intake_odbc Documentation

Release 0.1.0+1.g4851638.dirty

Joseph Crail

Nov 22, 2018

Contents:

Load data from ODBC backends into pandas data-frames using Intake with this package.

CHAPTER 1

Quickstart

intake-odbc provides quick and easy access to tabular data stored in ODBC data sources, which include a wide variety of traditional relational database systems such as MySQL and Microsoft SQL Server. Some DB systems such as PostgreSQL may have better/faster plugin implementations.

1.1 Installation

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

```
conda install -c intake intake-odbc
```

1.1.1 Setting up ODBC

Configuring ODBC is beyond the scope of this document, and generally not something that it performed by an end-user, and generally requires the installation of backend-specific drivers system-wide.

Specific documentation on the connection string and keyword arguments can be found on the TurbODBC website.

1.2 Usage

1.2.1 Ad-hoc

After installation, the functions intake.open_odbc and intake.open_odbc_partitioned will become available. Assuming you have an ODBC set up with a fully-configured connection named "MSSQL" the following would fetch the contents of mytable into a pandas dataframe:

```
import intake
source = intake.open_odbc('Driver={MSSQL}', 'SELECT * FROM mytable')
dataframe = source.read()
```

Two key arguments are required to define an ODBC data source: the DB connection parameters, and a SQL query to execute. The former may be as simple as a TurbODBC connection string, but can commonly be a set of keyword arguments, all of which are passed on to `turbodbc.connect()`. The query must have a valid syntax to be executed by the backend of choice.

In addition, the following arguments are meaningful for the non-partitioned data source:

- head_rows: how many rows are read from the start of the data to infer data

  types for discovery

- mssql: a special flag to mark datasets which are backed by MS SQL Server, which

  requires a different spelling of the `LIMIT` statement.

When using the partitioned ODBC source, further details are required in order to build the queries for each partition. It requires an index column to use for the `WHERE` statement, and the bounding values for each partition. The most explicit way to provide the boundaries is with the `divisions` keyword, or boundaries will be calculated as `npartitions` equally spaced boundaried between the min/max values. Note that some partitions may be empty.

Further arguments when using partitioning:

- index: Column to use for partitioning
- max, min: the range of values of the index column to consider for building partitiona;

  will execute a separate query to find these, if not given

- npartitions: the number of partitions to create
- divisions: explicit partition boundary values

1.2.2 Creating Catalog Entries

To us in a catalog entries must specify `driver: odbc` or `driver: odbc_partioned`. Further arguments should be provided, as for the `intake.open_*` ad-hoc commands, above. In particular, the connection parameters and query string are required, and also the index column, if using partitioning.

It should be noted that SQL query strings are generally quite long; the appropriate syntax may look like:

```
product_types:
   description: Randomly generated data
   driver: odbc_partitioned
   args:
     uri:
     sql_expr: |
         SELECT t.productid AS pid, t.productname, t.price, tt.typename
         FROM testtable t
         INNER JOIN testtypetable tt ON t.typeid=tt.typeid
     dsn: MSSQL
     mssql: true
```

Where in this case we provide a keyword argument to specify the connection and so leave the `uri` field empty, and the special YAML syntax with `"|"` is used to indicate the multi-line query, delimited by indentation.

Warning, while it is reasonable to include user parameters in the SQL query body, free-form strings or environment variables should not be used, since they will allow arbitrary code execution on the DB server (SQL injection). Similarly, the details of ODBC connections are unlikely to be useful as user parameters, except possibly to take the DB username and password from the environment.

1.2.3 Using a Catalog

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

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

The source may or may not be partitioned, depending on the plugin which was used and the parameters. Use `.discover()` to find out whether there is partitioning, and if there is, the partitions can be accessed independently.

`Dask` can be used to read a partitioned source in parallel (see method `.to_dask()`); note that there is some overhead to establishing connections from each worker, and the same ODBC drivers and configuration must exist on each machine, in the case of a distributed cluster.

CHAPTER 2

API Reference

intake_odbc.intake_odbc.ODBCSource(uri, sql_expr)	One-shot ODBC to dataframe reader
intake_odbc.intake_odbc. ODBCPartitionedSource(…)	ODBC partitioned reader

class intake_odbc.intake_odbc.**ODBCSource**(*uri*, *sql_expr*, *metadata=None*, ***odbc_kwargs*)

One-shot ODBC to dataframe reader

Parameters

uri: str or None Full connection string for TurbODBC. If using keyword parameters, this should be `None`

sql_expr: str Query expression to pass to the DB backend

Further connection arguments, such as username/password, and may also

include the following:

head_rows: int (10) Number of rows that are read from the start of the data to infer data types upon discovery

mssql: bool (False) Whether to use MS SQL Server syntax - depends on the backend target of the connection

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 (offset_tuple, container) corresponding to i-th partition.
`to_dask`()	Return a dask container for this data source
`to_spark`()	Provide an equivalent data object in Apache Spark
`yaml`([with_plugin])	Return YAML representation of this data-source

set_cache_dir	

class `intake_odbc.intake_odbc.`**`ODBCPartitionedSource`**(*uri*, *sql_expr*, *metadata=None*, ***odbc_kwargs*)

ODBC partitioned reader

This source produces new queries for each partition, where an index column is used to select rows belonging to each partition

Parameters

uri: str or None Full connection string for TurbODBC. If using keyword parameters, this should be `None`

sql_expr: str Query expression to pass to the DB backend

Further connection arguments, such as username/password, and may also

include the following:

head_rows: int (10) Number of rows that are read from the start of the data to infer data types upon discovery

mssql: bool (False) Whether to use MS SQL Server syntax - depends on the backend target of the connection

index: str Column to use for partitioning

max, min: str Range of values in index to consider (will query DB if not given)

npartitions: int Number of partitions to assume

divisions: list of values If given, use these as partition boundaries - and therefore ignore max/min and npartitions

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 (offset_tuple, container) corresponding to i-th partition.
to_dask()	Return a dask container for this data source
to_spark()	Provide an equivalent data object in Apache Spark
yaml([with_plugin])	Return YAML representation of this data-source

set_cache_dir	

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index