DuckDB as a Rudimentary Data Migration Tool

Josef Machytka

3 min read · Just now

After exploring how to use DuckDB as an intelligent ETL tool for PostgreSQL, and how to extend its ETL capabilities with simple Python code, I want to highlight another interesting use case of this versatile database: using DuckDB as a rudimentary data migration tool.

Previously, I demonstrated how to perform cross-database SELECT queries using DuckDB. Building on that, we can easily imagine how straightforward basic data migration becomes with DuckDB. Out of the box, it supports PostgreSQL, MySQL, and SQLite. However, by using simple Python code, we can extend it to other databases as well.

I emphasize the term *"rudimentary"* deliberately. DuckDB is not a full-fledged data migrator or a Change Data Capture (CDC) tool. However, if we just need to take a quick snapshot of a table or migrate a static database without too much hassle, DuckDB can be an excellent assistant.

Example: Migrating a Table from MySQL to PostgreSQL

Let us begin with a simple example: taking a snapshot of a table from MySQL and copying it to PostgreSQL using DuckDB's built-in extensions. Example created by ChatGPT showcases the handling of some MySQL-specific data types.

```
-- Create the table
CREATE TABLE special_data_types (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    status ENUM('active', 'inactive', 'pending') NOT NULL,
    permissions SET('read', 'write', 'execute') NOT NULL,
    small_number TINYINT NOT NULL,
    medium_number MEDIUMINT NOT NULL,
    description TEXT,
    data BLOB,
    created_at DATE NOT NULL
);

-- Insert 10 rows of data
INSERT INTO special_data_types (name, status, permissions, small_number, medium_number, description, data, created_at) VALUES
('Alice', 'active', 'read,write', 5, 1000, 'Alice description', 'Alice data', '2023-01-01'),
('Bob', 'inactive', 'read', 10, 2000, 'Bob description', 'Bob data', '2023-02-01'),
('Charlie', 'pending', 'write,execute', 15, 3000, 'Charlie description', 'Charlie data', '2023-03-01'),
('David', 'active', 'read,write,execute', 20, 4000, 'David description', 'David data', '2023-04-01'),
('Eve', 'inactive', 'execute', 25, 5000, 'Eve description', 'Eve data', '2023-05-01'),
('Frank', 'pending', 'read,write', 30, 6000, 'Frank description', 'Frank data', '2023-06-01'),
('Grace', 'active', 'read', 35, 7000, 'Grace description', 'Grace data', '2023-07-01'),
('Hank', 'inactive', 'write,execute', 40, 8000, 'Hank description', 'Hank data', '2023-08-01'),
('Ivy', 'pending', 'read,write,execute', 45, 9000, 'Ivy description', 'Ivy data', '2023-09-01'),
('Jack', 'active', 'execute', 50, 10000, 'Jack description', 'Jack data', '2023-10-01');
```

After starting both databases in Docker, I used the following approach:

1. Created and populated the MySQL table with test data using script shown above.
2. Launched DuckDB and attached both the MySQL and PostgreSQL databases using DuckDB's extensions.
3. Executed a single command: `CREATE TABLE <PostgreSQL_table> AS SELECT * FROM <MySQL_table>`.

```
D ATTACH 'host=mysql_container port=3306 user=root password=root database=duckdb_test' AS mysql_duckdb_test (TYPE MYSQL);
D SELECT * FROM mysql_duckdb_test.special_data_types;
```

id int32	name varchar	status varchar	permissions varchar	small_number int8	medium_number int32	description varchar	data blob	created_at date
1	Alice	active	read,write	5	1000	Alice description	Alice data	2023-01-01
2	Bob	inactive	read	10	2000	Bob description	Bob data	2023-02-01
3	Charlie	pending	write,execute	15	3000	Charlie description	Charlie data	2023-03-01
4	David	active	read,write,execute	20	4000	David description	David data	2023-04-01
5	Eve	inactive	execute	25	5000	Eve description	Eve data	2023-05-01
6	Frank	pending	read,write	30	6000	Frank description	Frank data	2023-06-01
7	Grace	active	read	35	7000	Grace description	Grace data	2023-07-01
8	Hank	inactive	write,execute	40	8000	Hank description	Hank data	2023-08-01
9	Ivy	pending	read,write,execute	45	9000	Ivy description	Ivy data	2023-09-01
10	Jack	active	execute	50	10000	Jack description	Jack data	2023-10-01

10 rows 9 columns

```
D ATTACH 'host=postgres_container port=5432 user=postgres password=postgres dbname=duckdb_test' as pg_duckdb_test (TYPE POSTGRES, SCHEMA 'public');
D CREATE TABLE pg_duckdb_test.special_data_types as SELECT * FROM mysql_duckdb_test.special_data_types;
```

Quick verification in PostgreSQL confirms the table was created successfully and data copied. Big advantage of this approach is that we do not need to know anything about the structure of the migrated tables — DuckDB handles the schema automatically during the migration process. While we might debate the efficiency or precision of DuckDB's type casting during the process, remember that this approach is designed for simplicity and speed of ad-hoc tasks, not for handling complex migrations.

```
duckdb_test=# \dS+ special_data_types
                                    Table "public.special_data_types"
    Column     |       Type        | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
---------------+-------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 id            | integer           |           |          |         | plain    |             |              |
 name          | character varying |           |          |         | extended |             |              |
 status        | character varying |           |          |         | extended |             |              |
 permissions   | character varying |           |          |         | extended |             |              |
 small_number  | smallint          |           |          |         | plain    |             |              |
 medium_number | integer           |           |          |         | plain    |             |              |
 description   | character varying |           |          |         | extended |             |              |
 data          | bytea             |           |          |         | extended |             |              |
 created_at    | date              |           |          |         | plain    |             |              |
Access method: heap

duckdb_test=# select * from special_data_types ;
 id |  name   |  status  |    permissions     | small_number | medium_number |     description     |                data                | created_at
----+---------+----------+--------------------+--------------+---------------+---------------------+------------------------------------+------------
  1 | Alice   | active   | read,write         |            5 |          1000 | Alice description   | \x416c6963652064617461             | 2023-01-01
  2 | Bob     | inactive | read               |           10 |          2000 | Bob description     | \x426f622064617461                 | 2023-02-01
  3 | Charlie | pending  | write,execute      |           15 |          3000 | Charlie description | \x436861726c696520646174617461     | 2023-03-01
  4 | David   | active   | read,write,execute |           20 |          4000 | David description   | \x4461766964206461746174617461     | 2023-04-01
  5 | Eve     | inactive | execute            |           25 |          5000 | Eve description     | \x4576652064617461                 | 2023-05-01
  6 | Frank   | pending  | read,write         |           30 |          6000 | Frank description   | \x4672616e6b2064617461             | 2023-06-01
  7 | Grace   | active   | read               |           35 |          7000 | Grace description   | \x4772616365206461746174617461     | 2023-07-01
  8 | Hank    | inactive | write,execute      |           40 |          8000 | Hank description    | \x48616e6b2064617461               | 2023-08-01
  9 | Ivy     | pending  | read,write,execute |           45 |          9000 | Ivy description     | \x4976792064617461                 | 2023-09-01
 10 | Jack    | active   | execute            |           50 |         10000 | Jack description    | \x4a61636b2064617461               | 2023-10-01
(10 rows)
```

Example: Extending Functionality for MS SQL Server

What if the source database is not directly supported by DuckDB, such as MS SQL Server? While DuckDB currently does not have an extension for MS SQL Server, this limitation can be surpassed using a simple Python code with `pyodbc` connector, and `polars` DataFrames.

```
import pyodbc
import polars as pl
import duckdb

conn_str = "DRIVER={ODBC Driver 17 for SQL Server};SERVER=localhost,1433;DATABAS
connection = pyodbc.connect(conn_str)

query = "SELECT * FROM dbo.small_mssql_table"
df = pl.read_database(query, connection)
connection.close()

duckdb_conn = duckdb.connect(database=':memory:')
duckdb_conn.execute("""
   ATTACH 'dbname=duckdb_test user=postgres password=postgres host=localhost por
   AS pg (TYPE POSTGRES, SCHEMA 'public')""")

duckdb_conn.execute("CREATE TABLE pg.small_mssql_table AS SELECT * FROM df")
duckdb_conn.close()
```

If the table is small enough to fit in memory, the entire process can be performed in a single step. For larger tables, we can process the data in batches to manage memory usage effectively.

```
import pyodbc
import polars as pl
import duckdb

source_table_name = 'dbo.big_mssql_table'
target_table_name = 'pg.big_mssql_table'

batch_size = 1000
```

```
conn_str = "DRIVER={ODBC Driver 17 for SQL Server};SERVER=localhost,1433;DATABAS
connection = pyodbc.connect(conn_str)

mssql_cursor = connection.cursor()
query = f"SELECT * FROM {source_table_name}"
mssql_cursor.execute(query)
rows = mssql_cursor.fetchmany(batch_size)

schema = {column[0]: pl.datatypes.DataType.from_python(column[1]) for column in
df = pl.DataFrame([dict(zip(schema.keys(), row)) for row in rows], schema=schema

duckdb_conn = duckdb.connect(database=':memory:')
duckdb_conn.execute("""
   ATTACH 'dbname=duckdb_test user=postgres password=postgres host=localhost por
   AS pg (TYPE POSTGRES, SCHEMA 'public')""")
duckdb_conn.execute(f"CREATE TABLE {target_table_name} AS SELECT * FROM df")

while rows:
   rows = mssql_cursor.fetchmany(batch_size)
   if rows:
       df = pl.DataFrame([dict(zip(schema.keys(), row)) for row in rows], schema
       duckdb_conn.execute(f"INSERT INTO {target_table_name} SELECT * FROM df")

mssql_cursor.close()
connection.close()
duckdb_conn.close()
```

Summary

DuckDB is a remarkably handy tool that simplifies many ad-hoc tasks, including rudimentary data migration. It is not designed to replace dedicated migration or CDC tools. However its flexibility and extensibility makes it an excellent choice for quick snapshots, static database migrations, and prototyping. By leveraging DuckDB's capabilities with Python, we can efficiently bridge the gap between databases, even those not natively supported.

Duckdb | Data Migration | Python | Polars Dataframe

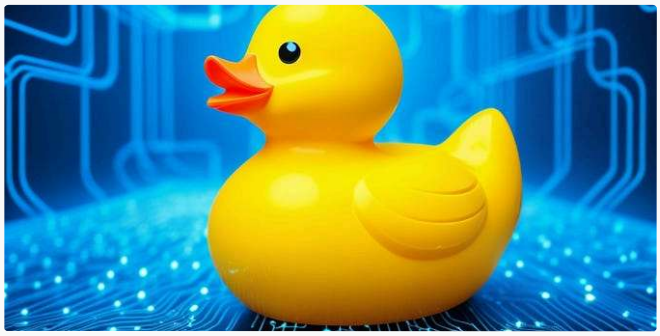
Written by Josef Machytka

12 Followers · 2 Following

Edit profile

I work as Professional Service Consultant - PostgreSQL specialist in NetApp Deutschland GmbH, Open Source Services division.

More from Josef Machytka



Josef Machytka

How DuckDB handles data not fitting into memory?

In my previous article about DuckDB I described how to use this database as an...

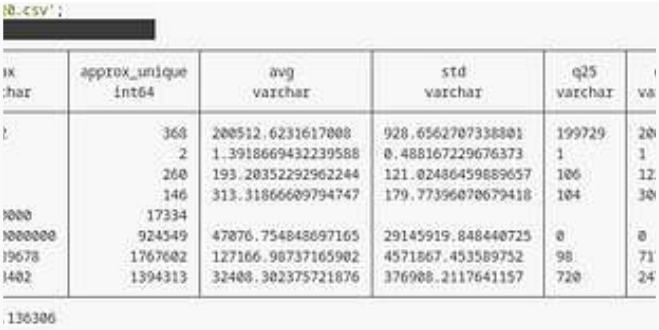
Nov 13 1

Josef Machytka

Extending DuckDB ETL Capabilities with Python

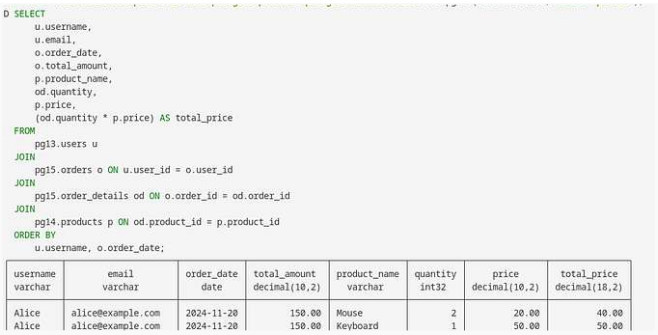
DuckDB has recently become my go-to solution for small ETL tasks. It is an...

6d ago 6



ix :har	approx_unique int64	avg varchar	std varchar	q25 varchar	va
	368	200512.6231617008	928.6562707338801	199729	20
	2	1.3918669432239588	0.488167229676373	1	1
	260	193.20352292962244	121.02486459889657	106	12
	146	313.31866609794747	179.77396070679418	104	30
)000	17334				
)000000	924549	47076.754848697165	29145919.848440725	0	0
19678	1767602	127166.98737165902	4571867.453589752	98	71
1402	1394313	32408.302375721876	376908.2117641157	720	24

136306



```
D SELECT
      u.username,
      u.email,
      o.order_date,
      o.total_amount,
      p.product_name,
      od.quantity,
      p.price,
      (od.quantity * p.price) AS total_price
  FROM
      pg13.users u
  JOIN
      pg15.orders o ON u.user_id = o.user_id
  JOIN
      pg15.order_details od ON o.order_id = od.order_id
  JOIN
      pg14.products p ON od.product_id = p.product_id
  ORDER BY
      u.username, o.order_date;
```

username varchar	email varchar	order_date date	total_amount decimal(10,2)	product_name varchar	quantity int32	price decimal(10,2)	total_price decimal(18,2)
Alice	alice@example.com	2024-11-20	150.00	Mouse	2	20.00	40.00
Alice	alice@example.com	2024-11-20	150.00	Keyboard	1	50.00	50.00

Josef Machytka

Using DuckDB as an Intelligent ETL tool for PostgreSQL

There is a lot of hype around DuckDB these days. At one PostgreSQL conference, I even...

Nov 2

Josef Machytka

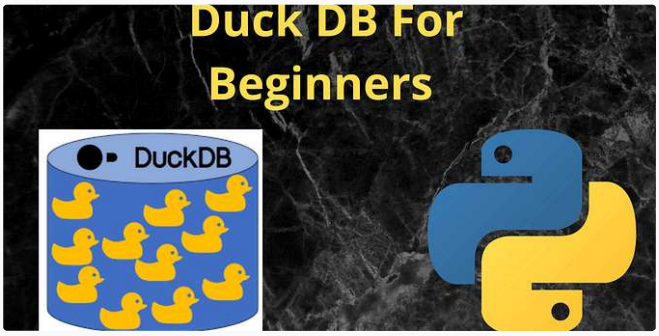
Easy Cross-Database Selects with DuckDB

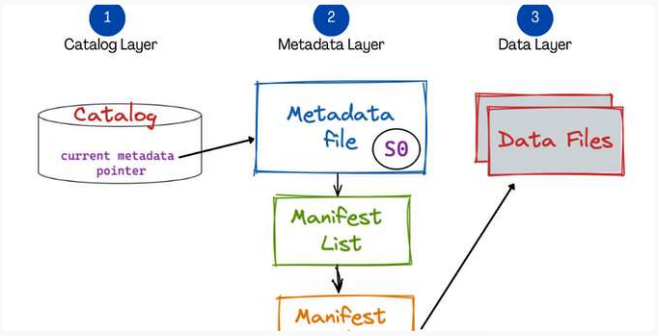
DuckDB was created with simplicity and ease of use in mind. In my previous article, I...

4d ago

See all from Josef Machytka

Recommended from Medium





Kevin Meneses González

Maximize Your Data with DuckDB: A Simple Guide to OLAP vs OLTP...

Introduction

Jul 22 10

Shraddha Shetty

Understanding the Table Structure of Apache Iceberg

Apache Iceberg is a high-performance, open table format designed for large-scale...

Aug 21 2

Lists

Coding & Development

11 stories · 922 saves

Predictive Modeling w/ Python

20 stories · 1691 saves

Practical Guides to Machine Learning

10 stories · 2059 saves

ChatGPT

21 stories · 894 saves

In Python in Plain English by Alexzap

Creating 50+ Stunning Charts in Python: The Basics with a Twist

A Deep Dive into Data Visualization for Beginners+ with Easy-to-Follow Examples &...

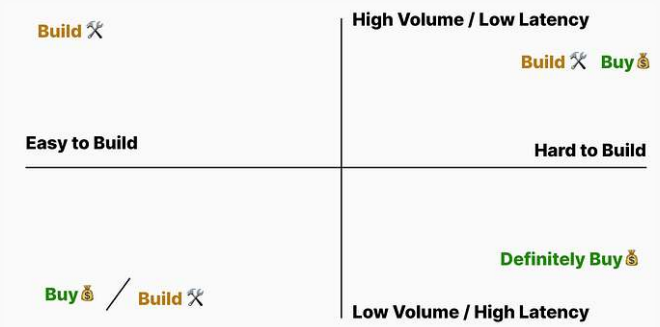
Nov 20 325 1

In Towards Data Science by Gustavo R Santos

Documenting Python Projects with MkDocs

Use Markdown to quickly create a beautiful documentation page for your projects

Nov 22 470 4



Hugo Lu

Python ELT with dlthub— why I both love and hate data load tool...

Writing your own connectors has always been satisfying and excruciating in equal measure...

4d ago 29 1

Dhiraj Mishra

Streaming data using Kafka, PostgreSQL, Spark Streaming,...

Introduction

Oct 4 1

See more recommendations