Optimizing Apache Spark

The Five Most Common Performance Problems

Storage

The 5 Most Common Performance Problems (The 5 Ss)
Storage

- Storage is our 4th problem area
- Traditionally it consisted of one specific type of problem (aka Tiny Files)
- But it is actually a class of problems that relates to high overhead with ingesting data
- That ingest can be the initial ingest from Blob Storage, JDBC, Kafka, EventHubs or even from a previous Spark stage

The 5 Most Common Performance Problems (The 5 Ss)

Storage - More Examples

We are going to take a look at a couple of examples:

- Tiny Files
- Scanning
- Schemas, Merging Schemas & Schema Evolution

If you had only 15 seconds to pick up as many coins as you can, one coin at a time, which pile do you want to work from?

$0.13 vs $3.25

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Tiny Files In Action

See Experiment #8923, contrast **Step B**, **StepC** and **Step D**

- Note the total execution time of each job
- In the **Spark UI**, see the **Stage Details** for the last stage of each step and note the **Input Size / Records**
- In the **Spark UI**, see the **Query Details** for the last job of each step and note the...
  - **number of files read**
  - **scan time total**
  - **filesystem read time total**
  - **size of files read**

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Tiny Files, Review

Step	Record Count	Execution Time	number of files read	scan time total	filesystem read time total	size of files read
B - Benchmark #1	~41 M	~3 minutes	6,273	20 minutes	~10 minutes	1,209 MB
C - Benchmark #2	~2.7 B	~10 minutes	100	1 hour	~1 hour	102 GB
D - Tiny Files	~34 M	~1.5 hours	345,612	12 hours	> 6 hours	2.1 GB

What can we do to mitigate the impact of tiny files?

The 5 Most Common Performance Problems (The 5 Ss)

Storage - Only 2.5 Options?

I might be wrong, but I think there are really only two options...

Scenario #1

- You caused the problem...
  - You can fix the problem

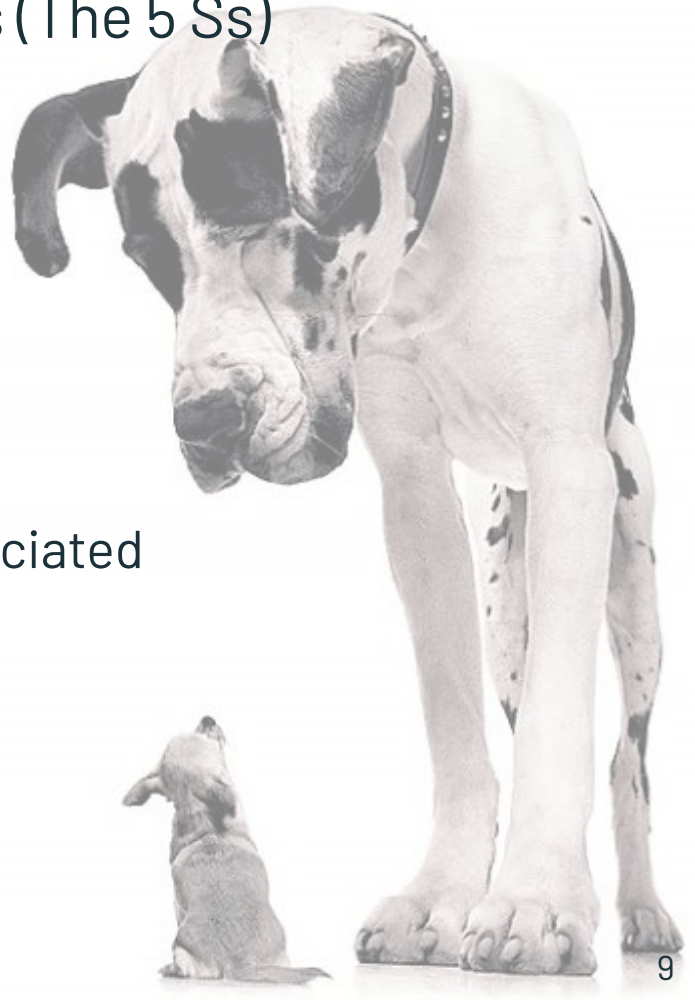
Scenario #2

- Someone else caused the problem...
  - Push back on design
  - Just live with it

The 5 Most Common Performance Problems (The 5 Ss)

Storage - The Ideal File Size

- The ideal part-file is between 128MB and 1GB
- Smaller than 128MB and we creep into the Tiny Files problem & its cousins
- Larger than 1GB part-files are general advised against mainly due in part to the problems associated with creating these large Spark-Partitions
- Remember...
  1 Spark-Partition == 1 Part-File upon write



The 5 Most Common Performance Problems (The 5 Ss)
Storage - Manual Compaction

- You can control the on-disk, part-file size
- The process does require some guessing
- Other “features” like partitioning and [especially] bucketing further complicates this process
- The key to it all is the notion that one Spark-task writes one part-file meaning that a 1GB part-files requires a 1GB Spark-partition

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Manual Compaction, How-To

The Algorithm

1. Determine the **size of your dataset on disk**
2. Decide what your ideal **part-file size** is
3. Compute the **number of spark-partitions** required (divide **size-on-disk** / **ideal-size**)
4. Configure a cluster with N cores *(more cores == less time)*
5. Read in your data, repartition by **N**, and then write to disk
6. Check the Spark UI for spill and any other issues

An Example In Action

1. Size on disk is **150 GB**
2. Assume **½ GB** part-files
3. **150 GB** / **½ GB** = **300 partitions**
4. 9 x **C4.8xlarge** (60 GB, 36 cores)
   - 9 VMs x 36 cores for 324 total cores
   - 60 GB / 2 = 30 GB execution *(default is 60% but 50% is safe)*
   - 30 GB /36 cores = 0.83 GB *(over our **½ GB** goal, but disk vs RAM)*
5. Read in your data, repartition by **300**, and then write to disk

How?

Homework: See how to manually compact tiny files in Experiment #2586

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Automatic Compaction

Databricks Delta's Optimize Operation

- See Optimize (Delta Lake on Databricks) for more information
- Targets a 1GB size for each part-file

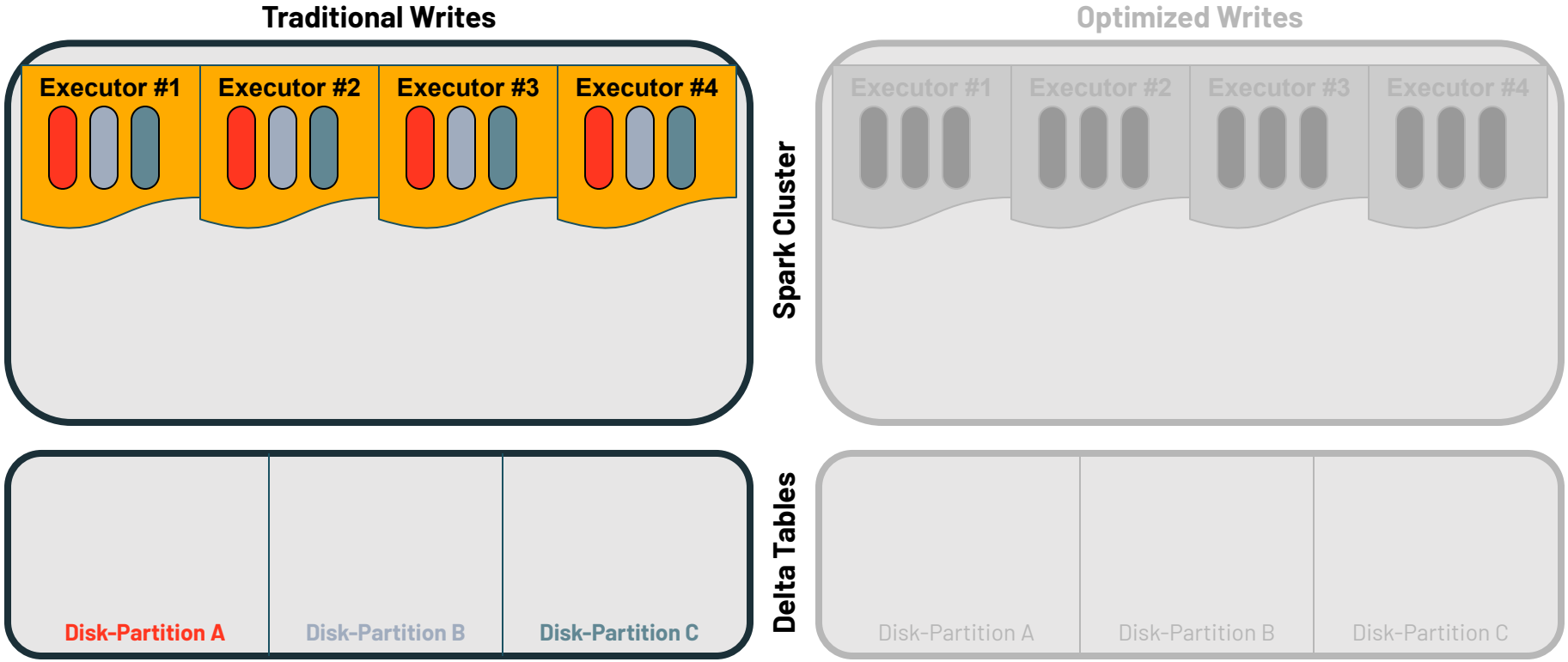
Databricks' Auto-Optimization Feature

- See Auto Optimize for more information
- Targets a 128MB size for each part-file
- Note the most optimal, but better than "tiny files"
- Enable these two options when on Databricks
  1. **spark.databricks.delta.optimizeWrite.enabled = true**
  2. **spark.databricks.delta.autoCompact.enabled = true**

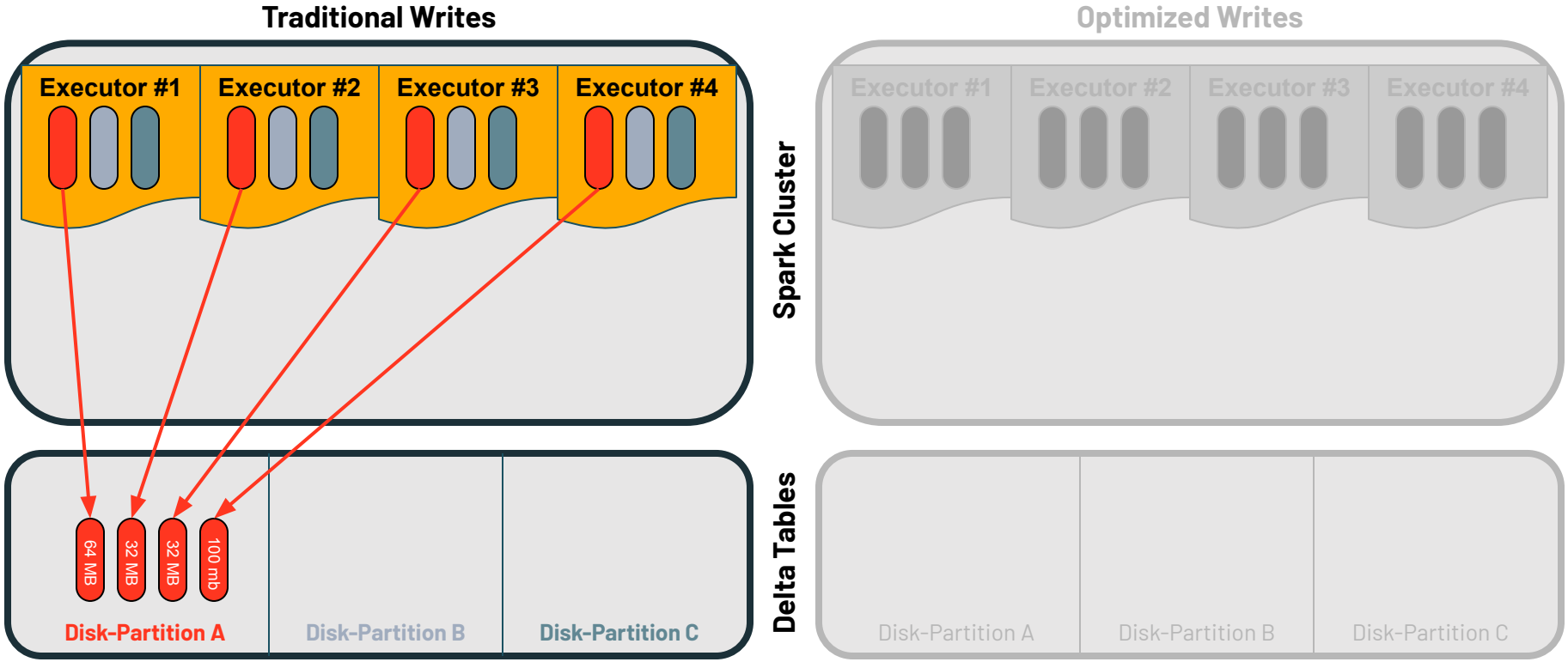
The 5 Most Common Performance Problems (The 5 Ss)
Storage - Why Auto Optimize?

- Manually compacting files, or writing them out correctly the first time, is the most efficient process
- But the Delta optimize and auto-automize can afford the ability to focus on [potentially] bigger problems
- To better understand this, let's take a look at an example of how automatic optimization works
- It's a little bit academic, but it helps to underscore how important this concept is

Storage - Traditional Writes

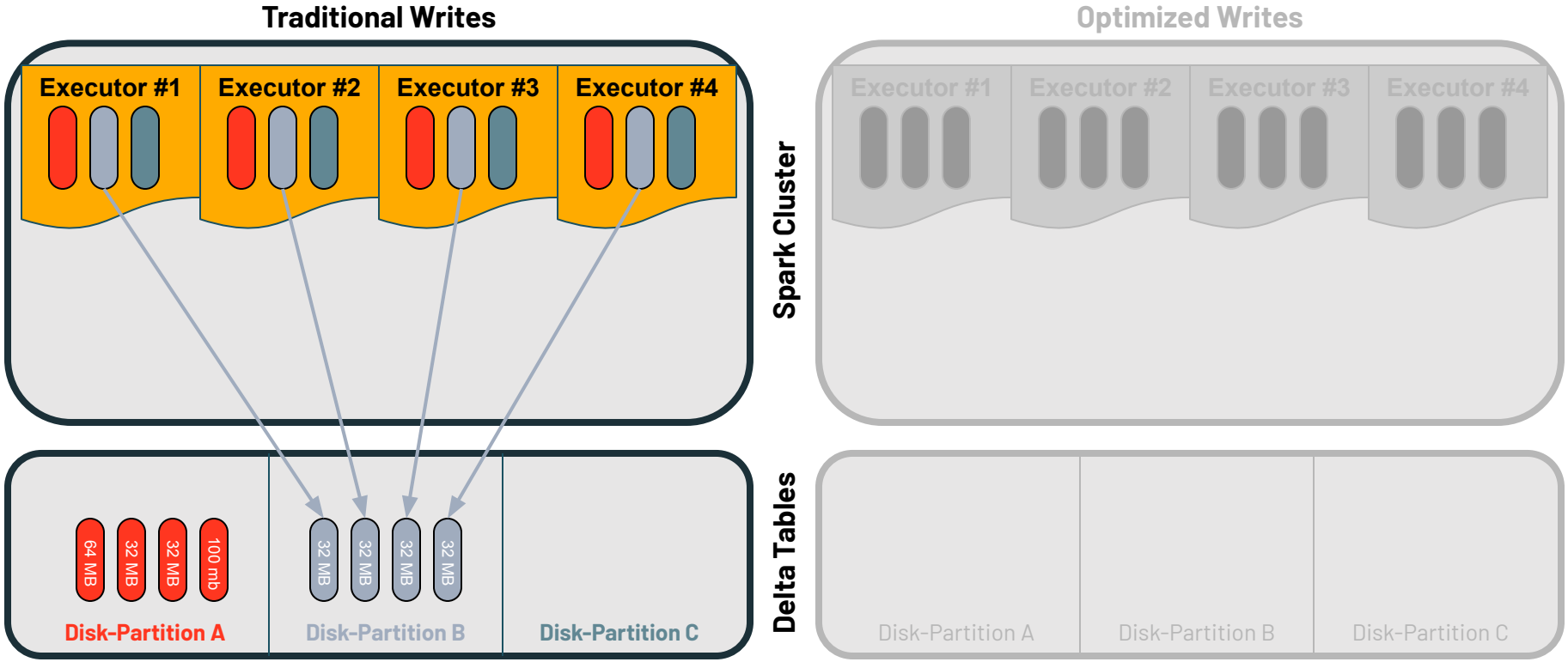


Storage - Traditional Writes



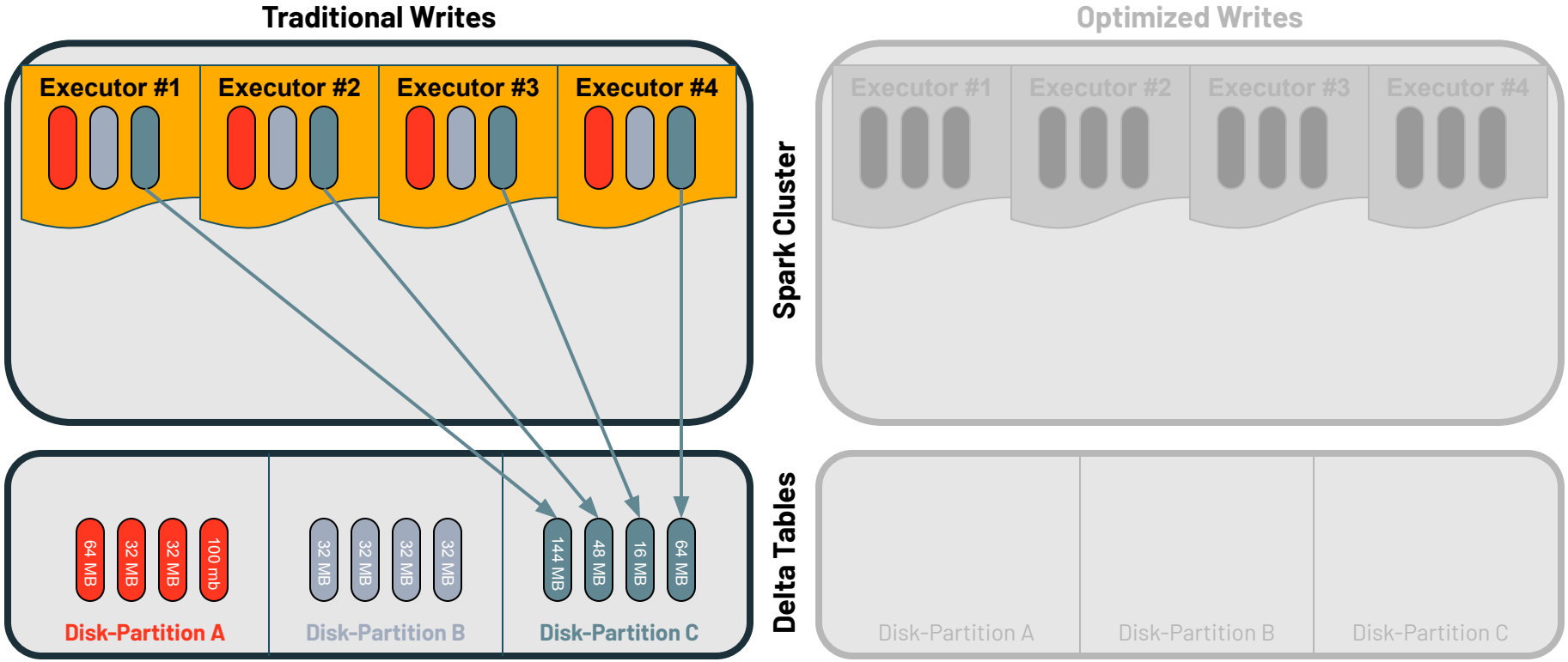
Each task will write one part file to the target disk-partition

Storage - Traditional Writes



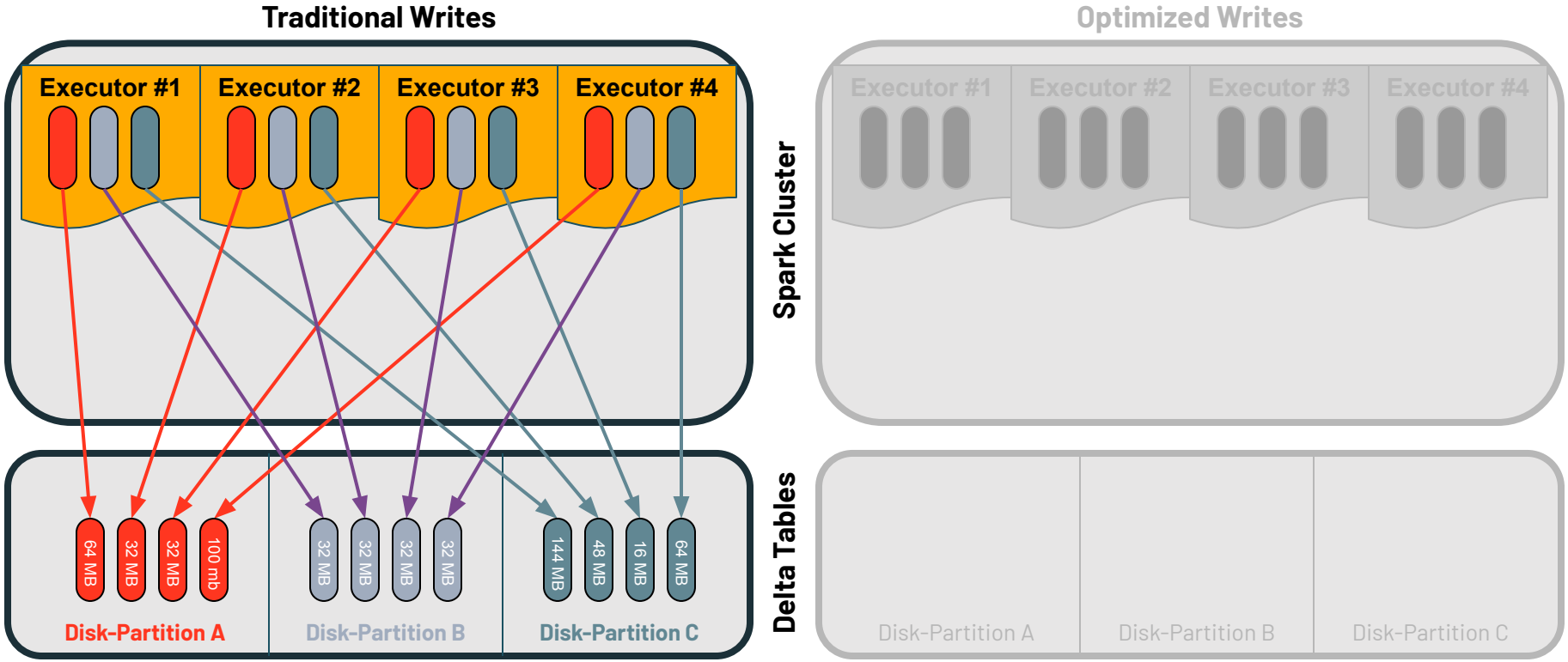
All writes take place at the same time...

Storage - Traditional Writes



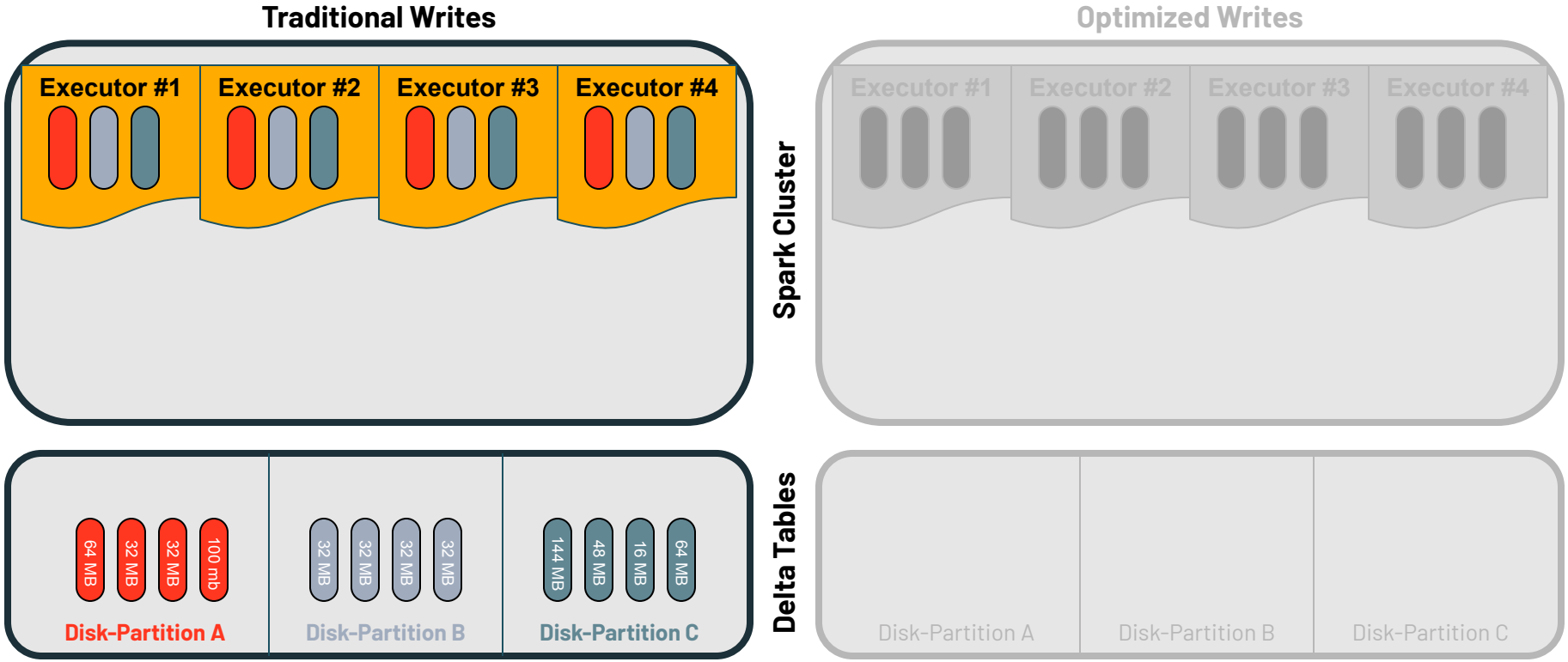
All writes take place at the same time...

Storage - Traditional Writes



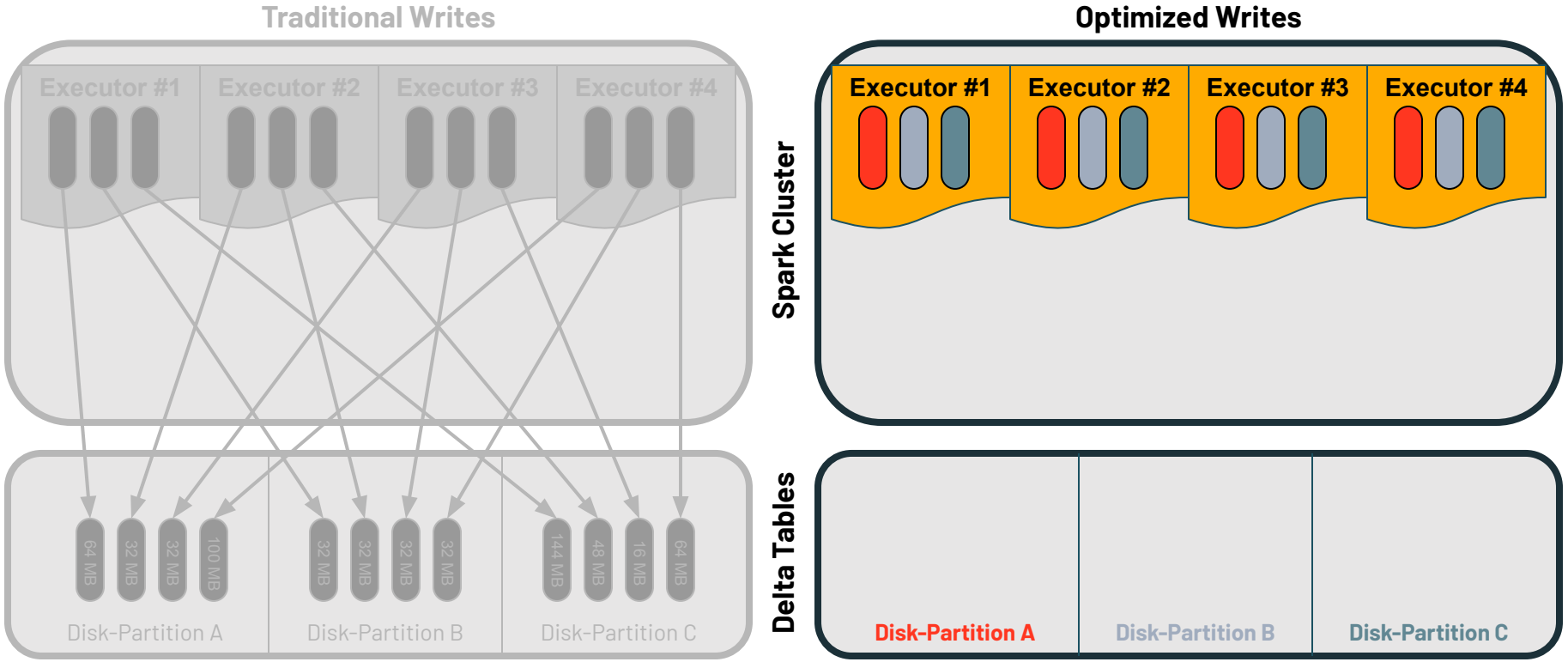
All writes take place at the same time...

Storage - Traditional Writes

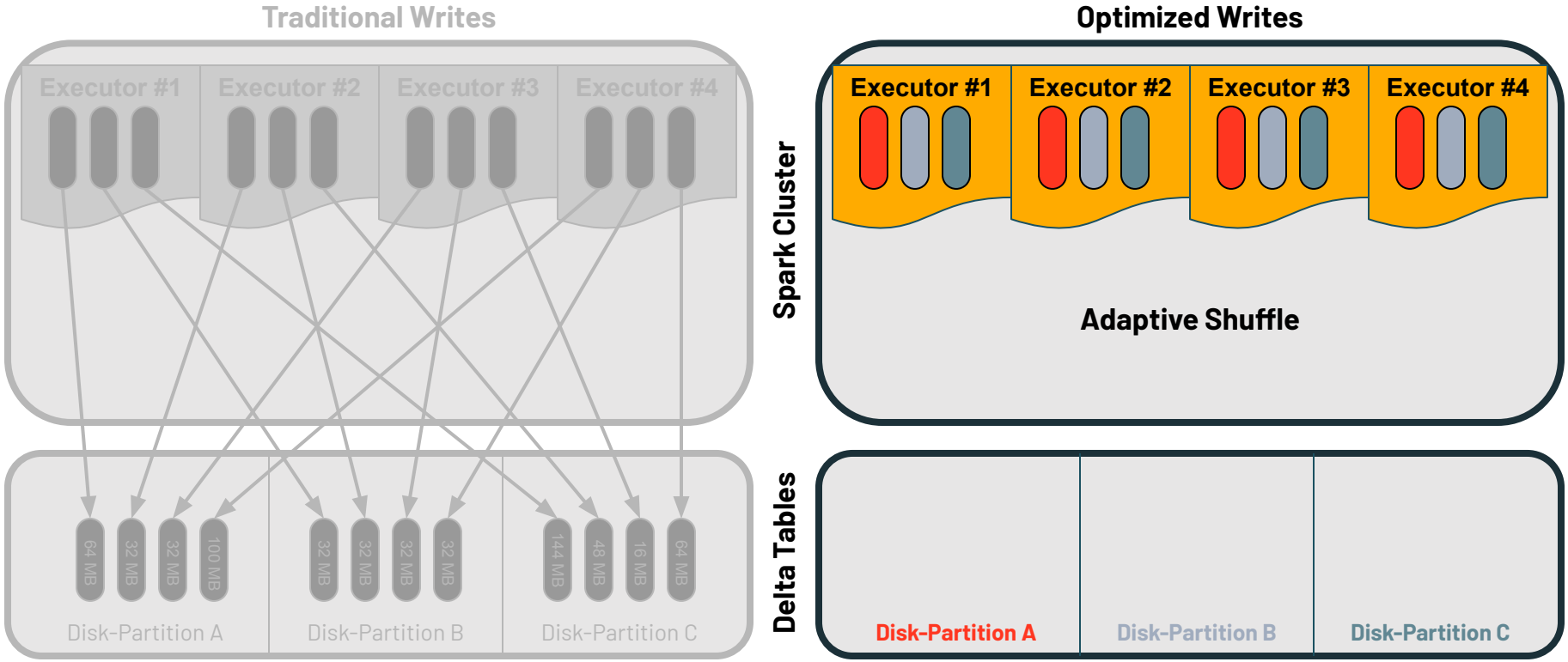


No optimization of the file size, potentially inducing the tiny-files problem

Storage - Optimized Writes

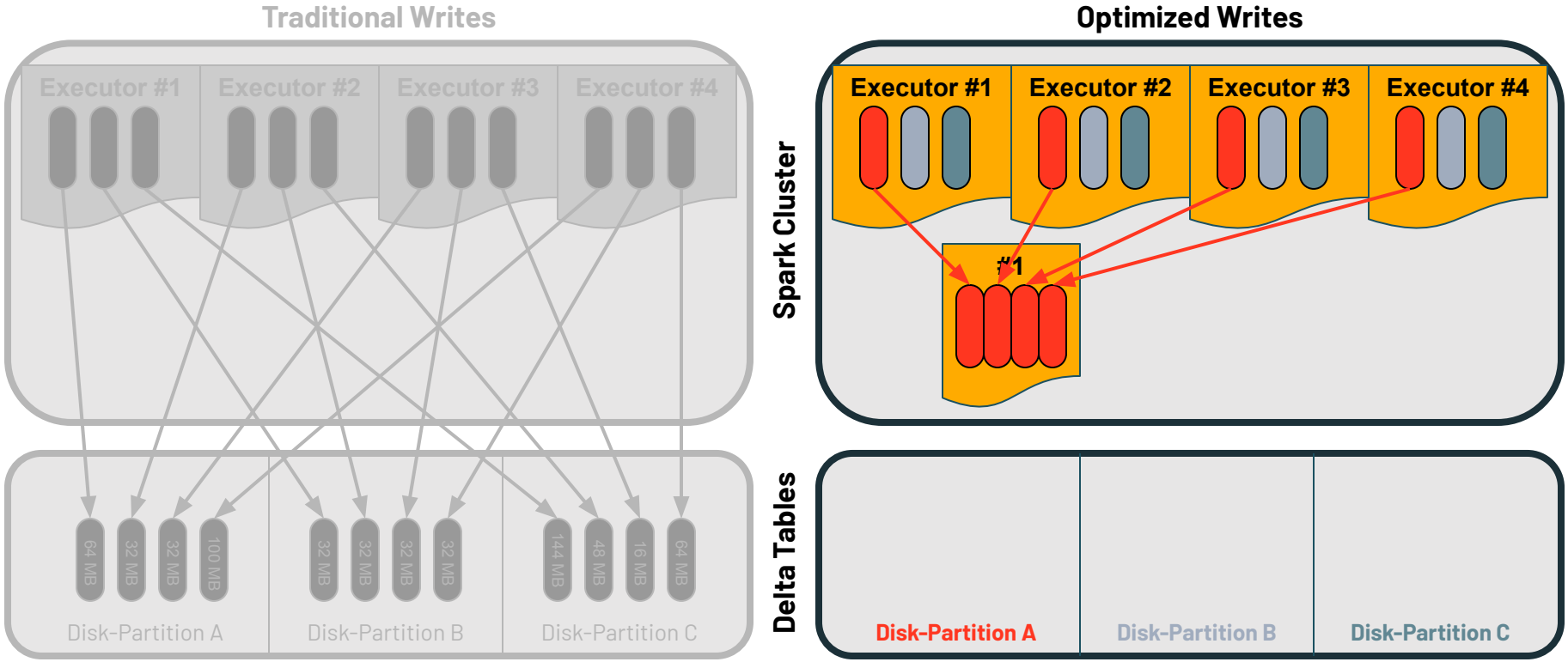


Storage - Optimized Writes



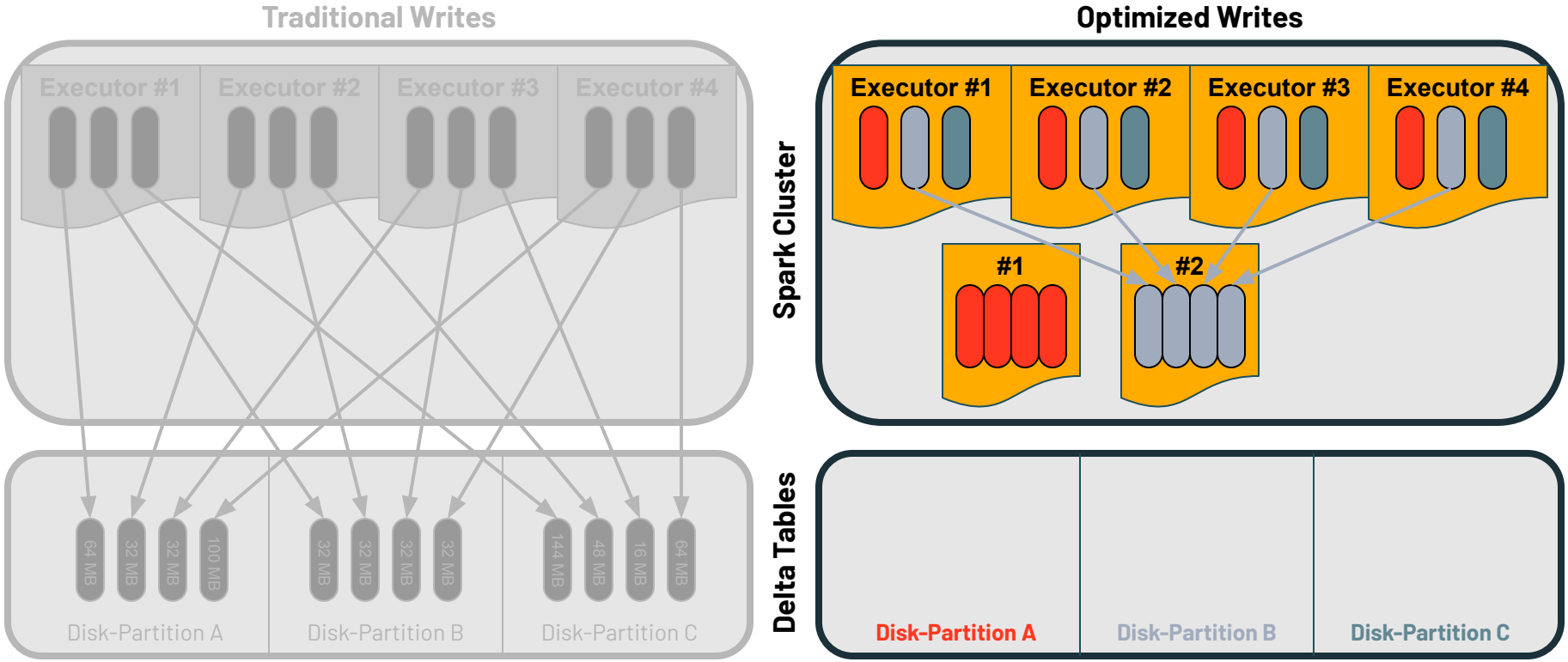
Engage in an adaptive shuffle which is offset by reduced disk IO

Storage - Optimized Writes



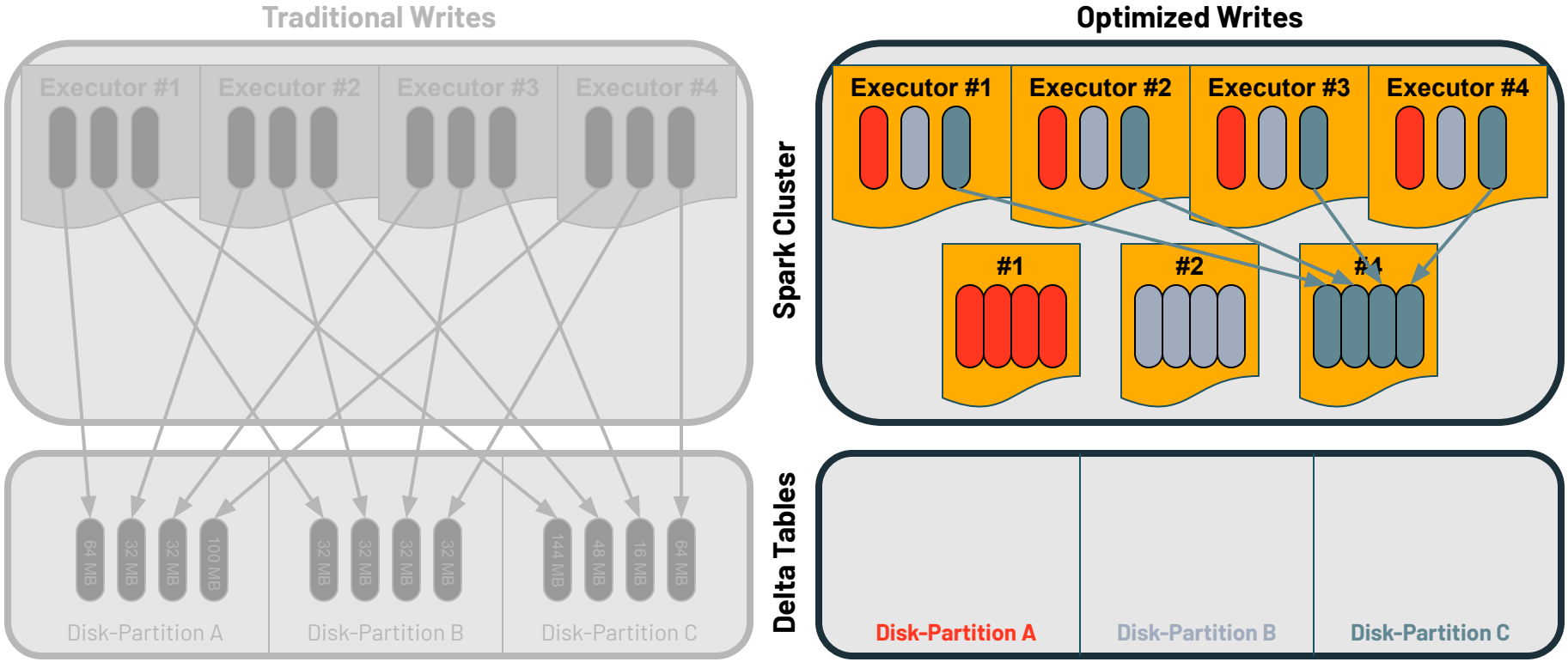
Based on the target disk-partition,
spark-partitions are grouped together

Storage - Optimized Writes



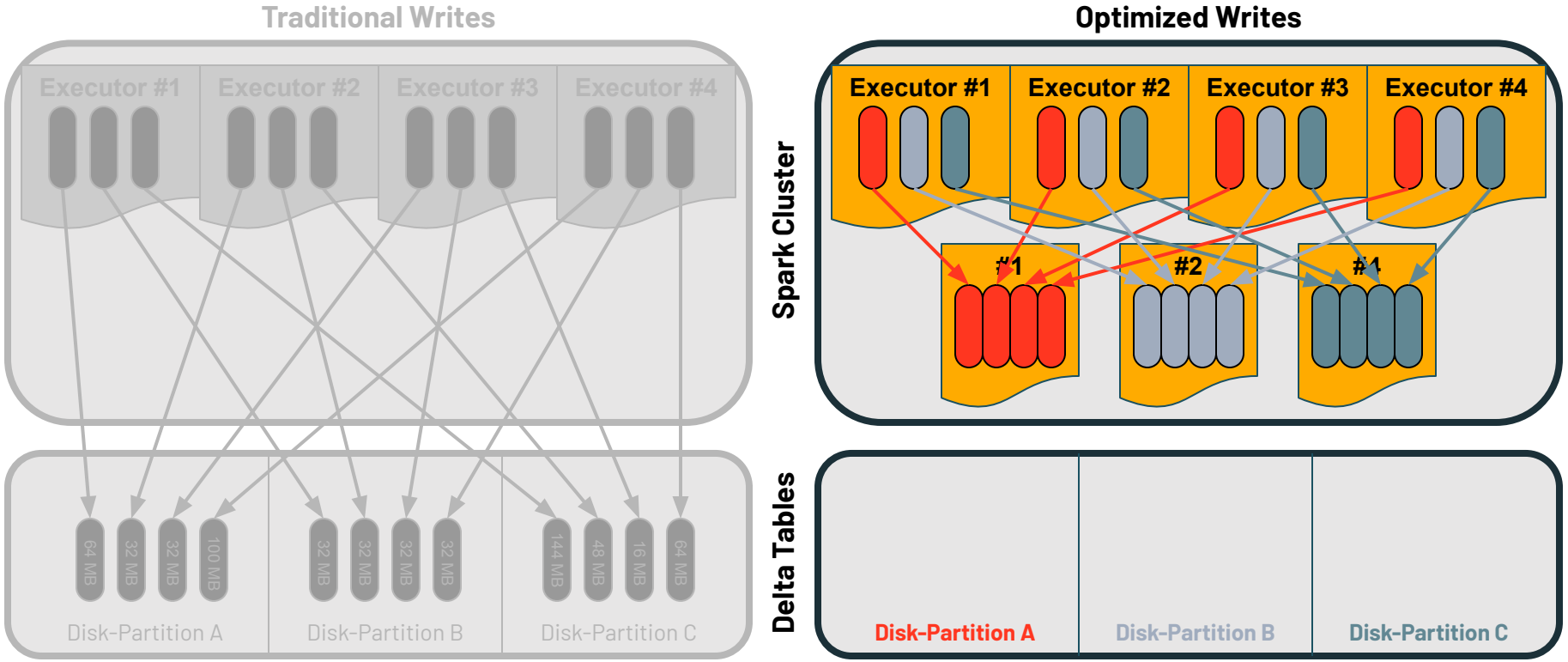
Based on the target disk-partition,
spark-partitions are grouped together

Storage - Optimized Writes



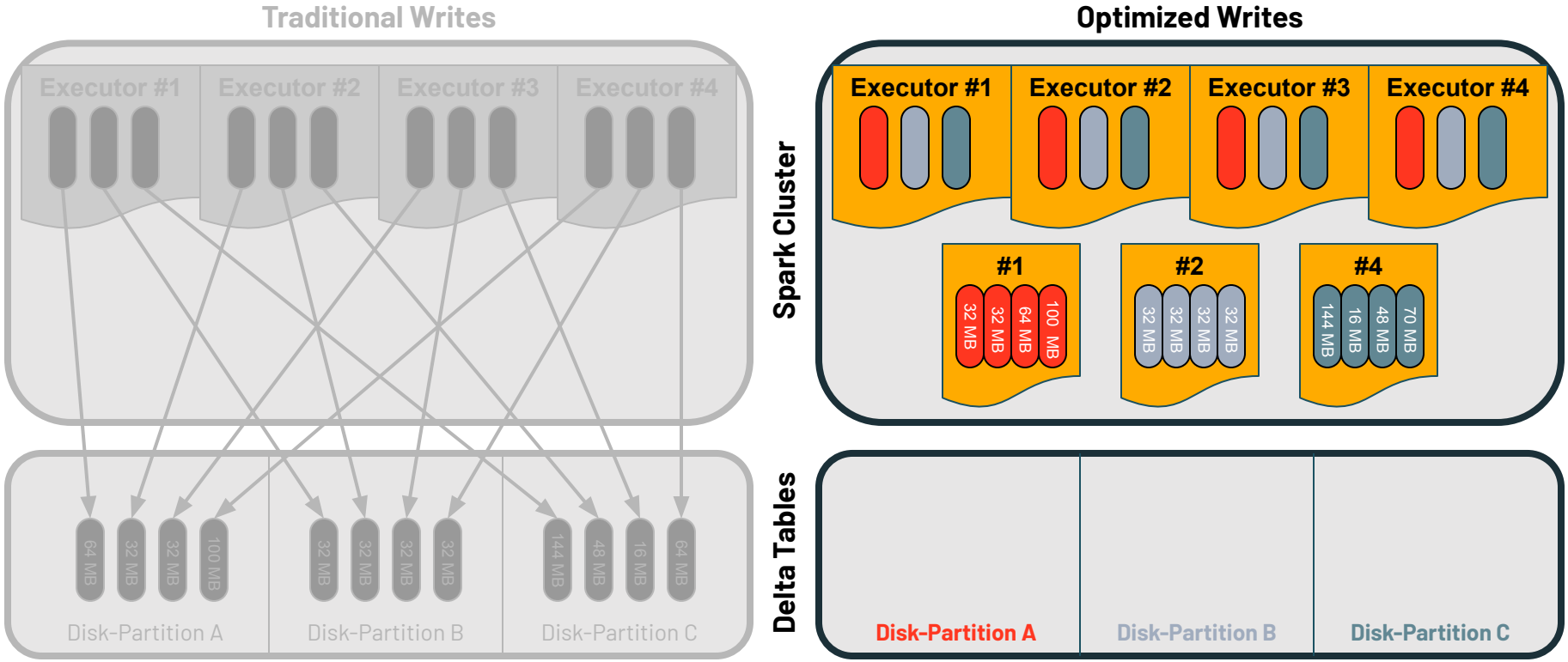
Based on the target disk-partition, spark-partitions are grouped together

Storage - Optimized Writes



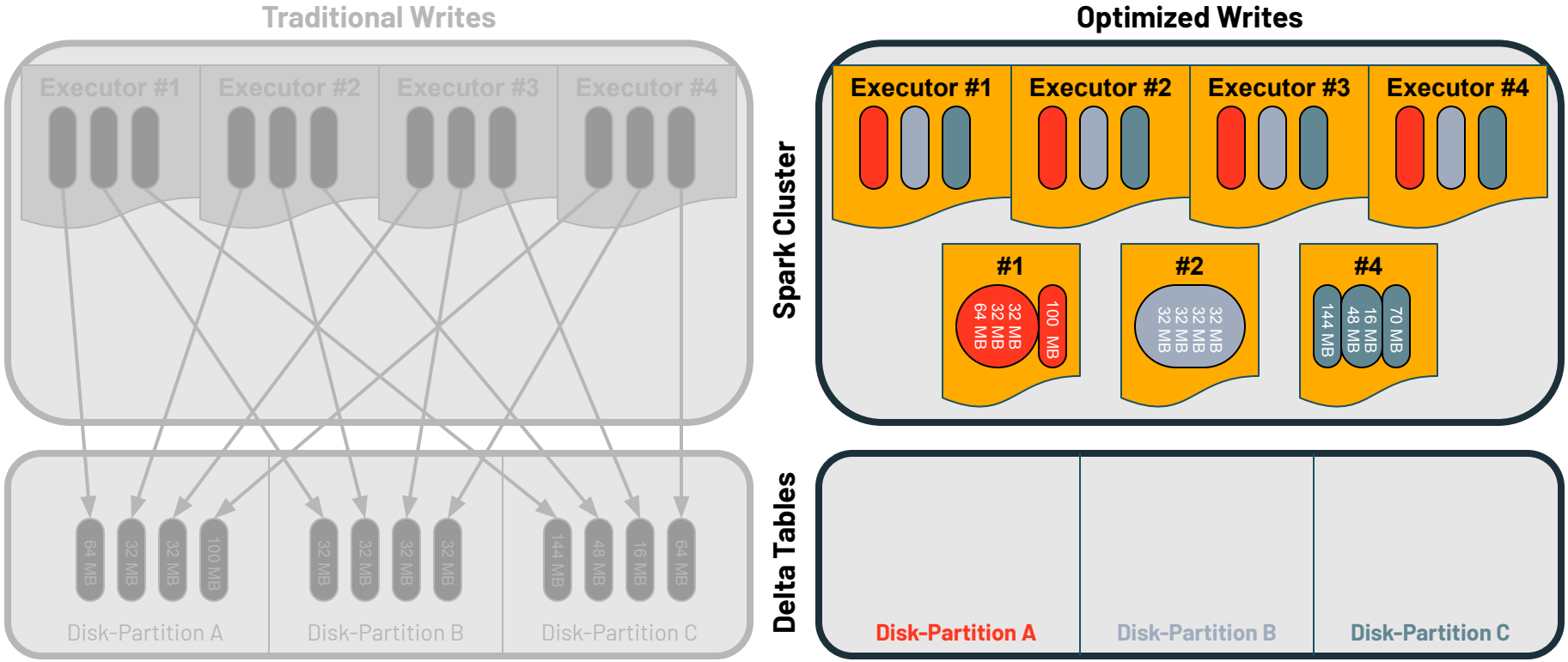
All partitions shuffled at the same time...

Storage - Optimized Writes



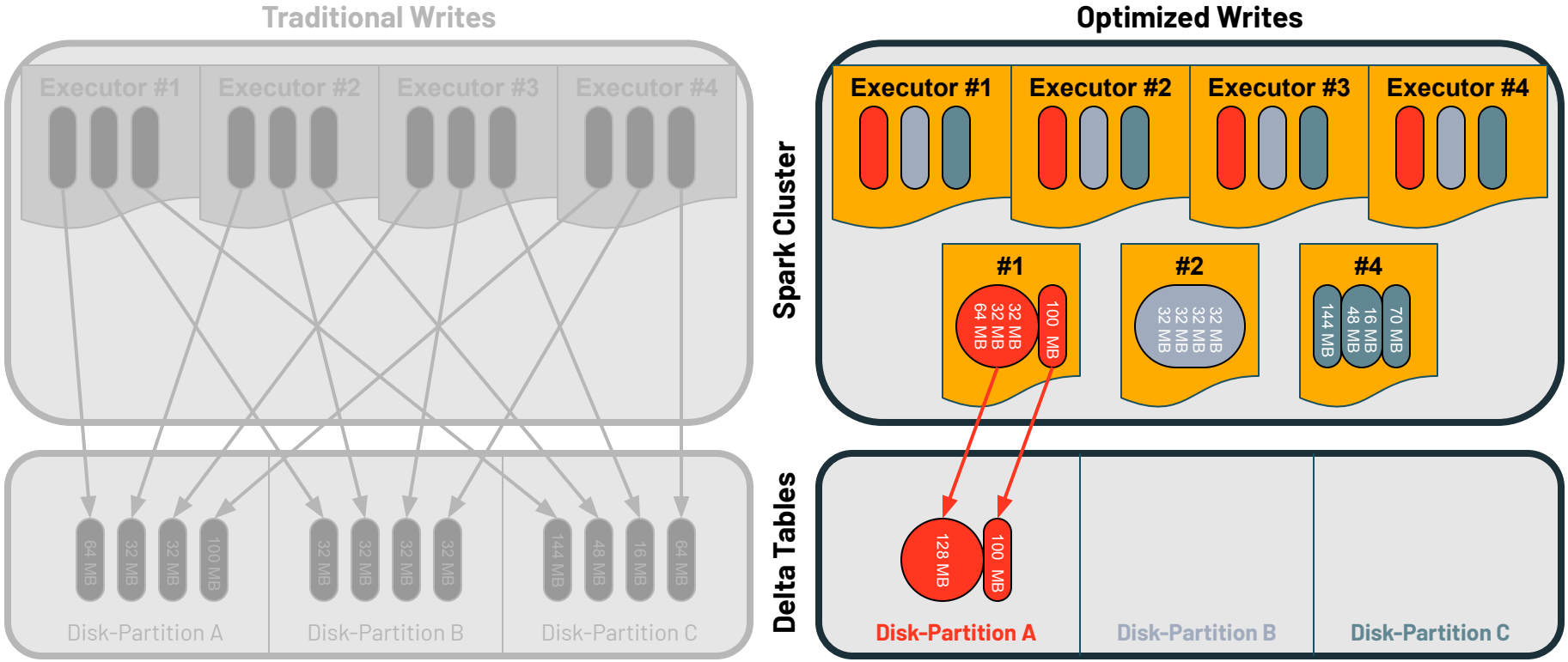
Estimate the final size of each partition on disk

Storage - Optimized Writes



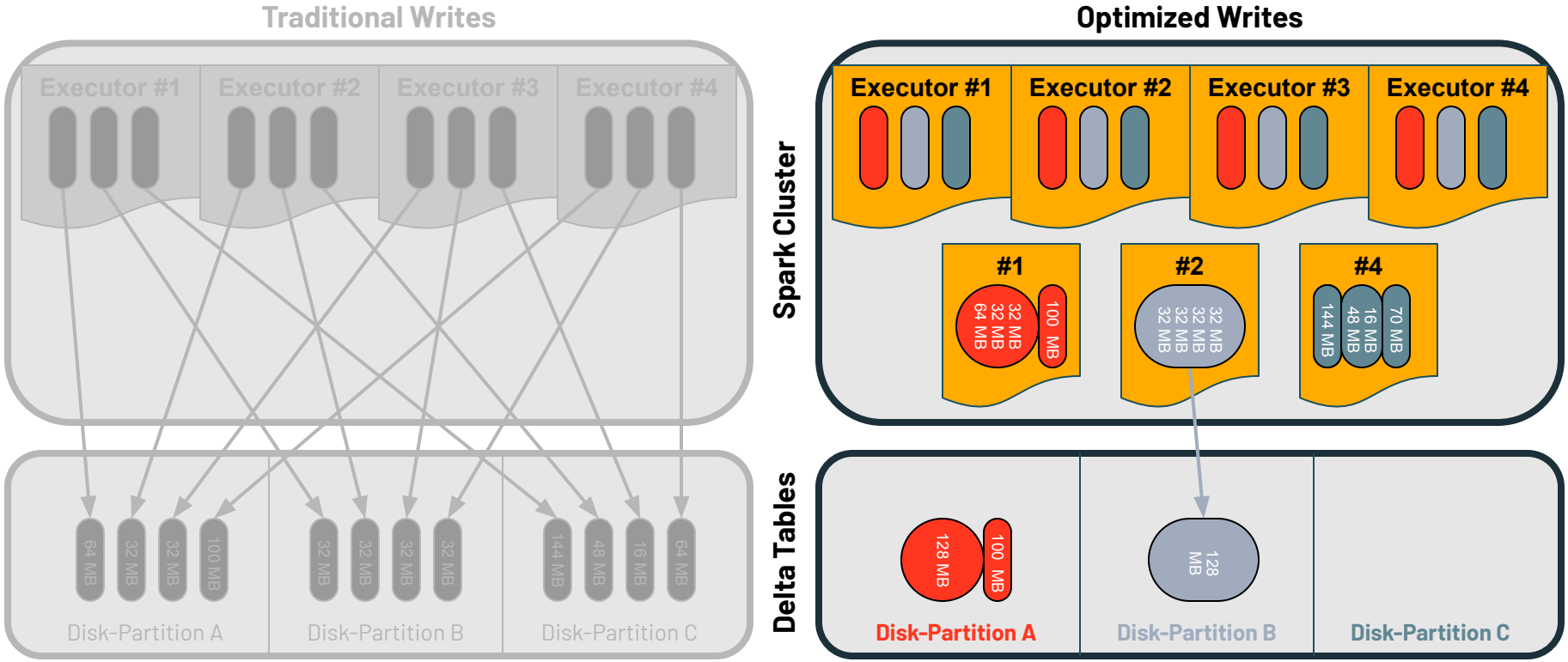
Small partitions are merged together, targeting a final size of 128 MB

Storage - Optimized Writes



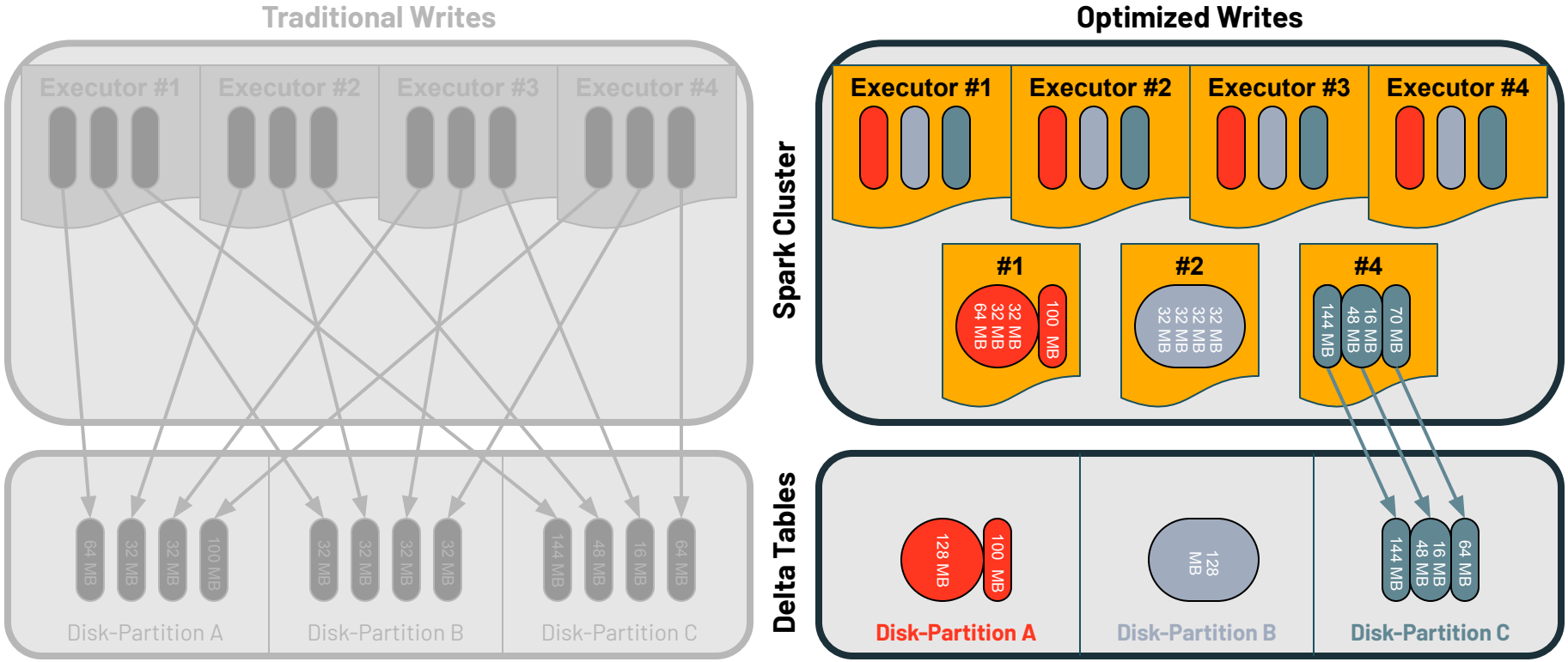
Each task will write one part file to the target disk-partition

Storage - Optimized Writes



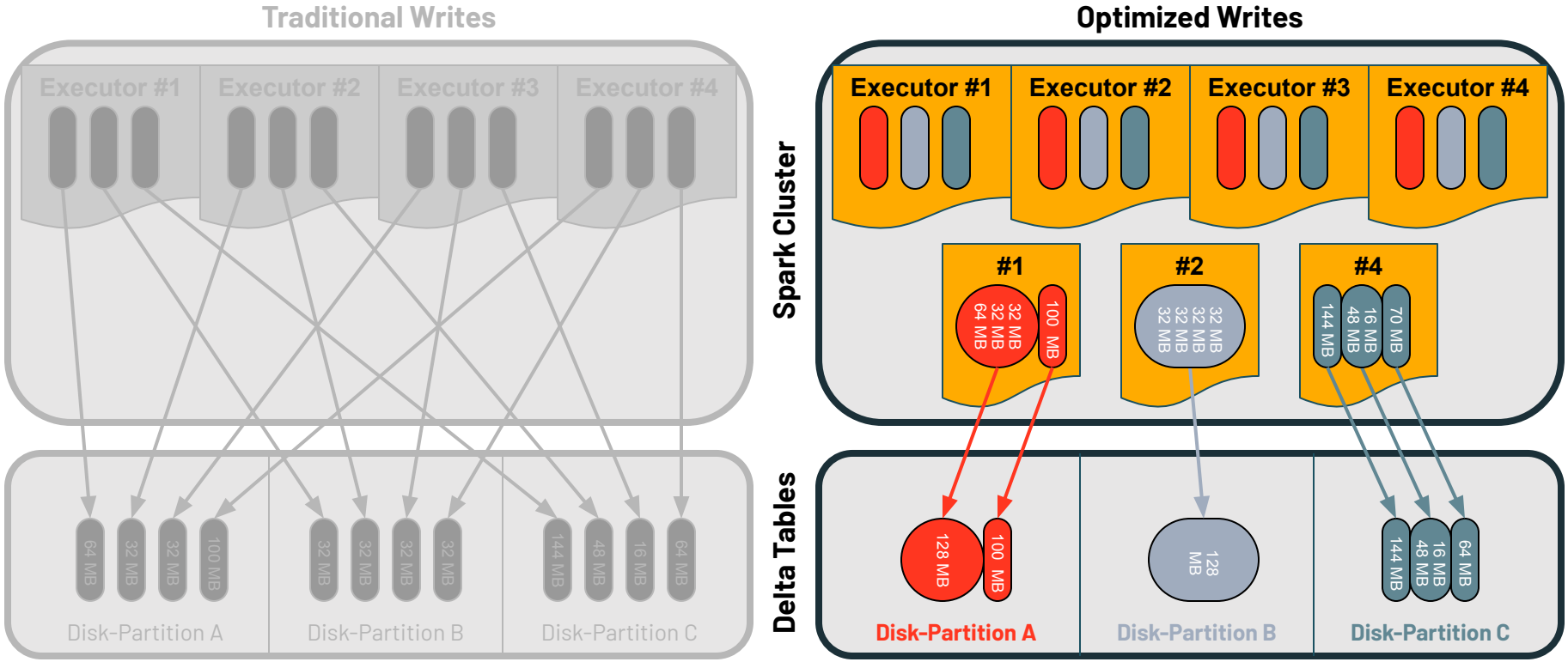
Each task will write one part file to the target disk-partition

Storage - Optimized Writes



All writes take place at the same time...

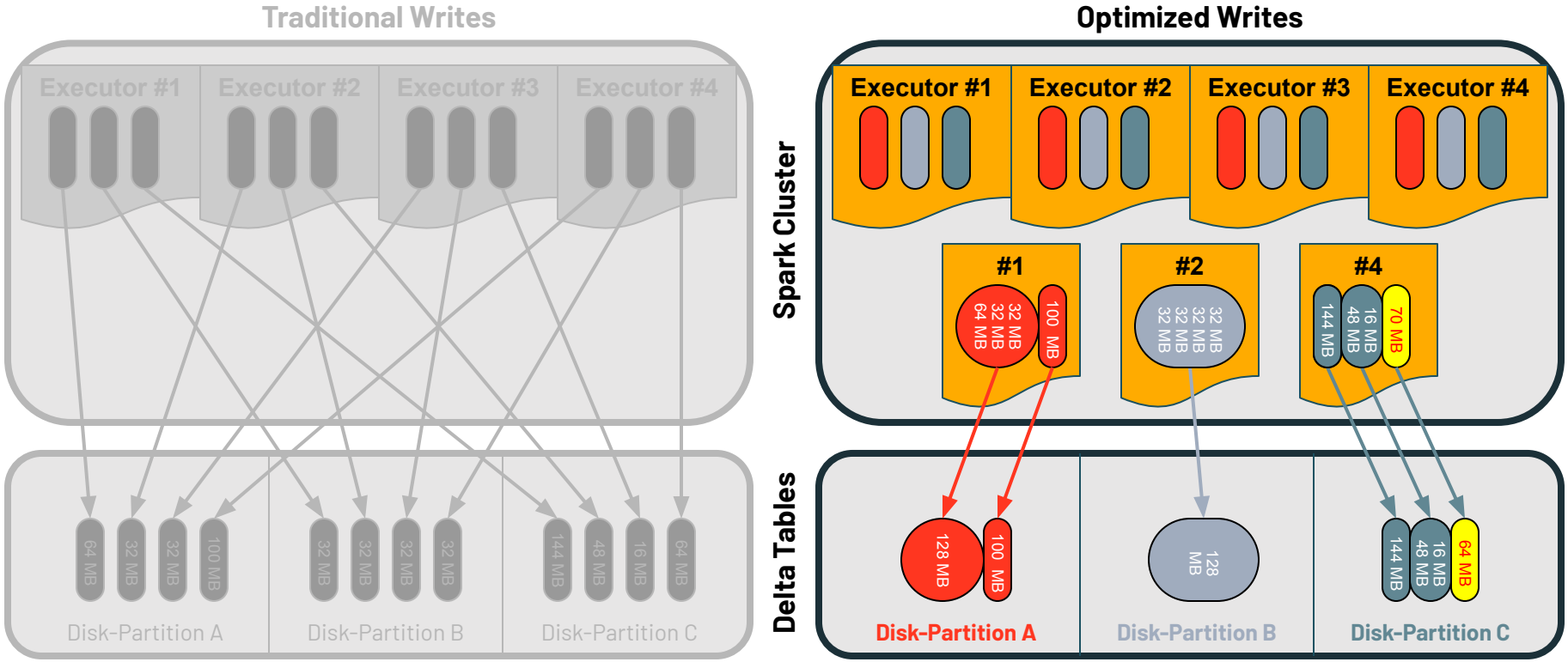
Storage - Optimized Writes



databricks

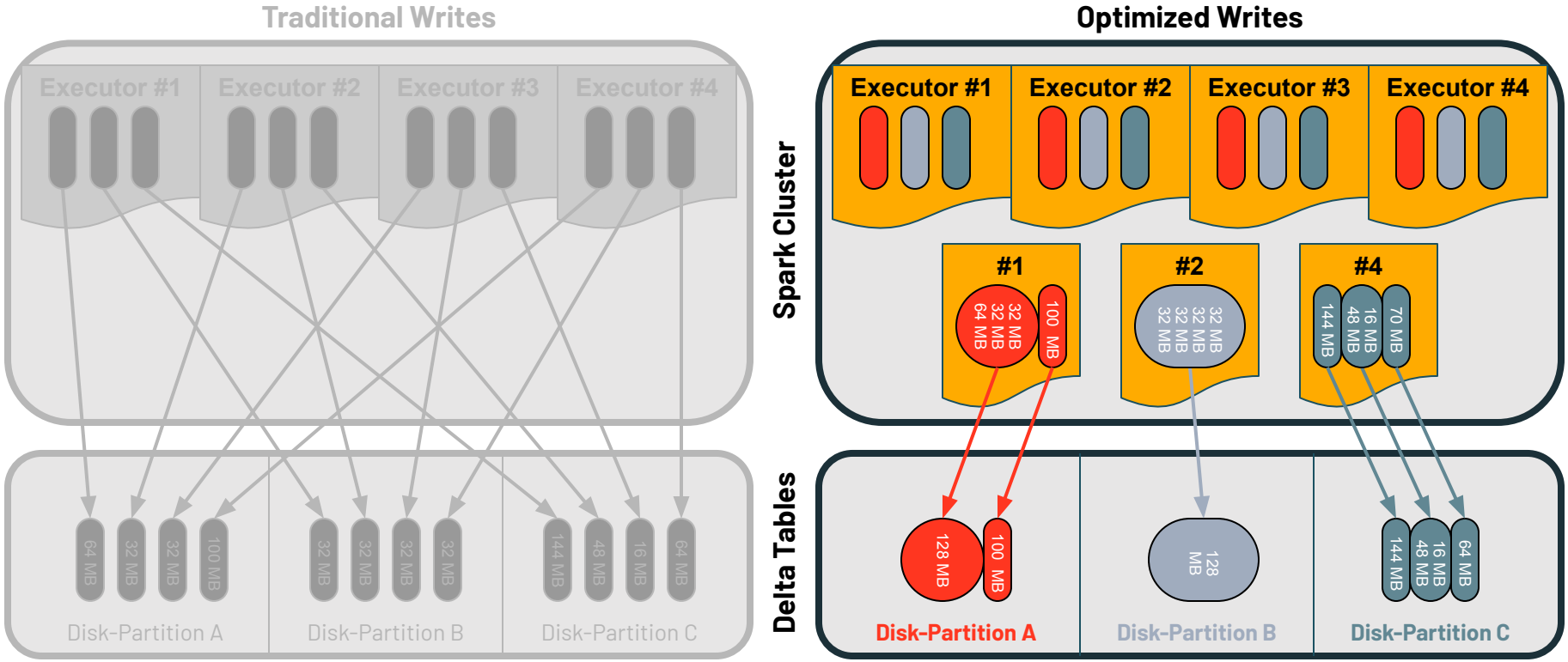
All writes take place at the same time...

Storage - Optimized Writes



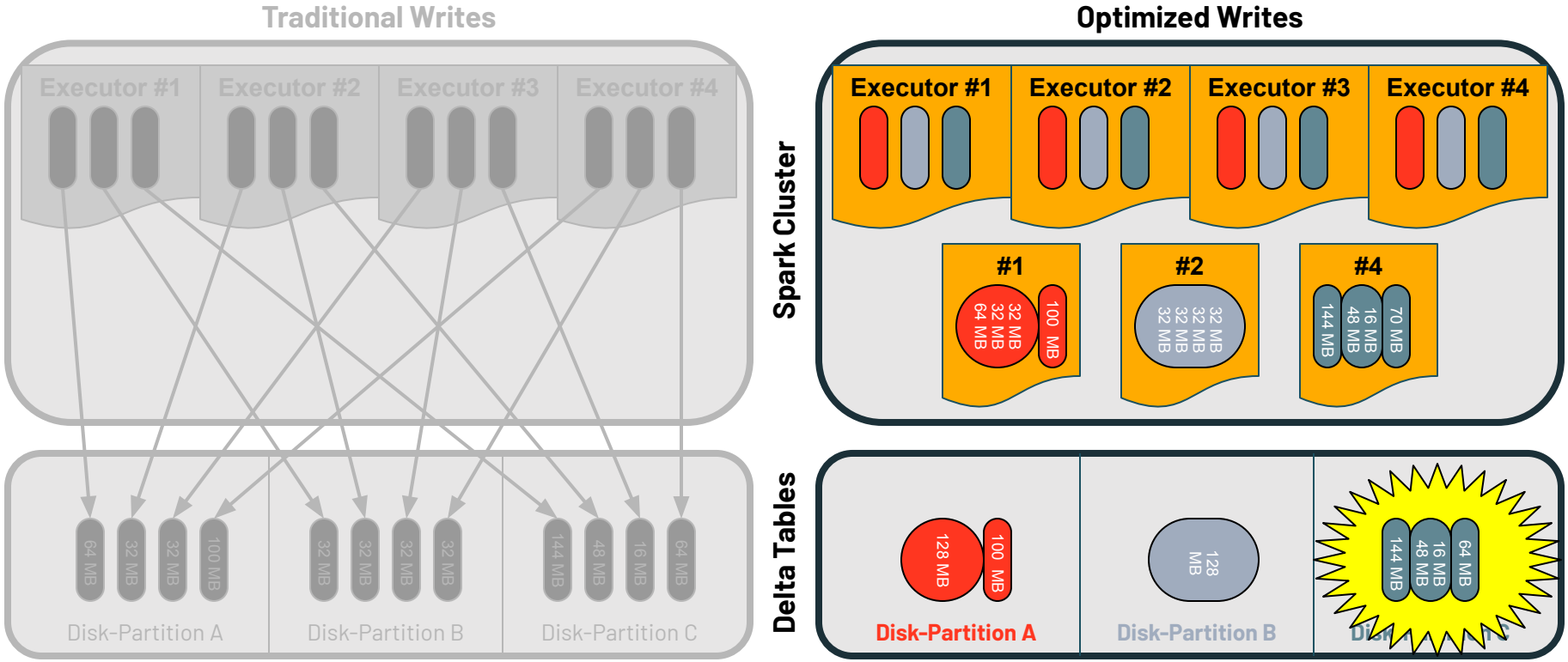
Estimation was high and subsequently not merged

Storage - Optimized Writes



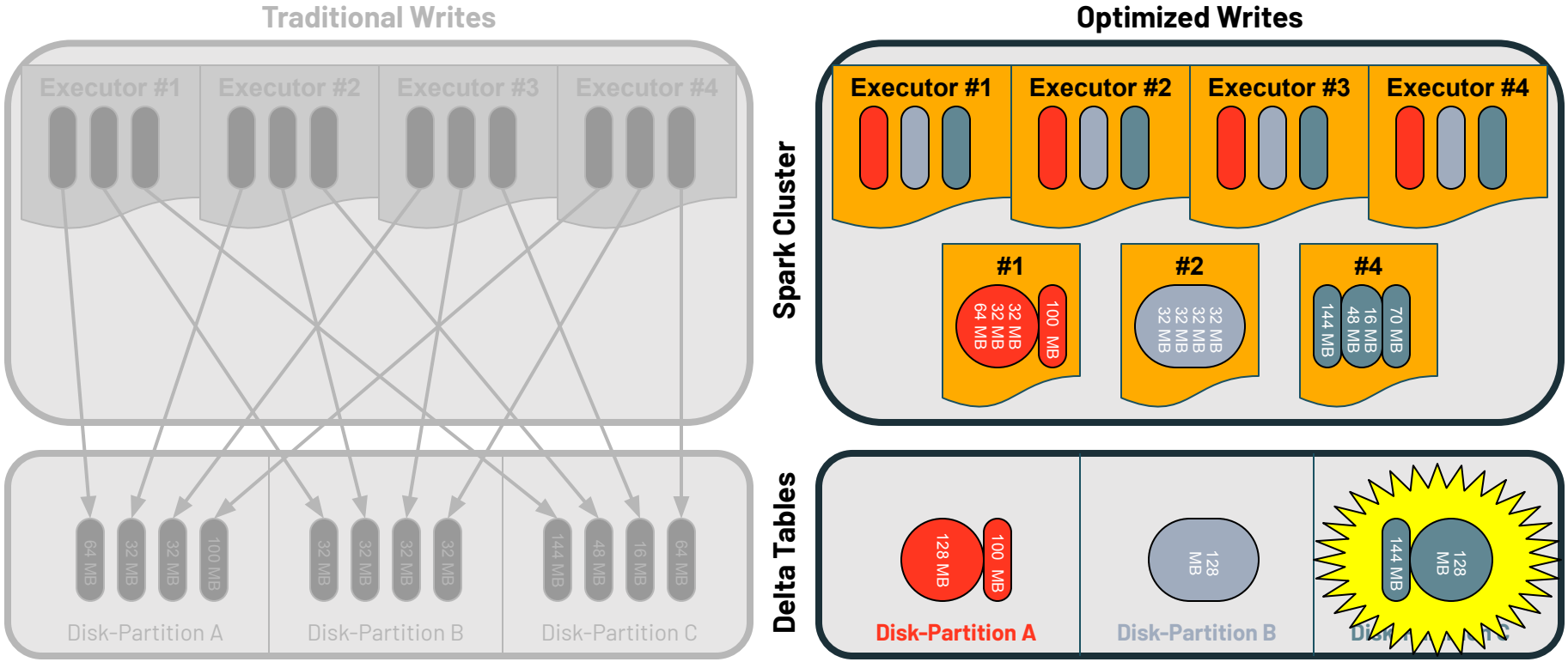
With files written to disk, a subsequent job can now compact smaller files

Storage - Optimized Writes



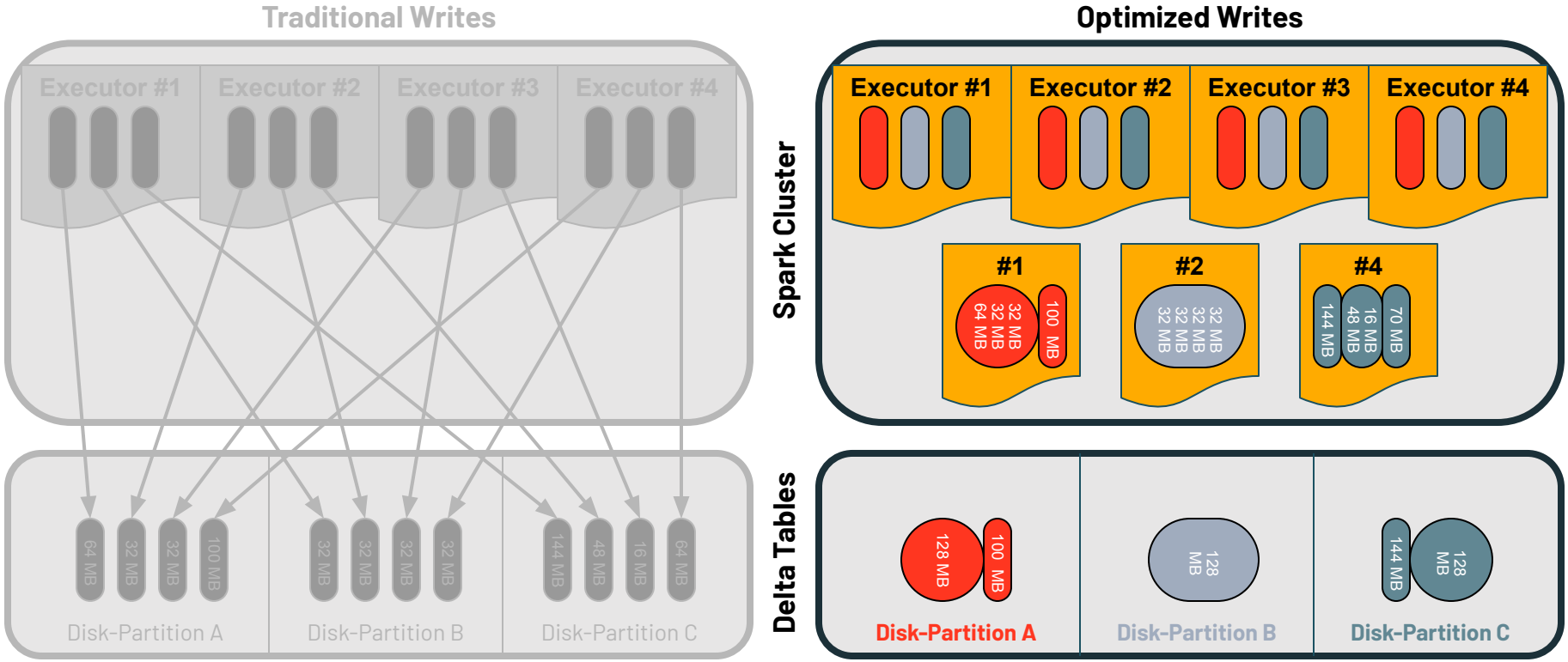
With files written to disk, a subsequent job can now compact smaller files

Storage - Optimized Writes



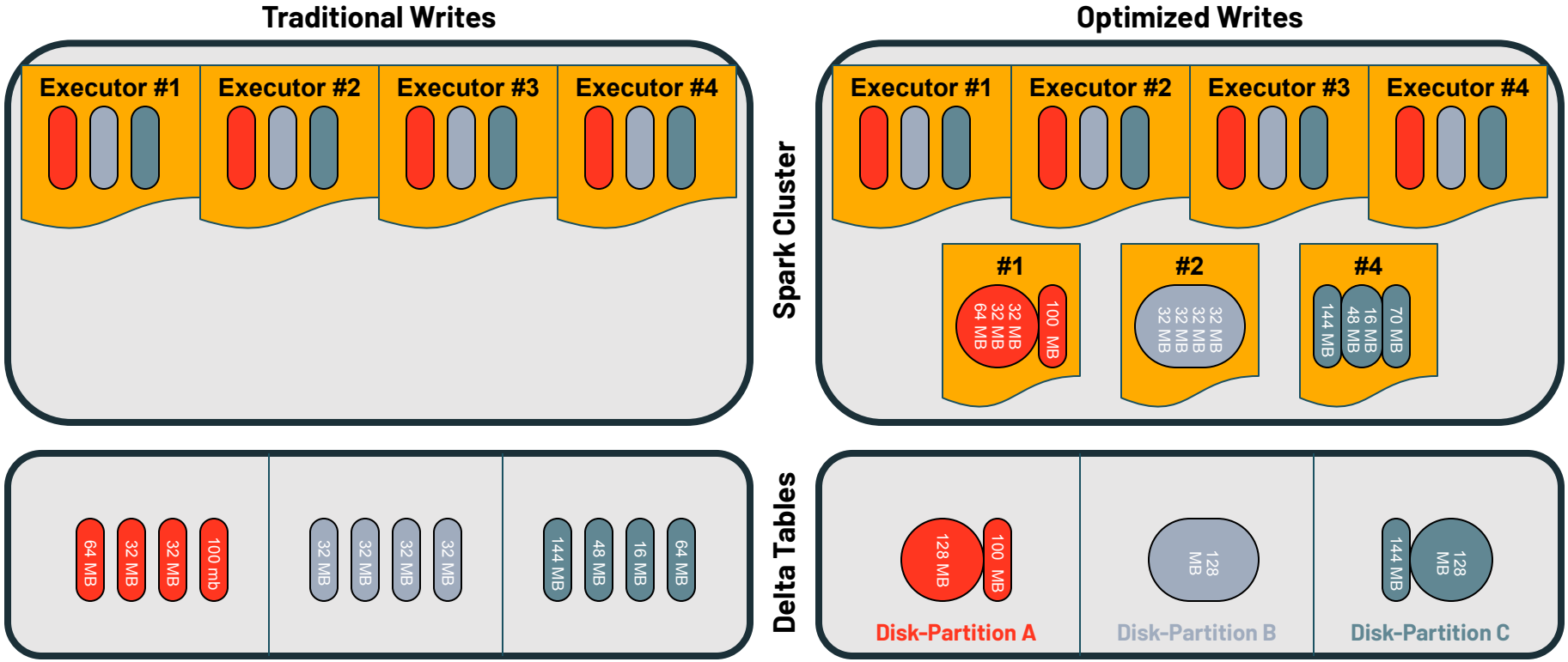
With files written to disk, a subsequent job can now compact smaller files

Storage - Optimized Writes



A little overhead with the write, huge payoffs on the subsequent reads

Storage - Traditional vs Optimized Writes



No optimization on file size, potentially inducing the tiny-files problem

Reduced disk IO, and optimized around 128 MB part files

Optimizing Apache Spark

The Five Most Common Performance Problems

Storage - Directory Scanning

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Directory Scanning

The next version of the "Tiny Files Problem" is Directory Scanning

- Here is the idea:
  - One can list the files in a single directory
  - A single list with thousands of files is still OK
  - The scan still is not as bad as the overhead of reading tiny files
- Highly partitioned datasets (data on disk) present a different problem:
  - For every disk-partition there is another directory to scan
  - Multiplied that number of directories by N secondary & M tertiary partitions
  - These have to be scanned by the driver one directory at a time

The 5 Most Common Performance Problems (The 5 Ss)

Storage - Scanning Example

Consider this common scenario:

- Consider 1 year's worth of data partitioned by year, month, day & hour
- 1 year * 12 months * 30 days * 24 hours = 8,640 distinct directories
- 10 years of data becomes 86,400 directories.

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Scanning In Action

See Experiment #8973

- Contrast **Step B**, **Step C** and **Step D**
  - Note the we are not executing any actions - only declaring the DataFrames
  - Note the total execution time for each command
  - Note the results for **countFiles(..)**
    - The **Records** total
    - The **Files** total
    - The **Directories** total
- See **Step E, F & G** for more variants and how they affect scanning
- For **Step J** open the **Spark UI** and look at the **Query Details** for the last job
  - Identify the proof that scanning is the root cause of this performance problem

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Scanning, Review

Step	Description	Duration	Records	Files	Directories
B	~100 records per part-file (tiny files)	~1 minute	34,562,416	345,612	1
C	Partitioned by year & month	~ 5 seconds	36,152,970	6,273	12
D	Partitioned by year, month, hour & day	~15 minutes	37,413,338	6,273	8,760

The 5 Most Common Performance Problems (The 5 Ss)

Storage - Scanning, Prove It

What proof is there in the **Query Details** for Experiment #8973, **Step J**, that scanning is the root cause of these performance problems?

What proof is there in the **Query Details** for Experiment #8973, **Step J**, that scanning is the root cause of these performance problems?

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Scanning, Prove It

Scan parquet +details
Stages: 75.0

number of files read	8,760
filesystem read data size total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
scan time total (min, med, max)	1.71 h (1.9 s, 6.3 s, 7.6 s)
estimated repeated reads high size total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
filesystem read data size (sampled) total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
filesystem read time (sampled) total (min, med, max)	16.6 m (137 ms, 1.2 s, 1.9 s)
metadata time	36 ms
size of files read	1108.6 MiB
estimated repeated reads low size total (min, med, max)	1108.5 MiB (107.0 KiB, 954.8 KiB, 7.1 MiB)
number of partitions read	8,760
rows output	37,413,338

What can we do to mitigate the impact of scanning?

The 5 Most Common Performance Problems (The 5 Ss)

Storage - Can we...?

What happens if we were to specify the schema?

- See Step H and determine if this solution works

Nope

What happens if we were to register it as a table?

- See Step I and determine if this solution works

Yes!

Optimizing Apache Spark

The Five Most Common Performance Problems

Storage - Schemas

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Schemas

- Inferring schemas (for JSON and CSV) require a full read of the file to determine data types, even if you only want a subset of the data

- Reading Parquet files requires a one-time read of the schema

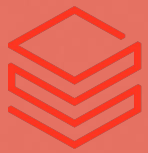
- However, supporting schema evolution is [potentially] expensive
  - If you have hundreds to thousands of part-files, each schema has to be read in and then merged which collectively can be fairly expensive
  - Schema merging was turned off by default starting with Spark 1.5
  - Enabled via the **spark.sql.parquet.mergeSchema** configuration option or the mergeSchema option

What can we do to mitigate the schema issues?

The 5 Most Common Performance Problems (The 5 Ss)
Storage - Schema Mitigation

There are several ways to mitigate some of these issues:

- Provide the schema every time
  - Especially for JSON and CSV
  - Applicable to Parquet and other formats
- Use tables - the backing meta store will track the table's schema
- Or just use Delta
  - Zero reads with a meta store
  - At most one read, even with schema evolution



databricks