BigDansing

A System for Big Data Cleansing
Table of Content

- Introduction
- Data Cleansing Process
- Architecture of BigDansing
- Experiment
- Discussion
“Inaccurate data has a direct impact ... the average company losing 12% of its revenue” — Ben Davis (Econsultancy)

“More than 25 Percent of Critical Data Used in Large Corporations is Flawed” — Gartner Inc.

“This is the digital universe. It is growing 40% a year into the next decade” — EMC2
**Data Cleansing**

Data cleansing, data cleaning, or data scrubbing is the process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database.

**Data Quality**

Data are of high quality “if they are fit for their intended uses in operations, decision making and planning". 
An Intuitive Example

**Quality Rule**

Two customers having the same zipcode cannot be in different cities

D(zipcode -> city)

---

**Dirty Dataset**

<table>
<thead>
<tr>
<th>Name</th>
<th>Zipcode</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Winnie</td>
<td>91340</td>
<td>San Francisco</td>
</tr>
<tr>
<td>Robbert</td>
<td>91340</td>
<td>New York</td>
</tr>
<tr>
<td>Emma</td>
<td>91340</td>
<td>San Francisco</td>
</tr>
</tbody>
</table>
## Quality Rule

Two customers having the same zipcode cannot be in different cities

\[ D(\text{zipcode} \rightarrow \text{city}) \]

## Dirty Dataset

<table>
<thead>
<tr>
<th>Name</th>
<th>Zipcode</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Winnie</td>
<td>91340</td>
<td>San Francisco</td>
</tr>
<tr>
<td>Robbert</td>
<td>91340</td>
<td>New York</td>
</tr>
<tr>
<td>Emma</td>
<td>91340</td>
<td>San Francisco</td>
</tr>
</tbody>
</table>
Challenges
In the context of big data

- **Scalability**
  Can not scale to large datasets

- **Abstraction**
  Need to understand both the quality rules and the distributed platform
Challenges
In the context of big data

- **Scalability**
  Can not scale to large datasets

- **Abstraction**
  Need to understand both the quality rules and the distributed platform
A Big Data Cleansing system to tackle efficiency, scalability, and ease-of-use issues in data cleansing
BigDansing
To address the challenges

- **Generic abstraction**
  - define customized rules together with traditional rules in a simple way
  - the underlying distributed platform is transparent

- **Fast and scalable** detection, repair and updates
  - 1.9B rows → 13B violations < 3 hours on 16 small machines
Data Cleansing Process

Data Cleansing Flowchart

Quality Rules

Dirty Data

Detect

Repair

Update

Clean Data

Source: https://www.slideshare.net/Zuhairkhayyat/bigdansing-presentation-slides-for-kaust
Architecture

Input:
- quality rules
- a dirty dataset
Architecture

RuleEngine:

- logical layer
- physical layer
- execution layer
general purpose distributed platform like Hadoop or Spark
Architecture

Repair Algorithm: run existing centralized algorithm as a black box in a distributed way.
Architecture

Output: a clean dataset
Let’s dig into the components one by one...
Architecture

Input:
- quality rules
- a dirty dataset
Dirty data is detected by

- **Declarative Rule**
  - Functional dependencies (FD)
    - e.g. Zipcode $\rightarrow$ City
  - Conditional functional dependency (CFD)
    - e.g. Country = 'Saudi Arabia', Zipcode $\rightarrow$ City
  - Denial constraints (DC)
    - e.g. $\forall t_1, t_2 \in D, \neg(t_1.Salary > t_2.Salary \land t_1.Rate < t_2.Rate)$

- **User Defined Function (UDF)**
  Duplicates, statistical errors
Abstraction
UDF-based Approach
Quality rules are represented by 5 functions: Scope, Block, Iterate, Detect, and GenFix.

- **declarative rules**
  - automatically be translated into logical operators

- **UDFs**
  - can be implemented using logical operators
**Logic Operators**

**Scope**

**Scope**

Input: data units

Output: data units

Example: Zipcode → City

<table>
<thead>
<tr>
<th>Name</th>
<th>Zipcode</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Annie</td>
<td>60601 NY</td>
</tr>
<tr>
<td>t2</td>
<td>Laure</td>
<td>90210 LA</td>
</tr>
<tr>
<td>t3</td>
<td>John</td>
<td>60601 CH</td>
</tr>
<tr>
<td>t4</td>
<td>Mark</td>
<td>90210 SF</td>
</tr>
<tr>
<td>t5</td>
<td>Robert</td>
<td>60601 CH</td>
</tr>
<tr>
<td>t6</td>
<td>Mary</td>
<td>90210 LA</td>
</tr>
</tbody>
</table>

Input: t1 – t6

Output:

t1 – t6 (Zipcode, City)
### Logic Operators

**Block**

Input: data units

Output: grouping key

Example: Zipcode → City

Input: t1 – t6

Output:

\[
\langle 60601, (t1,t3,t5)\rangle, \\
\langle 90210, (t2,t4,t6)\rangle
\]
**Iterate**

Input: a group of data units
Output: single tuple, tuple pair

Example: Zipcode $\rightarrow$ City

<table>
<thead>
<tr>
<th>Zipcode</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>60601 NY</td>
</tr>
<tr>
<td>t3</td>
<td>60601 CH</td>
</tr>
<tr>
<td>t5</td>
<td>60601 CH</td>
</tr>
</tbody>
</table>

Input: $<60601, (t1,t3,t5)>$
Output:

$<t1,t3>, <t1,t5>, <t3,t5>$
Detect

Input: data units
Output: Violation(s)

Example: Zipcode → City
Input: <t1,t3>, <t1,t5>, <t3,t5>
Output:
<t1.CITY ≠ t3.CITY>,
<t1.CITY ≠ t5.CITY>

<table>
<thead>
<tr>
<th>Zipcode</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>60601 NY</td>
</tr>
<tr>
<td>t3</td>
<td>60601 CH</td>
</tr>
<tr>
<td>t5</td>
<td>60601 CH</td>
</tr>
</tbody>
</table>
GenFix

Input: Violation
Output: possible fix(es)

Example: Zipcode → City

Input: <t1.CITY ≠ t3.CITY>,
< t1.CITY ≠ t5.CITY>

Output:
<t1.CITY = t3.CITY>,
<t1.CITY = t5.CITY>
Rule Engine

- three-layer
- on top of general purpose data processing framework such as MapReduce-like with execution abstraction
Rule Engine
Logical Plan

- Define the data unit flow

- Validating the plan:
  At least one input dataset
  For UDF: at least one detect
  For Rules: at least one rule

- Support simple and bushy plans
Physical operators are system specific
MPI, Hadoop, Spark

- Each physical operator is an independent execution unit
- Each logical operator $\rightarrow$ one physical operator

- BigDansing consolidate logical plans to improve I/O
- More physical operators can be added with different optimizations to improve logical plans
Run on the parallel general purpose data processing framework such as MapReduce-like

User don't need to care about how the distributed computation performs
Repair Algorithm

- equivalence class
- hypergraph-based
Implement two existing **serial** repair algorithms to run in distributed mode

- Equivalence class algorithm
- Hypergraph algorithm

Design a **distributed** version of the seminal equivalence class algorithm
Equivalence Class Algorithm

- Fix errors based on (=,≠)

- Based on heuristics:
  - Partition the possible fixes into different groups
  - Assign the highest frequency value to group

- Example:
  - Group 1: Zipcode = 60601
    - Highest frequency = CH
  - Group 2: Zipcode = 90210
    - Highest frequency = LA

<table>
<thead>
<tr>
<th>Name</th>
<th>Zipcode</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Annie</td>
<td>60601</td>
</tr>
<tr>
<td>t2</td>
<td>Laure</td>
<td>90210</td>
</tr>
<tr>
<td>t3</td>
<td>John</td>
<td>60601</td>
</tr>
<tr>
<td>t4</td>
<td>Mark</td>
<td>90210</td>
</tr>
<tr>
<td>t5</td>
<td>Robert</td>
<td>60601</td>
</tr>
<tr>
<td>t6</td>
<td>Mary</td>
<td>90210</td>
</tr>
<tr>
<td>t7</td>
<td>Jon</td>
<td>60601</td>
</tr>
</tbody>
</table>
Hypergraph Algorithm

- Fix errors based on (≤, =, ≥)
- Based on linear optimization and greedy MVC:
  Select hyper-graph node with highest edges
  Change its value depending on edge conditions

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annie</td>
<td>24000</td>
<td>15</td>
</tr>
<tr>
<td>Laure</td>
<td>25000</td>
<td>10</td>
</tr>
<tr>
<td>John</td>
<td>40000</td>
<td>25</td>
</tr>
<tr>
<td>Mark</td>
<td>88000</td>
<td>24</td>
</tr>
<tr>
<td>Robert</td>
<td>15000</td>
<td>15</td>
</tr>
<tr>
<td>Mary</td>
<td>81000</td>
<td>28</td>
</tr>
<tr>
<td>Jon</td>
<td>40000</td>
<td>25</td>
</tr>
</tbody>
</table>
More intuitively, TPCH datasets with 959M, 1271M, 1583M, and 1907M rows are of sizes **150GB, 200GB, 250GB**, and **300GB** respectively.
Quality Rules

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi_1$ (FD):</td>
<td>Zipcode $\rightarrow$ City</td>
</tr>
<tr>
<td>$\varphi_2$ (DC):</td>
<td>$\forall t_1, t_2 \in TaxB, \neg (t_1\text{.Salary} &gt; t_2\text{.Salary} \land t_1\text{.Rate} &lt; t_2\text{.Rate})$</td>
</tr>
<tr>
<td>$\varphi_3$ (FD):</td>
<td>o_custkey $\rightarrow$ c_address</td>
</tr>
<tr>
<td>$\varphi_4$ (UDF):</td>
<td>Two rows in Customer are duplicates</td>
</tr>
<tr>
<td>$\varphi_5$ (UDF):</td>
<td>Two rows in NCVoter are duplicates</td>
</tr>
<tr>
<td>$\varphi_6$ (FD):</td>
<td>Zipcode $\rightarrow$ State</td>
</tr>
<tr>
<td>$\varphi_7$ (FD):</td>
<td>PhoneNumber $\rightarrow$ Zipcode</td>
</tr>
<tr>
<td>$\varphi_8$ (FD):</td>
<td>ProviderID $\rightarrow$ City, PhoneNumber</td>
</tr>
</tbody>
</table>
Single-node experiments with $\Phi_1$ (Functional Dependencies)

TaxA dataset:
FD: Zipcode $\rightarrow$ City
FD: Zipcode $\rightarrow$ State

provides a finer granular abstraction allowing users to specify rules more efficiently
Multi-nodes experiments with $\Phi_1$(Functional Dependencies)

TPCH dataset:
FD: custkey $\rightarrow$ custAddress
16 Workers
Results

Scalability

TPCH Dataset:
FD: custkey → custAddress
Dataset: 500M rows
Repair quality

<table>
<thead>
<tr>
<th>Rule(s)</th>
<th>NADEEF</th>
<th>BigDansing</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>precision</td>
<td>recall</td>
<td>precision</td>
</tr>
<tr>
<td>$\varphi_6$</td>
<td>0.713</td>
<td>0.776</td>
<td>0.714</td>
</tr>
<tr>
<td>$\varphi_6&amp;\varphi_7$</td>
<td>0.861</td>
<td>0.875</td>
<td>0.861</td>
</tr>
<tr>
<td>$\varphi_6-\varphi_8$</td>
<td>0.923</td>
<td>0.928</td>
<td>0.924</td>
</tr>
</tbody>
</table>

| $\phi_D$ | $|R,G|/e$ | $|R,G|$ | $|R,G|/e$ | $|R,G|$ | Iter. |
|----------|----------|--------|----------|--------|-------|
|          | 17.1     | 8183   | 17.1     | 8221   | 5     |

TaxA/HAI Dataset:
Dataset: 100K--40M/166k rows
Outperform existing baseline systems up to more than two orders of magnitude without sacrificing the repair quality!
Discussion
Contributions

- A data cleansing framework rather than a data cleansing algorithm

- Very useful as the emerging of big data

- There have been extensive studies on data cleansing algorithm for decades. But now the demand is to run these algorithms on distributed system.
Discussion
Contributions

Flexibility and scalability

- **Logic Abstraction**
  define customized rules together with traditional rules in a simple way with an UDF-based approach

- **Flexibility of Repair Algorithm**
  adopt existing algorithms or user-defined ones by treating the repair algorithm as a black box

- **Portable and Easy-to-use**
  run on various platforms ranging from DBMS to MapReduce-like platforms

- **Scalability and Fast Computation**
  integrate on general purpose distributed framework
When treating UDFs as black boxes, it is hard to do static analysis, such as consistency and implication, for the given rules.

Dataset must be static and quality rules must be predefined.
stream data / real-time cleansing / modified quality rules

Require domain expertise.
hinder its adoption in industrial targeting non-expert users
In the experiment, the authors added random text/numerical/duplicate errors and the system considers only key-based blockers, which is not accurate for many real-world data sets, due to dirty/missing data.

It is usually hard, if not impossible, to guarantee the accuracy of the data cleaning process without verifying it via experts or external sources.
Discussion

Limitations
Thanks!