

- A traditional FD on \( R_{UK} \): \( \text{zip} \rightarrow \text{street} \)

- View definition: \( (R_{UK} \times (\text{CC: 44})) \cup (R_{USA} \times (\text{CC: 01})) \)

- The FD \textit{no longer} holds on the target data

- The FD is indeed propagated to the target, but as a CFD

\[
([\text{CC, zip}] \rightarrow [\text{street}], T_p)
\]

<table>
<thead>
<tr>
<th>CC</th>
<th>zip</th>
<th>street</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Constraint propagation

- **Input:**
  - A set $\Sigma$ of source constraints: FDs (or CFDs) on the sources
  - A view definition $\sigma$: a relational query
  - A view constraint $\varphi$

- **Question:** Is $\varphi$ propagated from $\Sigma$ via $\sigma$?

  For any source database $D$ that satisfies $\Sigma$, the view $\sigma(D)$ is guaranteed to satisfy $\varphi$. Why bother?

  - Data quality: no need to check zip $\rightarrow$ street on target data taken from UK
  - Query optimization: making use of the derived target constraints to answer queries on the view
  - Update management: an insertion of (CC = 44, AC = 20, city = EDI, ...) can be rejected without checking the data
Constraint propagation

- **Input:**
  - A set $\Sigma$ of source constraints: FDs (or CFDs) on the sources
  - A view definition $\sigma$: a relational query
  - A view constraint $\varphi$

- **Question:** Is $\varphi$ *propagated* from $\Sigma$ via $\sigma$?
  For any source database $D$ that satisfies $\Sigma$, the view $\sigma(D)$ is guaranteed to satisfy $\varphi$.

**Why bother?**

- **Data quality:** no need to check $\text{zip} \rightarrow \text{street}$ on target data taken from $R_{UK}$; particularly useful when the view is *virtual*
- **Query optimization:** making use of the derived target constraints to answer queries on the view
- **Update management:** an insertion of $(\text{CC} = 44, \text{AC} = 20, \text{city} = \text{EDI}, \ldots)$ can be rejected without checking the data
It is believed that the constraint propagation problem is

- **in PTIME** for views defined in terms of **SPCU** queries (selection, projection, Cartesian product, union),
- **undecidable** for views defined in relational algebra.
Propagating from Source FDs to View FDs

It is believed that the constraint propagation problem is

- in PTIME for views defined in terms of SPCU queries (selection, projection, Cartesian product, union),
- undecidable for views defined in relational algebra.

The PTIME result holds, but only in the absence of attributes with a finite domain:

**Theorem**

The propagation problem from source FDs to view FDs is already coNP-complete for SC views in the general setting.

There is interaction between domain constraints and constraint propagation analysis, for traditional FDs already!
In the absence of attributes with a finite domain: CFDs do not complicate the propagation analysis

In the general setting: CFDs make the analysis harder

Theorem

The propagation problem from source CFDs to view CFDs is

- in \( \text{PTIME} \) for SPCU views in the absence of attributes with a finite domain;
- in \( \text{coNP} \) for SPCU views, and is \( \text{coNP-hard} \) for S, C or P views in the general setting.

Theorem

In the general setting, the propagation problem from source FDs to view CFDs is in \( \text{PTIME} \) for PC views, and is \( \text{coNP-complete} \) for SC views.
Extension: adding negation and disjunction to CFDs

For New York area codes,

- if city is not in \( \{ \text{NYC, LI} \} \), then city uniquely determines AC – negation;
- if city = NYC, then AC must be one of 212, 718, 646, 347, 917 – disjunction
Extension: adding negation and disjunction to CFDs

For New York area codes,

- if city is not in \{NYC, LI\}, then city uniquely determines AC – negation;
- if city = NYC, then AC must be one of 212, 718, 646, 347, 917 – disjunction

- Extended CFDs (eCFDs) by adding negation and disjunction:

\[
\begin{align*}
\text{ecfd}_1: & \quad \text{city} \not\in \{\text{NYC, LI}\} \rightarrow \text{AC} \\
\text{ecfd}_2: & \quad \text{city} \in \{\text{NYC}\} \rightarrow \text{AC} \in \{212, 718, 646, 347, 917\}
\end{align*}
\]
Extension: adding negation and disjunction to CFDs

For New York area codes,

- if city is not in \{NYC, LI\}, then city uniquely determines AC – negation;
- if city = NYC, then AC must be one of 212, 718, 646, 347, 917 – disjunction

Extended CFDs (eCFDs) by adding negation and disjunction:

\begin{align*}
\text{ecfd}_1: & \quad \text{city} \not\in \{\text{NYC, LI}\} \rightarrow \text{AC} \\
\text{ecfd}_2: & \quad \text{city} \in \{\text{NYC}\} \rightarrow \text{AC} \in \{212, 718, 646, 347, 917\}
\end{align*}

While more expressive, eCFDs do not incur much extra complexity:

**Theorem**

For eCFDs, the satisfiability problem remains NP-complete and the implication problem remains coNP-complete.
Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
  - Conditional functional dependencies: for capturing inconsistencies in a single relation;
  - Conditional inclusion dependencies (CINDs):
    - contextual schema matching
    - capturing inconsistencies across different relations
    - reasoning about CINDs
    - reasoning about CFDs and CINDs taken together
  - Matching dependencies: for object identification
- Constraint-based methods for data cleaning
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues
The need for extending inclusion dependencies

- Example schema:

  Source:  order(title: string, type: string, price: real) 
  Target:  book(title: string, price: real, format: string) 
            CD(album: string, price: real, genre: string) 

- Inclusion dependencies (INDs) from the source to the target?
The need for extending inclusion dependencies

▶ Example schema:

**Source:** order(title: string, type: string, price: real)

**Target:** book(title: string, price: real, format: string)

CD(album: string, price: real, genre: string)

▶ Inclusion dependencies (INDs) from the source to the target?

\[
\text{order}(\text{title, price}) \subseteq \text{book}(\text{title, price}), \\
\text{order}(\text{title, price}) \subseteq \text{CD}(\text{album, price}).
\]

These traditional INDs do not make sense

<table>
<thead>
<tr>
<th>order</th>
<th>title</th>
<th>type</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_4):</td>
<td>Snow White</td>
<td>CD</td>
<td>7.99</td>
</tr>
<tr>
<td>(t_5):</td>
<td>Harry Potter</td>
<td>book</td>
<td>17.99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>book</th>
<th>title</th>
<th>price</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_6):</td>
<td>Harry Potter</td>
<td>17.99</td>
<td>hard-cover</td>
</tr>
<tr>
<td>(t_7):</td>
<td>Snow White</td>
<td>7.99</td>
<td>paper-cover</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CD</th>
<th>album</th>
<th>price</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_8):</td>
<td>J. Denver</td>
<td>7.94</td>
<td>country</td>
</tr>
<tr>
<td>(t_9):</td>
<td>Snow White</td>
<td>7.99</td>
<td>a-book</td>
</tr>
</tbody>
</table>
Extending inclusion dependencies for schema matching

- **Schema:**
  
  **Source:** `order(title: string, type: string, price: real)`
  
  **Target:** `book(title: string, price: real, format: string)`
  
  `CD(album: string, price: real, genre: string)`

- There are indeed inclusion dependencies, **under conditions:**
  
  $cind_1$: `(order(title, price; type = 'book') \subseteq book(title, price))$
  
  $cind_2$: `(order(title, price; type = 'CD') \subseteq CD(album, price))$

  - `order(title, price) \subseteq book(title, price)` holds only if type = book
  
  - `order(title, price) \subseteq CD(album, price)` holds only if type = CD

These constraints only apply to **subsets** of the `order` relation that satisfy certain patterns.
Capturing inconsistencies across different relations

CFDs detect inconsistencies in a single relation. As data-quality rules, CINDs capture inconsistencies across different relations.

- A constraint from CD to book: it holds only if the genre of a CD is audio book and if so, then the format of the matching book must be audio

\[ \text{cind}_3: (\text{CD}(\text{album}, \text{price}; \text{genre} = 'a-book') \subseteq \text{book}(\text{title}, \text{price}; \text{format} = 'audio')) \]

- The example database does not satisfy cind$_3$

<table>
<thead>
<tr>
<th>CD</th>
<th>album</th>
<th>price</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>t$_8$</td>
<td>J. Denver</td>
<td>7.94</td>
<td>country</td>
</tr>
<tr>
<td>t$_9$</td>
<td>Snow White</td>
<td>7.99</td>
<td>a-book</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>book</th>
<th>title</th>
<th>price</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>t$_6$</td>
<td>Harry Potter</td>
<td>17.99</td>
<td>hard-cover</td>
</tr>
<tr>
<td>t$_7$</td>
<td>Snow White</td>
<td>7.99</td>
<td>paper-cover</td>
</tr>
</tbody>
</table>
Conditional Inclusion Dependencies (CINDs)

An extension of inclusion dependencies:

▶ A CIND is a pair \((R_1[X] \subseteq R_2[Y], \ T_p[X_p \parallel Y_p])\), where
  
  ▶ \(R_1[X] \subseteq R_2[Y]\) is a standard IND from \(R_1\) to \(R_2\); 
  
  ▶ \(T_p\) is a pattern tableau over \(X_p\) of \(R_1\) and \(Y_p\) of \(R_2\) (distinct from \(X\) and \(Y\)), consisting of pattern tuples of constants only.

▶ Examples:

\[
(\text{order}(\text{title, price; type = 'book'}) \subseteq \text{book}(\text{title, price})) \\
(\text{CD(album, price; genre = 'a-book'}) \\
\subseteq \text{book}(\text{title, price; format = 'audio'}))
\]

▶ CINDs:

\[
(\text{order}(\text{title, price}) \subseteq \text{book}(\text{title, price}), \ T_1[\text{type}]) \\
(\text{CD(album, price}) \subseteq \text{book}(\text{title, price}), \ T_2[\text{genre} \parallel \text{format}])
\]

\[
T_1: \begin{array}{c|c}
\text{type} & \text{book} \\
\hline
\end{array} \quad T_2: \begin{array}{c|c|c}
\text{genre} & \text{format} \\
\text{a-book} & \text{audio} \\
\hline
\end{array}
\]
Special cases of CINDs

A CIND is a pair \((R_1[X] \subseteq R_2[Y], T_p[X_p \parallel Y_p])\), where

- \(R_1[X] \subseteq R_2[Y]\) is a standard IND from \(R_1\) to \(R_2\);
- \(T_p\) is a pattern tableau over \(X_p\) of \(R_1\) and \(Y_p\) of \(R_2\) (distinct from \(X\) and \(Y\)), consisting of pattern tuples of constants only.

Standard INDs are a special case of CINDs:

- IND: \(R_1[X] \subseteq R_2[Y]\)
- CIND: \((R_1[X] \subseteq R_2[Y], T_p[\emptyset])\)

CINDs subsume traditional INDs
Semantics of CINDs

- \( D = (D_1, D_2) \), where \( D_i \) is an instance of \( R_i \), \( i = 1, 2 \).
- \( D \) satisfies \((R_1[X] \subseteq R_2[Y], T_p[X_p \parallel Y_p])\) iff
  for any tuple \( s \) in \( D_1 \) and any pattern tuple \( t_p \) in \( T_p \),
  if \( s[X_p] = t_p[X_p] \) then there exists a tuple \( t \) in \( D_2 \) such that
    - \( s[X] = t[Y] \) and
    - \( t[Y_p] = t_p[Y_p] \).
Semantics of CINDs

- $D = (D_1, D_2)$, where $D_i$ is an instance of $R_i$, $i = 1, 2$.
- $D$ satisfies $(R_1[X] \subseteq R_2[Y], T_p[X_p \parallel Y_p])$ iff
  for any tuple $s$ in $D_1$ and any pattern tuple $t_p$ in $T_p$, if $s[X_p] = t_p[X_p]$ then there exists a tuple $t$ in $D_2$ such that
  - $s[X] = t[Y]$ and

- Pattern tuples:
  - $t_p[X_p]$ identifies a subset $\{s \mid s \in D_1, s[X_p] = t_p[X_p]\}$;
  - $s[X] = t[Y]$ and $t[Y_p] = t_p[Y_p]$: enforcing the standard IND $R_1[X] \subseteq R_2[Y]$ on the subset, and moreover, enforcing the $t_p[Y_p]$ pattern to the matching $R_2$ tuples.

Each pattern tuple $t_p$ is a constraint
Reasoning about conditional inclusion dependencies

For traditional INDs,

- any set of INDs is always satisfiable
- the implication problem is \textit{PSPACE-complete}.
- there is a sound and complete inference system
Reasoning about conditional inclusion dependencies

For traditional INDs,
- any set of INDs is always satisfiable
- the implication problem is \text{PSPACE-complete}.
- there is a sound and complete inference system

Good news: the complexity for CINDs does not hike up, to an extent

\textbf{Theorem}

\textit{In the general setting, any set of CINDs is satisfiable.}

Contrast this to its CFD counterpart (\text{NP-complete})
Reasoning about conditional inclusion dependencies

For traditional INDs,
- any set of INDs is always satisfiable
- the implication problem is \text{PSPACE-complete}.
- there is a sound and complete inference system

Good news: the complexity for CINDs does not hike up, to an extent

**Theorem**

\textit{In the general setting, any set of CINDs is satisfiable.}

Contrast this to its CFD counterpart (\text{NP-complete})

**Theorem**

\textit{The implication problem for CINDs is (1) \text{PSPACE-complete in the absence of finite-domain attributes, and (2) EXPTIME-complete in the general setting.}}
For traditional INDs,
- any set of INDs is always satisfiable
- the implication problem is \( \text{PSPACE-complete} \).
- there is a sound and complete inference system

Good news: the complexity for CINDs does not hike up, to an extent

**Theorem**

*In the general setting, any set of CINDs is satisfiable.*

Contrast this to its CFD counterpart (NP-complete)

**Theorem**

*The implication problem for CINDs is (1) \( \text{PSPACE-complete} \) in the absence of finite-domain attributes, and (2) \( \text{EXPTIME-complete} \) in the general setting.*

**Theorem**

*There is a *sound and complete* inference system for CINDs.*
CFDs and CINDs taken together

We need both CFDs and CINDs for

- data cleaning
- schema mapping

For traditional FDs and INDs taken together,

- the satisfiability problem is in $O(1)$ time, and
- the implication problem is undecidable.
CFDs and CINDs taken together

We need both CFDs and CINDs for
- data cleaning
- schema mapping

For traditional FDs and INDs taken together,
- the satisfiability problem is in $O(1)$ time, and
- the implication problem is undecidable.

In contrast,

**Theorem**

*For CFDs and CINDs taken together,*
- *the satisfiability problem becomes undecidable, and*
- *the implication problem remains undecidable.*

These call for effective heuristic algorithms


▶ L. Golab, H. Karloff, F. Korn, D Srivastava and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. VLDB 2008 (adding range to CFDs)


Other extensions of traditional dependencies:


Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
  - Conditional functional dependencies: for capturing inconsistencies in a single relation;
  - Conditional inclusion dependencies: for schema matching and capturing inconsistencies across different relations;
- Matching dependencies (MDs)
  - Object identification
  - Matching dependencies: dynamic constraints
  - Generic reasoning: implication of matching rules
- Constraint-based methods for data cleaning
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues
Object identification

Data deduplication, merge/purge, record linkage (matching): to identify tuples from one or more relations that refer to the same real-world object.
Object identification

Data deduplication, merge/purge, record linkage (matching): to identify tuples from one or more relations that refer to the same real-world object.

Example: credit-card fraud detection

- Schema: credit cards and billing transactions
  
  \[\text{card}(C\#, \text{type}, \text{SSN}, \text{FN}, \text{LN}, \text{addr}, \text{tel}, \text{email}), \]
  \[\text{billing}(C\#, \text{item}, \text{price}, \text{FN}, \text{SN}, \text{post}, \text{phn}, \text{email}).\]

- For any instance \((D_c, D_b)\) of \((\text{card}, \text{billing})\), \(t \in D_c, t' \in D_b,\)
  
  - if \(t[C\#] = t'[C\#],\)
  
  - then \(t[Y_c] \) and \(t'[Y_b]\) must match – refer to the same holder
  
  \(Y_c = [\text{FN}, \text{LN}, \text{addr}, \text{tel}, \text{email}], \quad Y_b = [\text{FN}, \text{SN}, \text{post}, \text{phn}, \text{email}].\)
Object identification

Data deduplication, merge/purge, record linkage (matching): to identify tuples from one or more relations that refer to the same real-world object.

Example: credit-card fraud detection

- **Schema:** credit cards and billing transactions
  
  card(C#, type, SSN, FN, LN, addr, tel, email),
  billing(C#, item, price, FN, SN, post, phn, email).

- **For any instance** \((D_c, D_b)\) of \((\text{card, billing})\), \(t \in D_c, t' \in D_b\),
  
  - if \(t[C#] = t'[C#]\),
  
  - then \(t[Y_c]\) and \(t'[Y_b]\) must match – refer to the same holder

  \(Y_c = [\text{FN, LN, addr, tel, email}]\), \(Y_b = [\text{FN, SN, post, phn, email}]\).

essential to data integration, data cleaning, ...
Matching rules

Challenges: unreliable data sources, different representations ...

Matching rules: what attributes to compare and how to compare the attributes: for card tuple \( t \) and billing tuple \( t' \),

- if \( t[\text{LN, addr}] \) and \( t'[\text{SN, post}] \) are identical, and
- if \( t[\text{FN}] \) and \( t'[\text{FN}] \) are similar w.r.t. a similarity relation \( \approx_d \),
- then \( t[Y_c] \) and \( t'[Y_b] \) match
Matching rules

Challenges: **unreliable** data sources, **different** representations ...

Matching rules: **what attributes** to compare and **how to compare** the attributes: for card tuple \( t \) and billing tuple \( t' \),

- if \( t[LN, \text{addr}] \) and \( t'[SN, \text{post}] \) are identical, and
- if \( t[FN] \) and \( t'[FN] \) are **similar** w.r.t. a similarity relation \( \approx_d \),

then \( t[Y_c] \) and \( t'[Y_b] \) match

We can identify \( t[Y_c] \) and \( t'[Y_b] \) even if they **radically differ** in some attributes

- comparing \( t[LN, \text{addr}, FN] \) and \( t'[SN, \text{post}, FN] \) instead of \( t[FN, LN, \text{addr}, \text{tel}, \text{email}] \) and \( t'[FN, SN, \text{post}, \text{phn}, \text{email}] \).

- **similarity** \( \approx_d \) in stead of equality on FN
Expressing matching rules as dependencies

if $t[\text{LN, addr}] = t'[\text{SN, post}]$ and $t[\text{FN}] \approx_{d} t'[\text{FN}]$, then $t[Y_c] \iff t'[Y_b]$

$\phi_1: \text{card}[\text{LN}] = \text{billing}[\text{SN}] \land \text{card}[\text{addr}] = \text{billing}[\text{post}] \land \text{card}[\text{FN}] \approx_{d} \text{billing}[\text{FN}] \rightarrow \text{card}[Y_c] \iff \text{billing}[Y_b]$

- Similarity metrics: edit distance, $q$-grams, Jaro distance, ...
- Matching operation $t[Z] \iff t'[Z']$: identifying $t[Z]$ and $t'[Z']$ via updates; $t[Z]$ and $t'[Z']$ may be radically different and cannot be matched using any metric $\approx$ known in advance;
Expressing matching rules as dependencies

if \( t[\text{LN, addr}] = t'[\text{SN, post}] \) and \( t[\text{FN}] \approx_d t'[\text{FN}] \), then \( t[Y_C] \Leftrightarrow t'[Y_B] \)

\( \phi_1: \text{card}[\text{LN}] = \text{billing}[\text{SN}] \land \text{card}[\text{addr}] = \text{billing}[\text{post}] \land \text{card}[\text{FN}] \approx_d \text{billing}[\text{FN}] \rightarrow \text{card}[Y_C] \Leftrightarrow \text{billing}[Y_B] \)

- Similarity metrics: edit distance, \( q \)-grams, Jaro distance, ...
- Matching operation \( t[Z] \Leftrightarrow t'[Z'] \): identifying \( t[Z] \) and \( t'[Z'] \) via updates; \( t[Z] \) and \( t'[Z'] \) may be radically different and cannot be matched using any metric \( \approx \) known in advance;
- Matching dependency \( \phi \): for any instances \( D = (D_c, D_b) \) and \( D' = (D'_c, D'_b) \) of (card, billing), \( D \) and \( D' \) satisfy \( \phi \) if
  - \( D \subseteq D' \): \( D' \) is an updated \( D \) via value updates (every tuple \( t \) in \( D \) is also in \( D' \) although its value of \( t \) might be changed)
  - for any \( t \in D_c, t' \in D_b \), if (1) \( t[\text{LN, addr}] = t'[\text{SN, post}] \) and (2) \( t[\text{FN}] \approx_d t'[\text{FN}] \) in \( D \), then \( t[Y_C] = t'[Y_B] \) and (1-2) in \( D' \)

Dynamic semantics: if condition \( C \) holds then identify \( x \) and \( y \)
Matching dependencies (MDs)

- An MD $\phi$ defined on schemas $(R_1, R_2)$:

$$\bigwedge_{j \in [1, k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Z_1] \iff R_2[Z_2]$$

- $\approx_j$'s are similarity relations;
- $X_1, X_2$ (resp. $Z_1, Z_2$): attribute lists of $R_1, R_2$

$\phi_1$: $\text{card}[\text{LN}] = \text{billing}[\text{SN}] \land \text{card}[\text{addr}] = \text{billing}[\text{post}] \land$

$$\text{card}[\text{FN}] \approx_d \text{billing}[\text{FN}] \rightarrow \text{card}[Y_c] \iff \text{billing}[Y_b]$$

$\phi_2$: $\text{card}[\text{tel}] = \text{billing}[\text{phn}] \rightarrow \text{card}[\text{addr}] \iff \text{billing}[\text{post}]$

$\phi_3$: $\text{card}[\text{email}] = \text{billing}[\text{email}] \rightarrow \text{card}[\text{FN, LN}] \iff \text{billing}[\text{FN, SN}]$
Matching dependencies (MDs)

▶ An MD $\phi$ defined on schemas $(R_1, R_2)$:

$$\bigwedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Z_1] \subseteq R_2[Z_2]$$

▶ $\approx_j$’s are similarity relations;
▶ $X_1, X_2$ (resp. $Z_1, Z_2$): attribute lists of $R_1, R_2$

$\phi_1$: card[LN] = billing[SN] $\land$ card[addr] = billing[post] $\land$

card[FN] $\approx_d$ billing[FN] $\rightarrow$ card[$Y_c$] $\subseteq$ billing[$Y_b$]

$\phi_2$: card[tel] = billing[phn] $\rightarrow$ card[addr] $\subseteq$ billing[post]

$\phi_3$: card[email] = billing[email] $\rightarrow$ card[FN, LN] $\subseteq$ billing[FN, SN]

▶ Dynamic semantics for $(D, D') \models \phi$: instances $D = (D_1, D_2)$ and $D' = (D'_1, D'_2)$ of $(R_1, R_2)$ satisfies the MD $\phi$ iff

▶ $D \sqsubseteq D'$
▶ for any tuples $(u, v)$ in $D$, if $\bigwedge_{j \in [1,k]} u[X_1[j]] \approx_j v[X_2[j]]$, then $u[Z_1] \approx v[Z_2]$ in $D'$ and $\bigwedge_{j \in [1,k]} u[X_1[j]] \approx_j v[X_2[j]]$ in $D'$. 
Reasoning about MDs: Derived MDs can add value

From a given set $\Sigma$ of MDs:

$\phi_1$: card[LN] ⇔ billing[SN] $\land$ card[addr] ⇔ billing[post] $\land$
           card[FN] $\approx_d$ billing[FN] $\rightarrow$ card[Yc] ⇔ billing[Yb]

$\phi_2$: card[tel] = billing[phn] $\rightarrow$ card[addr] ⇔ billing[post]

$\phi_3$: card[email] = billing[email] $\rightarrow$ card[FN,LN] ⇔ billing[FN,SN]

we can derive MDs for identifying card[Yc] and billing[Yb]:

card[email, addr] = billing[email, post] $\rightarrow$ card[Yc] ⇔ billing[Yb]

card[LN, tel] = billing[SN, phn] $\land$ card[FN] $\approx_d$ billing[FN]

$\rightarrow$ card[Yc] ⇔ billing[Yb]
Reasoning about MDs: Derived MDs can add value

From a given set $\Sigma$ of MDs:

$\phi_1$: card[LN] ⇔ billing[SN] \land card[addr] ⇔ billing[post] \land 
    card[FN] \approx_d billing[FN] \rightarrow card[Y_c] ⇔ billing[Y_b] 

$\phi_2$: card[tel] = billing[phn] → card[addr] ⇔ billing[post] 


we can derive MDs for identifying card[Y_c] and billing[Y_b]:

$$\text{card[email, addr] = billing[email, post] \rightarrow card}[Y_c] ⇔ billing[Y_b]$$

$$\text{card[LN, tel] = billing[SN, phn] \land card[FN] \approx_d billing[FN] \rightarrow card}[Y_c] ⇔ billing[Y_b]$$

Added value:

- When tuples differ in each of (LN, SN) and (addr, post), they can be identified via derived MDs, but not by the given MDs

- Dynamic semantics of MDs: if $C$ holds then identify $x$ and $y$. 

Wenfei Fan, Floris Geerts and Xibei Jia – A Revival of Integrity Constraints for Data Cleaning 48 / 86
Implication: Generic reasoning

A set $\Sigma$ of MDs entails another MD $\phi$, denoted by $\Sigma \models_m \phi$, iff for any instances $D, D'$, if $(D, D') \models \Sigma$, then $(D, D') \models \phi$

Derived MDs from $\Sigma = \{\phi_1, \phi_2, \phi_3\}$:

- card[email, addr] = billing[email, post] $\rightarrow$ card[Yc] $\Leftrightarrow$ billing[Yb]
- card[LN, tel] = billing[SN, phn] $\land$ card[FN] $\approx_d$ billing[FN] $\rightarrow$ card[Yc] $\Leftrightarrow$ billing[Yb]
Implication: Generic reasoning

A set $\Sigma$ of MDs entails another MD $\phi$, denoted by $\Sigma \models_m \phi$, iff for any instances $D, D'$, if $(D, D') \models \Sigma$, then $(D, D') \models \phi$.

Derived MDs from $\Sigma = \{\phi_1, \phi_2, \phi_3\}$:

\[
\begin{align*}
\text{card[email, addr]} &= \text{billing[email, post]} \rightarrow \text{card[Yc]} \iff \text{billing[Yb]} \\
\text{card[LN, tel]} &= \text{billing[SN, phn]} \land \text{card[FN]} \approx_d \text{billing[FN]} \\
&\rightarrow \text{card[Yc]} \iff \text{billing[Yb]}
\end{align*}
\]

Generic axioms for similarity relations

- **reflexive**: $x \approx x$;
- **symmetric**: if $x \approx y$ then $y \approx x$;
- **subsuming equality**: if $x = y$ then $x \approx y$. 
Implication: Generic reasoning

A set Σ of MDs entails another MD φ, denoted by Σ |=m φ, iff for any instances D, D', if (D, D') |= Σ, then (D, D') |= φ

Derived MDs from Σ = {φ₁, φ₂, φ₃}:


Generic axioms for similarity relations

- reflexive: x ≈ x;
- symmetric: if x ≈ y then y ≈ x;
- subsuming equality : if x = y then x ≈ y.

Derived MDs as logical consequence: no matter how matching rules are interpreted, if Σ is enforced, then so must be φ.
The implication problem for matching dependencies

The implication problem for MDs: to determine, given any $\Sigma$ and $\phi$, whether or not $\Sigma \models_m \phi$.

**Theorem**

The implication problem for matching dependencies is in $\text{PTIME}$. 

Matching dependencies vs. functional dependencies

- **MDs**: across different relations; **FDs**: on a single relation
- **MDs**: equality $\models$, similarity $\approx$, and matching $\leftrightarrow$; **FDs**: equality $\models$ only
- **MDs**: dynamic semantics with $\leftrightarrow$ (on two instances); **FDs**: standard semantics (on a single instance);
- **MDs**: implication via generic reasoning, with added value; **FDs**: if $D \not\models = \phi$, then $D \not\models = \Sigma$ provided that $\Sigma \models = \phi$. 

Implication of MDs: to derive matching rules on unreliable data.
The implication problem for matching dependencies

The implication problem for MDs: to determine, given any $\Sigma$ and $\phi$, whether or not $\Sigma \models_m \phi$.

**Theorem**

The implication problem for matching dependencies is in $PTIME$.

Matching dependencies vs. functional dependencies

- **MDs**: across different relations;
  - FDs: on a single relation
- **MDs**: equality $=, \approx$ and matching $\iff$;
  - FDs: equality $=$ only
- **MDs**: dynamic semantics with $\iff$ (on two instances);
  - FDs: standard semantics (on a single instance);
- **MDs**: implication via generic reasoning, with added value;
  - FDs: if $D \nmid \varphi$, then $D \nmid \Sigma$ provided that $\Sigma \models \varphi$
The implication problem for matching dependencies

The implication problem for MDs: to determine, given any $\Sigma$ and $\phi$, whether or not $\Sigma \models_m \phi$.

**Theorem**

The implication problem for matching dependencies is in $PTIME$.

Matching dependencies vs. functional dependencies

- **MDs**: across different relations;
  - **FDs**: on a single relation
- **MDs**: equality $=$, similarity $\approx$ and matching $\leftrightarrow$;
  - **FDs**: equality $=$ only
- **MDs**: dynamic semantics with $\leftrightarrow$ (on two instances);
  - **FDs**: standard semantics (on a single instance);
- **MDs**: implication via generic reasoning, with added value;
  - **FDs**: if $D \not\models \varphi$, then $D \not\models \Sigma$ provided that $\Sigma \models \varphi$

Implication of MDs: to derive matching rules on unreliable data.
Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
- Constraint-based methods for data cleaning
  - SQL-based CFD violation detection methods
  - CFD-based repairing methods
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues
Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
- Constraint-based methods for data cleaning
  - SQL-based CFD violation detection methods
    - Batch
    - Incremental
  - CFD-based repairing methods
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues
Detection of CFD violations in SQL

Recall:

- a CFD $\varphi = R(X \rightarrow Y, T_p)$ is violated by two tuples $u, v \in D$ if there exists a pattern tuple $t_p \in T_p$ such that

$$u[X] = v[X] \not\approx t_p[X] \text{ but } (u[Y] \neq v[Y] \text{ or } u[Y] = v[Y] \neq t_p[Y]).$$

- A single tuple can already violate a CFD (in contrast to FDs).

Goal: SQL-based detection

Given a database $D$ and a set $\Sigma$ of CFDs, we want to detect all violations for $\Sigma$ in $D$ using SQL only.

- This makes detection feasible in any standard relational DBMS.
- No need for additional functionality.
Detecting CFD violations in SQL: single CFD case

We first focus on a single CFD $\varphi = R(X \rightarrow A, T_p)$:
Detecting CFD violations in SQL: single CFD case

- We first focus on a single CFD $\varphi = R(X \rightarrow A, T_p)$:

- In case $\varphi$ has a constant right-hand side ($t_p[A] = a$):

  $$Q^C_\varphi \text{ SELECT } * \text{ FROM } R \ t, \ T_p \ t_p$$
  $$\text{WHERE } t[X] \neq t_p[X] \& t_p[A] \neq ' ' \& t[A] \neq t_p[A]$$

- This detects single tuple violations

- In case that $\varphi$ has a variable right-hand side ($t_p[A] =$ ' '):

  $$Q^V_\varphi \text{ SELECT DISTINCT } X \text{ FROM } R \ t, \ T_p \ t_p$$
  $$\text{WHERE } t[X] \neq t_p[X] \& t_p[A] = ' '$$
  $$\text{GROUP BY } X$$
  $$\text{HAVING COUNT(DISTINCT A) > 1}$$

- This detects multiple tuple violations

These queries do not change when the pattern tableau $T_p$ changes. (We treat the pattern tableau as an ordinary table.)

Can be trivially extended to CFDs with general RHS
We first focus on a single CFD $\varphi = R(X \rightarrow A, T_p)$:

In case $\varphi$ has a constant right-hand side ($t_p[A] = a$):

$$Q^C_{\varphi} \quad \text{SELECT} \quad * \quad \text{FROM} \quad R \quad t, \quad T_p \quad t_p$$

$$\text{WHERE } t[X] \not\approx t_p[X] \& t_p[A] \neq \text{'}-\text{'} \& t[A] \neq t_p[A]$$

This detects single tuple violations

In case that $\varphi$ has a variable right-hand side ($t_p[A] = \text{'}-\text{'}$):

$$Q^V_{\varphi} \quad \text{SELECT DISTINCT} \quad X \quad \text{FROM} \quad R \quad t, \quad T_p \quad t_p$$

$$\text{WHERE } t[X] \not\approx t_p[X] \& t_p[A] = \text{'}-\text{'}$$

$$\text{GROUP BY } X \quad \text{HAVING COUNT(DISTINCT A)} > 1$$

This detects multiple tuple violations
Detecting CFD violations in SQL: single CFD case

- We first focus on a single CFD $\varphi = R(X \rightarrow A, T_p)$:
- In case $\varphi$ has a constant right-hand side ($t_p[A] = a$):
  $$Q_{\varphi}^C \text{ SELECT } * \text{ FROM } R \ t, \ T_p \ t_p \text{ WHERE } t[X] \neq t_p[X] \& t_p[A] \neq '-' \& t[A] \neq t_p[A]$$
  This detects single tuple violations
- In case that $\varphi$ has a variable right-hand side ($t_p[A] = '-'$):
  $$Q_{\varphi}^V \text{ SELECT DISTINCT } X \text{ FROM } R \ t, \ T_p \ t_p \text{ WHERE } t[X] \neq t_p[X] \& t_p[A] = '-' \text{ GROUP BY } X \text{ HAVING COUNT(DISTINCT A) > 1}$$
  This detects multiple tuple violations

These queries do not change when the pattern tableau $T_p$ changes. (We treat the pattern tableau as an ordinary table.)
Detecting CFD violations in SQL: single CFD case

- We first focus on a single CFD \( \varphi = R(X \rightarrow A, T_p) \):
  - In case \( \varphi \) has a constant right-hand side (\( t_p[A] = a \)):
    \[
    Q_C^\varphi \quad \text{SELECT } * \quad \text{FROM } R \ t, \ T_p \ t_p \\
    \text{WHERE } t[X] \neq t_p[X] \& \ t_p[A] \neq ' - ' \& \ t[A] = t_p[A]
    \]
  - This detects single tuple violations
  - In case that \( \varphi \) has a variable right-hand side (\( t_p[A] = ' - ' \)):
    \[
    Q_V^\varphi \quad \text{SELECT DISTINCT } X \quad \text{FROM } R \ t, \ T_p \ t_p \\
    \text{WHERE } t[X] \neq t_p[X] \& \ t_p[A] = ' - ' \\
    \text{GROUP BY } X \quad \text{HAVING COUNT(DISTINCT } A) > 1
    \]
  - This detects multiple tuple violations

These queries do not change when the pattern tableau \( T_p \) changes.
(We treat the pattern tableau as an ordinary table.)

Can be trivially extended to CFDs with general RHS
We next consider a set $\Sigma$ consisting of multiple CFDs.
We next consider a set $\Sigma$ consisting of multiple CFDs.

**Naive approach:**

1. Treat each CFD in $\Sigma$ separately and use single CFD queries.
2. As many queries are needed as there are CFDs.
3. Merge all CFDs and detect violations using a fixed number of queries.
4. We merge all pattern tableaus into a single (extended) pattern tableau.
5. Use two SQL queries similar to the ones for a single CFD.

Experiments show that the merge approach is very efficient.
We next consider a set $\Sigma$ consisting of multiple CFDs.

Naive approach:
- Treat each CFD in $\Sigma$ separately and use single CFD queries.
We next consider a set $\Sigma$ consisting of multiple CFDs.

**Naive approach:**
- Treat each CFD in $\Sigma$ separately and use single CFD queries.
  - As many queries needed as there are CFDs

Experiments show that the merge approach is very efficient.
Detecting CFD violations in SQL: multiple CFDs

We next consider a set $\Sigma$ consisting of multiple CFDs.

- **Naive** approach:
  - Treat each CFD in $\Sigma$ separately and use single CFD queries.
    - As many queries needed as there are CFDs

- **Merge** approach:
We next consider a set $\Sigma$ consisting of multiple CFDs.

**Naive approach:**
- Treat each CFD in $\Sigma$ separately and use single CFD queries.
  - As many queries needed as there are CFDs.

**Merge approach:**
- Merge all CFDs and detect violations using a fixed number of queries.

Experiments show that the merge approach is very efficient.
We next consider a set $\Sigma$ consisting of multiple CFDs.

**Naive** approach:
- Treat each CFD in $\Sigma$ separately and use single CFD queries.
- $\Rightarrow$ As many queries needed as there are CFDs.

**Merge** approach:
- Merge all CFDs and detect violations using a fixed number of queries.
- We merge all pattern tableaus into a single (extended) pattern tableau.
Detecting CFD violations in SQL: multiple CFDs

We next consider a set $\Sigma$ consisting of multiple CFDs.

**Naive** approach:
- Treat each CFD in $\Sigma$ separately and use single CFD queries.
  $\Rightarrow$ As many queries needed as there are CFDs.

**Merge** approach:
- Merge all CFDs and detect violations using a **fixed** number of queries.
- We **merge** all pattern tableaus into a **single** (extended) pattern tableau.
- Use **two SQL queries** similar to the ones for a single CFD.
We next consider a set Σ consisting of multiple CFDs

**Naive** approach:
- Treat each CFD in Σ separately and use single CFD queries.
  ⇒ As many queries needed as there are CFDs

**Merge** approach:
- Merge all CFDs and detect violations using a fixed number of queries.
- We merge all pattern tableaus into a single (extended) pattern tableau.
- Use two SQL queries similar to the ones for a single CFD.

Experiments show that the merge approach is very efficient.
Detecting CFD violations in SQL: multiple CFDs

Step 1: Merge pattern tableaux
▶ Consider $\varphi_1 = \text{Cust}([\text{CC}, \text{AC}, \text{phn}] \rightarrow [\text{street}, \text{city}, \text{zip}], T_p)$ with

\[
T_p = \begin{array}{cccccc}
\text{CC} & \text{AC} & \text{phn} & \text{street} & \text{city} & \text{zip} \\
- & - & - & - & - & - \\
01 & 908 & - & - & \text{MH} & - \\
\end{array}
\]

▶ Consider $\varphi_2 = \text{Cust}([\text{CC}, \text{AC}] \rightarrow [\text{city}], T'_p)$ with

\[
T'_p = \begin{array}{ccc}
\text{CC} & \text{AC} & \text{city} \\
- & - & - \\
01 & 215 & \text{PHI} \\
44 & 141 & \text{GLA} \\
\end{array}
\]

▶ Then the result of merging the tableaux $\varphi_1$ and $\varphi_2$ is:

\[
T^L_p = \begin{array}{cccc}
\text{id} & \text{CC} & \text{AC} & \text{phn} \\
1 & - & - & - \\
2 & 01 & 908 & - \\
3 & - & - & \text{@} \\
4 & 01 & 215 & \text{@} \\
5 & 44 & 141 & \text{@} \\
\end{array}
\]

and

\[
T^R_p = \begin{array}{cccc}
\text{id} & \text{street} & \text{city} & \text{zip} \\
1 & - & - & - \\
2 & - & \text{MH} & - \\
3 & \text{@} & - & \text{@} \\
4 & \text{@} & \text{PHI} & \text{@} \\
5 & \text{@} & \text{GLA} & \text{@} \\
\end{array}
\]
Detecting CFD violations in SQL: multiple CFDs

Step 1: Merge pattern tableaux (cont’d)

Then the result of merging the tableaux $\varphi_1$ and $\varphi_2$ is:

<table>
<thead>
<tr>
<th>id</th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>908</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>@</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td>215</td>
<td>@</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>141</td>
<td>@</td>
</tr>
</tbody>
</table>

$T_p^L = id CC AC phn$ and $T_p^R = id street city zip$

<table>
<thead>
<tr>
<th>id</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>MH</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>@</td>
<td>-</td>
<td>@</td>
</tr>
<tr>
<td>4</td>
<td>@</td>
<td>PHI</td>
<td>@</td>
</tr>
<tr>
<td>5</td>
<td>@</td>
<td>GLA</td>
<td>@</td>
</tr>
</tbody>
</table>

Merging an additional CFD $\varphi_3 = Cust([city] \rightarrow [zip], T''_p)$

with $T''_p = city zip$ results in

<table>
<thead>
<tr>
<th>id</th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>@</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>908</td>
<td>-</td>
<td>@</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td>215</td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>141</td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td>6</td>
<td>@</td>
<td>@</td>
<td>@</td>
<td>-</td>
</tr>
</tbody>
</table>

$T_p^L = id CC AC phn city$ and $T_p^R = id street city zip$

<table>
<thead>
<tr>
<th>id</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>MH</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>@</td>
<td>-</td>
<td>@</td>
</tr>
<tr>
<td>4</td>
<td>@</td>
<td>PHI</td>
<td>@</td>
</tr>
<tr>
<td>5</td>
<td>@</td>
<td>GLA</td>
<td>@</td>
</tr>
<tr>
<td>6</td>
<td>@</td>
<td>@</td>
<td>@</td>
</tr>
</tbody>
</table>
Detecting CFD violations in SQL: multiple CFDs

Attributes $T_p^L$: $Z = \{B_i\}$; attributes $T_p^R$: $W = \{C_j\}$.

Step 2: Single tuple violations:

\[
\begin{align*}
\text{Step 2: Single tuple violations:} \\
&\quad Q^C_{S} \quad \text{SELECT} \quad * \quad \text{FROM} \quad R \quad t, \quad T_p^L \quad t_p^L, \quad T_p^R \quad t_p^R \\
&\quad \text{WHERE} \quad t_p^L[id] = t_p^R[id] \land t[Z] \simeq t_p^L[Z] \land t[W] \neq t_p^R[W]
\end{align*}
\]

Step 3: Multiple tuple violations:

\[
\begin{align*}
\text{Step 3: Multiple tuple violations:} \\
&\quad Q^V_{S} \quad \text{SELECT DISTINCT} \quad Z \quad \text{FROM Macro } t^M \\
&\quad \text{GROUP BY } Z \quad \text{HAVING COUNT(DISTINCT } W)>1 \\
&\quad \text{where Macro is:} \\
&\quad \text{SELECT (CASE } t_p^L[B_i] \text{ WHEN ‘@’ THEN ‘@’ ELSE } t[B_i] \text{ END) AS } B_i \ldots \\
&\quad \text{(CASE } t_p^R[C_j] \text{ WHEN ‘@’ THEN ‘@’ ELSE } t[C_j] \text{ END) AS } C_j \ldots \\
&\quad \text{FROM } R \quad t, \quad T_p^L \quad t_p^L, \quad T_p^R \quad t_p^R \\
&\quad \text{WHERE} \quad t_p^L[id] = t_p^R[id] \land t[Z] \simeq t_p^L[Z] \land
\quad (t_p^R[C_1] = ‘_’ \lor \cdots \lor t_p^R[C_n] = ‘_’) \\
\end{align*}
\]

Works efficient in practice and can be generalized to extended CFDs.
Multiple tuple violation query: Example

- Recall merged tableaux:

\[
T_p^L = \begin{array}{ccccc}
1 & - & - & - & @ \\
2 & 01 & 908 & - & @ \\
3 & - & - & @ & @ \\
4 & 01 & 215 & @ & @ \\
5 & 44 & 141 & @ & @ \\
6 & @ & @ & @ & @ \\
\end{array} \quad \text{and} \quad T_p^R = \begin{array}{cccc}
1 & - & - & - \\
2 & - & MH & - \\
3 & @ & - & @ \\
4 & @ & PHI & @ \\
5 & @ & GLA & @ \\
6 & @ & @ & @ \\
\end{array}
\]

- Database,

<table>
<thead>
<tr>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t2</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t3</td>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>John Oxford Ter.</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
</tbody>
</table>
Multiple tuple violation query: Example

- Recall merged tableaux:

  \[ T_p^L = \]

  \begin{tabular}{|c|c|c|c|c|}
  \hline
  id & CC & AC & phn & city \\
  \hline
  1 & _ & _ & _ & @ \\
  2 & 01 & 908 & _ & @ \\
  3 & _ & _ & @ & @ \\
  4 & 01 & 215 & @ & @ \\
  5 & 44 & 141 & @ & @ \\
  6 & @ & @ & @ & _ \\
  \hline
  \end{tabular}

  and

  \[ T_p^R = \]

  \begin{tabular}{|c|c|c|c|c|}
  \hline
  id & street & city & zip \\
  \hline
  1 & _ & _ & _ \\
  2 & _ & MH & _ \\
  3 & @ & _ & @ \\
  4 & @ & PHI & @ \\
  5 & @ & GLA & @ \\
  6 & @ & @ & @ \\
  \hline
  \end{tabular}

- Database, Macro relation,

  \begin{tabular}{|c|c|c|c|c|c|c|c|}
  \hline
  CC & AC & phn & city & street & city & zip \\
  \hline
  44 & 131 & 1234567 & @ & Mayfield & EDI & EH4 8LE \\
  44 & 131 & @ & @ & @ & EDI & @ \\
  @ & @ & @ & EDI & @ & @ & EH4 8LE \\
  44 & 131 & 3456789 & @ & Crichton & NYC & EH4 8LE \\
  44 & 131 & @ & @ & @ & NYC & @ \\
  @ & @ & @ & NYC & @ & @ & EH4 8LE \\
  44 & 131 & 5678910 & @ & Oxford Ter. & EDI & EH4 8LE \\
  \hline
  \end{tabular}
Multiple tuple violation query: Example

- Recall merged tableaux:

\[
T^L_p = \begin{array}{cccc}
1 & - & - & @ \\
2 & 01 & 908 & @ \\
3 & - & - & @ & @ \\
4 & 01 & 215 & @ & @ \\
5 & 44 & 141 & @ & @ \\
6 & @ & @ & @ & @
\end{array}
\]

\[
T^R_p = \begin{array}{cccc}
1 & - & - & - \\
2 & - & @ & - \\
3 & @ & - & @ \\
4 & @ & @ & @ \\
5 & @ & @ & @ \\
6 & @ & @ & @
\end{array}
\]

- Database, Macro relation, Group by to detect violations.

<table>
<thead>
<tr>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>city</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>@</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>@</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>@</td>
<td>Oxford Tre.</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>@</td>
<td>@</td>
<td></td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>@</td>
<td>@</td>
<td></td>
<td>NYC</td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>@</td>
<td>@</td>
<td>EDI</td>
<td></td>
<td></td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>@</td>
<td>@</td>
<td>@</td>
<td>NYC</td>
<td></td>
<td></td>
<td>EH4 8LE</td>
</tr>
</tbody>
</table>
Goal:
Avoid detection of violations from scratch after updates are issued.

- Key idea: materialize auxiliary relation $\text{Aux}(D)$
- $\text{Aux}(D)$ stores condensed representation of violations in current databases.
- An incremental SQL-based detection method can be devised that leverages $\text{Aux}(D)$.
  - Tuple insertions and deletions are treated differently.
  - Incremental nature requires SQL update statements.
- Incremental method outperforms naive (from scratch) method already for reasonably sized updates.


Data quality: An overview

Revisions of constraints for improving the quality of data

Constraint-based methods for data cleaning
  - SQL-based CFD violation detection methods
  - Constraint-based repairing
    - Different repair models
    - FD and CFD repair methods
    - Incremental repair methods
    - Semi-automated repair methods
    - Consistent query answering

Demo: SemanDaQ, a constraint-based data cleaning tool

Open research issues
Constraint-based data cleaning

Constraint-based data repairing

- Input: a relational database $D$ and a set $\Sigma$ of constraints.
- Output: a repair $D'$ of $D$ such that $D'$ satisfies $\Sigma$.

Depends on the repair model and class of constraints considered.

Most common repair models (list is not exhaustive):

- $D$-repair: find maximal subset $D' \subseteq D$ such that $D' \models \Sigma$. (tuple deletions.)

- $S$-repair: find $D'$ such that $(D \setminus D') \cup (D' \setminus D)$ is minimal and $D' \models \Sigma$. (symmetric difference.)

- $U$-repair: find $D'$ such that $D' \models \Sigma$ and cost $(D', D)$ is minimal, where cost is a some cost function. (value modifications.)
# Repair checking problem

<table>
<thead>
<tr>
<th>Repair checking problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Input: two relational databases $D$, $D'$ and a set $\Sigma$ of constraints.</td>
</tr>
<tr>
<td>▶ Output: “yes” if $D'$ is a repair of $D$; “no” otherwise.</td>
</tr>
</tbody>
</table>

Studied already in a variety of contexts and is non-trivial:

## Theorem

*The repair checking problem is*

- *coNP-complete for full dependencies and S-repairs.*
- *in PTIME for FDs and acyclic INDs and D-repairs.*
- *coNP-complete for one FD and one IND taken together for D-repairs.*
The repair checking problem: $U$-repairs

- The $U$-repair model is the most flexible and most commonly used in practice:
  - Allows for value modifications.
  - Limits deletion of tuples, i.e., minimizes loss of information.
  - Limits creation of entirely new tuples.
- Depends on choice of an appropriate cost function, e.g.,

$$\text{cost}(D', D) = \sum_{t \in D} \sum_{t' \in D'} \sum_{A \in R} w(t, A) \cdot \text{dist}(t[A], t'[A])$$

$t'$ is the tuple in $D'$ corresponding to $t$.

$w(t, A)$ denotes the accuracy of the attribute $A$.

$\text{dist}(\cdot, \cdot)$ measures the distance between values.

**Theorem**

The repair checking problem is NP-complete for either a fixed set of FDs, CFDs or INDs for $U$-repairs with the above cost function.
The repair checking problem: $U$-repairs

- The $U$-repair model is the most flexible and most commonly used in practice:
  - Allows for value modifications.
  - Limits deletion of tuples, i.e., minimizes loss of information.
  - Limits creation of entirely new tuples.
- Depends on choice of an appropriate cost function, e.g.,

$$\text{cost}(D', D) = \sum_{t \in D} \sum_{t' \in D'} \sum_{A \in R} w(t, A) \cdot \text{dist}(t[A], t'[A])$$

- $t'$ is the tuple in $D'$ corresponding to $t$. 
The repair checking problem: $U$-repairs

- The $U$-repair model is the most **flexible** and most commonly **used** in practice:
  - Allows for **value modifications**.
  - Limits deletion of tuples, i.e., minimizes loss of information.
  - Limits creation of entirely new tuples.
- Depends on choice of an appropriate **cost function**, e.g.,

$$
\text{cost}(D', D) = \sum_{t \in D, t' \in D'} \sum_{A \in R} w(t, A) \cdot \text{dist}(t[A], t'[A])
$$

- $t'$ is the tuple in $D'$ corresponding to $t$.
- $w(t, A)$ denotes the **accuracy** of the attribute $A$. 

The repair checking problem: $U$-repairs

- The $U$-repair model is the most flexible and most commonly used in practice:
  - Allows for value modifications.
  - Limits deletion of tuples, i.e., minimizes loss of information.
  - Limits creation of entirely new tuples.
- Depends on choice of an appropriate cost function, e.g.,

\[
\text{cost}(D', D) = \sum_{t \in D} \sum_{t' \in D'} \sum_{A \in R} w(t, A) \cdot \text{dist}(t[A], t'[A])
\]

- $t'$ is the tuple in $D'$ corresponding to $t$.
- $w(t, A)$ denotes the accuracy of the attribute $A$.
- $\text{dist}(\ , \ )$ measures the distance between values.
The repair checking problem: \textit{U}-repairs

- The \textit{U}-repair model is the most \textbf{flexible} and \textbf{most commonly used} in practice:
  - Allows for \textbf{value modifications}.
  - \textbf{Limits deletion} of tuples, i.e., minimizes \textbf{loss of information}.
  - \textbf{Limits creation} of entirely new tuples.
- Depends on choice of an appropriate \textbf{cost function}, e.g.,
  
  \[
  \text{cost}(D', D) = \sum_{t \in D, t' \in D'} \sum_{A \in R} w(t, A) \cdot \text{dist}(t[A], t'[A])
  \]
  
  - $t'$ is the tuple in $D'$ corresponding to $t$.
  - $w(t, A)$ denotes the \textbf{accuracy} of the attribute $A$.
  - $\text{dist}(\ , \ )$ measures the \textbf{distance} between \textbf{values}.

\textbf{Theorem}

The repair checking problem is \textbf{NP-complete} for either a \textbf{fixed set} of FDs, CFDs or INDs for \textit{U}-repairs with the above cost function.
Finding repairs

Repair checking algorithms do not (always) provide a repair...

- Finding a repair a major algorithmic challenge.
- Key to success of constraint-based data cleaning.
- Algorithms exist for census data:
  - Constraints: “edits” (not very expressive)
  - Tuple-based: no interaction between tuples (in contrast to e.g., FDs)
  - Small amount of data (in contrast to size of typical database)
  - Has been applied to both categorical and quantitative data

Challenge

How to find a $U$-repair $D'$ of $D$ with respect to a set of FDs, CFDs or CINDs in an automated way?
Finding a $U$-repair

<table>
<thead>
<tr>
<th>$U$-repair</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> a relational database $D$ and a set $\Sigma$ of constraints</td>
</tr>
<tr>
<td><strong>Output:</strong> a repair $D'$ of $D$ such that $\text{cost}(D', D)$ is minimal and $D' \models \Sigma$.</td>
</tr>
</tbody>
</table>

- From the previous results on repair checking problem:
  - Intractable for simple constraints such as FDs, INDs, and CFDs.
  - We necessarily have to rely on heuristics.
- We first look at FDs, then consider CFDs.
Resolving FD-violations

<table>
<thead>
<tr>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- An **FD-violation** can always be resolved by performing a chase on the database, i.e., by “merging equivalence classes”: 
Resolving FD-violations

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁:</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₂:</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₃:</td>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₄:</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- An **FD-violation** can always be **resolved** by performing a chase on the database, i.e., by “**merging equivalence classes**”:
- Initially, each \((t, A)\) is its own eq class with **target value** \(t[A]\).
Resolving FD-violations

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_2$</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_3$</td>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_4$</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- An **FD-violation** can always be resolved by performing a chase on the database, i.e., by “merging equivalence classes”:
- Initially, each $(t, A)$ is its own eq class with target value $t[A]$.
- For $fd_1 : R([\text{zip}] \rightarrow [\text{city}])$, $(t_1, \text{zip}), (t_2, \text{zip})$ and $(t_3, \text{zip})$ are merged, target value is selected based on cost function: e.g., EDI
Resolving FD-violations

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$:</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_2$:</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_3$:</td>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_4$:</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- An FD-violation can always be resolved by performing a chase on the database, i.e., by “merging equivalence classes”:
- Initially, each $(t, A)$ is its own eq class with target value $t[A]$.
- For $fd_1 : R([zip] → [city])$, $(t_1, zip), (t_2, zip)$ and $(t_3, zip)$ are merged, target value is selected based on cost function: e.g., EDI
- For $fd_2 : R([name, street, zip] → [phn])$, $(t_2, phn), (t_3, phn)$ are merged, target value is selected based on cost function, e.g., 3456789
Resolving FD-violations

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_2'$</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>$t_4$</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>

- An **FD-violation** can always be resolved by performing a chase on the database, i.e., by “merging equivalence classes”:
  - Initially, each $(t, A)$ is its own eq class with target value $t[A]$.
  - For $fd_1 : R([zip] \rightarrow [city])$, $(t_1, zip),(t_2, zip)$ and $(t_3, zip)$ are merged, target value is selected based on cost function: e.g., EDI
  - For $fd_2 : R([name, street, zip] \rightarrow [phn])$, $(t_2, phn),(t_3, phn)$ are merged, target value is selected based on cost function, e.g., 3456789
- Finally, equivalence classes are materialized with selected target values $\Rightarrow$ repair.
Resolving FD-violations:

- Different processing order of FDs results in different repairs.
- Heuristics needed to decide the best order, i.e., order that minimizes cost function.

Heuristic: Greedy Repair

- Keep track of number of tuples violated by FDs.
- Select the next FD to be processed that resolves the most violations.
- Resolve violations of current FD.
- Repeat until no violations are left.

Theorem

_Greedy Repair_ terminates and produces a repair $D'$ of $D$ in PTIME.

- Can be extended to work for INDs.
Finding a repair for CFDs

Can the same approach still be applied for CFDs?

- $\text{cfd}_1 : R(A \rightarrow B, (\_||\_))$, $\text{cfd}_2 : R(C \rightarrow B, \{(c_1||b_1), (c_2||b_2)\})$.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$:</td>
<td>a</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$t_2$:</td>
<td>a</td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
</tbody>
</table>

Start with initial equivalence classes
- For $\text{cfd}_1$ merge $(t_1, B)$ and $(t_2, B)$. Target value is either $b_1$ or $b_2$.
- For $\text{cfd}_2$ equivalence class $\{(t_1, B), (t_2, B)\}$ should be split!
  But then $\text{cfd}_1$ is violated again!

A naive application does not lead to a repair in the case of CFDs...
Finding a repair for CFDs

Can the same approach still be applied for CFDs?

▷ \text{cfd}_1 : \mathbf{R}(A \rightarrow B, (_\parallel_)), \text{cfd}_2 : \mathbf{R}(C \rightarrow B, \{(c_1 \parallel b_1), (c_2 \parallel b_2)\}).

\begin{array}{|c|c|c|}
\hline
A & B & C \\
\hline
\text{t}_1 : & a & b_1 & c_1 \\
\text{t}_2 : & a & b_2 & c_2 \\
\hline
\end{array}

▷ Start with initial equivalence classes
Finding a repair for CFDs

Can the same approach still be applied for CFDs?

- \( \text{cfd}_1 : R(A \rightarrow B, (\_ \parallel \_)), \text{cfd}_2 : R(C \rightarrow B, \{(c_1 \parallel b_1), (c_2 \parallel b_2)\}) \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1):</td>
<td>a</td>
<td>(b_1)</td>
<td>(c_1)</td>
</tr>
<tr>
<td>(t_2):</td>
<td>a</td>
<td>(b_2)</td>
<td>(c_2)</td>
</tr>
</tbody>
</table>

- Start with initial equivalence classes
- For \( \text{cfd}_1 \) merge \((t_1, B)\) and \((t_2, B)\). Target value is either \(b_1\) or \(b_2\).
Finding a repair for CFDs

Can the same approach still be applied for CFDs?

- $\text{cfd}_1: R(A \rightarrow B, (\_\parallel \_)), \text{cfd}_2: R(C \rightarrow B, \{(c_1\parallel b_1), (c_2\parallel b_2)\})$.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$:</td>
<td>$a$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$t_2$:</td>
<td>$a$</td>
<td>$b_2$</td>
</tr>
</tbody>
</table>

- Start with initial equivalence classes
- For $\text{cfd}_1$ merge $(t_1, B)$ and $(t_2, B)$. Target value is either $b_1$ or $b_2$.
- For $\text{cfd}_2$ equivalence class $\{(t_1, B), (t_2, B)\}$ should be split! But then $\text{cfd}_1$ is violated again!
Can the same approach still be applied for CFDs?

- cfd₁ : \( R(A \rightarrow B, (\_\|\_)) \), cfd₂ : \( R(C \rightarrow B, \{(c_1\|b_1), (c_2\|b_2)\}) \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁:</td>
<td>a</td>
<td>b₁</td>
<td>c₁</td>
</tr>
<tr>
<td>t₂:</td>
<td>a</td>
<td>b₂</td>
<td>c₂</td>
</tr>
</tbody>
</table>

- Start with initial equivalence classes
- For cfd₁ merge \((t₁, B)\) and \((t₂, B)\). **Target value** is either \(b₁\) or \(b₂\).
- For cfd₂ equivalence class \\{\((t₁, B), (t₂, B)\)\} should be split!

But then cfd₁ is violated again!

A naive application does not lead to a repair in the case of CFDs ...
Greedy Repair for CFDs

- Pattern tuples in CFDs enforce target values to equivalence classes, preventing them to be merged.
- Equivalence classes in left-hand sides of dependencies need to be modified.
- Sometimes no target values can be assigned and we have to put null.
Greedy Repair for CFDs

- Pattern tuples in CFDs enforce target values to equivalence classes, preventing them to be merged.
- Equivalence classes in left-hand sides of dependencies need to be modified.
- Sometimes no target values can be assigned and we have to put null.

First cfd₁: equivalence class is locked; Target value $b₁$
Greedy Repair for CFDs

- **Pattern tuples** in CFDs enforce target values to equivalence classes, preventing them to be merged.
- Equivalence classes in **left-hand sides** of dependencies need to be modified.
- Sometimes no target values can be assigned and we have to put null.

\[
\begin{align*}
\text{cfd}_1 & : R(A \rightarrow B, (-\|-)) \\
\text{cfd}_2 & : R(C \rightarrow B, \{(c_1\|b_1), (c_2\|b_2)\})
\end{align*}
\]

\[
\begin{array}{|c|c|c|}
\hline
A & B & C \\
\hline
a & b_1 & c_1 \\
\hline
a & b_2 & c_1 \\
\hline
\end{array}
\]

First cfd\(_1\): equivalence class is **locked**; Target value \(b_1\)

To resolve cfd\(_2\): Modify **left-hand side** (C-attribute)
Greedy Repair for CFDs

- Pattern tuples in CFDs enforce target values to equivalence classes, preventing them to be merged.
- Equivalence classes in left-hand sides of dependencies need to be modified.
- Sometimes no target values can be assigned and we have to put null.

\[
\begin{align*}
\text{cfd}_1 & : R(A \rightarrow B, (-\parallel -)) \\
\text{cfd}_2 & : R(C \rightarrow B, \{(c_1\parallel b_1), (c_2\parallel b_2)\})
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>(b_1)</td>
<td>(c_1)</td>
</tr>
</tbody>
</table>

First cfd\(_1\): equivalence class is locked; Target value \(b_1\)
To resolve cfd\(_2\): Modify left-hand side (C-attribute)
Greedy Repair for CFDs

- Pattern tuples in CFDs enforce target values to equivalence classes, preventing them to be merged.
- Equivalence classes in left-hand sides of dependencies need to be modified.
- Sometimes no target values can be assigned and we have to put null.

\[
\begin{align*}
\text{cfd}_1 & : R(A \rightarrow B, (\_||\_)) \\
\text{cfd}_2 & : R(C \rightarrow B, \{(c_1||b_1), (c_2||b_2)\})
\end{align*}
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b₁</td>
<td>c₁</td>
</tr>
<tr>
<td>a</td>
<td>b₂</td>
<td>c₂</td>
</tr>
</tbody>
</table>

First cfd₁: equivalence class is locked; Target value \(b₁\)
To resolve cfd₂: Modify left-hand side (C-attribute)

Or, first cfd₂: equivalence classes are locked together with values;
Greedy Repair for CFDs

- Pattern tuples in CFDs enforce target values to equivalence classes, preventing them to be merged.
- Equivalence classes in left-hand sides of dependencies need to be modified.
- Sometimes no target values can be assigned and we have to put null.

\[
\begin{align*}
cfd_1 &: R(A \rightarrow B, (-\|-_)) \\
cfd_2 &: R(C \rightarrow B, \{(c_1\|b_1), (c_2\|b_2)\})
\end{align*}
\]

First cfd_1: equivalence class is locked; Target value b_1
To resolve cfd_2: Modify left-hand side (C-attribute)

Or, first cfd_2: equivalence classes are locked together with values;
To resolve cfd_1: Modify left-hand side (A-attribute): null value
Greedy Repair for CFDs

- Equivalence class-based approach can be extended to CFDs.
  - Left and right hand side attributes need to be changed.
  - Equivalences classes that are merged cannot be split (locking).
  - However, null values might still need to be introduced.

- By selecting the best next CFD (that resolves the most violations) we obtain a greedy algorithm for repairing databases using CFDs.

- Patterns in CFDs are used as much as possible to select the target values of the equivalence classes.

**Theorem**

*Greedy repair always terminates and produces a repair $D'$ for a given set $\Sigma$ of CFDs in PTIME.*
Pattern tuples indeed help improving the quality of repairs:

- Consider the CFD

\[
\text{cfd}_3: ([\text{CC} = 01, \text{AC} = 908, \text{phn}] \rightarrow [\text{street}, \text{city} = '\text{MH}', \text{zip}]).
\]

<table>
<thead>
<tr>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁:</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₂:</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₃:</td>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>Rick</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₄:</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>NYC</td>
<td>07974</td>
</tr>
</tbody>
</table>
Pattern tuples indeed help improving the quality of repairs:

- Consider the CFD
cfd₃: ([CC = 01, AC = 908, phn] → [street, city = ‘MH’, zip]).

- Forces target value of (t₄, city) to be ‘MH’.

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁:</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₂:</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>NYC</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₃:</td>
<td>44</td>
<td>131</td>
<td>5678910</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>t₄:</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>MH</td>
<td>07974</td>
</tr>
</tbody>
</table>
Pattern tuples indeed help improving the quality of repairs:

- Consider the CFD
  \[ \text{cfd}_3: ([\text{CC} = 01, \text{AC} = 908, \text{phn}] \rightarrow [\text{street}, \text{city} = ‘\text{MH’}, \text{zip}]). \]
- Forces target value of \((t_4, \text{city})\) to be ‘\text{MH’}.

<table>
<thead>
<tr>
<th></th>
<th>CC</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_1):</td>
<td>44</td>
<td>131</td>
<td>1234567</td>
<td>Mike</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>(t_2):</td>
<td>44</td>
<td>131</td>
<td>3456789</td>
<td>Rick</td>
<td>Crichton</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>(t_3):</td>
<td>01</td>
<td>908</td>
<td>3456789</td>
<td>Joe</td>
<td>Mtn Ave</td>
<td>MH</td>
<td>07974</td>
</tr>
</tbody>
</table>
Two related repairing problems

Incremental data repairing

- Input: a relational database $D$, a set $\Sigma$ of constraints, $D \models \Sigma$ and an update $\Delta D$.
- Output: a repair $D'$ of $D \oplus \Delta D$ such that $D'$ satisfies $\Sigma$.

Local repairing problem

- Input: a relational database $D$, a set $\Sigma$ of constraints, $D \models \Sigma$ and a tuple $t$.
- Output: a repair $D'$ of $D \oplus \{t\}$ such that $D'$ satisfies $\Sigma$.

Theorem

The incremental and local repair problems are both NP-complete.
The incremental repairing problem can be solved using local repairing:
- Add one tuple at a time

The local repair problem allows alternative repairing methods:
- For the inserted tuple $t$
  - Find the best set of attributes $X$ in which to modify $t$ such that $D \cup \{\hat{t}\} \models \Sigma$.

Various greedy strategies can be devised to find the best set of attributes.

Values from $D$ can be used to modify $t$ (values in $D$ are “clean”)

Incremental repairing methods seem very promising in practice.
Semi-automated repairing: Sampling

- So far, all repairing methods were automated.
- It is often desirable to involve human experts in this process.
- One way is to use a sampling approach:
  1. After a repair $D'$ is found; provide a sample $\hat{D}'$ to the user;
  2. The user inspects this sample and improves it, the result replaces $\hat{D}'$ in $D'$;
  3. Possible new violations are introduced in $D'$; repair again and present user with new sample.
- One can also allow the user to modify the set of constraints ("dirty constraints").

Under certain statistical assumptions, the sampling approach guarantees accurate repairs, i.e., repairs that are close to what the user wants (performance guarantee).
Avoiding finding repairs: Consistent query answering

- Avoid choosing a single repair \( D' \).
- Instead, find the answer to a query \( Q \) on all possible repairs \( D' \).
- Only tuples are included that are present in \( Q(D') \) for each repair \( D' \).
  - Challenge: without materializing all possible repairs.
- Complexity and methods depend on repair model and query language.
- Long line of research (tutorial on its own).

Interesting to look at consistent query answering in the presence of CFDs, CINDs ....
References

**Repair checking:**


**D-repairs**


**S-repairs**


**Repairing algorithms:**

**Census**


**FDs/INDs**


**CFDs**


**Consistent query answering:**


Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
- Constraint-based methods for data cleaning
- **Demo: SemanDaQ, a constraint-based data cleaning tool**
- Open research issues
Demo: SemanDaQ

SemanDaQ (Semantic Data Quality) — a constraint-based data cleaning tool

We will demonstrate

- Conditional Functional Dependencies (CFDs)
- Data exploration guided by constraints
- Inconsistencies Detection
- Data quality map
- Data quality report
- Constraints repair
- Data cleaning review (data auditing)
Data cleaning tools in research community

Data Profiling

- Bellman (AT&T), a data quality browser, finding keys, join path (SIGMOD’02, CleanDB’06)

Error Correction

- Potter’s Wheel (UC Berkeley), interactive data cleaning (VLDB’01)
- AJAX (INRIA), declarative data cleaning as a flow of operations (SIGMOD’00 demo)

Record Matching

- IntelliClean (NUS), rule-based record matching (KDD’00)
- Febrl (ANU), data standardisation (segmentation and cleaning) and probabilistic record linkage ("fuzzy" matching), open source

Consistent Query Answering

- ConQuer (Univ. of Toronto), for primary key constraints and a large subset of conjunctive queries (SIGMOD’05 demo)
- Hippo (SUNY at Buffalo), for denial constraints and quantifier-free first-order queries (CIKM’04)
Commercial data cleaning software

**IBM** Information Server **QualityStage**
- Investigate Stage → Standardize Stage → Match Stages → Survive Stage
- Data cleaning is conducted in **Standardize Stage** by using three classes of rule sets (domain preprocessor, domain specific and validation) and in **Survive Stage** which creates a best representation of the matched data

**Microsoft** SQL Server Integration Services (**SSIS**) 2005
- Profiling → Cleansing → Auditing, employing Fuzzy Lookup and Fuzzy Grouping

**Oracle** Warehouse Builder (**OWB**) 10gR2
- Profiling → Cleansing

Software from other vendors focusing on data profiling, record matching, address cleansing and geocoding (enrichment):
- **DataFlux (SAS)**: dfPower Studio
- **Trillium (Harte-Hanks)**: TS Insight, TS Quality, TS Enrichment ...
- **Business Objects (SAP)**: Data Quality XI, Data Insight XI
- **Group 1 (PitneyBowes)**: Customer Data Quality Platform, ...
- **Informatica, Human Inference, Uniserv, Innovative Systems, DataLever, DataMentors, Netrics, Datanomic, Data Tactics** ...
Outline

- Data quality: An overview
- Revisions of constraints for improving the quality of data
- Constraint-based methods for data cleaning
- Demo: SemanDaQ, a constraint-based data cleaning tool
- Open research issues
Data repairing based on Master Data

Input:
- a master database $D_M$,
- a database $D$,
- a set $\Sigma$ of CFDs defined on $D$, and
- a set $\Gamma$ of MDs on $D_M$ and $D$;

Output: a repair $D'$ of $D$ such that $D' \models \Sigma$ and $(D_1, D_2) \models \Gamma$ where $D_1 = (D, D_M)$ and $D_2 = (D', D_M)$.

Open issues:
- The interaction between data repairing and object identification: how to efficiently combine the two processes?
- The interaction between object identification and schema matching: how to effectively derive schema matching from MDs and vice versa, when $D$ and $D_M$ have different schemas?

Data repairing by leveraging master data
Detecting stale tuples

- Input: a database $D$;
- Question: can we identify stale tuples from $D$? What semantic rules do we need for this task?

Example: **NI#** is a key of the following relation

<table>
<thead>
<tr>
<th>NI#</th>
<th>AC</th>
<th>phn</th>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC1234566</td>
<td>131</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Mayfield</td>
<td>EDI</td>
<td>EH4 8LE</td>
</tr>
<tr>
<td>SC1234566</td>
<td>020</td>
<td>1234567</td>
<td>M. Smith</td>
<td>Portland</td>
<td>LDN</td>
<td>W1B 1JL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Removing any one of the two tuples leads to a repair. But which one is **correct**?
- Is it necessary to add a temporal attribute to each tuple?
- One may need to keep both records. How to adjust the semantics of constraints to accommodate this?
Missing tuples vs. missing values

- Input: a database $D$ and a query $Q$;
- Question: can $Q$ be answered given the information in $D$?

Example query: for a group of patients, find their family medical history of the last three generations.

- How to characterize the completeness of $D$ for $Q$?
- Given $Q$ and $D$, how to determine whether or not $D$ is complete for $Q$?
- The study of incomplete information has mostly focused on missing values. What about missing tuples?

Summary

Revising traditional constraints to improve the quality of data:

- Conditional functional dependencies
- Conditional inclusion dependencies
- Matching dependencies
- Denial constraints, ...
Summary

Revising traditional constraints to improve the quality of data:
- Conditional functional dependencies
- Conditional inclusion dependencies
- Matching dependencies
- Denial constraints, ...

Developing data quality tools based on constraints:
- Techniques for reasoning about data quality rules
- Repairing methods
- Profiling algorithms, ...
Summary

Revising traditional constraints to improve the quality of data:
▶ Conditional functional dependencies
▶ Conditional inclusion dependencies
▶ Matching dependencies
▶ Denial constraints, ...

Developing data quality tools based on constraints:
▶ Techniques for reasoning about data quality rules
▶ Repairing methods
▶ Profiling algorithms, ...

A rich source of questions and vitality
▶ Repairing algorithms with performance guarantee
▶ Constraints in Master Data Management
▶ Incomplete information (missing tuples), ...
Summary

Revising traditional constraints to improve the quality of data:
- Conditional functional dependencies
- Conditional inclusion dependencies
- Matching dependencies
- Denial constraints, ...

Developing data quality tools based on constraints:
- Techniques for reasoning about data quality rules
- Repairing methods
- Profiling algorithms, ...

A rich source of questions and vitality
- Repairing algorithms with performance guarantee
- Constraints in Master Data Management
- Incomplete information (missing tuples), ...

Constraints: A principled approach to improving data quality