Advanced Data Management (CSCI 680/490)

Data Cleaning

Dr. David Koop

Northern Illinois University

Types of Dirty Data Problems

- Separator Issues: e.g. CSV without respecting double quotes
  - `12, 13, "Doe, John", 45`
- Naming Conventions: `NYC` vs. `New York`
- Missing required fields, e.g. key
- Different representations: `2` vs. `two`
- Truncated data: `"Janice Keihanaikukauakahihuliheekahaunaele"` becomes `"Janice Keihanaikukauakahihuliheek"` on Hawaii license
- Redundant records: may be exactly the same or have some overlap
- Formatting issues: `2017-11-07` vs. `07/11/2017` vs. `11/07/2017`

[J. Canny et al.]

Dirty Data: Data Scientist's View

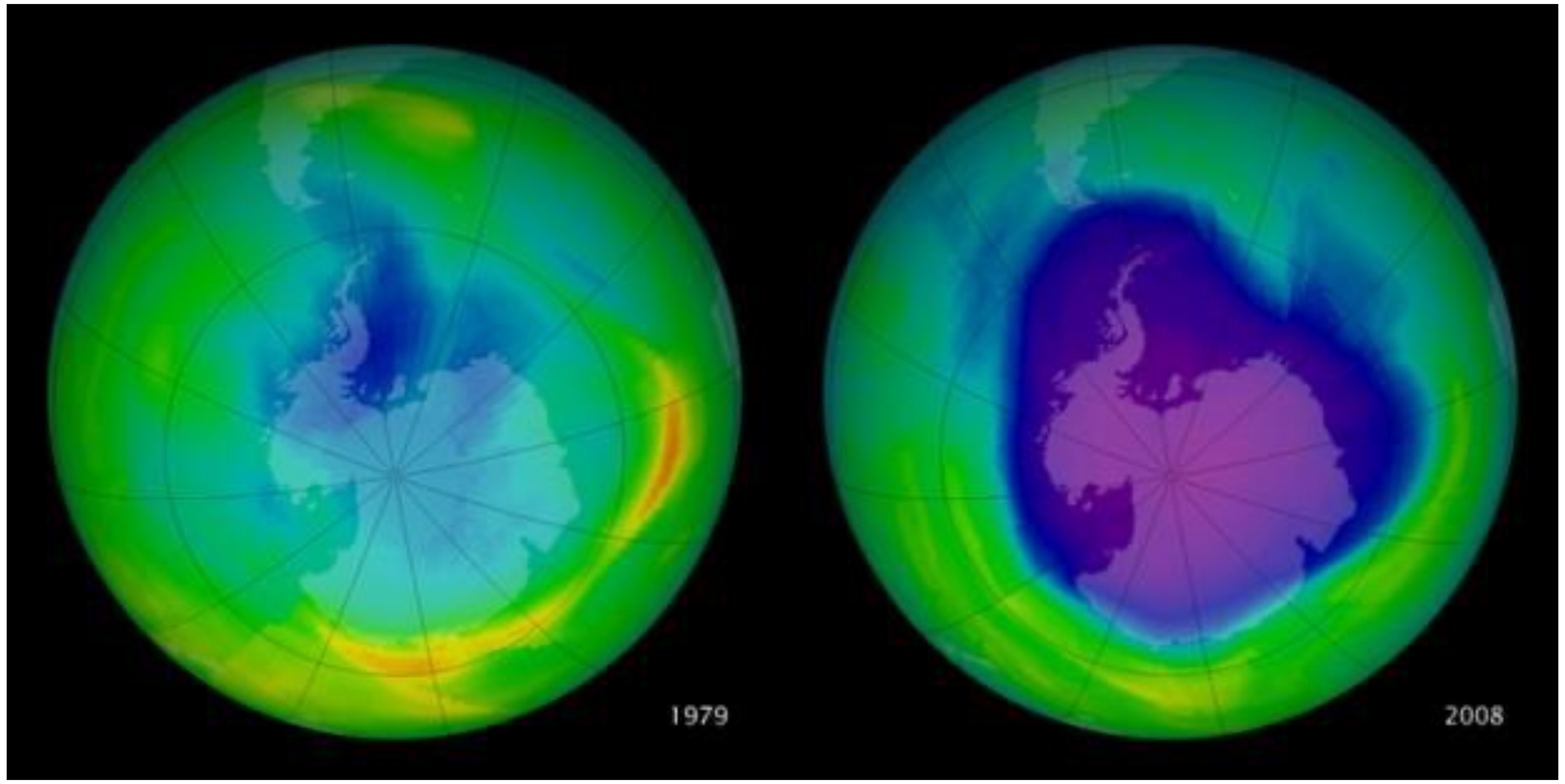
- Combination of:
  - Statistician's View: data has non-ideal samples for model
  - Database Expert's View: missing data, corrupted data
  - Domain Expert's View: data doesn't pass the smell test
- All of the views present problems with the data
- The goal may dictate the solutions:
  - Median value: don't worry too much about crazy outliers
  - Generally, aggregation is less susceptible by numeric errors
  - Be careful, the data may be correct…

[J. Canny et al.]

Be careful how you detect dirty data

- The appearance of a hole in the earth's ozone layer over Antarctica, first detected in 1976, was so unexpected that scientists didn't pay attention to what their instruments were telling them; they thought their instruments were malfunctioning.

  – National Center for Atmospheric Research



[Wikimedia]

Wrangler

- Data cleaning takes a lot of **time** and **human effort**
- "Tedium is the message"
- Repeating this process on multiple data sets is even worse!
- Solution:
  - interactive interface (mixed-initiative)
  - transformation language with natural language "translations"
  - suggestions + "programming by demonstration"

Potter's Wheel: Example

		Stewart,Bob
Anna	Davis	
		Dole,Jerry
Joan	Marsh	

Format
'(.*), (.*)' to '\2 \1'

		Bob Stewart
Anna	Davis	
		Jerry Dole
Joan	Marsh	

Split at ' '

		Bob	Stewart
Anna	Davis		
		Jerry	Dole
Joan	Marsh		

2 Merges

Bob	Stewart
Anna	Davis
Jerry	Dole
Joan	Marsh

[V. Raman and J. Hellerstein, 2001]

Potter's Wheel: Transforms

Transform	Definition		
Format	$\phi(R, i, f)$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, f(a_i)) \mid (a_1, \ldots, a_n) \in R\}$
Add	$\alpha(R, x)$	$=$	$\{(a_1, \ldots, a_n, x) \mid (a_1, \ldots, a_n) \in R\}$
Drop	$\pi(R, i)$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \mid (a_1, \ldots, a_n) \in R\}$
Copy	$\kappa((a_1, \ldots, a_n), i)$	$=$	$\{(a_1, \ldots, a_n, a_i) \mid (a_1, \ldots, a_n) \in R\}$
Merge	$\mu((a_1, \ldots, a_n), i, j, \text{glue})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, a_i \oplus \text{glue} \oplus a_j) \mid (a_1, \ldots, a_n) \in R\}$
Split	$\omega((a_1, \ldots, a_n), i, \text{splitter})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, \text{left}(a_i, \text{splitter}), \text{right}(a_i, \text{splitter})) \mid (a_1, \ldots, a_n) \in R\}$
Divide	$\delta((a_1, \ldots, a_n), i, \text{pred})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, a_i, \text{null}) \mid (a_1, \ldots, a_n) \in R \wedge \text{pred}(a_i)\} \cup$ $\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, \text{ null}, a_i) \mid (a_1, \ldots, a_n) \in R \wedge \neg\text{pred}(a_i)\}$
Fold	$\lambda(R, i_1, i_2, \ldots i_k)$	$=$	$\{(a_1, \ldots, a_{i_1-1}, a_{i_1+1}, \ldots, a_{i_2-1}, a_{i_2+1}, \ldots, a_{i_k-1}, a_{i_k+1}, \ldots, a_n, a_{i_l}) \mid$ $(a_1, \ldots, a_n) \in R \wedge 1 \leq l \leq k\}$
Select	$\sigma(R, \text{pred})$	$=$	$\{(a_1, \ldots, a_n) \mid (a_1, \ldots, a_n) \in R \wedge \text{pred}((a_1, \ldots, a_n))\}$

Notation: R is a relation with n columns. i, j are column indices and a_i represents the value of a column in a row. x and glue are values. f is a function mapping values to values. $x \oplus y$ concatenates x and y. splitter is a position in a string or a regular expression, left$(x, \text{splitter})$ is the left part of x after splitting by splitter. pred is a function returning a boolean.

[V. Raman and J. Hellerstein, 2001]

Interface

- Automated Transformation Suggestions
- Editable Natural Language Explanations

  - Fill **Bangladesh** by **copying** values from **above**
  - Fill **Bangladesh** by [averaging / ✓ copying / interpolating] values from **above**
  - Fill **Bangladesh** by **averaging** the **5** values from **above**

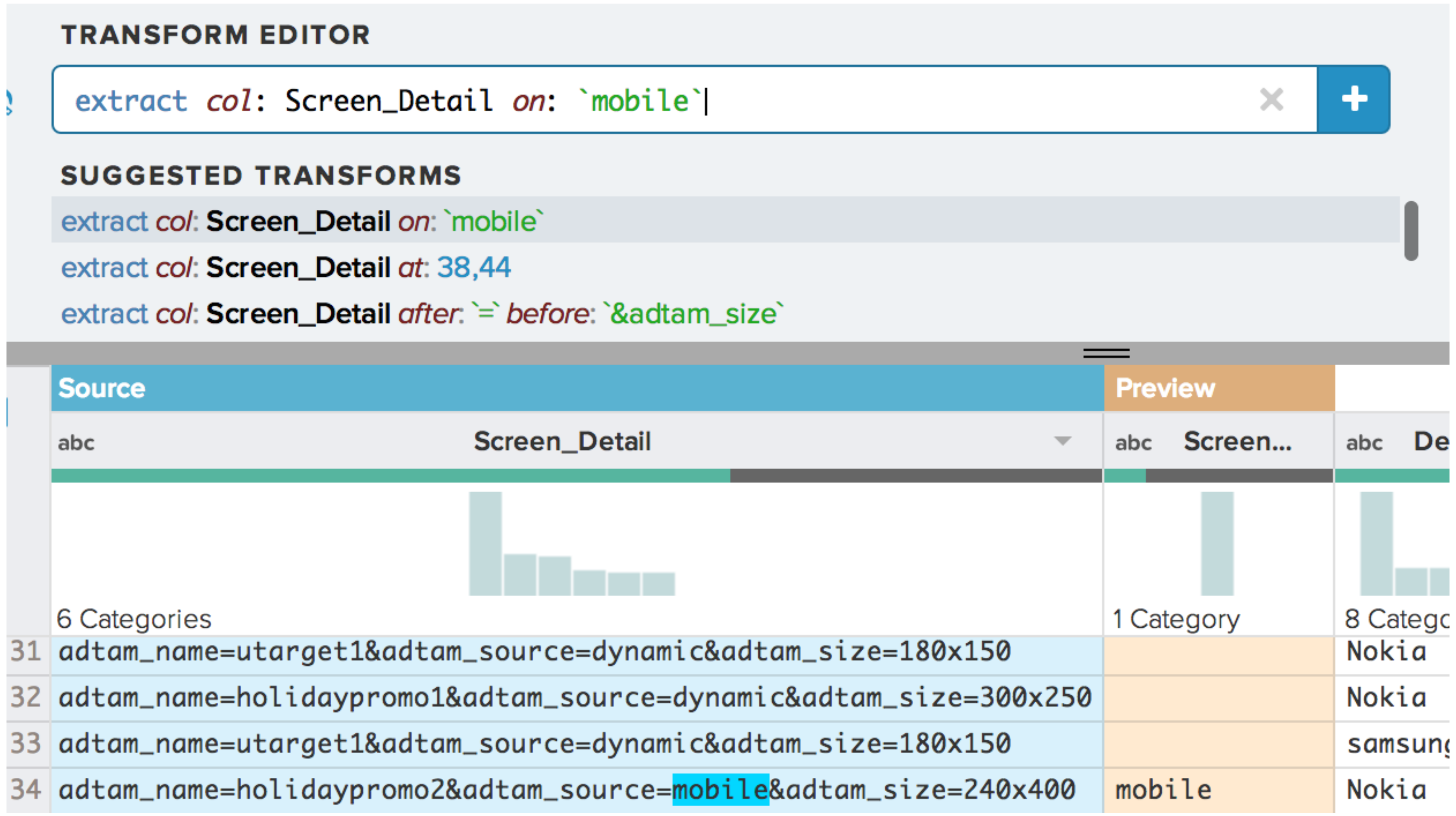
split	split1	split2	split3	split4
	2004	2004	2004	2003
STATE	Participation Rate 2004	Mean SAT I Verbal	Mean SAT I Math	Participation Rate
New York	87	497	510	82
Connecticut	85	515	515	84
Massachusetts	85	518	523	82
New Jersey	83	501	514	85
New Hampshire	80	522	521	75
D.C.	77	489	476	77
Maine	76	505	501	70
Pennsylvania	74	501	502	73
Delaware	73	500	499	73
Georgia	73	494	493	66

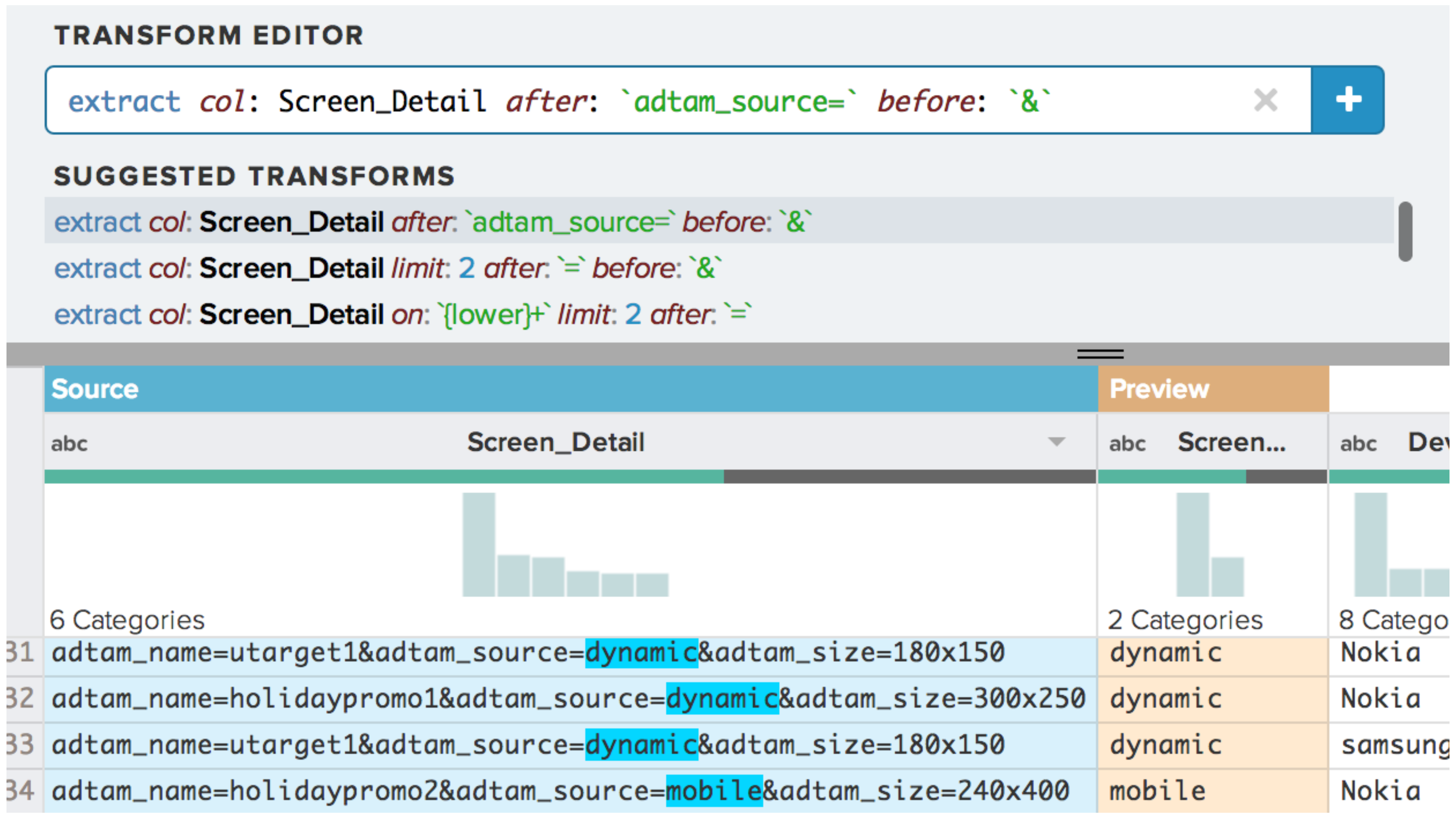
split	fold	fold1	value
New York	2004	Participation Rate 2004	87
New York	2004	Mean SAT I Verbal	497
New York	2004	Mean SAT I Math	510
New York	2003	Participation Rate 2003	82
New York	2003	Mean SAT I Verbal	496
New York	2003	Mean SAT I Math	510
Connecticut	2004	Participation Rate 2004	85
Connecticut	2004	Mean SAT I Verbal	515
Connecticut	2004	Mean SAT I Math	515
Connecticut	2003	Participation Rate 2003	84
Connecticut	2003	Mean SAT I Verbal	512

- Visual Transformation Previews
- Transformation History

[S. Kandel et al., 2011]

Improvements in Prediction





Update suggestions when given more information

[Heer et al., 2015]

Differences with Extract-Transform-Load (ETL)

- ETL:
  - Who: IT Professionals
  - Why: Create static data pipeline
  - What: Structured data
  - Where: Data centers
- "Modern Data Preparation":
  - Who: Analysts
  - Why: Solve problems by designing recipes to use data
  - What: Original, custom data blended with other data
  - Where: Cloud, desktop

[J. M. Hellerstein et al., 2018]

Test 1

- This Wednesday, Feb. 23
- In-class, 3:30-4:45pm in PM 153
- Format:
  - Multiple Choice
  - Free Response
- Information posted online

Paper Critique

- Foofah: Transforming Data By Example, Z. Jin et al., 2017
- Due Monday **before** class, submit via Blackboard
- Read the paper
- Look up references if necessary
- Keep track of things you are confused by or that seem problematic
- Write a few sentences summarizing the paper's contribution
- Write more sentences discussing the paper and what you think the paper does well or doesn't do well at
- For this response, compare/contrast with Wrangler/Trifacta
- Length: 1/2-1 page

Data Formats

Comma-separated values (CSV) Format

- Comma is a field separator, newlines denote records
  - ```
    a,b,c,d,message
    1,2,3,4,hello
    5,6,7,8,world
    9,10,11,12,foo
    ```
- May have a header (`a,b,c,d,message`), but not required
- No type information: we do not know what the columns are (numbers, strings, floating point, etc.)
  - Default: just keep everything as a string
  - Type inference: Figure out the type to make each column based on values
- What about commas in a value? → double quotes
    ```

Delimiter-separated Values

- Comma is a **delimiter**, specifies boundary between fields
- Could be a tab, pipe (`|`), or perhaps spaces instead
- All of these follow similar styles to CSV

Fixed-width Format

- Old school
- Each field gets a certain number of spots in the file
- Example:

```
- id8141    360.242940   149.910199   11950.7
  id1594    444.953632   166.985655   11788.4
  id1849    364.136849   183.628767   11806.2
  id1230    413.836124   184.375703   11916.8
  id1948    502.953953   173.237159   12468.3
```

- Specify exact character ranges for each field, e.g. 0-6 is the id

Reading & Writing Data

Reading Data in Python

- Use the `open()` method to open a file for reading
  - `f = open('huck-finn.txt')`
- Usually, add an `'r'` as the second parameter to indicate "read"
- Can iterate through the file (think of the file as a collection of lines):
  - ```
    f = open('huck-finn.txt', 'r')
    for line in f:
        if 'Huckleberry' in line:
            print(line.strip())
    ```
- Using `line.strip()` because the read includes the newline, and print writes a newline so we would have double-spaced text
- Closing the file: `f.close()`
    ```

With Statement: Improved File Handling

- With statement does "enter" and "exit" handling (similar to the finally clause):
- In the previous example, we need to remember to call `f.close()`
- Using a with statement, this is done automatically:
  - ```
    with open('huck-finn.txt', 'r') as f:
        for line in f:
            if 'Huckleberry' in line:
                print(line.strip())
    ```
- This is more important for writing files!
  - ```
    with open('output.txt', 'w') as f:
        for k, v in counts.items():
            f.write(k + ': ' + v + '\n')
    ```
- Without `with`, we need `f.close()`

Reading & Writing Data in Pandas

Format	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

[https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html]

Types of arguments for readers

- Indexing: choose a column to index the data, get column names from file or user
- Type inference and data conversion: automatic or user-defined
- Datetime parsing: can combine information from multiple columns
- Iterating: deal with very large files
- Unclean Data: skip rows (e.g. comments) or deal with formatted numbers (e.g. 1,000,345)

read_csv

- Convenient method to read csv files
- Lots of different options to help get data into the desired format
- Basic: `df = pd.read_csv(fname)`
- Parameters:
  - `path`: where to read the data from
  - `sep` (or `delimiter`): the delimiter (`','`, `' '`, `'\t'`, `'\s+'`)
  - `header`: if `None`, no header
  - `index_col`: which column to use as the row index
  - `names`: list of header names (e.g. if the file has no header)
  - `skiprows`: number of list of lines to skip

More read_csv/read_tables arguments

Argument	Description
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA.
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dict containing column number of name mapping to functions (e.g., `{'foo': f}` would apply the function `f` to all values in the `'foo'` column).
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read from beginning of file.
`iterator`	Return a `TextParser` object for reading file piecemeal.
`chunksize`	For iteration, size of file chunks.

[W. McKinney, Python for Data Analysis]

Chunked Reads

- With very large files, we may not want to read the entire file
- Why?
  - Time
  - Want to understand part of data before processing all of it
- Reading only a few rows:
  - `df = pd.read_csv('example.csv', nrows=5)`
- Reading chunks:
  - Get an iterator that returns the next chunk of the file
  - `chunker = pd.read_csv('example.csv', chunksize=1000)`
  - ```
    for piece in chunker:
        process_data(piece)
    ```
    ```

Python csv module

- Also, can read csv files outside of pandas using csv module
  - ```
    import csv
    with open('persons_of_concern.csv', 'r') as f:
        for i in range(3):
            next(f)
        reader = csv.reader(f)
        records = [r for r in reader] # r is a list
    ```
- or
  - ```
    import csv
    with open('persons_of_concern.csv', 'r') as f:
        for i in range(3):
            next(f)
        reader = csv.DictReader(f)
        records = [r for r in reader] # r is a dict
    ```

Writing CSV data with pandas

- Basic: `df.to_csv(<fname>)`
- Change delimiter with sep kwarg:
  - `df.to_csv('example.dsv', sep='|')`
- Change missing value representation
  - `df.to_csv('example.dsv', na_rep='NULL')`
- Don't write row or column labels:
  - `df.to_csv('example.csv', index=False, header=False)`
- Series may also be written to csv

eXtensible Markup Language (XML)

- Older, self-describing format with nesting; each field has tags
- Example:
  - ```
    <INDICATOR>
      <INDICATOR_SEQ>373889</INDICATOR_SEQ>
      <PARENT_SEQ></PARENT_SEQ>
      <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
      <INDICATOR_NAME>Escalator Avail.</INDICATOR_NAME>
      <PERIOD_YEAR>2011</PERIOD_YEAR>
      <PERIOD_MONTH>12</PERIOD_MONTH>
      <CATEGORY>Service Indicators</CATEGORY>
      <FREQUENCY>M</FREQUENCY>
      <YTD_TARGET>97.00</YTD_TARGET>
    </INDICATOR>
    ```
- Top element is the **root**
    ```

XML

- No built-in method
- Use lxml library (also can use ElementTree)
- ```
  from lxml import objectify
  path = 'datasets/mta_perf/Performance_MNR.xml'
  parsed = objectify.parse(open(path))
  root = parsed.getroot()
  data = []
  skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
                 'DESIRED_CHANGE','DECIMAL_PLACES']
  for elt in root.INDICATOR:
      el_data = {}
      for child in elt.getchildren():
          if child.tag in skip_fields:
              continue
          el_data[child.tag] = child.pyval
      data.append(el_data)
  perf = pd.DataFrame(data)
  ```

[W. McKinney, Python for Data Analysis]
  ```

JavaScript Object Notation (JSON)

- A format for web data
- Looks very similar to python dictionaries and lists
- Example:
  - ```
    {"name": "Wes",
     "places_lived": ["United States", "Spain", "Germany"],
     "pet": null,
     "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
                  {"name": "Katie", "age": 33, "pet": "Cisco"}] }
    ```
- Only contains literals (no variables) but allows null
- Values: strings, arrays, dictionaries, numbers, booleans, or null
  - Dictionary keys must be strings
  - Quotation marks help differentiate string or numeric values
    ```

What is the problem with reading this data?

- ```
  [{"name": "Wes",
     "places_lived": ["United States", "Spain", "Germany"],
     "pet": null,
     "siblings": [
        {"name": "Scott", "age": 25, "pet": "Zuko"},
        {"name": "Katie", "age": 33, "pet": "Cisco"}]
    },
    {"name": "Nia",
     "address": {"street": "143 Main",
                 "city": "New York",
                 "state": "New York"},
     "pet": "Fido",
     "siblings": [
        {"name": "Jacques", "age": 15, "pet": "Fido"}]
    },
   …
  ]
  ```
  ```

Reading JSON data

- Python has a built-in `json` module
  - ```
    with open('example.json') as f:
        data = json.load(f)
    ```
  - Can also load/dump to strings:
    - `json.loads, json.dumps`
- Pandas has `read_json, to_json` methods
    ```

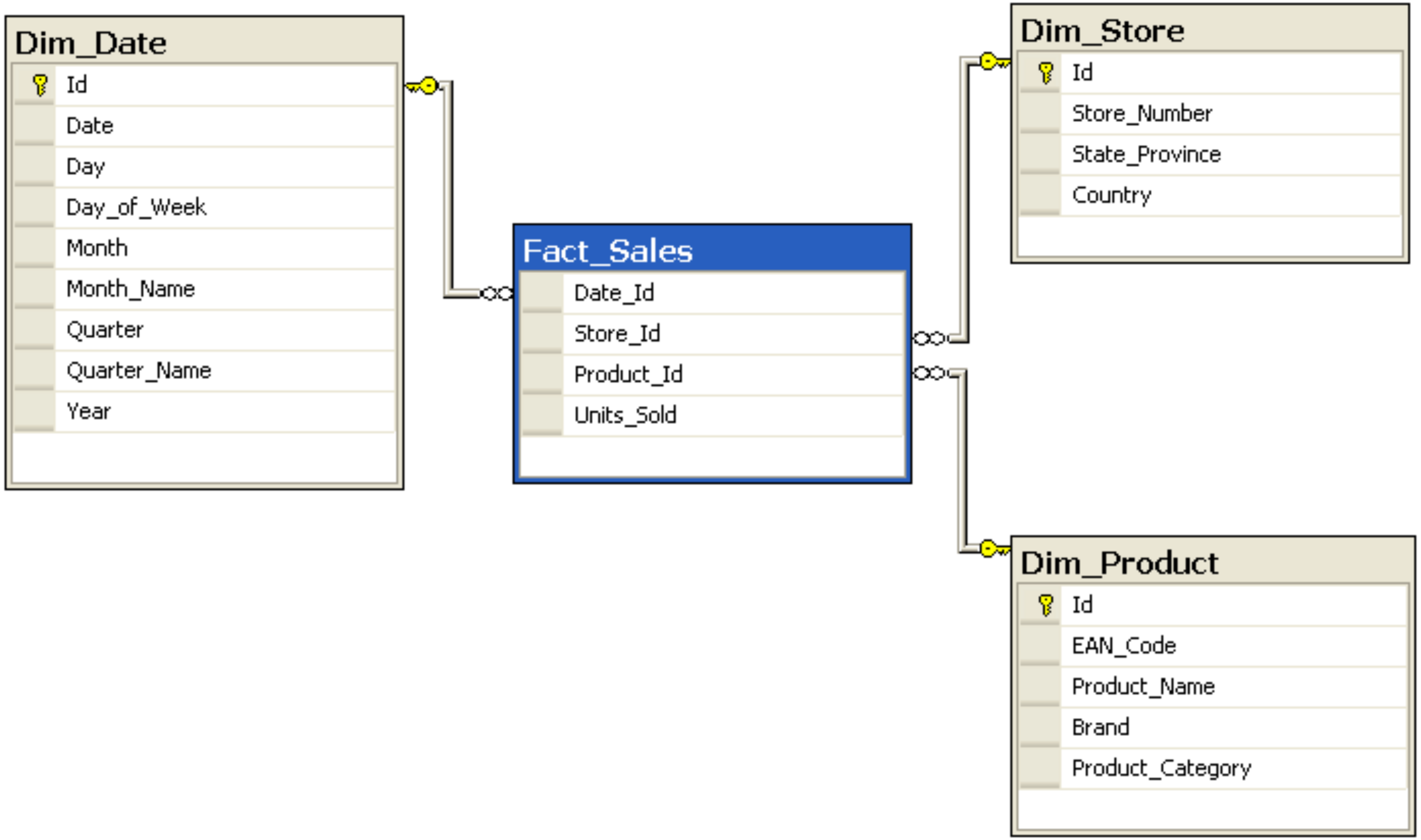
JSON Orientation

- Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:
  - `split`: dict like `{index -> [index], columns -> [columns], data -> [values]}`
  - `records`: list like `[{column -> value}, ... , {column -> value}]`
  - `index`: dict like `{index -> {column -> value}}`
  - `columns`: dict like `{column -> {index -> value}}`
  - `values`: just the values array

Binary Formats

- CSV, JSON, and XML are all text formats
- What is a binary format?
- Pickle: Python's built-in serialization
- HDF5: Library for storing large scientific data
  - Hierarchical Data Format, supports **compression**
  - Interfaces in C, Java, MATLAB, etc.
  - Use `pd.HDFStore` to access
  - Shortcuts: `read_hdf/to_hdf`, need to specify object
- Excel: need to specify sheet when a spreadsheet has multiple sheets
  - `pd.ExcelFile` or `pd.read_excel`

Databases



[Wikipedia]

Databases

- Relational databases are similar to multiple data frames but have more features
  - Links between tables via foreign keys
  - SQL to create, store, and query data
- duckdb is an OLAP database with support for python **and** pandas
- Python has a database API which lets you access most database systems through a common API.

Python DBAPI Example

```
import duckdb
query = """CREATE TABLE test(a VARCHAR(20), b VARCHAR(20),
                             c REAL, d INTEGER);"""
conn = duckdb.connect('mydata.sqlite')
conn.execute(query)
conn.commit()
# Insert some data
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
conn.executemany(stmt, data)
conn.commit()
```

[based on W. McKinney, Python for Data Analysis]

Databases

- Similar syntax from other database systems (sqlite, MySQL, Microsoft SQL Server, Oracle, etc.)
- SQLAlchemy: Python package that abstracts away differences between different database systems
- SQLAlchemy gives support for reading queries to data frame:
  - ```
    import sqlalchemy as sqla
    db = sqla.create_engine('sqlite:///mydata.sqlite')
    pd.read_sql('select * from test', db)
    ```
    ```

Data Cleaning

Classifying Data Quality Problems

Data Quality Problems

- *Single-Source Problems*
  - *Schema Level*
    (Lack of integrity constraints, poor schema design)
    - Uniqueness
    - Referential integrity
    ...
  - *Instance Level*
    (Data entry errors)
    - Misspellings
    - Redundancy/duplicates
    - Contradictory values
    ...
- *Multi-Source Problems*
  - *Schema Level*
    (Heterogeneous data models and schema designs)
    - Naming conflicts
    - Structural conflicts
    ...
  - *Instance Level*
    (Overlapping, contradicting and inconsistent data)
    - Inconsistent aggregating
    - Inconsistent timing
    ...

[E. Rahm & H. H. Do, 2000]

Single-Source Schema Problems

Scope/Problem		Dirty Data	Reasons/Remarks
Attribute	Illegal values	bdate=30.13.70	values outside of domain range
Record	Violated attribute dependencies	age=22, bdate=12.02.70	age = (current date – birth date) should hold
Record type	Uniqueness violation	emp_1=(name=”John Smith”, SSN=”123456”) emp_2=(name=”Peter Miller”, SSN=”123456”)	uniqueness for SSN (social security number) violated
Source	Referential integrity violation	emp=(name=”John Smith”, deptno=127)	referenced department (127) not defined

[E. Rahm & H. H. Do, 2000]

Single-Source Instance Problems

Scope/Problem		Dirty Data	Reasons/Remarks
Attribute	Missing values	phone=9999-999999	unavailable values during data entry (dummy values or null)
	Misspellings	city="Liipzig"	usually typos, phonetic errors
	Cryptic values, Abbreviations	experience="B"; occupation="DB Prog."	
	Embedded values	name="J. Smith 12.02.70 New York"	multiple values entered in one attribute (e.g. in a free-form field)
	Misfielded values	city="Germany"	
Record	Violated attribute dependencies	city="Redmond", zip=77777	city and zip code should correspond
Record type	Word transpositions	$name_1$= "J. Smith", $name_2$="Miller P."	usually in a free-form field
	Duplicated records	emp_1=(name="John Smith",...); emp_2=(name="J. Smith",...)	same employee represented twice due to some data entry errors
	Contradicting records	emp_1=(name="John Smith", bdate=12.02.70); emp_2=(name="John Smith", bdate=12.12.70)	the same real world entity is described by different values
Source	Wrong references	emp=(name="John Smith", deptno=17)	referenced department (17) is defined but wrong

[E. Rahm & H. H. Do, 2000]

Multi-Source Schema & Instance Problems

Customer (source 1)

CID	*Name*	*Street*	*City*	*Sex*
11	Kristen Smith	2 Hurley Pl	South Fork, MN 48503	0
24	Christian Smith	Hurley St 2	S Fork MN	1

Client (source 2)

Cno	*LastName*	*FirstName*	*Gender*	*Address*	*Phone/Fax*
24	Smith	Christoph	M	23 Harley St, Chicago IL, 60633-2394	333-222-6542 / 333-222-6599
493	Smith	Kris L.	F	2 Hurley Place, South Fork MN, 48503-5998	444-555-6666

Customers (integrated target with cleaned data)

No	*LName*	*FName*	*Gender*	*Street*	*City*	*State*	*ZIP*	*Phone*	*Fax*	*CID*	*Cno*
1	Smith	Kristen L.	F	2 Hurley Place	South Fork	MN	48503-5998	444-555-6666		11	493
2	Smith	Christian	M	2 Hurley Place	South Fork	MN	48503-5998			24	
3	Smith	Christoph	M	23 Harley Street	Chicago	IL	60633-2394	333-222-6542	333-222-6599		24

[E. Rahm & H. H. Do, 2000]

HoloClean

- A holistic data cleaning framework that combines qualitative methods with quantitative methods:
  - Qualitative: use integrity constraints or external data sources
  - Quantitative: use statistics of the data
- Driven by probabilistic inference. Users only need to provide a dataset to be cleaned and describe high-level domain specific signals.
- Can scale to large real-world dirty datasets and perform automatic repairs with high accuracy

[T. Rekatsinas et al., 2017]

Example: Input Data

(A) Input Database External Information (Chicago food inspections)

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	***Chicago***	IL	***60608***
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t4	***Johnnyo's***	Johnnyo's	3465 S Morgan ST	***Cicago***	IL	60608

Conflicts due to c2

Does not obey data distribution

Conflict due to c2

(B) Functional Dependencies

$$\text{c1: DBAName} \rightarrow \text{Zip}$$
$$\text{c2: Zip} \rightarrow \text{City, State}$$
$$\text{c3: City, State, Address} \rightarrow \text{Zip}$$

[T. Rekatsinas et al., 2017]

Example: Fixing via Minimality

(A) Input Database External Information (Chicago food inspections)

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	***Chicago***	IL	***60608***
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t4	***Johnnyo's***	Johnnyo's	3465 S Morgan ST	***Cicago***	IL	60608

Conflicts due to c2

Does not obey data distribution

Conflict due to c2

(B) Functional Dependencies

$$\text{c1: DBAName} \rightarrow \text{Zip}$$
$$\text{c2: Zip} \rightarrow \text{City, State}$$
$$\text{c3: City, State, Address} \rightarrow \text{Zip}$$

(E) Repair using Minimality w.r.t FDs

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60609**
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60609**
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60609**
t4	**Johnnyo's**	Johnnyo's	3465 S Morgan ST	**Cicago**	IL	60608

[T. Rekatsinas et al., 2017]

Example: Fixing via External Matches

(C) Matching Dependencies

$$\begin{aligned}
&\text{m1: } \text{Zip} = \text{Ext_Zip} \rightarrow \text{City} = \text{Ext_City} \\
&\text{m2: } \text{Zip} = \text{Ext_Zip} \rightarrow \text{State} = \text{Ext_State} \\
&\text{m3: } \text{City} = \text{Ext_City} \wedge \text{State} = \text{Ext_State} \wedge \\
&\qquad \wedge \text{Address} = \text{Ext_Address} \rightarrow \text{Zip} = \text{Ext_Zip}
\end{aligned}$$

(D) External Information (Address listings in Chicago)

Ext_Address	Ext_City	Ext_State	Ext_Zip
3465 S Morgan ST	Chicago	IL	60608
1208 N Wells ST	Chicago	IL	60610
259 E Erie ST	Chicago	IL	60611
2806 W Cermak Rd	Chicago	IL	60623

(A) Input Database External Information (Chicago food inspections)

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	***Chicago***	IL	***60608***
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t4	***Johnnyo's***	Johnnyo's	3465 S Morgan ST	***Cicago***	IL	60608

Conflicts due to c2

Does not obey data distribution

Conflict due to c2

(F) Repair using Matching Dependencies

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	60608
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60608**
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60608**
t4	**Johnnyo's**	Johnnyo's	3465 S Morgan ST	**Chicago**	IL	60608

[T. Rekatsinas et al., 2017]

Example: Fixing via Statistics

(A) Input Database External Information
(Chicago food inspections)

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	***Chicago***	IL	***60608***
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	***60609***
t4	***Johnnyo's***	Johnnyo's	3465 S Morgan ST	***Cicago***	IL	60608

Conflicts due to c2

Does not obey data distribution

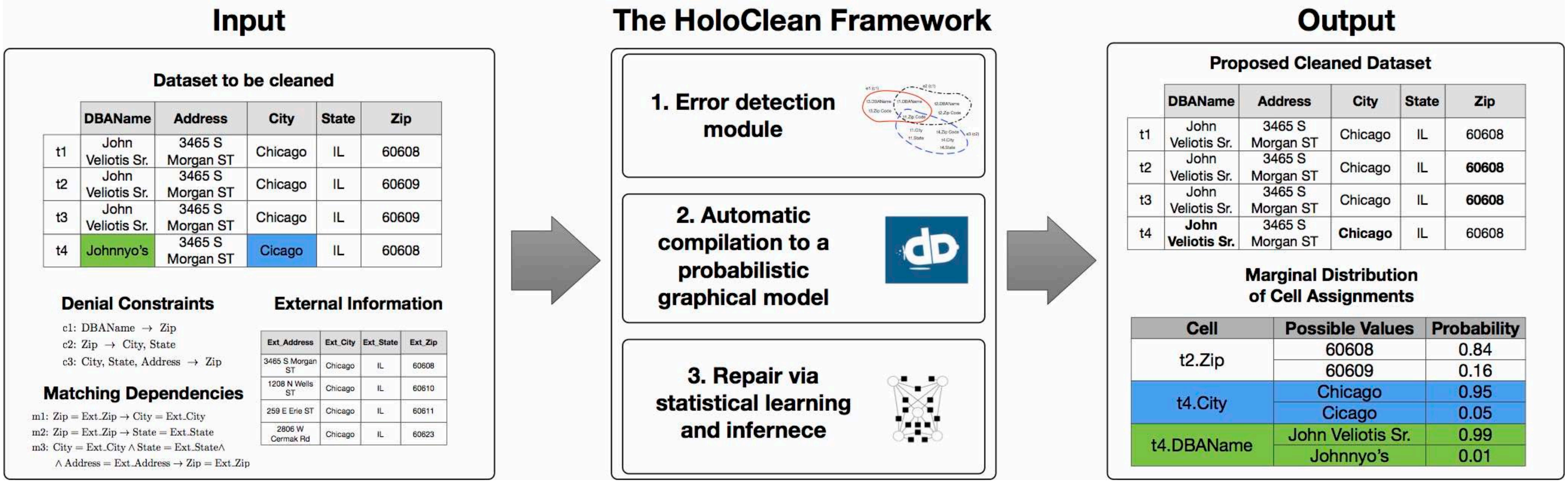
Conflict due to c2

(G) Repair that leverages Quantitative Statistics

	DBAName	AKAName	Address	City	State	Zip
t1	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	60608
t2	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60609**
t3	John Veliotis Sr.	Johnnyo's	3465 S Morgan ST	Chicago	IL	**60609**
t4	**John Veliotis Sr.**	Johnnyo's	3465 S Morgan ST	**Chicago**	IL	60608

[T. Rekatsinas et al., 2017]

HoloClean



[T. Rekatsinas et al., 2017]

Data Cleaning in pandas

Handling Missing Data

- Filtering out missing data:
  - Can choose rows or columns
- Filling in missing data:
  - with a default value
  - with an interpolated value
- In pandas:

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

[W. McKinney, Python for Data Analysis]

Filling in missing data

- fillna arguments:

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

[W. McKinney, Python for Data Analysis]

Filtering and Cleaning Data

- Find duplicates
  - `duplicated`: returns boolean Series indicating whether row is a duplicate—first instance is **not marked** as a duplicate
- Remove duplicates:
  - `drop_duplicates`: drops all rows where `duplicated` is `True`
  - `keep`: which value to keep (first or last)
- Can pass specific columns to check for duplicates, e.g. check only key column

Changing Data

- Convert strings to upper/lower case
- Convert Fahrenheit temperatures to Celsius
- Create a new column based on another column

```
In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

```
In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

[W. McKinney, Python for Data Analysis]

Replacing Values

- `fillna` is a special case
- What if `-999` in our dataset was identified as a missing value?

```
In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

```
In [62]: data.replace(-999, np.nan)
Out[62]:
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

- Can pass list of values or dictionary to change different values

Clamping Values

- Values above or below a specified thresholds are set to a max/min value

```
In [93]: data.describe()
Out[93]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.049091     0.026112    -0.002544    -0.051827
std       0.996947     1.007458     0.995232     0.998311
min      -3.645860    -3.184377    -3.745356    -3.428254
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.525865     2.735527     3.366626
```

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3

In [98]: data.describe()
Out[98]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.050286     0.025567    -0.001399    -0.051765
std       0.992920     1.004214     0.991414     0.995761
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.000000     2.735527     3.000000
```

Computing Indicator Values

- Useful for machine learning
- Want to take possible values and map them to 0-1 indicators
- Example:

```
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                    'data1': range(6)})

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

- Example: Genres in movies

String Transformation

- One of the reasons for Python's popularity is string/text processing
- `split(<delimiter>)`: break a string into pieces:

```
- s = "12,13, 14"
  slist = s.split(',') # ["12", "13", " 14"]
```

- `<delimiter>.join([<str>])`: join several strings by a delimiter

```
- ":".join(slist) # "12:13: 14"
```

- `strip()`: remove leading and trailing whitespace

```
- [p.strip() for p in slist] # ["12", "13", "14"]
```

String Transformation

- `replace(<from>,<to>)`: change substrings to another substring
  - `s.replace(',', ':') # "12:13: 14"`
- `upper()/lower()`: casing
  - `"AbCd".upper () # "ABCD"`
  - `"AbCd".lower() # "abcd"`

String Transformations

- `index(<str>)`: find where a substring first occurs (Error if not found)
  - ```
    s = "12,13, 14"
    s.index(',') # 2
    s.index(':') # ValueError raised
    ```
- `find(<str>)`: same as index but `-1` if not found
  - ```
    s.find(',') # 2
    s.find(':') # -1
    ```
- `startswith()`/`endswith()`: boolean checks for string occurrence
  - ```
    s.startswith("1") # True
    s.endswith("5") # False
    ```
    ```

String Methods

Argument	Description
`count`	Return the number of non-overlapping occurrences of substring in the string.
`endswith`	Returns `True` if string ends with suffix.
`startswith`	Returns `True` if string starts with prefix.
`join`	Use string as delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string; raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
`replace`	Replace occurrences of string with another string.
`strip, rstrip, lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip, lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
`ljust, rjust`	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

[W. McKinney, Python for Data Analysis]

Regular Expressions

- AKA regex
- A syntax to better specify how to decompose strings
- Look for patterns rather than specific characters
  - `"31" in "The last day of December is 12/31/2020."`
- May work for some questions but now suppose I have other lines like: `"The last day of September is 9/30/2020."`
- …and I want to find dates that look like:
  - `<numbers>/<numbers>/<numbers>`
- Cannot search for every combination!
  - `\d+/\d+/\d+`

Regular Expressions

- Character classes:
  - `\d` = digits
  - `\s` = spaces
  - `\w` = word character `[a-zA-Z0-9_]`
  - `[a-z]` = lowercase letters (square brackets indicate a set of chars)
- Repeating characters or patterns
  - `+` = one or more (any number)
  - `*` = zero or more (any number)
  - `?` = zero or one
  - `{<number>}` = a specific number (or range) of occurrences

Regular Expressions in Python

- `import re`
- `re.search(<pattern>, <str_to_check>)`
  - Returns `None` if no match, information about the match otherwise
- Capturing information about what is in a string → **parentheses**
- `(\d+)/\d+/\d+` will **capture** information about the month
-
```
match = re.search('(\d+)/\d+/\d+','12/31/2016')
if match:
    match.group() # 12
```
- `re.findall(<pattern>, <str_to_check>)`
  - Finds all matches in the string, search only finds the first match
- Can pass in flags to alter methods: e.g. `re.IGNORECASE`

Pandas String Methods

- Any column or series can have the string methods (e.g. replace, split) applied to the entire series
- Fast (vectorized) on whole columns or datasets
- use `.str.<method_name>`
- `.str` is **important**!
  - ```
    data = pd.Series({'Dave': 'dave@google.com',
                      'Steve': 'steve@gmail.com',
                      'Rob': 'rob@gmail.com',
                      'Wes': np.nan})
    data.str.contains('gmail')
    data.str.split('@').str[1]
    data.str[-3:]
    ```
    ```