DuckDB - Parquet & query plans

Lecture 18

Dr. Colin Rundel

Data and plans

Data

One month of yellow taxi trips in New York City, from the TLC's trip record data. The September 2026 file is 3.9 million rows and 21 columns in 67 MB.

Preview | Schema

DuckDB

```
D SELECT *
  FROM 'yellow_tripdata_2026-09.parquet';
```

VendorID int32	tpep_pickup_datetime timestamp	… …	Airport_fee double	cbd_congestion_fee double	request_source varchar
2	2026-09-01 00:06:31	…	2.0	0.0	NULL
2	2026-09-01 00:45:04	…	0.0	0.75	NULL
1	2026-09-01 00:10:55	…	0.0	0.75	NULL
.	.	…	.	.	.
.	.	…	.	.	.
.	.	…	.	.	.
2	2026-09-30 23:15:30	…	NULL	0.75	HV0003
2	2026-09-30 23:56:47	…	NULL	0.75	HV0003
2	2026-09-30 23:32:15	…	NULL	0.75	HV0003

3876044 rows (6 shown) use .last to show entire result 21 columns (5 shown)

The chunks in this deck run from a directory holding the data files, so file names appear without a path. The files are

EXPLAIN

Prefixing a query with `EXPLAIN` returns the plan DuckDB will run rather than the result. The plan is a tree of operations, read from the bottom up.

DuckDB

```
D EXPLAIN
  SELECT payment_type, count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet'
  GROUP BY ALL
  ORDER BY n DESC;
```

```
┌─ Order By ─────────────────┐
│ Order By: n DESC           │
│ ~6 rows                    │
└────────────────────────────┘
┌─ Perfect Hash Group By ────┐
│ Groups: #0                 │
│ Aggregates: count_star()   │
│ ~6 rows                    │
└────────────────────────────┘
┌─ Projection ───────────────┐
│ Projections: payment_type  │
│ ~3,876,044 rows            │
└────────────────────────────┘
┌─ Parquet Scan ─────────────┐
│ Function: PARQUET_SCAN     │
│ Projections: payment_type  │
│ ~3,876,044 rows            │
└────────────────────────────┘
```

`EXPLAIN` is not part of ANSI SQL but is widely supported - what it returns and how it is read varies widely between engines.

Reading a plan

The following are some of the steps you will see in DuckDB's plans:

Operator	Description
`* Scan`	Read a table or file, e.g. `Table Scan`, `Parquet Scan`
`Filter`	Apply a `WHERE` condition
`Projection`	Compute and select output columns (`SELECT`)
`Hash Join`	Join by building a hash table on one input and probing with the other
`Hash Group By`	Group rows and compute aggregates with a hash table (`GROUP BY`)
`Ungrouped Aggregate`	A single aggregate over all rows
`Order By`	Sort rows
`Top N`	`ORDER BY` with a `LIMIT`, keeps only the needed rows

Each box also lists the columns it needs (`Projections`), any filters it applies, and the estimated number of rows, which is what the optimizer uses to choose between plans. Estimates come from statistics such as a column's range or a file's row count, so they can be far from the actual counts.

EXPLAIN ANALYZE

Adding the ANALYZE keyword runs the query and records the number of rows and time per operation.

```
D EXPLAIN ANALYZE
  SELECT payment_type, count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet'
  GROUP BY ALL
  ORDER BY n DESC;
```

```
┌─ Summary ──────────────┐
│ Total Time: 0.0063s │
│ Data Read: 457.4 kB │
└────────────────────────┘

┌─ Order By ──────────────────────────┐
│ Order By: n DESC                    │
│ 6 rows                        0µs │
└─────────────────────┬───────────────┘
┌─ Perfect Hash Group By ─────────────┐
│ Groups: #0                          │
│ Aggregates: count_star()            │
│ 6 rows                     10.0ms │
└─────────────────────┬───────────────┘
┌─ Projection ────────┴───────────────┐
│ Projections: payment_type           │
│ 3,876,044 rows                0µs │
└─────────────────────┬───────────────┘
┌─ Table Scan ────────┴───────────────┐
│ Function: PARQUET_SCAN              │
│ Projections: payment_type           │
│ Total Files Read: 1                 │
│ Filename(s):                        │
```

```
┌──────────────────────────────────────┐
│ yellow_tripdata_2026-09.parquet      │
│ Row Groups Scanned: 4 / 4            │
│ 3,876,044 rows           10.0ms      │
└──────────────────────────────────────┘
```

Aside - No indexes

Unlike most databases, DuckDB rarely uses an index. In SQLite, the trip count by `payment_type` sorts rows into a temporary B-tree to group them.

```
|--SCAN taxi USING COVERING INDEX taxi_payment
`--USE TEMP B-TREE FOR ORDER BY
```

- An index pays off when a query needs a few rows. Analytical queries read much of the table, where one scan beats many lookups.
- Compressed columns and vectorized execution make scans cheap, and zone maps let a scan skip row groups.
- Pre-built indexes must be planned in advance, take space, and slow every write.

DuckDB builds ART indexes for `PRIMARY KEY` and `UNIQUE` constraints or with `CREATE INDEX`. They enforce constraints and

Parquet Details

Row vs column storage

Row oriented (CSV, most databases)

```
name,  salary, dept
Alice, 52000,  Accounting
Bob,   40000,  Accounting
Carol, 30000,  Sales
```

Reading one column still passes over every other field in each row. Values of mixed types side by side compress poorly.

Column oriented (Parquet, DuckDB)

```
name:   [Alice, Bob, Carol, ...]
salary: [52000, 40000, 30000, ...]
dept:   [Accounting, Accounting, Sales]
```

Reading one column skips the others entirely. Values of one type side by side compress well.

Analytical queries touch a few columns of many rows, which is exactly the access pattern column storage is good at. Parquet is the standard columnar file format, used by Spark, Arrow, `pandas`, `polars`, DuckDB, and most cloud warehouses.

Inside a Parquet file

A Parquet file is split by rows first, then by columns.

The rows are divided into row groups. Each row group then holds one column chunk per column, with that column's values for those rows, encoded and compressed in pages.

The footer holds the schema plus, for every column chunk, its location, size, and optional statistics (minimum, maximum, and null count).

Structure	What a reader can do with it
column chunks	read only the columns the query needs (projection pushdown)
row group stats	skip row groups whose min / max cannot match a filter
typed values	load numbers and dates in their binary form, without converting text
encodings	store repeated values once (dictionary) or as runs (run length)
footer	learn the schema and plan without reading the data

File metadata

DuckDB exposes the footer through table functions. Our file has four large row groups of about a million rows each, with statistics recorded for every column chunk.

DuckDB

```
D SELECT num_rows, num_row_groups, format_version, file_size_bytes
  FROM parquet_file_metadata('yellow_tripdata_2026-09.parquet');
```

num_rows int64	num_row_groups int64	format_version int64	file_size_bytes uint64
3876044	4	2	67374896

DuckDB

```
D SELECT path_in_schema, row_group_id AS rg, row_group_num_rows AS n,
         stats_min, stats_max
  FROM parquet_metadata('yellow_tripdata_2026-09.parquet')
  WHERE path_in_schema = 'tpep_pickup_datetime';
```

path_in_schema varchar	rg int64	n int64	stats_min varchar	stats_max varchar
tpep_pickup_datetime	0	1048576	2026-08-31 23:46:38	2026-09-21 00:38:58
tpep_pickup_datetime	1	1048576	2009-01-01 09:32:44	2026-09-28 00:32:42
tpep_pickup_datetime	2	1048576	2026-09-01 00:00:02	2026-10-07 07:40:40
tpep_pickup_datetime	3	730316	2026-09-13 07:00:00	2026-09-30 23:59:59

Counting from metadata

The Parquet footer already records the row count. DuckDB can answer this unfiltered `count(*)` without reading the data columns, just as it did in Lec17.

DuckDB

```
D SELECT count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet';
```

```
┌──────────┐
│    n     │
│  int64   │
├──────────┤
│ 3876044  │
└──────────┘
```

DuckDB

```
D EXPLAIN
  SELECT count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet';
```

```
┌─ Column Data Scan ─┐
│ ~1 row             │
└────────────────────┘
```

DuckDB

```
D EXPLAIN ANALYZE
  SELECT count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet';
```

```
┌─ Summary ──────────┐
│ Total Time: 0.0000s│
│ Data Read: 131.0 kB│
└────────────────────┘
```

Projection pushdown

The scan reads only the columns a query refers to, and the summary box reports how much of the file that was.

```
D EXPLAIN ANALYZE
  SELECT sum(trip_distance)
  FROM 'yellow_tripdata_2026-09.parquet';
```

```
┌─ Summary ─────────────┐
│ Total Time: 0.0040s │
│ Data Read:   5.5 MB │
└───────────────────────┘

┌─ Ungrouped Aggregate ─────────────┐
│ Aggregates: sum(#0)               │
│ 1 row                        0µs  │
└───────────────────────────────────┘
┌─ Projection ──────────────────────┐
│ Projections: trip_distance        │
│ 3,876,044 rows               0µs  │
└───────────────────────────────────┘
┌─ Table Scan ──────────────────────┐
│ Function: PARQUET_SCAN            │
│ Projections: trip_distance        │
│ Total Files Read: 1               │
│ Filename(s):                      │
│ yellow_tripdata_2026-09.parquet   │
│ Row Groups Scanned: 4 / 4         │
│ 3,876,044 rows            10.0ms  │
└───────────────────────────────────┘
```

```
D EXPLAIN ANALYZE
  SELECT sum(trip_distance), sum(total_amount),
         sum(tip_amount)
  FROM 'yellow_tripdata_2026-09.parquet';
```

```
┌─ Summary ─────────────┐
│ Total Time: 0.0108s │
│ Data Read:  15.7 MB │
└───────────────────────┘

┌─ Ungrouped Aggregate ─────────────┐
│ Aggregates: sum(#0), sum(#1),     │
│             sum(#2)               │
│ 1 row                     10.0ms  │
└───────────────────────────────────┘
┌─ Projection ──────────────────────┐
│ Projections: trip_distance,       │
│              total_amount,        │
│              tip_amount           │
│ 3,876,044 rows               0µs  │
└───────────────────────────────────┘
┌─ Table Scan ──────────────────────┐
│ Function: PARQUET_SCAN            │
│ Projections: trip_distance,       │
│              total_amount,        │
│              tip_amount           │
│ Total Files Read: 1               │
│ Filename(s):                      │
│ yellow_tripdata_2026-09.parquet   │
│ Row Groups Scanned: 4 / 4         │
│ 3,876,044 rows            30.0ms  │
└───────────────────────────────────┘
```

Filter pushdown and zone maps

Filter pushdown applies a condition inside the scan, so only matching rows reach later operators. Row-group pruning uses the footer's min / max statistics, which DuckDB calls zone maps, to skip groups that cannot contain a match. Pushdown can help even when no groups can be skipped.

DuckDB

```
D EXPLAIN ANALYZE
  SELECT count(*)
  FROM 'yellow_tripdata_2026-09.parquet'
  WHERE tpep_pickup_datetime >= '2026-09-25';
```

```
┌─ Summary ──────────┐
│ Total Time: 0.0088s │
│ Data Read:  12.9 MB │
└────────────────────┘

┌─ Ungrouped Aggregate ──────────────────────────────────┐
│ Aggregates: count_star()                               │
│ 1 row                                              0µs │
└────────────────────────────┬───────────────────────────┘
┌─ Table Scan ───────────────┴───────────────────────────┐
│ Function: PARQUET_SCAN                                 │
│ Filters:                                               │
│ tpep_pickup_datetime >= '2026-09-25 00:00:00'::TIMESTAMP │
│ Total Files Read: 1                                    │
│ Filename(s):                                           │
│ yellow_tripdata_2026-09.parquet                        │
│ Row Groups Scanned: 3 / 4                              │
│ 791,526 rows                                      20.0ms │
└────────────────────────────────────────────────────────┘
```

Sorting the file

Rewriting the file with DuckDB sorted by pickup time changes the row group size, from four groups of about a million rows to 32 of ~120,000, and makes each of those smaller groups cover a narrow range of times.

DuckDB

```
D COPY (FROM 'yellow_tripdata_2026-09.parquet' ORDER BY tpep_pickup_datetime)
    TO 'yellow_tripdata_sorted.parquet';
```

DuckDB

```
D SELECT path_in_schema, row_group_id, stats_min, stats_max
  FROM parquet_metadata('yellow_tripdata_sorted.parquet')
  WHERE path_in_schema = 'tpep_pickup_datetime';
```

path_in_schema varchar	row_group_id int64	stats_min varchar	stats_max varchar
tpep_pickup_datetime	0	2009-01-01 09:32:44	2026-09-02 07:04:14
tpep_pickup_datetime	1	2026-09-02 07:04:14	2026-09-03 08:53:39
tpep_pickup_datetime	2	2026-09-03 08:53:39	2026-09-04 10:16:17
.	.	.	.
.	.	.	.
.	.	.	.
tpep_pickup_datetime	29	2026-09-28 16:07:50	2026-09-29 17:24:12
tpep_pickup_datetime	30	2026-09-29 17:24:13	2026-09-30 16:18:22
tpep_pickup_datetime	31	2026-09-30 16:18:23	2026-10-07 07:40:40

32 rows (6 shown) use .last to show entire result 4 columns

Sort order matters

With the rows in time order, the same filter skips every row group it cannot match.

DuckDB

```
D EXPLAIN ANALYZE
  SELECT count(*)
  FROM 'yellow_tripdata_sorted.parquet'
  WHERE tpep_pickup_datetime >= '2026-09-25';
```

```
┌─ Summary ─────────┐
│ Total Time: 0.0022s │
│ Data Read:   2.7 MB │
└─────────────────────┘

┌─ Ungrouped Aggregate ──────────────────────────────────────┐
│ Aggregates: count_star()                                   │
│ 1 row                                                  0µs │
└────────────────────────────────────────────────────────────┘

┌─ Table Scan ───────────────────────────────────────────────┐
│ Function: PARQUET_SCAN                                     │
│ Filters:                                                   │
│ tpep_pickup_datetime >= '2026-09-25 00:00:00'::TIMESTAMP   │
│ Total Files Read: 1                                        │
│ Filename(s):                                               │
│ yellow_tripdata_sorted.parquet                             │
│ Row Groups Scanned: 7 / 32                                 │
│ 791,526 rows                                        10.0ms │
└────────────────────────────────────────────────────────────┘
```

A filter on `PULocationID` still reads every row group of this file, since sorting by time scatters each location across the whole file. Sort by the columns you filter on.

Skipping whole files

Recall the BTS flight data has one Parquet file per year, in folders with names like `Year=2024` (we called this Hive partitioning). DuckDB reads `Year` from the folder name and uses a filter on it to skip other years' files.

DuckDB

```
D EXPLAIN ANALYZE
  SELECT avg(DepDelay) AS delay
  FROM 'Year=*/*.parquet'
  WHERE Year = 2024;
```

```
┌─ Summary ─────────────┐
│ Total Time: 0.0116s   │
│ Data Read:   8.2 MB   │
└───────────────────────┘

┌─ Projection ──────────────────────────────┐
│ Projections: "/"(#0, CAST(#1 AS DOUBLE))  │
│ 1 row                                  0µs│
└─────────────────┬─────────────────────────┘
┌─ Ungrouped Aggregate ─────────────────────┐
│ Aggregates: sum(#0), count(#1)            │
│ 1 row                                  0µs│
└─────────────────┬─────────────────────────┘
┌─ Projection ──────────────────────────────┐
│ Projections: DepDelay, DepDelay           │
│ 3,461,319 rows                         0µs│
└─────────────────┬─────────────────────────┘
┌─ Table Scan ──────────────────────────────┐
│ Function: PARQUET_SCAN                    │
│ Projections: DepDelay                     │
│ File Filters: Year = 2024                 │
│ Scanning Files: 1/38                      │
└───────────────────────────────────────────┘
```

```
Total Files Read: 1
Filename(s): Year=2024/data_0.parquet
Row Groups Scanned: 48 / 48
3,461,319 rows                    80.0ms
```

`Scanning Files: 1/38` shows partition pruning: whole files are skipped. Row-group pruning can skip groups within the remaining files. This query runs from the flight-data folder used in Lec17 (`/data/rdu/` on the department servers).

Average distance three way

The taxi data as the Parquet file, as a table loaded into DuckDB with `CREATE TABLE`, and as a CSV file.

DuckDB

```
D SELECT avg(trip_distance) AS dist,
         count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet'
  WHERE PULocationID = 132;
```

dist double	n int64
14.889403043154276	151619

Run Time (s): real 0.007 user 0.027430 sys 0.001392

DuckDB

```
D SELECT avg(trip_distance) AS dist,
         count(*) AS n
  FROM taxi
  WHERE PULocationID = 132;
```

dist double	n int64
14.88940304315423	151619

Run Time (s): real 0.001 user 0.010021 sys 0.000710

DuckDB

```
D SELECT avg(trip_distance) AS dist,
         count(*) AS n
  FROM 'yellow_tripdata_2026-09.csv'
  WHERE PULocationID = 132;
```

dist double	n int64
14.889403043154212	151619

Run Time (s): real 0.188 user 0.890606 sys 0.167949

Where the time goes

```sql
D EXPLAIN ANALYZE
  SELECT avg(trip_distance) AS dist, count(*) AS n
  FROM 'yellow_tripdata_2026-09.csv'
  WHERE PULocationID = 132;
```

```
┌─ Summary ─────────────┐
│ Total Time:  0.125s │
│ Data Read: 509.5 MB │
└───────────────────────┘

┌─ Projection ──────────────────────────────┐
│ Projections: "/"(#0, CAST(#2 AS DOUBLE)),  │
│              #1                            │
│ 1 row                                  0µs │
└───────────────────────────────────────────┘
┌─ Ungrouped Aggregate ─────────────────────┐
│ Aggregates: sum(#0), count_star(),         │
│             count(#1)                      │
│ 1 row                                  0µs │
└───────────────────────────────────────────┘
┌─ Projection ──────────────────────────────┐
│ Projections: trip_distance, trip_distance  │
│ 151,619 rows                           0µs │
└───────────────────────────────────────────┘
┌─ Projection ──────────────────────────────┐
│ Projections: #1                            │
│ 151,619 rows                           0µs │
└───────────────────────────────────────────┘
┌─ Filter ──────────────────────────────────┐
│ Expression: PULocationID = 132             │
│ 151,619 rows                        10.0ms │
└───────────────────────────────────────────┘
┌─ Table Scan ──────────────────────────────┐
│ Function: READ_CSV_AUTO                    │
│ Projections: PULocationID, trip_distance   │
│ Total Files Read: 1                        │
│ Filename(s): yellow_tripdata_2026-09.csv   │
│ 3,876,044 rows                     920.0ms │
└───────────────────────────────────────────┘
```

```sql
D EXPLAIN ANALYZE
  SELECT avg(trip_distance) AS dist, count(*) AS n
  FROM 'yellow_tripdata_2026-09.parquet'
  WHERE PULocationID = 132;
```

```
┌─ Summary ─────────────┐
│ Total Time: 0.0069s │
│ Data Read:   8.7 MB │
└───────────────────────┘

┌─ Projection ──────────────────────────────┐
│ Projections: "/"(#0, CAST(#1 AS DOUBLE)),  │
│              #1                            │
│ 1 row                                  0µs │
└───────────────────────────────────────────┘
┌─ Ungrouped Aggregate ─────────────────────┐
│ Aggregates: sum(#0), count_star()          │
│ 1 row                                  0µs │
└───────────────────────────────────────────┘
┌─ Projection ──────────────────────────────┐
│ Projections: trip_distance                 │
│ 151,619 rows                           0µs │
└───────────────────────────────────────────┘
┌─ Table Scan ──────────────────────────────┐
│ Function: PARQUET_SCAN                     │
│ Projections: trip_distance                 │
│ Filters: PULocationID = 132                │
│ Total Files Read: 1                        │
│ Filename(s):                               │
│ yellow_tripdata_2026-09.parquet            │
│ Row Groups Scanned: 4 / 4                  │
│ 151,619 rows                        20.0ms │
└───────────────────────────────────────────┘
```

Compression

Part of the story is also the size of the files on disk (or memory),

Format	File	Size
CSV	`yellow_tripdata_2026-09.csv`	394.4M
Parquet, uncompressed	`yellow_tripdata_2026-09_uncompressed.parquet`	108.5M
DuckDB database	`yellow_tripdata_2026-09.duckdb`	95M
Parquet, Snappy	`yellow_tripdata_2026-09_snappy.parquet`	77.5M
Parquet, `zstd`	`yellow_tripdata_2026-09.parquet`	64.2M

Snappy is DuckDB's default parquet compression scheme since every reader supports it. The original file uses `zstd`,

Parquet vs DuckDB files

A DuckDB database file is laid out like a Parquet file. Each table is split into row groups of 122,880 rows, and each column within a group is stored and compressed separately with min / max statistics.

	Parquet file	**DuckDB file**
Holds	One table	Many tables, views, and constraints
Compression	Encodings plus Snappy or `zstd`	Lightweight encodings chosen for fast reads
Changes	Written once, rewrite to change	`INSERT`, `UPDATE`, `DELETE` in transactions
Datasets	Split across many files and folders	One file
Readers	Nearly every data tool	DuckDB

Parquet is the better choice for sharing and archiving data. A DuckDB file suits data that you query and update repeatedly from DuckDB.

Query plans

TPC-H

TPC-H is a standard benchmark modeled on a wholesale business. A line item is one product entry within an order. DuckDB's `tpch` extension generates the data with `CALL dbgen(sf = 1)`.

One customer → many orders; one order → many line items.

Table	Rows	Columns used here
`customer`	150,000	`c_custkey`, `c_mktsegment`
`orders`	1,500,000	`o_orderkey`, `o_custkey`, `o_orderdate`, `o_orderpriority`, `o_totalprice`
`lineitem`	6,001,215	`l_orderkey`, `l_shipdate`, `l_quantity`, `l_extendedprice`, `l_discount`, `l_returnflag`, `l_linestatus`

`orders` joins `customer` on `o_custkey = c_custkey`, and `lineitem` joins `orders` on `l_orderkey = o_orderkey`.

The full schema has eight tables (also `part`, `supplier`, `partsupp`, `nation`, and `region`) and lives in `tpch.duckdb`, which

Hash joins

Checking every order against every line item would mean trillions of comparisons. A hash join reads each table once instead. Take simplified versions of the two tables,

`orders`

o_orderkey	o_orderpriority
10	HIGH
20	LOW
30	HIGH

`lineitem`

l_item	l_orderkey
A	10
B	10
C	20
D	20
E	40

```sql
SELECT l_item, o_orderpriority
FROM orders JOIN lineitem ON o_orderkey = l_orderkey;
```

The join builds a hash table from the smaller input, keyed on the join column, then probes it with each row of the larger input. Each probe is a direct lookup rather than a search.

Build and probe

1. Build from orders

Hash each order, keyed by `o_orderkey`.

Key (`o_orderkey`)	Value (`o_orderpriority`)
10	HIGH
20	LOW
30	HIGH

2. Probe with line items

Look up each line item's `l_orderkey`.

`l_item`	`l_orderkey`	`o_orderpriority` (lookup result)
A	10	HIGH
B	10	HIGH
C	20	LOW
D	20	LOW
E	40	no match

This inner join produces four rows. Orders 10 and 20 each appear twice because two line items match each. Item E is dropped because no order has key 40, and order 30 is never looked up because no line item refers to it.

Outer joins use the same build and probe, and differ only in what happens to unmatched rows. An unmatched line item, like E, is kept with a `NULL` priority. To keep unmatched orders, like 30, the hash table marks each entry that finds a match

A real query

How much revenue did each order priority bring in since 1998?

DuckDB

```
D SELECT o_orderpriority, count(*) AS n,
         round(sum(l_extendedprice * (1 - l_discount))) AS revenue
  FROM orders JOIN lineitem ON o_orderkey = l_orderkey
  WHERE o_orderdate >= '1998-01-01'
  GROUP BY ALL ORDER BY o_orderpriority;
```

o_orderpriority varchar	n int64	revenue decimal(38,0)
1-URGENT	107012	3885193491
2-HIGH	106337	3870039030
3-MEDIUM	107845	3910329588
4-NOT SPECIFIED	105487	3828180032
5-LOW	108346	3940079945

```
Run Time (s): real 0.018 user 0.062314 sys 0.013773
```

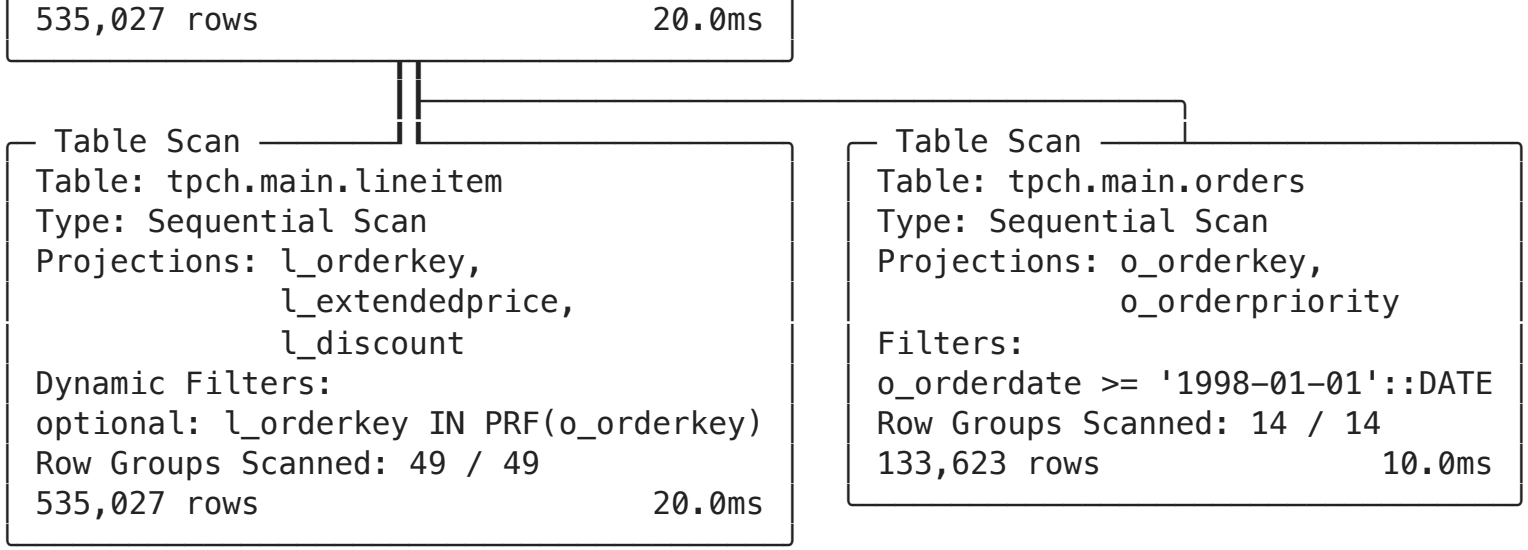
The join plan

DuckDB

```
D EXPLAIN ANALYZE
  SELECT o_orderpriority, count(*) AS n,
         round(sum(l_extendedprice * (1 - l_discount))) AS revenue
  FROM orders JOIN lineitem ON o_orderkey = l_orderkey
  WHERE o_orderdate >= '1998-01-01'
  GROUP BY ALL ORDER BY o_orderpriority;
```

```
┌─ Summary ─┐
│Total Time: 0.0086s│
└───────────┘

┌─ Order By ─────────────────────────────┐
│Order By: o_orderpriority ASC           │
│5 rows                            0µs   │
└────────────────────────────────────────┘
                    │
┌─ Projection ───────────────────────────┐
│Projections: #0, n, revenue             │
│5 rows                            0µs   │
└────────────────────────────────────────┘
                    │
┌─ Hash Group By ────────────────────────┐
│Groups: #0                              │
│Aggregates: count_star(), sum(#1)       │
│5 rows                          10.0ms  │
└────────────────────────────────────────┘
                    │
┌─ Projection ───────────────────────────┐
│Projections:                            │
│o_orderpriority,                        │
│l_extendedprice * (1.00 - l_discount)   │
│535,027 rows                      0µs   │
└────────────────────────────────────────┘
                    │
┌─ Hash Join ────────────────────────────┐
│Join Type: INNER                        │
│Conditions: l_orderkey = o_orderkey     │
```


535,027 rows
20.0ms
Table Scan
Table: tpch.main.lineitem
Type: Sequential Scan
Projections: l_orderkey,
l_extendedprice,
l_discount
Dynamic Filters:
optional: l_orderkey IN PRF(o_orderkey)
Row Groups Scanned: 49 / 49
535,027 rows
20.0ms
Table Scan
Table: tpch.main.orders
Type: Sequential Scan
Projections: o_orderkey,
o_orderpriority
Filters:
o_orderdate >= '1998-01-01'::DATE
Row Groups Scanned: 14 / 14
133,623 rows
10.0ms

Reading the join plan

- `Table Scan` of `orders` (right) applies the date filter inside the scan and reads only `o_orderkey` and `o_orderpriority`. Its 133,623 rows form the build side.
- `Table Scan` of `lineitem` (left) reads three columns. Its `Dynamic Filters` line uses the order keys from the hash table to drop line items with no matching order, so only 535,027 of 6 million rows leave the scan.
- `Hash Join` builds a hash table from the orders, then probes it with the line items. The table must be in memory before probing starts, so the optimizer puts the smaller input on the build side, estimating about 300,000 orders against 6 million line items.
- `Hash Group By` reduces the joined rows to one row per priority. Each order can match several line items, and `n` counts the matched line items.

Optimization

These two queries filter `orders` at different points, after the join and before it. `EXPLAIN` gives the exact same plan for both, so DuckDB runs them identically. Write the version that reads best.

Queries | Plan

DuckDB

```
D SELECT o_orderpriority, count(*) AS n
  FROM orders JOIN lineitem
    ON o_orderkey = l_orderkey
  WHERE o_orderdate >= '1998-01-01'
  GROUP BY ALL;
```

DuckDB

```
D SELECT o_orderpriority, count(*) AS n
  FROM lineitem JOIN (
    FROM orders
    WHERE o_orderdate >= '1998-01-01'
  ) AS o ON l_orderkey = o_orderkey
  GROUP BY ALL;
```

In the shared plan, the optimizer

- moved the date filter into the `orders` scan
- read only the columns each scan needs
- chose `orders`, the smaller input, as the build side
- added a dynamic filter to the `lineitem` scan, visible with `EXPLAIN ANALYZE`

`dbplyr` and `duckplyr`

Two routes from `dplyr` to DuckDB

	`dbplyr`	`duckplyr`
Translates verbs to	SQL sent to the database	DuckDB's relational API, no SQL
Works with	Any database with a backend	DuckDB only
Input	A database table, `tbl()`	A data frame or file, `as_duckdb_tibble()`, `read_parquet_duckdb()`
Result	A lazy table, `collect()` for a tibble	A data frame, computed when needed
Unsupported function	Passed through to SQL as is	Falls back to `dplyr` in R
Semantics	SQL's (`NULL`, ordering, aggregates)	`dplyr`'s, with the same results as `dplyr`

`duckplyr` is the tidyverse's drop-in DuckDB backend. It promises unchanged `dplyr` code, DuckDB speed where possible, and `dplyr` behavior where not.

duckplyr frames

`read_parquet_duckdb()` and `as_duckdb_tibble()` return a `duckplyr_df`. Nothing is read until a result is needed, and printing shows a preview.

```R
taxi = duckplyr::read_parquet_duckdb("yellow_tripdata_2026-09.parquet")
class(taxi)
```

```
[1] "prudent_duckplyr_df" "duckplyr_df"         "tbl_df"
[4] "tbl"                 "data.frame"
```

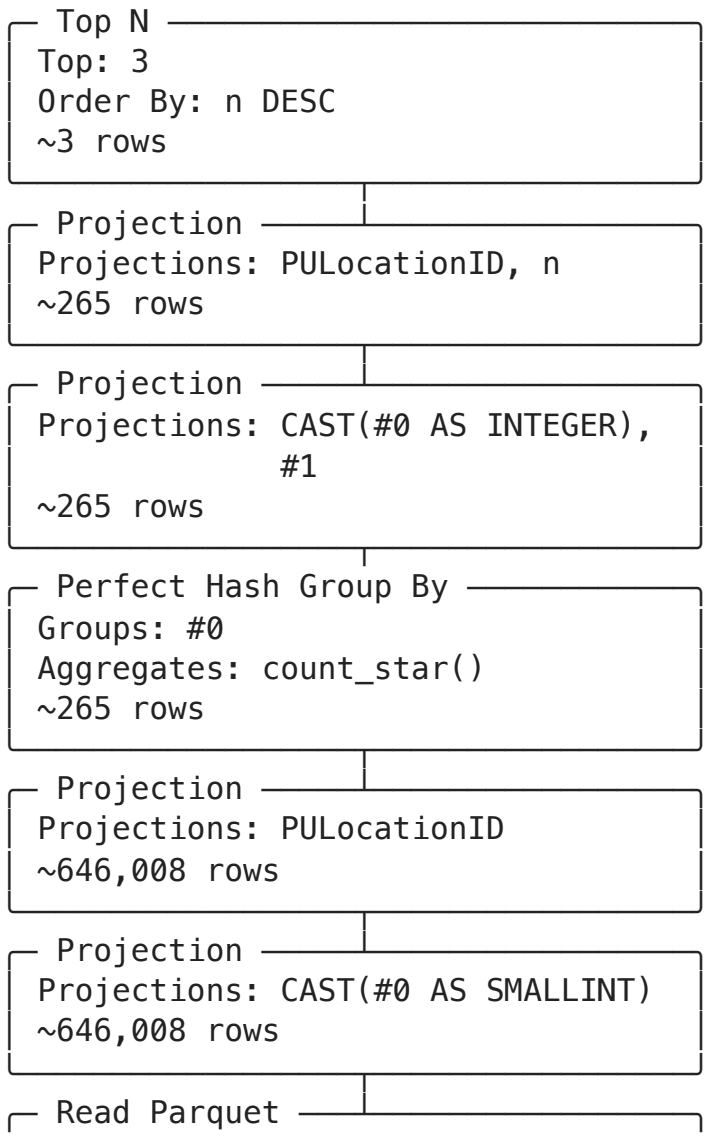
```R
(q = taxi |>
  filter(payment_type == 1) |>
  summarize(n = n(), .by = PULocationID) |>
  arrange(desc(n)) |>
  head(3)
)
```

```
# A duckplyr data frame: 2 variables
  PULocationID      n
         <int>  <int>
1          237 133315
2          236 116457
3          161 105679
```

The plan

`explain()` on a `duckplyr` frame prints the DuckDB plan,

```r
explain(q)
```



```
Function: READ_PARQUET
Projections: PULocationID
Filters:
CAST(payment_type AS DOUBLE) = 1.0
~646,008 rows
```

Large input, small result

Our taxi summary reads millions of trips but returns only three rows.

- The input can be larger than RAM while the result fits comfortably in R or pandas.
- DuckDB can process data in chunks and spill intermediate data to temporary disk storage when needed. Spilling makes disk space and speed part of the query's resource needs.
- Collecting a large result into an ordinary R or pandas data frame still requires memory for that result. Filter and summarize in DuckDB before bringing the result into your analysis.

Materialization

A lazy query describes work that has not yet produced a stored result. Materialization executes that work and stores the result as concrete data.

Like with `dbplyr`, `collect()` explicitly brings the result into R as a tibble:

```r
q |> collect()
```

```
# A tibble: 3 × 2
  PULocationID      n
*        <int>  <int>
1          237 133315
2          236 116457
3          161 105679
```

A `duckplyr` frame can also materialize automatically when an R function asks for its data.

Prudence controls when this is allowed.

Prudence

A `duckplyr` frame is a data frame, so any R function can ask for its rows. Whether that is allowed is controlled by its prudence,

- `lavish`, materialize whenever asked (the default for `as_duckdb_tibble()`),
- `thrifty`, materialize if the result is small (the default for `read_parquet_duckdb()`),
- `stingy`, never materialize automatically, `collect()` or `as_tibble()` must be explicit.

```r
nrow(taxi)
```

```
Error:
! GetQueryResult: Materialization would result in more than 47619 rows. Use `collect()` or
`as_tibble()` to materialize.
```

See the prudence article. Prudence protects against accidentally pulling a 10 GB Parquet file into R because a function

Fallback

When `duckplyr` cannot translate an operation for DuckDB, it falls back to `dplyr`, silently by default. `fallback_config(info = TRUE)` reports when it happens.

```r
duckplyr::fallback_config(info = TRUE)
taxi_lavish = duckplyr::read_parquet_duckdb("yellow_tripdata_2026-09.parquet", prudence = "lavish")
```

```r
taxi_lavish |> filter(tpep_pickup_datetime >= as.POSIXct("2026-09-15", tz = "UTC")) |> count()
```

```
Error processing duckplyr query with DuckDB, falling back to dplyr.
Caused by error in `filter()`:
! Can't translate function `as.POSIXct()`.

# A duckplyr data frame: 1 variable
        n
    <int>
1 2160914
```

`duckplyr` cannot translate this `as.POSIXct()` call, so the filter runs in R after materializing the file. Computing the constant first gives DuckDB a timestamp value it can compare directly:

```r
cutoff = as.POSIXct("2026-09-15", tz = "UTC")
taxi_lavish |> filter(tpep_pickup_datetime >= cutoff) |> count()
```

```
# A duckplyr data frame: 1 variable
        n
    <int>
1 2160914
```

Fallback and prudence

A thrifty or stingy frame refuses a fallback that would materialize too much, so the mistake surfaces as an error rather than a slow silent success.

```r
taxi |> filter(tpep_pickup_datetime >= as.POSIXct("2026-09-15", tz = "UTC")) |> count()
```

```
Error processing duckplyr query with DuckDB, falling back to dplyr.
Caused by error:
! GetQueryResult: Materialization would result in more than 47619 rows. Use `collect()` or
`as_tibble()` to materialize.

Error in `filter()`:
! This operation cannot be carried out by DuckDB, and the input is a
  stingy duckplyr frame.
ℹ Use `compute(prudence = "lavish")` to materialize to temporary storage and
  continue with duckplyr.
ℹ See `vignette("prudence")` for other options.
Caused by error:
! GetQueryResult: Materialization would result in more than 47619 rows. Use `collect()` or
`as_tibble()` to materialize.
```

See the fallback article for what `duckplyr` can and cannot translate. `fallback_sitrep()` summarizes the fallbacks that

Which one

Use `dbplyr` when

- the data already lives in a database, DuckDB or otherwise, and should stay there,
- you want to see or hand edit the SQL, or need a DuckDB function with no R equivalent,
- you need to write tables, which goes through `DBI` and `compute()`.

Use `duckplyr` when

- you have `dplyr` code and data frames or Parquet files, and want it faster without rewriting,
- `dplyr` semantics matter (results must match `dplyr` exactly),
- you are happy to let fallback handle the parts DuckDB cannot.

You can also use both in one pipeline. `duckplyr::as_tbl()` turns a `duckplyr` frame into a `dbplyr` table when you need to see or edit the SQL.

`library(duckplyr)` also calls `methods_overwrite()`, which makes every data frame in the session go through DuckDB.

Python

The ecosystem

In Python, DuckDB works alongside `pandas` and `polars` rather than replacing them.

- `duckdb` adds SQL and a relational API (method chains that build a query lazily) over files, remote data, and data frames already in memory.
- Like `polars`, DuckDB has a lazy, optimized, multithreaded engine. The main difference is the interface, SQL first rather than expression first.
- Unlike `pandas`, DuckDB plans a whole query before running it and uses every core.
- All three exchange data through Arrow, so moving a result from one to another is usually cheap.

DuckDB can query `pandas`, `polars`, and Arrow objects in place, and return results as any of them.

Same file, three readers

`timeit()` runs a function once and prints the elapsed time and the result.

```py
f, col = "yellow_tripdata_2026-09.parquet", "trip_distance"
```

```py
timeit("pandas",          lambda: pd.read_parquet(f)[col].mean())
```

```
pandas              0.368 s   4.969
```

```py
timeit("pandas columns",  lambda: pd.read_parquet(f, columns=[col])[col].mean())
```

```
pandas columns      0.014 s   4.969
```

```py
timeit("polars lazy",     lambda: pl.scan_parquet(f).select(pl.col(col).mean()).collect().item())
```

```
polars lazy         0.015 s   4.969
```

```py
timeit("duckdb",          lambda: duckdb.sql(f"SELECT avg({col}) FROM '{f}'").fetchall()[0][0])
```

```
duckdb              0.017 s   4.969
```

The engines are all fast (other than pandas). The difference is whether the reader was told, or worked out, that only one of the twenty-one columns was needed.

Relational API

`duckdb.read_parquet()`, `duckdb.read_csv()`, and `duckdb.sql()` return a relation (a lazy description of a query). Methods add operations, nothing runs until a result is requested.

This is the same taxi summary as the earlier `duckplyr` pipeline: the three pickup locations with the most trips where `payment_type = 1`.

```py
r = (duckdb.read_parquet("yellow_tripdata_2026-09.parquet")
     .filter("payment_type = 1")
     .aggregate("PULocationID, count(*) AS n")
     .order("n DESC")
     .limit(3)); r
```

PULocationID int32	n int64
237	133315
236	116457
161	105679

Printing a relation runs it for a preview, as a `dbplyr` table does. Expressions inside the methods are SQL fragments.

Getting results out

A relation converts to whichever library you want to continue in. It stores the query rather than its result, so each conversion runs the query again.

```py
r.pl()
```

shape: (3, 2)

PULocationID	n
i32	i64
237	133315
236	116457
161	105679

```py
r.df()
```

```
   PULocationID       n
0           237  133315
1           236  116457
2           161  105679
```

```py
r.fetchall()
```

```
[(237, 133315), (236, 116457), (161, 105679)]
```

```py
r.write_parquet("top3.parquet")
```

Replacement scans

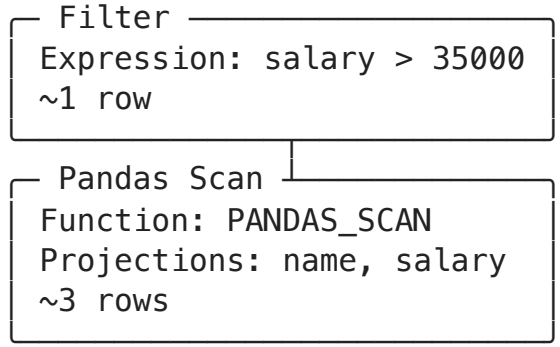
A SQL query can refer to a Python variable by name. DuckDB looks up `pandas`, `polars`, and Arrow objects in the calling scope and scans them in place.

```py
employees = pd.DataFrame({
  "name": ["Alice", "Bob", "Carol"],
  "salary": [52000, 40000, 30000]
})
```

```py
duckdb.sql(
  "FROM employees WHERE salary > 35000"
).df()
```

```
    name  salary
0  Alice   52000
1    Bob   40000
```

```py
print(duckdb.sql(
  "FROM employees WHERE salary > 35000"
).explain())
```



Which tool

	`pandas`	`polars`	`duckdb`
Model	eager, in memory	lazy or eager, in memory	SQL, lazy, spills to disk
Larger than memory	no	streaming for some queries	yes
Parallel	no	yes	yes
Strengths	ecosystem, plotting, modeling inputs	expressions, speed, strict types	SQL, files and remote data, joins at scale
Weak spots	memory, speed, inconsistent API	smaller ecosystem	no in place mutation of frames

In practice they are used together. Read and reduce with DuckDB or `polars`, hand the small result to `pandas` for plotting or modeling.

Summary

Takeaways

- Parquet splits rows into row groups, and each row group stores every column as a separate column chunk. Projection pushdown reads only needed columns; filter pushdown applies conditions inside the scan. Row-group pruning uses min / max statistics to skip groups, and sorting by filter columns makes this more effective.
- For these aggregates, CSV requires a full scan of the text. Parquet is well suited to analytical data at rest, and a DuckDB database can be faster for data queried repeatedly.
- `EXPLAIN` shows the physical plan with estimates, `EXPLAIN ANALYZE` adds actual rows and time per operator. The optimizer chooses how to execute a join, but cannot determine whether its conditions express the relationship you intended.
- `dbplyr` translates to SQL and works with any database. `duckplyr` uses DuckDB directly with `dplyr` semantics and falls back to `dplyr`, which is convenient and occasionally very slow.
- Arrow enables efficient interchange between DuckDB, `polars`, and `pandas`. Some conversions avoid copies; others allocate memory depending on the types and representations involved.

Learning more

- DuckDB performance guide
- Querying Parquet with precision using DuckDB
- DuckDB tricks part 3 (CSV vs Parquet vs native, sorting)
- `duckplyr` articles on prudence and fallback
- DuckDB Python API and relational API

Reference

Other result formats

Other result formats include `to_arrow_table()`, `fetchnumpy()` (a dict of arrays), `fetchall()` (a list of tuples), and `to_arrow_reader(batch_size)` to stream a large result in batches.

Further afield

- Extensions add features to the engine, `spatial` (geometry types and GIS functions), `httpfs` and `azure` (remote storage), `json`, `iceberg` and `delta` (lakehouse table formats), and community extensions.
- `duckdb -ui` opens a notebook style interface in the browser for exploring a database.
- `ibis` is a Python dataframe API that compiles to SQL for around twenty backends, with DuckDB as the default. It plays the role `dbplyr` does in R.
- The DuckDB engine also runs in the browser (`duckdb-wasm`) and from Java, Go, Rust, Node, and Swift.

See extensions and the DuckDB UI.