A Revival of Integrity Constraints for Data Cleaning

Wenfei Fan[1,2] Floris Geerts[1] Xibei Jia[1]

[1]School of Informatics, University of Edinburgh

[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

NI#	AC	phn	name	street	city	zip
⋮						
SC1234566	131	1234567	M. Smith	Mayfield	EDI	EH4 8LE
SC1234566	020	1234567	M. Smith	Portland	LDN	W1B 1JL
⋮						

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

NI#	AC	phn	name	street	city	zip
⋮						
SC1234566	131	1234567	M. Smith	Mayfield	EDI	EH4 8LE
SC1234566	020	1234567	M. Smith	Portland	LDN	W1B 1JL
⋮						

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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\gg$ 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: age $= 82$ and 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: age ≤ 200 vs. age $= 45$
- Completeness: whether a given query can be answered given the information available.
  Example: age $=$ 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.

NI#	AC	phn	name	street	city	zip
...						
SC1234566	131	1234567	M. Smith	Mayfield	EDI	EH4 8LE
SC1234566	020	1234567	M. Smith	Portland	LDN	W1B 1JL
...						

Functional dependency: NI# $\rightarrow$ 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:

- R. Fagin and M. Y. Vardi. The theory of data dependencies - An overview. In *ICALP*, 1984.

Surveys on data quality

- C. Batini and M. Scannapieco. Data Quality: Concepts, Methodologies and Techniques. Springer, 2006.
- E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. IEEE Data Eng. Bull. 23(4): 3-13, 2000.

Sources of the statistics

- W. W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. The Data Warehousing Institute, 2002.
- L. English. Plain English on data quality: Information quality management: The next frontier. DM Review Magazine, April 2000.
- Gartner. Forecast: Data quality tools, worldwide, 2006-2011. 2007.
- C. C. Shilakes and J. Tylman. Enterprise information portals. Merrill Lynch, 1998.
- T. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM* 41(2): 79-82, 1998.

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), ...

  Cust(CC: int, AC: int, phn: int, name: string, street: string, city: string, zip: string)

- Instance:

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

The need for revising traditional constraints

- Instance:

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

- 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

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

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

  cfd_1: ([CC = 44, zip] $\rightarrow$ [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

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

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

  cfd_1: ([CC = 44, zip] $\rightarrow$ [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 fd_1: [CC, AC, phn] $\longrightarrow$ [street, city, zip] by adding conditions (patterns of semantically related constants)

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

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])

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

None of the tuples in the example database is clean

The need for the revision: to improve the quality of data

cfd_1: ([CC = 44, zip] → [street])
cfd_2: ([CC = 44, AC = 131, phn] → [street, city = 'EDI', zip])
cfd_3: ([CC = 01, AC = 908, phn] → [street, city = 'MH', 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 φ;
  - 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 dom(A) or a wildcard '_' (unnamed variable) that draws values from dom(A).
- A single CFD representing cfd_2, cfd_3 and FD fd_1:
  - Cust([CC, AC, phn] $\rightarrow$ [street, city, zip], T_p)
  - pattern tableau T_p:

CC	AC	phn	street	city	zip
44	131	_	_	EDI	_
01	908	_	_	MH	_
_	_	_	_	_	_

Each pattern tuple t_p is a constraint

Special cases of CFDs

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

CC	AC	phn	street	city	zip
_	_	_	_	_	_

Special cases of CFDs

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

CC	AC	phn	street	city	zip
_	_	_	_	_	_

- Constant CFDs:
  - $Cust([CC, AC] \rightarrow [city], T_p)$
  - pattern tableau T_p:

CC	AC	city
44	131	EDI
01	908	MH

The semantics of CFDs

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

The semantics of CFDs

- Operator $\asymp$ for matching patterns:
  - a matches b ($a \asymp b$)
    - either a or b is _
    - both a and b are constants and $a = b$.
  - tuple t_1 matches tuple t_2 ($t_1 \asymp t_2$): defined component-wise
    - $(a, b) \asymp (a, \_)$ but $(a, b) \not\asymp (a, c)$.
- 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 $u[X] = v[X] \asymp t_p[X]$, then $u[Y] = v[Y] \asymp t_p[Y]$

The semantics of CFDs

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

- 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 $u[X] = v[X] \asymp t_p[X]$, then $u[Y] = v[Y] \asymp t_p[Y]$

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

Violation of CFDs

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

T_p:

	CC	AC	phn	street	city	zip
	_	_	_	_	_	_
	44	131	_	_	EDI	_
t_p	01	908	_	_	MH	_

- Tuple t_3 violates the CFD:
  - $t_3[CC, AC, phn] = t_3[CC, AC, phn] \asymp t_p[CC, AC, phn]$
  - $t_3[city] \not\asymp t_p[city]$

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

Violation of CFDs

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

T_p:

	CC	AC	phn	street	city	zip
	_	_	_	_	_	_
	44	131	_	_	EDI	_
t_p	01	908	_	_	MH	_

- Tuple t_3 violates the CFD:
  - $t_3[CC, AC, phn] = t_3[CC, AC, phn] \asymp t_p[CC, AC, phn]$
  - $t_3[city] \not\asymp t_p[city]$

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

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:

- $\varphi = R(A \to B, T_p)$, where $T_p =$

A	B
_	b_1
_	b_2

- For any nonempty database D and for any tuple t in D, φ 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:

- $\varphi = R(A \to B, T_p)$, where $T_p =$

A	B
_	b_1
_	b_2

- For any nonempty database D and for any tuple t in D, φ says that $t[B]$ must be both b_1 and b_2.

The satisfiability problem is to determine, given a schema $\mathcal{R}$ and a set Σ of constraints defined on $\mathcal{R}$, whether or not there exists a nonempty instance D of $\mathcal{R}$ that satisfies all constraints φ in Σ.

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:

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

In the same setting as the classical dependency theory

Recall domain specification in a schema:

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:

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.

However, in practice we have to consider a more general setting

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:

A	B
true	b_1
false	b_2

T_2:

B	A
b_1	false
b_2	true

If dom(A) is Boolean, then Σ 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:

A	B
true	b_1
false	b_2

T_2:

B	A
b_1	false
b_2	true

If dom(A) is Boolean, then Σ 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:

A	B
true	b_1
false	b_2

T_2:

B	A
b_1	false
b_2	true

If dom(A) is Boolean, then Σ 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

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

T_1=

A	B
true	b_1
false	b_2

T_2=

B	A
b_1	false
b_2	true

If dom(A) is Boolean, then Σ 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 Σ of constraints and a single constraint ϕ defined on $\mathcal{R}$, whether or not Σ implies ϕ, denoted by $\Sigma \models \phi$, i.e., whether for any instance D of $\mathcal{R}$ that satisfies Σ, D also satisfies ϕ.

To remove redundant data quality rules

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

- $\varphi = R(A \to C, T_p)$, $\phi_1 = R(A \to B, T_1)$, $\phi_2 = R(B \to C, T_2)$;
- $T_p =$

A	C
a	c

$T_1 =$

A	B
_	b

$T_2 =$

B	C
_	c

Implication: eliminating redundancies

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

To remove redundant data quality rules

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

- $\varphi = R(A \to C, T_p)$, $\phi_1 = R(A \to B, T_1)$, $\phi_2 = R(B \to C, T_2)$;
- $T_p =$

A	C
a	c

$T_1 =$

A	B
_	b

$T_2 =$

B	C
_	c

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:

$$\begin{aligned} \text{Reflexivity}: &\quad \text{If } Y \subseteq X, \text{ then } X \rightarrow Y \\ \text{Augmentation}: &\quad \text{If } X \rightarrow Y, \text{ then } XZ \rightarrow YZ \\ \text{Transitivity}: &\quad \text{If } X \rightarrow Y \text{ and } Y \rightarrow Z, \text{ then } X \rightarrow Z \end{aligned}$$

Sound and complete: $\Sigma \models \phi$ iff ϕ can be inferred from Σ 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 ϕ can be inferred from Σ 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 $t_p[Y] \preceq t'_p[Y]$ $\quad (a \preceq \_,\ a \preceq a,\ \_ \preceq \_)$,
- 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:

	satisfiability	implication	finite axiomatizability
CFD	$O(n^2)$	$O(n^2)$	yes
FD	$O(1)$	$O(n)$	yes

- General setting:

	satisfiability	implication	finite axiomatizability
CFD	NP-complete	coNP-complete	yes
FD	$O(1)$	$O(n)$	yes

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: int, phn: int, name: string, street: string, city: string, zip: string)

- A traditional FD on R_{UK}: zip $\rightarrow$ 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

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: int, phn: int, name: string, street: string, city: string, zip: string)

- A traditional FD on R_{UK}: zip $\rightarrow$ 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
- The FD is indeed propagated to the target, but as a CFD

([CC, zip] $\rightarrow$ [street], T_p)

CC	zip	street
44	_	_

Constraint propagation

- Input:
  - A set Σ of source constraints: FDs (or CFDs) on the sources
  - A view definition σ: a relational query
  - A view constraint φ
- Question: Is φ *propagated* from Σ via σ?

  For any source database D that satisfies Σ, the view $\sigma(D)$ is guaranteed to satisfy φ.

Constraint propagation

- Input:
  - A set Σ of source constraints: FDs (or CFDs) on the sources
  - A view definition σ: a relational query
  - A view constraint φ
- Question: Is φ *propagated* from Σ via σ?

  For any source database D that satisfies Σ, the view $\sigma(D)$ is guaranteed to satisfy φ.

Why bother?

- Data quality: no need to check zip $\longrightarrow$ 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 (CC $=$ 44, AC $=$ 20, city $=$ EDI, ...) can be rejected without checking the data

Propagation 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.

Propagation 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!

Propagation from source CFDs (FDs) to view CFDs

- 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 PTIME for SPCU views in the absence of attributes with a finite domain;*
- *in coNP for SPCU views, and is 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 PTIME for PC views, and is coNP-complete for SC views

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

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:

  ecfd_1: city $\notin \{\text{NYC}, \text{LI}\} \rightarrow$ AC
  ecfd_2: city $\in \{\text{NYC}\} \rightarrow$ AC $\in \{212, 718, 646, 347, 917\}$

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:

  ecfd_1: city $\notin \{\text{NYC}, \text{LI}\} \rightarrow$ AC
  ecfd_2: city $\in \{\text{NYC}\} \rightarrow$ AC $\in \{212, 718, 646, 347, 917\}$

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?

order(title, price) $\subseteq$ book(title, price),
order(title, price) $\subseteq$ CD(album, price).

These traditional INDs
do not make sense

order	title	type	price
t_4:	Snow White	CD	7.99
t_5:	Harry Potter	book	17.99

book	title	price	format
t_6:	Harry Potter	17.99	hard-cover
t_7:	Snow White	7.99	paper-cover

CD	album	price	genre
t_8:	J. Denver	7.94	country
t_9:	Snow White	7.99	a-book

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

  cind_3: (CD(album, price; genre = 'a-book') $\subseteq$ book(title, price; format = 'audio'))

- The example database does not satisfy cind_3

CD	album	price	genre
t_8:	J. Denver	7.94	country
t_9:	Snow White	7.99	a-book

book	title	price	format
t_6:	Harry Potter	17.99	hard-cover
t_7:	Snow White	7.99	paper-cover

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:

  (order(title, price; type = 'book') $\subseteq$ book(title, price))
  (CD(album, price; genre = 'a-book')
  $\subseteq$ book(title, price; format = 'audio'))

- CINDs:

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

T_1:

type	
book	

T_2:

genre	format
a-book	audio

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
  - $t[Y_p] = t_p[Y_p]$.

- 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 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 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)

Reasoning about conditional inclusion dependencies

For traditional INDs,

- any set of INDs is always satisfiable
- the implication problem is 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) PSPACE-complete in the absence of finite-domain attributes, and (2) EXPTIME-complete in the general setting.

Reasoning about conditional inclusion dependencies

For traditional INDs,

- any set of INDs is always satisfiable
- the implication problem is 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) PSPACE-complete in the absence of finite-domain attributes, and (2) 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

References

CFDs W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. TODS:33(2), June 2008.

CINDs L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In VLDB, 2007.

eCFDs L. Bravo, W. Fan, F. Geerts, and S. Ma. Increasing the expressivity of conditional functional dependencies without extra complexity. In *ICDE*, 2008.

- 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)
- W. Fan, S. Ma, Y. Hu, J. Liu, and Y. Wu. Propagating functional dependencies with conditions. In *VLDB*, 2008.

Other extensions of traditional dependencies:

- P. D. Bra and J. Paredaens. Conditional dependencies for horizontal decompositions. In *ICALP*, 1983.
- M. J. Maher. Constrained dependencies. *TCS* 173(1): 113-149, 1997.
- M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999. (denial constraints)

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
  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 (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 = [FN, LN, addr, tel, email], Y_b = [FN, SN, post, phn, 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 (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 = [FN, LN, addr, tel, email], Y_b = [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[LN, addr] and t'[SN, 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

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, addr] and t'[SN, 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, addr, FN] and t'[SN, post, FN] instead of t[FN, LN, addr, tel, email] and t'[FN, SN, post, phn, email].
- similarity $\approx_d$ in stead of equality on FN

Expressing matching rules as dependencies

if t[LN, addr] $=$ t'[SN, post] and t[FN] $\approx_d$ t'[FN], then $t[Y_c] \rightleftharpoons t'[Y_b]$

ϕ_1: card[LN] $=$ billing[SN] $\wedge$ card[addr] $=$ billing[post] $\wedge$
card[FN] $\approx_d$ billing[FN] $\rightarrow$ card$[Y_c] \rightleftharpoons$ billing$[Y_b]$

- Similarity metrics: edit distance, q-grams, Jaro distance, ...,
- Matching operation $t[Z] \rightleftharpoons t'[Z']$: 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[LN, addr] $= t'$[SN, post] and t[FN] $\approx_d t'$[FN], then $t[Y_c] \rightleftharpoons t'[Y_b]$

ϕ_1: card[LN] $=$ billing[SN] $\wedge$ card[addr] $=$ billing[post] $\wedge$
card[FN] $\approx_d$ billing[FN] $\rightarrow$ card$[Y_c] \rightleftharpoons$ billing$[Y_b]$

- Similarity metrics: edit distance, q-grams, Jaro distance, ...,
- Matching operation $t[Z] \rightleftharpoons t'[Z']$: 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 ϕ: for any instances $D = (D_c, D_b)$ and $D' = (D'_c, D'_b)$ of (card, billing), D and D' satisfy ϕ if
  - $D \sqsubseteq D'$: 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[LN, addr] $= t'$[SN, post] and (2) t[FN] $\approx_d t'$[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 ϕ defined on schemas (R_1, R_2):

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

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

ϕ_1: card[LN] = billing[SN] $\wedge$ card[addr] = billing[post] $\wedge$
card[FN] $\approx_d$ billing[FN] $\rightarrow$ card[Y_c] $\rightleftharpoons$ billing[Y_b]

ϕ_2: card[tel] = billing[phn] $\rightarrow$ card[addr] $\rightleftharpoons$ billing[post]

ϕ_3: card[email] = billing[email] $\rightarrow$ card[FN,LN] $\rightleftharpoons$ billing[FN,SN]

Matching dependencies (MDs)

- An MD ϕ defined on schemas (R_1, R_2):

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

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

ϕ_1: card[LN] = billing[SN] $\wedge$ card[addr] = billing[post] $\wedge$
card[FN] $\approx_d$ billing[FN] $\rightarrow$ card[Y_c] $\rightleftharpoons$ billing[Y_b]

ϕ_2: card[tel] = billing[phn] $\rightarrow$ card[addr] $\rightleftharpoons$ billing[post]

ϕ_3: card[email] = billing[email] $\rightarrow$ card[FN,LN] $\rightleftharpoons$ 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 ϕ 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 Σ of MDs:

ϕ_1: $\text{card}[\text{LN}] \rightleftharpoons \text{billing}[\text{SN}] \wedge \text{card}[\text{addr}] \rightleftharpoons \text{billing}[\text{post}] \wedge \text{card}[\text{FN}] \approx_d \text{billing}[\text{FN}] \rightarrow \text{card}[Y_c] \rightleftharpoons \text{billing}[Y_b]$

ϕ_2: $\text{card}[\text{tel}] = \text{billing}[\text{phn}] \rightarrow \text{card}[\text{addr}] \rightleftharpoons \text{billing}[\text{post}]$

ϕ_3: $\text{card}[\text{email}] = \text{billing}[\text{email}] \rightarrow \text{card}[\text{FN,LN}] \rightleftharpoons \text{billing}[\text{FN,SN}]$

we can derive MDs for identifying $\text{card}[Y_c]$ and $\text{billing}[Y_b]$:

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

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

Reasoning about MDs: Derived MDs can add value

From a given set Σ of MDs:

ϕ_1: $\text{card}[\text{LN}] \rightleftharpoons \text{billing}[\text{SN}] \wedge \text{card}[\text{addr}] \rightleftharpoons \text{billing}[\text{post}] \wedge \text{card}[\text{FN}] \approx_d \text{billing}[\text{FN}] \rightarrow \text{card}[Y_c] \rightleftharpoons \text{billing}[Y_b]$

ϕ_2: $\text{card}[\text{tel}] = \text{billing}[\text{phn}] \rightarrow \text{card}[\text{addr}] \rightleftharpoons \text{billing}[\text{post}]$

ϕ_3: $\text{card}[\text{email}] = \text{billing}[\text{email}] \rightarrow \text{card}[\text{FN,LN}] \rightleftharpoons \text{billing}[\text{FN,SN}]$

we can derive MDs for identifying $\text{card}[Y_c]$ and $\text{billing}[Y_b]$:

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

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

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.

Implication: Generic reasoning

A set Σ of MDs entails another MD ϕ, 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 $\Sigma = \{\phi_1, \phi_2, \phi_3\}$:

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

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

Implication: Generic reasoning

A set Σ of MDs entails another MD ϕ, 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 $\Sigma = \{\phi_1, \phi_2, \phi_3\}$:

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

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

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 $\Sigma \models_m \phi$, iff for any instances D, D', if $(D, D') \models \Sigma$, then $(D, D') \models \phi$

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

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

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

Generic axioms for similarity relations

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

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 Σ and ϕ, whether or not $\Sigma \models_m \phi$.

Theorem

The implication problem for matching dependencies is in PTIME.

The implication problem for matching dependencies

The implication problem for MDs: to determine, given any Σ and ϕ, 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 $\rightleftharpoons$;
  FDs: equality $=$ only
- MDs: dynamic semantics with $\rightleftharpoons$ (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 $\Sigma \models \varphi$

The implication problem for matching dependencies

The implication problem for MDs: to determine, given any Σ and ϕ, 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 $\rightleftharpoons$;
  FDs: equality $=$ only
- MDs: dynamic semantics with $\rightleftharpoons$ (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

  $u[X] = v[X] \asymp t_p[X]$ but $(u[Y] \neq v[Y]$ or $u[Y] = v[Y] \not\asymp t_p[Y])$.

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

Goal: SQL-based detection

Given a database D and a set Σ of CFDs, we want to detect all violations for Σ 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 φ has a constant right-hand side ($t_p[A] = a$):

  Q_φ^C `SELECT *` `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] \neq$ '_' & $t[A] \neq t_p[A]$
- This detects single 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 φ has a constant right-hand side ($t_p[A] = a$):

  Q_φ^C `SELECT *` `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] \neq$ '_' & $t[A] \neq t_p[A]$
- This detects single tuple violations
- In case that φ has a variable right-hand side ($t_p[A] =$'_'):

  Q_φ^V `SELECT DISTINCT` X `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] =$ '_'
  `GROUP BY` X `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 φ has a constant right-hand side ($t_p[A] = a$):

  Q_φ^C `SELECT *` `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] \neq$ '_' & $t[A] \neq t_p[A]$
- This detects single tuple violations
- In case that φ has a variable right-hand side ($t_p[A] =$'_'):

  Q_φ^V `SELECT DISTINCT` X `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] =$ '_'
  `GROUP BY` X `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 φ has a constant right-hand side ($t_p[A] = a$):

  Q_φ^C `SELECT *` `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] \neq$ '_' & $t[A] \neq t_p[A]$
- This detects single tuple violations
- In case that φ has a variable right-hand side ($t_p[A] =$'_'):

  Q_φ^V `SELECT DISTINCT` X `FROM` R t, T_p t_p
  `WHERE` $t[X] \asymp t_p[X]$ & $t_p[A] =$ '_'
  `GROUP BY` X `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

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs
- Naive approach:

Detecting CFD violations in SQL: multiple CFDs

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

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs
- Naive approach:
  - Treat each CFD in Σ separately and use single CFD queries.
    $\Rightarrow$ As many queries needed as there are CFDs

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs
- Naive approach:
  - Treat each CFD in Σ separately and use single CFD queries.
    $\Rightarrow$ As many queries needed as there are CFDs
- Merge approach:

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs
- Naive approach:
  - Treat each CFD in Σ 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.

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs
- Naive approach:
  - Treat each CFD in Σ 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 Σ consisting of multiple CFDs
- Naive approach:
  - Treat each CFD in Σ 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.

Detecting CFD violations in SQL: multiple CFDs

- We next consider a set Σ consisting of multiple CFDs
- Naive approach:
  - Treat each CFD in Σ 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.

Experiments show that the merge approach is very efficient.

Detecting CFD violations in SQL: multiple CFDs

Step 1: Merge pattern tableaux

- Consider $\varphi_1 = \mathit{Cust}([\mathit{CC}, \mathit{AC}, \mathit{phn}] \rightarrow [\mathit{street}, \mathit{city}, \mathit{zip}], T_p)$ with

$T_p =$

CC	AC	phn	street	city	zip
_	_	_	_	_	_
01	908	_	_	MH	_

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

$T'_p =$

CC	AC	city
_	_	_
01	215	PHI
44	141	GLA

- Then the result of merging the tableaux φ_1 and φ_2 is:

$T_p^L =$

id	CC	AC	phn
1	_	_	_
2	01	908	_
3	_	_	@
4	01	215	@
5	44	141	@

and $T_p^R =$

id	street	city	zip
1	_	_	_
2	_	MH	_
3	@	_	@
4	@	PHI	@
5	@	GLA	@

Detecting CFD violations in SQL: multiple CFDs

Step 1: Merge pattern tableaux (cont'd)

- Then the result of merging the tableaux φ_1 and φ_2 is:

$T_p^L =$

id	*CC*	*AC*	*phn*
1	_	_	_
2	01	908	_
3	_	_	@
4	01	215	@
5	44	141	@

and $T_p^R =$

id	*street*	*city*	*zip*
1	_	_	_
2	_	*MH*	_
3	@	_	@
4	@	*PHI*	@
5	@	*GLA*	@

- Merging an additional CFD $\varphi_3 = \mathit{Cust}([\mathit{city}] \rightarrow [\mathit{zip}], T_p'')$ with $T_p'' =$

city	*zip*
_	_

results in

$T_p^L =$

id	*CC*	*AC*	*phn*	*city*
1	_	_	_	@
2	01	908	_	@
3	_	_	@	@
4	01	215	@	@
5	44	141	@	@
6	@	@	@	_

and $T_p^R =$

id	*street*	*city*	*zip*
1	_	_	_
2	_	*MH*	_
3	@	_	@
4	@	*PHI*	@
5	@	*GLA*	@
6	@	@	@_

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:

- Q_Σ^C `SELECT * FROM` $R\ t,\ T_p^L\ t_p^L,\ T_p^R\ t_p^R$
  `WHERE` $t_p^L[\mathrm{id}] = t_p^R[\mathrm{id}]\ \&\ t[Z] \asymp t_p^L[Z]\ \&\ t[W] \not\asymp t_p^R[W]$

Step 3: Multiple tuple violations:

- Q_Σ^V `SELECT DISTINCT` Z `FROM` Macro t^M
  `GROUP BY` Z `HAVING COUNT(DISTINCT` W`)>` 1

  where Macro is:

  `SELECT` (`CASE` $t_p^L[B_i]$ `WHEN '@' THEN '@' ELSE` $t[B_i]$ `END) AS` $B_i \cdots$
  (`CASE` $t_p^R[C_j]$ `WHEN '@' THEN '@' ELSE` $t[C_j]$ `END) AS` $C_j \cdots$
  `FROM` $R\ t, T_p^L\ t_p^L, T_p^R\ t_p^R$
  `WHERE` $t_p^L[\mathrm{id}] = t_p^R[\mathrm{id}]\ \&\ t[Z] \asymp t_p^L[Z]\ \&$
  $(t_p^R[C_1] =$ '_' `OR` $\cdots$ `OR` $t_p^R[C_n] =$ '_' $)$

Works efficient in practice and can be generalized to extended CFDs.

Multiple tuple violation query: Example

- Recall merged tableaux:

$T_p^L =$

id	*CC*	*AC*	*phn*	*city*
1	_	_	_	@
2	01	908	_	@
3	_	_	@	@
4	01	215	@	@
5	44	141	@	@
6	@	@	@	_

and $T_p^R =$

id	*street*	*city*	*zip*
1	_	_	_
2	_	*MH*	_
3	@	_	@
4	@	*PHI*	@
5	@	*GLA*	@
6	@	@	@_

- Database,

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	John	Oxford Ter.	EDI	EH4 8LE

Multiple tuple violation query: Example

- Recall merged tableaux:

$T_p^L =$

id	*CC*	*AC*	*phn*	*city*
1	_	_	_	@
2	01	908	_	@
3	_	_	@	@
4	01	215	@	@
5	44	141	@	@
6	@	@	@	_

and $T_p^R =$

id	*street*	*city*	*zip*
1	_	_	_
2	_	*MH*	_
3	@	_	@
4	@	*PHI*	@
5	@	*GLA*	@
6	@	@	@_

- Database, Macro relation,

CC	AC	phn	city	street	city	zip
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

Multiple tuple violation query: Example

- Recall merged tableaux:

$T_p^L =$

id	*CC*	*AC*	*phn*	*city*
1	_	_	_	@
2	01	908	_	@
3	_	_	@	@
4	01	215	@	@
5	44	141	@	@
6	@	@	@	_

and $T_p^R =$

id	*street*	*city*	*zip*
1	_	_	_
2	_	*MH*	_
3	@	_	@
4	@	*PHI*	@
5	@	*GLA*	@
6	@	@	@_

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

CC	AC	phn	city	street	city	zip
44	131	1234567	@	Mayfield	EDI	EH4 8LE
44	131	3456789	@	Crichton	NYC	EH4 8LE
44	131	5678910	@	Oxford Tre.	EDI	EH4 8LE
44	131	@	@	@	EDI	@
44	131	@	@	@	NYC	@
@	@	@	EDI	@	@	EH4 8LE
@	@	@	NYC	@	@	EH4 8LE

Incremental detection algorithm

Goal:

Avoid detection of violations from scratch after updates are issued.

- Key idea: materialize auxiliary relation Aux(D)
- Aux(D) stores condensed representation of violations in current databases.
- An incremental SQL-based detection method can be devised that leverages 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.

References

CFDs W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. TODS:33(2), June 2008.

eCFDs L. Bravo, W. Fan, F. Geerts, and S. Ma. Increasing the expressivity of conditional functional dependencies without extra complexity. In *ICDE*, 2008.

General A. Chandel, N. Koudas, K. Pu, D. Srivastava: Fast Identification of Relational Constraint Violations. In *ICDE*, 2007. (Uses indices based on Boolean Decision Diagrams).

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
  - 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 Σ of constraints.
- Output: a repair D' of D such that D' satisfies Σ.

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

Repair checking problem

- Input: two relational databases D, D' and a set Σ of constraints.
- Output: "yes" if D' is a repair of D; "no" otherwise.

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.,

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

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.,

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

  - 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.,

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

  - 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.,

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

  - t' is the tuple in D' corresponding to t.
  - $w(t, A)$ denotes the accuracy of the attribute A.
  - $\mathit{dist}(\ ,\ )$ measures the distance between values.

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.,

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

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

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

U-repair

- Input: a relational database D and a set Σ of constraints
- Output: a repair D' of D such that cost(D', D) is minimal and $D' \models \Sigma$.

- 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

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	NYC	07974

- An FD-violation can always be resolved by performing a chase on the database, i.e., by "merging equivalence classes":

Resolving FD-violations

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	NYC	07974

- 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

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	NYC	07974

- 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 $\text{fd}_1 : R([zip] \longrightarrow [city])$, (t_1, zip),(t_2, zip) and (t_3, zip) are merged, target value is selected based on cost function: e.g., EDI

Resolving FD-violations

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	NYC	07974

- 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 $\text{fd}_1 : R([zip] \longrightarrow [city])$, (t_1, zip),(t_2, zip) and (t_3, zip) are merged, target value is selected based on cost function: e.g., EDI
- For $\text{fd}_2 : R([name, street, zip] \longrightarrow [phn])$, (t_2, phn),(t_3, phn) are merged, target value is selected based on cost function, e.g., 3456789

Resolving FD-violations

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2':	44	131	3456789	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	NYC	07974

- 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 $\text{fd}_1 : R([zip] \longrightarrow [city])$, (t_1, zip),(t_2, zip) and (t_3, zip) are merged, target value is selected based on cost function: e.g., EDI
- For $\text{fd}_2 : R([name, street, zip] \longrightarrow [phn])$, (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.

Finding a U-repair for FDs: Greedy approach

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?

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

Finding a repair for CFDs

Can the same approach still be applied for CFDs?

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

- Start with initial equivalence classes

Finding a repair for CFDs

Can the same approach still be applied for CFDs?

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

- Start with initial equivalence classes
- For 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?

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

- Start with initial equivalence classes
- For cfd_1 merge (t_1, B) and (t_2, B). Target value is either b_1 or b_2.
- For cfd_2 equivalence class $\{(t_1, B), (t_2, B)\}$ should be split! But then cfd_1 is violated again!

Finding a repair for CFDs

Can the same approach still be applied for CFDs?

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

- Start with initial equivalence classes
- For cfd_1 merge (t_1, B) and (t_2, B). Target value is either b_1 or b_2.
- For cfd_2 equivalence class $\{(t_1, B), (t_2, B)\}$ should be split! But then cfd_1 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 .

$\text{cfd}_1 : R(A \rightarrow B, (\_ \| \_))$

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

First cfd_1: equivalence class is locked; Target value b_1

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 .

$\text{cfd}_1 : R(A \to B, (\_ \| \_))$

$\text{cfd}_2 : R(C \to B, \{(c_1 \| b_1), (c_2 \| b_2)\})$

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_1

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 .

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

t_1':

A	B	C
a	b_1	c_1

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 .

$\text{cfd}_1 : R(A \rightarrow B, (\_ \| \_))$

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

	A	B	C
t_1:	a	b_1	c_1
t_2:	a	b_2	c_2

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;

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 .

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

	A	B	C
t_1':	null	b_1	c_1
t_2':	null	b_2	c_2

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 Σ of CFDs in PTIME.

CFD-repair: Example

Pattern tuples indeed help improving the quality of repairs:

- Consider the CFD

  cfd_3: ([CC = 01, AC = 908, phn] → [street, city = 'MH', zip]).

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	NYC	07974

CFD-repair: Example

Pattern tuples indeed help improving the quality of repairs:

- Consider the CFD

  cfd_3: ([CC = 01, AC = 908, phn] → [street, city = 'MH', zip]).
- Forces target value of (t_4,city) to be 'MH'.

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	44	131	5678910	Rick	Crichton	EDI	EH4 8LE
t_4:	01	908	3456789	Joe	Mtn Ave	MH	07974

CFD-repair: Example

Pattern tuples indeed help improving the quality of repairs:

- Consider the CFD

  cfd_3: ([CC = 01, AC = 908, phn] → [street, city = 'MH', zip]).
- Forces target value of (t_4,city) to be 'MH'.

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	EDI	EH4 8LE
t'_2:	44	131	3456789	Rick	Crichton	EDI	EH4 8LE
t'_4:	01	908	3456789	Joe	Mtn Ave	MH	07974

Two related repairing problems

Incremental data repairing

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

Local repairing problem

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

Theorem

The incremental and local repair problems are both NP-complete.

Incremental data repairing

- 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:

M. Arenas, L. E. Bertossi, J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, 1999.

D-repairs J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. Inf. Comput. 197(1-2):90- 121, 2005

S-repairs S. Staworko. Declarative inconsistency handling in relational and semi-structured databases. PhD thesis, 2007.

Repairing algorithms:

Census I. Fellegi, D. Holt, A Systematic Approach to Automatic Edit and Imputation, Journal of the American Statistical association, 1976.

FDs/INDs P. Bohannon, M. Flaster, W. Fan, R. Rastogi: A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *SIGMOD*, 2005.

CFDs G. Cong, W. Fan, F. Geerts, X. Jia, S. Ma: Improving Data Quality: Consistency and Accuracy. In *VLDB*, 2007.

Consistent query answering:

L. Bertossi. Consistent query answering in databases. SIGMOD Rec. 35(2): 68-76, 2006.

J. Chomicki. Consistent query answering: Five easy pieces. In ICDT, 2007.

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, Datactics ...

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 Σ of CFDs defined on D, and
  - a set Γ 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

NI#	AC	phn	name	street	city	zip
			...			
SC1234566	131	1234567	M. Smith	Mayfield	EDI	EH4 8LE
SC1234566	020	1234567	M. Smith	Portland	LDN	W1B 1JL
			...			

- 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?

R. van der Meyden. Logical approaches to incomplete information: A survey. In J. Chomicki and G. Saake (eds.): *Logics for Databases and Information Systems*: 307-356, 1998.

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