A Revival of Integrity Constraints for Data Cleaning

Wenfei Fan\textsuperscript{1,2,1,*} Floris Geerts\textsuperscript{1,†} Xibei Jia\textsuperscript{1,†}
\textsuperscript{1}University of Edinburgh \textsuperscript{2}Bell Laboratories
wenfei@inf.ed.ac.uk research.bell-labs.com fgeerts@inf.ed.ac.uk xibei.jia@ed.ac.uk

Abstract
Integrity constraints, a.k.a. data dependencies, are being widely used for improving the quality of schema. Recently constraints have enjoyed a revival for improving the quality of data. The tutorial aims to provide an overview of recent advances in constraint-based data cleaning.

1. An Overview
Real world data is often dirty: inconsistent, inaccurate, incomplete and/or stale. Recent statistics reveal that enterprises typically expect data error rates of approximately 1%–5%, some above 30%. It is reported that dirty data costs US businesses 600 billion dollar annually, and that erroneously priced data in retail databases alone costs US consumers $2.5 billion each year. It is also estimated that data cleaning accounts for 30%–80% of the development time and budget in most data warehouse projects. While the prevalent use of the Web has made it possible to extract and integrate data from diverse sources, it has also increased the risks, on an unprecedented scale, of creating and propagating dirty data.

In light of these, there has been increasing demand for data quality tools, for effectively detecting and repairing errors in the data. To this end, integrity constraints yield a principled approach to improving data quality. Integrity constraints are almost as old as relational databases themselves. A variety of constraint formalisms have been proposed \cite{7}, and have been being widely used to improve the quality of schema. Recently constraints have enjoyed a revival, for improving the quality of data.

We provide an overview of recent advances in constraint-based data cleaning. We argue that classical constraints often need to be revised or extended in order to capture more errors in real-life data, and to match, repair and query inconsistent data. The tutorial draws materials from over 80 references; only the most relevant ones are included here.

2. Improving Data Quality: An Overview
Data quality has been studied in distinct areas: in statistics since the 1960s, in management since the 1980s, and in computer science with renewed interests since the 1990s \cite{2}.

It is often measured in terms of consistency, accuracy, completeness, and timeliness, etc. We present a brief overview of data quality issues. A complete survey of data quality is beyond the scope of this tutorial (see \cite{2} for a survey).

Research activities. Research on data quality has been mostly focusing on (a) error correction, a.k.a. data imputation, (b) object identification, a.k.a. record linkage, merge-purge, data deduplication and record matching, and (c) profiling, to discover meta-data from sample data. There is also an intimate connection between data quality and data integration, data standardization, data acquisition, cost estimation, schema evolution, and even schema matching.

A variety of approaches have been put forward to tackle these problems: probabilistic, empirical, rule-based, and logic-based. There have also been a number of commercial tools, most notably ETL tools (extraction, transformation, loading), as well as research prototype systems, e.g., Ajax, Potter’s Wheel, Artkos, and tools from Telcordia.

Constraint-based data cleaning. These methods follow the logic-based approach, by specifying the semantics of data in terms of integrity constraints. Constraint-based methods, declarative in nature, have shown promise as a systematic method for reasoning about the semantics of the data, and for deducing and discovering rules for cleaning the data and identifying objects, among other things.

Constraint-based data cleaning has mostly focused on two topics, introduced in \cite{1}: repairing is to find another database that is consistent and minimally differs from the original database; and consistent query answering is to find an answer to a given query in every repair of the original database, without editing the data. Constraints have also been recently studied for object identification \cite{10}.

In this tutorial we present two extensions of traditional dependencies, for data repairing (Section 3) and object identification (Section 4), respectively. We refer the reader to \cite{5} for recent survey on consistent query answering.

3. Adding Conditions to Constraints
Constraints adopted for detecting inconsistencies are mostly traditional dependencies such as functional dependencies (FDs) and inclusion dependencies (INDs). These constraints are required to hold on entire relation(s) and often fail to capture errors commonly found in real-life data. We circumvent these limitations by considering extensions of FDs and INDs, referred to as conditional functional dependencies (CFDs) and conditional inclusion dependencies (CINDs), respectively, by additionally specifying patterns of semantically related values; these patterns impose conditions on what part of the relation(s) the dependencies are to hold and which combinations of values should occur together.

Conditional functional dependencies \cite{8}. An example CFD is customer(\{CC = 44, zip\} → \{street\}), which asserts
that for customers in the UK (cc = 44), zip code determines street. It is an “FD” that is to hold on the subset of tuples that satisfies the pattern cc = 44. It is not a traditional FD since it is defined with constants and is not required to hold on the entire data set. Another example CFD is customer(cc = 41, ac = 908, phn) → [street, city = ‘MH’, zip]. It asserts that for any two US customers, if they have area code 908 and have the same phn, then they must have the same street and zip, and moreover, city must be MH. From the above it is clear that CFDs, when used to specify the consistency of data, i.e., to characterize errors as violations of these dependencies, are able to capture more inconsistencies than their traditional FD counterparts.

Conditional inclusion dependencies [4]. Consider two relations, book and CD, specifying customer orders of books and CDs, respectively. For schema matching or data cleaning, one might want to specify inclusion relationships from, e.g., CDs to book. However, while indeed there exist inclusion dependencies, they only hold under certain conditions, i.e., they are in the form of CINDs. An example CIND is book(title, price, format = ‘A-BOOK’) → book(title, price, format = ‘AUDIO’), which asserts that for each CD tuple t, if its genre is ‘A-BOOK’ (audio book), then there must be a book tuple t’ such that title and price of t’ agree with album and price of t; and moreover, the format of t’ must be ‘AUDIO’. Like CFDs, these constraints are required to hold on a subset of tuples satisfying certain patterns. They are specified with data values and cannot be expressed as standard CINDs.

4. Extending Constraints with Similarity

Essential to data cleaning and data integration is object identification: it is often necessary to identify tuples from one or more relations that refer to the same real-world object. This is nontrivial since the data sources may not be error free or may represent the same object differently.

A key issue for object identification concerns how to determine matching keys and rules [2], i.e., what attributes should be selected and how they should be compared in order to identify tuples. Constraints can help here in automatically deriving matching keys and rules, and thus improve match quality and increase the degree of automation.

Consider two data sources: card(c#, ssn, fn, ln, addr, phn, email, type), and billing(c#, fn, ln, addr, phn, email, item, price). Here a card tuple specifies a credit card (c# and type) issued to a person identified by ssn, fn, ln, addr, phn and email. A billing tuple indicates that a transaction of a card number c#, issued to a holder (fn, sn, addr, phn and email). Given an instance (Dx, Dy) of (card, billing), for fraud detection, one has to ensure that for any tuple t ∈ Dx and t’ ∈ Dy, if t[c#] = t’[c#], then t[Y] and t’[Y] refer to the same holder, where Y = [fn, ln, addr, phn, email].

Consider the following matching rules. (a) If {phn} and t[phn] match, then t[addr] and t’[addr] should refer to the same address (even if t[addr] and t’[addr] might be radically different). (b) If {email} and t[emial] match, then t[fn, ln] and t’[fn, ln] match. (c) If {ln, addr} and t[ln, addr] are identical and t[fn] and t’[fn] are similar w.r.t. a similarity operator =, then t[Y] and t’[Y] match. Then, from these one can deduce the following, referred to relative candidate keys (rck), as an extension of relational keys.

rck1: {email, addr} → {fn, ln, phn}
rck2: {ln, phn, fn}, {ln, phn, fn} [1] = = similarity */

Here rck1 asserts that if t[ln, phn] and t’[ln, phn] are identical and if t[fn] and t’[fn] are similar, then t[Y] and t’[Y] match; similarly for rck1. Hence instead of comparing the entire Y list, one can inspect the attributes in rck1 or rck2: if t and t’ match on rck1 or rck2, then so do t[Y] and t’[Y].

The derived rckks improve match quality: when t and t’ differ in some pairs of attributes, e.g., (addr, addr), they can still be matched via other attributes, e.g., (ln, phn, fn, ln, phn, fn). In other words, true matches may be identified by derived rckks, even when they cannot be found by the given matching rules from which the rckks are derived.

In contrast to traditional candidate keys, rckks are defined in terms of both equality and similarity; further, they are defined across multiple relations, rather than on a single relation. Moreover, to cope with unreliable data, rckks adopt a semantics very different from its traditional counterpart.

5. Constraints in Data Cleaning Tools

While constraints should logically become an essential part of data-cleaning tools, we are not aware of any commercial tools with this functionality. To show that constraints can be effectively used in data cleaning, we present SEMANDAQ [9], a research prototype system for data repairing. SEMANDAQ supports (a) specifications of CFDs, (b) automatic detections of CFD violations, based on efficient SQL-based techniques [8, 4], and (c) repairing, i.e., given a set of CFDs and a dirty database, it finds a candidate repair that minimally differs from the original data and satisfies the CFDs [6]. We show how the user can inspect and modify this repair, and how these manual changes affect the repair.

6. Open Problems and Emerging Applications

The study of constraint-based data cleaning has raised as many questions as it has answered. A number of open questions have to be settled. (a) The interaction between error correction and object identification. (b) Database repairs in master data management. (c) Scalable repairing algorithms with performance guarantee (precision and recall). (d) Incremental repairing methods. (e) XML data cleaning. (f) Integration of constraint-based methods with probabilistic databases and incomplete information management. (g) Managing dirty sources in data exchange and integration.

7. References