How to Identify and Fix the Small Files Problem in a Data Lake

Step-by-step workshop to detect small file issues and fix them using compaction strategies in Apache Iceberg with Apache Spark.

Joseph Machado

2026-05-06

Table of contents

1 Introduction

Large datasets are stored as individual files. Many small files per dataset make reads expensive!

You might be wondering.

> Why are too many small files a problem? & How to identify this issue on Spark UI?
>
> If there's a strategy to prevent creating tiny files in the first place
>
> Is it practical to consolidate them through an independent process? If yes, how?

Imagine being able to **set up systems to create optimally sized files.**

Ensure you don't get pinged by stakeholders: "Hey, this data is slow, can you check?"

We cover:

1. Why are **many small files a problem**

2. Using **maintenance functions** to optimally size files
3. Optimal **file sizing how-tos for partitioned tables**
4. How **vendors (mostly) handle it automatically**

Its **highly recommended to follow along with code. The setup instructions are available here**.

We use Apache Iceberg in this post, but the concepts apply to delta as well.

2 Many small files => Spark wastes time opening files

Spark excels at processing large data. Opening many small files **wastes compute time ($$$).**

For each file to read, Spark will:

1. Read parquet footer `metadata`.
2. Identify the `data chunk to read`, based on metadata and the query to run.
3. `Read the required data chunk` from the parquet file.

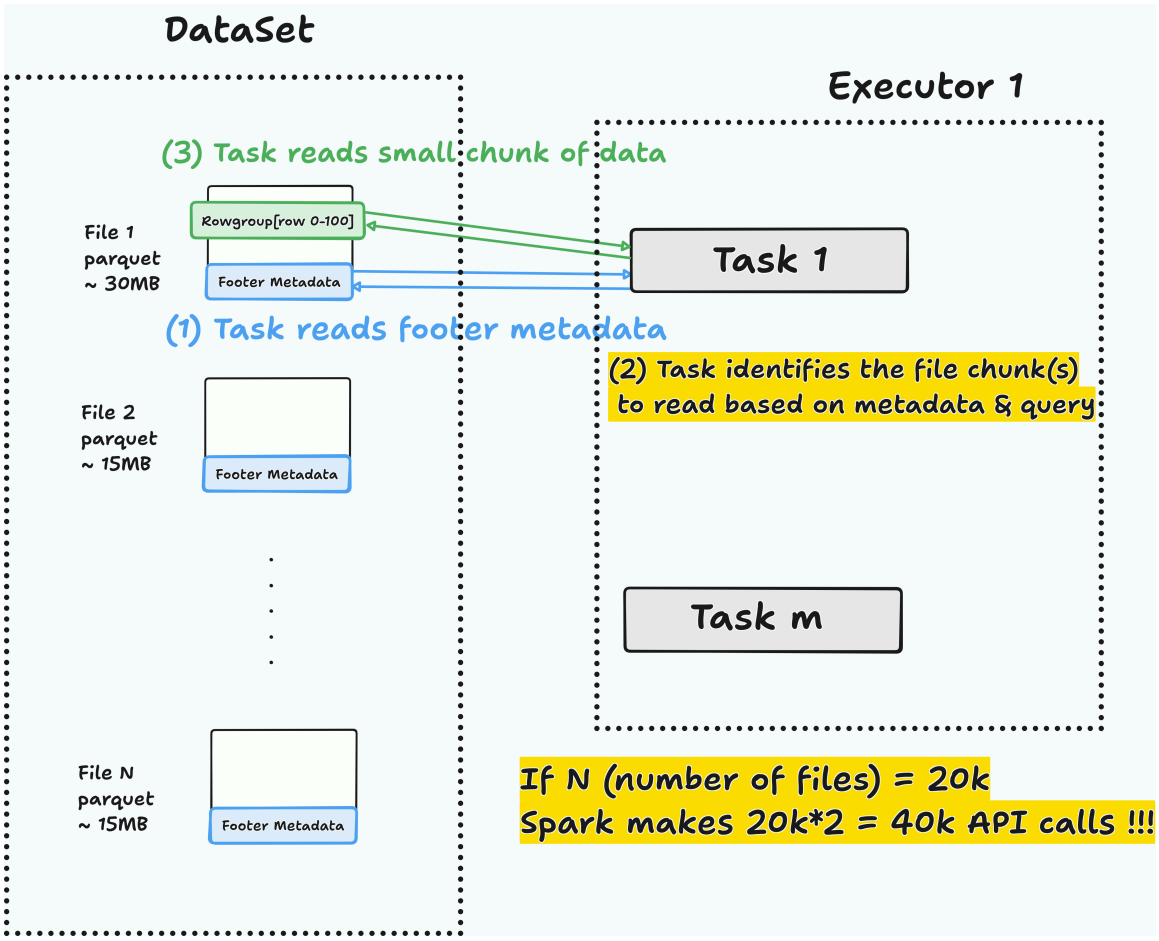


Figure 1: Small Files IO Problem

Let's run a query to check how it performs.

```
%%sql
SELECT
  MONTH (l_receiptdate) AS receipt_month,
  COUNT(*) AS num_line_items
FROM lineitem
GROUP BY 1 ORDER BY 2 desc LIMIT 10
```

Line 2

SQL magic to run Spark SQL

Go to the `SQL/DataFrame` tab in the Spark UI at http://localhost:4040 and select the first read stage.

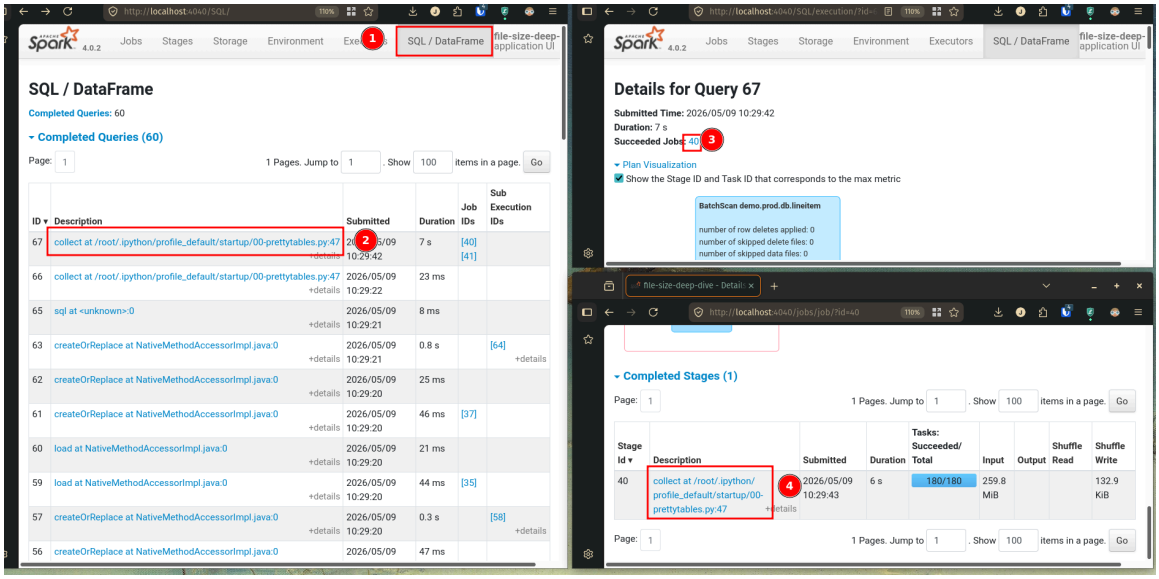
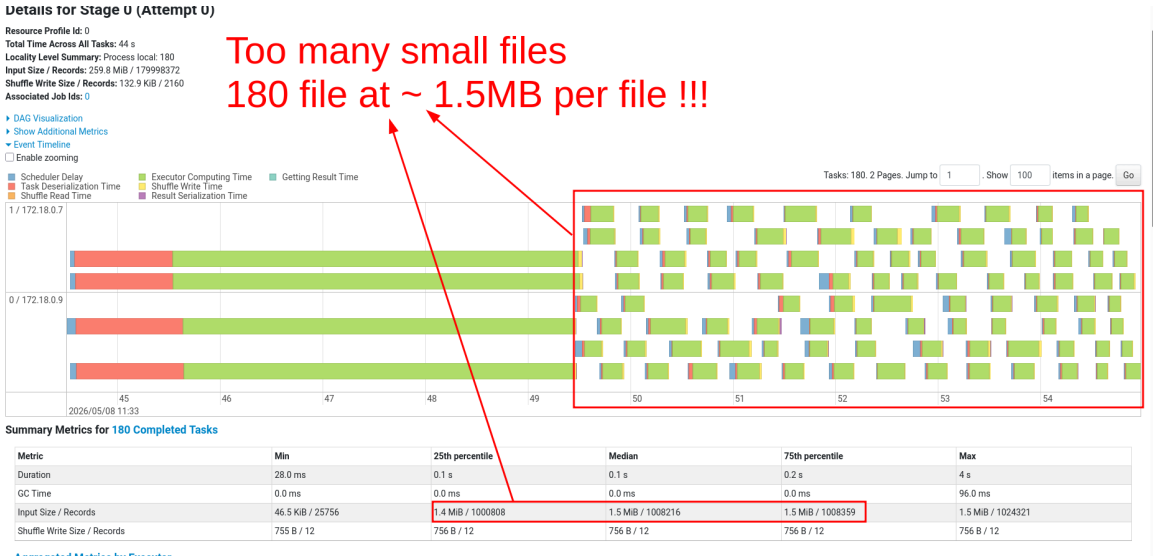


Figure 2: Get to the Stage tab

In the stages tab, we see the `event timeline`. Many small tasks (1 task = 1 green chunk) indicate a many-small-files (or partitions) problem.



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	28.0 ms	0.1 s	0.1 s	0.2 s	4 s
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	96.0 ms
Input Size / Records	46.5 KiB / 25756	1.4 MiB / 1000808	1.5 MiB / 1008216	1.5 MiB / 1008359	1.5 MiB / 1024321
Shuffle Write Size / Records	755 B / 12	756 B / 12	756 B / 12	756 B / 12	756 B / 12

Figure 3: Many small green chunks = many small files

1.5MB is the average input size, from the summary section. This is tiny!

Recommended optimal file size is between 512MB and 1GB.

3 Maintenance function is the simplest fix

Table format maintenance functions combine small files into optimally sized files.

1. Apache Iceberg has rewrite-data-files
2. Delta Lake has optimize

Let's resize our data.

```
%%sql
CREATE TABLE prod.db.lineitem_resized
AS SELECT * FROM prod.db.lineitem;
```

Line 2

Create a new table

```
spark.sql("""CALL
demo.system.rewrite_data_files('prod.db.lineitem_resized')""")
```

The maintenance function combines small files into optimally sized files (default: `512 MB`).

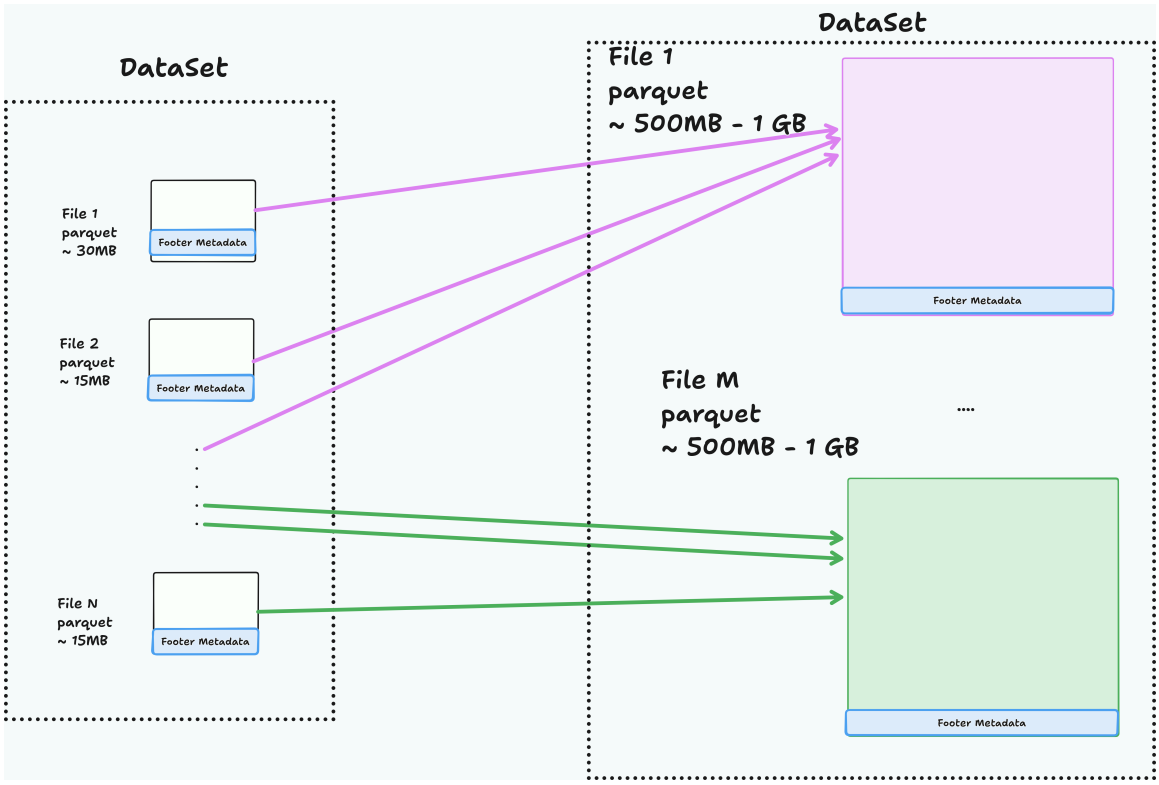


Figure 4: Combining small files to 512MB files

Re-trying our query on the optimized table.

```
%%sql
SELECT
  MONTH (l_receiptdate) AS receipt_month,
  COUNT(*) AS num_line_items
FROM lineitem_resized
GROUP BY 1 ORDER BY 2 desc LIMIT 10
```

Now Spark can concentrate on processing the data.

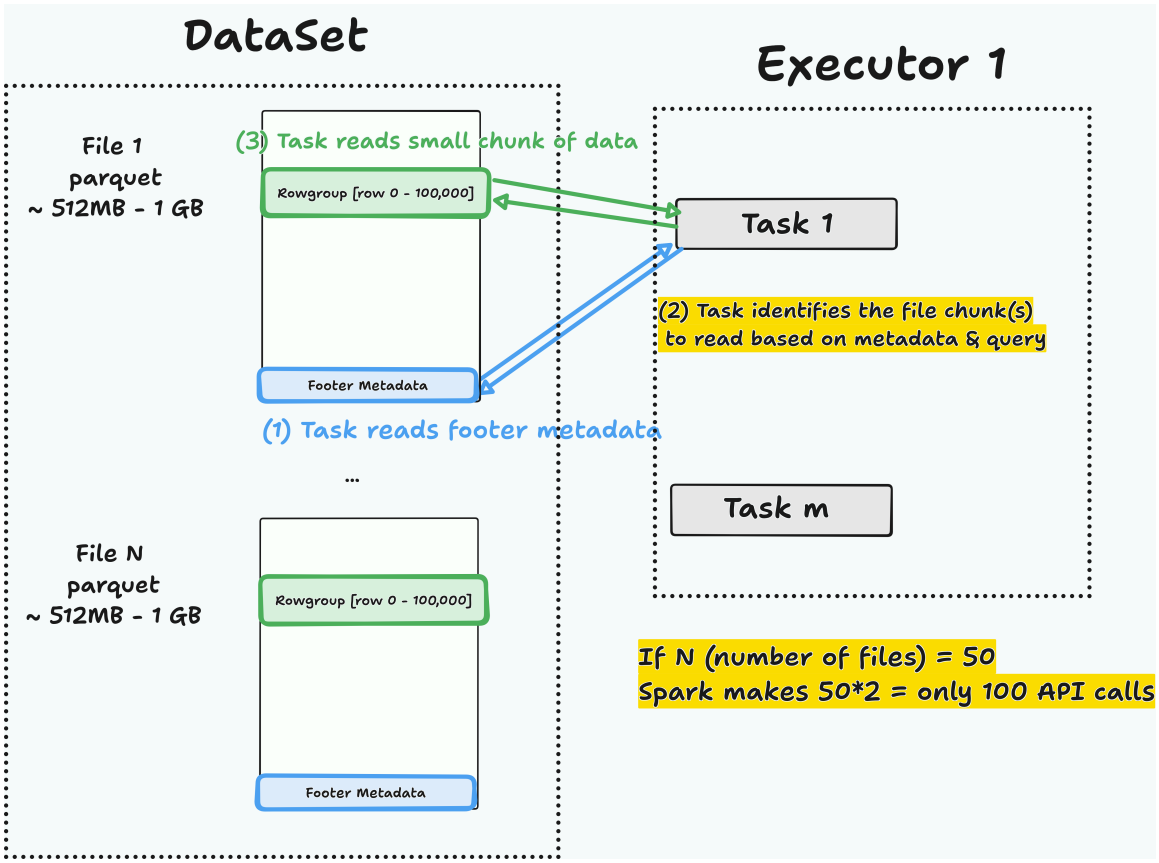


Figure 5: Optimal Files IO

Go to the `SQL/DataFrame` tab in the Spark UI at http://localhost:4040 and select the first read stage for this job.

In the Stages tab, we can see the task event time for longer-running tasks.

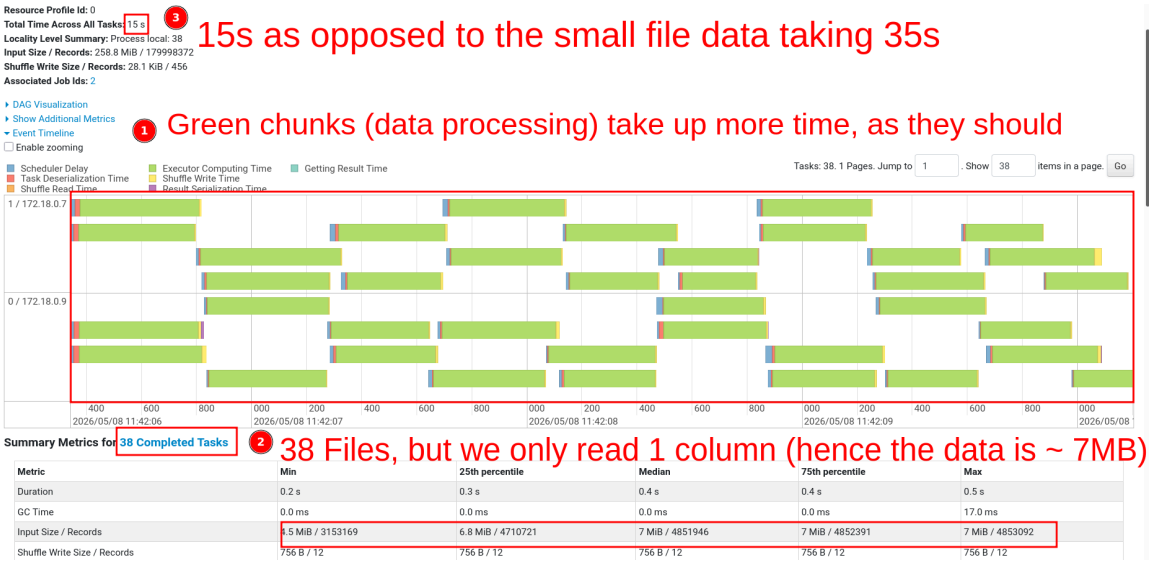


Figure 6: Larger green chunks = Spark processing data

Processing time dropped by **67% (45s → 15s)**.

> **Note**
>
> If you don't own the pipeline or can't afford a longer runtime, schedule a maintenance job. When possible, run maintenance as part of ETL.

The maintenance function includes an **option to target only specific partitions**. And an optional sort order when optimizing file sizes (docs).

E.g., run a daily maintenance function targeting files upserted in the prior day.

4 Partition data before insert

4.1 Inserting into partitioned table does not guarantee optimal file size

Let's see if inserting data into partitioned tables will write files of optimal size.

```
%%sql
CREATE TABLE
  IF NOT EXISTS prod.db.lineitem_part_year (
    l_orderkey BIGINT,
    l_partkey BIGINT,
    l_suppkey BIGINT,
    l_linenumber INT,
    l_quantity DECIMAL(15, 2),
    l_extendedprice DECIMAL(15, 2),
    l_discount DECIMAL(15, 2),
    l_tax DECIMAL(15, 2),
    l_returnflag STRING,
    l_linestatus STRING,
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct STRING,
    l_shipmode STRING,
    l_comment STRING
  ) USING iceberg PARTITIONED BY (YEAR (l_shipdate)) TBLPROPERTIES (
    'format-version' = '2'
  );
```

```
%%sql
INSERT INTO prod.db.lineitem_part_year
SELECT * FROM prod.db.lineitem;
```

```
table_name = 'prod.db.lineitem_part_year'
print_file_sizes(table_name)
```

Line 2

print_file_sizes is a function we define at setup

#	file_path	file_size_mb	writer_task
1	`.../l_shipdate_year=1994/...00001-00001.parquet`	99.98	219
2	`.../l_shipdate_year=1994/...00002-00001.parquet`	100.00	220
3	`.../l_shipdate_year=1993/...00003-00001.parquet`	100.01	206

The Iceberg writer task only allows ~384 MB of memory before writing data to the output file. This is set with the property `spark.sql.iceberg.advisory-partition-size`.

Writing out to parquet results in 4x compression, so 384 MB in-memory ⟶ 100 MB files.

There may be cases where the 384 MB goes to multiple partitions (thus file sizes will be smaller than 100 MB)

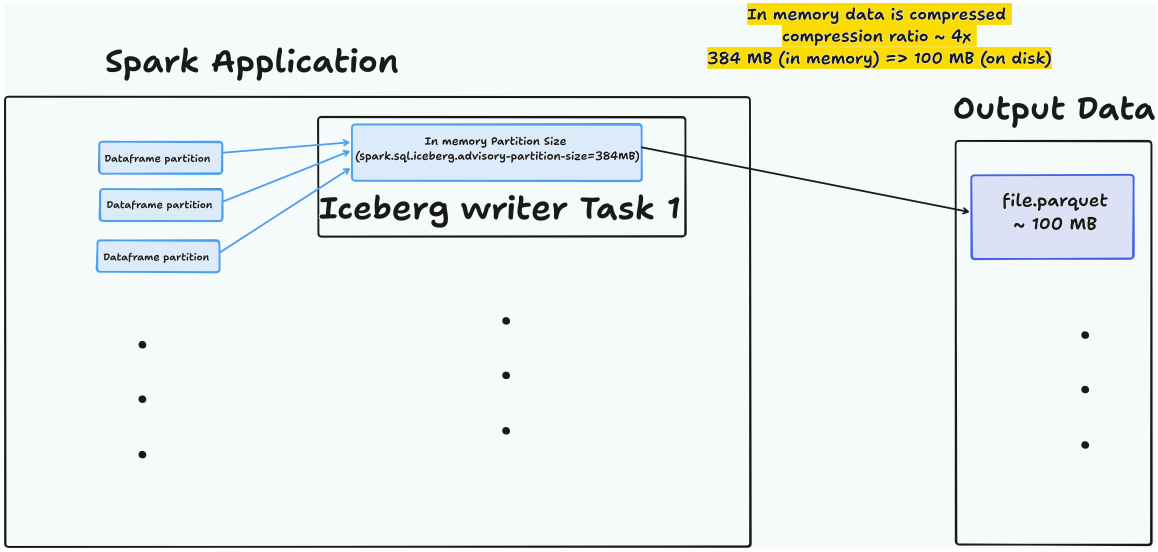


Figure 7: Iceberg Writer

Check out the input size (~350 MB) to output size (~100 MB) in the stages tab.

1. Compression ratio from 368MB -> 100MB
2. The row counts remain the same between input and output

Previous 1 Next

Search:

Duration	GC Time	Shuffle Read Fetch Wait Time	Shuffle Remote Reads	Output Size / Records	Shuffle Read Size / Records	Errors
18 s	0.2 s	0.0 ms	210.5 MiB	105.1 MiB / 4313350	368.8 MiB / 4313350	
17 s	0.1 s	0.0 ms	157.1 MiB	104.6 MiB / 4291555	367 MiB / 4291555	
20 s	0.2 s	0.0 ms	157.6 MiB	104.6 MiB / 4292915	367.1 MiB / 4292915	
18 s	0.2 s	0.0 ms	157.3 MiB	104.6 MiB / 4291401	367 MiB / 4291401	
20 s	0.2 s	0.0 ms	183.4 MiB	104.5 MiB / 4290351	366.9 MiB / 4290351	
20 s	0.2 s	0.0 ms	183.4 MiB	104.5 MiB / 4290113	366.9 MiB / 4290113	
18 s	0.2 s	0.0 ms	209.4 MiB	104.4 MiB / 4284736	366.4 MiB / 4284736	
19 s	0.2 s	0.0 ms	182 MiB	103.8 MiB / 4258685	364.2 MiB / 4258685	

Figure 8: Compression Ratio

4.2 Increasing task memory does not guarantee optimal file size

Let's try with 2GB task memory. Assuming 4x compression, the output files should be ~500 MB.

```
%%sql
CREATE TABLE
  IF NOT EXISTS prod.db.lineitem_part_year_2gb_intask_mem (
    l_orderkey BIGINT,
    l_partkey BIGINT,
    l_suppkey BIGINT,
    l_linenumber INT,
    l_quantity DECIMAL(15, 2),
    l_extendedprice DECIMAL(15, 2),
    l_discount DECIMAL(15, 2),
    l_tax DECIMAL(15, 2),
    l_returnflag STRING,
    l_linestatus STRING,
    l_shipdate DATE,
    l_commitdate DATE,
    l_receiptdate DATE,
    l_shipinstruct STRING,
    l_shipmode STRING,
    l_comment STRING
  ) USING iceberg PARTITIONED BY (YEAR (l_shipdate)) TBLPROPERTIES (
    'format-version' = '2',
    'write.spark.advisory-partition-size-bytes' = '2147483648'
  );
```

Line 22

Setting in-memory size to 2 GB

```
%%sql
INSERT INTO
  prod.db.lineitem_part_year_2gb_intask_mem
SELECT * FROM prod.db.lineitem;
```

```
table_name = 'prod.db.lineitem_part_year_2gb_intask_mem'
print_file_sizes(table_name)
```

Line 2

print_file_sizes is a function we define at setup

#	file_path	file_size_mb	writer_task
1	.../year=1996/00001.parquet	281.27	413
2	.../year=1998/00000.parquet	492.10	412
3	.../year=1994/00009.parquet	510.58	421
4	.../year=1993/00005.parquet	510.67	417

⚠ Warning

- Our executors need sufficient memory to be able to handle the 2GB per task requirement.
- Compressions ratios vary depending on your data, experiment.

Output files are mostly optimally sized.

Previous 1 Next

1. Compression ratio 1.7GB -> 490MB

2. Row counts remain the same

Search:

GC Time	Shuffle Read Fetch Wait Time	Shuffle Remote Reads	Output Size / Records	Shuffle Read Size / Records	Errors
0.8 s	0.0 ms	877.1 MiB	492.1 MiB / 20563469	1.7 GiB / 20563469	
1 s	0.0 ms	995.9 MiB	561.9 MiB / 23312986	1.9 GiB / 23312986	
1.0 s	0.0 ms	1016 MiB	558.6 MiB / 23168518	1.9 GiB / 23168518	
0.4 s	0.0 ms	363.7 MiB	197.8 MiB / 8213962	702.4 MiB / 8213962	
1 s	0.0 ms	980.1 MiB	549.1 MiB / 22807766	1.9 GiB / 22807766	
1 s	0.0 ms	991.5 MiB	546.7 MiB / 22599553	1.9 GiB / 22599553	
0.2 s	0.0 ms	210.1 MiB	113.8 MiB / 4708647	407.6 MiB / 4708647	
1 s	0.0 ms	993.9 MiB	550.5 MiB / 22601501	1.9 GiB / 22601501	
0.3 s	0.0 ms	211.5 MiB	114.8 MiB / 4719891	409.6 MiB / 4719891	
0.7 s	0.0 ms	963.4 MiB	546.5 MiB / 22592955	1.9 GiB / 22592955	
0.3 s	0.0 ms	209.8 MiB	113.8 MiB / 4709124	407.1 MiB / 4709124	

Previous 1 Next

Figure 9: Compression

We can see a few non-optimal files in the stages tab.

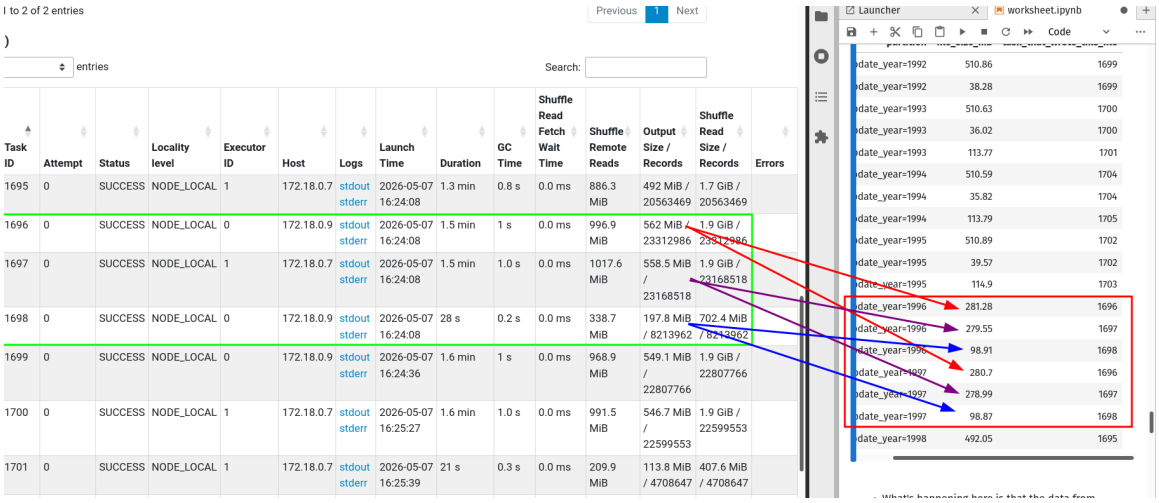


Figure 10: Iceberg writers handling data from mutliple partitions

Here is what's happening. Different `year(l_shipdate)` rows are handled by the same writer, leading to less than optimal file sizes.

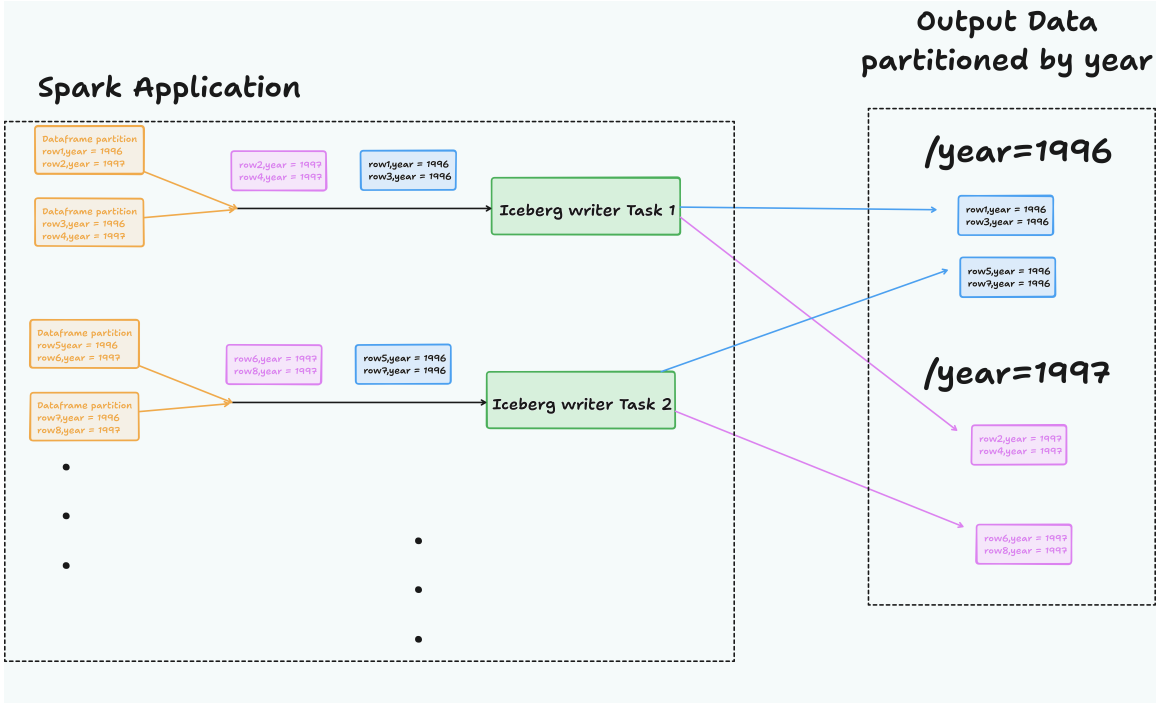


Figure 11: Iceberg Write

4.3 Increase task memory and partition before insert

Partitioning before insert will fix this. Let's see how.

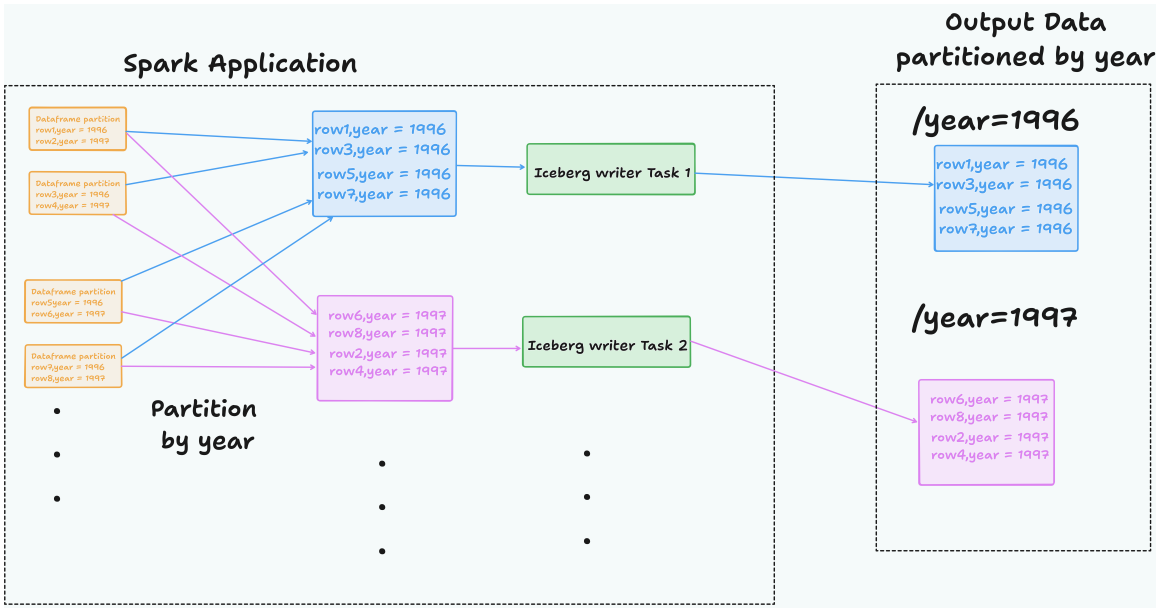


Figure 12: Partition & Iceberg Write

Let's partition data in Spark and then insert it into the table.

```
# This forces same-year rows to the same Icebreg write task
import pyspark.sql.functions as F

spark.table("prod.db.lineitem")\
  .repartition(F.year(F.col("l_shipdate"))) \
  .writeTo("prod.db.lineitem_part_year_2gb_intask_mem") \
  .overwritePartitions()
```

We can see that all the files are optimally sized.

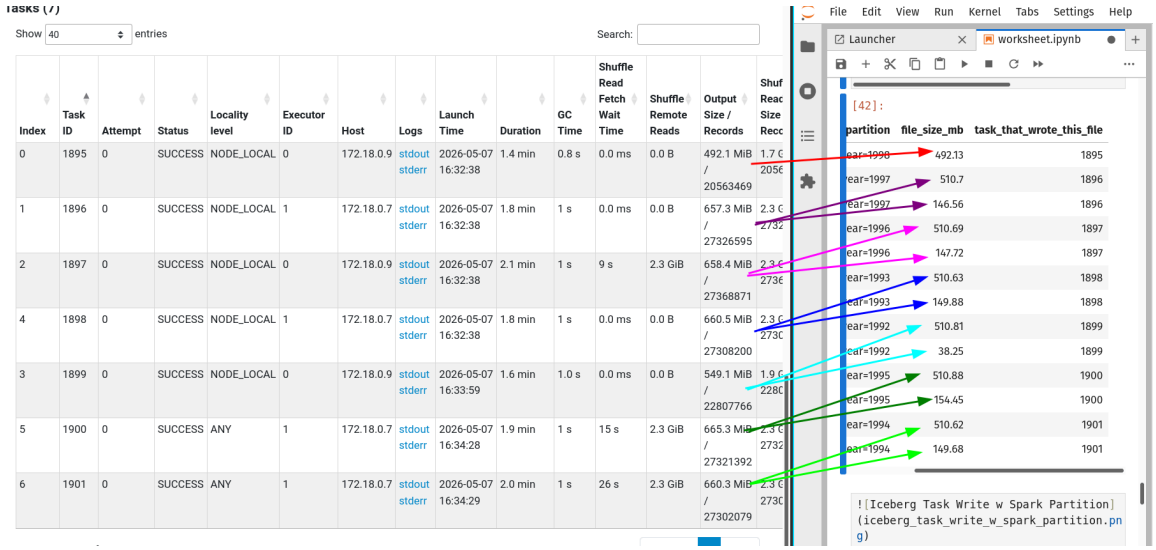


Figure 13: One partition data per iceberg writer

When an output file size exceeds **512MB**, the Iceberg writer opens a new file.

5 Alternatives & future work

An alternative is using `sort` before writing to the output file. This is beneficial for filters on the sorted column(s) and also creates output files of optimal size.

In Iceberg, this can be done by

1. Setting write.distribution-mode to sort.
2. Setting a sort-order property
3. Sorting with Spark before writing to the output

> ⚠ Warning
>
> Global sorting is extremely expensive as you shuffle and then sort per partition. Sorting is beneficial for columns with high cardinality.

As of `Iceberg 1.10.0`: "Future work in Spark should allow Iceberg to automatically adjust this (`spark.sql.adaptive.advisoryPartitionSizeInBytes`) parameter at write time to match the `write.target-file-size-bytes`."

Iceberg docs

6 Vendors do this (& more)

In addition to file resizing, when using table formats, we need to clean up deleted data (preserved for history) & manifest files.

Vendors like Snowflake, BigQuery, and Databricks automate these for you.

They also enable you to fine-tune file sizes when needed.

7 Conclusion

To recap, we saw

1. Why are **many small files a problem**
2. Using **maintenance functions** to optimally size files
3. Optimal **file sizing how-tos for partitioned tables**
4. How **vendors (mostly) handle it automatically**

Choose easy or cheap, you can't have it both ways.

If you manage your own data lake, make sure to **handle maintenance** of your data storage layer using the techniques in this post.

Your future self and stakeholders will thank you.

If you found this helpful or learned something new, please share this article in your socials; it helps out a lot.

8 Essential reading

1. What is a table format?
2. How to setup Spark locally
3. Docker for data engineers