# CONTENTS

1 Introduction .......................... 3
  1.1 Uses .................................. 3
  1.2 Modules ................................ 3

2 Installation .............................. 5
  2.1 Quick Start ............................ 5
  2.2 Requirements ......................... 5
  2.3 Customized Installation ............... 6

3 metadata — Information about data structure .......................... 7
  3.1 Field types ............................ 7
  3.2 Fields and Field Lists .................. 8
  3.3 Concrete storage type ................. 10
  3.4 Field mapping ......................... 10

4 ds — Data Stores ......................... 13
  4.1 Overview ............................. 13
  4.2 Data Sources .......................... 15
  4.3 Data Targets .......................... 15
  4.4 Base Classes ................................ 15
  4.5 Sources ................................ 17
  4.6 Targets ................................ 19

5 streams — Data Analysis and Processing Streams ....................... 23
  5.1 Data Processing Streams ............... 23
  5.2 Running Streams ........................ 25
  5.3 Forking Forks with Higher Order Messaging .......................... 26
  5.4 Custom nodes .......................... 28

6 Node Reference .......................... 31
  6.1 Sources .............................. 31
  6.2 Record Operations ...................... 36
  6.3 Field Operations ....................... 42
  6.4 Targets ................................ 45

7 probes — Data Auditing Probes ................................ 51

8 Data Quality ............................ 53

9 Command Line Tools ....................... 57
Know what you have and learn what is happening to your data

Brewery is a stream-based data analysis and auditing framework with focus on understandability, auditability, usability and versatility.

Contents:
INTRODUCTION

Brewery is a Python framework for data analysis and data quality measurement. Principle of the framework are streams of structured data that flow between processing nodes.

Priorities of the framework are:

- understandability of the analysis process
- auditability of the data being analyzed (frequent use of metadata)
- usability
- versatility

Speed is currently a minor priority of the framework. If you are concerned about performance, you can still use the framework in your thinking and designing process, to experience the data you are about to process. Brewery provides several ways how to get just small samples the data. However, if you know how to improve any parts of the framework, you are welcome.

1.1 Uses

When you might consider using brewery?

- data analysis
- data monitoring
- data auditing
- learn more about unknown datasets
- feed auditing and analysis results back to data stores
- streaming data in heterogenous environment - between different stores

Even though Data Brewery is not a full-featured ETL framework it is possible to use it for simple operations, for playing around with data, piping data from one store to another.

1.2 Modules

The framework consists of several modules:

- metadata – field types and field type operations, describe structure of data (available directly from the brewery package namespace)
• \texttt{ds} – structured data streams data sources and data targets
• \texttt{streams} – data processing streams
• nodes – analytical and processing stream nodes (see \textit{Node Reference})
• \texttt{probes} – analytical and quality data probes
2.1 Quick Start

Here are quick installation instructions for the impatient.

Brewery is being developed for Python 2.7, reported to work on 2.6.

Satisfy soft dependencies that cover most of the use cases. For more information read below:

```
pip install sqlalchemy xlrd
```

Install brewery:

```
pip install brewery
```

Try:

```
import brewery
```

```
URL = "https://raw.github.com/Stiivi/cubes/master/examples/hello_world/data.csv"
```

```
b = brewery.create_builder()
b.csv_source(URL)
b.audit(distinct_threshold=None)
b.pretty_printer()
```

```
b.stream.run()
```

Or the same from the command line:

```
$ curl https://raw.github.com/Stiivi/cubes/master/examples/hello_world/data.csv | 
   brewery pipe audit pretty_printer
```

2.2 Requirements

The framework currently does not have any hard dependency on other packages. All dependencies are optional and you need to install the packages only if certain features are going to be used.
<table>
<thead>
<tr>
<th>Package</th>
<th>Feature</th>
<th>Source</th>
<th>Recommended version</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlalchemy</td>
<td>Streams from/to SQL databases.</td>
<td><a href="http://www.sqlalchemy.org">http://www.sqlalchemy.org</a></td>
<td>&gt; 0.7</td>
</tr>
<tr>
<td>gdata</td>
<td>Google data (spreadsheet) source/target</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xlrd</td>
<td>Reading MS Excel XLS Files.</td>
<td><a href="http://pypi.python.org/pypi/xlrd">http://pypi.python.org/pypi/xlrd</a></td>
<td></td>
</tr>
</tbody>
</table>

### 2.3 Customized Installation

The project sources are stored in the [Github repository](https://github.com/Stiivi/brewery).

Download from Github:

```
git clone git://github.com/Stiivi/brewery.git
```

Install:

```
cd brewery
python setup.py install
```
While working with structured data it is helpful to know how the structure looks like, what are the fields, what are their types.

### 3.1 Field types

There are two kinds of field types: storage type and analytical type. The storage type specifies how the value is being stored in the source, the type is normalized. Another type is analytical type which is used in data mining, defines if the field can be used by particular algorithm and how the field is treated by mining algorithms.

#### Storage types

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>names, labels, short descriptions; mostly implemented as VARCHAR type in database, or can be found as CSV file fields</td>
</tr>
<tr>
<td>text</td>
<td>longer texts, long descriptions, articles</td>
</tr>
<tr>
<td>integer</td>
<td>discrete values</td>
</tr>
<tr>
<td>float</td>
<td>numerical value with floating point</td>
</tr>
<tr>
<td>boolean</td>
<td>binary value, mostly implemented as small integer</td>
</tr>
<tr>
<td>date</td>
<td>calendar date representation</td>
</tr>
</tbody>
</table>

#### Analytical types
<table>
<thead>
<tr>
<th><strong>Analytical Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>set</em></td>
<td>Values represent categories, like colors or contract. types. Fields of this type might be numbers which represent for example group numbers, but have no mathematical interpretation. For example addition of group numbers 1+2 has no meaning.</td>
</tr>
<tr>
<td><em>ordered_set</em></td>
<td>Similar to <em>set</em> field type, but values can be ordered in a meaningful order.</td>
</tr>
<tr>
<td><em>discrete</em></td>
<td>Set of integers - values can be ordered and one can perform arithmetic operations on them, such as: 1 contract + 2 contracts = 3 contracts</td>
</tr>
<tr>
<td><em>flag</em></td>
<td>Special case of <em>set</em> type where values can be one of two types, such as 1 or 0, ‘yes’ or ‘no’, ‘true’ or ‘false’.</td>
</tr>
<tr>
<td><em>range</em></td>
<td>Numerical value, such as financial amount, temperature</td>
</tr>
<tr>
<td><em>default</em></td>
<td>Analytical type is not explicitly set and default type for fields storage type is used. Refer to the table of default types.</td>
</tr>
<tr>
<td><em>typeless</em></td>
<td>Field has no analytical relevance.</td>
</tr>
</tbody>
</table>

Default analytical types:

- *integer* is *discrete*
- *float* is *range*
- *unknown, string, text, date* are typeless

### 3.2 Fields and Field Lists

Main metadata class is `Field` which gives information about name, types and other useful data attributes. Field might represent a database column in a SQL database, a key in a dictionary-like record...

```python
class brewery.metadata.Field(name, storage_type='unknown', analytical_type='typeless', concrete_storage_type=None, missing_values=None, label=None)
```

Metadata - information about a field in a dataset or in a datastream.

**Attributes**

- *name* - field name
- *label* - optional human readable field label
- *storage_type* - Normalized data storage type. The data storage type is abstracted
- *concrete_storage_type* (optional, recommended) - Data store/database dependent storage type - this is the real name of data type as used in a database where the field comes from or where the field is going to be created (this might be null if unknown)
- *analytical_type* - data type used in data mining algorithms
- *missing_values* (optional) - Array of values that represent missing values in the dataset for given field

```python
to_dict()
```

Return dictionary representation of the field.

In most cases we are dealing with structured data here, therefore we are working with multiple fields and values at once. For that purpose there is `FieldList` – ordered list of field descriptions:

Fields can be compared using `==` and `!=` operators. They are equal if all attributes are equal. Getting a string representation `str(field)` of a field returns field name.
name

```python
class brewery.metadata.FieldList(fields=None)
    Create a list of Field objects from a list of strings, dictionaries or tuples

    How fields are constructed:
    • string: field name is set
    • tuple: (field_name, storage_type, analytical_type), the field_name is obligatory, rest is optional
    • dict: contains key-value pairs for initializing a Field object

    For strings and in if not explicitly specified in a tuple or a dict case, then following rules apply:
    • storage_type is set to unknown
    • analytical_type is set to typeless

    append(field)
    Appends a field to the list. This method requires field to be instance of Field

    copy(fields=None)
    Return a shallow copy of the list.

    Parameters
    • fields - list of fields to be copied.

    field(name)
    Return a field with name name

    fields(names=None)
    Return a tuple with fields. names specifies which fields are returned. When names is None all fields are returned.

    index(field)
    Return index of a field

    indexes(fields)
    Return a tuple with indexes of fields from fields in a data row. Fields should be a list of Field objects or strings.

    This method is useful when it is more desirable to process data as rows (arrays), not as dictionaries, for example for performance purposes.

    names(indexes=None)
    Return names of fields in the list.

    Parameters
    • indexes - list of indexes for which field names should be collected. If set to None then all field names are collected - this is default behaviour.

    retype(dictionary)
    Retype fields according to the dictionary. Dictionary contains field names as keys and field attribute dictionary as values.

    selectors(fields=None)
    Return a list representing field selector - which fields are selected from a row.
```

In addition, the FieldList behaves as a list: implements `len()`, `del`, `[]` with field index, `+=` for appending fields, `+` for creating new field list by concatenating two other lists.

---

3.2. Fields and Field Lists
Field lists are used in data sources, data targets, processing streams, nodes, ... They are mostly present in the form of a fields attribute (in a class) or function parameter with the same name. To make it easy to quickly construct list of fields with all necessary metadata you can do:

```python
import brewery.metadata as metadata

fields = metadata.FieldList(["organisation", "address", "type", "amount"])
```

If you are implementing a function that changes data structure, do not change the fields you have received from the source. Make a copy and do modifications in the copy:

```python
import brewery.streams

class AppendTimestampNode(brewery.streams.Node):
    def initialize(self):
        # Create a copy
        fields = self.input.fields.copy()

        # Append custom field(s)
        timestamp_field = Field("timestamp", storage_type = "date")
        fields.append(timestamp_field)

        self.output_fields = fields
```

### 3.3 Concrete storage type

Each field can have specified concrete storage type - closest type definition to the real storage. Value of this attribute is dependent on a backend providing field information about data source or data target. For example, SQL backend can use type class or type class instance. Reason for storing concrete storage type is to preserve the type in homogenous environment in the first place. Second reason is to allow custom mappings between backend data types.

Brewery does not perform any mapping currently. If the backends are not compatible, the concrete storage is simply ignored and default type from normalized plain storage_type is used.

### 3.4 Field mapping

Quite common operation is field renaming and dropping of unused fields, for example those that were already transformed. This might be also called field filtering.

```python
class brewery.metadata.FieldMap (rename=None, drop=None, keep=None)

    Creates a field map. rename is a dictionary where keys are input field names and values are output field names.
    drop is list of field names that will be dropped from the stream. If keep is used, then all fields are dropped except those specified in keep list.

    field_selectors (fields)
    Returns selectors of fields to be used by itertools.compress(). This is the preferred way of field filtering.

    map (fields)
    Map fields according to the FieldMap: rename or drop fields as specified. Returns a FieldList object.

    row_filter (fields)
    Returns an object that will convert rows with structure specified in fields. You can use the object to filter fields from a row (list, array) according to this map.
```
For example our requirement is to do following field mapping/filtering:

```python
import brewery.metadata as metadata

fields = metadata.FieldList(['d_org', 'd_addr', 'type', 'amount'])

map = metadata.FieldMap(rename = {'d_org': 'organisation', 'd_addr': 'address'}, drop = ['type'])
mapped_fields = map.map(fields)
print(mapped_fields.names())

# Now we have mapped_fields = ['organisation', 'address', 'amount']
```

To apply field mapping onto a row (list, tuple), there is `RowFieldFilter`. Following example shows how to filter fields from list of rows:

```python
# Assume that we have rows with structure specified in previous example in 'fields'

filter = map.row_filter(fields)

output = []
for row in rows:
    output.append(filter.filter(row))

# Output will contain only fields as in 'mapped_fields' from the previous example
```

```python
class brewery.metadata.RowFieldFilter(selectors=None)
    Create an instance of RowFieldFilter. indexes is a list of indexes that are passed to output.

    filter(row)
        Filter a row according to indexes.
```
**Warning:** This module will be very likely renamed from *ds* to ‘stores’. Currently there is confusion whether ‘ds’ means ‘data stores’ or ‘data streams’. There is also another module called *streams*: processing streams based on nodes connected with pipes.

### 4.1 Overview

Data stores provide interface for common way of reading from and writing to various structured data stores through structured data streams. It allows you to read CSV file and merge it with Excel spreadsheet or Google spreadsheet, then perform cleansing and write it to a relational database table or create a report.

The data streams can be compared to file-like stream objects where structured data is being passed instead of bytes. There are two ways how to look at structured data: as a set of lists of values or as a set of key-value pairs (set of dictionaries). Some sources provide one or the other way of looking at the data, some processing is better with the list form, another might be better with the dictionary form. Brewery allows you to use the form which is most suitable for you.

At any time you are able to retrieve stream metadata: list of fields being streamed. For more information see metadata where you can find more information.
Figure 4.2: Example of streaming data as sequence of rows - tuples of values.

Figure 4.3: Example of streaming data as sequence of records - dictionaries with key-value pairs.
4.2 Data Sources

<table>
<thead>
<tr>
<th>Data source</th>
<th>Description</th>
<th>Dataset reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>csv</td>
<td>Comma separated values (CSV) file/URI resource</td>
<td>file path, file-like object, URL</td>
</tr>
<tr>
<td>xls</td>
<td>MS Excel spreadsheet</td>
<td>file path, URL</td>
</tr>
<tr>
<td>gdoc</td>
<td>Google Spreadsheet</td>
<td>spreadsheet key or name</td>
</tr>
<tr>
<td>sql</td>
<td>Relational database table</td>
<td>connection + table name</td>
</tr>
<tr>
<td>mongodb</td>
<td>MongoDB database collection</td>
<td>connection + table name</td>
</tr>
<tr>
<td>yamldir</td>
<td>Directory containing yaml files - one file per record</td>
<td>directory</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>Elastic Search – Open Source, Distributed, RESTful, Search Engine</td>
<td></td>
</tr>
</tbody>
</table>

Data sources should implement:

- initialize() - delayed initialisation: get fields if they are not set, open file stream, ...
- rows() - returns iterable with value tuples
- records() - returns iterable with dictionaries of key-value pairs

Should provide property fields, optionally might provide assignment of this property.

4.3 Data Targets

<table>
<thead>
<tr>
<th>Data target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csv</td>
<td>Comma separated values (CSV) file/URI resource</td>
</tr>
<tr>
<td>sql</td>
<td>Relational database table</td>
</tr>
<tr>
<td>mongodb</td>
<td>MongoDB database collection</td>
</tr>
<tr>
<td>yamldir</td>
<td>Directory containing yaml files - one file per record</td>
</tr>
<tr>
<td>jsondir</td>
<td>Directory containing json files - one file per record (not yet)</td>
</tr>
<tr>
<td>html</td>
<td>HTML file or a string target</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>Elastic Search – Open Source, Distributed, RESTful, Search Engine</td>
</tr>
</tbody>
</table>

Data targets should implement:

- initialize() - create dataset if required, open file stream, open db connection, ...
- append(object) - appends object as row or record depending whether it is a dictionary or a list

4.4 Base Classes

Use these classes as super classes for your custom structured data sources or data targets.

```python
class brewery.dsDataStream
    A data stream object – abstract class.
```

The subclasses should provide:

- **fields**

  *fields* are FieldList objects representing fields passed through the receiving stream - either read from data source (DataSource.rows()) or written to data target (DataTarget.append()).

Subclasses should populate the fields property (or implement an accessor).

The subclasses might override:

- **initialize()**
• finalize()

The class supports context management, for example:

```python
with ds.CSVDataSource("output.csv") as s:
    for row in s.rows():
        print row
```

In this case, the initialize() and finalize() methods are called automatically.

finalize()

Subclasses might put finalisation code here, for example:

- closing a file stream
- sending data over network
- writing a chart image to a file

Default implementation does nothing.

initialize()

Delayed stream initialisation code. Subclasses might override this method to implement file or handle opening, connecting to a database, doing web authentication, ... By default this method does nothing.

The method does not take any arguments, it expects pre-configured object.

class brewery.ds.DataSource
Abstrac class for data sources.

read_fields(limit=0, collapse=False)

Read field descriptions from data source. You should use this for datasets that do not provide metadata directly, such as CSV files, document bases databases or directories with structured files. Does nothing in relational databases, as fields are represented by table columns and table metadata can obtained from database easily.

Note that this method can be quite costly, as by default all records within dataset are read and analysed.

After executing this method, stream fields is set to the newly read field list and may be configured (set more appropriate data types for example).

Arguments

- limit: read only specified number of records from dataset to guess field properties
- collapse: whether records are collapsed into flat structure or not

Returns: tuple with Field objects. Order of fields is datastore adapter specific.

records()

Return iterable object with dict objects. This is one of two methods for reading from data source. Subclasses should implement this method.

rows()

Return iterable object with tuples. This is one of two methods for reading from data source. Subclasses should implement this method.

class brewery.ds.DataTarget
Abstrac class for data targets.

append(object)

Append an object into dataset. Object can be a tuple, array or a dict object. If tuple or array is used, then value position should correspond to field position in the field list, if dict is used, the keys should be valid field names.
4.5 Sources

```python
class brewery.ds.CSVDataSource(resource, read_header=True, dialect=None, encoding=None, detect_header=False, sample_size=200, skip_rows=None, empty_as_null=True, fields=None, **reader_args)
```

Creates a CSV data source stream.

**Attributes**

- `resource`: file name, URL or a file handle with CSV data
- `read_header`: flag determining whether first line contains header or not. `True` by default.
- `encoding`: source character encoding, by default no conversion is performed.
- `detect_headers`: try to determine whether data source has headers in first row or not
- `sample_size`: maximum bytes to be read when detecting encoding and headers in file. By default it is set to 200 bytes to prevent loading huge CSV files at once.
- `skip_rows`: number of rows to be skipped. Default: `None`
- `empty_as_null`: treat empty strings as `Null` values

**Note:** avoid auto-detection when you are reading from remote URL stream.

```python
class brewery.ds.GoogleSpreadsheetDataSource(spreadsheet_key=None, spreadsheet_name=None, worksheet_id=None, worksheet_name=None, query_string='', username=None, password=None)
```

Creates a Google Spreadsheet data source stream.

**Attributes**

- `spreadsheet_key`: The unique key for the spreadsheet, this usually in the the form `‘pk23...We’ or ‘o23...423.12,,3’`.
- `spreadsheet_name`: The title of the spreadsheets.
- `worksheet_id`: ID of a worksheet
- `worksheet_name`: name of a worksheet
- `query_string`: optional query string for row selection
- `username`: Google account user name
- `password`: Google account password
You should provide either spreadsheet_key or spreadsheet_name, if more than one spreadsheet with given name are found, then the first in list returned by Google is used.

For worksheet selection you should provide either worksheet_id or worksheet_name. If more than one worksheet with given name are found, then the first in list returned by Google is used. If no worksheet_id nor worksheet_name are provided, then first worksheet in the workbook is used.

For details on query string syntax see the section on sq under http://code.google.com/apis/spreadsheets/reference.html#list_Parameters

initialize()
Connect to the Google documents, authenticate.

class brewery.ds.XLSDataSource(resource, sheet=None, encoding=None, skip_rows=None, read_header=True)
Creates a XLS spreadsheet data source stream.

Attributes
- resource: file name, URL or file-like object
- sheet: sheet index number (as int) or sheet name (as str)
- read_header: flag determining whether first line contains header or not. True by default.

initialize()
Initialize XLS source stream:

class brewery.ds.SQLDataSource(connection=None, url=None, table=None, statement=None, schema=None, autoinit=True, **options)
Creates a relational database data source stream.

Attributes
- url: SQLAlchemy URL - either this or connection should be specified
- connection: SQLAlchemy database connection - either this or url should be specified
- table: table name
- statement: SQL statement to be used as a data source (not supported yet)
- autoinit: initialize on creation, no explicit initialize() is needed
- options: SQL alchemy connect() options

initialize()
Initialize source stream. If the fields are not initialized, then they are read from the table.

class brewery.ds.MongoDBDataSource(collection, database=None, host=None, port=None, expand=False, **mongo_args)
Creates a MongoDB data source stream.

Attributes
- collection: mongo collection name
- database: database name
- host: mongo database server host, default is localhost
- port: mongo port, default is 27017
- expand: expand dictionary values and treat children as top-level keys with dot ‘.’ separated key path to the child.
initialize()

Initialize Mongo source stream:

class brewery.ds.YamlDirectoryDataSource(path, extension='yml', expand=False, filename_field=None)

Creates a YAML directory data source stream.

The data source reads files from a directory and treats each file as single record. For example, following directory will contain 3 records:

data/
  contract_0.yml
  contract_1.yml
  contract_2.yml

Optionally one can specify a field where file name will be stored.

Attributes

- path: directory with YAML files
- extension: file extension to look for, default is yml, if none is given, then all regular files in the directory are read
- expand: expand dictionary values and treat children as top-level keys with dot ‘.’
  separated key path to the child. Default: False
- filename_field: if present, then filename is streamed in a field with given name, or if record is requested, then filename will be in first field.

4.6 Targets

class brewery.ds.CSVDataTarget(resource, write_headers=True, truncate=True, encoding='utf-8', dialect=None, fields=None, **kwds)

Creates a CSV data target

Attributes

- resource: target object - might be a filename or file-like object
- write_headers: write field names as headers into output file
- truncate: remove data from file before writing, default: True

class brewery.ds.SQLDataTarget(connection=None, url=None, table=None, schema=None, truncate=False, create=False, replace=False, add_id_key=False, id_key_name=None, buffer_size=None, fields=None, concrete_type_map=None, **options)

Creates a relational database data target stream.

Attributes

- url: SQLAlchemy URL - either this or connection should be specified
- connection: SQLAlchemy database connection - either this or url should be specified
- table: table name
- truncate: whether truncate table or not
- create: whether create table on initialize() or not
• replace: Set to True if creation should replace existing table or not, otherwise initialization will fail on attempt to create a table which already exists.

• options: other SQLAlchemy connect() options

• add_id_key: whether to add auto-increment key column or not. Works only if create is True

• id_key_name: name of the auto-increment key. Default is ‘id’

• buffer_size: size of INSERT buffer - how many records are collected before they are inserted using multi-insert statement. Default is 1000

• fields : fieldlist for a new table

Note: avoid auto-detection when you are reading from remote URL stream.

finalize()
Closes the stream, flushes buffered data

initialize()
Initialize source stream:

class brewery.ds.MongoDBDataTarget (collection, database=None, host=None, port=None, truncate=False, expand=False, **mongo_args)

Creates a MongoDB data target stream.

Attributes
• collection: mongo collection name
• database: database name
• host: mongo database server host, default is localhost
• port: mongo port, default is 27017
• expand: expand dictionary values and treat children as top-level keys with dot ‘.’ separated key path to the child..
• truncate: delete existing data in the collection. Default: False

initialize()
Initialize Mongo source stream:

class brewery.ds.YamlDirectoryDataTarget (path, filename_template='record_${__index}.yml', expand=False, filename_start_index=0, truncate=False)

Creates a directory data target with YAML files as records.

Attributes
• path: directory with YAML files
• extension: file extension to use
• expand: expand dictionary values and treat children as top-level keys with dot ‘.’ separated key path to the child. Default: False
• filename_template: template string used for creating file names. ${key} is replaced with record value for key. __index is used for auto-generated file index from filename_start_index. Default filename template is record_${__index}.yml which results in filenames record_0.yml, record_1.yml, ...
• filename_start_index - first value of __index filename template value, by default 0
• filename_field: if present, then filename is taken from that field.
• truncate: remove all existing files in the directory. Default is False.

class brewery.ds.StreamAuditor\(\text{distinct\_threshold}=10\)

Target stream for auditing data values from stream. For more information about probed value properties, please refer to brewery.dq.FieldStatistics

append\((\text{obj})\)

Probe row or record and update statistics.

field_statistics

Return field statistics as dictionary: keys are field names, values are \text{brewery.dq.FieldStatistics} objects

class brewery.ds.SimpleHTMLDataTarget\(\text{resource, html\_header=True, html\_footer=None, write\_headers=True, table\_attributes=None}\)

Creates a HTML data target with simple naive HTML generation. No package that generates document node tree is used, just plain string concatenation.

Attributes

• resource: target object - might be a filename or file-like object - you can stream HTML table data into existing opened file.

• write_headers: create table headers, default: True. Field labels will be used, if field has no label, then field name will be used.

• table_attributes: \text{<table>} node attributes such as class, id, ...

• html_header: string to be used as HTML header. If set to None only \text{<table>} will be generated. If set to True then default header is used. Default is True.

• html_header: string to be used as HTML footer. Works in similar way as to html_header.

Note: No HTML escaping is done. HTML tags in data might break the output.
CHAPTER
FIVE

STREAMS — DATA ANALYSIS AND PROCESSING STREAMS

The data processing stream is a network of data processing nodes connected by data pipes. There are several data processing node types:

- **source nodes** - provide data from data sources such as CSV files or database tables
- **target nodes** - nodes for consuming data and storing them or creating data visualizations
- **record nodes** - perform operations on whole records, such as merging, joining, aggregations
- **field nodes** - perform operations on particular fields, such as text substitution, field renaming, deriving new fields, restructuring

5.1 Data Processing Streams

Figure 5.1: Example of a processing stream:

- load YAML fields from a directory - each file represents one record
- Strip all string fields.
- Remove duplicates and store unique records in a SQL database table
- Perform a data audit and pretty-print output using a formatted text printer

Constructing a stream the “object way”:

```python
from brewery.nodes import *
from brewery.streams import *
```
import brewery.metadata as metadata

# Prepare nodes

nodes = {
    "source": YamlDirectorySourceNode(path = "data/donations"),
    "strip": StringStripNode(),
    "distinct": DistinctNode(keys = ["year", "receiver", "project"],
    "target": SQLTableTarget(url = "postgres://localhost/data", table = "donations"),
    "audit": AuditNode(),
    "print": FormattedPrinterNode(output = "audit.txt")
}

# Configure nodes

nodes["source"].fields = metadata.FieldList(["year", "integer"],
("receiver", "string"),
("project", "string"),
("requested_amount", "float"),
("received_amount", "float"),
("source_comment", "string")])

nodes["print"].header = u"field nulls empty

-----------------------------------------------"

nodes["print"].format = u"{field_name:<30.30} {null_record_ratio:3.2%} {empty_string_count:>10}"

connections = [ ("source", "strip"),
    ("strip", "distinct"),
    ("distinct", "target"),
    ("strip", "audit"),
    ("audit", "print")
]

# Create and run stream

stream = Stream(nodes, connections)
stream.run()

The created audit.txt file will contain:

field nulls empty
-----------------------------------------------
year 0.00% 0
receiver 0.00% 5
project 0.51% 0
requested_amount 0.70% 0
received_amount 6.40% 0
source_comment 99.97% 0

The core class is Stream:

class brewery.streams.Stream(nodes=nodes, connections=connections)
    Creates a data stream.

Parameters

- nodes - dictionary with keys as node names and values as nodes
- connections - list of two-item tuples. Each tuple contains source and target node or source
and target node name.

- `stream` - another stream or

**configure (config=None)**
Configure node properties based on configuration. Only named nodes can be configured at the moment.

`config` is a list of dictionaries with keys: `node` - node name, `parameter` - node parameter name, `value` - parameter value

**fork ()**
Creates a construction fork of the stream. Used for constructing streams in functional fashion. Example:

```python
stream = Stream()

fork = stream.fork()
fork.csv_source("fork.csv")
fork.formatted_printer()

stream.run()
```

Fork responds to node names as functions. The function arguments are the same as node constructor (`__init__` method) arguments. Each call will append new node to the fork and will connect the new node to the previous node in the fork.

To configure current node you can use `fork.node`, like:

```python
fork.csv_source("fork.csv")
fork.node.read_header = True
```

To set actual node name use `set_name()`:

```python
fork.csv_source("fork.csv")
fork.set_name("source")
```

```python
...
```

```python
source_node = stream.node("source")
```

To fork a fork, just call `fork()`

**run ()**
Run all nodes in the stream.

Each node is being wrapped and run in a separate thread.

When an exception occurs, the stream is stopped and all caught exceptions are stored in attribute `exceptions`.

**update (nodes=None, connections=None)**
Adds nodes and connections specified in the dictionary. Dictionary might contain node names instead of real classes. You can use this method for creating stream from a dictionary that was created from a JSON file, for example.

The stream is constructed using nodes. For more information about nodes see *Node Reference*.

## 5.2 Running Streams

Streams are being run using `Stream.run ()`. The stream nodes are executed in parallel - each node is run in separate thread.
Stream raises `StreamError` if there are issues with the network before or during initialization and finalization phases. When the stream is run and something happens, then `StreamRuntimeError` is raised which contains more detailed information:

```python
class brewery.streams.StreamRuntimeError (message=None, node=None, exception=None)
    Exception raised when a node fails during `run()` phase.

    Attributes:
       • message: exception message
       • node: node where exception was raised
       • exception: exception that was raised while running the node
       • traceback: stack traceback
       • inputs: array of field lists for each input
       • output: output field list

    print_exception (output=None)
    Prints exception and details in human readable form. You can specify IO stream object in `output` parameter.
    By default text is printed to standard output.
```

Preferred way of running the stream in manually written scripts is:

```python
try:
    stream.run()
except brewery.streams.StreamRuntimeError as e:
    e.print_exception()
```

### 5.3 Forking Forks with Higher Order Messaging

There is another way of constructing streams which uses “higher order messaging”. It means, that instead of constructing the stream from nodes and connections, you pretend to “call” functions that process your data. In fact the function call is interpreted as step in processing stream construction.

```python
trunk.csv_source("data.csv")
trunk.sample(1000)
trunk.aggregate(keys = ["year"])
trunk.formatted_printer(...) 
```

Executing the functions as written might be very expensive in terms of time and memory. What is in fact happening is that instead of executing the data processing functions a stream network is being constructed and the construction is being done by using forked branches. To start, an empty stream and first fork has to be created:

```python
from brewery.streams import *

stream = Stream()
main = stream.fork()
...
```

Now we have fork `main`. Each function call on `main` will append a new processing node to the fork and the new node will be connected to the previous node of the fork.
Function names are based on node names in most of the cases. There might be custom function names for some nodes in the future, but now there is just simple rule:

1. de-camelize node name: CSVSourceNode to csv source node
2. replace spaces with underscores: csv_source_node
3. remove ‘node’ suffix: csv_source

Arguments to the function are the same as arguments for node constructor. If you want to do more node configuration you can access current node with node attribute of the fork:

```python
main.node.keys = ["country"]
```

Run the stream as if it was constructed manually from nodes and connections:

```python
stream.run()
```

There are plenty of situations where linear processing is not sufficient and we will need to have branches. To create another branch, we fork() a fork. For example, to attach a data audit to the stream insert following code right after the node we want to audit:

```python
# we are in main at node after which we want to have multiple branches
audit = trunk.fork()
audit.audit()
audit.value_threshold(...)
audit.formatted_printer(...)  
# continue main.* branch here...
```

### 5.3.1 Example

```python
from brewery.streams import Stream
from brewery.metadata import FieldList

stream = Stream()

a_list = [
    {"i": 1, "name": "apple"},
    {"i": 2, "name": "banana"},
]```
fields = FieldList(["i", "name"])
trunk = stream.fork()
trunk.record_list_source(a_list = a_list, fields = fields)
trunk.derive("i*100 + len(name)")
csv_branch = trunk.fork()
trunk.record_list_target()
record_target = trunk.node
csv_branch.csv_target("test_stream.csv")

stream.run()

for record in record_target.records:
    print record

Output will be:

{'i': 1, 'name': 'apple', 'new_field': 105}
{'i': 2, 'name': 'banana', 'new_field': 207}
{'i': 3, 'name': 'orange', 'new_field': 306}

The newly created test_stream.csv file will contain:

```
i,name,new_field
1,apple,105
2,banana,207
3,orange,306
```

### 5.4 Custom nodes

To implement custom node, one has to subclass the Node class:

```python
class brewery.streams.Node
    Creates a new data processing node.

    Attributes
    • inputs: input pipes
    • outputs: output pipes
    • description: custom node annotation

    configure(config, protected=False)
    Configure node.

    Parameters
    • config - a dictionary containing node attributes as keys and values as attribute values. Key type is ignored as it is used for node creation.
    • protected - if set to True only non-protected attributes are set. Attempt to set protected attribute will result in an exception. Use protected when you are configuring nodes through a user interface or a custom tool. Default is False: all attributes can be set.

    If key in the config dictionary does not refer to a node attribute specified in node description, then it is ignored.
```
finalize()  
Finalizes the node. Default implementation does nothing.

classmethod identifier()  
Returns an identifier name of the node class. Identifier is used for construction of streams from dictionaries or for any other out-of-program constructions.

Node identifier is specified in the node_info dictionary as name. If no explicit identifier is specified, then decamelized class name will be used with node suffix removed. For example: CSVSourceNode will be csv_source.

initialize()  
Initializes the node. Initialization is separated from creation. Put any Node subclass initialization in this method. Default implementation does nothing.

input  
Return single node input if exists. Convenience property for nodes which process only one input. Raises exception if there are no inputs or are more than one input.

input_fields  
Return fields from input pipe, if there is one and only one input pipe.

output_field_names  
Convenience method for getting names of fields generated by the node. For more information see brewery.nodes.Node.output_fields()

output_fields  
Return fields passed to the output by the node.

Subclasses should override this method. Default implementation returns same fields as input has, raises exception when there are more inputs or if there is no input connected.

put(obj)  
Put row into all output pipes.

Raises NodeFinished exception when node’s target nodes are not receiving data anymore. In most cases this exception might be ignored, as it is handled in the node thread wrapper. If you want to perform necessary clean-up in the run() method before exiting, you should handle this exception and then re-raise it or just simply return from run().

This method can be called only from node’s run() method. Do not call it from initialize() or finalize().

put_record(obj)  
Put record into all output pipes. Convenience method. Not recommended to be used.

Depreciated.

reset_type(name)  
Remove all retype information for field name

retype(name, **attributes)  
Retype an output field name to field field.

run()  
Main method for running the node code. Subclasses should implement this method.

Node uses pipes for communication. SimplePipe is abstract class that should be used as base class for any Pipe implementation:

The Pipe class uses Python threading for node thread concurrency:

class brewery.streams.Pipe(buffer_size=1000)  
Creates uni-directional data pipe for passing data between two threads in batches of size buffer_size.
If receiving node is finished with source data and does not want anything any more, it should send `done_receiving()` to the pipe. In most cases, stream runner will send `done_receiving()` to all input pipes when node’s `run()` method is finished.

If sending node is finished, it should send `done_sending()` to the pipe, however this is not necessary in most cases, as the method for running stream flushes outputs automatically on when node `run()` method is finished.

### closed()
Return `True` if pipe is closed - not sending or not receiving data any more.

### done_receiving()
Close pipe from either side

### done_sending()
Close pipe from sender side

### put(obj)
Put data object into the pipe buffer. When buffer is full it is enqueued and receiving node can get all buffered data objects.

Puttin object into pipe is not thread safe. Only one thread sohuld write to the pipe.

### rows()
Get data object from pipe. If there is no buffer ready, wait until source object sends some data.
6.1 Sources

6.1.1 CSV Source

**Synopsis:** Read data from a comma separated values (CSV) file.

**Identifier:** csv_source (class: brewery.nodes.CSVSourceNode)

Source node that reads comma separated file from a filesystem or a remote URL.

It is recommended to configure node fields before running. If you do not do so, fields are read from the file header if specified by `read_header` flag. Field storage types are set to `string` and analytical type is set to `typeless`.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>File name or URL containing comma separated values</td>
</tr>
<tr>
<td>fields</td>
<td>fields contained in the file</td>
</tr>
<tr>
<td>read_header</td>
<td>flag determining whether first line contains header or not</td>
</tr>
<tr>
<td>skip_rows</td>
<td>number of rows to be skipped</td>
</tr>
<tr>
<td>encoding</td>
<td>resource data encoding, by default no conversion is performed</td>
</tr>
<tr>
<td>delimiter</td>
<td>record delimiter character, default is comma ','</td>
</tr>
<tr>
<td>quotechar</td>
<td>character used for quoting string values, default is double quote</td>
</tr>
</tbody>
</table>

6.1.2 ElasticSearch Source
**Synopsis:** Read data from ElasticSearch engine

**Identifier:** es_source (class: brewery.nodes.ESSourceNode)

Source node that reads from an ElasticSearch index.

See ElasticSearch home page for more information: [http://www.elasticsearch.org/](http://www.elasticsearch.org/)

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>document_type</td>
<td>ElasticSearch document type name</td>
</tr>
<tr>
<td>expand</td>
<td>expand dictionary values and treat children as top-level keys with dot &quot;.&quot; separated key path to the child</td>
</tr>
<tr>
<td>database</td>
<td>database name</td>
</tr>
<tr>
<td>host</td>
<td>database server host, default is localhost</td>
</tr>
<tr>
<td>port</td>
<td>database server port, default is 27017</td>
</tr>
</tbody>
</table>

### 6.1.3 Callable Generator Source

**Synopsis:** Uses a callable as record generator

**Identifier:** generator_function_source (class: brewery.nodes.GeneratorFunctionSourceNode)

Source node uses a callable to generate records.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>Function (or any callable)</td>
</tr>
<tr>
<td>fields</td>
<td>Fields the function generates</td>
</tr>
<tr>
<td>args</td>
<td>Function arguments</td>
</tr>
<tr>
<td>kwargs</td>
<td>Function key-value arguments</td>
</tr>
</tbody>
</table>

### 6.1.4 Google Spreadsheet Source

**Synopsis:** Read data from a Google Spreadsheet.

**Identifier:** google_spreadsheet_source (class: brewery.nodes.GoogleSpreadsheetSourceNode)

Source node that reads Google Spreadsheet.
You should provide either spreadsheet_key or spreadsheet_name, if more than one spreadsheet with given name are found, then the first in list returned by Google is used.

For worksheet selection you should provide either worksheet_id or worksheet_name. If more than one worksheet with given name are found, then the first in list returned by Google is used. If no worksheet_id nor worksheet_name are provided, then first worksheet in the workbook is used.

For details on query string syntax see the section on sq under http://code.google.com/apis/spreadsheets/reference.html#list_Parameters

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spreadsheet_key</td>
<td>The unique key for the spreadsheet</td>
</tr>
<tr>
<td>spreadsheet_name</td>
<td>The title of the spreadsheets</td>
</tr>
<tr>
<td>worksheet_id</td>
<td>ID of a worksheet</td>
</tr>
<tr>
<td>worksheet_name</td>
<td>name of a worksheet</td>
</tr>
<tr>
<td>query_string</td>
<td>optional query string for row selection</td>
</tr>
<tr>
<td>username</td>
<td>Google account user name</td>
</tr>
<tr>
<td>password</td>
<td>Google account password</td>
</tr>
</tbody>
</table>

6.1.5 Record List Source

Synopsis: Provide list of dict objects as data source.

Identifier: record_list_source (class: brewery.nodes.RecordListSourceNode)

Source node that feeds records (dictionary objects) from a list (or any other iterable) object.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_list</td>
<td>List of records represented as dictionaries.</td>
</tr>
<tr>
<td>fields</td>
<td>Fields in the list.</td>
</tr>
</tbody>
</table>

6.1.6 Row List Source

Synopsis: Provide list of lists or tuples as data source.

Identifier: row_list_source (class: brewery.nodes.RowListSourceNode)

Source node that feeds rows (list/tuple of values) from a list (or any other iterable) object.
Table 6.6: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>List of rows represented as lists or tuples.</td>
</tr>
<tr>
<td>fields</td>
<td>Fields in the list.</td>
</tr>
</tbody>
</table>

### 6.1.7 SQL Source

**Synopsis:** Read data from a sql table.

**Identifier:** sql_source (class: brewery.nodes.SQLSourceNode)

Source node that reads from a sql table.

Table 6.7: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>SQLAlchemy URL</td>
</tr>
<tr>
<td>table</td>
<td>table name</td>
</tr>
</tbody>
</table>

### 6.1.8 Data Stream Source

**Synopsis:** Generic data stream data source node.

**Identifier:** stream_source (class: brewery.nodes.StreamSourceNode)

Generic data stream source. Wraps a brewery.ds data source and feeds data to the output.

The source data stream should configure fields on initialize().

Note that this node is only for programatically created processing streams. Not useable in visual, web or other stream modelling tools.

Table 6.8: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>Data stream object.</td>
</tr>
</tbody>
</table>
6.1.9 XLS Source

**Synopsis:** Read data from an Excel (XLS) spreadsheet file.

**Identifier:** xls_source (class: brewery.nodes.XLSSourceNode)

Source node that reads Excel XLS files.

It is recommended to configure node fields before running. If you do not do so, fields are read from the file header if specified by `read_header` flag. Field storage types are set to `string` and analytical type is set to `typeless`.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>File name or URL containing comma separated values</td>
</tr>
<tr>
<td>fields</td>
<td>fields contained in the file</td>
</tr>
<tr>
<td>sheet</td>
<td>Sheet index number (as int) or sheet name (as string)</td>
</tr>
<tr>
<td>read_header</td>
<td>flag determining whether first line contains header or not</td>
</tr>
<tr>
<td>skip_rows</td>
<td>number of rows to be skipped</td>
</tr>
<tr>
<td>encoding</td>
<td>resource data encoding, by default no conversion is performed</td>
</tr>
</tbody>
</table>

6.1.10 YAML Directory Source

**Synopsis:** Read data from a directory containing YAML files

**Identifier:** yaml_directory_source (class: brewery.nodes.YamlDirectorySourceNode)

Source node that reads data from a directory containing YAML files.

The data source reads files from a directory and treats each file as single record. For example, following directory will contain 3 records:

```
data/
  contract_0.yml
  contract_1.yml
  contract_2.yml```

Optionally one can specify a field where file name will be stored.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>Path to a directory</td>
</tr>
<tr>
<td>extension</td>
<td>file extension to look for, default is yml. If none is given, then all regular files in the directory are read.</td>
</tr>
<tr>
<td>filename_field</td>
<td>name of a new field that will contain file name</td>
</tr>
</tbody>
</table>
6.2 Record Operations

6.2.1 Aggregate Node

Synopsis: Aggregate values grouping by key fields.

Identifier: aggregate (class: brewery.nodes.AggregateNode)

Table 6.11: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys</td>
<td>List of fields according to which records are grouped</td>
</tr>
<tr>
<td>record_count_field</td>
<td>Name of a field where record count will be stored. Default is record_count</td>
</tr>
<tr>
<td>measures</td>
<td>List of fields to be aggregated.</td>
</tr>
</tbody>
</table>

6.2.2 Append

Synopsis: Concatenate input streams.

Identifier: append (class: brewery.nodes.AppendNode)

Sequentially append input streams. Concatenation order reflects input stream order. The input streams should have same set of fields.

6.2.3 Data Audit

Synopsis: Perform basic data audit.

Identifier: audit (class: brewery.nodes.AuditNode)

Node checks stream for empty strings, not filled values, number distinct values.
Audit note passes following fields to the output:

- **field_name** - name of a field from input
- **record_count** - number of records
- **null_count** - number of records with null value for the field
- **null_record_ratio** - ratio of null count to number of records
- **empty_string_count** - number of strings that are empty (for fields of type string)
- **distinct_count** - number of distinct values (if less than distinct threshold). Set to None if there are more distinct values than **distinct_threshold**.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinct_threshold</td>
<td>number of distinct values to be tested. If there are more than the threshold, then values are not included any more and result <strong>distinct_values</strong> is set to None</td>
</tr>
</tbody>
</table>

### 6.2.4 Derive Node

**Synopsis:** *Derive a new field using an expression.*

**Identifier:** derive (class: brewery.nodes.DeriveNode)

Derive a new field from other fields using an expression or callable function.

The parameter names of the callable function should reflect names of the fields:

```python
def get_half(i, **args):
    return i / 2
```

`node.formula = get_half`

You can use `**record` to catch all or rest of the fields as dictionary:

```python
def get_half(**record):
    return record["i"] / 2
```

`node.formula = get_half`

The formula can be also a string with python expression where local variables are record field values:

`node.formula = "i / 2"`

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field_name</td>
<td>Derived field name</td>
</tr>
<tr>
<td>formula</td>
<td>Callable or a string with python expression that will evaluate to new field value</td>
</tr>
<tr>
<td>analytical_type</td>
<td>Analytical type of the new field</td>
</tr>
<tr>
<td>storage_type</td>
<td>Storage type of the new field</td>
</tr>
</tbody>
</table>
6.2.5 Distinct Node

**Synopsis:** Pass only distinct records (discard duplicates) or pass only duplicates

**Identifier:** distinct (class: brewery.nodes.DistinctNode)

Node will pass distinct records with given distinct fields.

If `discard` is False then first record with distinct keys is passed to the output. This is used to find all distinct key values.

If `discard` is True then first record with distinct keys is discarded and all duplicate records with same key values are passed to the output. This mode is used to find duplicate records. For example: there should be only one invoice per organisation per month. Set `distinct_fields` to `organisation` and `month`, set `discard` to True. Running this node should give no records on output if there are no duplicates.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distinct_fields</td>
<td>List of key fields that will be considered when comparing records</td>
</tr>
<tr>
<td>discard</td>
<td>Field where substitution result will be stored. If not set, then original field will be replaced with new value.</td>
</tr>
</tbody>
</table>

6.2.6 Function Select

**Synopsis:** Select records by a predicate function (python callable).

**Identifier:** function_select (class: brewery.nodes.FunctionSelectNode)

Select records that will be selected by a predicate function.

Example: configure a node that will select records where `amount` field is greater than 100

```python
def select_greater_than(value, threshold):
    return value > threshold

node.function = select_greater_than
node.fields = ["amount"]
node.kwargs = {"threshold": 100}
```

The `discard` flag controls behaviour of the node: if set to True, then selection is inversed and fields that function evaluates as True are discarded. Default is False - selected records are passed to the output.
### 6.2.7 Merge Node

**Synopsis:** Merge two or more streams

**Identifier:** `merge` (class: `brewery.nodes.MergeNode`)

Merge two or more streams (join).

Inputs are joined in a star-like fashion: one input is considered master and others are details adding information to the master. By default master is the first input. Joins are specified as list of tuples: `(input_tag, master_input_key, other_input_key)`.

Following configuration code shows how to add region and category details:

```python
code = [1, "region_code", "code"],
[2, "category_code", "code"]
```

Master input should have fields `region_code` and `category_code`, other inputs should have `code` field with respective values equal to master keys.

```python
code = [1, "region_code", "code"],
[2, ("category_code", "year"), ("code", "year")]
```

As a key you might use either name of a single field or list of fields for compound keys. If you use compound key, both keys should have same number of fields. For example, if there is categorisation based on year:

The detail key might be omitted if it the same as in master input:

```python
code = [1, "region_code"],
[2, "category_code"]
```

Master input should have fields `region_code` and `category_code`, input #1 should have `region_code` field and input #2 should have `category_code` field.

To filter-out fields you do not want in your output or to rename fields you can use `maps`. It should be a dictionary where keys are input tags and values are either `FieldMap` objects or dictionaries with keys `rename` and `drop`.

Following example renames `source_region_name` field in input 0 and drops field `id` in input 1:

```python
node.maps = {
    0: FieldMap(rename = ["source_region_name":"region_name"],
    1: FieldMap(drop = ["id"])
}
```
It is the same as:

```javascript
node.maps = {
    0: { "rename" : {"source_region_name":"region_name"} },
    1: { "drop" : ["id"] }
}
```

The first option is preferred, the dictionary based option is provided for convenience in cases nodes are being constructed from external description (such as JSON dictionary).

**Note:** Limitations of current implementation (might be improved in the future):

- only inner join between datasets: that means that only those input records are joined that will have matching keys
- “detail” datasets should have unique keys, otherwise the behaviour is undefined
- master is considered as the largest dataset

How does it work: all records from detail inputs are read first. Then records from master input are read and joined with cached input records. It is recommended that the master dataset set is the largest from all inputs.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>joins</td>
<td>Join specification (see node documentation)</td>
</tr>
<tr>
<td>master</td>
<td>Tag (index) of input dataset which will be considered as master</td>
</tr>
<tr>
<td>maps</td>
<td>Specification of which fields are passed from input and how they are going to be (re)named</td>
</tr>
<tr>
<td>join_types</td>
<td>Dictionary where keys are stream tags (indexes) and values are types of join for the stream. Default is ‘inner’. – <strong>Not implemented</strong></td>
</tr>
</tbody>
</table>

### 6.2.8 Sample Node

**Synopsis:** *Pass data sample from input to output.*

**Identifier:** sample (class: `brewery.nodes.SampleNode`)

Create a data sample from input stream. There are more sampling possibilities:

- fixed number of records
- number of records, random
- % of records, random
- get each n-th record (*not yet implemented*)

Node can work in two modes: pass sample to the output or discard sample and pass the rest. The mode is controlled through the `discard` flag. When it is false, then sample is passed and rest is discarded. When it is true, then sample is discarded and rest is passed.
There are currently three sampling methods: `method = "first"` takes the first `size` records, `method = "random"` selects `size` records at random from the entire pipe, whereas `method = "percent"` selects, in expectation, `size` percent of the stream. Note that `percent` selects each element with probability $p = \frac{size}{100}$, so the actual size of the sample will vary. If you need an exact number of elements in your sample, use the `random` method.

Random sampling is important if you want to get an overview of the entire dataset. The `random` and `percent` methods ensure that the sample will be representative of the dataset. The same is not true for `first`, because the records may have been ordered in the input stream, and problematic records may be under- or overrepresented in the first batch.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Size of the sample to be passed to the output</td>
</tr>
<tr>
<td>discard</td>
<td>flag whether the sample is discarded or included</td>
</tr>
<tr>
<td>method</td>
<td>&quot;first&quot; (default): take first <code>size</code> records, &quot;random&quot;: take <code>size</code> records at random, &quot;percent&quot;: take each record with <code>size/100</code> probability</td>
</tr>
</tbody>
</table>

### 6.2.9 Select

**Synopsis:** Select or discard records from the stream according to a predicate.

**Identifier:** `select` (class: `brewery.nodes.SelectNode`)

Select or discard records from the stream according to a predicate.

The parameter names of the callable function should reflect names of the fields:

```python
def is_big_enough(i, **args):
    return i > 1000000
```

```python
node.condition = is_big_enough
```

You can use `**record` to catch all or rest of the fields as dictionary:

```python
def is_big_enough(**record):
    return record["i"] > 1000000
```

```python
node.condition = is_big_enough
```

The condition can be also a string with python expression where local variables are record field values:

```python
node.condition = "i > 1000000"
```

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>Callable or a string with python expression that will evaluate to a boolean value</td>
</tr>
<tr>
<td>discard</td>
<td>flag whether the records matching condition are discarded or included</td>
</tr>
</tbody>
</table>
6.2.10 Set Select

**Synopsis:** Select records by a predicate function.

**Identifier:** set_select (class: brewery.nodes.SetSelectNode)

Select records where field value is from predefined set of values.

Use case examples:

- records from certain regions in *region* field
- records where *quality* status field is *low* or *medium*

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>Field to be tested.</td>
</tr>
<tr>
<td>value_set</td>
<td>set of values that will be used for record selection</td>
</tr>
<tr>
<td>discard</td>
<td>flag whether the selection is discarded or included</td>
</tr>
</tbody>
</table>

6.3 Field Operations

6.3.1 Binning

**Synopsis:** Derive a field based on binned values (histogram)

**Identifier:** binning (class: brewery.nodes.BinningNode)

Derive a bin/category field from a value.

**Warning:** Not yet implemented

Binning modes:

- fixed width (for example: by 100)
- fixed number of fixed-width bins
- n-tiles by count or by sum
- record rank
6.3.2 Coalesce Value To Type Node

**Synopsis:** Coalesce Value to Type

**Identifier:** coalesce_value_to_type (class: brewery.nodes.CoalesceValueToTypeNode)

Coalesce values of selected fields, or fields of given type to match the type.

- **string, text**
  - Strip strings
  - if non-string, then it is converted to a unicode string
  - Change empty strings to empty (null) values

- **float, integer**
  - If value is of string type, perform string cleansing first and then convert them to respective numbers or to null on failure

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields</td>
<td>List of fields to be cleansed. If none given then all fields of known storage type are cleansed</td>
</tr>
<tr>
<td>types</td>
<td>List of field types to be coalesced (if no fields given)</td>
</tr>
<tr>
<td>empty_values</td>
<td>dictionary of type -&gt; value pairs to be set when field is considered empty (null)</td>
</tr>
</tbody>
</table>

6.3.3 Field Map

**Synopsis:** Rename or drop fields from the stream.

**Identifier:** field_map (class: brewery.nodes.FieldMapNode)

Node renames input fields or drops them from the stream.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map_fields</td>
<td>Dictionary of input to output field name.</td>
</tr>
<tr>
<td>drop_fields</td>
<td>List of fields to be dropped from the stream - incompatible with keep_fields.</td>
</tr>
<tr>
<td>keep_fields</td>
<td>List of fields to keep from the stream - incompatible with drop_fields.</td>
</tr>
</tbody>
</table>
6.3.4 String Strip

Synopsis: Strip characters.

Identifier: string_strip (class: brewery.nodes.StringStripNode)

Strip spaces (or other specified characters) from string fields.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields</td>
<td>List of string fields to be stripped. If none specified, then all fields of storage type string are stripped</td>
</tr>
<tr>
<td>chars</td>
<td>Characters to be stripped. By default all white-space characters are stripped</td>
</tr>
</tbody>
</table>

6.3.5 Text Substitute

Synopsis: Substitute text in a field using regular expression.

Identifier: text_substitute (class: brewery.nodes.TextSubstituteNode)

Substitute text in a field using regular expression.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>Field containing a string or text value where substitution will be applied</td>
</tr>
<tr>
<td>derived_field</td>
<td>Field where substitution result will be stored. If not set, then original field will be replaced with new value.</td>
</tr>
<tr>
<td>substitutions</td>
<td>List of substitutions: each substitution is a two-element tuple (pattern, replacement) where pattern is a regular expression that will be replaced using replacement</td>
</tr>
</tbody>
</table>

6.3.6 Value Threshold
Synopsis: Bin values based on a threshold.

Identifier: value_threshold (class: brewery.nodes.ValueThresholdNode)

Create a field that will refer to a value bin based on threshold(s). Values of range type can be compared against one or two thresholds to get low/high or low/medium/high value bins.

Note: this node is not yet implemented

The result is stored in a separate field that will be constructed from source field name and prefix/suffix.

For example:

- amount < 100 is low
- 100 <= amount <= 1000 is medium
- amount > 1000 is high

Generated field will be amount_threshold and will contain one of three possible values: low, medium, high

Another possible use case might be for binning after data audit: we want to measure null record count and we set thresholds:

- ratio < 5% is ok
- 5% <= ratio <= 15% is fair
- ratio > 15% is bad

We set thresholds as (0.05, 0.15) and values to ("ok", "fair", "bad")

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thresholds</td>
<td>List of fields of range type and threshold tuples (field, low, high) or (field, low)</td>
</tr>
<tr>
<td>bin_names</td>
<td>Names of bins based on threshold. Default is low, medium, high</td>
</tr>
<tr>
<td>prefix</td>
<td>field prefix to be used, default is none.</td>
</tr>
<tr>
<td>suffix</td>
<td>field suffix to be used, default is '_bin'</td>
</tr>
</tbody>
</table>

### 6.4 Targets

#### 6.4.1 CSV Target

Synopsis: Write rows as comma separated values into a file

Identifier: csv_target (class: brewery.nodes.CSVTargetNode)

Node that writes rows into a comma separated values (CSV) file.

Attributes

- resource: target object - might be a filename or file-like object
- write_headers: write field names as headers into output file
• truncate: remove data from file before writing, default: True

Table 6.25: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>Target object - file name or IO object.</td>
</tr>
<tr>
<td>write_headers</td>
<td>Flag determining whether to write field names as file headers.</td>
</tr>
<tr>
<td>truncate</td>
<td>If set to True all data from file are removed. Default True</td>
</tr>
</tbody>
</table>

### 6.4.2 SQL Table Target

**Synopsis:** *Feed data rows into a relational database table*

**Identifier:** `sql_table_target (class: brewery.nodes.DatabaseTableTargetNode)`

Feed data rows into a relational database table.

Table 6.26: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>Database URL in form: adapter://user:password@host/database</td>
</tr>
<tr>
<td>connection</td>
<td>SQLAlchemy database connection - either this or url should be specified</td>
</tr>
<tr>
<td>table</td>
<td>table name</td>
</tr>
<tr>
<td>truncate</td>
<td>If set to True all data table are removed prior to node execution. Default is False - data are appended to the table</td>
</tr>
<tr>
<td>create</td>
<td>create table if it does not exist or not</td>
</tr>
<tr>
<td>replace</td>
<td>Set to True if creation should replace existing table or not, otherwise node will fail on attempt to create a table which already exists</td>
</tr>
<tr>
<td>buffer_size</td>
<td>how many records are collected before they are inserted using multi-insert statement. Default is 1000</td>
</tr>
<tr>
<td>options</td>
<td>other SQLAlchemy connect() options</td>
</tr>
</tbody>
</table>

### 6.4.3 Formatted Printer

**Synopsis:** *Print input using a string formatter to an output IO stream*

**Identifier:** `formatted_printer (class: brewery.nodes.FormattedPrinterNode)`

Target node that will print output based on format.

Refer to the python formatting guide:
Example:
Consider we have a data with information about donations. We want to pretty print two fields: `project` and `requested_amount` in the form:

```
Hlavicka - makovicka 27550.0
Obecna kniznica - symbol moderneho vzdelavania 132000.0
Vzdelavanie na europskej urovni 60000.0
```

Node for given format is created by:

```
node = FormattedPrinterNode(format = u"{project:<50.50} {requested_amount:>20}"
```

Following format can be used to print output from an audit node:

```
node.header = u"field nulls empty distinct\n" 
node.format = u"{field_name:<30.30} {null_record_ratio: >7.2%} \
"{empty_string_count:>10} {distinct_count:>10}"
```

Output will look similar to this:

```
field nulls empty distinct
------------------------------------------------------------
file 0.00% 0 32
source_code 0.00% 0 2
id 9.96% 0 907
receiver_name 9.10% 0 1950
project 0.05% 0 3628
requested_amount 22.90% 0 924
received_amount 4.98% 0 728
source_comment 99.98% 0 2
```

Table 6.27: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>Format string to be used. Default is to print all field values separated by tab character.</td>
</tr>
<tr>
<td>target</td>
<td>IO object. If not set then sys.stdout will be used. If it is a string, then it is considered a filename.</td>
</tr>
<tr>
<td>delimiter</td>
<td>Record delimiter. By default it is new line character.</td>
</tr>
<tr>
<td>header</td>
<td>Header string - will be printed before printing first record</td>
</tr>
<tr>
<td>footer</td>
<td>Footer string - will be printed after all records are printed</td>
</tr>
</tbody>
</table>

### 6.4.4 Pretty Printer

**Synopsis:** Print input using a pretty formatter to an output IO stream

**Identifier:** pretty_printer (class: brewery.nodes.PrettyPrinterNode)

Target node that will pretty print output as a table.
Table 6.28: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>IO object. If not set then sys.stdout will be used. If it is a string, then it is considered a filename.</td>
</tr>
<tr>
<td>max_column_width</td>
<td>Maximum column width. Default is unlimited. If set to None, then it is unlimited.</td>
</tr>
<tr>
<td>min_column_width</td>
<td>Minimum column width. Default is 0 characters.</td>
</tr>
</tbody>
</table>

6.4.5 Record List Target

**Synopsis:** *Store data as list of dictionaries (records)*

**Identifier:** `record_list_target` (class: `brewery.nodes.RecordListTargetNode`)

Target node that stores data from input in a list of records (dictionary objects) object.

To get list of fields, ask for `output_fields`.

Table 6.29: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>records</td>
<td>Created list of records represented as dictionaries.</td>
</tr>
</tbody>
</table>

6.4.6 Row List Target

**Synopsis:** *Store data as list of tuples*

**Identifier:** `row_list_target` (class: `brewery.nodes.RowListTargetNode`)

Target node that stores data from input in a list of rows (as tuples).

To get list of fields, ask for `output_fields`.

Table 6.30: Attributes

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rows</td>
<td>Created list of tuples.</td>
</tr>
</tbody>
</table>
6.4.7 SQL Table Target

**Synopsis:** Feed data rows into a relational database table

**Identifier:** sql_table_target (class: brewery.nodes.SQLTableTargetNode)

Feed data rows into a relational database table.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>Database URL in form: adapter://user:password@host/database</td>
</tr>
<tr>
<td>connection</td>
<td>SQLAlchemy database connection - either this or url should be specified</td>
</tr>
<tr>
<td>table</td>
<td>table name</td>
</tr>
<tr>
<td>truncate</td>
<td>If set to True all data table are removed prior to node execution. Default is False - data are appended to the table</td>
</tr>
<tr>
<td>create</td>
<td>create table if it does not exist or not</td>
</tr>
<tr>
<td>replace</td>
<td>Set to True if creation should replace existing table or not, otherwise node will fail on attempt to create a table which already exists</td>
</tr>
<tr>
<td>buffer_size</td>
<td>how many records are collected before they are inserted using multi-insert statement. Default is 1000</td>
</tr>
<tr>
<td>options</td>
<td>other SQLAlchemy connect() options</td>
</tr>
</tbody>
</table>

6.4.8 Data Stream Target

**Synopsis:** Generic data stream data target node.

**Identifier:** stream_target (class: brewery.nodes.StreamTargetNode)

Generic data stream target. Wraps a brewery.ds data target and feeds data from the input to the target stream.

The data target should match stream fields.

Note that this node is only for programatically created processing streams. Not useable in visual, web or other stream modelling tools.

<table>
<thead>
<tr>
<th>attribute</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>Data target object.</td>
</tr>
</tbody>
</table>
class brewery.probes.MissingValuesProbe
Data quality statistics for a dataset field

    Attributes
    • count: total count of null records

    probe(value)
    Probe the value.

class brewery.probes.StatisticsProbe
Data quality statistics for a dataset field

    Attributes
    • min - minimum value found
    • max - maximum value found
    • sum - sum of values
    • count - count of values
    • average - average value

class brewery.probes.DistinctProbe(threshold=None)
Probe for distinct values.

class brewery.probes.StorageTypeProbe
Probe for guessing field data type

    Attributes:
    • field: name of a field which statistics are being presented
    • storage_types: found storage types
    • unique_storage_type: if there is only one storage type, then this is set to that type

    unique_storage_type
    Return storage type if there is only one. This should always return a type in relational databases, but does not have to in databases such as MongoDB.
Functions and classes for measuring data quality.

Example of auditing a CSV file:

```python
from brewery import ds
from brewery import dq

# Open a data stream
src = ds.CSVDataSource("data.csv")
src.initialize()

# Prepare field statistics
stats = {}
fields = src.field_names

for field in fields:
    stats[field] = dq.FieldStatistics(field)

record_count = 0

# Probe values
for row in src.rows():
    for i, value in enumerate(row):
        stats[fields[i]].probe(value)

    record_count += 1

# Finalize statistics
for stat in stats.items():
    finalize(record_count)

Auditing using `brewery.ds.StreamAuditor`:

# ... suppose we have initialized source stream as src

# Create auditor stream target and initialize field list
auditor = ds.StreamAuditor()
auditor.fields = src.fields
auditor.initialize()

# Perform audit for each row from source:
for row in src.rows():
    auditor.append(row)
```

53
# Finalize results, close files, etc.
 auditor.finalize()

# Get the field statistics
 stats = auditor.field_statistics

class brewery.dq.FieldStatistics (key=None, distinct_threshold=10)

Data quality statistics for a dataset field

Attributes

- `field`: name of a field for which statistics are being collected
- `value_count`: number of records in which the field exist. In relationad database table this is equal to number of rows, in document based database, such as MongoDB, it is number of documents that have a key present (being null or not)
- `record_count`: total count of records in dataset. This should be set explicitly on finalisation. See FieldStatistics.finalize(). In relational database this should be the same as `value_count`
- `value_ratio`: ratio of value count to record count, 1 for relational databases
- `null_count`: number of records where field is null
- `null_value_ratio`: ratio of records with nulls to total number of probed values = `null_value_ratio / value_count`
- `null_record_ratio`: ratio of records with nulls to total number of records = `null_value_ratio / record_count`
- `empty_string_count`: number of empty strings
- `storage_types`: list of all encountered storage types (CSV, MongoDB, XLS might have different types within a field)
- `unique_storage_type`: if there is only one storage type, then this is set to that type
- `distinct_values`: list of collected distinct values
- `distinct_threshold`: number of distinct values to collect, if count of distinct values is greater than threshold, collection is stopped and `distinct_overflow` will be set. Set to 0 to get all values. Default is 10.

`dict()`
Return dictionary representation of receiver.

`finalize(record_count=None)`
Compute final statistics.

Parameters

- `record_count`: final number of records in probed dataset. See FieldStatistics() for more information.

`probe(value)`
Probe the value:

- increase found value count
- identify storage type
- probe for null and for empty string

54 Chapter 8. Data Quality
•probe distinct values: if their count is less than `distinct_threshold`. If there are more distinct values than the `distinct_threshold`, then `distinct_overflow` flag is set and list of distinct values will be empty.

```python
class brewery.dq.FieldTypeProbe (field)
  Probe for guessing field data type

  Attributes:

  • field: name of a field which statistics are being presented
  • storage_types: found storage types
  • unique_storage_type: if there is only one storage type, then this is set to that type

  `unique_storage_type`
  Return storage type if there is only one. This should always return a type in relational databases, but does not have to in databases such as MongoDB.
```
9.1 brewery

Tool for performing brewery framework functionality from command line.

Usage:

```
brewery command [command_options]
```

Commands are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>Run a stream</td>
</tr>
<tr>
<td>graph</td>
<td>Generate graphviz structure from stream</td>
</tr>
</tbody>
</table>

9.1.1 run

Example:

```
brewery run stream.json
```

The json file should contain a dictionary with nodes and connections.

9.1.2 graph

Generate a graphviz graph structure.

Example:

```
brewery run stream.json > graph.dot
dot -o graph.png -T png out.dot
```

9.1.3 nodes

List available nodes. If a node name is specified, then node information, including list of node attributes is displayed.

Example:

```
brewery nodes
brewery nodes csv_source
```
9.1.4 pipe

Create and run non-branched pipe stream. Each argument is either a node or a node attribute. The attribute has form attribute_name=value. There should be at least one node defined. If there is no source node, then CSV on standard input is assumed. If there is no target node, then CSV on standard output is assumed.

Example - audit a CSV:

cat data.csv | brewery pipe audit

Make output nicer:

cat data.csv | brewery pipe audit pretty_printer

Read CSV from a file and store in newly created SQLite database table:

```
brewery pipe csv_source resource=data.csv \ 
    sql_table_target \ 
        url=sqlite:///data.sqlite \ 
        table=data \ 
        create=1 \ 
        replace=1
```

**Warning:** This command is not fully working. There is no type conversion of values, which might cause problems. There is no way to specify non-scalar values (arrays, dictionaries). Some nodes might not have properly implemented attributes, therefore you might get error of non-existing attribute even if the attribute is there.

9.2 mongoaudit

Audit mongo database collections from data quality perspective.

Usage:

```
mongoaudit [-h] [-H HOST] [-p PORT] [-t THRESHOLD] [-f {text, json}] database collection
```

Here is a foo:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-H HOST, --host HOST</td>
<td>host with MongoDB server</td>
</tr>
<tr>
<td>-p PORT, --port PORT</td>
<td>port where MongoDB server is listening</td>
</tr>
<tr>
<td>-t THRESHOLD, --threshold THRESHOLD</td>
<td>threshold for number of distinct values (default is 10)</td>
</tr>
<tr>
<td>-f {text, json}, --format {text, json}</td>
<td>output format (default is text)</td>
</tr>
</tbody>
</table>

The *threshold* is number of distinct values to collect, if distinct values is greater than threshold, no more values are being collected and *distinct_overflow* will be set. Set to 0 to get all values. Default is 10.
9.2.1 Measured values

<table>
<thead>
<tr>
<th>Probe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>name of a field which statistics are being presented</td>
</tr>
<tr>
<td>record_count</td>
<td>total count of records in dataset</td>
</tr>
<tr>
<td>value_count</td>
<td>number of records in which the field exist. In RDB this is equal to record_count, in document based database, such as MongoDB it is number of documents that have a key present (being null or not)</td>
</tr>
<tr>
<td>value_ratio</td>
<td>ratio of value count to record count, 1 for relational databases</td>
</tr>
<tr>
<td>null_record_ratio</td>
<td>ratio of null value count to record count</td>
</tr>
<tr>
<td>null_value_ratio</td>
<td>ratio of null value count to present value count (same as null_record_ratio for relational databases)</td>
</tr>
<tr>
<td>null_count</td>
<td>number of records where field is null</td>
</tr>
<tr>
<td>unique_storage_type</td>
<td>if there is only one storage type, then this is set to that type</td>
</tr>
<tr>
<td>distinct_threshold</td>
<td></td>
</tr>
</tbody>
</table>

9.2.2 Example output

Text output:

flow:
  storage type: unicode
  present values: 1257 (10.09%)
  null: 0 (0.00% of records, 0.00% of values)
  empty strings: 0
  distinct values:
    'spending'
    'income'

pdf_link:
  storage type: unicode
  present values: 22 (95.65%)
  null: 0 (0.00% of records, 0.00% of values)
  empty strings: 0

JSON output:

```json
{
  "pdf_link": {
    "unique_storage_type": "unicode",
    "value_ratio": 0.956521739130435,
    "distinct_overflow": [true],
    "key": "pdf_link",
    "null_value_ratio": 0,
    "null_record_ratio": 0,
    "record_count": 23,
    "storage_types": [
      "unicode"
    ],
    "distinct_values": [],
    "empty_string_count": 0,
    "null_count": 0,
    "value_count": 22
  }
}
```

9.2. mongoaudit
Note: This tool will change into generic data source auditing tool and will support all datastores that brewery will support, such as relational databases or plain structured files.
CHAPTER
TEN

EXAMPLES

Contents:

10.1 Merge multiple CSV (or XLS) Files with common subset of columns into one CSV

Note:  This example can be found in the source distribution in examples/merge_multiple_files directory. Referenced CSV files are included there as well.

10.1.1 Problem Definition

We have multiple CSV files, for example with grant listing, from various sources and from various years. The files have couple common columns, such as grant receiver, grant amount, however they might contain more additional information.

Files we have:

- grants_2008.csv contains receiver, amount, date
- grants_2009.csv contains id, receiver, amount, contract_number, date
- grants_2919.csv contains receiver, subject, requested_amount, amount, date

10.1.2 Objective

Create one CSV file by sequentially merging all input CSV files and using all columns.

Based on our source files we want output csv with fields:

- receiver
- amount
- date
- id
- contract_number
- subject
- requested_amount
In addition, we would like to know where the record comes from, therefore we add file which will contain original filename.

### 10.1.3 Solution

Import brewery and all other necessary packages:

```python
import brewery
from brewery import ds
import sys
```

Specify sources:

```python
sources = [
    {
        "file": "grants_2008.csv",
        "fields": ["receiver", "amount", "date"]
    },
    {
        "file": "grants_2009.csv",
        "fields": ["id", "receiver", "amount", "contract_number", "date"]
    },
    {
        "file": "grants_2010.csv",
        "fields": ["receiver", "subject", "requested_amount", "amount", "date"]
    }
]
```

It is highly recommended to explicitly name fields contained within source files and do not rely on source headers. In this case we also make sure, that the field (column) names are normalised. That means that if in one file receiver is labeled just as “receiver” and in another it is “grant receiver” we will get the same field.

You can store sources in an external file, for example as json or yaml and read it in your script.

Now collect all the fields:

```python
# Create list of all fields and add filename to store information
# about origin of data records
all_fields = brewery.FieldList(["file"])

# Go through source definitions and collect the fields
for source in sources:
    for field in source["fields"]:
        if field not in all_fields:
            all_fields.append(field)
```

Prepare the output stream into merged.csv and specify fields we have found in sources and want to write into output:

```python
out = ds.CSVDataTarget("merged.csv")
out.fields = brewery.FieldList(all_fields)
out.initialize()
```

Go through all sources and merge them:

```python
for source in sources:
    path = source["file"]
```
src = ds.CSVDataSource(path, read_header=False, skip_rows=1)
src.fields = ds.FieldList(source["fields"])  
src.initialize()

for record in src.records():
    # Add file reference into output - to know where the row comes from
    record["file"] = path
    out.append(record)

# Close the source stream
src.finalize()

Now you have a sparse CSV files which contains all rows from source CSV files in one merged.csv.

You can “pretty print” it with:

$ cat merged.csv | brewery pipe pretty_printer

And you can see the completeness aspect of data quality with simple audit:

$ cat merged.csv | brewery pipe audit pretty_printer

10.1.4 Variations

You can have a directory with YAML files (one per record/row) as output instead of one CSV just by changing data stream target. See brewery.ds.YamlDirectoryDataTarget for more information.

out = ds.YamlDirectoryDataTarget("merged_grants")

Directory merged_grants must exist before running the script.

Or directly into a SQL database. The following will initialize SQL table target stream which will remove all existing records from the table before inserting. Note that the table grants must exist in database opendata and must contain columns with names equal to fields specified in all_fields. See brewery.ds.SQLDataTarget for more information.

out = ds.SQLDataTarget(url = "postgres://localhost/opendata",
                        table = "grants",
                        truncate = True)

Refer to source streams and source targets in the API documentation for more information about possibilities.

See Also:

Module brewery.ds List of various data sources and data targets.
Function brewery.ds.fieldlist() All streams use list of brewery.ds.Field objects for field metadata. This function will convert list of strings into list of instances of Field class.

10.2 Stream: Append Sources, Clean, Store and Audit

Situation: We have two directories containing YAML files with donations of same structure (or at least same subset of fields that we are interested in):
Some numeric fields are represented as strings, contain leading or trailing spaces, spaces between numbers.

Objective: We want to create a CSV file `donations.csv` that will contain records from both directories. Moreover we want to clean the fields: strip spaces from strings and convert numbers stored as strings into numbers. Also we want to know, how many of fields are filled in.

### 10.2.1 Solution

Problem can be solved using following data stream:

The stream consists of following nodes (from left to right):

- two YAML directory sources (*YAML Directory Source*)
- *Append* - sequentially concatenate streams
- *Coalesce Value To Type Node* - fix field values according to specified type, for example convert strings into integers for fields of type *integer*
- *CSV Target*
- *Data Audit*
- *Formatted Printer*
10.2.2 Code

Import brewery streams and nodes:

```python
from brewery.streams import *
from brewery.nodes import *
```

Create a dictionary containing nodes. We will refer to the nodes by name later.

```python
nodes = {
    "source1": YamlDirectorySourceNode(path = "donations/source1"),
    "source2": YamlDirectorySourceNode(path = "donations/source2"),
    "append": AppendNode(),
    "clean": CoalesceValueToTypeNode(),
    "output": CSVTargetNode(resource = "donations.csv"),
    "audit": AuditNode(distinct_threshold = None),
    "print": FormattedPrinterNode()
}
```

Connect the nodes:

```python
connections = [ ("source1", "append"),
                ("source2", "append"),
                ("append", "clean"),
                ("clean", "output"),
                ("clean", "audit"),
                ("audit", "print")
]
```

Specify fields that we are going to process from sources. Also specify their types for automated cleansing. For more information about fields see `brewery.ds.Field` and `brewery.ds.FieldList`. If you are not creating `FieldList` object directly, then make sure that you convert an array using `brewery.ds.fieldlist()`.

```python
fields = [ "file",
           ("source_code", "string"),
           ("id", "string"),
           ("receiver_name", "string"),
           ("project", "string"),
           ("requested_amount", "float"),
           ("received_amount", "float"),
           ("source_comment", "string")
]
```

```python
nodes["source1"].fields = ds.fieldlist(fields)
nodes["source2"].fields = ds.fieldlist(fields)
```

Configure printer node (Formatted Printer) to create nicely aligned text output:

```python
nodes["print"].header = u"field nulls empty distinct\n" + "---------------------------------------------------------------------------"
```

```python
nodes["print"].format = u"{field_name:<30.30} {null_record_ratio: >7.2%} "{empty_string_count:>10} {distinct_count:>10}"
```

Create `brewery.streams.Stream` and run it:

```python
stream = Stream(nodes, connections)
stream.run()
```

Stream will create the `donations.csv` and will produce a report on standard output that will look something like this:
10.2.3 Improvement

We know how complete (non-null) our fields are. However, are they complete enough? Say we want at least 95% completeness. We can learn from our report which fields are complete enough or not, based on the nulls report column. We still have to read the number and decide.

To aid our decision, in addition to percentage of nulls we add a flag whether the field is ok or not based on threshold. If the field null percent is greater than 5% the field quality fails and we mark it as fail, otherwise the field test passes and we mark it as ok. To derive the flag we insert a Value Threshold node.

![Updated data stream with value threshold node.](image)

```python
nodes = {
    "source1": YamlDirectorySourceNode(path = "donations/source1"),
    "source2": YamlDirectorySourceNode(path = "donations/source2"),
    "append": AppendNode(),
    "clean": CoalesceValueToTypeNode(),
    "output": CSVTargetNode(resource = "donations.csv"),
    "audit": AuditNode(distinct_threshold = None),
    "threshold": ValueThresholdNode(),  # <-- this was is added
    "print": FormattedPrinterNode()
}

Rewire nodes:
connections = [ ("source1", "append"),
                ("source2", "append"),
```
We consider field to be *ok* when null count is less than 5%, otherwise test fails. Therefore we configure threshold node like this:

```python
nodes["threshold"].thresholds = [ ["null_record_ratio", 0.05] ]
nodes["threshold"].bin_names = ("ok", "fail")
```

Update report template to include new derived field:

```python
nodes["print"].header = u"field nulls status distinct\n------------------------------------------------------------"
nodes["print"].format = u"{field_name:<30.30} {null_record_ratio: >7.2%} \\
" "{null_record_ratio_bin:>10} {distinct_count:>10}"
```

The output should look like this:

<table>
<thead>
<tr>
<th>field</th>
<th>nulls</th>
<th>status</th>
<th>distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>0.00%</td>
<td>ok</td>
<td>32</td>
</tr>
<tr>
<td>source_code</td>
<td>0.00%</td>
<td>ok</td>
<td>2</td>
</tr>
<tr>
<td>id</td>
<td>9.96%</td>
<td>fail</td>
<td>907</td>
</tr>
<tr>
<td>receiver_name</td>
<td>9.10%</td>
<td>fail</td>
<td>1950</td>
</tr>
<tr>
<td>project</td>
<td>0.05%</td>
<td>ok</td>
<td>3628</td>
</tr>
<tr>
<td>requested_amount</td>
<td>22.90%</td>
<td>fail</td>
<td>924</td>
</tr>
<tr>
<td>received_amount</td>
<td>4.98%</td>
<td>ok</td>
<td>728</td>
</tr>
<tr>
<td>source_comment</td>
<td>99.98%</td>
<td>fail</td>
<td>2</td>
</tr>
</tbody>
</table>

See Also:

- **YAML Directory Source**
- **Append**
- **Coalesce Value To Type Node**
- **CSV Target**
- **Data Audit**
- **Formatted Printer**
- **Value Threshold**
DEVELOPMENT NOTES

This is temporary documentation with development notes and feature proposals.

11.1 Server API

REST

resource: /stream/ /stream/123/nodes/
CONTACT AND GETTING HELP

If you have questions, problems or suggestions, you can send a message to Google group or write to me (Stefan Urbanek - author).

Report bugs in github issues tracking

There is an IRC channel #databrewery on server irc.freenode.net.
Brewery is licensed under MIT license with small addition:

Copyright (c) 2011-2012 Stefan Urbanek, see AUTHORS for more details

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

If your version of the Software supports interaction with it remotely
through a computer network, the above copyright notice and this permission
notice shall be accessible to all users.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.

Simply said, that if you use it as part of software as a service (SaaS) you have to provide the copyright notice in an
about, legal info, credits or some similar kind of page or info box.
INDICES AND TABLES

- genindex
- modindex
- search
PYTHON MODULE INDEX

d
ds. 13

m
metadata. 7

p
probes. 51

s
streams. 23