How to Identify and Fix Small Files Problem with Spark & Iceberg

Hands-on workshop covering compaction strategies to detect & fix small file issues in Apache Iceberg with Apache Spark.

Joseph Machado

2026-05-06

Table of contents

1 Introduction 1
  1.1 If you don't have a vendor, run the rewrite_data_files function 2
  1.2 Code Setup 2
2 Many small files => Spark wastes time opening them 2
3 The `rewrite_data_files` function is the simplest fix 4
4 Partition data before insert 7
  4.1 Inserting into a partitioned table does not guarantee optimal file size 7
  4.2 Increasing task memory does not guarantee optimal file size 9
  4.3 Increase task memory and partition before insert 12
5 Alternatives & future work 14
6 Vendors do this (& more) 14
7 Conclusion 14
8 Essential reading 14

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?"

By the end of this post, you will know how to detect and fix the problem of small files.

1.1 If you don't have a vendor, run the rewrite_data_files function

If you have a vendor, they take care of fixing the small files problem.

If not, you can use the Iceberg function **rewrite_data_files** in your pipeline or as a scheduled manitenance pipeline.

1.2 Code Setup

Follow along with code using this **setup instructions**.

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

2 Many small files => Spark wastes time opening them

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.

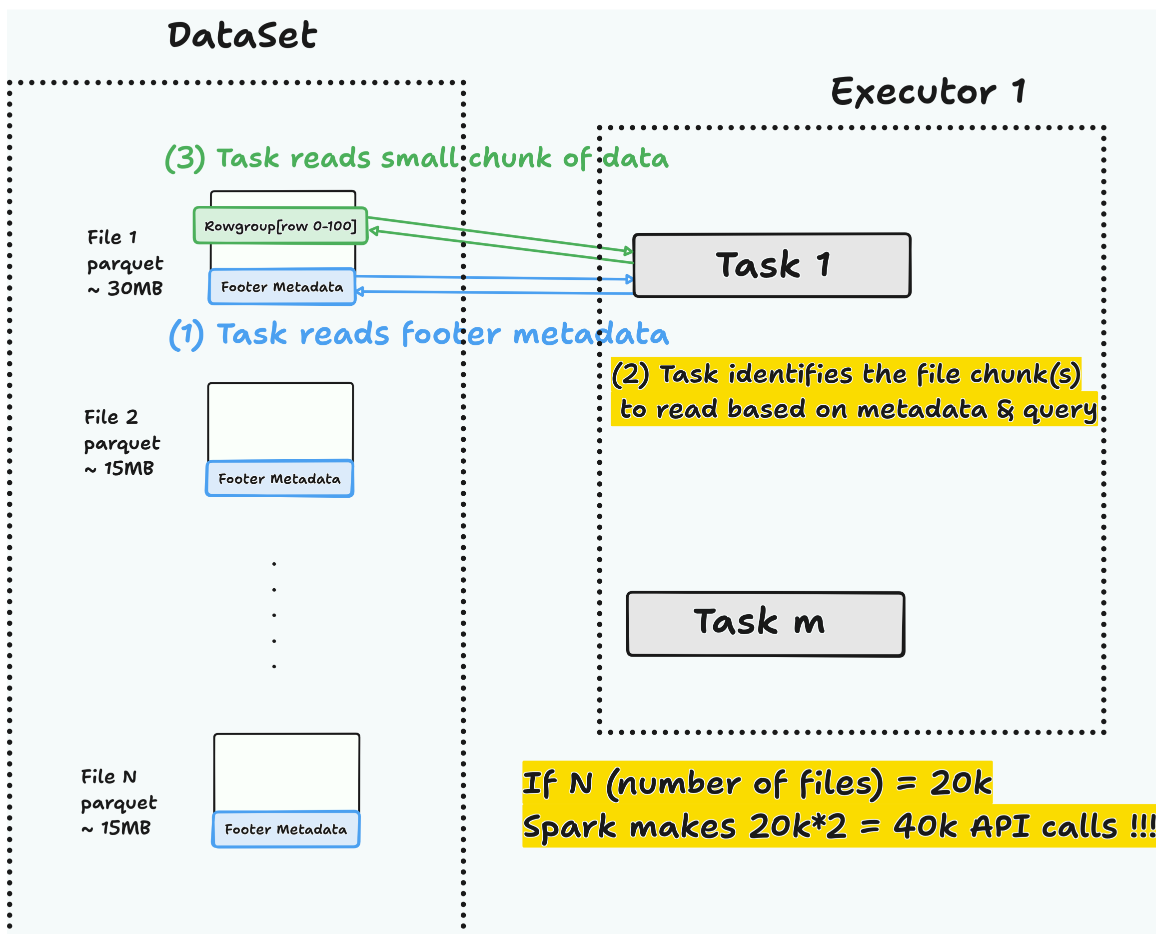


Figure 1: Small Files I/O Problem

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 stage tab from sql/dataframe tab

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

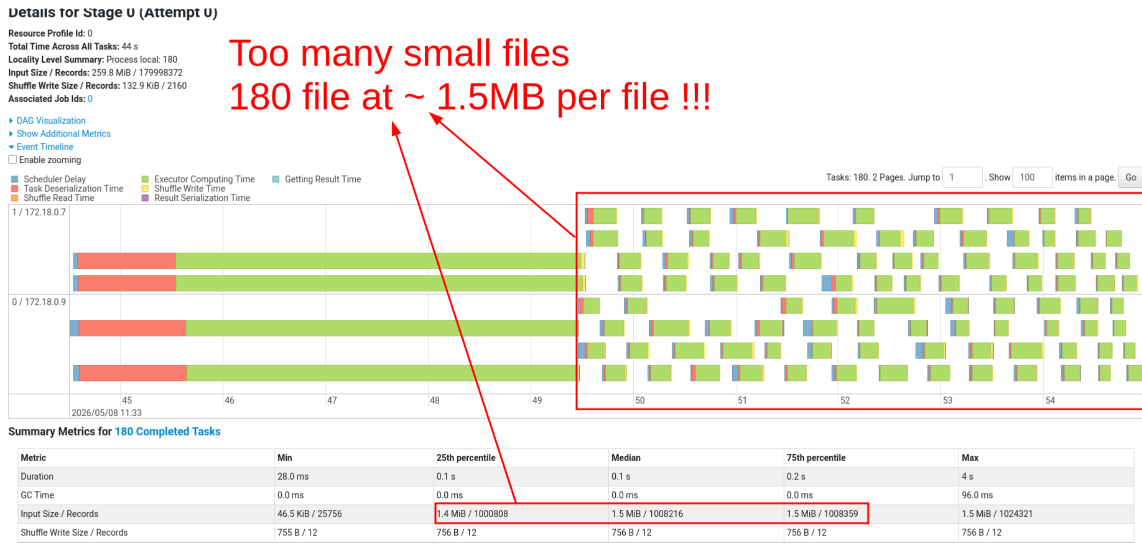


Figure 3: Many small green chunks = many small files

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

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

3 The `rewrite_data_files` 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 see how `rewrite_data_files` works.

```
%%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 size: `512 MB`).

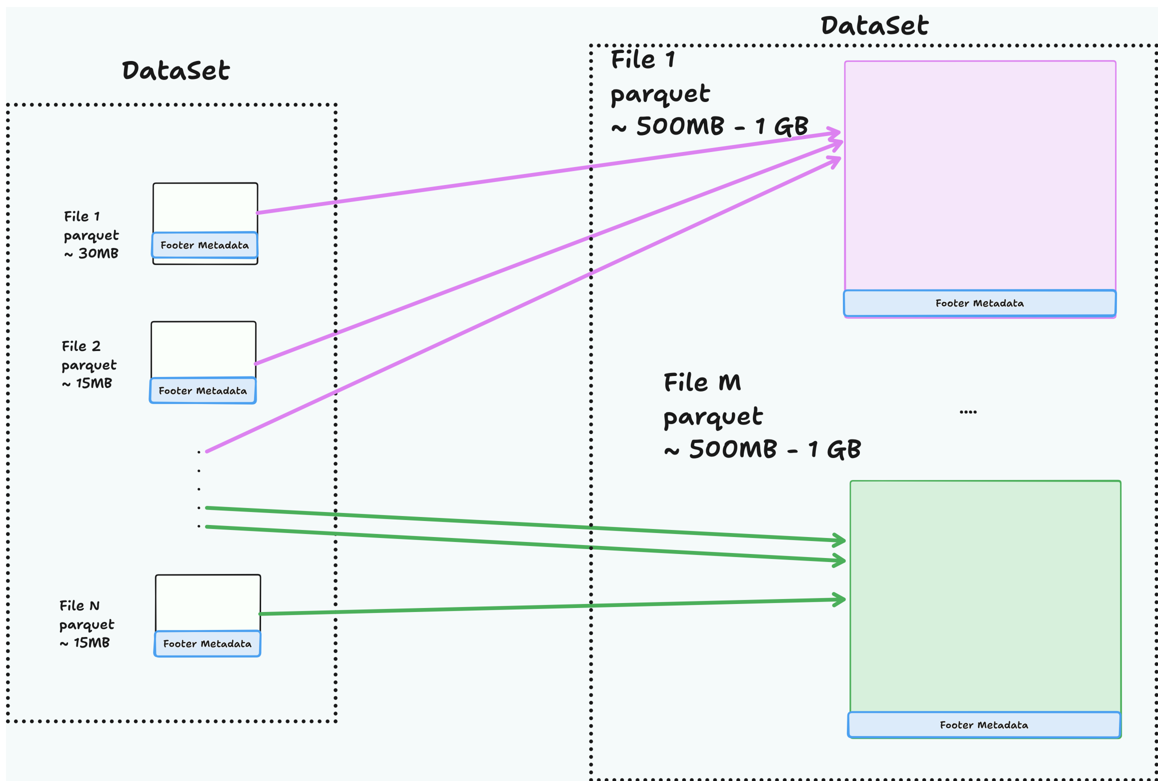


Figure 4: Combining small files to 512MB files

Now Spark can concentrate on processing the data.

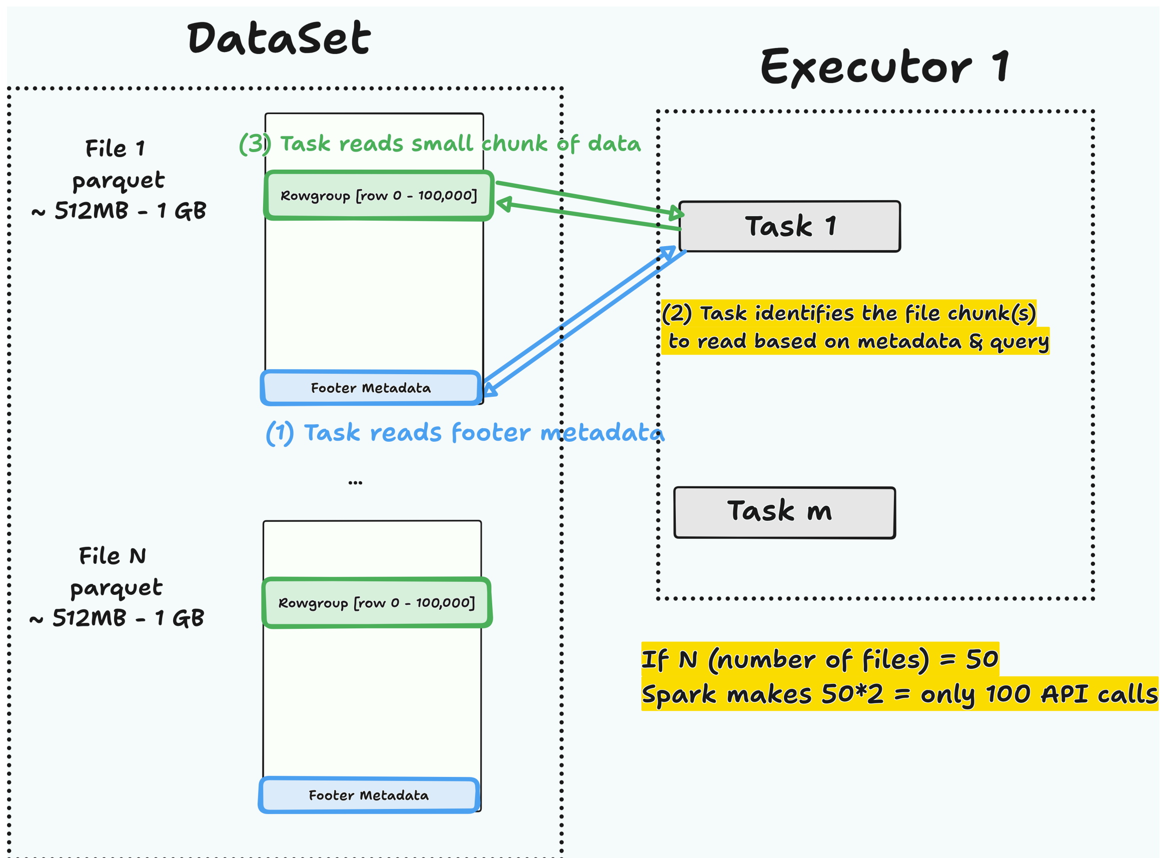


Figure 5: Optimal Files I/O

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
```

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.

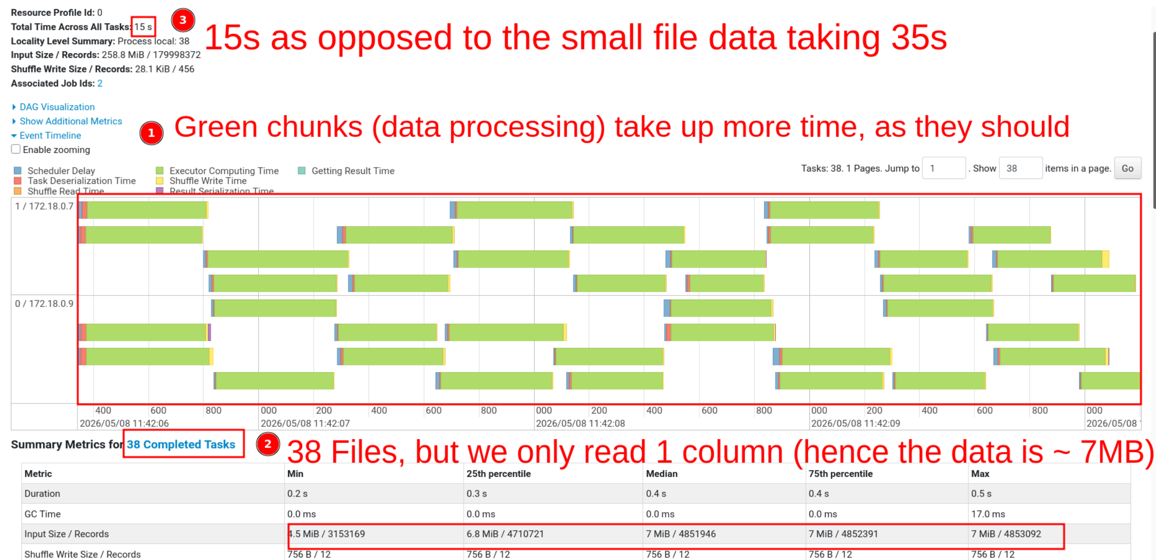


Figure 6: Larger green chunks = Spark processing data

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 inserted only in the prior day.

4 Partition data before insert

4.1 Inserting into a 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 (ref)of memory before writing data to the output file. This is set with the property `spark.sql.iceberg.advisory-partition-size`.

Writing to parquet results in 4x compression (data-dependent), so 384 MB in-memory becomes a 100 MB file.

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

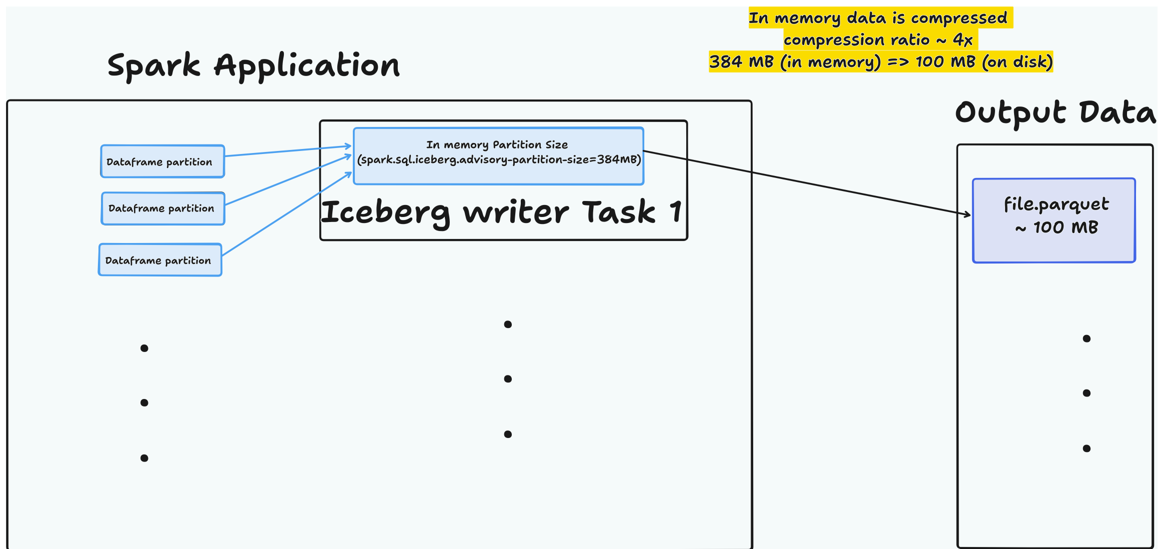


Figure 7: Iceberg writer in-memory to file compression

Check out the input size (~350 MB) to output size (~100 MB) in the task metrics section of 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: Task metrics: in-memory to file compression

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

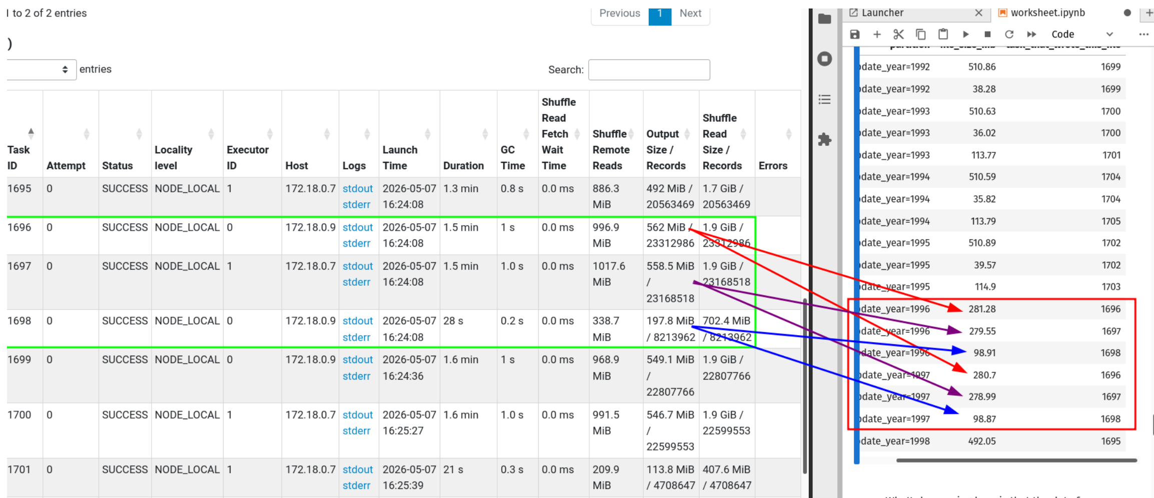
Search:

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 some non-optimal files in the stages tab.



Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Read Fetch Wait Time	Shuffle Remote Reads	Output Size / Records	Shuffle Read Size / Records	Errors
1695	0	SUCCESS	NODE_LOCAL	1	172.18.0.7	stdout stderr	2026-05-07 16:24:08	1.3 min	0.8 s	0.0 ms	886.3 MiB	492 MiB / 20563469	1.7 GiB / 20563469	
1696	0	SUCCESS	NODE_LOCAL	0	172.18.0.9	stdout stderr	2026-05-07 16:24:08	1.5 min	1 s	0.0 ms	996.9 MiB	562 MiB / 23312986	1.9 GiB / 23312986	
1697	0	SUCCESS	NODE_LOCAL	1	172.18.0.7	stdout stderr	2026-05-07 16:24:08	1.5 min	1.0 s	0.0 ms	1017.6 MiB	558.5 MiB / 23168518	1.9 GiB / 23168518	
1698	0	SUCCESS	NODE_LOCAL	0	172.18.0.9	stdout stderr	2026-05-07 16:24:08	28 s	0.2 s	0.0 ms	338.7 MiB	197.8 MiB / 8213962	702.4 MiB / 8213962	
1699	0	SUCCESS	NODE_LOCAL	0	172.18.0.9	stdout stderr	2026-05-07 16:24:36	1.6 min	1 s	0.0 ms	968.9 MiB	549.1 MiB / 22807766	1.9 GiB / 22807766	
1700	0	SUCCESS	NODE_LOCAL	1	172.18.0.7	stdout stderr	2026-05-07 16:25:27	1.6 min	1.0 s	0.0 ms	991.5 MiB	546.7 MiB / 22599553	1.9 GiB / 22599553	
1701	0	SUCCESS	NODE_LOCAL	1	172.18.0.7	stdout stderr	2026-05-07 16:25:39	21 s	0.3 s	0.0 ms	209.9 MiB	113.8 MiB / 4708647	407.6 MiB / 4708647	

partition		
date_year=1992	510.86	1699
date_year=1992	38.28	1699
date_year=1993	510.63	1700
date_year=1993	36.02	1700
date_year=1993	113.77	1701
date_year=1994	510.59	1704
date_year=1994	35.82	1704
date_year=1994	113.79	1705
date_year=1995	510.89	1702
date_year=1995	39.57	1702
date_year=1995	114.9	1703
date_year=1996	281.28	1696
date_year=1996	279.55	1697
date_year=1996	98.91	1698
date_year=1997	280.7	1696
date_year=1997	278.99	1697
date_year=1997	98.87	1698
date_year=1998	492.05	1695

Figure 10: Iceberg writers handling data from mutliple partitions

Here is what's happening.

Rows with different `year(l_shipdate)` are handled by the same writer, which gets written to separate files, leading to less than optimal file sizes.

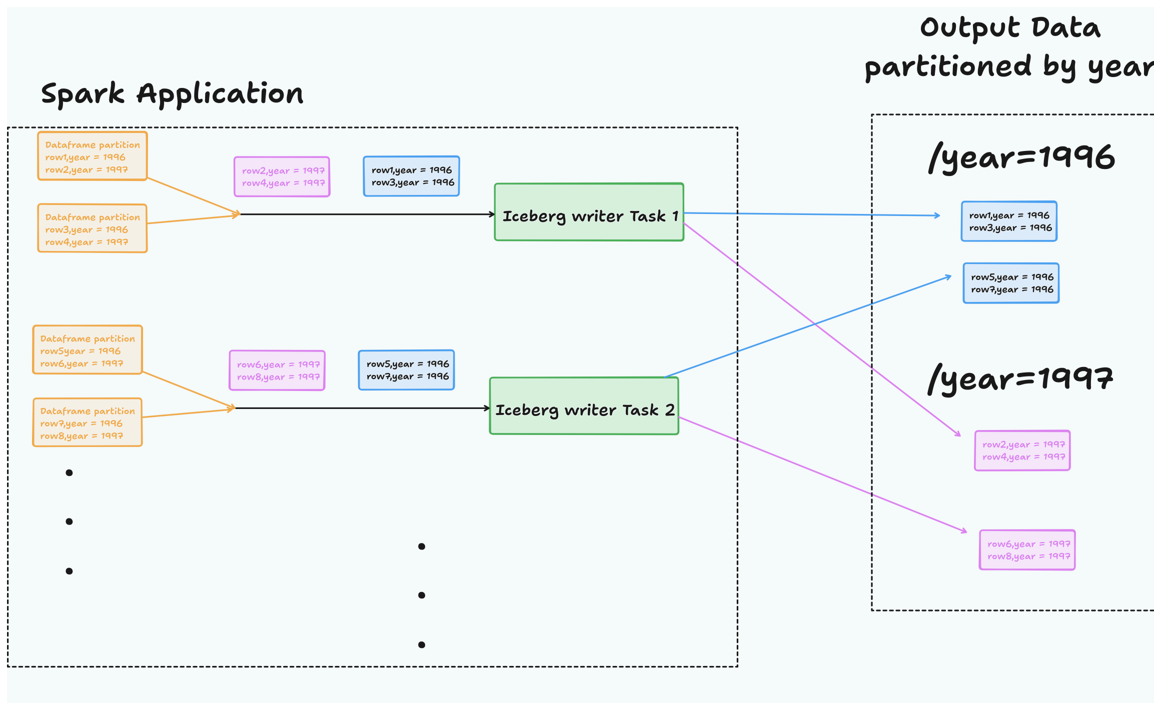


Figure 11: Iceberg writes writing to multiple partitions

4.3 Increase task memory and partition before insert

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

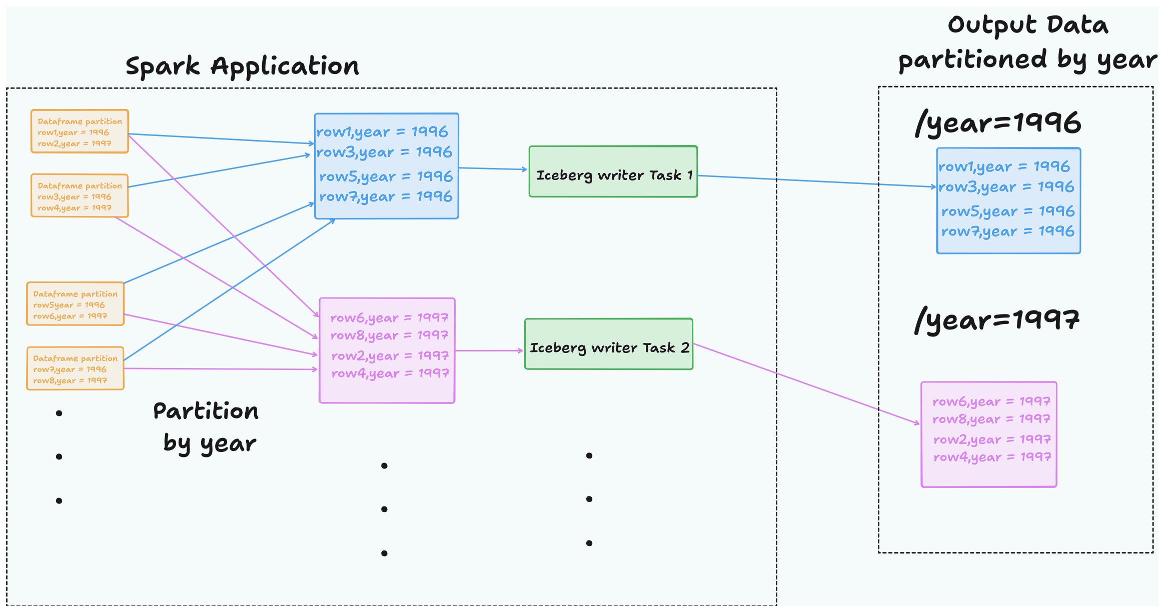


Figure 12: Partition & then write

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.

Tasks (7)

Show 40 entries Search:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time	Shuffle Read Fetch Wait Time	Shuffle Remote Reads	Output Size / Records	Shuf Read Size Reco
0	1895	0	SUCCESS	NODE_LOCAL	0	172.18.0.9	stdout stderr	2026-05-07 16:32:38	1.4 min	0.8 s	0.0 ms	0.0 B	492.1 MiB / 20563469	1.7 G 2056
1	1896	0	SUCCESS	NODE_LOCAL	1	172.18.0.7	stdout stderr	2026-05-07 16:32:38	1.8 min	1 s	0.0 ms	0.0 B	657.3 MiB / 27326595	2.3 G 2732
2	1897	0	SUCCESS	NODE_LOCAL	0	172.18.0.9	stdout stderr	2026-05-07 16:32:38	2.1 min	1 s	9 s	2.3 GiB	658.4 MiB / 27368871	2.3 G 2736
4	1898	0	SUCCESS	NODE_LOCAL	1	172.18.0.7	stdout stderr	2026-05-07 16:32:38	1.8 min	1 s	0.0 ms	0.0 B	660.5 MiB / 27308200	2.3 G 2730
3	1899	0	SUCCESS	NODE_LOCAL	0	172.18.0.9	stdout stderr	2026-05-07 16:33:59	1.6 min	1.0 s	0.0 ms	0.0 B	549.1 MiB / 22807766	1.9 G 2280
5	1900	0	SUCCESS	ANY	1	172.18.0.7	stdout stderr	2026-05-07 16:34:28	1.9 min	1 s	15 s	2.3 GiB	665.3 MiB / 27321392	2.3 G 2732
6	1901	0	SUCCESS	ANY	1	172.18.0.7	stdout stderr	2026-05-07 16:34:29	2.0 min	1 s	26 s	2.3 GiB	660.3 MiB / 27302079	2.3 G 2730

File Edit View Run Kernel Tabs Settings Help

Launcher worksheet.ipynb

[42]:

partition	file_size_mb	task_that_wrote_this_file
ear=1998	492.13	1895
ear=1997	510.7	1896
ear=1997	146.56	1896
ear=1996	510.69	1897
ear=1996	147.72	1897
ear=1993	510.63	1898
ear=1993	149.88	1898
ear=1992	510.81	1899
ear=1992	38.25	1899
ear=1995	510.88	1900
ear=1995	154.45	1900
ear=1994	510.62	1901
ear=1994	149.68	1901

```
![Iceberg Task Write w Spark Partition](iceberg_task_write_w_spark_partition.png)
```

Figure 13: One partition data per iceberg writer with rollover

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

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 Databricks automate these for you. And vendors like Snowflake have their own representation.

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 (yet). If you have a vendor, they take care of fixing the small files problem.

If not, you can use the Iceberg function **rewrite_data_files** in your pipeline or as a scheduled manitenance pipeline.

Your future self and stakeholders will thank you.

If you found this helpful or learned something new, please share this article on your socials; it really helps.

8 Essential reading

1. What is a table format?
2. How to setup Spark locally

3. Docker for data engineers
4. Parquet Format
5. Databricks Unity Catalog