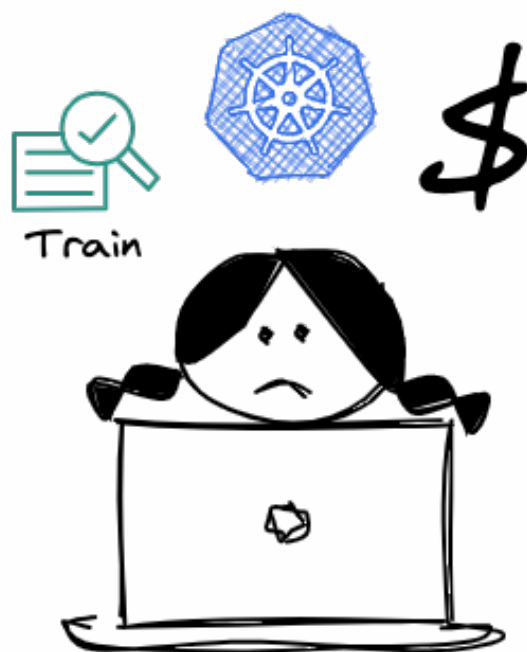


# Simplify Data Science Workflows on BigQuery with Fugue and Python

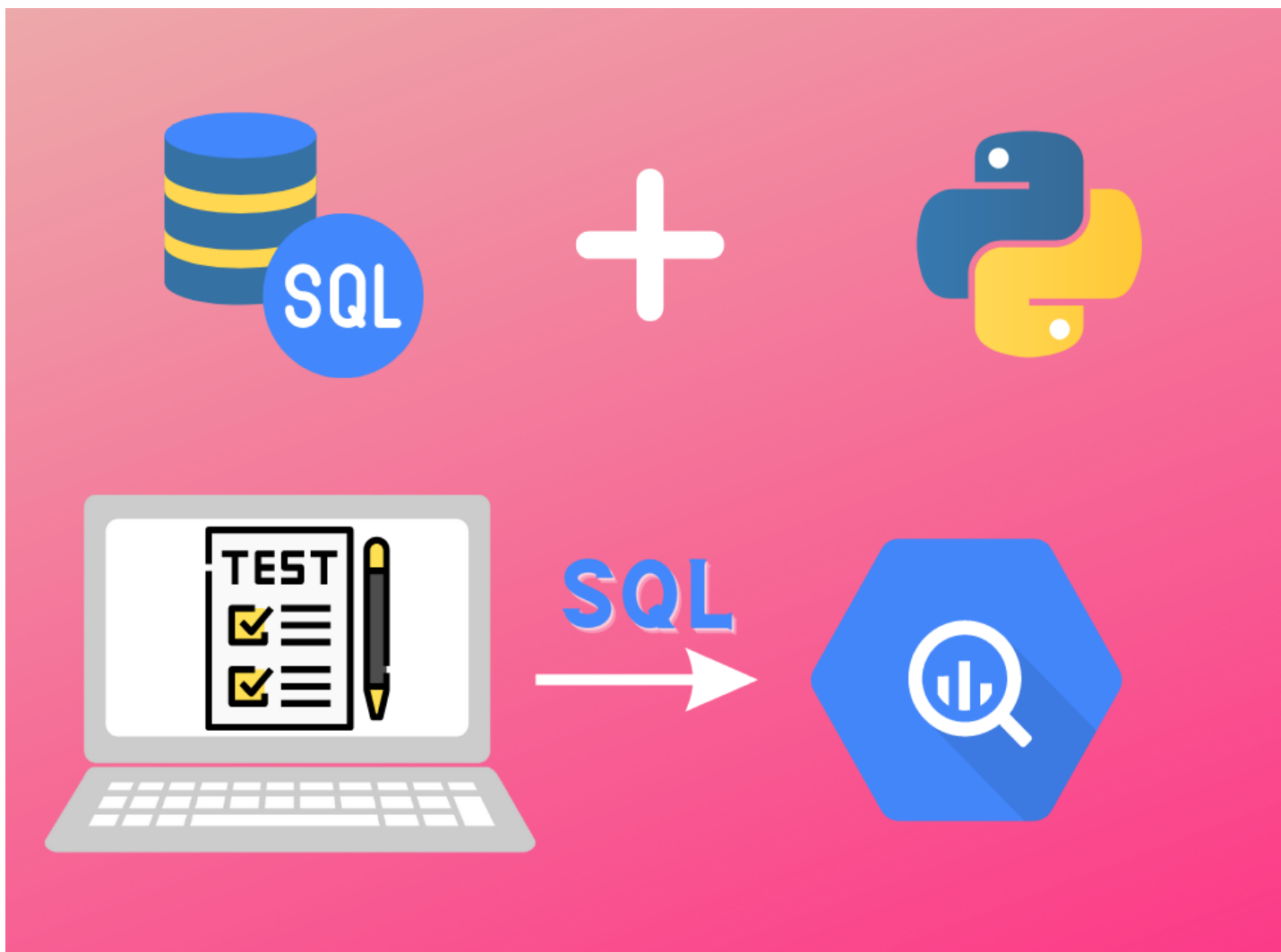
Many data teams begin by establishing an analytics practice on data warehouses such as BigQuery. However, solely relying on BigQuery for data science workloads may not be the best approach due to various reasons:

- **Advanced needs beyond SQL:** Use cases such as data validation, visualization, and machine learning forecasting may require more advanced functionalities beyond the limitations of SQL grammar.
- **Costly for exploration:** BigQuery may not be the most cost-effective solution for data science tasks due to its iterative nature, which involves extensive feature engineering and algorithm experimentation.



For data scientists working with data on BigQuery, an ideal solution would enable them to:

- Use both SQL and Python to query data from BigQuery.
- Interactively test various SQL queries locally
- Effortlessly switch back to BigQuery after thorough testing.

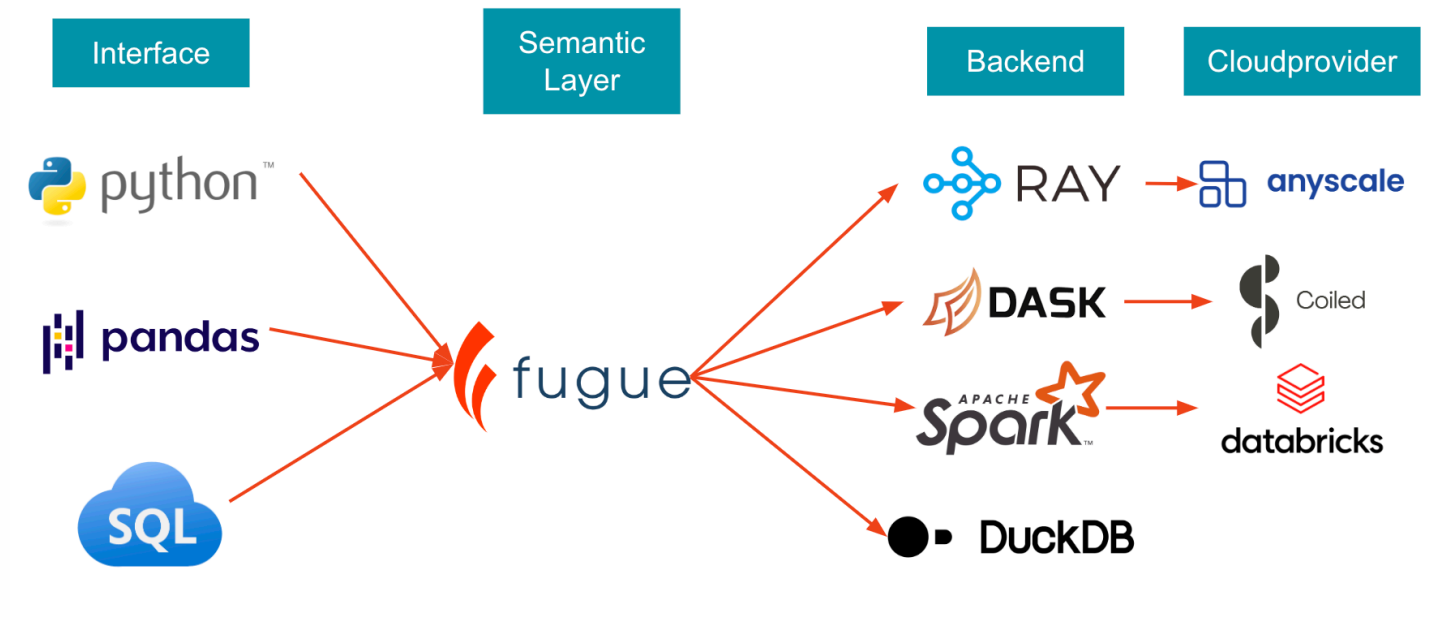


The FugueSQL BigQuery integration allows you to do exactly that.

# What is Fugue?

**Fugue** is an open-source project that simplifies big data development by porting Python, Pandas, and SQL to distributed computing engines such as Spark, Dask, and Ray.

In this article, we will use FugueSQL to simplify development on BigQuery.



## *Installation and Setup*

### **Install Fugue BigQuery**

To install Fugue BigQuery integration, type:

```
pip install fugue-warehouses[bigquery]
```

### **Authenticate to Google BigQuery**

To authenticate to Google BigQuery, the standard method is to specify the location of a credential JSON file using the `GOOGLE_APPLICATION_CREDENTIALS` environment variable.

```
import os  os.environ['GOOGLE_APPLICATION_CREDENTIALS'] =  
'path/to/your/credential.json'
```

[How to obtain the credential JSON file.](#)

## Querying a Table

Begin by setting up FugueSQL for use in Jupyter Notebook or Lab:

```
from fugue_jupyter import setup
setup(run_js=True)
```

To run SQL queries using FugueSQL BigQuery integration, simply add `%%fsql bq` at the beginning of the cell.

```
%%fsql bq
SELECT name, gender, SUM(number) AS ct
FROM `bigquery-public-data.usa_names.usa_1910_2013`
GROUP BY name, gender
PRINT
```

	name:str	gender:str	ct:long
0	Hazel	F	187622
1	Lucy	F	148270
2	Nellie	F	97063
3	Lena	F	94894
4	Thelma	F	201771
5	Ruth	F	728315
6	Elizabeth	F	1490772
7	Mary	F	3728041
8	Annie	F	273829
9	Alma	F	123875

BigQueryDataFrame: name:str,gender:str,ct:long

# Parametrization

FugueSQL allows you to use Jinja templating to parameterize your SQL queries.

The query below parametrizes the `table` variable, which is particularly helpful when transitioning from a development table to a production table.

```
table = "bigquery-public-data.usa_names.usa_1910_2013"
```

```
%%fsql bq
SELECT name, gender, SUM(number) AS ct
FROM `{{table}}`
GROUP BY name, gender
PRINT
```

	name:str	gender:str	ct:long
0	Hazel	F	187622
1	Lucy	F	148270
2	Nellie	F	97063
3	Lena	F	94894
4	Thelma	F	201771
5	Ruth	F	728315
6	Elizabeth	F	1490772
7	Mary	F	3728041
8	Annie	F	273829
9	Alma	F	123875

BigQueryDataFrame: name:str,gender:str,ct:long

## *Breaking Up Queries*

Fugue includes several improvements to standard SQL that facilitate query breakdown. These enhancements are illustrated in the query below:

- The equal sign assigns the output of the query to the `df` variable, which is subsequently reused in another operation.
- `TAKE` returns the whole row. `PREPARTITION BY` partitions the data by gender. `PRESORT` sorts the data by the `ct` column in descending order.
- `YIELD` makes the DataFrame available to subsequent Jupyter Notebook cells.

```
n = 3
```

```
%%fsql bq
df = SELECT name, gender, SUM(number) AS ct
FROM `{{table}}`
GROUP BY name, gender

names = TAKE {{n}} ROWS FROM df PREPARTITION BY gender PRESORT ct DESC
YIELD DATAFRAME
```

We can now access the `names` variable and perform additional queries in a separate Jupyter Notebook cell.

```
%%fsql bq
SELECT name FROM names
PRINT
```

	name:str
0	James
1	John
2	Robert
3	Mary
4	Patricia
5	Elizabeth

BigQueryDataFrame: name:str



# *FugueSQL Python Extensions*

## **TRANSFORM**

FugueSQL allows you to integrate Python functions into your SQL queries by using the **TRANSFORM** keyword.

The following query shows how the `get_decade` function is applied to the data using TRANSFORM to generate a new column named `decade`.

```
import pandas as pd

# schema: *, decade:long
def get_decade(df: pd.DataFrame) -> pd.DataFrame:
    return df.assign(decade=df["year"] // 10 * 10)
```

```
%%fsql bq
SELECT *
FROM `{{table}}`
WHERE name IN (SELECT name FROM names)
TRANSFORM USING get_decade()
YIELD LOCAL DATAFRAME AS decade_df
PRINT
```

	state:str	gender:str	year:long	name:str	number:long	decade:long
0	IL	F	1910	Mary	1076	1910
1	LA	F	1910	Elizabeth	100	1910
2	MS	F	1910	Mary	762	1910
3	MD	F	1911	Mary	425	1910
4	MI	F	1911	Mary	456	1910
5	MN	F	1911	Mary	267	1910
6	MO	F	1913	Mary	971	1910
7	CO	F	1914	Elizabeth	82	1910
8	ND	F	1914	Mary	142	1910
9	VT	F	1914	Mary	106	1910

PandasDataFrame: state:str,gender:str,year:long,name:str,number:long,decade:long

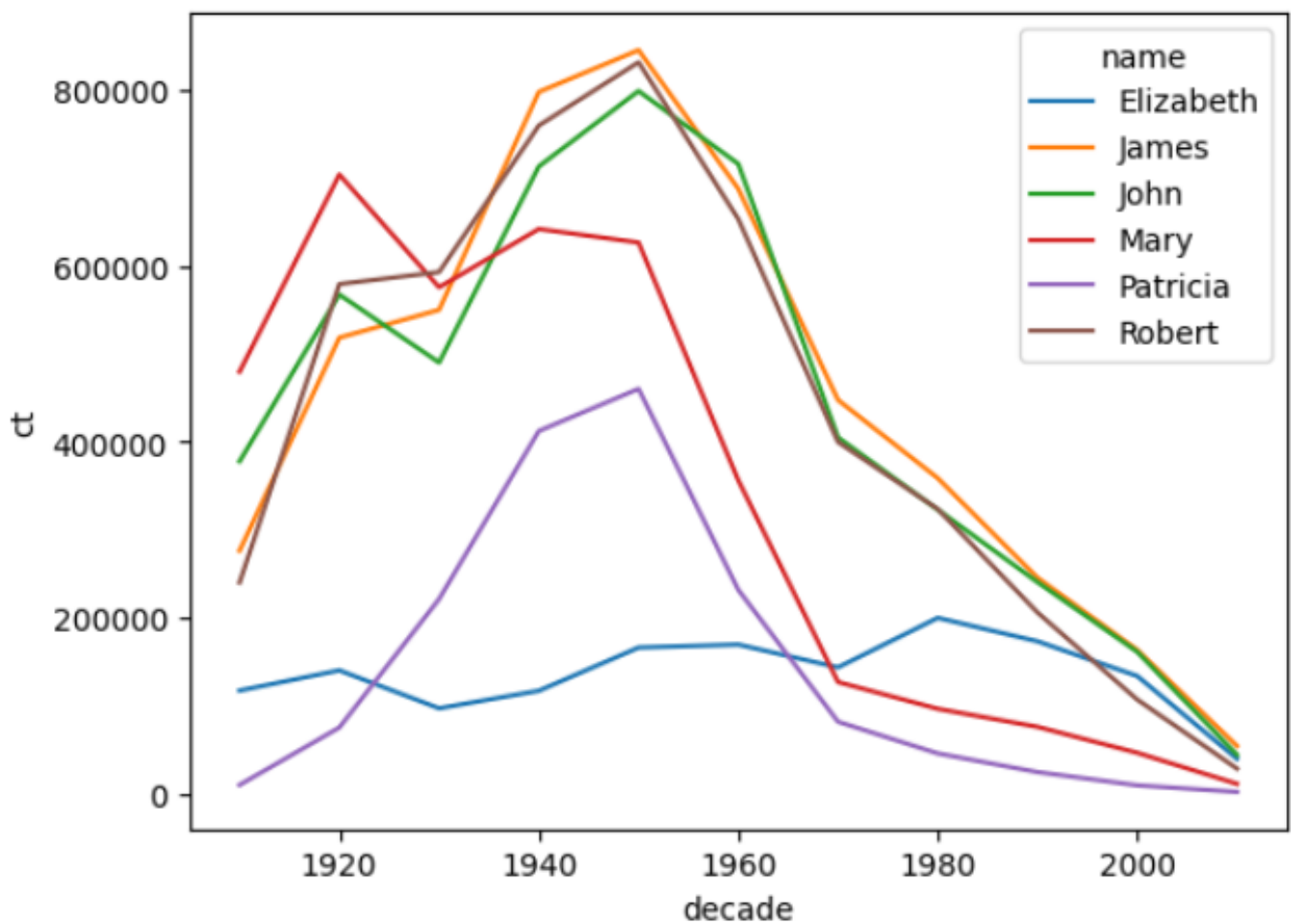
With TRANSFORM, data scientists can now perform feature engineering on data and utilize new machine learning models on BigQuery.

# OUTPUT

FugueSQL also provides built-in extensions that facilitate integration with other plotting libraries, such as seaborn.

In the code below, `OUTPUT USING sns:lineplot()` brings the query results to pandas and then generates the line plot using seaborn.

```
%%fsql
SELECT name, decade, SUM(number) AS ct
FROM decade_df
GROUP BY name, decade
ORDER BY decade
OUTPUT USING sns:lineplot(x="decade",y="ct",hue="name")
```



## *Productionizing SQL Queries*

To transition from notebooks to production, we can eliminate the intermediate YIELD statements and pass the query as a string to the `fugue_sql()` function.

```
import fugue.api as fa
res = fa.fugue_sql("""
SELECT name, gender, SUM(number) AS ct
  FROM `{{table}}`
  GROUP BY name, gender

names = TAKE {{n}} ROWS PREPARTITION BY gender PRESORT ct
DESC

SELECT name, year, SUM(number) AS ct
  FROM `{{table}}`
 WHERE name IN (SELECT name FROM names)
  GROUP BY name, year
  ORDER BY year
""", engine="bq", table=table, n=n)
```

We can then use the `as_pandas()` function to convert the output to pandas for further analysis.

```
fa.as_pandas(res).head()
```

	name	year	ct
0	John	1910	11479
1	Patricia	1910	260
2	Mary	1910	22906
3	Elizabeth	1910	5791
4	James	1910	9202

# Iterating on Big Data

FugueSQL offers the capability to sample a BigQuery table into a smaller DataFrame using the `SAMPLE` keyword. This speeds up the iteration process and avoids the need to work with the complete dataset every time.

Here, the `YIELD` keyword is used again to make the test DataFrame available.

```
%%fsql bq
SELECT *
FROM `{{table}}`
SAMPLE 1 PERCENT
YIELD LOCAL DATAFRAME AS test
```

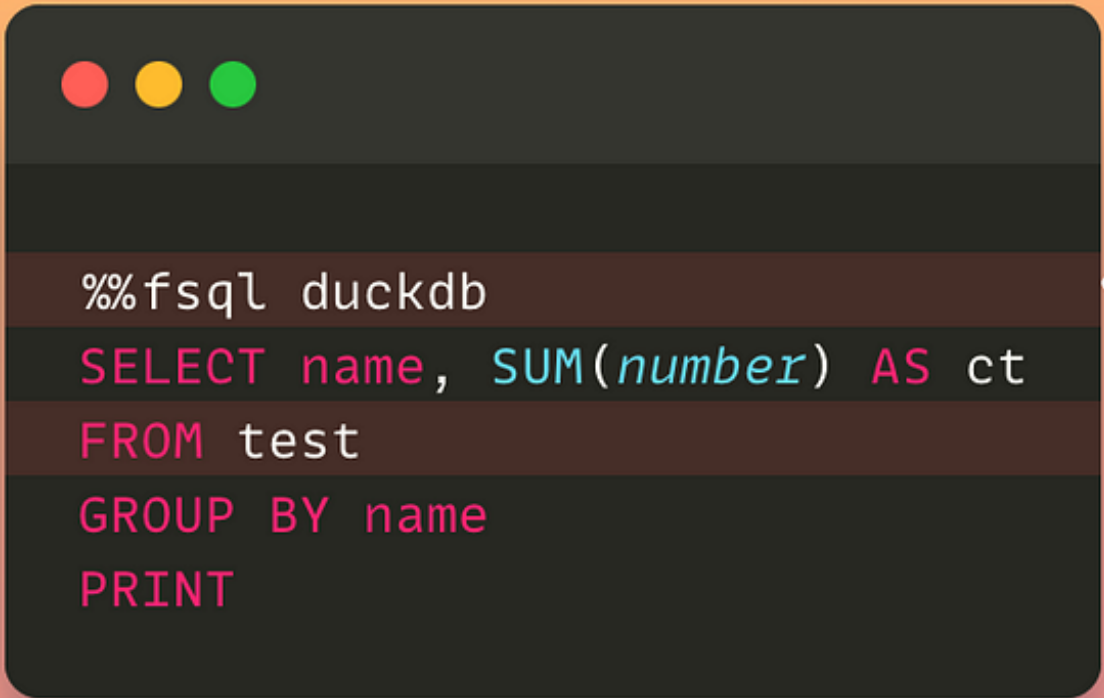
In the next cell, we can test the query on the test DataFrame using FugueSQL with [DuckDB](#) backend to speed up the code.

```
%%fsql duckdb
SELECT name, SUM(number) AS ct
FROM test
GROUP BY name
PRINT
```

	name:str	ct:long
0	Pasquale	913
1	Franklin	13262
2	Earnest	6032
3	Will	2332
4	Kim	22036
5	Willis	4437
6	Irving	4126
7	Fransisco	346
8	Mickey	2499
9	Trenton	4581

DuckDataFrame: name:str,ct:long

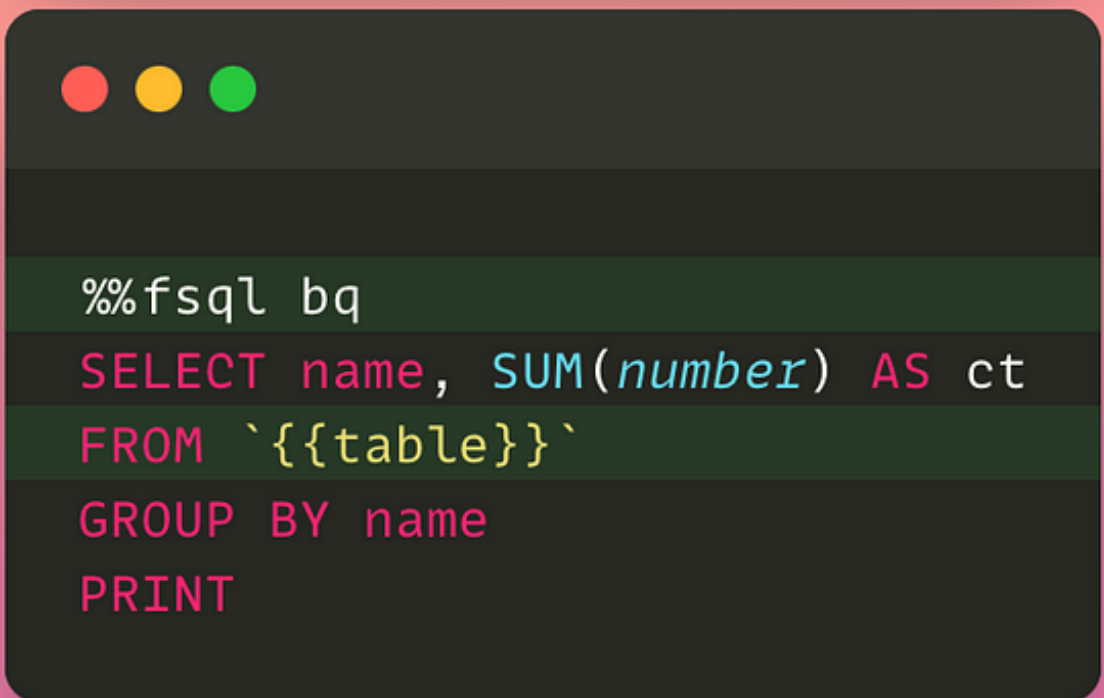
Once the query has undergone thorough testing, switching the engine to bq is a simple task.



```
%%fsql duckdb
```

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a SQL query for duckdb. A white arrow points from the right side of this window to the terminal window below it.

```
SELECT name, SUM(number) AS ct  
FROM test  
GROUP BY name  
PRINT
```



```
%%fsql bq
```

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a SQL query for bq. A white arrow points from the right side of the terminal window above it to this window.

```
SELECT name, SUM(number) AS ct  
FROM `{{table}}`  
GROUP BY name  
PRINT
```



## *Combining BigQuery with Spark, Dask, or Ray*

If you are dealing with vast amounts of data in BigQuery, it may be too slow for a single machine to handle. Fugue provides a convenient way to integrate BigQuery with distributed computing frameworks such as Spark, Dask, and Ray.

In the query below, the `transform()` function applies the `median` function in a distributed manner to each partition of a Dask DataFrame.

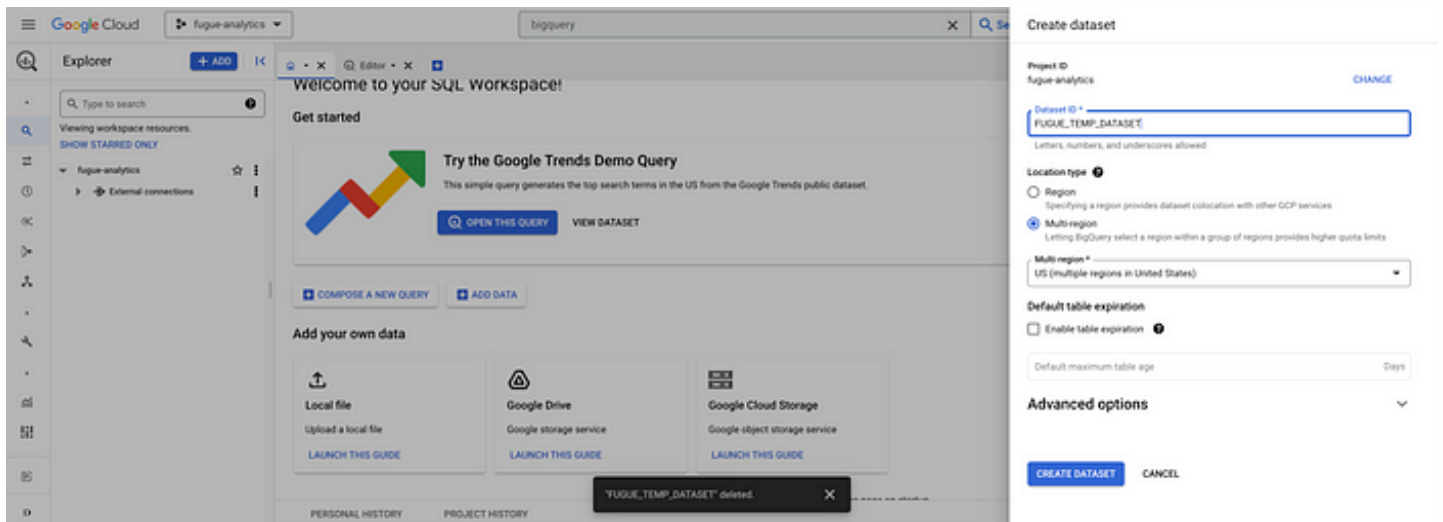
```
import pandas as pd
from typing import List, Any

# schema: *
def median(df:pd.DataFrame) -> List[List[Any]]:
    return [[df.state.iloc[0], df.number.median()]]

fa.transform(
    ("bq", """SELECT state, number
    FROM `bigquery-public-data.usa_names.usa_1910_2013`
    TABLESAMPLE SYSTEM (1 PERCENT)"""),
    median,
    partition="state",
    engine="dask"
).compute().head()
```

	state	number
0	AK	9
1	AL	13
2	AR	13
3	AZ	12
4	CA	13

When the query is executed, the data is automatically persisted in a temporary dataset. By default, the dataset is named FUGUE\_TEMP\_DATASET and must be created in the BigQuery workspace, as shown below.



# Conclusion

---

Congratulations! You have just learned how to quickly iterate on data that lives in BigQuery with FugueSQL. Fugue's Python and SQL interoperability provides a frictionless developer experience with minimal boilerplate code.