How DuckDB handles data not fitting into memory?

Josef Machytka
4 min read · 2 days ago

In my previous article about DuckDB I described how to use this database as an intelligent ETL tool for PostgreSQL or MySQL. During my testing, I worked with a relatively large (4.5 GB) CSV file from Kaggle, which fit comfortably into the memory of my testing machine. DuckDB's documentation directly mentions that its primary use case is for datasets that fit into memory, noting that while it can handle data spilling to disk, they don't recommend it as a typical use case.

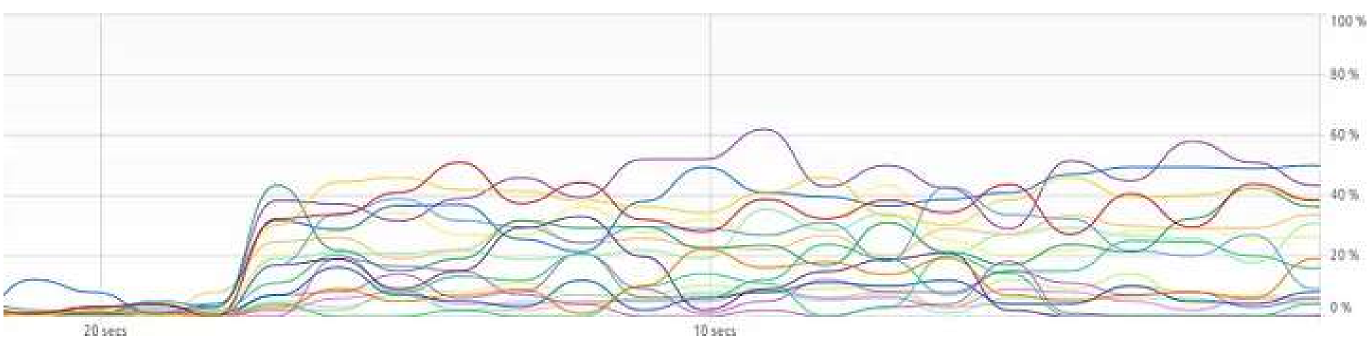
But I was curious about how DuckDB handles data that doesn't fit into memory, so I decided to put it to the test. To simulate this scenario, I used ChatGPT to give me Python code which will generate a large CSV file. I frequently use AI to automate tedious, time-consuming tasks, and this approach saved me considerable time in preparing a realistic testing examples.

ChatGPT gave me a well-structured, non-trivial table with 11 columns of various data types and Python code to generate the data. My test machine has 20 cores and 64 GB of RAM running Ubuntu 24. After generating the CSV file to a size of 100 GB, I found it contained 399,682,000 rows. While this is an artificial example — it's unlikely DuckDB would typically need to import such a massive file — this test allowed me to explore DuckDB's handling of large data and understand why it's so quick in processing operations.

Task 1: Counting Rows

The first task was straightforward: I asked DuckDB to count the rows in CSV file.

As soon as the operation began, I could see in System Monitor that DuckDB was using multiple cores.



The task completed in approximately 42 seconds.

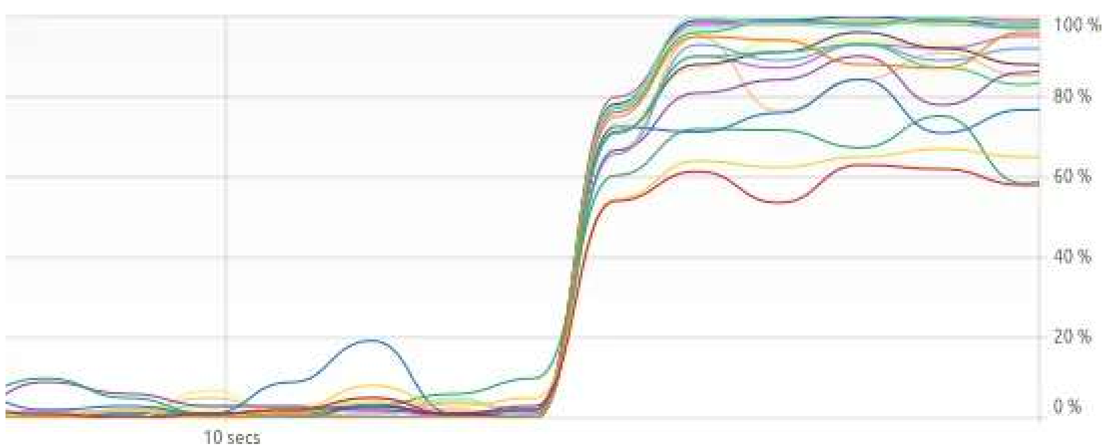
```
D select count(*) from './data/data_100.csv';
100% ████████████████████████████████████████████████████████████
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│    399682000 │
└──────────────┘
Run Time (s): real 41.447 user 106.476844 sys 35.288722
```

Task 2: Summarizing Basic Statistics

For the second task, I asked DuckDB to summarize basic statistics from the file using the SUMMARIZE command.

```
D summarize select * from './data/data_100.csv';
  8% █████                                                      ▏
```

DuckDB again utilized multiple cores, this time even more intensively. Eight cores were running at over 98%.



This task took around 79 seconds, clearly documenting DuckDB's efficient parallel processing capabilities.

column_name varchar	column_type varchar	min varchar	max varchar	approx_unique int64	…	q50 varchar	q75 varchar	count int64	null_percentage decimal(9,2)
user_id	BIGINT	1	399682000	418759233	…	199903003	299807768	399682000	0.00
first_name	VARCHAR	AAA	zzzzyQjd	322345046	…			399682000	0.00
last_name	VARCHAR	AAA	zzzzuRCU	267610286	…			399682000	0.00
email	VARCHAR	aaa.aao222645931@e…	zzzzzzmg.rplh31199…	389505211	…			399682000	0.00
signup_date	TIMESTAMP	2022-02-11 06:58:44	2024-11-07 09:00:49	8201541	…	2023-06-26 09:41:1…	2024-03-02 15:31:3…	399682000	0.00
last_login	TIMESTAMP	2022-02-11 06:58:49	2024-11-07 09:00:49	8201541	…	2024-05-04 18:53:1…	2024-08-31 19:29:5…	399682000	0.00
is_active	BOOLEAN	false	true	2	…			399682000	0.00
account_balance	DOUBLE	0.0	10000.0	1012389	…	4997.571189812575	7499.3417345063135	399682000	0.00
country_code	VARCHAR	AU	US	11	…			399682000	0.00
favorite_number	BIGINT	1	100	96	…	50	75	399682000	0.00
profile_text	VARCHAR	41rCAFEMHuw6bB…	zzzzyCjVhwNEr8Qtvc…	424701465	…			399682000	0.00
checksum	VARCHAR	00000005-dac2-4875…	fffffff8-4a11-4d3d…	392541443	…			399682000	0.00
12 rows									12 columns (9 shown)

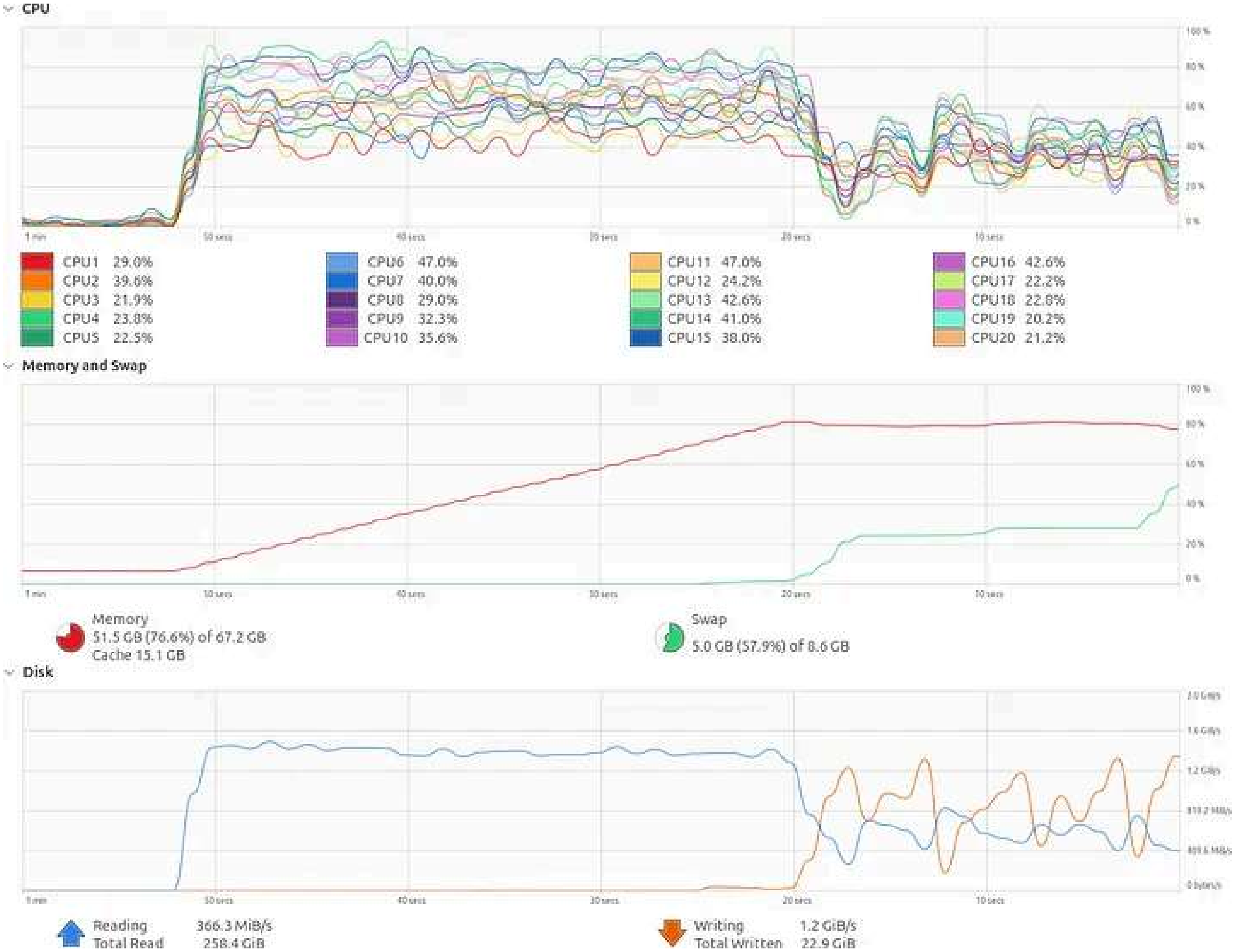
```
Run Time (s): real 78.423 user 1279.974541 sys 57.487294
```

Task 3: Importing Data into an In-Memory Table

For the third test, I asked DuckDB to import the 100 GB CSV file into an in-memory table. According to the documentation, DuckDB can work with data that doesn't fit into memory, so I wanted to see how it would handle this massive import.

```
D create table data_100gb_imported as select * from './data/data_100.csv';
 54%
```

This time, I observed DuckDB's approach to handling such a large dataset. Once again, all cores were in use, though less intensively than before. Memory usage grew significantly, and once it reached around 80% of available RAM, the machine began to use swap space.



DuckDB also started creating temporary storage files in the .tmp directory, which at the end reached 63 GB total size.

```
total 65999904
drwxr-xr-x 2 josef josef        4096 Nov 13 06:55 .
drwxrwxr-x 9 josef josef        4096 Nov 13 06:46 ..
-rw-rw-r-- 1 josef josef  1048576000 Nov 13 06:53 duckdb_temp_storage-0.tmp
-rw-rw-r-- 1 josef josef  2138570752 Nov 13 06:53 duckdb_temp_storage-1.tmp
-rw-rw-r-- 1 josef josef  4196925440 Nov 13 06:53 duckdb_temp_storage-2.tmp
-rw-rw-r-- 1 josef josef  8388870144 Nov 13 06:53 duckdb_temp_storage-3.tmp
-rw-rw-r-- 1 josef josef 16779837440 Nov 13 06:54 duckdb_temp_storage-4.tmp
-rw-rw-r-- 1 josef josef 33578024960 Nov 13 06:55 duckdb_temp_storage-5.tmp
-rw-rw-r-- 1 josef josef  1453064192 Nov 13 06:55 duckdb_temp_storage-6.tmp
```

The import operation took almost 190 seconds (3 minutes and 10 seconds). While CPU usage was this time much lower, DuckDB still used masive parallel processing.

```
D create table data_100gb_imported as select * from './data/data_100.csv';
100% ████████████████████████████████████████████████████████████
Run Time (s): real 189.233 user 696.525375 sys 322.379269
```

By the end of the task, the duckdb_memory() metadata function showed high memory and disk usage. The numbers didn't exactly match the data file size because DuckDB uses compression in storage.

```
D select * from duckdb_memory();
┌──────────────────┬────────────────────┬─────────────────────────┐
│       tag        │ memory_usage_bytes │ temporary_storage_bytes │
│     varchar      │       int64        │          int64          │
├──────────────────┼────────────────────┼─────────────────────────┤
│ BASE_TABLE       │         1022869504 │                       0 │
│ HASH_TABLE       │                  0 │                       0 │
│ PARQUET_READER   │                  0 │                       0 │
│ CSV_READER       │                  0 │                       0 │
│ ORDER_BY         │                  0 │                       0 │
│ ART_INDEX        │                  0 │                       0 │
│ COLUMN_DATA      │                  0 │                       0 │
│ METADATA         │                  0 │                       0 │
│ OVERFLOW_STRINGS │                  0 │                       0 │
│ IN_MEMORY_TABLE  │        52407304192 │            101904583592 │
│ ALLOCATOR        │                  0 │                       0 │
│ EXTENSION        │                  0 │                       0 │
├──────────────────┴────────────────────┴─────────────────────────┤
│ 12 rows                                               3 columns │
└─────────────────────────────────────────────────────────────────┘
Run Time (s): real 0.101 user 0.002855 sys 0.000370
```

Summary

In these tests DuckDB demonstrated its capability to gracefully manage even data that exceeds available memory. However, it also showed that, in such cases, it requires large temporary storage files on disk. This means DuckDB can indeed be used for large-scale ETL tasks, as long as sufficient disk space is available to support the operations.

After this experiment, my appreciation for DuckDB has grown. It not only handled the 100 GB CSV file but did it efficiently, using parallelism to minimize processing time. DuckDB proves to be a powerful option even for datasets that don't fit into memory. After this test I love DuckDB even more.

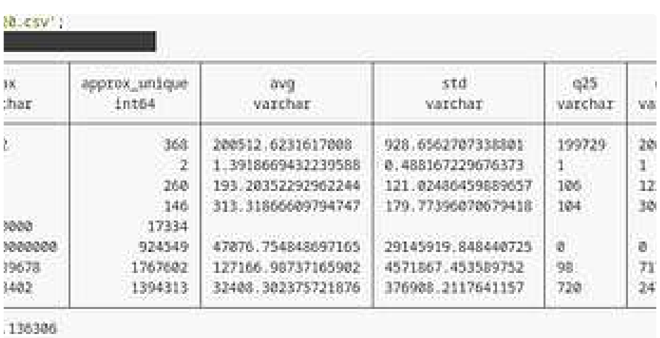
Written by Josef Machytka

Edit profile

4 Followers

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

More from Josef Machytka



Josef Machytka

Josef Machytka

Using DuckDB as an Intelligent ETL tool for PostgreSQL

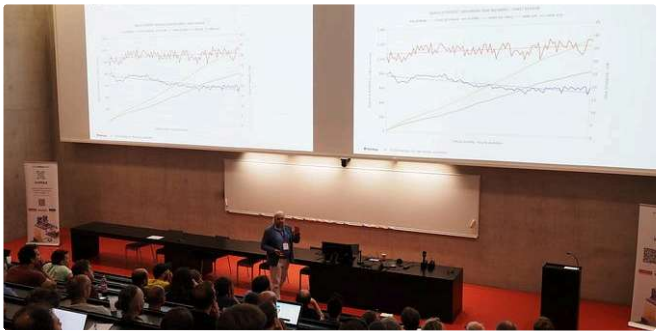
There is a lot of hype around DuckDB these days. At one PostgreSQL conference, I even...

Nov 2

AI Hallucinations are caused by Quantum Pigeons Nesting in...

This is not a new discovery in quantum physics — it is a playful deliberate...

Nov 5



Josef Machytka

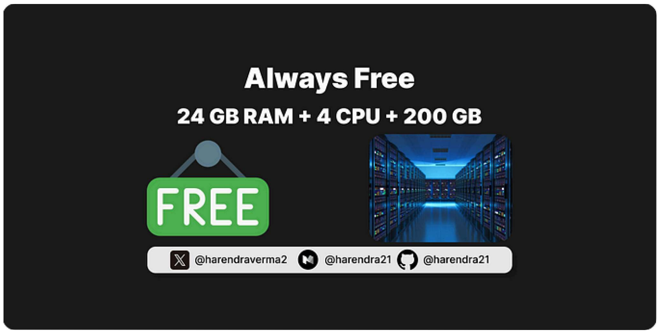
Josef Machytka: Speaker Portfolio

Expert Talks on PostgreSQL, Databases, Data Ingestion and Data Analysis

Nov 3

See all from Josef Machytka

Recommended from Medium

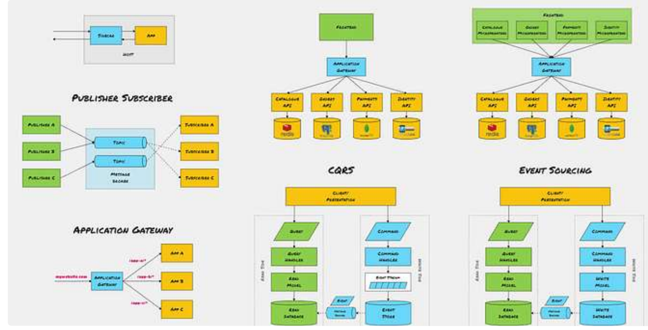


Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26 4.5K 55



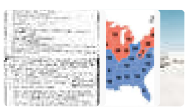
Matt Bentley in Level Up Coding

My Favourite Software Architecture Patterns

Exploring my most loved Software Architecture patterns and their practical...

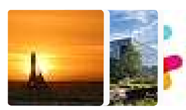
2d ago 1.5K 26

Lists



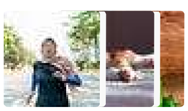
Staff Picks

765 stories · 1437 saves



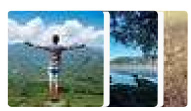
Stories to Help You Level-Up at Work

19 stories · 864 saves



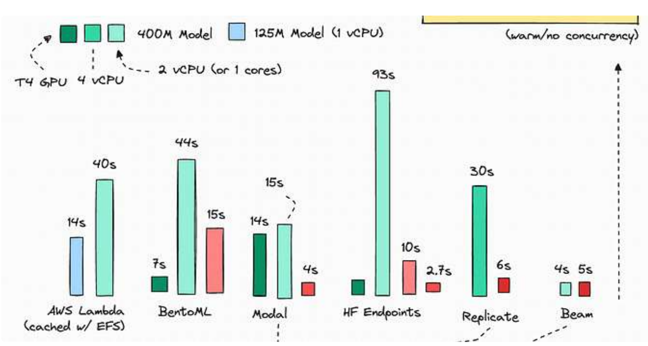
Self-Improvement 101

20 stories · 3013 saves

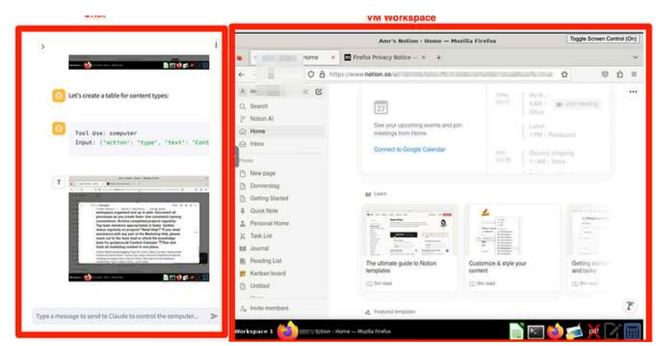


Productivity 101

20 stories · 2558 saves



Ida Silfverskiöld in Towards Data Science



AI Rabbit in CodeX

Economics of Hosting Open Source LLMs

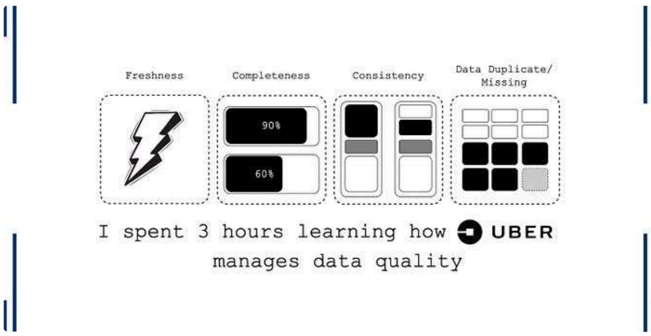
Leveraging various deployment options

2d ago 622 8

Has Anthropic Claude just wiped out an entire industry?

If you have been following the news, you may have read about a new feature (or should I ca...

Oct 27 1.91K 31



Vu Trinh in Data Engineer Things

I spent 3 hours learning how Uber manages data quality.

From the standard to the data quality platform

5d ago 136 1

Marek Sirkovský

New Lock object and history

Discover what's new for locking in .NET 9—even the ancient lock keyword can get an...

Nov 4 116

See more recommendations