vinum

Dmitry Koval

Mar 19, 2021

CONTENTS:

Vinum is a SQL query processor for Python, designed for data analysis workflows and in-memory analytics.

CHAPTER

ONE

WHEN SHOULD I USE VINUM?

Vinum is running inside of the host Python process and allows to execute any functions available to the interpreter as UDFs. If you are doing data analysis or running ETL in Python, Vinum allows to execute efficient SQL queries with an ability to call native Python UDFs.

CHAPTER

TWO

KEY FEATURES:

- Vinum is running inside of the host Python process and has a hybrid query execution model - whenever possible it would prefer native compiled version of operators and only executes python interpreted code where strictly necessary (ie. for native Python UDFs).
- Allows to use functions available within the host Python interpreter as UDFs, including native Python, NumPy, Pandas, etc.
- Vinum's execution model doesn't require input datasets to fit into memory, as it operates on the stream batches. However, final result is fully materialized in memory.
- Written in the mix of C++ and Python and built from ground up on top of Apache Arrow, which provides the foundation for moving data and enables minimal overhead for transferring data to and from Numpy and Pandas.

2.1 Installation

2.1.1 With pip

```
pip install vinum
```

2.1.2 Usage example

```
>>> import vinum as vn
>>> tbl = vn.read_csv('test.csv')
>>> res_tbl = tbl.sql('SELECT * FROM t WHERE fare > 5 LIMIT 3')
>>> res_tbl.to_pandas()
   id                          ts        lat        lng  fare
0   1   2010-01-05 16:52:16.0000002  40.711303 -74.016048  16.9
1   2  2011-08-18 00:35:00.00000049  40.761270 -73.982738   5.7
2   3   2012-04-21 04:30:42.0000001  40.733143 -73.987130   7.7
```

2.2 Getting started

2.2.1 Install

```
pip install vinum
```

2.2.2 Examples

Query python dict

Create a a Table from a python dict and return result of the query as a Pandas DataFrame.

```
>>> import vinum as vn
>>> data = {'value': [300.1, 2.8, 880], 'mode': ['air', 'bus', 'air']}
>>> tbl = vn.Table.from_pydict(data)
>>> tbl.sql_pd("SELECT value, np.log(value) FROM t WHERE mode='air'")
   value    np.log
0  300.1  5.704116
1  880.0  6.779922
```

Query pandas dataframe

```
>>> import pandas as pd
>>> import vinum as vn
>>> data = {'col1': [1, 2, 3], 'col2': [7, 13, 17]}
>>> pdf = pd.DataFrame(data=data)
>>> tbl = vn.Table.from_pandas(pdf)
>>> tbl.sql_pd('SELECT * FROM t WHERE col2 > 10 ORDER BY col1 DESC')
   col1  col2
0     3    17
1     2    13
```

Query csv

```
>>> import vinum as vn
>>> tbl = vn.read_csv('test.csv')
>>> res_tbl = tbl.sql('SELECT * FROM t WHERE fare > 5 LIMIT 3')
>>> res_tbl.to_pandas()
   id                          ts        lat        lng  fare
0   1   2010-01-05 16:52:16.0000002  40.711303 -74.016048  16.9
1   2  2011-08-18 00:35:00.00000049  40.761270 -73.982738   5.7
2   3   2012-04-21 04:30:42.0000001  40.733143 -73.987130   7.7
```

Compute Euclidean distance with numpy functions

Use any numpy functions via the 'np.*' namespace.

```
>>> import vinum as vn
>>> tbl = vn.Table.from_pydict({'x': [1, 2, 3], 'y': [7, 13, 17]})
>>> tbl.sql_pd('SELECT *, np.sqrt(np.square(x) + np.square(y)) dist '
...            'FROM t ORDER BY dist DESC')
   x   y       dist
0  3  17  17.262677
1  2  13  13.152946
2  1   7   7.071068
```

Compute Euclidean distance with vectorized UDF

Register UDF performing vectorized operations on Numpy arrays.

```
>>> import vinum as vn
>>> vn.register_numpy('distance',
...                   lambda x, y: np.sqrt(np.square(x) + np.square(y)))
>>> tbl = vn.Table.from_pydict({'x': [1, 2, 3], 'y': [7, 13, 17]})
>>> tbl.sql_pd('SELECT *, distance(x, y) AS dist '
...            'FROM t ORDER BY dist DESC')
   x   y       dist
0  3  17  17.262677
1  2  13  13.152946
2  1   7   7.071068
```

Compute Euclidean distance with python UDF

Register Python lambda function as UDF.

```
>>> import math
>>> import vinum as vn
>>> vn.register_python('distance', lambda x, y: math.sqrt(x**2 + y**2))
>>> tbl = vn.Table.from_pydict({'x': [1, 2, 3], 'y': [7, 13, 17]})
>>> tbl.sql_pd('SELECT x, y, distance(x, y) AS dist FROM t')
   x   y       dist
0  1   7   7.071068
1  2  13  13.152946
2  3  17  17.262677
```

Group by z-score

```
>>> import numpy as np
>>> import vinum as vn
>>> def z_score(x: np.ndarray):
...     """Compute Standard Score"""
...     mean = np.mean(x)
...     std = np.std(x)
...     return (x - mean) / std
...
>>> vn.register_numpy('score', z_score)
```

(continues on next page)

(continued from previous page)

```
>>> tbl = vn.read_csv('test.csv')
>>> tbl.sql_pd('select int(score(fare)) AS bucket, avg(fare), count(*) '
...            'FROM t GROUP BY bucket ORDER BY bucket')
   bucket        avg  count
0       0   8.111630     92
1       1  19.380000      3
2       2  27.433333      3
3       3  34.670000      1
4       6  58.000000      1
```

2.3 SQL

Currently, only SELECT statement is supported.

2.3.1 SELECT

Vinum uses SQL parser from PostgreSQL and strives to follow its SQL dialect.

SELECT [DISTINCT] (col, [,. . .]) { * | column | expression } [[AS] alias] [, . . .]
[FROM table (is ignored, present only for compatibility)]
[WHERE condition]
[GROUP BY { column | expression | alias} [, . . .]]
[HAVING condition]
[ORDER BY { column | expression } [ASC | DESC] [, . . .]]
[LIMIT limit, [offset]]

Notes:

- Subqueries and JOINs are currently not supported.

2.3.2 SQL Operators

Unary

`-arg` - negation
`~arg` - binary not

Math

`+` - addition
`-` - subtraction
`*` - multiplication
`\` - division
`%` - modulo

Binary

`&` - binary and
`|` - binary or

Logical

`AND` - Logical AND
`OR` - Logical OR
`NOT` - Logical NOT
=, == - Equals
!=, <> - Not equals
> - Greater than
>= - Greater than or equals to
< - Less than
<= - Less than or equals to
`IS NULL` - Is NULL
`IS NOT NULL` - Is not NULL
`IN` - Value is in a list
`NOT IN` - Value is not in a list
`BETWEEN` - Value is within a range
`NOT BETWEEN` - Value is not not within a range
`LIKE` - String value matches a pattern
`NOT LIKE` - String value does not match a pattern

String

`||` - concat

2.3.3 Built in functions

List of Built-in SQL functions.

Numpy functions

np.*

np.* - All the functions from the *numpy* package are supported by default via the *np.** namespace.

For example:

```
select np.log(total) from passengers
select np.power(np.min(size), 3) as cubed from measurements
```

Type cast functions

to_bool

to_bool(arg) - Casts argument to bool type.

to_float

to_float(arg) - Casts argument to float type.

to_int

to_int(arg) - Casts argument to int type.

to_str

to_str(arg) - Casts argument to str type.

Math functions

abs

abs(arg) - returns the absolute value of the numerical argument.

See: numpy.absolute

sqrt

sqrt(arg) - returns the square root of the numerical argument.

See: numpy.sqrt

cos

cos(arg) - returns the cosine of the argument.

See: numpy.cos

sin

sin(arg) - returns the sine of the argument.

See: numpy.sin

tan

tan(arg) - returns the tangent of the argument.

See: numpy.tan

power

power(arg, power) - returns the argument(s) raised to the power.

See: numpy.power

log

log(arg) - returns the natural log of the argument.

See: numpy.log

log2

log2(arg) - returns the log base 2 of the argument.

See: numpy.log2

log10

log10(arg) - returns the log base 10 of the argument.

See: numpy.log10

Math constants

e

e() - returns the e constant.

pi

pi() - returns the pi constant.

String functions

concat

concat(arg1, arg2, . . .) - concatenate string arguments.

If argument is not a string type, would be converted to string.

See: numpy.char.add

upper

upper(arg) - convert a string to uppercase.

See: numpy.char.upper

lower

lower(arg) - convert a string to lowercase.

See: numpy.char.lower

Datetime functions

date

date(arg) - converts the argument to *date* type.

Input is either a string in ISO8601 format or integer timestamp.

Use *date('now')* for current date.

See: numpy.datetime

datetime

datetime(arg, unit) - converts the argument to *datetime* type.

Input is either a string in ISO8601 format or integer timestamp.

Supported units are: ['D', 's', 'ms', 'us', 'ns']

'D' - days
's' - seconds
'ms' - milliseconds
'us' - microseconds
'ns' - nanoseconds

Use *datetime('now')* for current datetime.

See: numpy.datetime

from_timestamp

from_timestamp(arg, unit) - converts the integer timestamp to *datetime* type. Argument represents integer value of the timestamp, ie number of seconds (or milliseconds) since epoch.

Supported units are : ['s', 'ms', 'us', 'ns']

's' - seconds
'ms' - milliseconds
'us' - microseconds
'ns' - nanoseconds

See: numpy.datetime

timedelta

timedelta(arg, unit) - returns the *timedelta* type. Argument represents the duration.

Supported units are : [‘Y’, ‘M’, ‘W’, ‘D’, ‘h’, ‘m’, ‘s’, ‘ms’, ‘us’, ‘ns’]

‘Y’ - years
‘M’ - months
‘W’ - weeks
‘D’ - days
‘h’ - hours
‘m’ - minutes
‘s’ - seconds
‘ms’ - milliseconds
‘us’ - microseconds
‘ns’ - nanoseconds

See: numpy.datetime.timedelta

is_busday

is_busday(arg) - returns True if the argument is a ‘business’ day.

See: numpy.datetime.is_busday

Aggregate functions

count

count(*) - returns the number of all rows in the group.
count(expr | column) - returns the number of non-null rows in the group.

min

min(expr | column) - returns the minimum value in the group.

max

max(expr | column) - returns the maximum value in the group.

sum

sum(expr | column) - returns the sum of the values in the group.

avg

avg(expr | column) - returns the arithmetic mean of the values in the group.

2.3.4 User Defined Functions

Functions to define UDFs.

register_numpy

register_python

2.4 API reference.

Vinum API reference.

2.4.1 Table

Vinum Table class.

2.4.2 Stream Reader

Vinum Stream Reader class.

2.4.3 Input/Output functions

List of Input/Output functions and factory methods.

Table.from_arrow

Table.from_pandas

stream_csv

read_csv

read_json

read_parquet

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search