

Introducing FugueSQL — SQL for Pandas, Spark, and Dask DataFrames

As a data scientist, you might be familiar with both Pandas and SQL. However, there might be some queries, transformations that you feel comfortable doing in SQL instead of Python.

Wouldn't it be nice if you can query a pandas DataFrame like below:

```
import pandas as pd

df = pd.DataFrame({"col1": [1, 2, 3, 4], "col2": ["a", "b", "c", "c"]})
df
```

	col1	col2
0	1	a
1	2	b
2	3	c
3	4	c

... using SQL?

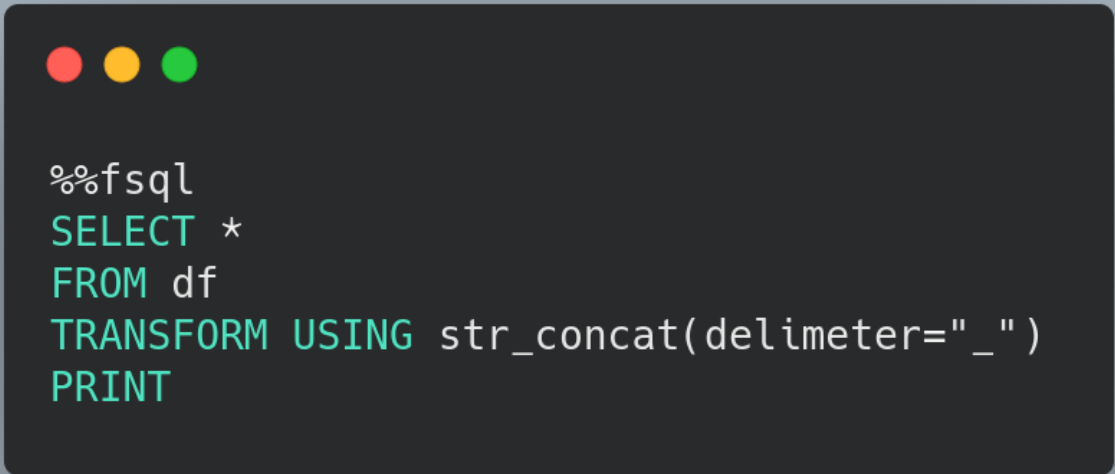
```
%%fsql
```

```
SELECT *  
FROM df  
WHERE col2="c"  
PRINT
```

	col1	col2
0	3	c
1	4	c

Or use a Python function within a SQL query?

```
# schema: *, col3:str
def str_concat(df: pd.DataFrame, delimiter: str) ->
pd.DataFrame:
    df = df.assign(
        col3=df["col1"].astype(str) + delimiter + df["col2"]
    )
    return df
```



```
%%fsql
SELECT *
FROM df
TRANSFORM USING str_concat(delimiter="_")
PRINT
```

That is when FugueSQL comes in handy.

What is FugueSQL?

FugueSQL is a Python library that allows users to combine Python code and SQL commands. This gives users the flexibility to switch between Python and SQL within a Jupyter Notebook or a Python script.

To install FugueSQL, type:

```
pip install fugue[sql]
```

To run on Spark or Dask execution engines, type:

```
pip install fugue[sql, spark]  
pip install fugue[sql, dask]  
pip install fugue[all]
```

In this article, we will explore some utilities of FugueSQL and compare FugueSQL with other tools such as pandasql.

FugueSQL in a Notebook

FugueSQL comes with a Jupyter notebook extension that allows users to interactively query DataFrames with syntax highlighting.

To use it, import the setup function from `fugue_notebook` to register the `%%fsql` cell magic.

```
from fugue_notebook import setup
setup()
```

To understand how the %%fsql cell magic, let's start with creating a pandas DataFrame:

```
import pandas as pd

df = pd.DataFrame({"col1": [1, 2, 3, 4], "col2": ["a",
"b", "c", "c"]})
df
```

	col1	col2
0	1	a
1	2	b
2	3	c
3	4	c

Now, you can query like how you would normally do in SQL by adding the `%%fsql` at the beginning of the cell.

```
%%fsql
```

```
SELECT *  
FROM df  
WHERE col2="c"  
PRINT
```

	col1	col2
0	3	c
1	4	c

schema: col1:long,col2:str

In the code above, only PRINT does not follow standard SQL. This is similar to the pandas `head()` and Spark `show()` operations to display a number of rows.

Operations such as GROUP BY are similar to standard SQL syntax.

```
%%fsql
```

```
SELECT col2, AVG(col1) AS avg_col1  
FROM df  
GROUP BY col2  
PRINT
```

	col2	avg_col1
0	a	1.0
1	b	2.0
2	c	3.5

schema: col2:str,avg_col1:double

An Enhanced SQL Interface

For SQL users, nothing shown above is out of the ordinary except for the PRINT statement. However, Fugue also adds some enhancements to standard SQL, allowing it to handle end-to-end data workflows gracefully.

Deal With Temp Tables

SQL users often have to use temp tables or common table expressions (CTE) to hold intermediate transformations. Luckily, **FugueSQL supports the creation of intermediate tables through a variable assignment.**

For example, after transforming df , we can assign it to another variable called df2 and save df2 to a file using SAVE variable OVERWRITE file_name

```
%%fsql
```

```
df2 = SELECT *  
FROM df  
WHERE col2="c"
```

```
SAVE df2 OVERWRITE '/tmp/df2.csv' (header=true)
```

Now, if we want to apply more transformation to df2 , simply load it from the file we saved previously.

```
%%fsql
```

```
df3 = LOAD '/tmp/df2.csv' (header=true)
```

```
SELECT *  
FROM df3  
PRINT
```

	col1	col2
0	3	c
1	4	c

schema: col1:str,col2:str

Pretty cool, isn't it?

Added Keywords

SQL's grammar is meant for querying, which means that it lacks keywords to manipulate data. FugueSQL adds some keywords for common DataFrame operations. For example:

- DROP

```
%%fsql
```

```
df4 = DROP COLUMNS col2 IF EXISTS FROM df  
PRINT df4
```

	col1
0	1
1	2
2	3
3	4

schema: col1:long

■ FILL NULLS PARAMS

```
import numpy as np

null_df = pd.DataFrame(
    {"col1": [np.nan, np.nan, 1],
     "col2": [2, 3, np.nan]}
)
```

```
%%fsql
-- Fill nan at col1 with 1 and nan at col2 with 2
df1 = FILL NULLS PARAMS col1:1, col2:2 FROM null_df
PRINT df1
```

	col1	col2
0	1.0	2.0
1	1.0	3.0
2	1.0	2.0

schema: col1:double,col2:double

- SAMPLE

```
%%fsql
```

```
df2 = SAMPLE 2 ROWS SEED 42 FROM df
```

```
PRINT df2
```

```
df3 = SAMPLE 50 PERCENT SEED 1 FROM df
```

```
PRINT df3
```

	col1	col2
0	2	b
1	4	c

schema: col1:long,col2:str

	col1	col2
0	4	c
1	3	c

schema: col1:long,col2:str

For a full list of operators, check the [FugueSQL operator docs](#).

Integrate With Python

FugueSQL also allows you to use Python functions within a SQL query using TRANSFORM .

For example, to use the function `str_concat` in a SQL query:

```
def str_concat(df, delimiter):  
    df = df.assign(  
        col3=df["col1"].astype(str) + delimiter + df["col2"]  
    )  
    return df
```

... simply add the following components to the function:

- Output schema hint (as a comment)
- Type annotations (`pd.DataFrame`)

```
# schema: *, col3:str  
def str_concat(df: pd.DataFrame, delimiter: str) ->  
pd.DataFrame:  
    df = df.assign(  
        col3=df["col1"].astype(str) + delimiter + df["col2"]  
    )  
    return df
```

Cool! Now we are ready to add it to a SQL query:

```
%%fsql
```

```
SELECT *
```

```
FROM df
```

```
TRANSFORM USING str_concat(delimiter="_")
```

```
PRINT
```

	col1	col2	col3
0	1	a	1_a
1	2	b	2_b
2	3	c	3_c
3	4	c	4_c

schema: col1:long,col2:str,col3:str

Scaling to Big Data

FugueSQL Spark

One of the beautiful properties of SQL is that it is agnostic to the size of the data. The logic is expressed in a scale-agnostic manner and will remain the same even if running on Pandas, Spark, or Dask.

With FugueSQL, we can apply the same logic on the Spark execution engine just by specifying `%%fsql spark`. We don't even need to edit the `str_concat` function to bring it to Spark as Fugue takes care of porting it.


```
%%fsql spark
SELECT *
FROM df
TRANSFORM USING str_concat(delimiter="_")
PRINT
```

	col1	col2	col3
0	1	a	1_a
1	2	b	2_b
2	3	c	3_c
3	4	c	4_c

schema: col1:long,col2:str,col3:str

PREPARTITION BY

One of the important parts of distributed computing is partitioning. For example, to get the median value in each logical group, the data needs to be partitioned such that each logical group lives on the same worker.

To describe this, FugueSQL has the PREPARTITION BY keyword. Fugue's prepartition-transform semantics are equivalent to the pandas groupby-apply . The only difference is that prepartition-transform scales to the distributed setting as it dictates the location of the data.

```
# schema: *
def get_median(df: pd.DataFrame) -> List[Dict[str, Any]]:
    return [
        {
            "col1": df["col1"].median(),
            "col2": df["col2"].iloc[0]}
    ]
```

```
%fsql spark
SELECT *
FROM df
TRANSFORM PREPARTITION BY col2 USING get_median
PRINT
```

	col1	col2
0	1	a
1	2	b
2	3	c

schema: col1:long,col2:str

Note that the `get_median` function above gets called once for each distinct value in the column `col2` . Because the data is partitioned beforehand, we can just pull the first value of `col2` to know what group we are working with.

FugueSQL in Production

To bring FugueSQL out of Jupyter notebooks and into Python scripts, all we need to do is wrap the FugueSQL query inside a `fsql` class. We can then call the `.run()` method and choose an execution engine.

```
from fugue_sql import fsql
import fugue_spark

fsql(
    """SELECT *
       FROM df
       TRANSFORM PREPARTITION BY col2 USING get_median
       PRINT"""
).run("spark")
```

	col1	col2
0	1	a
1	2	b
2	3	c

What Is the Difference Between FugueSQL and pandasql?

If you know pandasql, you might wonder: Why should you use FugueSQL if pandasql already allows you to run SQL with pandas?

pandasql has a single backend, SQLite. It introduces a large overhead to transfer data between pandas and SQLite. On the other hand, FugueSQL supports multiple local backends: pandas, DuckDB and SQLite.

When using the pandas backend, Fugue directly translates SQL to pandas operations, so there is no data transfer at all. DuckDB has superb pandas support, so the overhead of data transfer is also negligible. Both Pandas and DuckDB are preferred FugueSQL backends for local data processing.

Fugue also has support for Spark, Dask, and cuDF (through blazingSQL) as backends.

Conclusion

Congratulations! You have just learned how to use FugueSQL as a SQL interface for operating on Python DataFrames. With FugueSQL, you can now use SQL syntax to express end-to-end data workflows and scale to distributed computing seamlessly!

This article does not exhaustively cover FugueSQL features. For more information about Fugue or FugueSQL, check the resources below.

- [Github Repo](#)
- [FugueSQL Documentation](#)
- [Fugue Slack](#)

Feel free to play and fork the source code of this article [here](#).