Using DuckDB as an Intelligent ETL tool for PostgreSQL

Josef Machytka

4 min read · 1 day ago

There is a lot of hype around DuckDB these days. At one PostgreSQL conference, I even saw a large poster comparing DuckDB with PostgreSQL, presenting DuckDB as a superior tool. Having worked with various databases over many years, I was really curious about this new tool and decided to test it deeper. My experiments confirmed that DuckDB is indeed a valuable tool, filling gaps in areas where other databases fall short or show limited focus.

I love DuckDB. It is already a powerful tool and has the potential to be a game-changer for specific use cases within the next year. However, by my opinion some articles rather exaggerate its current capabilities. It only reached production maturity in June of this year and is currently at version 1.1.2, which has addressed many bugs. Some compelling extensions, which I would love to use and recommend to clients, remain marked as "experimental." But hopefully in the next six to twelve months, DuckDB will truly shine. That said, it already supports significant use cases today. So let us dive into one.

I have always worked with big data, including importing large external data files into PostgreSQL and MySQL. A few years ago, ad-hoc import of a CSV file several gigabytes in size into these databases was almost a celebrated achievement. Even now, it requires manual checks and preparations, especially in PostgreSQL. Existing external tools are not always user friendly and most of them are really slow. This is why I am thrilled that DuckDB can greatly streamline this process..

What sets DuckDB apart when working with external data files is its simplicity. There is no need to create database objects manually or even check the file's structure beforehand. DuckDB implements external data sources via table functions API, enabling us to select data from external files directly within the `FROM` clause of a query.

But what is the most amazing part about DuckDB when it comes to external data files, is the absolute simplicity of usage. No need to create any object manually, no need to even check structure of the file. DuckDB implements all external data sources through table functions API. Which means we can directly select data from the external file in the FROM clause of the query.

For my tests, I used a 100-million-row dataset from Kaggle. This 4.5 GB CSV file contains 113 million rows of Japanese trade statistics dating back to 1988. The structure is loosely described as eight INT columns, and the file lacks a header. In the past, importing this type of data would typically involve several minutes of manual setup and problem-solving.

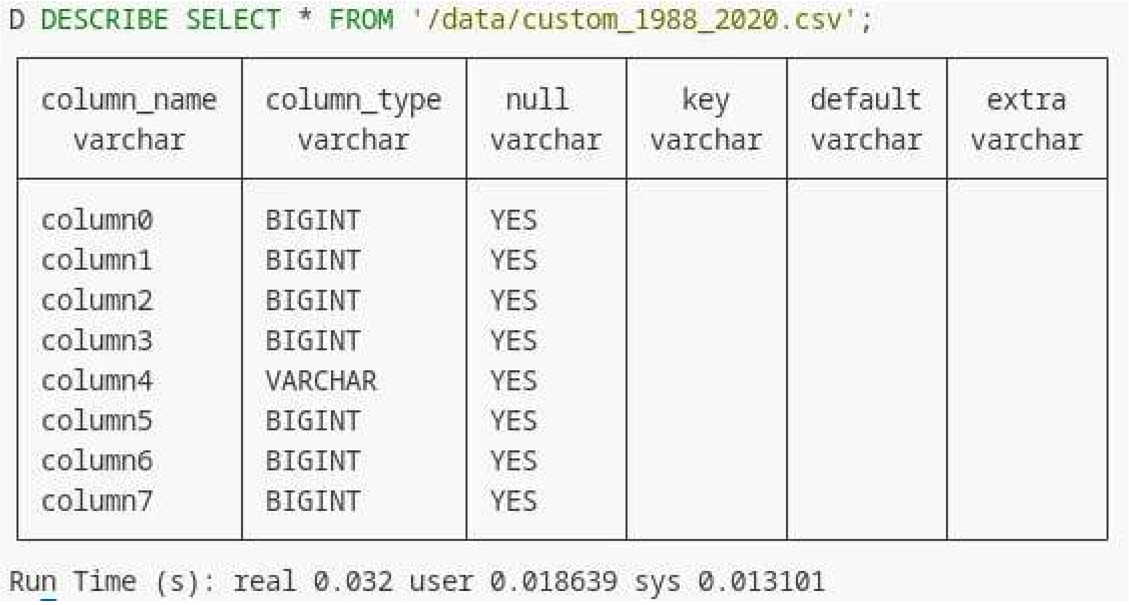
DuckDB, however, handled this file effortlessly. The data selection was as simple as:

```
SELECT * FROM '/data/custom_1988_2020.csv';
```

To inspect the file structure, I used:

```
DESCRIBE SELECT * FROM '/data/custom_1988_2020.csv';
```

To my surprise, DuckDB immediately identified an issue with the dataset: numeric values in column 5 were formatted as strings, causing conversion errors when attempting to import the data as eight INT columns, as described on Kaggle. DuckDB flagged this issue in just 0.03 seconds.

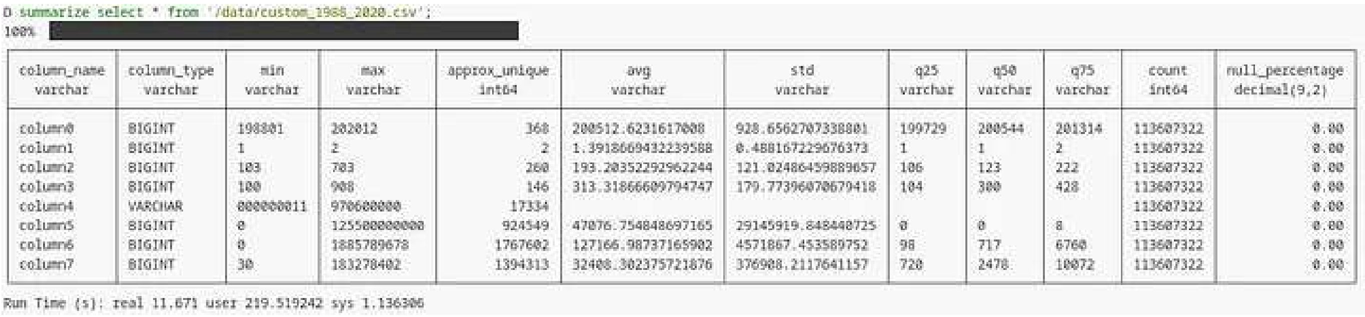


```
D DESCRIBE SELECT * FROM '/data/custom_1988_2020.csv';
┌─────────────┬─────────────┬─────────┬─────────┬─────────┬─────────┐
│ column_name │ column_type │  null   │   key   │ default │  extra  │
│   varchar   │   varchar   │ varchar │ varchar │ varchar │ varchar │
├─────────────┼─────────────┼─────────┼─────────┼─────────┼─────────┤
│ column0     │ BIGINT      │ YES     │         │         │         │
│ column1     │ BIGINT      │ YES     │         │         │         │
│ column2     │ BIGINT      │ YES     │         │         │         │
│ column3     │ BIGINT      │ YES     │         │         │         │
│ column4     │ VARCHAR     │ YES     │         │         │         │
│ column5     │ BIGINT      │ YES     │         │         │         │
│ column6     │ BIGINT      │ YES     │         │         │         │
│ column7     │ BIGINT      │ YES     │         │         │         │
└─────────────┴─────────────┴─────────┴─────────┴─────────┴─────────┘
Run Time (s): real 0.032 user 0.018639 sys 0.013101
```

Another amazing feature is DuckDB's ability to create basic column statistics with minimal effort. Using:

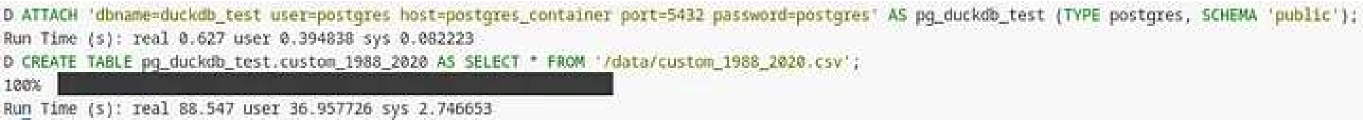
```
SUMMARIZE SELECT * FROM '/data/custom_1988_2020.csv';
```

I received comprehensive statistics within approximately 12 seconds. Doing this in other databases would require writing custom analytical queries first, which makes DuckDB a clear winner for quick data profiling.



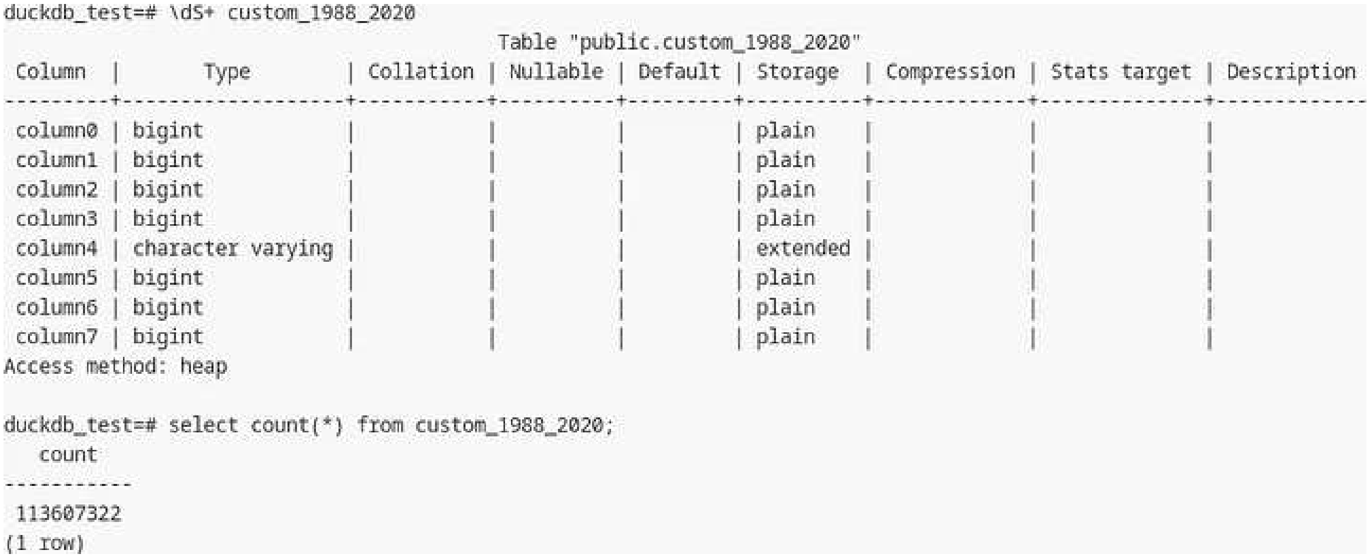
```
D summarize select * from '/data/custom_1988_2020.csv';
100% ▕████████████████████████████████████████████████████████████▏
┌─────────────┬─────────────┬───────────┬──────────────┬───────────────┬────────────────────┬────────────────────┬─────────┬─────────┬─────────┬───────────┬─────────────────┐
│ column_name │ column_type │    min    │     max      │ approx_unique │        avg         │        std         │   q25   │   q50   │   q75   │   count   │ null_percentage │
│   varchar   │   varchar   │  varchar  │   varchar    │     int64     │      varchar       │      varchar       │ varchar │ varchar │ varchar │   int64   │  decimal(9,2)   │
├─────────────┼─────────────┼───────────┼──────────────┼───────────────┼────────────────────┼────────────────────┼─────────┼─────────┼─────────┼───────────┼─────────────────┤
│ column0     │ BIGINT      │ 198801    │ 202012       │           368 │ 200512.6231617008  │ 928.6562707338801  │ 199729  │ 200544  │ 201314  │ 113607322 │            0.00 │
│ column1     │ BIGINT      │ 1         │ 2            │             2 │ 1.3918669432239588 │ 0.488167229676373  │ 1       │ 1       │ 2       │ 113607322 │            0.00 │
│ column2     │ BIGINT      │ 103       │ 703          │           260 │ 193.20352292962244 │ 121.02486459889657 │ 106     │ 123     │ 222     │ 113607322 │            0.00 │
│ column3     │ BIGINT      │ 100       │ 908          │           146 │ 313.31866609794747 │ 179.77396070679418 │ 104     │ 300     │ 428     │ 113607322 │            0.00 │
│ column4     │ VARCHAR     │ 000000011 │ 970600000    │         17334 │                    │                    │         │         │         │ 113607322 │            0.00 │
│ column5     │ BIGINT      │ 0         │ 125500000000 │        924549 │ 47076.754848697165 │ 29145919.848440725 │ 0       │ 0       │ 8       │ 113607322 │            0.00 │
│ column6     │ BIGINT      │ 0         │ 1885789678   │       1767602 │ 127166.98737165902 │ 4571867.453589752  │ 98      │ 717     │ 6760    │ 113607322 │            0.00 │
│ column7     │ BIGINT      │ 30        │ 183278402    │       1394313 │ 32408.302375721876 │ 376908.2117641157  │ 720     │ 2478    │ 10072   │ 113607322 │            0.00 │
└─────────────┴─────────────┴───────────┴──────────────┴───────────────┴────────────────────┴────────────────────┴─────────┴─────────┴─────────┴───────────┴─────────────────┘
Run Time (s): real 11.671 user 219.519242 sys 1.136306
```

Transferring data from DuckDB to PostgreSQL is straightforward and involves just two steps. First, you attach the remote PostgreSQL database and specify a remote schema name. Then, you import the data using "CREATE TABLE in PostgreSQL AS SELECT * FROM data file":



```
D ATTACH 'dbname=duckdb_test user=postgres host=postgres_container port=5432 password=postgres' AS pg_duckdb_test (TYPE postgres, SCHEMA 'public');
Run Time (s): real 0.627 user 0.394838 sys 0.082223
D CREATE TABLE pg_duckdb_test.custom_1988_2020 AS SELECT * FROM '/data/custom_1988_2020.csv';
100% ▕████████████████████████████████████████████████████████████▏
Run Time (s): real 88.547 user 36.957726 sys 2.746653
```

Credentials can be provided directly within the `ATTACH` command or managed through DuckDB's `CREATE SECRET` function. And in PostgreSQL we can see our new table:



```
duckdb_test=# \dS+ custom_1988_2020
                                               Table "public.custom_1988_2020"
 Column  |       Type        | Collation | Nullable | Default | Storage  | Compression | Stats target | Description
---------+-------------------+-----------+----------+---------+----------+-------------+--------------+-------------
 column0 | bigint            |           |          |         | plain    |             |              |
 column1 | bigint            |           |          |         | plain    |             |              |
 column2 | bigint            |           |          |         | plain    |             |              |
 column3 | bigint            |           |          |         | plain    |             |              |
 column4 | character varying |           |          |         | extended |             |              |
 column5 | bigint            |           |          |         | plain    |             |              |
 column6 | bigint            |           |          |         | plain    |             |              |
 column7 | bigint            |           |          |         | plain    |             |              |
Access method: heap

duckdb_test=# select count(*) from custom_1988_2020;
   count
-----------
 113607322
(1 row)
```

And that’s it! That’s the entire process for importing a 4.5 GB CSV file into PostgreSQL. The same method works for importing data into MySQL. This also works seamlessly with JSON and Parquet files, and hopefully, extensions for other data formats will be available soon.

After my tests, I really love DuckDB and I am convinced that it is an essential addition to the toolkit of any data scientist, data analyst, or database administrator. However, it is not a competitor to “big” databases; instead, it complements them by addressing weaknesses in areas that are often overlooked or underestimated.

Postgresql Duckdb Csv Import Data Analysis

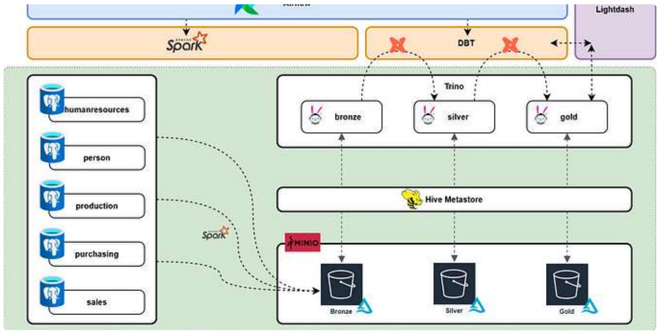
Written by Josef Machytka

Edit profile

0 Followers

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

Recommended from Medium



Thanh Enc in Data Engineer Things

PoC Data Platform project utilizing modern data stack (Airflow, Spar…

PoC Data Platform is an innovative project designed to demonstrate how data from…

Oct 8 150 1

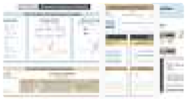
Nikita Shpilevoy

Advanced SQL Techniques and Complex Queries in PostgreSQL

PostgreSQL stands out as a open source RDBMS known for its adherence, to SQL…

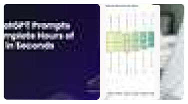
Oct 1 5

Lists



Practical Guides to Machine Learning

10 stories · 1995 saves



ChatGPT prompts

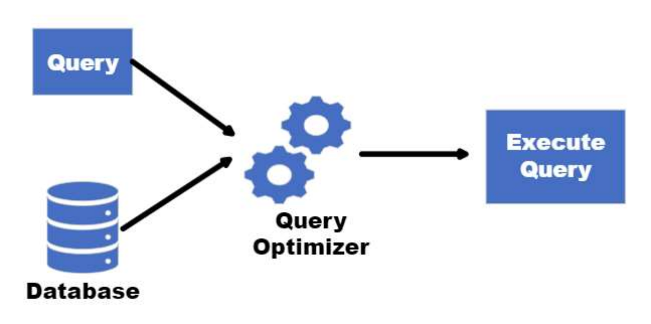
50 stories · 2169 saves

Staff Picks

755 stories · 1417 saves

Natural Language Processing

1789 stories · 1394 saves



Anita Shaw

Optimizing PostgreSQL Queries

Optimizing PostgreSQL Queries: Techniques and Examples

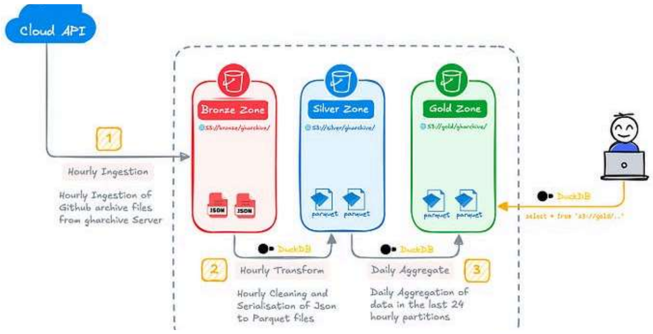
Sep 27 1

Abdur Rahman in Stackademic

Python is No More The King of Data Science

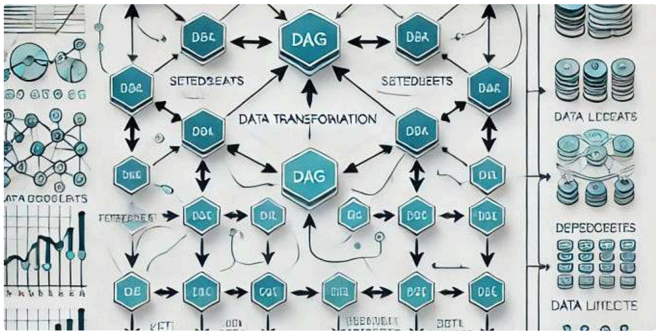
5 Reasons Why Python is Losing Its Crown

Oct 23 2K 16



Alireza Sadeghi

Building a High-Performance Data Pipeline Using DuckDB



Stefan Krawczyk

Open Source Python Data Lineage with OpenLineage and Hamilton

Using DuckDB to Serialise, Transform, and Aggregate Data in Data Lakes

Oct 20 250 2 •••

Data lineage for SQL is basically a solved problem. For python based workflows, that i…

Sep 25 34 1 •••

See more recommendations