Using the Dataiku DSS Python API for Interfacing with SQL Databases

July 22, 2020

Marlan Crosier

PREMERA
Corporate Data & Analytics

Introduction

- Marlan Crosier, Senior Data Scientist
- Premera Blue Cross, a health insurer based in Seattle covering about 2 million members in Washington State, Alaska, and across the U.S.
- Data Science team has used DSS for about 2 years
- Use DSS for developing and deploying predictive models, primarily code based

In this presentation…

- **Purpose:**
  Share practical suggestions for making effective use of the Python API for interfacing with SQL databases across several use cases

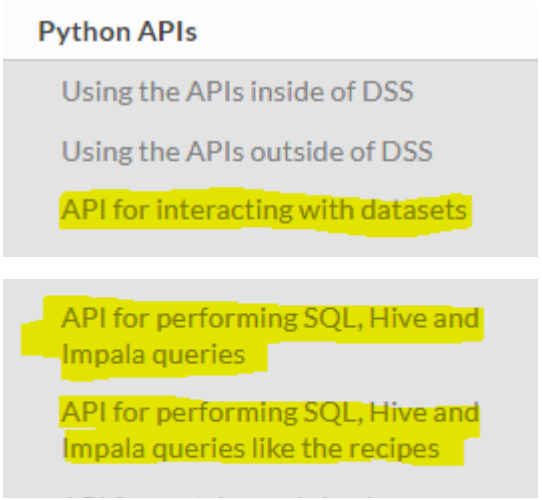
- **Agenda:**
  - Reading data
  - Writing data
  - Executing SQL

Introductory Notes

- Focus is on datasets that reference SQL tables but much of the content will apply to other types of datasets
- Tested with Netezza & Teradata, may be slight variations with other databases (e.g., we have run into a couple of small issues that are Netezza-specific)
- Tested on DSS version 6.03
- Not all the examples work in Jupyter Notebooks (all work in Python recipes)
- Assume you have a working knowledge of Python and SQL

Relevant DSS Documentation

https://doc.dataiku.com/dss/latest/



Reading Data

Load Data from Dataset into a DataFrame

```
import pandas as pd
import dataiku

dataset_object = dataiku.Dataset("DATASET_NAME")
dataframe = dataset_object.get_dataframe()
```

- SQL Table that DATASET_NAME points will be loaded
- Load is via Pandas read_table()

get_dataframe() Arguments

—

- Recommend *infer_with_pandas=False*
  - Default option (True) is to determine data types by examining the data
  - A good choice for text or similar files that don't already have types
  - Use cases: columns that are mostly numeric but sometimes alphanumeric, have leading zeros, etc.
- Avoid using *columns* argument as it currently doesn't work properly

get_dataframe() and Missing Values

- Missing values in Integer columns
  - Will get error if load a SQL table that has NULL values in an Integer column (numpy issue)
  - Either change to float type in the SQL table or use a numeric missing indicator (e.g., -99)
  - If *infer_with_pandas=True* column will be changed to Float automatically
- "NA" in string columns
  - Be aware that strings such "NA" will get converted to NaN in the Pandas dataframe
  - You may have filled NULL values in the SQL table with an "NA" string and then be surprised to see that the values are again missing in the dataframe
  - See Pandas read_table "na_values" argument for a full list of the strings that will get converted

get_dataframe() data type mapping

SQL Table	DSS Schema	Dataframe
byteint / tinyint	tinyint	object
smallint	smallint	object
integer	int	int32
bigint	bigint	int64
real	float	float32
double / numeric	double	float64
char / varchar	string	object
date / timestamp / datetime	date	datetime64[ns]

Note that currently tinyint and smallint are loaded into the object type.

You can convert to int types after DataFrame is loaded

Load SQL Query Output into DataFrame

```
from dataiku.core.sql import SQLExecutor2

executor = SQLExecutor2(dataset='DATASET4')

query = "SELECT * FROM PYSQL_TABLE4 WHERE KEY <= 3"

df = executor.query_to_df(query)
```

- Make sure to reference the SQL table name, not the DSS dataset name
- In this case, the dataset is DATASET4 while the table name is the project key prefixed to TABLE4.

SQL Query with “infer_from_schema”

```
query = """
SELECT KEY, COL1, CAST(COL1 * 2 AS BIGINT) AS NEWCOL
FROM PYSQL_TABLE4
"""

df = executor.query_to_df(query, infer_from_schema=True)
```

- *Infer_from_schema = True* does the same thing as get_dataframe’s *infer_with_pandas = False*
- The same mapping is used for the query result as is used for a DSS dataset

SQL Query – Complete Example

```
from string import Template
project_variables = dataiku.get_custom_variables()

query_with_vars = """
SELECT KEY, COL1, COL2
FROM ${projectKey}_TABLE4
WHERE DT >= '${beginDate}'
"""
query = Template(query_with_vars).substitute(project_variables)

df = executor.query_to_df(query, infer_from_schema=True)
```

- *To get SQL table name: dataiku.Dataset("DATASET4").get_location_info()['info']['table']*

What if the Data Won't Fit into Memory?

Option	Details
Sample	E.g., pass sampling= 'head' & limit= 10000 to get_dataframe() or other data reading methods
Chunk	Use iter_dataframes() or similar methods
Optimize	Use memory efficient data types

Optimizing Memory Use

- Best for machine learning to have full data available for training
- Less complex than managing data chunks
- Smaller dataframes => improved performance
- Our approach:
  - Optimize SQL table data types (e.g., use tinyint rather than integer)
  - Enhance DSS use of Pandas types when loading data into dataframe (tinyint, smallint)
  - Use Pandas category type for string columns (if many common values)
  - Up to 10X reduction in memory use (depending on table)

Memory Optimized Data Loading

```
ds = dataiku.Dataset("DATASET_NAME")

schema = ds.read_schema()
(names, dtypes, dtcols) = \
    ds.get_dataframe_schema_st(schema, infer_with_pandas=False)

type_map = (('tinyint', np.int8), ('smallint', np.int16), ('string', 'category'))
for colspec in schema:
    for dss_type, pandas_type in type_map:
        if colspec['type'] == dss_type:
            dtypes[colspec['name']] = pandas_type

df = next(ds.iter_dataframes_forced_types(names, dtypes, dtcols, chunksize=1E12))
```

- Set Pandas data types for tinyint, smallint, and string (only if many common values!)
- Use iter_dataframes_forced_types as it supports specifying Pandas types

Writing Data

Populating a Dataset from a DataFrame

- Most straightforward to use *write_with_schema()* which infers the DSS schema and the SQL table data types from the Pandas dataframe

```
ds = dataiku.Dataset('OUTPUT_DATASET_NAME')
ds.write_with_schema(df)
```

- Can of course set dataframe column types as desired so can avoid some of the previously mentioned load data issues; however:
  - Tinyint (int8) and smallint (int16) columns currently set to VARCHAR(500)
  - String (object) columns set to VARCHAR(500)
- Note that the SQL table is dropped and recreated each time

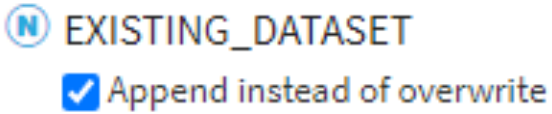
Populating a Dataset with a Predefined Schema

- Use write_dataframe for full control over output SQL table data types
- Preset dataset schema (and thereby SQL table data types)
  - In UI under dataset Settings, Schema tab
  - Programmatically via *set_schema()* (use *get_schema() to see structure*)
- Dataframe types must be compatible with the SQL table data types

```
ds = dataiku.Dataset('DATASET_NAME')
ds.write_dataframe(df)
```

Appending Rows to an Existing Table

- Simply a variation on “Populating a Dataset with a Defined Schema”
- On recipe Inputs/Outputs tab, check "Append instead of overwrite“ box



- Appending not allowed in a Jupyter Notebook

An Alternative Approach to Appending Rows

- For many use cases, appending records directly may not the best option
- Instead we often:
  - Write data to a SQL table that is dropped and recreated each time
  - Use a subsequent SQL recipe to append records to another table
- Main advantage is reliability
  - Records to be appended may already be in the destination table (due to failures/mistakes)
  - May want exclude such duplicate records or perhaps update certain columns
  - SQL well suited to handle these situations in terms of both functionality and performance

Executing SQL

Populate a Dataset from a SQL Query

Use *exec_recipe_fragment* for SQL statements/scripts can't write in SQL recipe

```
in_ds = dataiku.Dataset("INPUT_DATASET")
in_tbl = in_ds.get_location_info()['info']['table']
in_df = in_ds.get_dataframe(infer_with_pandas=False)
out_ds = dataiku.Dataset("OUTPUT_DATASET")

stmts = """DROP TABLE TMP_INPUT IF EXISTS;
CREATE TEMP TABLE TMP_INPUT AS SELECT * FROM {0} LIMIT 0;
INSERT INTO TMP_INPUT SELECT * FROM {0} WHERE INTEGER_COL = {1}"""
pre_stmts = pre_stmts.format(in_tbl, str(in_df.at[2, 'INTEGER_COL']))

query = "SELECT * FROM TMP_INPUT"
SQLExecutor2.exec_recipe_fragment(out_ds, query, pre_queries=[stmts])
```

Notes on exec_recipe_fragment

- *final_query* must be a select; additional statements can go in *pre_queries*
- *pre_queries* must be a list (Netezza: multiple statements in one item list OK; Teradata: one list item per statement is required with DDL statements)
- Must specify input dataset as queries run under this dataset's connection
- All/some of *pre_queries* executed twice: once to determine output dataset schema and then again "for real"
  - Need DROP TABLE X IF EXISTS in *pre_queries* or DROP TABLE X in *post_queries*
  - Create tables first and then populate them (so tables populated only once)

Execute Arbitrary SQL Statements

Clear tables, create tables, rename tables, update indexes, etc.

```
executor = SQLExecutor2(connection='NZ_Lab’)

query =  "select count(*) as rec_cnt from OUTPUT_TABLE"
pre_queries = ["truncate table OUTPUT_TABLE; commit"]
df = executor.query_to_df(query, pre_queries=pre_queries)

assert (df.at[0, 'REC_CNT'] == 0), "Truncate failed."
```

- Statements go in *pre_queries*
- May need terminating *commit* (needed in Netezza but not Teradata)
- Use *query* to check status of statements

Resources

- This slide deck
- DSS project with Python recipe examples
- Enter into DSS CODE SAMPLES