28 May 2024

DuckDB Doesn't Need Data To Be a Database

One of the many enjoyable things about databases is that they generally try to separate how data is represented internally (say on disk) from how it is used. To the point that it has become the norm to not even store the data on the same hardware that is running the queries.

Data*bases* have gotten so good at this, that the term is almost misleading now. "Base" suggests something rigid, without which the data would slip away. But the data is always there, just bits on a nameless hard disk. The structure and the accessibility that a modern database provides exist completely independently from that hard disk. That's right – most databases no longer have any data in them.

DuckDB is a database built for this age, and a particularly lovely one at that.

Say you run a robotaxi service. And you're maintaining a growing dataset of interesting ride patterns in blob storage, that you'd like to understand better. The data is split into a separate Parquet file for each day. How do you share that dataset with an analyst?

You could just mail them everything, but the dataset is getting way too big for that. Plus it'd be out of date instantly. If this was a blogpost you would just send a link, but there isn't really a single thing to link to here. And sending someone a hundred links to raw blobs in S3 can sour any working relationship.

Instead you whip up a little database, just for them:

```
# Send
import duckdb
db = duckdb.connect("weird_rides.db")
db.sql("""
    CREATE VIEW weird_rides
    AS SELECT pickup_at, dropoff_at, trip_distance, total_amount
    FROM 's3://robotaxi-inc/daily-ride-data/*.parquet'
    WHERE fare_amount > 100 AND trip_distance < 10.0
""")
db.close()
```

This produces a tiny file called `weird_rides.db`. It contains none of the actual data. What it does contain are the above instructions for how to work with this pile of blobs, as if it were a single table of relevant data points.

You publish this database file to blob storage as well, right next to your data. Now you have a single thing to link to: `s3://robotaxi-inc/virtual-datasets/weird_rides.db`.

Instead of opening a web browser, the receipient of the link starts their own local DuckDB session, and attaches to the referenced database.

```
# Receive
import duckdb
conn = duckdb.connect()
conn.sql("""
ATTACH 's3://robotaxi-inc/virtual-datasets/weird_rides.db'
AS rides_db (READ_ONLY)
""")
```

```
conn.sql("SELECT * FROM rides_db.weird_rides LIMIT 5")
#  ┌─────────────────────┬─────────────────────┬───────────────┬──────────────┐
#  │      pickup_at      │     dropoff_at      │ trip_distance │ total_amount │
#  │      timestamp      │      timestamp      │     float     │    float     │
#  ├─────────────────────┼─────────────────────┼───────────────┼──────────────┤
#  │ 2019-04-01 00:03:20 │ 2019-04-01 00:03:54 │           0.0 │       240.35 │
#  │ 2019-04-01 00:16:16 │ 2019-04-01 00:16:21 │           0.0 │       138.36 │
#  │ 2019-04-01 02:01:22 │ 2019-04-01 02:01:28 │           0.0 │       192.96 │
#  │ 2019-04-01 06:26:44 │ 2019-04-01 06:27:14 │           0.0 │        115.3 │
#  │ 2019-04-01 07:28:12 │ 2019-04-01 07:28:12 │           0.0 │        127.2 │
#  └─────────────────────┴─────────────────────┴───────────────┴──────────────┘
```

This query is the first time that any data has to be downloaded from S3. DuckDB supports partial reading, which means that only the columns used in the definition of the `weird_rides` view have to be fetched, and that filters like `fare_amount > 100` can be used to discard even more irrelevant data during the download.

From the perspective of the receiver, this way of accessing the data will remain working unchanged, almost no matter what happens to the underlying data. Changes in format, different partitioning strategies, schema changes – through all of it the receiver's view remains the same.

With DuckDB as a browser for the data cloud, relational datasets are always just a hyperlink away.

Get in touch via 𝕏 (Twitter)