vinum

Dmitry Koval

May 15, 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

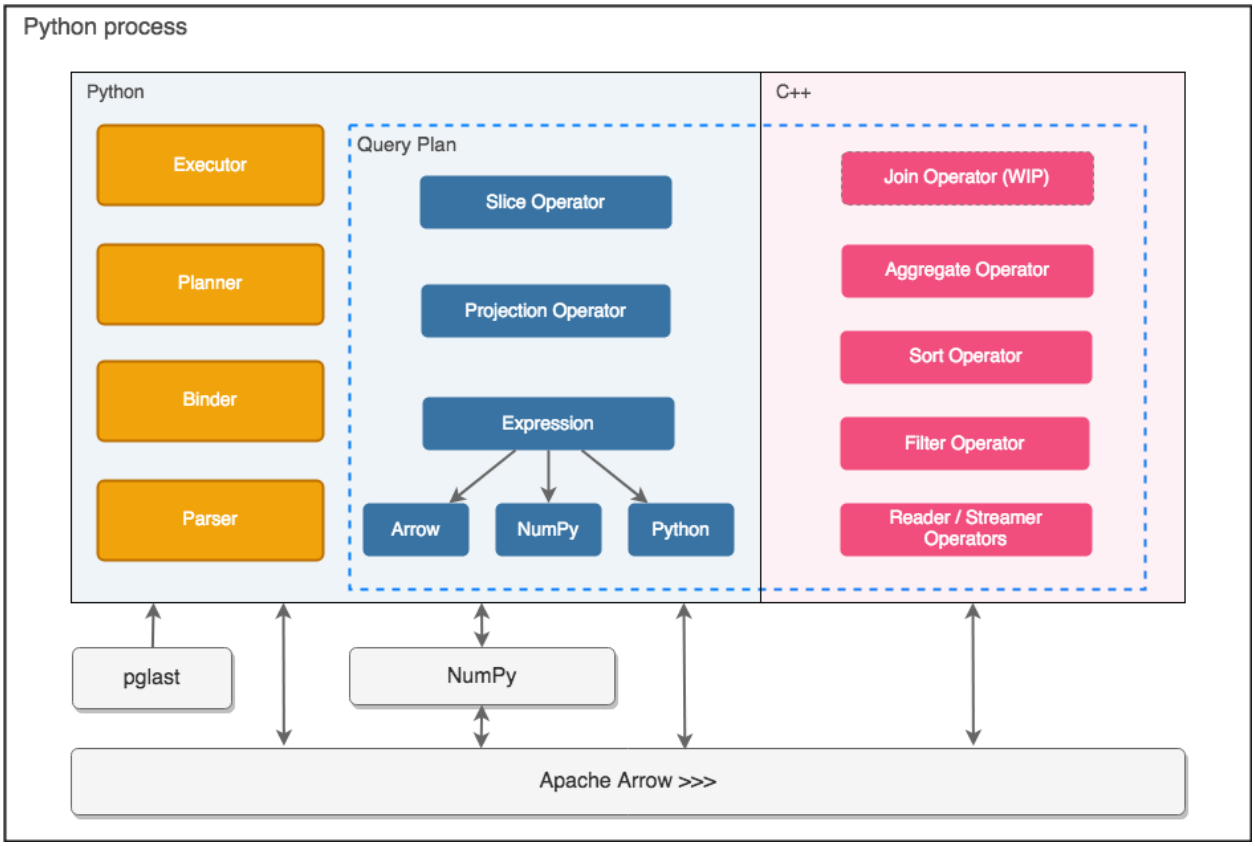
KEY FEATURES:

- 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, etc.
- Vinum's execution model doesn't require input datasets to fit into memory, as it operates on a stream of record batches. However, final result is fully materialized in memory.
- Written in the mix of C++ and Python and is 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.

CHAPTER

THREE

ARCHITECTURE

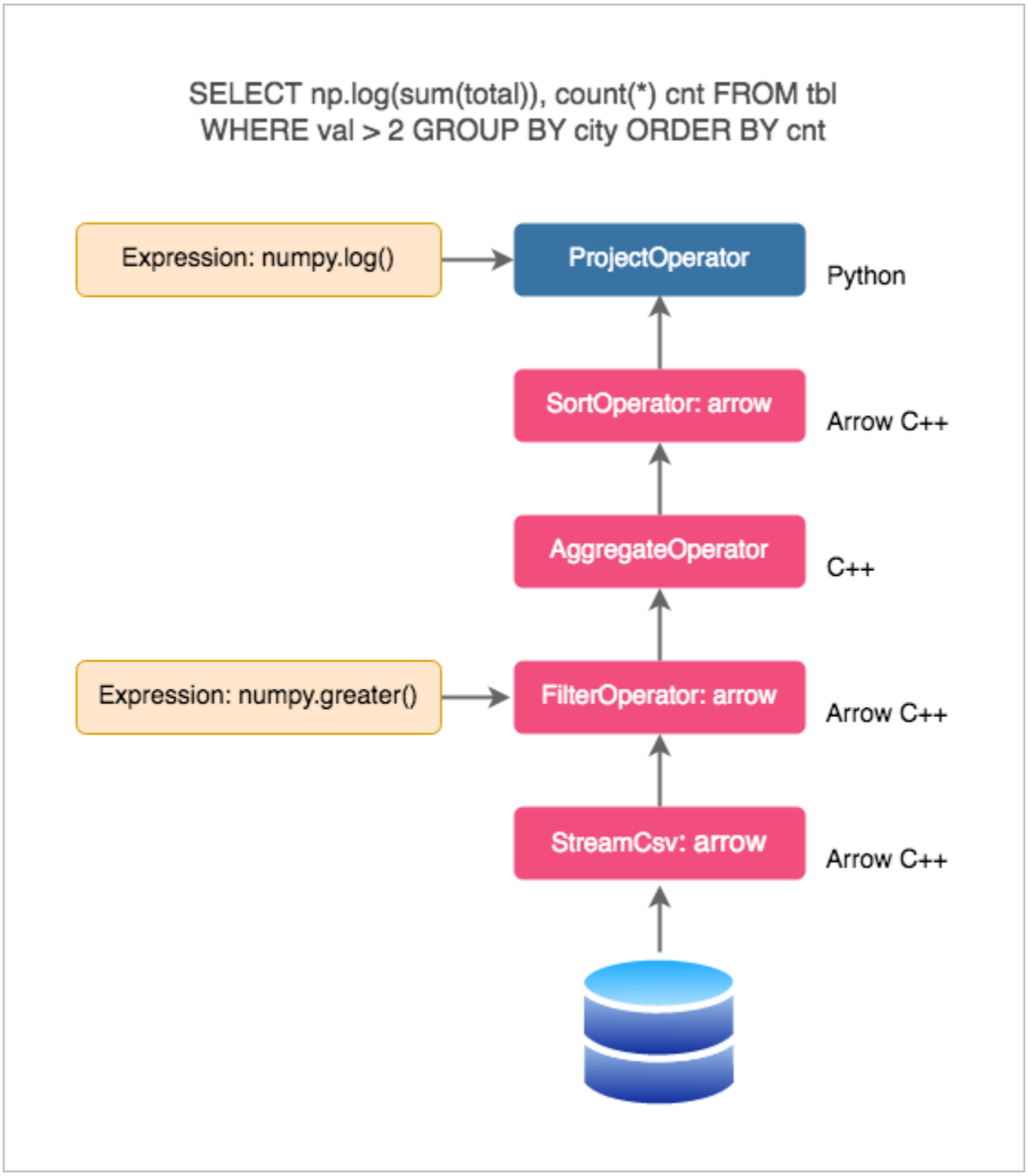


Vinum uses PostgresSQL parser provided by pglast project.

Query planner and executor are implemented in Python, while all the physical operators are either implemented in C++ or use compiled vectorized kernels from Arrow or NumPy. The only exception to this is native python UDFs, which are running within interpreted Python.

Query execution model is based on the vectorized model described in the prolific paper by P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In CIDR, 2005.

Example of a query plan:



3.1 Installation

3.1.1 With pip

```
pip install vinum
```

3.1.2 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
```

3.2 Getting started

3.2.1 Install

```
pip install vinum
```

3.2.2 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
```

Run query on a csv stream

For larger datasets or datasets that won't fit into memory - stream_csv() is the recommended way to execute a query. Compressed files are also supported and can be streamed without prior extraction.

```
>>> import vinum as vn
>>> query = 'select passenger_count pc, count(*) from t group by pc'
>>> vn.stream_csv('taxi.csv.bz2').sql(query).to_pandas()
   pc  count
0   0    165
1   5   3453
...
```

Read and query csv

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> res_tbl = tbl.sql('SELECT key, fare_amount, passenger_count FROM t '
...                   'WHERE fare_amount > 5 LIMIT 3')
>>> res_tbl.to_pandas()
                           key  fare_amount  passenger_count
0   2010-01-05 16:52:16.0000002         16.9                1
1  2011-08-18 00:35:00.00000049          5.7                2
2   2012-04-21 04:30:42.0000001          7.7                1
```

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
```

(continues on next page)

(continued from previous page)

```
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)
>>> tbl = vn.read_csv('test.csv')
>>> tbl.sql_pd('select to_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
```

3.3 SQL

Currently, only SELECT statement is supported.

3.3.1 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.

3.3.2 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

3.3.3 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

now

now() - returns current datetime.

Returns current time as a datetime with seconds resolution.

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.

3.3.4 User Defined Functions

Functions to define UDFs.

register_numpy

vinum.**register_numpy**(*function_name: str*, *function*) → None

Register Numpy function as a User Defined Function (UDF). UDF can perform vectorized operations on arrays passed as arguments.

Parameters

- **function_name** (*str*) – Name of the User Defined Function.
- **function** (*callable*) – Function to be used as a UDF. Function has to operate on vectorized numpy arrays. Numpy arrays will be passed as input arguments to the function and it should return numpy array.

See also:

register_python Register Python function as a User Defined Function.

Notes

Numpy package is imported under *np* namespace. You can invoke any function from the *np.** namespace.

Arguments of the function would be numpy arrays of provided columns. UDF can perform vectorized operations on arrays passed as arguments. The function would be called only once.

Function names are case insensitive.

Examples

Define a function operating with Numpy arrays. Numpy function perform vectorized operations on input numpy arrays.

```
>>> import numpy as np
>>> import vinum as vn
>>> vn.register_numpy('cube', lambda x: np.power(x, 3))
>>> tbl = vn.Table.from_pydict({'len': [1, 2, 3], 'size': [7, 13, 17]})
>>> tbl.sql_pd('SELECT cube(size) from t ORDER BY cube(size) DESC')
   cube
0  4913
1  2197
2   343
```

```
>>> import numpy as np
>>> 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 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
```

Please note that *x* and *y* arguments are of *np.array* type. In both of the cases function perform vectorized operations on input numpy arrays.

```
>>> import numpy as np
>>> import vinum as vn
>>> def z_score(x: np.array):
...     """Compute Standard Score"""
...     mean = np.mean(x)
...     std = np.std(x)
...     return (x - mean) / std
...
>>> vn.register_numpy('score', z_score)
>>> tbl = vn.Table.from_pydict({'x': [1, 2, 3], 'y': [7, 13, 17]})
>>> tbl.sql_pd('select x, score(x), y, score(y) from t')
   x     score   y   score_1
0  1 -1.224745   7 -1.297771
1  2  0.000000  13  0.162221
2  3  1.224745  17  1.135550
```

Please note that *x* argument is of *np.array* type.

register_python

vinum.**register_python**(*function_name: str*, *function*) → None
Register Python function as a User Defined Function (UDF).

Parameters

- **function_name** (*str*) – Name of the User Defined Function.
- **function** (*callable, python function*) – Function to be used as a UDF.

See also:

register_numpy Register Numpy function as a User Defined Function.

Notes

Python functions are "vectorized" before use, via `numpy.vectorize`. For better performance, please try to use numpy UDFs, operating in terms of numpy arrays. See *vinum.register_numpy()*.

Function would be invoked for individual rows of the Table.

Any python packages used inside of the function should be imported before the invocation.

Function names are case insensitive.

Examples

Using lambda as a UDF:

```
>>> import vinum as vn
>>> vn.register_python('cube', lambda x: x**3)
>>> tbl = vn.Table.from_pydict({'len': [1, 2, 3], 'size': [7, 13, 17]})
>>> tbl.sql_pd('SELECT cube(size) from t ORDER BY cube(size) DESC')
   cube
0  4913
1  2197
2   343
```

```
>>> 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
```

Using regular python function:

```
>>> import vinum as vn
>>> def sin_taylor(x):
...     "Taylor series approximation of the sine trig function around 0."
...     return x - x**3/6 + x**5/120 - x**7/5040
...
>>> vn.register_python('sin', sin_taylor)
>>> tbl = vn.Table.from_pydict({'x': [1, 2, 3], 'y': [7, 13, 17]})
>>> tbl.sql_pd('select sin(x) as sin_x, sin(y) as sin_y from t '
...            'order by sin_y')
      sin_x     sin_y
0  0.141120 -0.961397
1  0.909297  0.420167
2  0.841471  0.656987
```

3.4 API reference.

Vinum API reference.

3.4.1 Table

Vinum Table class.

class vinum.**Table**(*arrow_table: pyarrow.lib.Table*, *reader=None*)

Table represents a tabular dataset and provides SQL SELECT interface for data manipulation. Consists of a set of named columns of equal length. Essentially, is the same abstraction as a table in the relational databases world.

Provides minimum overhead transfer to and from Pandas DataFrame as well as Arrow Table, powered by Apache Arrow framework.

There are two major ways to instantiate *Table*:

1. By invoking `Table.from_*` factory methods.
2. **By using convenice functions, such as:** *vinum.read_csv()*, *vinum.read_parquet()*, *vinum.read_json()*.

By default, all the Numpy functions are available via 'np.*' namespace.

User Defined Function can be registered via *vinum.register_python()* or *vinum.register_numpy()*.

Parameters **arrow_table** (*pyarrow.Table*) – Arrow Table containing the dataset

Examples

```
>>> import pyarrow as pa
>>> import vinum as vn
>>> data = {'col1': [1, 2, 3], 'col2': [7, 13, 17]}
>>> arrow_table = pa.Table.from_pydict(data)
>>> tbl = vn.Table(arrow_table)
>>> tbl.sql_pd('select * from t')
   col1  col2
0     1     7
```

(continues on next page)

(continued from previous page)

```
1     2    13
2     3    17
```

```
>>> import pandas as pd
>>> import vinum as vn
>>> pdf = pd.DataFrame(data={'col1': [1, 2, 3], 'col2': [7, 13, 17]})
>>> tbl = vn.Table.from_pandas(pdf)
>>> tbl.sql('select * from t')
<vinum.core.table.Table object at 0x114cff7f0>
```

Notice that *vinum.Table.sql()* returns *vinum.Table* object type.

```
>>> tbl.sql_pd('select * from t')
   col1  col2
0     1     7
1     2    13
2     3    17
```

Notice that *vinum.Table.sql_pd()* returns `pandas.DataFrame`.

To register a Numpy UDF:

```
>>> import vinum as vn
>>> vn.register_numpy('product', lambda x, y: x*y)
>>> tbl.sql_pd('select product(col1, col2) from t')
   product
0        7
1       26
2       51
```

'product' UDF defined above, would perform vectorized multiplication on arrays, represented by columns 'col1' and 'col2'.

Attributes

schema Return schema of the table.

Methods

explain(query[, print_query_tree])	Returns a query plan as a string.
from_arrow(arrow_table)	Constructs a *Table* from `pyarrow.Table`
from_pandas(data_frame)	Constructs a *Table* from `pandas.DataFrame`
from_pydict(pydict)	Constructs a *Table* from `dict`
head(n)	Return first n rows of a table as Pandas DataFrame
sql(query)	Executes SQL SELECT query on a Table and returns the result of the query.
sql_pd(query)	Executes SQL SELECT query on a Table and returns the result of the query as a Pandas DataFrame.
to_arrow()	Convert *Table* to 'pyarrow.Table`.
to_pandas()	Convert *Table* to 'pandas.DataFrame`.
to_string()	Return string representation of a *Table*.

explain(*query: str*, *print_query_tree=False*)
Returns a query plan as a string.

Parameters

- **query** (*str*) – SQL SELECT query.
- **print_query_tree** (*bool, optional*) – Set to True to also return an AST of the query.

Returns str

Return type Query Plan.

See also:

sql Executes SQL SELECT query on a Table and returns the result of the query.

sql_pd Executes SQL SELECT query on a Table and returns the result of the query as a Pandas Table.

Examples

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> print(tbl.explain('select to_int(fare_amount) fare, count(*) from t '
...                   'group by fare order by fare limit 3'))
```

Query plan:

Operator: MaterializeTableOperator

Operator: SliceOperator

Operator: ProjectOperator

Column: to_int_4304514592 Column: count_star_4556372144

Operator: SortOperator

Column: to_int_4304514592

Operator: AggregateOperator

Operator: ProjectOperator

VectorizedExpression: IntCastFunction Column: fare_amount

Operator: ProjectOperator

Column: fare_amount

Operator: TableReaderOperator

classmethod from_arrow(*arrow_table: pyarrow.lib.Table*)
Constructs a *Table* from `pyarrow.Table`
Parameters **arrow_table** (*pyarrow.Table* object) –
Returns Vinum Table instance.
Return type *vinum.Table*

Examples

```
>>> import pyarrow as pa
>>> import vinum as vn
>>> data = {'col1': [1, 2, 3], 'col2': [7, 13, 17]}
>>> arrow_table = pa.Table.from_pydict(data)
>>> tbl = vn.Table.from_arrow(arrow_table)
>>> tbl.sql_pd('select * from t')
   col1  col2
0     1     7
1     2    13
2     3    17
```

classmethod from_pandas(*data_frame*)

Constructs a *Table* from `pandas.DataFrame`

Parameters **data_frame** (*pandas.DataFrame* object) –

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> 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')
   col1  col2
0     1     7
1     2    13
2     3    17
```

classmethod from_pydict(*pydict: Dict*)

Constructs a *Table* from `dict`

Parameters **pydict** (*Python dictionary*) –

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> import vinum as vn
>>> data = {'col1': [1, 2, 3], 'col2': [7, 13, 17]}
>>> tbl = vn.Table.from_pydict(data)
>>> tbl.sql_pd('select * from t')
   col1  col2
0     1     7
1     2    13
2     3    17
```

head(*n: int*) → pd.DataFrame

Return first n rows of a table as Pandas DataFrame

Parameters **n** (*int*) – Number of first rows to return.

Returns
Return type pandas.DataFrame

property schema
Return schema of the table.
Returns
Return type pyarrow.Schema

sql(*query: str*)
Executes SQL SELECT query on a Table and returns the result of the query.
Parameters **query** (*str*) – SQL SELECT query.
Returns Vinum Table instance.
Return type *vinum.Table*
See also:

sql_pd Executes SQL SELECT query on a Table and returns the result of the query as a Pandas DataFrame.

Notes

Only SELECT statements are supported. For SELECT statements, JOINs and subqueries are currently not supported. However, optimizations aside, one can run a subsequent query on the result of a query, to model the behaviour of subqueries.

Table name in 'select * from table' clause is ignored. The table of the underlying DataFrame is used to run a query.

By default, all the Numpy functions are available via 'np.*' namespace.

User Defined Function can be registered via *vinum.register_python()* or *vinum.register_numpy()*

Examples

Using 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)
>>> res_tbl = tbl.sql('select * from t')
>>> res_tbl.to_pandas()
   col1  col2
0     1     7
1     2    13
2     3    17
```

Running queries on a csv file:

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> res_tbl = tbl.sql('select key, fare_amount from t limit 3')
>>> res_tbl.to_pandas()
                           key  fare_amount
```

(continues on next page)

(continued from previous page)

```
0   2009-06-15 17:26:21.0000001          4.5
1   2010-01-05 16:52:16.0000002         16.9
2  2011-08-18 00:35:00.00000049          5.7
```

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> res_tbl = tbl.sql('select to_int(fare_amount) fare, count(*) from t '
...                   'group by fare order by fare limit 3')
>>> res_tbl.to_pandas()
   fare  count_star
0    -5           1
1    -3           1
2    -2           4
```

sql_pd(*query: str*)

Executes SQL SELECT query on a Table and returns the result of the query as a Pandas DataFrame.

This is a convience method which runs **:method:`vinum.Table.sql`** and then calls **:method:`vinum.Table.to_pandas`** on the result. Equivalent to:

```
>>> res_tbl = tbl.sql('select * from t')
>>> res_tbl.to_pandas()
   col1  col2
0     1     7
1     2    13
2     3    17
```

Parameters **query** (*str*) – SQL SELECT query.
Returns Pandas DataFrame.
Return type pandas.DataFrame

See also:

sql Executes SQL SELECT query on a Table and returns the result of the query.

Notes

Only SELECT statements are supported. For SELECT statements, JOINs and subqueries are currently not supported. However, optimizations aside, one can run a subsequent query on the result of the query, to model the behaviour of subqueries.

Table name in `select * from table` clause is ignored. The table of the underlying Table object is used to run a query.

Examples

Using 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')
   col1  col2
0     1     7
1     2    13
2     3    17
```

Running queries on a csv file:

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> tbl.sql_pd('select key, passenger_count from t limit 3')
                           key  passenger_count
0   2009-06-15 17:26:21.0000001                1
1   2010-01-05 16:52:16.0000002                1
2  2011-08-18 00:35:00.00000049                2
```

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> res_tbl = tbl.sql('select to_int(fare_amount) fare, count(*) from t '
...                   'group by fare order by fare limit 3')
>>> res_tbl.to_pandas()
   fare  count_star
0    -5           1
1    -3           1
2    -2           4
```

to_arrow() → pyarrow.lib.Table

Convert *Table* to ‘pyarrow.Table`.

Returns

Return type pyarrow.Table

to_pandas()

Convert *Table* to ‘pandas.DataFrame`.

Returns

Return type pandas.DataFrame

to_string() → str

Return string representation of a *Table*.

Returns

Return type str

3.4.2 Stream Reader

Vinum Stream Reader class.

class vinum.**StreamReader**(*reader*)

StreamReader represents a stream of data which is used is an input for query processor.

Since input file may not fit into memory, StreamReader is the recommended way to execute queries on large files.

StreamReader instances are created by vinum.stream_* functions, for example: *vinum.stream_csv()*.

Parameters **reader** (*pa.RecordBatchFileReader*) – Arrow Stream Reader

Attributes

reader

Methods

sql(query)	Executes SQL SELECT query on an input stream and return the result as a Table materialized in memory.

sql(*query: str*)

Executes SQL SELECT query on an input stream and return the result as a Table materialized in memory.

Parameters **query** (*str*) – SQL SELECT query.

Returns Vinum Table instance.

Return type *vinum.Table*

See also:

sql_pd Executes SQL SELECT query on a Table and returns the result of the query as a Pandas DataFrame.

Notes

Only SELECT statements are supported. For SELECT statements, JOINs and subqueries are currently not supported. However, optimizations aside, one can run a subsequent query on the result of a query, to model the behaviour of subqueries.

Table name in 'select * from table' clause is ignored. The table of the underlying DataFrame is used to run a query.

By default, all the Numpy functions are available via 'np.*' namespace.

User Defined Function can be registered via *vinum.register_python()* or *vinum.register_numpy()*

Examples

Run aggregation query on a csv stream:

```
>>> import vinum as vn
>>> query = 'select passenger_count pc, count(*) from t group by pc'
>>> vn.stream_csv('taxi.csv').sql(query).to_pandas()
   pc  count
0   0    165
1   5   3453
2   6    989
```

(continues on next page)

(continued from previous page)

```
3   1  34808
4   2   7386
5   3   2183
6   4   1016
```

3.4.3 Input/Output functions

List of Input/Output functions and factory methods.

Table.from_arrow

classmethod Table.**from_arrow**(*arrow_table: pyarrow.lib.Table*)

Constructs a *Table* from `pyarrow.Table`

Parameters **arrow_table** (*pyarrow.Table* object) –

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> import pyarrow as pa
>>> import vinum as vn
>>> data = {'col1': [1, 2, 3], 'col2': [7, 13, 17]}
>>> arrow_table = pa.Table.from_pydict(data)
>>> tbl = vn.Table.from_arrow(arrow_table)
>>> tbl.sql_pd('select * from t')
   col1  col2
0     1     7
1     2    13
2     3    17
```

Table.from_pandas

classmethod Table.**from_pandas**(*data_frame*)

Constructs a *Table* from `pandas.DataFrame`

Parameters **data_frame** (*pandas.DataFrame* object) –

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> 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')
   col1  col2
```

(continues on next page)

(continued from previous page)

```
0     1     7
1     2    13
2     3    17
```

stream_csv

vinum.**stream_csv**(*input_file*, *read_options=None*, *parse_options=None*, *convert_options=None*)

Open a streaming reader of CSV data.

Reading using this function is always single-threaded.

This function is a thin convenience wrapper around `pyarrow.csv.open_csv`, which returns `StreamReader`.

Parameters

- **input_file** (*string, path or file-like object*) – The location of CSV data. If a string or path, and if it ends with a recognized compressed file extension (e.g. ".gz" or ".bz2"), the data is automatically decompressed when reading.
- **read_options** (*pyarrow.csv.ReadOptions, optional*) – Options for the CSV reader (see pyarrow.csv.ReadOptions constructor for defaults)
- **parse_options** (*pyarrow.csv.ParseOptions, optional*) – Options for the CSV parser (see pyarrow.csv.ParseOptions constructor for defaults)
- **convert_options** (*pyarrow.csv.ConvertOptions, optional*) – Options for converting CSV data (see pyarrow.csv.ConvertOptions constructor for defaults)

Returns

Return type *StreamReader*

Examples

Run aggregation query on a csv stream:

```
>>> import vinum as vn
>>> query = 'select passenger_count pc, count(*) from t group by pc'
>>> vn.stream_csv('taxi.csv').sql(query).to_pandas()
   pc  count
0   0    165
1   5   3453
2   6    989
3   1  34808
4   2   7386
5   3   2183
6   4   1016
```

read_csv

vinum.**read_csv**(*input_file*, *read_options=None*, *parse_options=None*, *convert_options=None*, *memory_pool=None*) → *vinum.api.table.Table*

Read a *Table* from a stream of CSV data.

This function is a thin convenience wrapper around `pyarrow.csv.read_csv`, which returns `Table`.

Parameters

- **input_file** (*string, path or file-like object*) – The location of CSV data. If a string or path, and if it ends with a recognized compressed file extension (e.g. ".gz" or ".bz2"), the data is automatically decompressed when reading.
- **read_options** (*pyarrow.csv.ReadOptions, optional*) – Options for the CSV reader (see pyarrow.csv.ReadOptions constructor for defaults)
- **parse_options** (*pyarrow.csv.ParseOptions, optional*) – Options for the CSV parser (see pyarrow.csv.ParseOptions constructor for defaults)
- **convert_options** (*pyarrow.csv.ConvertOptions, optional*) – Options for converting CSV data (see pyarrow.csv.ConvertOptions constructor for defaults)
- **memory_pool** (*MemoryPool, optional*) – Pool to allocate Table memory from

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> import vinum as vn
>>> tbl = vn.read_csv('taxi.csv')
>>> res_tbl = tbl.sql('select key, fare_amount from t limit 3')
>>> res_tbl.to_pandas()
                          key  fare_amount
0   2009-06-15 17:26:21.0000001          4.5
1   2010-01-05 16:52:16.0000002         16.9
2  2011-08-18 00:35:00.00000049          5.7
```

read_json

vinum.**read_json**(*input_file*, *read_options=None*, *parse_options=None*, *memory_pool=None*) → *vinum.api.table.Table*

Read a *Table* from a stream of JSON data. This function is a thin convenience wrapper around `pyarrow.csv.read_json` which returns `Table`.

Parameters

- **input_file** (*string, path or file-like object*) – The location of JSON data. Currently only the line-delimited JSON format is supported.
- **read_options** (*pyarrow.json.ReadOptions, optional*) – Options for the JSON reader (see ReadOptions constructor for defaults)
- **parse_options** (*pyarrow.json.ParseOptions, optional*) – Options for the JSON parser (see ParseOptions constructor for defaults)
- **memory_pool** (*MemoryPool, optional*) – Pool to allocate Table memory from

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> import vinum as vn
>>> tbl = vn.read_json('test_data.json')
>>> tbl.sql_pd('select * from t limit 3')
   id  origin    destination    fare
0   1  London  San Francisco  1348.1
1   2  Berlin         London   256.3
2   3  Munich         Malaga   421.7
```

read_parquet

vinum.**read_parquet**(*source*, *columns=None*, *use_threads=True*, *metadata=None*, *use_pandas_metadata=False*, *memory_map=False*, *read_dictionary=None*, *filesystem=None*, *filters=None*, *buffer_size=0*, *partitioning='hive'*, *use_legacy_dataset=True*) → *vinum.api.table.Table*

Read a *Table* from Parquet format. This function is a thin convenience wrapper around `pyarrow.parquet.read_table`, which returns `Table`.

Parameters

- **source** (*str, pyarrow.NativeFile, or file-like object*) – If a string passed, can be a single file name or directory name. For file-like objects, only read a single file. Use pyarrow.BufferReader to read a file contained in a bytes or buffer-like object.
- **columns** (*list*) – If not None, only these columns will be read from the file. A column name may be a prefix of a nested field, e.g. 'a' will select 'a.b', 'a.c', and 'a.d.e'.
- **use_threads** (*bool, default True*) – Perform multi-threaded column reads.
- **metadata** (*FileMetaData*) – If separately computed
- **read_dictionary** (*list, default None*) – List of names or column paths (for nested types) to read directly as DictionaryArray. Only supported for BYTE_ARRAY storage. To read a flat column as dictionary-encoded pass the column name. For nested types, you must pass the full column "path", which could be something like level1.level2.list.item. Refer to the Parquet file's schema to obtain the paths.
- **memory_map** (*bool, default False*) – If the source is a file path, use a memory map to read file, which can improve performance in some environments.
- **buffer_size** (*int, default 0*) – If positive, perform read buffering when deserializing individual column chunks. Otherwise IO calls are unbuffered.
- **partitioning** (*Partitioning or str or list of str, default "hive"*) – The partitioning scheme for a partitioned dataset. The default of "hive" assumes directory names with key=value pairs like "/year=2009/month=11". In addition, a scheme like "/2009/11" is also supported, in which case you need to specify the field names or a full schema. See the `pyarrow.dataset.partitioning()` function for more details.
- **use_pandas_metadata** (*bool, default False*) – If True and file has custom pandas schema metadata, ensure that index columns are also loaded
- **use_legacy_dataset** (*bool, default True*) – Set to False to enable the new code path (experimental, using the new Arrow Dataset API). Among other things, this allows to pass *filters* for all columns and not only the partition keys, enables different partitioning schemes, etc.
- **filesystem** (*FileSystem, default None*) – If nothing passed, paths assumed to be found in the local on-disk filesystem.
- **filters** (*List[Tuple] or List[List[Tuple]] or None (default)*) – Rows which do not match the filter predicate will be removed from scanned data. Partition keys embedded in a nested directory structure will be exploited to avoid loading files at all if they contain no matching rows. If *use_legacy_dataset* is True, filters can only reference partition keys and only a hive-style directory structure is supported. When setting *use_legacy_dataset* to False, also within-file level filtering and different partitioning schemes are supported. Predicates are expressed in disjunctive normal form (DNF), like `[[('x', '=', 0), ...], ...]`. DNF allows arbitrary boolean logical combinations of single column predicates. The innermost tuples each describe a single column predicate. The list of inner predicates is interpreted as a conjunction (AND), forming a more selective and multiple column predicate. Finally, the most outer list combines these filters as a disjunction (OR). Predicates may also be passed as List[Tuple]. This form is interpreted as a single conjunction. To express OR in predicates, one must use the (preferred) List[List[Tuple]] notation.

Returns Vinum Table instance.

Return type *vinum.Table*

Examples

```
>>> import vinum as vn
>>> tbl = vn.read_parquet('users.parquet')
>>> tbl.sql_pd('select * from t limit 3')
    registration_dttm  id first_name  ...      usage                title
0 2016-02-03 07:55:29   1     Amanda  ...   49756.53     Internal Auditor
1 2016-02-03 17:04:03   2     Albert  ...  150280.17        Accountant IV
2 2016-02-03 01:09:31   3     Evelyn  ...  144972.51  Structural Engineer
```

[3 rows x 13 columns]

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX