How to use Pandas to access databases

NOVEMBER 10

ANT

Authored by: Your Name

Logo Name

Source: https://medium.com/jbennetcodes/how-to-use-pandas-to-access-databases-e4e74e6a329e

How to use Pandas to access databases

and is that the right thing to do

Irina Truong

Nov 26, 2019 · 4 min read

Pandas is a great tool to explore the data stored in files (comma-delimited, tab-delimited, Parquet, HDF5, etc). In fact, most tutorials that you'll find on Pandas will start with reading some kind of a sample file (or files), most likely using .**read_csv:**

```
1  import pandas as pd
2
3
4  df = pd.read_csv("sample_file.csv")
```

read_sample.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/ce2f232ef8eda4cff95280d2ab7be19b

Sometimes, the data that you want to analyze is stored in a different kind of storage, for example, a relational database. It is possible to load this data into Pandas dataframes, with an SQLAlchemy connection (or a DBAPI2 connection for sqlite):

```
1  import pandas as pd
2
3  from sqlalchemy import create_engine
4  conn = create_engine('mysql://root@localhost/test')
5  df = pd.read_sql_table('abc', conn)
```

read_table_sqlalchemy.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/ea28d615a831dc568a806d8b1b1dcbc6

The code is very simple, and it looks nice and easy. Load all the tables into dataframes, and do the analysis on them in Pandas.

But should you?

As it often happens, the answer it not black-and-white. It depends, mostly, on the size of your data. Why does it matter? Remember that with Pandas, things are not lazy. The Pandas dataframe is a structure in memory. If your table has lots of fields and millions of records, and you try loading the whole thing into memory, you might just crash your computer, or at the very least, have an OOM (Out Of Memory exception). And loading multiple large tables? Not going to happen.

So what are the options then? When the data is stored in a database, start exploring the data from there. The database is a storage highly optimized for querying. It would be wrong not to take advantage of it.

Explore the database using a CLI

A lot of useful things about the data can be found out using SQL queries and your favorite database client. Here are some example queries (using sakila database in MySQL and mycli client):

```
1  mysql root@localhost:sakila> show tables
2
3  mysql root@localhost:sakila> select count(*) from payment
4
5  mysql root@localhost:sakila> describe payment
```

sql_explore_sakila.sql hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/48c162a8f2d20f7718acb7bd5d1180af

Note that since these queries only return one record, or a few records at most, they can be safely issued via Pandas as well:

```
In [13]: pd.read_sql_query('show tables', conn)

In [14]: pd.read_sql_query('select count(*) from payment', conn)

In [15]: pd.read_sql_query('select * from payment limit 5', conn)
```

pandas_sql_query.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/f067ef367d846e7dd323fcfe3e6fe339

On the other hand, you'd lose SQL autocompletion, syntax highlighting, and perhaps other features that your database client provides.

For the queries above, it comes to preference — whether you use pandas to query the database or the database client application. This is not the case with queries that retrieve a lot of data. When using Pandas, it makes sense to minimize the amount of data you load into memory. There are a few strategies for that:

Limit the fields to retrieve

Hopefully, when doing the initial exploration of the data (as shown above), you zeroed out on the subset of tables and fields you're interested in. So query for only those fields and tables when loading your Pandas dataframes, with **pd.read_sql_query:**

```
# don't do this
pd.read_sql_query('select * from table1', conn)

# do this
pd.read_sql_query('select field1, field2, field3 from table1', conn)
```

pd_read_sql_query2.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/e85f7694cb4ec621f06c59a84c9b81da

Limit the records to retrieve

This means either sampling the records (**LIMIT XXX** clause might do that), or only retrieving records that fit a specific criteria (if you only want to analyze payments in the current year, it does not make sense to load the whole table, so you're going to need a **WHERE** condition in that SQL query):

```
# don't do this
pd.read_sql_query("select field1, field2, field3 from table1")

# do this
pd.read_sql_query("select field1, field2, field3 from table1 limit 1000")

# or this
pd.read_sql_query("select field1, field2, field3 from table1 where field1 = 'category1'")
```

pd_sql_query3.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/2aefe79c0af40f870d6bb2703f94abca

Let database server handle joins

Any time you think of retrieving records from tables into Pandas dataframes with the purpose of later joining these dataframes, it's probably not the best idea. Databases are highly optimized for joins. It makes much more sense to unload this task to the database server. Besides, one dataframe in memory (the resulting view of the **JOIN**) is better than two:

```
# don't do this
t1 = pd.read_sql_query("select t1.field1, t1.field2 from table1 t1")
t2 = pd.read_sql_query("select t2.field1, t2.field2 from table2 t2")
df = t1.merge(t2, how="inner", on="field1")

# do this
pd.read_sql_query("select t1.field1, t1.field2, t2.field2 "
                  "from table1 t1"
                  "join table2 t2 on t1.field1 = t1.field1")
```

pd_join_database.py hosted with ♥ by GitHub view raw

https://gist.github.com/j-bennet/ca2a85974c898707c02e3155308e0348

Estimate memory usage

It is very easy to estimate memory usage with Pandas, with **.memory_usage** call on dataframe:

```
In [29]: df = pd.read_sql_query('select * from payment limit 100', conn)

In [31]: df.memory_usage().sum()
Out[31]: 5728
```

pd_estimate_mem.py hosted with ♥ by GitHub view raw

https://gist.github.com/j-bennet/546a4c20fc9b178155acdc4aa6a158ab

If a sample of 100 records take up 5,728 bytes, then 1,000,000 records will take up approximately 57280000 bytes, or 54Mb. This is very inexact, but at least it can give you an idea.

Reduce memory usage with data types

Sometimes, it is possible to use more memory efficient datatypes on the dataframe fields. By default, Pandas will read all integer data types in database as **int64**, even though they might have been defined as smaller data types in database. For example, let's look at this table:

```
mysql root@localhost:sakila> describe payment
+--------------+----------------------+------+-----+-------------------+-------------------------
| Field        | Type                 | Null | Key | Default           | Extra
+--------------+----------------------+------+-----+-------------------+-------------------------
| payment_id   | smallint(5) unsigned | NO   | PRI | <null>            | auto_increment
| customer_id  | smallint(5) unsigned | NO   | MUL | <null>            |
| staff_id     | tinyint(3) unsigned  | NO   | MUL | <null>            |
| rental_id    | int(11)              | YES  | MUL | <null>            |
| amount       | decimal(5,2)         | NO   |     | <null>            |
| payment_date | datetime             | NO   |     | <null>            |
| last_update  | timestamp            | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED on up
+--------------+----------------------+------+-----+-------------------+-------------------------
```

db_data_types.sql hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/f3a9311aaceba79616bd0753451d7b1e

When loaded with Pandas, data types are not equivalent to those in database:

```
In [62]: pd.read_sql_table('payment', conn).dtypes
Out[62]:
payment_id               int64
customer_id              int64
staff_id                 int64
rental_id              float64
amount                 float64
payment_date    datetime64[ns]
last_update     datetime64[ns]
dtype: object
```

pd_data_types.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/ebdbcaf1fcca2f1957780ac5321ef247

Database **smallint** and **tinyint** types got converted into **int64** — which means using more memory that we have to. Can we do better?

Here is one way to optimize things:

- Read the table (or query) in chunks, providing the **chunksize** parameter.

```
In [52]: df_iter = pd.read_sql_query('select * from payment limit 100', conn, chunksize=4)
```

pd_read_chunksize.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/ce556e9faed37c48336b34d442297b63

Since we assume the table is large, we can't load the whole result into memory at once.

- Convert datatypes of each chunk to smaller datatypes.

```
In [64]: def convert_int_types(df_iter):
    ...:     for chunk in df_iter:
    ...:         chunk["payment_id"] = chunk["payment_id"].astype(np.uint16)
    ...:         chunk["customer_id"] = chunk["customer_id"].astype(np.uint16)
    ...:         chunk["staff_id"] = chunk["staff_id"].astype(np.uint8)
    ...:         yield chunk
    ...:
    ...:
```

pd_convert_int_types.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/a9a4971c7d48ee81341be4981e826504

Here, we know which fields can be converted into smaller types, because we saw the table definition in database.

- Concatenate updated chunks into a new dataframe.

```
In [66]: df3 = pd.concat(convert_int_types(df_iter))

In [67]: df3.memory_usage().sum()
Out[67]: 4500
```

pandas_convert_concat.py hosted with ❤ by GitHub view raw

https://gist.github.com/j-bennet/3225de6081e210f56dd5bde21f07507e

Not bad — we now have 4500 bytes instead of 5728 bytes, a 21% reduction in memory. This excellent IPython notebook shows how to optimize data types even further, and provides a generic function to do that:

Reducing DataFrame memory size by ~65%

Download Open Datasets on 1000s of Projects + Share Projects on One Platform. Explore Popular Topics Like Government…

www.kaggle.com

Summary

To reiterate the important points of using Pandas to explore a database:

- Dataframes in Pandas are not lazy, they are loaded into memory, be aware of the memory usage.
- Start exploring with a SQL client to determine the size and shape of data.
- Proceed based on the size of data, to either load whole tables into Pandas, or query for only selected fields and possibly limit to a sample of records.
- Let database do the joins, it's good at it.
- Estimate the size of data. If necessary, use memory-efficient data types.

Happy exploring!

Special thanks to Sheila Tüpker, who inspired this article by asking a very good question.

https://www.kaggle.com/arjanso/reducing-dataframe-memory-size-by-65

Reducing DataFrame memory size by ~65%

Python notebook using data from Zillow Prize: Zillow's Home Value Prediction (Zestimate) · 50,263 views · 3y ago

This notebook shows how I reduce the size of the properties dataset by selecting smaller datatypes.

I noticed the size of the properties dataset is pretty big for a lower/mid-range laptop so I made a script to make the dataset smaller without losing information.

This notebook uses the following approach:

1. **Iterate over every column**
2. **Determine if the column is numeric**
3. **Determine if the column can be represented by an integer**
4. **Find the min and the max value**
5. **Determine and apply the smallest datatype that can fit the range of values**

This reduces the dataset from approx. 1.3 GB to 466 MB

1 | load packages

In [1]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

2 | Function for reducing memory usage of a pandas dataframe

In [2]:

```
def reduce_mem_usage(props):
    start_mem_usg = props.memory_usage().sum() / 1024**2
    print("Memory usage of properties dataframe is :",start_mem_usg," MB")
    NAlist = [] # Keeps track of columns that have missing values filled in.
    for col in props.columns:
        if props[col].dtype != object:  # Exclude strings

            # Print current column type
            print("******************************")
            print("Column: ",col)
            print("dtype before: ",props[col].dtype)

            # make variables for Int, max and min
            IsInt = False
            mx = props[col].max()
```

```
            mn = props[col].min()

            # Integer does not support NA, therefore, NA needs to be filled
            if not np.isfinite(props[col]).all():
                NAlist.append(col)
                props[col].fillna(mn-1,inplace=True)

            # test if column can be converted to an integer
            asint = props[col].fillna(0).astype(np.int64)
            result = (props[col] - asint)
            result = result.sum()
            if result > -0.01 and result < 0.01:
                IsInt = True


            # Make Integer/unsigned Integer datatypes
            if IsInt:
                if mn >= 0:
                    if mx < 255:
                        props[col] = props[col].astype(np.uint8)
                    elif mx < 65535:
                        props[col] = props[col].astype(np.uint16)
                    elif mx < 4294967295:
                        props[col] = props[col].astype(np.uint32)
                    else:
                        props[col] = props[col].astype(np.uint64)
                else:
                    if mn > np.iinfo(np.int8).min and mx < np.iinfo(np.int8).max:
                        props[col] = props[col].astype(np.int8)
                    elif mn > np.iinfo(np.int16).min and mx < np.iinfo(np.int16).max:
                        props[col] = props[col].astype(np.int16)
                    elif mn > np.iinfo(np.int32).min and mx < np.iinfo(np.int32).max:
                        props[col] = props[col].astype(np.int32)
                    elif mn > np.iinfo(np.int64).min and mx < np.iinfo(np.int64).max:
                        props[col] = props[col].astype(np.int64)

            # Make float datatypes 32 bit
            else:
                props[col] = props[col].astype(np.float32)

            # Print new column type
            print("dtype after: ",props[col].dtype)
            print("*******************************")

    # Print final result
    print("___MEMORY USAGE AFTER COMPLETION:___")
    mem_usg = props.memory_usage().sum() / 1024**2
    print("Memory usage is: ",mem_usg," MB")
    print("This is ",100*mem_usg/start_mem_usg,"% of the initial size")
    return props, NAlist
```

3 | Load Data

```
props = pd.read_csv(r"../input/properties_2016.csv")  #The properties dataset

#train = pd.read_csv(r"../input/train_2016_v2.csv")   # The parcelid's with their outcomes
#samp = pd.read_csv(r"../input/sample_submission.csv")  #The parcelid's for the testset
```

4 | Run function

```
props, NAlist = reduce_mem_usage(props)
print("_____________________")
print("")
print("Warning: the following columns have missing values filled with 'df['column_name'].m
in() -1': ")
print("_____________________")
print("")
print(NAlist)
```