A revival of integrity constraints for data cleaning

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
Proceedings of the VLDB Endowment (PVLDB)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
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 [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 [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 [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 [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 [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 [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 [8]. An example CFD is customer([cc = 44, zip] → [street]), which asserts

---

A Revival of Integrity Constraints for Data Cleaning

Wenfei Fan1,2,†† Xibei Jia1,††
1University of Edinburgh
wenfei@inf.ed.ac.uk, research.bell-labs.com
2Bell Laboratories
fgeerts@inf.ed.ac.uk xibei.jia@ed.ac.uk

Adjunct professor, Harbin Institute of Technologies.
† Supported in part by EPSRC EP/E029213.
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([fn = 01, ac = 908, phn] \rightarrow [street, city \in '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 (CD(album, price, genre = 'A-BOOK') \rightarrow (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 holder (fn, ln, addr, phn and email). Given an instance (Dc, Db) of (card, billing), for fraud detection, one has to ensure that for any tuple t ∈ Dc and t′ ∈ Db, 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 [t[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 [t[emai] and t′[email] match, then [t[fn, ln] and t′[fn, ln] match. (c) If [t[ln, addr] and t′[ln, addr] are identical and t[FN] and t′[FN] are similar w.r.t. a similarity operator s, 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.

cfk1: ([email, addr], [email, addr] \implies [ln, phn])
cfk2: ([ln, phn, fn], [ln, phn, fn] \implies [fn, fn] s: similarity */ Here rck1 asserts that if [t[ln, phn] and t′[ln, phn] are identi-