intake_avro Documentation

Release 0.0.1

Joseph Crail

Feb 05, 2019

Contents:

This package enables the loading of Apache Avro files within the Intake data loading and catalog system. Two plugins are contained: for fast loading of strictly tabular data to pandas dataframes, and slower reading of more complicatedly structured data as a sequence of python dictionaries.

Each avro file becomes one partition.

CHAPTER 1

Quickstart

intake_avro provides quick and easy access to tabular data stored in the Apache Avro binary, columnar format.

1.1 Installation

To use this plugin for intake, install with the following command:

```
conda install -c conda-forge intake-avro
```

1.2 Usage

1.2.1 Ad-hoc

After installation, the functions intake.open_avro_table and intake.open_avro_sequence will become available. The former, much faster method can be used to open one or more Avro files with *flat* schema into dataframes, but the latter can be used for any files and produces generic sequences of dictionaries.

Assuming some Avro files in a given path, the following would load them into a dataframe:

```
import intake
source = intake.open_avro_table('data_path/*.avro')
dataframe = source.read()
```

There will, by default, be partitions within each file of about 100MB in size. To skip scanning files for the purpose of partitioning, you can pass *blocksize=None*.

Arguments to the open_avro_* functions:

- urlpath : the location of the data. This can be a single file, a list of specific files,

  or a glob string (containing "*"). The URLs can be local files or, if using a protocol specifier such as 's3://', a remote file location.

- *blocksize*: defines the partitioning within input files. The special value *None* avoids

  partitioning within files - you get exactly one partition per input file. This avoids some upfront overhead to scan for block markers within files, so may be desirable in some cases. The default value of about 100MB, so for small files, there will be no difference.

- `storage_options` : other parameters that are to be passed to the filesystem

  implementation, in the case that a remote filesystem is referenced in `urlpath`. For specifics, see the Dask documentation.

A source so defined will provide the usual methods such as `discover` and `read_partition`.

1.2.2 Creating Catalog Entries

To use for a data-source within a catalog, a spec may look something like

sources:

> **test:** description: Sample description of some avro dataset driver: avro_table args:
>
> > urlpath: '{{ CATALOG_DIR }}/data.*.avro'

and entries must specify `driver: avro_table` or `driver: avro_sequence`. The further arguments are exactly the same as for the `open_avro_*` functions.

1.2.3 Using a Catalog

Assuming a catalog file called `cat.yaml`, containing a Avro source `pdata`, one could load it into a dataframe as follows:

```
import intake
cat = intake.Catalog('cat.yaml')
df = cat.pdata.read()
```

The type of the output will depend on the plugin that was defined in the catalog. You can inspect this before loading by looking at the `.container` attribute, which will be either `"dataframe"` or `"python"`.

The number of partitions will be at least one for the number of files pointed to.

CHAPTER 2

API Reference

`intake_avro.source.AvroTableSource`(urlpath)	Source to load tabular Avro datasets.
`intake_avro.source.AvroSequenceSource`(urlpath)	Source to load Avro datasets as sequence of Python dicts.

class `intake_avro.source.`**`AvroTableSource`**(*urlpath*, *blocksize=100000000*, *metadata=None*, *storage_options=None*)

Source to load tabular Avro datasets.

Parameters

urlpath: str Location of the data files; can include protocol and glob characters.

blocksize: int or None Partition the input files by roughly this number of bytes. Actual partition sizes will depend on the inherent structure of the data files. If None, each input file will be one partition, no file scanning will be needed ahead of time

storage_options: dict or None Parameters to pass on to the file-system backend

Attributes

cache_dirs

datashape

description

`hvplot` Returns a hvPlot object to provide a high-level plotting API.

`plot` Returns a hvPlot object to provide a high-level plotting API.

`plots` List custom associated quick-plots

Methods

`close()`	Close open resources corresponding to this data source.
`discover()`	Open resource and populate the source attributes.
`read()`	Load entire dataset into a container and return it
`read_chunked()`	Return iterator over container fragments of data source
`read_partition(i)`	Return a part of the data corresponding to i-th partition.
`to_dask()`	Create lazy dask dataframe object
`to_spark()`	Pass URL to spark to load as a DataFrame
`yaml([with_plugin])`	Return YAML representation of this data-source

set_cache_dir	

read()

Load entire dataset into a container and return it

to_dask()

Create lazy dask dataframe object

to_spark()

Pass URL to spark to load as a DataFrame

Note that this requires `org.apache.spark.sql.avro.AvroFileFormat` to be installed in your spark classes.

This feature is experimental.

class intake_avro.source.**AvroSequenceSource**(*urlpath*, *blocksize=100000000*, *metadata=None*, *storage_options=None*)

Source to load Avro datasets as sequence of Python dicts.

Parameters

urlpath: str Location of the data files; can include protocol and glob characters.

blocksize: int or None Partition the input files by roughly this number of bytes. Actual partition sizes will depend on the inherent structure of the data files. If None, each input file will be one partition, no file scanning will be needed ahead of time

storage_options: dict or None Parameters to pass on to the file-system backend

Attributes

cache_dirs

datashape

description

hvplot Returns a hvPlot object to provide a high-level plotting API.

plot Returns a hvPlot object to provide a high-level plotting API.

plots List custom associated quick-plots

Methods

`close()`	Close open resources corresponding to this data source.
`discover()`	Open resource and populate the source attributes.
`read``()`	Load entire dataset into a container and return it
`read_chunked()`	Return iterator over container fragments of data source
`read_partition(i)`	Return a part of the data corresponding to i-th partition.
`to_dask``()`	Create lazy dask bag object
`to_spark()`	Provide an equivalent data object in Apache Spark
`yaml([with_plugin])`	Return YAML representation of this data-source

set_cache_dir	

`read``()`
Load entire dataset into a container and return it

`to_dask``()`
Create lazy dask bag object

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index