



Duckdb for Data Scientists:

Handling large data files in Jupyter Notebook

TABLE OF CONTENTS

01

What is
Duckdb

02

Overview of the
Python API

03

How to use Duckdb into your
data science workflow



01

What is Duckdb?

WHAT IS DUCKDB?

DuckDB is an in-process, embedded SQL OLAP database designed for analytical workloads.

DuckDB is extremely lightweight and easy to deploy without the need for managing a separate database server.



IDEAL USE CASES

- **Data Analytics:** great performance and SQL features needed for effective data analysis.
- **Prototyping and Development:** Quickly spin up a local Duckdb instance for testing and prototyping.
- **Data Science and Machine Learning:** seamlessly integration with Pandas make it a natural choice for data scientists.



KEY FEATURES

- **In-memory:** DuckDB operates in-memory, making it lightning fast.
- **Embedded:** it runs directly in your application without the need of a database server.
- **SQL Support:** Supports a rich subset of SQL, ideal for data analysis.
- **Columnar Storage:** Enables fast, vectorized query execution, optimized for analytical workloads.



DATA SOURCES

- AWS S3 buckets and storage with S3-compatible API
- Azure Blob Storage
- Cloudflare R2
- CSV
- Delta Lake
- Excel (via the spatial extension)
- httpfs
- Iceberg
- JSON
- MySQL
- Parquet
- PostgreSQL
- SQLite



02

Python API Overview

Duckdb Connection



```
# Install
# !pip install duckdb

import duckdb

# In-memory database
conn = duckdb.connect()

# Or file-based database
# conn = duckdb.connect(database='filename.db')
```

Data Input



```
# read files into a Relation
duckdb.read_csv("file.csv")
duckdb.read_parquet("file.parquet")
duckdb.read_json("file.json")

# query directly from a file
duckdb.sql("SELECT * FROM 'file.csv'")
duckdb.sql("SELECT * FROM 'file.parquet'")
duckdb.sql("SELECT * FROM 'file.json'")
```

Result Conversion



```
# convert query results to a variety of formats

duckdb.sql("SELECT 42").fetchall() # Python objects
duckdb.sql("SELECT 42").df() # Pandas DataFrame
duckdb.sql("SELECT 42").pl() # Polars DataFrame
duckdb.sql("SELECT 42").arrow() # Arrow Table
duckdb.sql("SELECT 42").fetchnumpy() # NumPy Arrays
```

Loading from CSV files




```
# create in-memory database
```

```
conn = duckdb.connect()
```

```
# read csv and store it into a pandas dataframe
```

```
df = conn.execute("""select * from 'filename.csv' """).df()
```

Loading from CSV files



```
# load CSV with options
df = conn.execute(""" SELECT * FROM read_csv('file.csv',
        delim = '|',
        header = true,
        columns = { 'column1': 'DATE',
                    'column2': 'VARCHAR'
                  });
""").df()
```

Loading from Multiple CSV files



```
# read multiple files
df = conn.execute("SELECT * FROM '*.csv' ").df()

# read from a list of files
df = conn.execute("
    SELECT * FROM
    read_csv(['file1.csv', 'file2.csv'],
    union_by_name = true,
    filename = true);
")
```

Loading from Compressed CSV files



```
# read from compressed CSV files
```

```
df = conn.execute(""" SELECT * FROM 'compressed_file.gz' """).df()
```

Create views from CSV files



```
# keep the data stored inside the CSV file and query it directly from the file
conn.execute(" CREATE VIEW  my_view  AS SELECT * FROM READ_CSV('file.csv'); ")
df = conn.execute(" SELECT * FROM my_view ").df()
```


Query Metadata of a Parquet file



```
# Query Parquet metadata
df = conn.execute(""" SELECT *
FROM parquet_metadata('file.parquet');
""").df()
```

Write query results to a Parquet file



```
# Export query results to a parquet file
conn.execute(""" COPY
                (SELECT count(1) FROM my_table_or_view)
                TO 'file.parquet'
                (FORMAT 'parquet', COMPRESSION 'zstd'); """)
```

Export database to a Parquet file



```
# export database to parquet file  
conn.execute(""" EXPORT DATABASE 'directory' (FORMAT PARQUET); """)
```

https://duckdb.org/docs/guides/performance/file_formats.html#parquet-file-sizes

Insert into a table



```
# insert into a table
conn.execute ("""CREATE TABLE staff (id INTEGER, full_name VARCHAR, role_id
              INTEGER); INSERT INTO staff VALUES (1, 'John Doe', 5); """)
```

Avoid using lots of INSERT statements when inserting more than a few rows, loading data in a loop and executing the statements in auto-commit mode.

Query DataFrames Directly



```
import pandas as pd
import polars as pl
import pyarrow as pa

pandas_df = pd.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM pandas_df")

polars_df = pl.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM polars_df")

arrow_table = pa.Table.from_pydict({"a": [42]})
duckdb.sql("SELECT * FROM arrow_table")
```

Write Data to Disk



```
# write data to disk using SQL
```

```
duckdb.sql("SELECT 42").write_parquet("file.parquet") # Write to a Parquet file
```

```
duckdb.sql("SELECT 42").write_csv("file.csv") # Write to a CSV file
```

```
duckdb.sql("COPY (SELECT 42) TO 'file.parquet'") # Copy to a Parquet file
```

Using Prepared Statements

```
# prepared statements

conn.execute("INSERT INTO staff VALUES (?, ?, ?)", [3, "Mary Jane", 1])

# insert several rows using a prepared statement
conn.executemany("INSERT INTO staff VALUES (?, ?, ?)",
                 [[4, "Anti Mary", 10], [5, "Mark Williams", 2]] )

# query the database using a prepared statement:
conn.execute("SELECT full_name FROM staff WHERE role_id = ?", [10])
print(conn.fetchall())

#Query using the $ notation for a prepared statement and reused values
conn.execute("SELECT $1, $1, $2", ["duck", "goose"])
print(conn.fetchall())
[('duck', 'duck', 'goose')]
```

Using Named Parameters

```
# named parameters
```

```
result = duckdb.execute("""  
    SELECT  
    $my_param,  
    $other_param  
    """,  
    {  
        "my_param": 5,  
        "other_param": "DuckDB"  
    })  
    .fetchall()  
  
print(result)  
[(5, 'DuckDB')]
```


Except



select all rows in the first relation that do not exist in the second relation

```
rel_1 = conn.sql(" SELECT * FROM range(10) ")
```

```
rel_2 = conn.sql(" SELECT * FROM range(5) ")
```

```
rel_1.except_(rel_2).show()
```

Filter



```
# filter rows that do not satisfy the condition.
```

```
rel = conn.sql(" SELECT * FROM range(10) tbl(table_name) ")  
rel.filter("table_name > 5").limit(3).show()
```

Intersect



```
# return all rows that occur in both relations
```

```
rel_1 = conn.sql("SELECT * FROM range(10) tbl(table1)")
```

```
rel_2 = conn.sql("SELECT * FROM range(5) tbl(table2)")
```

```
rel_1.intersect(rel_2).show()
```

Join



```
# join relations based on the provided condition.
```

```
rel_1 = conn.sql(" SELECT * FROM range(5) tbl(table1) ").set_alias("rel_1")  
rel_2 = conn.sql(" SELECT * FROM range(0, 20) tbl(table2) ").set_alias("rel_2")  
rel_1.join(rel_2, " rel_1.table1 + 10 = rel_2.table2 ").show()
```



03

Using Duckdb into your Data Science Workflow

DEMO

Why you should use Duckdb?

1. **Ease of Use:** There's no need to set up and manage a separate database server.
2. **Performance:** DuckDB is designed for high-performance analytical queries, even on large datasets.
3. **Flexibility:** DuckDB can handle both in-memory and persistent data, making it suitable for various use cases, from temporary data analysis to long-term data storage.
4. **Scalability:** While it's lightweight and embedded, DuckDB can scale to handle very large datasets, especially when working with data stored in formats like Parquet.
5. **Integration with Data Science Workflows:** Duckdb allows you to use SQL directly on your data without the overhead of transferring data between different systems.
6. **No Server Overhead:** There's no need to worry about managing server resources, connections, or network latency.

Why you should NOT use Duckdb?

- **Transactional Processes:** As DuckDB is an analytics database, it has minimal support for transactions and parallel write access. Therefore, you couldn't use it in applications and APIs that process and store input data arriving arbitrarily.
- **Limited Memory:** While Duckdb supports spilling over memory (out-of-memory processing) to disk, that feature is aimed more at exceptional situations, where the final portion of processing won't fit into memory. In most cases, that means you'll have a limit of a few hundred gigabytes for processing, with not all of it needing to be in memory at the same time, as DuckDB optimizes loading only what's needed.
- **Big Data:** DuckDB focuses on the long tail of data analytics use cases, so if you're in an enterprise environment with a complex setup of data sources, tools, and applications processing many terabytes of data, DuckDB might not be the right choice for you.
- **Streaming Data:** DuckDB does not support processing live data streams that update continuously. Data updates should happen in bulk by loading new tables or large chunks of new data at once.

Duckdb too slow?

1. **Do you have enough memory?** DuckDB works best if you have 5-10 GB memory per CPU core.
2. **Are you using a fast disk?** Network-attached disks (such as cloud block storage) cause write-intensive and large work-loads to slow down.
3. **Are you using indexes or constraints (primary key, unique, etc.)?** If possible, try disabling them.
4. **Are you using the correct types?** i.e. use TIMESTAMP for datetime
5. **Are you reading from Parquet files?** Do they have row group sizes between 100k and 1M and file sizes between 100 MB to 10 GB?
6. **Does the query plan look right?** Study it with EXPLAIN.
7. **Is the workload running in parallel?** Use htop or the operating system's task manager to observe this.
8. **Is DuckDB using too many threads?** Try limiting the amount of threads.

QUESTIONS?

<https://duckdb.org/duckdb-docs.pdf>

<https://marclamberti.com/blog>

MotherDuck YouTube channel



<https://github.com/montjoile/PyFR-2024>

Merci !

Do you have any questions?

Sarairis.garcia@gmail.com

[Linkedin.in/sarairisgarcia](https://www.linkedin.com/in/sarairisgarcia)

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.

Please, keep this slide for attribution.