DuckDB

# DuckDB Deep Dive and Lambda Function Simulation for Local Data Processing


## Abstract

This research explores the capabilities of DuckDB, a modern in-process SQL OLAP database engine, by evaluating its performance in handling large-scale datasets stored in CSV and Parquet formats. It also demonstrates how DuckDB can simulate AWS Lambda-style serverless execution locally using Python functions, making it accessible for data analytics without cloud infrastructure or cost. By running SQL queries over real datasets and benchmarking key metrics such as query time, memory usage, and file size, this study provides a deep dive into how DuckDB performs and compares with other competitors like Spark, Pandas, and SQLite.


## 1. Introduction

The modern data landscape demands tools that are fast, flexible, and easy to use for analytical workloads. DuckDB has emerged as a powerful OLAP database engine that requires no server and runs entirely within the process. It is ideal for developers and data scientists who need to query large files locally using SQL.

Simultaneously, serverless computing models like AWS Lambda have changed how developers run lightweight applications without managing infrastructure. In this project, we simulate Lambda behavior locally using DuckDB embedded in a Python function. This hybrid model showcases how modern analytics workflows can be made lightweight, cost-free, and efficient. The use case revolves around New York City Taxi data, a publicly available large dataset ideal for benchmark testing.

## 2. Background & Motivation

DuckDB was released in 2019 by Hannes Mühleisen and Mark Raasveldt at CWI (Centrum Wiskunde & Informatica) in the Netherlands. Often called "SQLite for Analytics," it was built to offer the analytical power of large engines like Spark and Presto, but with the simplicity of an embedded SQL database.

Competitors:

- SQLite: OLTP focused

- Pandas: In-memory but limited for large data

- Spark/Trino: Distributed but complex setup

Why DuckDB is better:

- Embedded, lightweight, easy to install

- Parquet-native and SQL-powered

- Benchmark results prove superior performance for local analytics

## 3. Objectives

• Demonstrate DuckDB for CSV and Parquet querying

• Simulate serverless Lambda-style execution locally

• Benchmark performance: query time, memory, disk size

• Highlight DuckDB's advantages over common alternatives

## 4. Tools & Technologies

• DuckDB (Python API)

• Pandas

• tracemalloc

• Python (local runtime)

• Jupyter Notebook

• NYC Yellow Taxi Trip Data (January 2016)

## 5. Methodology & Demonstrations

The screenshots below include real code used for loading data, converting formats, executing SQL queries with DuckDB, and simulating Lambda behavior.

Figure: Query on CSV File

```
csv_file = '/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.csv'

# SQL query on CSV
df_csv = duckdb.query(f"""
    SELECT passenger_count, COUNT(*) AS total_trips, AVG(total_amount) AS avg_fare
    FROM '{csv_file}'
    GROUP BY passenger_count
    ORDER BY passenger_count
""").to_df()

df_csv
```

Figure: Simulated Lambda Function

```
def fake_lambda_handler(event=None, context=None):
    result = duckdb.query("""
        SELECT passenger_count, COUNT(*) AS total_trips, AVG(total_amount) AS avg_fare
        FROM read_parquet('/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.parquet')
        GROUP BY passenger_count
        ORDER BY passenger_count
    """).to_df()
    return result

# Simulate a Lambda function call
fake_lambda_handler()
```

Figure: Memory Usage Tracking

```
# --- Memory usage for CSV ---
tracemalloc.start()
con.execute("SELECT COUNT(*) FROM '/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.csv'").fetchone()
current_csv, peak_csv = tracemalloc.get_traced_memory()
tracemalloc.stop()

# --- Memory usage for Parquet ---
tracemalloc.start()
con.execute("SELECT COUNT(*) FROM read_parquet('/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.parquet')").fetchone()
current_parquet, peak_parquet = tracemalloc.get_traced_memory()
tracemalloc.stop()
```

Figure: Querying Parquet File

```
parquet_file = '/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.parquet'

df_parquet = duckdb.query(f"""
    SELECT passenger_count, COUNT(*) AS total_trips, AVG(total_amount) AS avg_fare
    FROM read_parquet('{parquet_file}')
    GROUP BY passenger_count
    ORDER BY passenger_count
""").to_df()

df_parquet
```

**Figure: Query Time Measurement**

```
# --- Query on CSV ---
start_csv = time.time()
con.execute("SELECT COUNT(*) FROM '/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.csv'").fetchone()
end_csv = time.time()
csv_time = round(end_csv - start_csv, 2)

# --- Query on Parquet ---
start_parquet = time.time()
con.execute("SELECT COUNT(*) FROM read_parquet('/Users/shreevikasj/Desktop/yellow_tripdata_2016-01.parquet')").fetchone()
end_parquet = time.time()
parquet_time = round(end_parquet - start_parquet, 2)
```

## 6. Evaluation & Results

Performance Summary:
- CSV Query Time: 0.79 sec
- Parquet Query Time: 0.0 sec
- CSV Memory Usage: 1.31 KB
- Parquet Memory Usage: 1.12 KB
- CSV File Size: 1629.52 MB
- Parquet File Size: 228.69 MB
- CSV Full Scan Time: 0.83 sec
- Parquet Full Scan Time: 0.0 sec

## 7. Limitations

DuckDB Limitations:
- Not distributed (no multi-node scaling)
- No user authentication or role management
- No real-time ingestion support

Lambda Simulation Limitations:
- No event triggers like actual AWS Lambda
- Cold start time not simulated
- No real-time streaming or cloud integration

## 8. Conclusion

DuckDB provides excellent performance for local analytics and significantly outperforms traditional CSV processing tools like Pandas. Parquet support, in-process speed, and SQL flexibility make it ideal for lightweight analytics workflows. This project shows that AWS Lambda-like logic can be successfully simulated locally using DuckDB.