Improving point queries in DataFusion

Nuno Faria

Boston Apache DataFusion Meetup - Sep 2026

Introduction

- **Point queries:** queries that operate over a relatively small subset of the data.
- Workloads characterized by small operations but at a higher rate.
- For a reasonable performance, reads should complete in a few milliseconds (or less).
- Optimizing point reads in DataFusion:
  - Storage (Parquet files)
  - Execution

```
SELECT *
FROM T
WHERE k = ?
```

```
SELECT *
FROM T1
JOIN T2 ON T1.k = T2.k
WHERE T1.v = ?
```

Environment

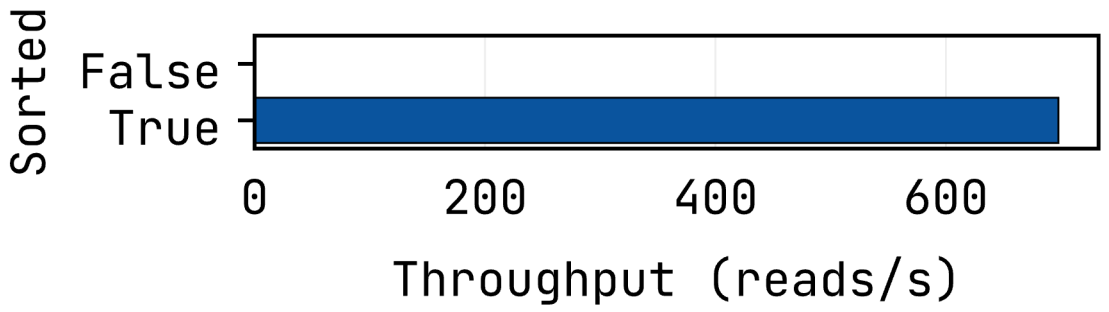
- Machine:
  - Ubuntu 26.04 LTS x86_64
  - c4d-standard-4 (4 vCPUs, 15GB RAM) (for most tests)
    c4d-highcpu-8 (8 vCPUs, 15GB RAM) (for tests that use multiple cores)
  - 100GB HyperDisk, 6k IOPS, 500 MB/s
- Dataset: 100M rows (*k* bigint, *v* varchar) *
- Workload: filter by *k* (1 row), 1 client *
- DataFusion 54, rustc 1.97.1
- Each optimization builds on the last.

* Unless otherwise stated.

Storage

File Ordering

- **Test:** Sorted by key (True) vs random (False).



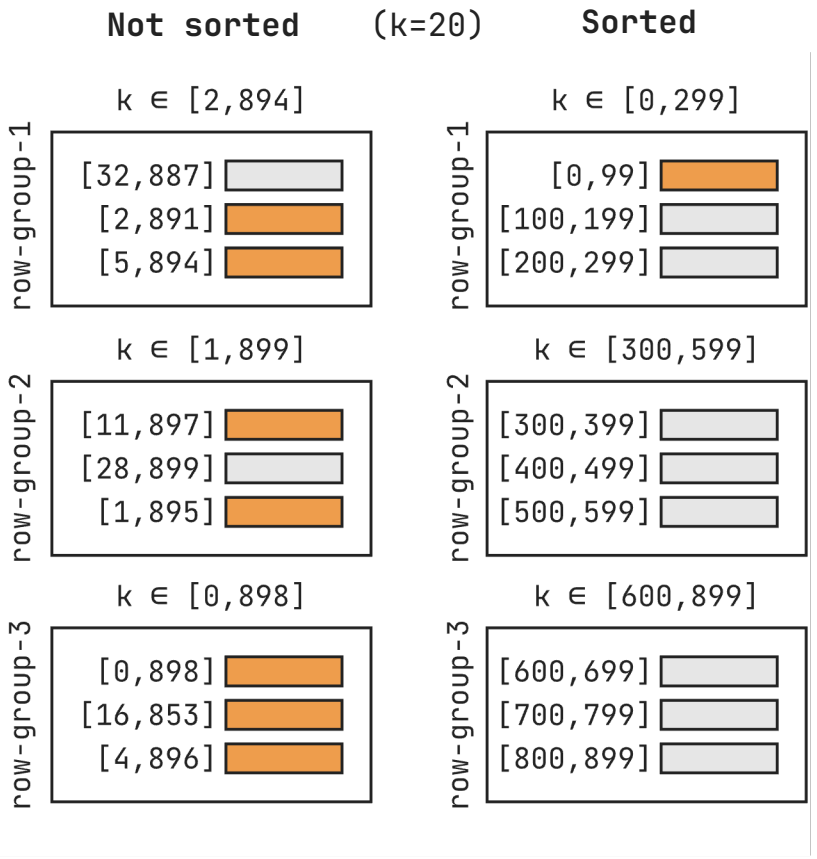
- Point reads on a sorted file are 6000x faster!

Not sorted

```
DataSourceExec
  output_rows=2.46 M
  row_groups=67/96
  pages=122
  scan_time=1.58s
```

Sorted

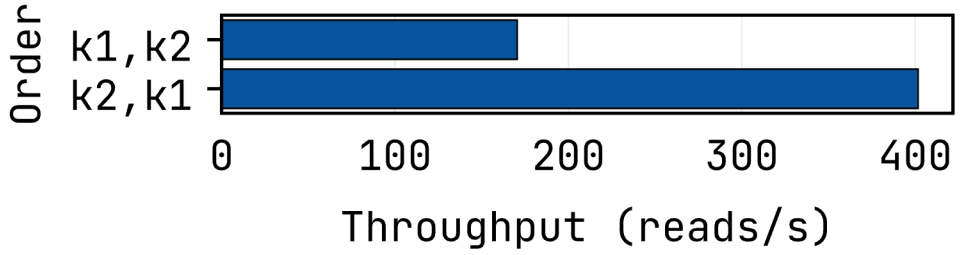
```
DataSourceExec
  output_rows=20.48 K
  row_groups=1/96
  pages=1
  scan_time=414.28µs
```



Storage

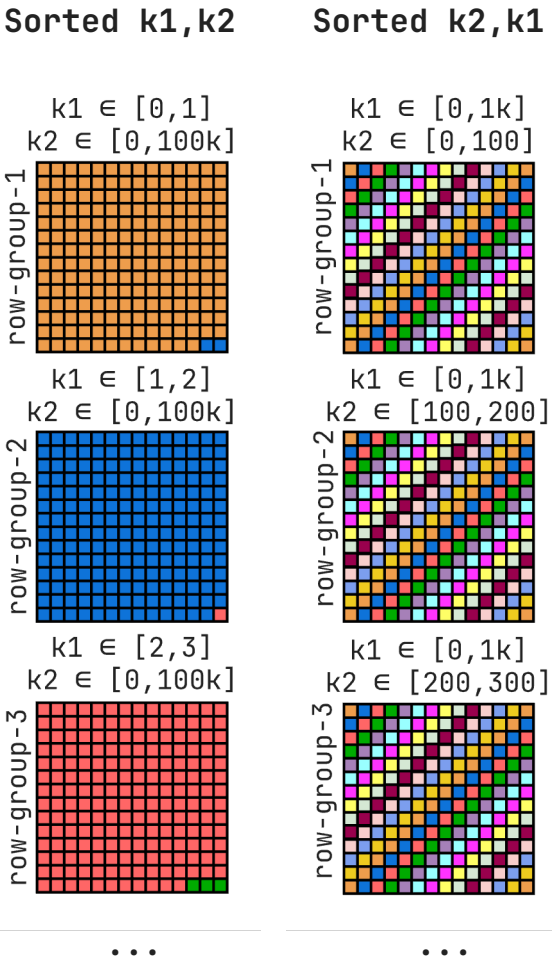
Multi-Column Filters

- If the filtering key has two (or more) columns, sorting the file by k1,k2 might result in multiple row groups being accessed if the number of rows per unique k1 is not divisible by the number of rows in each row group.
- **Test:** Table with schema *k1,k2,v*, sorted by *k1,k2* and *k2,k1*. There are 1k unique k1 values and 100k unique k2 (~100k rows per k1 value). Row group size = 100k.



Sorted by k1,k2
DataSourceExec (avg)
row_groups=2/1K

Sorted by k2,k1
DataSourceExec (avg)
row_groups=1/1k



k1=0 ∧ k2=0
k1,k2: 1 row g
k2,k1: 1 row g

k1=1 ∧ k2=0
k1,k2: 2 row g
k2,k1: 1 row g

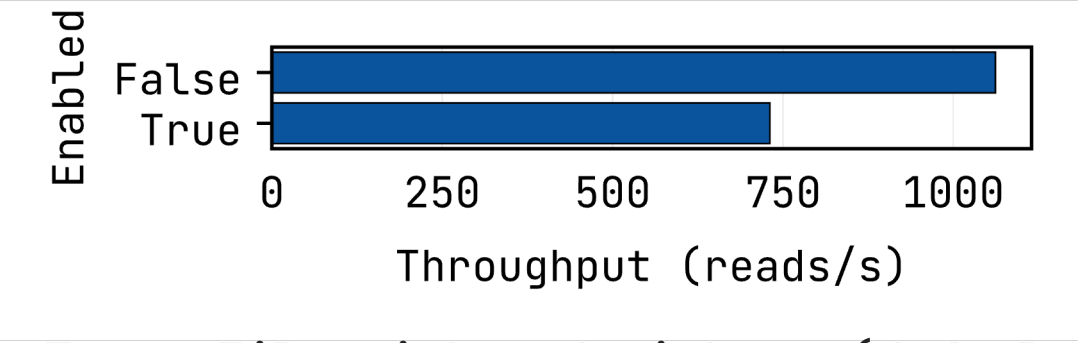
k1=1 ∧ k2=100
k1,k2: 2 row g
k2,k1: 2 row g

In this example, the probability of scanning multiple groups in the file sorted by k2,k1 is reduced, as there are more unique k2 values.

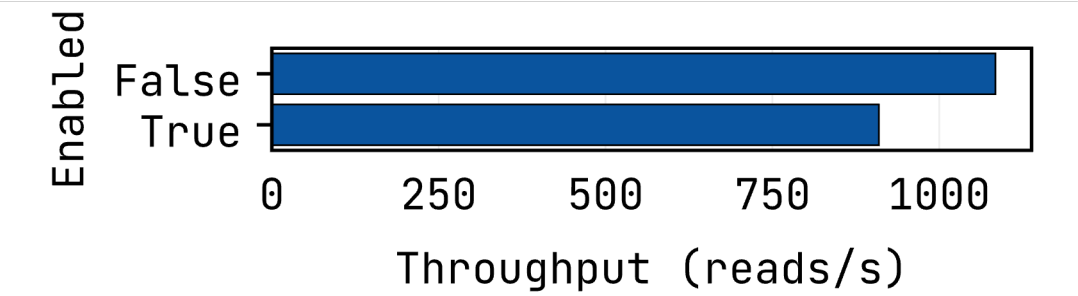
Storage

Additional Metadata

- **Test:** File with (default) and without **dictionary encoding.**



- **Test:** File with and without (default) **bloom filters.**

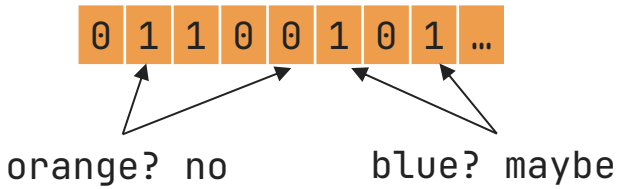


Dictionary page

index	value
0	red
1	green
2	blue

Data page

value
0
0
2
1
1

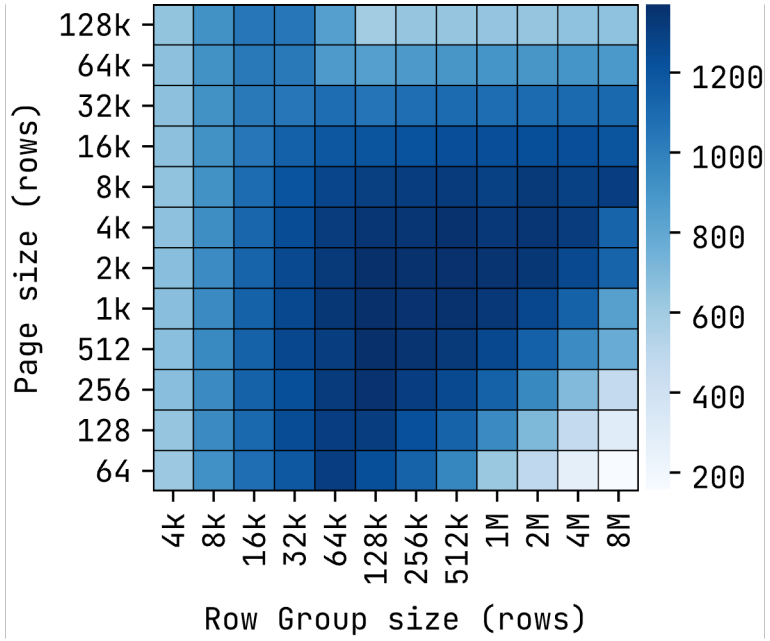


- Disable both for high-cardinality columns.

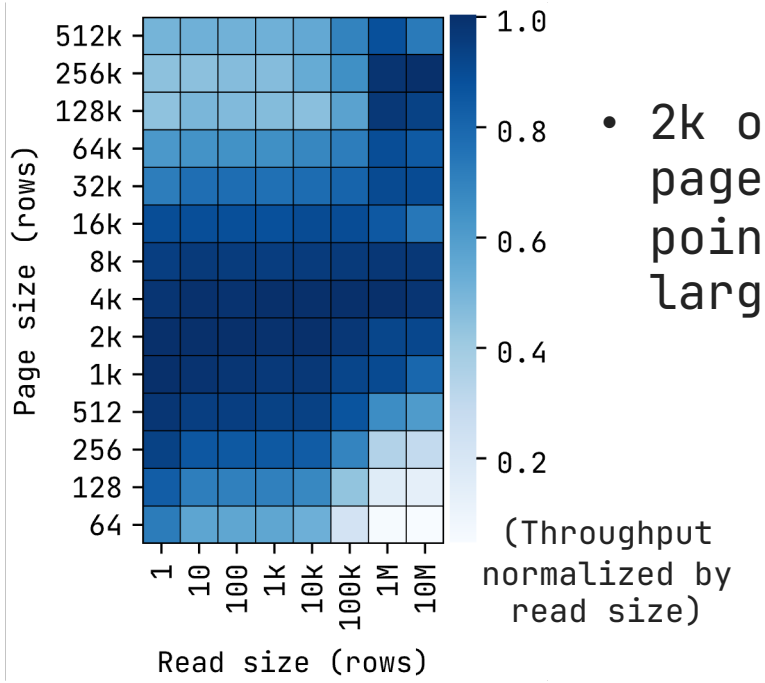
Storage

File Configuration

- **Test:** File with different row group and page sizes. Default: 20k rows/page, 1M rows/group.



- **Test:** Scans of different sizes ("WHERE k BETWEEN ? AND ? + read_size"), with files of different page sizes (row groups fixed to 512k).



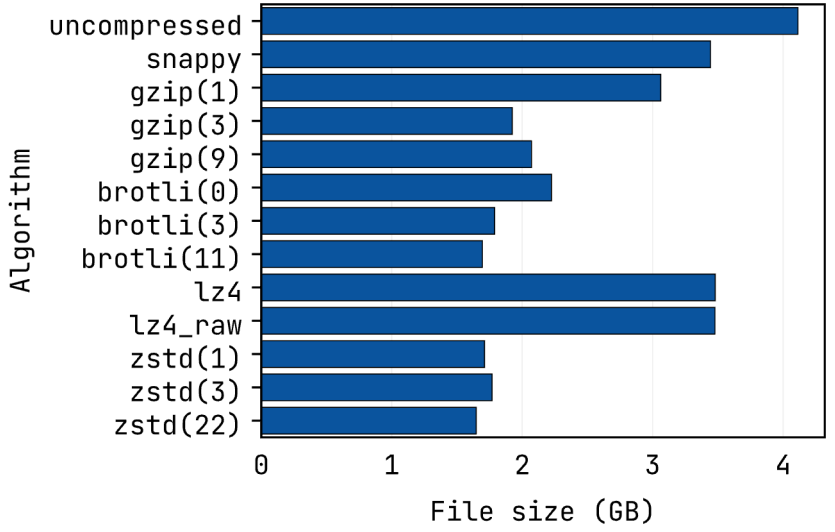
- 2k or 4k sized pages for both point reads and large scans.

Storage
Compression

- **Test:** Different compression algorithms with different read sizes.

Algorithm	1	10	100	1k	10k	100k	1M	10M
uncompressed	1379	1277	1276	1256	1140	585	88	5.4
snappy	1323	1224	1219	1170	856	230	24	3.1
gzip(1)	1185	1073	1050	938	432	76	8.8	.94
gzip(3)	1130	1053	1035	918	413	71	8.1	.90
gzip(9)	1139	1060	1053	936	431	75	8.7	.97
brotli(0)	805	765	741	583	206	32	3.4	.36
brotli(3)	832	786	761	613	222	35	3.7	.42
brotli(11)	941	887	867	703	247	33	3.5	.37
lz4	1357	1250	1246	1203	944	301	38	4.4
lz4_raw	1353	1242	1242	1201	941	299	38	4.3
zstd(1)	1310	1213	1209	1158	855	241	29	3.3
zstd(3)	1301	1205	1198	1145	821	220	27	2.9
zstd(22)	1302	1207	1197	1148	832	226	27	3.0

Read size (rows)

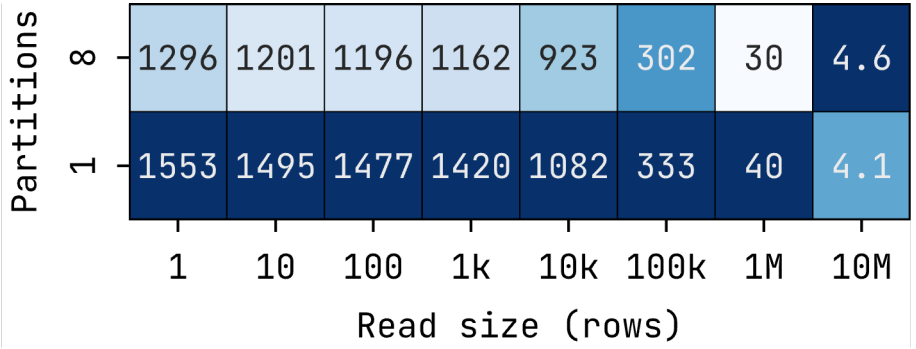


- LZ4 for speed, ZSTD for space.

Execution

Target Partitions

- "target_partitions" controls the parallelism in a query (default = number of cores).
- **Test:** target_partitions set to 1 and 8 with different read sizes.

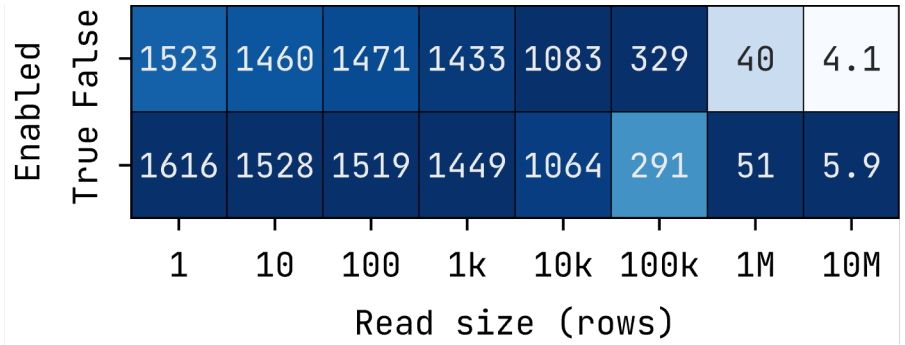


  - For small reads, it is better to set it to 1.

Execution

Pushdown Filters

- "pushdown_filters" pushes filters directly to the Parquet scan, removing a FilterExec node.
- Even disabled (default), row groups and pages are still pruned based on the predicate, but entire pages are returned.
- **Test:** pushdown_filters enabled and disabled with different read sizes.



- Enable for small reads.

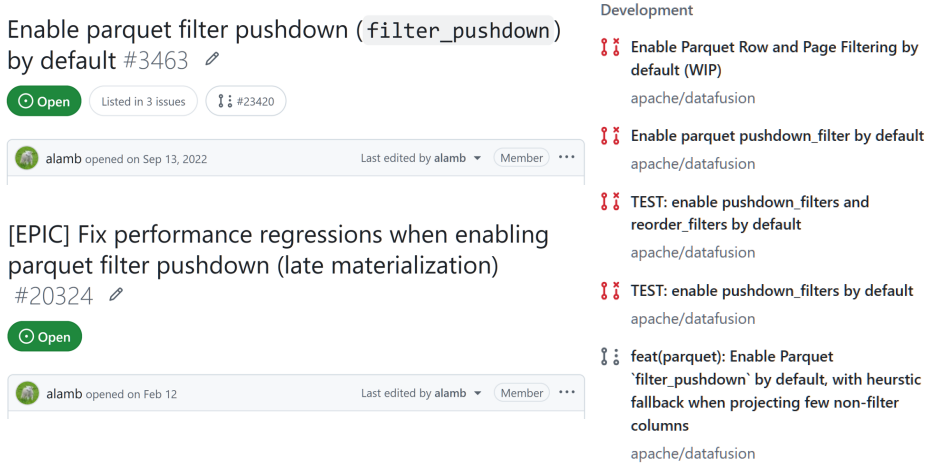
Retrieving a single row

pushdown_filters disabled

```
FilterExec (output_rows=1)
  DataSourceExec (output_rows=1k)
```

pushdown_filters enabled

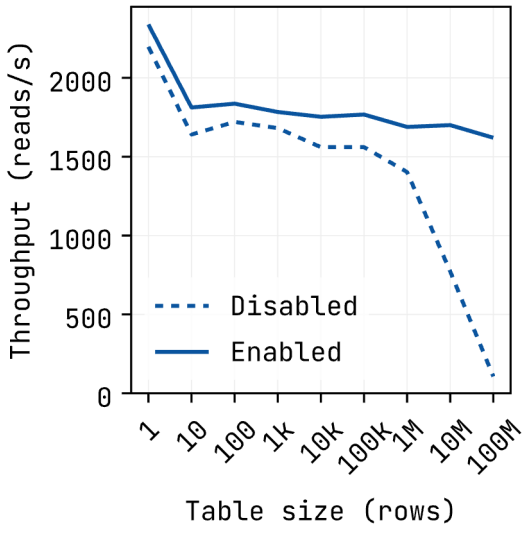
```
DataSourceExec (output_rows=1)
```



Execution

Cache Metadata

- DataFusion caches Parquet metadata since version 50.0.0 (Aug 2025). This reduces repeated work on consecutive reads.
- **Test:** Enabling and disabling the metadata cache with different table sizes.



Plan statistics for 100M rows

Enabled
DataSourceExec:
  time_elapsed_opening=373.17µs

Disabled
DataSourceExec:
  time_elapsed_opening=12.33ms

feat: Cache Parquet metadata in built in parquet reader #16971

Merged alamb merged 4 commits into apache:main from nuno-faria:cache_parquet_metadata on Aug 2, 2025

Conversation 42 Commits 4 Checks 28 Files changed 18

nuno-faria commented on Jul 29, 2025 Member

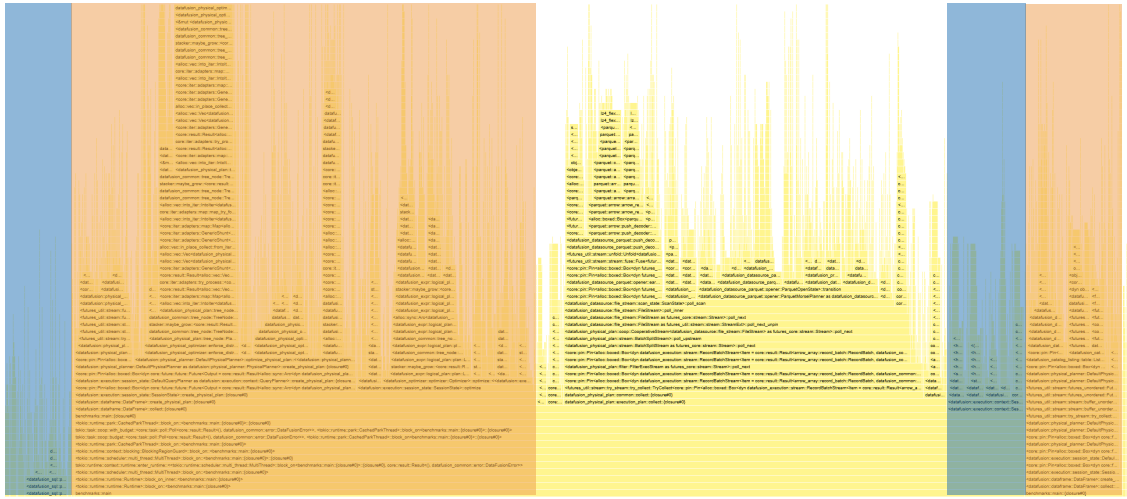
- Keep it enabled. Increase “metadata_cache_limit" if necessary (default=50MB).

```
> SELECT path, metadata_size_bytes, hits
FROM metadata_cache();
+-----------+-----------------------+------+
| path      | metadata_size_bytes | hits |
+-----------+-----------------------+------+
| t.parquet | 19752725              | 1    |
+-----------+-----------------------+------+
```

Execution

Prepared Statements

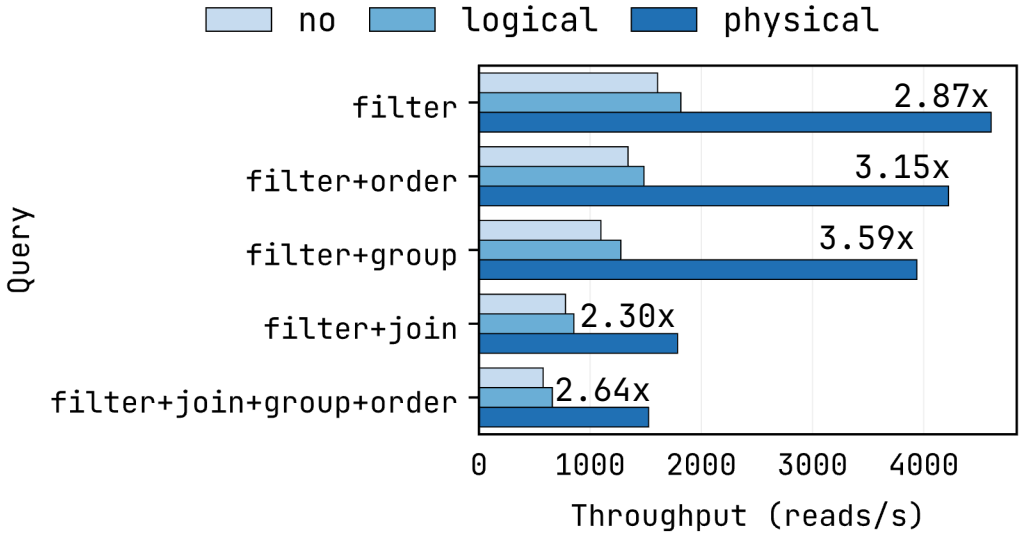
- A large part of the time in small queries is spent on creating the physical plan.



Logical plan: ~12%
Physical plan: ~50%

- DataFusion supports prepared statements, but only reuses the logical plans.

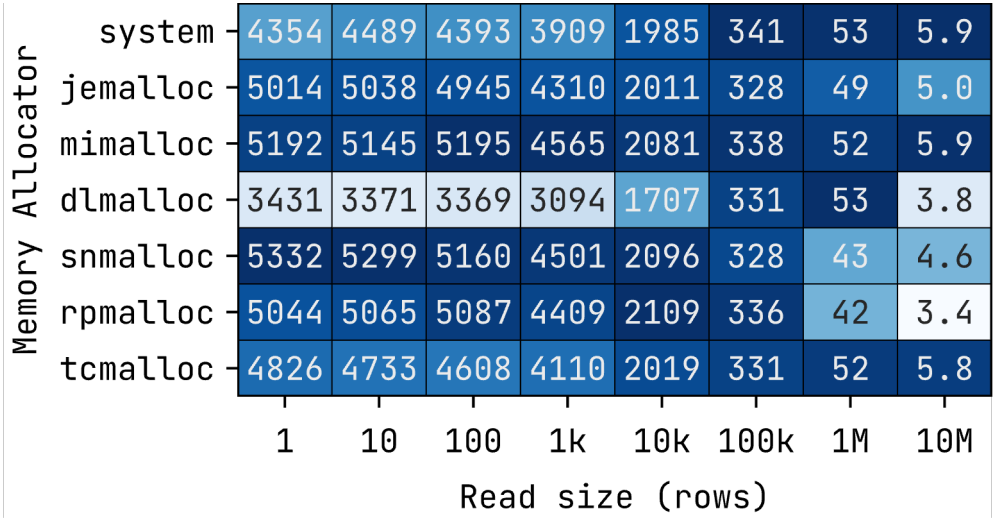
- **Test:** Queries with no prepared statements (no), DataFusion prepared statements (logical), and a custom implementation that reuses physical plans (physical).



Execution

Memory Allocator

- **Test:** Different memory allocators with different read sizes (default=system).



- mimalloc is the best overall choice. snmalloc is also good for point reads.

Execution
Runtime

- DataFusion uses the Tokio runtime. Works well for few large queries with a lot of I/O, but can make it harder to scale multiple clients.

- **Test**: Execute queries with multiple clients with "tokio::task::spawn":

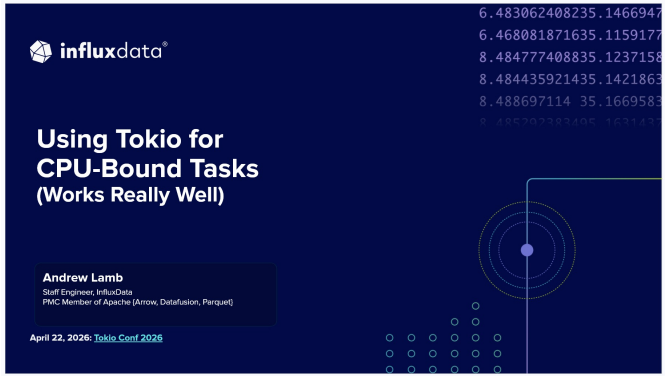
Existing runtime

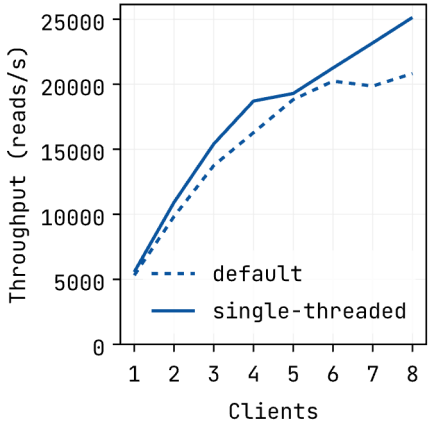
```
tokio::task::spawn(
  async move {
    let ctx = ...
```

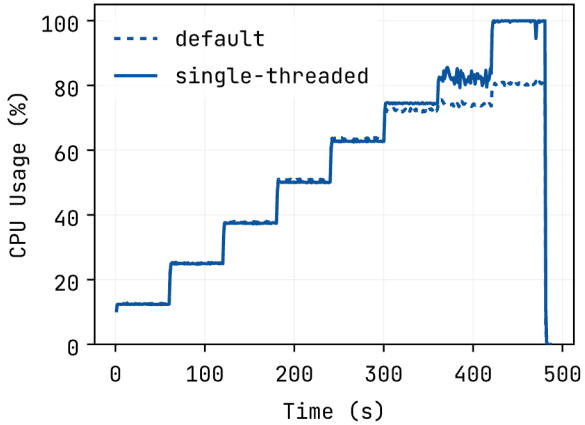
New single-threaded runtime for each client

```
tokio::task::spawn(async move {
  std::thread::spawn(move || {
    let rt = new_current_thread()
      .enable_all()
      .build()
      .unwrap();
    rt.block_on(async {
      let ctx = ...
```

Andrew Lamb talk @ Tokio Conf 2026







Summary

Optimization	Throughput (reads/s)	Response time (ms)
Sorted file	698	1.43
Disable Dictionary Encoding	1062	0.94
Row Group / Page Configuration	1370	0.73
Target Partitions	1553	0.64
Pushdown Filters	1616	0.62
Prepared Statements	4606	0.22
Mimalloc Allocator	5192	0.19
Single-threaded Runtime	5433	0.18

Each optimization builds on the last.

- Throughput 7.78x faster than the sorted file baseline.

Improving point queries in DataFusion

Nuno Faria

Boston Apache DataFusion Meetup - Sep 2026

This work is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) on the scope of the CMU Portugal Program within the project ADAPQO (DOI:10.54499/2024.12585.CMU).