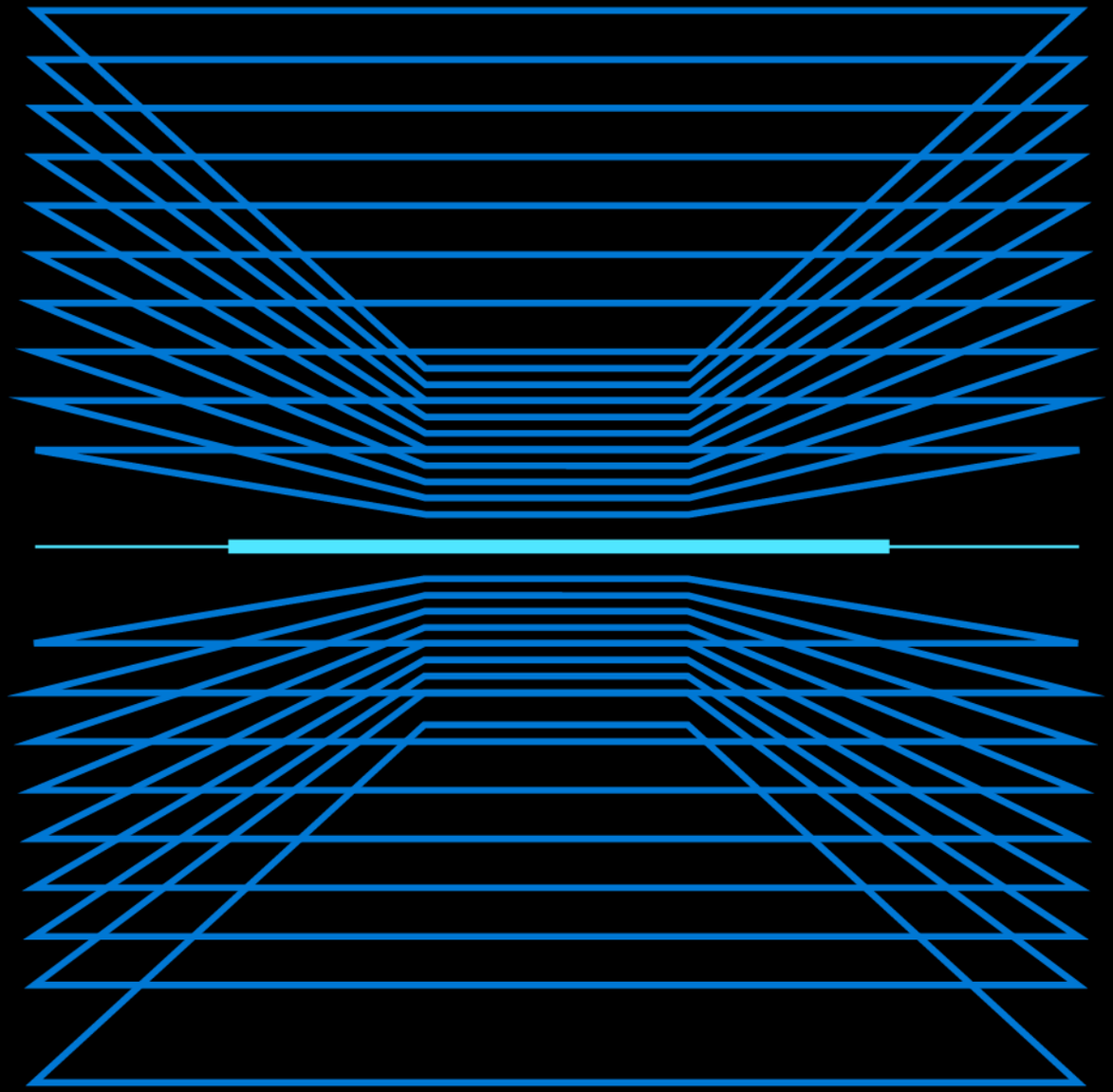
Microsoft

DPNS ’26 Spark Performance Engineering

Tuning, Optimization, Debugging & Beyond

Miles Cole - Principal Program Manager @ Microsoft
Jorge Docampo Carro - Senior Program Manager @ Microsoft



Introduction

About Miles



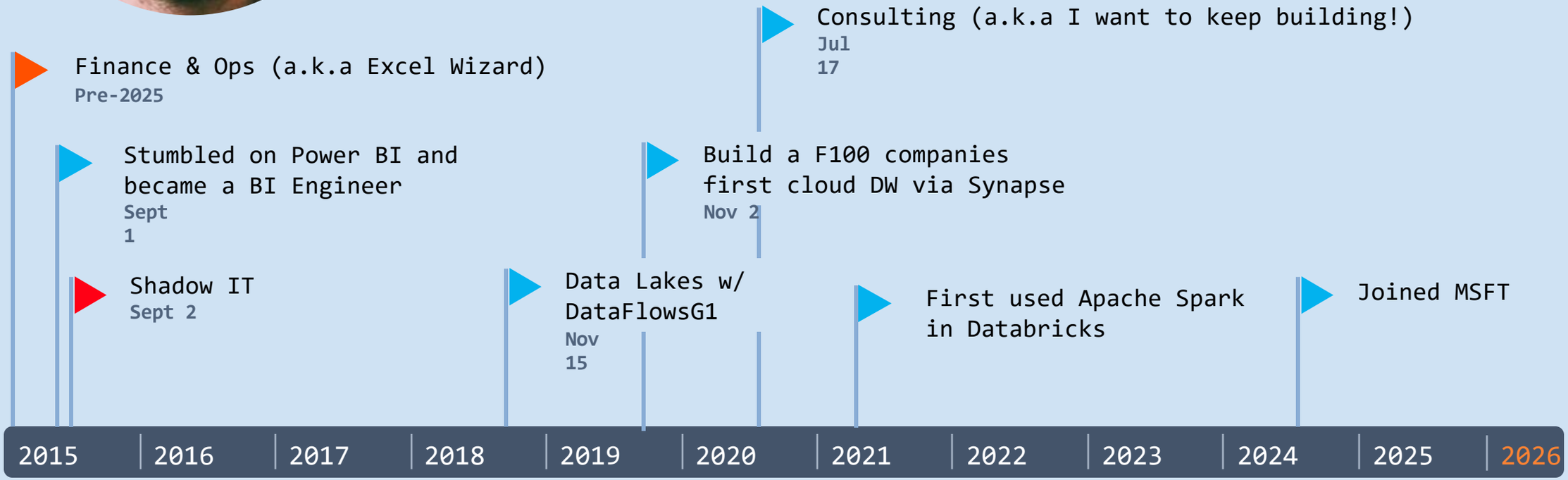
Miles Cole

Principal Program Manager, Azure Data CAT – Spark Specialist @ Microsoft

From finance and operations to shadow IT to building software and data platforms

https://milescole.dev

The Journey



About Jorge

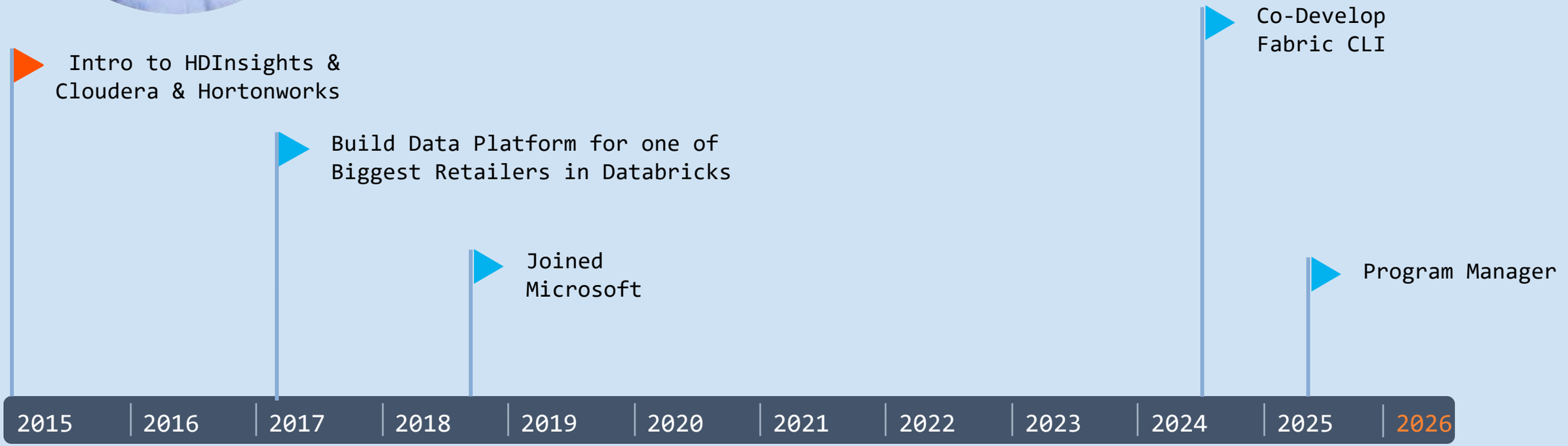


Jorge Docampo Carro

Senior Program Manager, Azure Data CAT – Spark Specialist @ Microsoft

From performance engineer to consulting to building data platforms and products

The Journey



Today's Agenda

Start-End	Duration	Focus
09:00-09:15	30m	Welcome & Intros
09:15-10:30	75m	**Module 0 – Foundational Theory**
10:30-10:40	10m	*AM Break*
10:40-12:00	80m	**Module 1 – Performance Diagnostics**
12:00-13:00	60m	*Lunch*
13:00-14:00	60m	**Module 2 – Delta Table Design and Optimization**
14:00-15:00	60m	**Module 3 – Query Optimization Patterns**
15:00-15:30	20m	*PM Break*
15:30:16:30	60m	**Module 4 – Advanced Debugging**
16:30-17:00	30m	Q&A

©Microsoft Corporation

Premium sponsors

VENZO_

INSPARI

valantic companies

itm8®

Eddytor™

dataon

Power your data

Standard sponsors

Backstage

Columbus

TheVIT

DATA - ANALYTICS – AI

twoday

©Microsoft Corporation

Raffle Prizes

EUROPEAN MICROSOFT FABRIC Community Conference









©Microsoft Corporation

Lab Installation

Installing the Lab

1. Install Fabric Jumpstart

```
%pip install fabric-jumpstart --quiet
```

2. Install the Lab

```
import fabric_jumpstart
import fabric_jumpstart as jumpstart

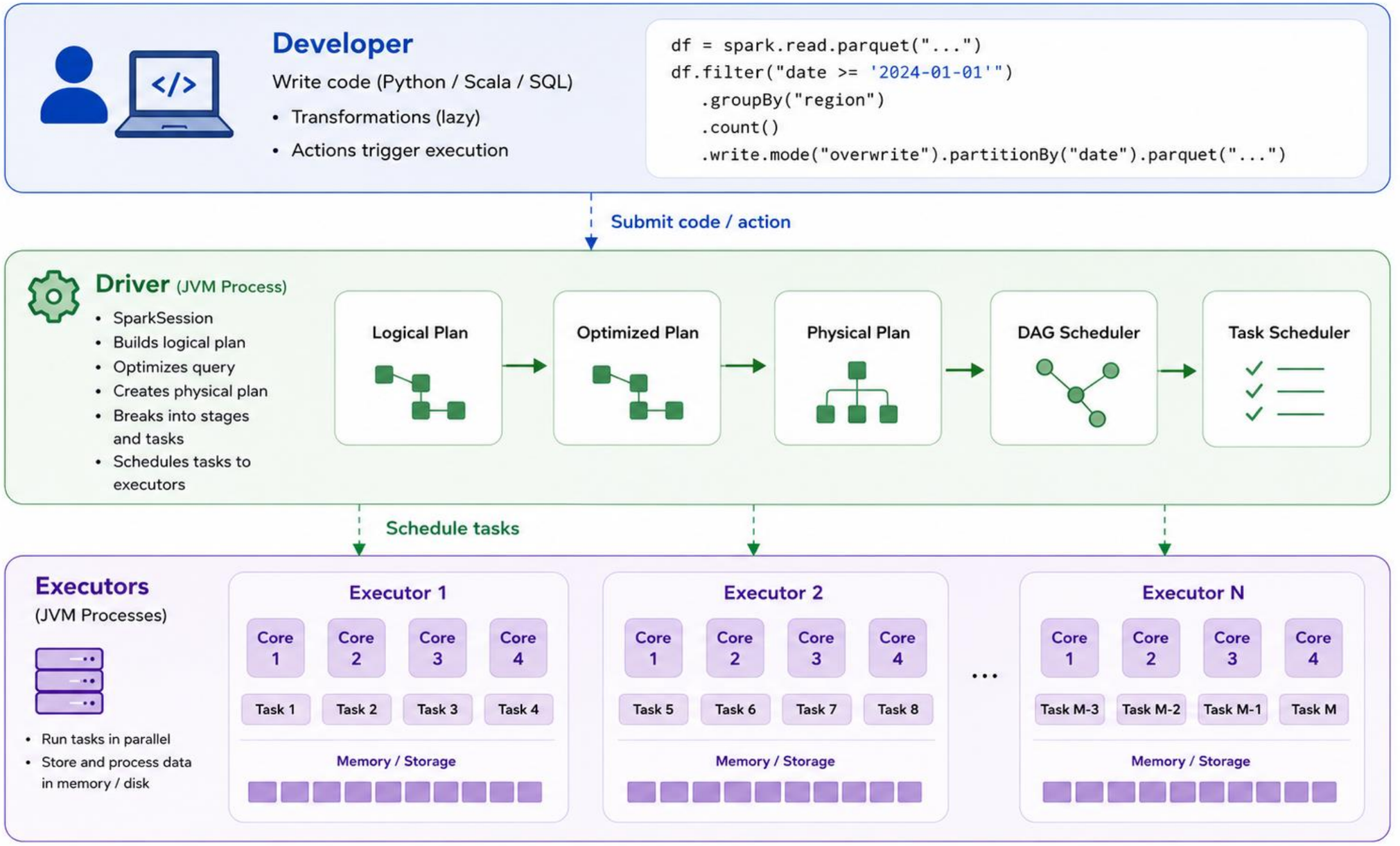
jumpstart._install_from_github(
    logical_id="spark-performance-engineering",
    repo_url="https://github.com/mwc360/fabric-spark-performance-workshop.git",
    repo_ref="dpns_2026",
    entry_point="00-getting-started.Notebook",
    items_in_scope=["Lakehouse", "Notebook", "Environment", "SparkJobDefinition"],
    workspace_path="workspace/",
    name="Spark Performance Engineering",
)
```

©Microsoft Corporation

Module 0
Fundamentals

0A — Spark Execution Architecture

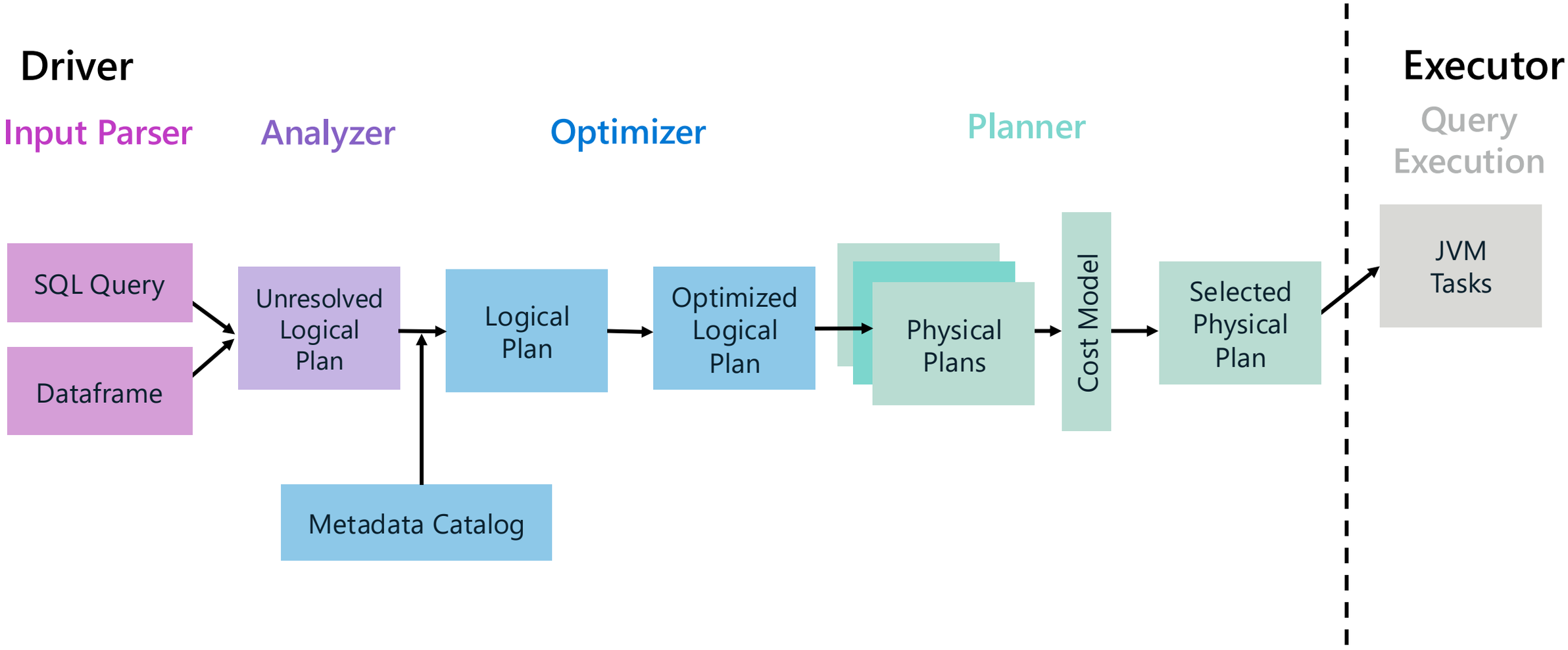
The Spark Execution Model



©Micro

Spark Execution Architecture

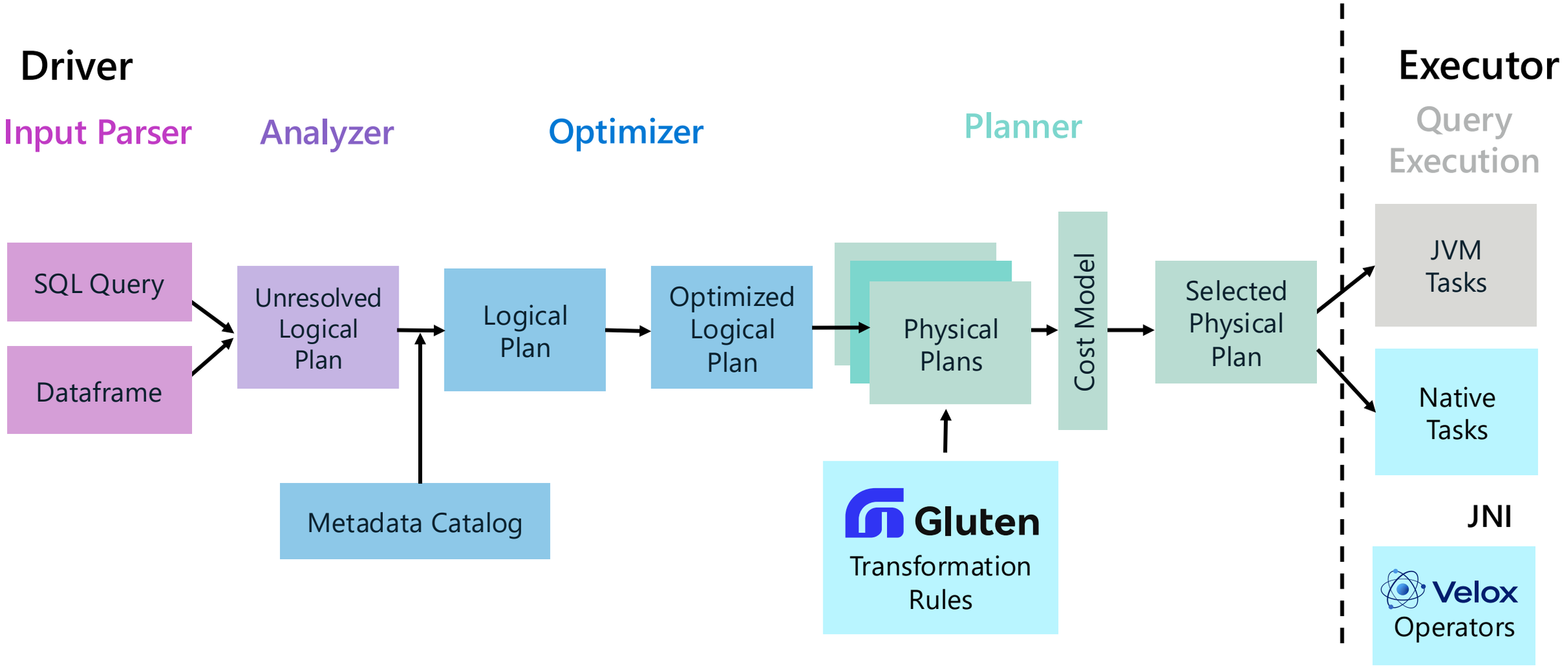
Query lifecycle



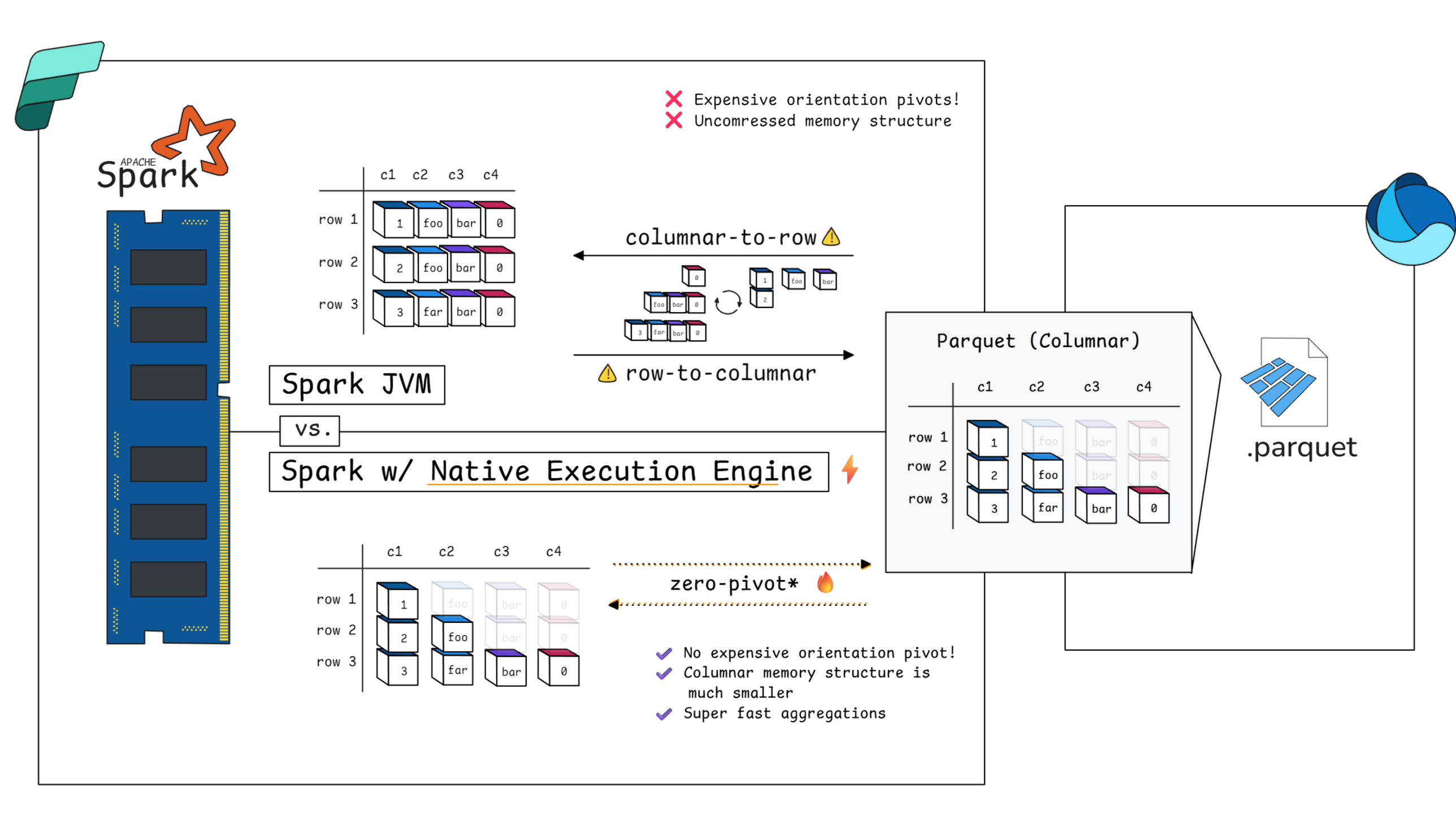
©Microsoft Corporation

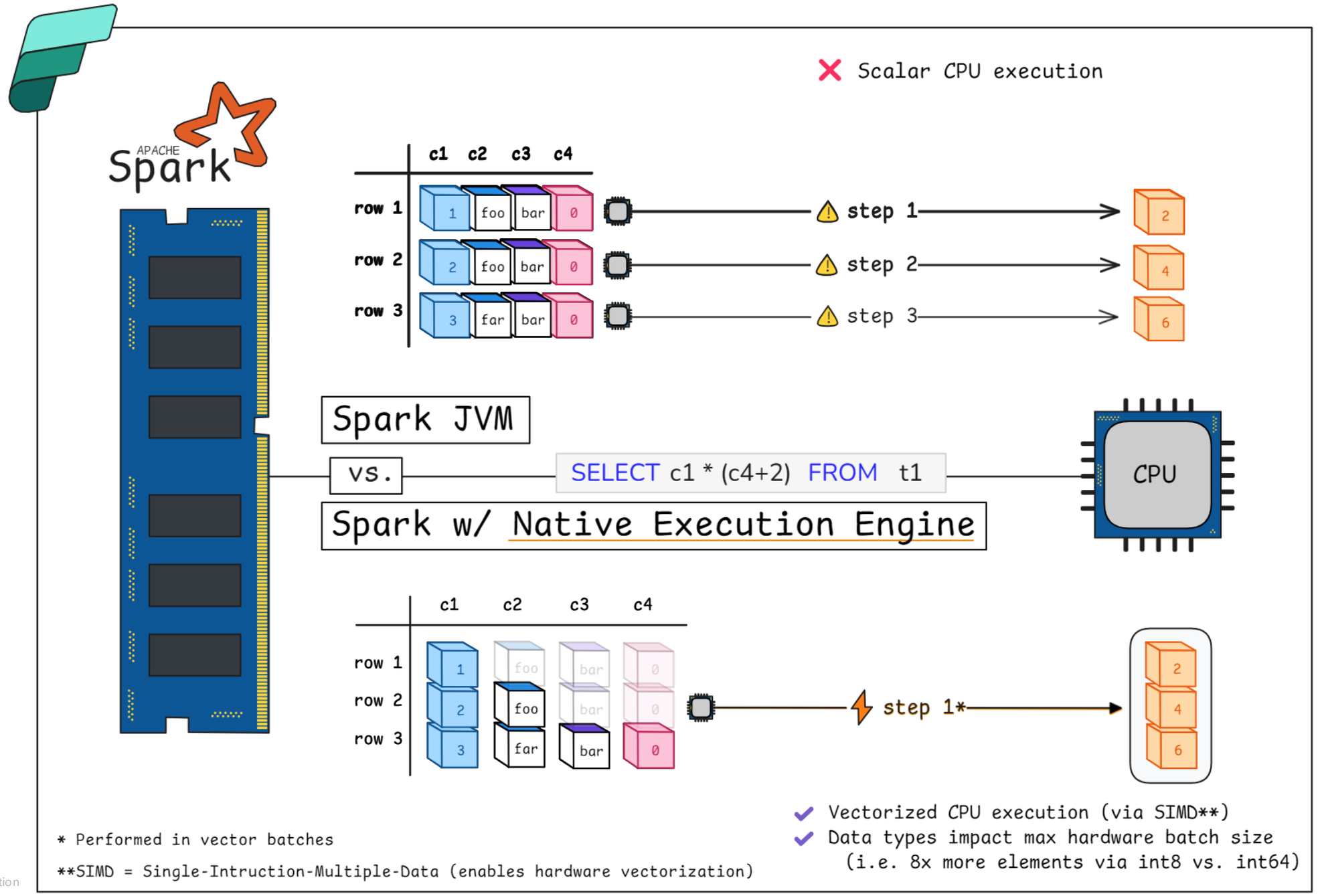
Native Execution Engine Integration

Query lifecycle



©Microsoft Corporation


Expensive orientation pivots!
Uncomressed memory structure
APACHE Spark
c1 c2 c3 c4
row 1 1 foo bar 0
row 2 2 foo bar 0
row 3 3 far bar 0
columnar-to-row
row-to-columnar
Spark JVM
vs.
Spark w/ Native Execution Engine
Parquet (Columnar)
.parquet
zero-pivot*
No expensive orientation pivot!
Columnar memory structure is much smaller
Super fast aggregations



©Microsoft Corporation

0B — Thinking Distributed

Why Thinking Distributed Matters

Distributed computing mindset

Network Is the Bottleneck

- Every shuffle serializes data, writes to disk, transfers over network, then deserializes — avoid unnecessary shuffles
- A single collect() on 500M rows moves all data to the driver and will result in OOM — avoid collecting large results to the driver

Parallel Computing Enables Scale

- Distributed systems scale by parallelizing work, not by making a single machine do more work.

Partition-Aware Programming

- Think in partitions, not rows — your data is distributed into partitions (a.k.a. slices) where code runs independently on each partition

Anti-Patterns of Single-Node Thinking

- Calling .toPandas() before filtering, using Python UDFs for operations Spark has built-in, mixing and matching engines

❌ BAD: Switching between engines may bring data to driver

```python
# Anti-pattern: pulls ALL data to driver node
df = spark.sql("...")
pdf = df.toPandas()
result = pdf[pdf['amount'] > 1000].sum()
```

✅ GOOD: Distributed Spark-native operations

```python
# Runs distributed — no data movement to driver
from pyspark.sql import functions as F
result = (df
    .filter(F.col('amount') > 1000)
    .agg(F.sum('amount').alias('total'))
)
result.show()  # only result is sent to driver
```

©Microsoft Corporation

Core Concepts: Task Parallelism & Data Partitions

Parallelism fundamentals

Partition Sizing Rules of Thumb

- Target 128 MB – 512 MB per partition for optimal throughput; too small = scheduling overhead, too large = GC pressure and spill
- Number of partitions ≈ (dataset size in MB) / 128 — adjust based on cluster memory

Task Scheduling

- 1 partition = 1 task; 1 task requires 1 slot (vCore) — if partitions > slots, tasks queue and wait
- Ideal ratio: partitions = 2–3× total executor cores to keep all cores busy with balanced work

Default Parallelism

- spark.default.parallelism (defaults to # of executor cores) governs RDD ops
- spark.sql.shuffle.partitions (default 200) governs DataFrame shuffles
- Adaptive Query Execution (AQE) will coalesce partitions as needed

Inspecting and adjusting partition counts

```python
# Check current partition count
print(f'Partitions: {df.rdd.getNumPartitions()}')

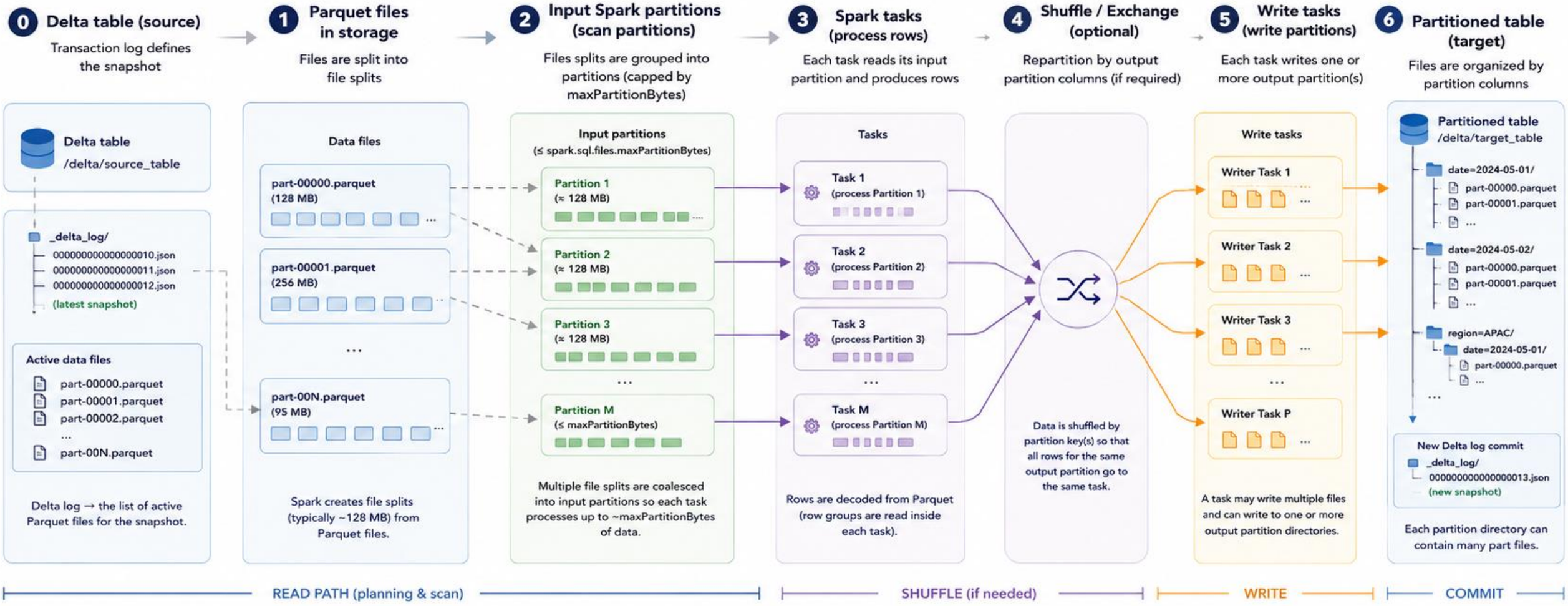

# Check cluster default parallelism
print(spark.sparkContext.defaultParallelism)
print(spark.conf.get('spark.sql.shuffle.partitions'))


# Repartition to target count (causes shuffle)
df = df.repartition(48)


# Coalesce to reduce partitions WITHOUT full shuffle
df = df.coalesce(16)
```

©Microsoft Corporation

End-to-End: Data Lifecycle & Partitions (Spark read from Delta → write to partitioned table)



What is a partition?

Row Group (inside Parquet file)
The smallest chunk of rows in a Parquet file with min/max stats, used for predicate pushdown.

File Split
A contiguous byte range of a file (~128 MB by default) that can be read independently.

Input (Spark) Partition / Scan Partition
A unit of work for a task created by grouping file splits, capped by spark.sql.files.maxPartitionBytes.

Shuffle Partition
A partition of rows after an exchange based on partition keys (if shuffle/repartition is performed).

Output Partition (Table Partition)
A logical partition defined by partition columns (e.g., date=..., region=...).

File (Output)
A physical file written by a task within an output partition directory. Many files can exist per partition.

0C — Thinking Lazy

Why Thinking Lazy Matters

Lazy evaluation

How Catalyst Optimizes

- Spark builds a DAG of transformations but executes nothing until an action (show, count, write) is called
- Catalyst reorders filters before joins (predicate pushdown), eliminates unused columns (projection pruning), and folds constants

What Breaks Optimization

- Premature actions (count, show, collect) force execution of the partial plan — Catalyst cannot optimize across action boundaries
- Calling .cache() too early materializes a suboptimal plan before Catalyst finishes rewriting
- Python UDFs are opaque to Catalyst — it cannot push predicates through them or reorder around them

Best Practice

- Chain transformations into a single pipeline; call the action only at the end so Catalyst sees the full picture
- Use df.explain(True) to verify Catalyst is applying expected optimizations before running at scale

❌ Premature action breaks Catalyst optimization

```python
# Forces execution BEFORE join
count = df_raw.count() # ACTION: triggers full scan
print(f'Rows: {count}')
result = df_raw.join(dim, 'key') # separate plan
```

✅ Single pipeline — Catalyst optimizes end-to-end

```python
result = (df_raw
    .filter(F.col('year') >= 2023)      # lazy
    .join(dim, 'key')                   # lazy
    .select('key', 'value', 'category') # lazy
)
result.explain(True)  # inspect optimized
                      # logical plan
```

©Microsoft Corporation

0D — AQE

Adaptative Query Execution (AQE)

AQE

What it is

- Adaptive Query Execution allows Spark to optimize the physical execution plan at runtime, using real statistics collected while the job is running.
- Without AQE, Spark mainly relies on estimates available before execution. With AQE, Spark can revise decisions once it has better information about actual data size and distribution.

Key Idea

- Spark depends less on static estimates and adapts the execution plan based on what actually happens during the query.

©Microsoft Corporation

How AQE Works During Execution

AQE

Execution Flow

1. Spark creates an initial physical plan using Catalyst and available statistics.
2. The query is divided into query stages, usually separated by shuffle boundaries.
3. Completed stages produce runtime statistics such as data size, row counts, and partition distribution.
4. Spark uses those statistics to re-optimize the remaining physical plan.
5. The next stages run using the adapted plan.

Example

- If Spark discovers after a shuffle that one side of a join is much smaller than expected, it can replace a Sort Merge Join with a Broadcast Hash Join before executing the join.

Important Detail

- AQE does not continuously rewrite the entire query. It adapts the plan at safe points where stage outputs are already materialized and reliable runtime statistics are available.

©Microsoft Corporation

Problems AQE Helps Solve

AQE

Common Performance Issues

- Too many or too few shuffle partitions, causing either excessive task overhead or poor parallelism.
- Suboptimal join strategies, such as using Sort Merge Join when Broadcast Hash Join would be cheaper.
- Data skew, where a few partitions contain much more data than the rest and create stragglers.
- Rigid execution plans that cannot react to the real size and distribution of intermediate data.

Outcome

- AQE can reduce execution time and improve cluster resource usage by making better decisions during the query rather than relying only on pre-execution assumptions.

©Microsoft Corporation

Main AQE Optimizations

AQE

Coalescing shuffle partitions

Spark dynamically reduces the number of small shuffle partitions after observing the actual shuffle output size.
This reduces scheduling overhead and avoids running many tiny tasks.

Switching join strategy

Spark can change the physical join strategy at runtime when actual data sizes differ from estimates.
A common case is switching from Sort Merge Join to Broadcast Hash Join when one side is small enough to broadcast.

Skew join optimization

Spark detects unusually large shuffle partitions and splits them into smaller pieces.
This reduces stragglers and improves the total completion time of the stage.

©Microsoft Corporation

How AQE Works During Execution

AQE

Best practices

Use the Spark UI to compare the initial physical plan with the adaptive plan.
Do not treat AQE as a substitute for good partitioning, data layout, and query design.
Keep table statistics useful when possible, especially for cost-based planning before runtime statistics exist.
Validate performance changes with representative benchmarks, not only small test queries.

Final message

AQE reduces the cost of bad estimates and makes Spark SQL workloads more resilient, but strong performance engineering practices still matter.

©Microsoft Corporation

0E — Optimizing I/O

Narrow vs. Wide Transformations

Transformation types

Narrow Transformations (no shuffle)

- Each output partition depends on exactly one input partition — no data exchange between executors
- Examples: select(), filter(), withColumn(), map(), flatMap(), union() (same schema)

Wide Transformations (shuffle required)

- Output partitions depend on multiple input partitions — triggers a shuffle (stage boundary)
- Examples: groupBy().agg(), join(), orderBy(), distinct(), repartition()

Stage Boundaries

- Every wide transformation creates a new stage — check the Spark UI DAG to count stage boundaries
- Fewer stages = fewer shuffles = faster execution; restructure logic to minimize wide dependencies

Narrow vs. Wide — observe stage boundaries

```python
# Narrow — single stage, no shuffle
narrow_df = (df
    .filter(F.col('status') == 'active')
    .select('id', 'name', 'amount')
    .withColumn('tax', F.col('amount') * 0.08)
)


# Wide — triggers shuffle, new stage boundary
wide_df = (narrow_df
    .groupBy('name')
    .agg(F.sum('amount').alias('total'))
    .orderBy(F.desc('total'))
)


# Inspect: look for Exchange nodes = shuffle
wide_df.explain()
```

©Microsoft Corporation

Shuffle Internals

Understanding shuffle

Sort-Merge Shuffle (default in Spark 3.x)

- Map side: each task sorts its output by partition key, writes one file per reducer → creates shuffle files on local disk
- Reduce side: each reducer fetches its partitions from all mappers via HTTP, then merge-sorts them

Spill Implications

- If a task's shuffle buffer exceeds spark.shuffle.spill threshold, data spills to disk — severe performance degradation
- Spill shows up in Spark UI Stage metrics: 'Shuffle Spill (Memory)' vs. 'Shuffle Spill (Disk)'

Tuning Shuffle

- spark.sql.shuffle.partitions controls post-shuffle partition count — too few = OOM, too many = small file overhead
- With AQE enabled, Spark auto-coalesces small shuffle partitions — prefer AQE over manual tuning

Shuffle configuration and monitoring

```
# Default shuffle partitions (200 is Spark default)
spark.conf.set('spark.sql.shuffle.partitions', '48')


# Enable Adaptive Query Execution (on by default in Fabric)
spark.conf.set(
    'spark.sql.adaptive.enabled', 'true')


# AQE: auto-coalesce small shuffle partitions
spark.conf.set(
    'spark.sql.adaptive.coalescePartitions.enabled', 'true')


# AQE: target post-shuffle partition size (default 64MB)
spark.conf.set(
    'spark.sql.adaptive.advisoryPartitionSizeInBytes', '128MB')
```

©Microsoft Corporation

Broadcast vs. Shuffle Joins

JOIN strategies

Broadcast Hash Join (BHJ)

- Small table is serialized and broadcast to every executor — no shuffle of the large table
- Default threshold: spark.sql.autoBroadcastJoinThreshold = 10 MB; increase cautiously (driver must hold the table)
- Best when: one side < 100 MB, avoids expensive shuffle of the large side entirely

Sort-Merge Join (SMJ)

- Both sides shuffled by join key, sorted, then merge-joined — default for large-large joins
- Produces stable performance but 2× shuffle cost; benefits from pre-bucketed tables

AQE Auto-Switching

- With AQE enabled (default), Spark detects at runtime if a side is small enough and dynamically switches SMJ → BHJ mid-execution

Controlling join strategy

```python
from pyspark.sql.functions import broadcast


# Force broadcast join via hint (overrides threshold)
result = large_df.join(broadcast(small_df), 'key')


# Check / set broadcast threshold
print(spark.conf.get('spark.sql.autoBroadcastJoinThreshold'))
spark.conf.set('spark.sql.autoBroadcastJoinThreshold', '50MB')


# Verify chosen join strategy in physical plan
result.explain()  # look for BroadcastHashJoin vs SortMergeJoin
```

©Microsoft Corporation

Parquet Internals: Row Groups, Columns, Pages

FILE format DEEP DIVE

Parquet Is Columnar — Not Row-Oriented

- Data is stored column-by-column within each row group, enabling per-column I/O and encoding

Hierarchy: Row Group → Column Chunk → Page

- Row Group: horizontal partition (~1 GB default), unit of read parallelism
- Column Chunk: one column's data within a row group, independently readable
- Page: smallest I/O unit (~1 MB default), compressed and encoded independently

Encoding Types Applied Per Column Chunk

- Dictionary encoding: maps repeated values to integer IDs — highly effective for low-cardinality strings
- RLE (Run-Length Encoding): compresses consecutive repeated values — ideal for sorted or clustered columns
- Delta encoding: stores differences between consecutive values — effective for timestamps and monotonic IDs
- Bit-packing: reduces integer width to the minimum bits required — saves space for small-range integers

Footer Metadata: The Reader's Roadmap

- File footer contains schema, row group offsets, column chunk offsets, and min/max/null statistics
- Understanding this hierarchy helps debug: slow reads (too many pages), poor pushdown (bad stats), and bloated files (wrong encoding)

©Microsoft Corporation

Which data layout would be faster to do a full scan via Spark?

20GB Delta Table

1 file w/ 160 - 128MB row groups



VS

20GB Delta Table

~160 files w/ 128MB row groups



©Microsoft Corporation

0F — Delta Lake

Which data layout would be faster for a selective lookup?

20GB Delta Table

1 file w/ 160 - 128MB row groups



VS

20GB Delta Table

~160 files w/ 128MB row groups



©Microsoft Corporation

File Skipping

Read Less Data

How File Skipping Works

Query:

```
1  SELECT * FROM sales WHERE date = 2024-03-15'
```

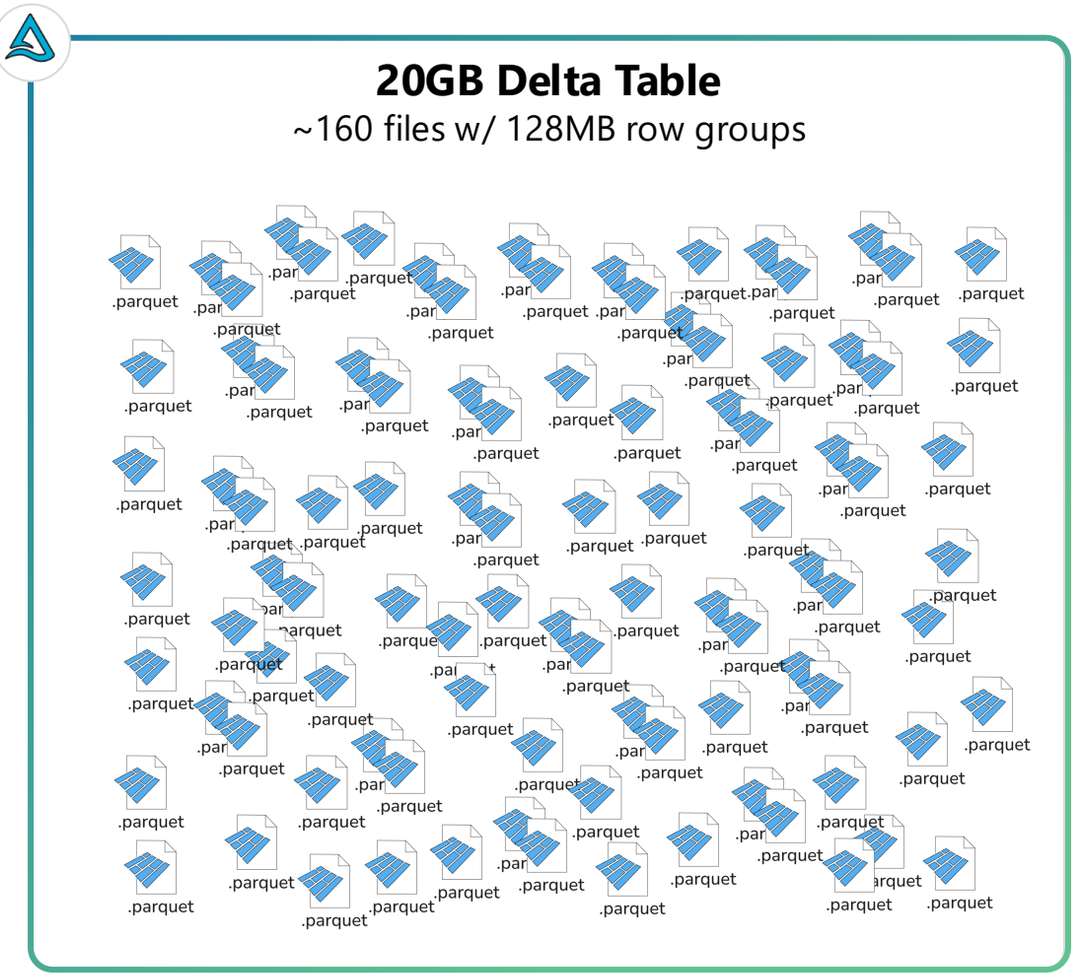
File	Min / Max Stats	Result
File 1	`date [2024-01-01 .. 2024-02-28]` →	**SKIPPED**
File 2	`date [2024-03-01 .. 2024-03-31]` →	**READ ✓**
File 3	`date [2024-04-01 .. 2024-05-31]` →	**SKIPPED**
File 4	`date [2024-06-01 .. 2024-07-31]` →	**SKIPPED**
File 5	`date [2024-08-01 .. 2024-09-30]` →	**SKIPPED**

→ 4 of 5 files skipped = **80% less data scanned**

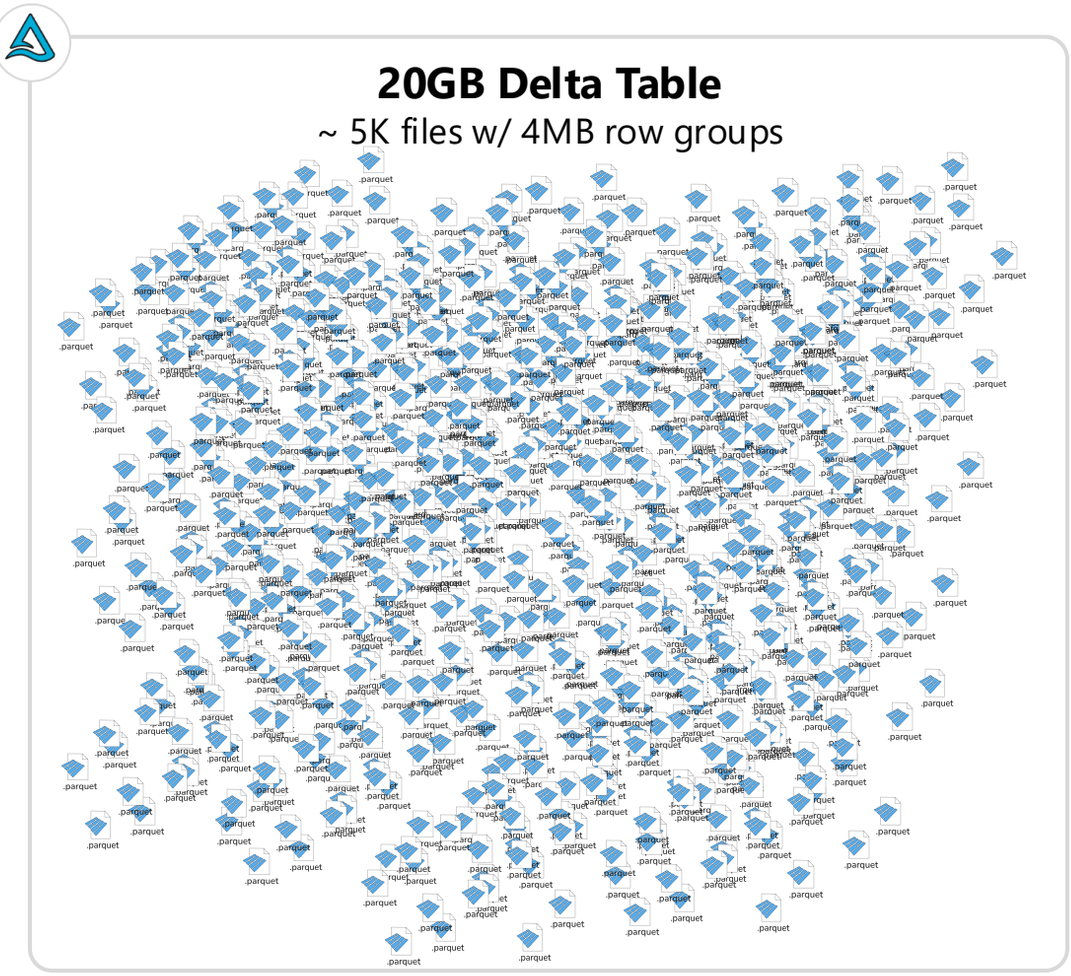
Min/max stats in the Delta log enable this optimization

©Microsoft Corporation

Is more always better?



VS



©Microsoft Corporation

Why File Size Matters



Delta **File Skipping** works at the **file level**

Too few files hurts file skipping potential



CPU cores read a **single file** at a time

Too many files incurs excessive I/O cycles to read the data

©Microsoft Corporation

Module 1
Performance Diagnostics

Module 2
Delta Table Design and Optimization

What do we want to optimize for?

Delta table optimization is centered around **minimizing I/O** throughout the data lifecycle. To do this we need to consider:

File size

- Drives grain of file skipping. If all data is in 1 file, no skipping is possible.
- Drives grain of files impacted by UPDATE / DELETE / MERGE operations.

Physical / logical data organization

- Drives potential for data skipping based on min/max file boundaries.
- Drives compression potential

Level of write amplification

- Impacts the job runtime, more data rewritten == longer running jobs.
- Impacts cache hit rates for repeat reads.

©Microsoft Corporation

2A — File Size

Which data layout would be faster for a selective lookup?

20GB Delta Table

1 file w/ 160 - 128MB row groups



VS

20GB Delta Table

~160 files w/ 128MB row groups



©Microsoft Corporation

File Skipping

Read Less Data

Why file skipping matters

- Delta logs store per-file min/max stats for indexed columns
- Engine skips files where predicates fall outside the min/max range
  - ⚠️ File type in **predicate must match column type**!!
  - ⚠️ **CharType() isn't supported** by file skipping!!
  - ⚠️ **BooleanType() isn't supported** by stats collection!!
- Fewer files scanned = faster queries and reduced I/O

Default behavior

- Only the first 32 columns have stats collected
- Stats tracked: min, max, and null count per column per file
- Collecting stats on long strings is costly and adds write overhead

Maximizing column coverage

- Position frequently filtered columns in the first 32 positions
- Move long strings or low-selectivity columns beyond position 32
- Use delta.dataSkippingStatsColumns to explicitly control which columns get stats
- Increase delta.dataSkippingNumIndexedCols for broader default coverage

©Microsoft Corporation

How File Skipping Works

Query: `SELECT * FROM sales WHERE date = '2024-03-15'`

File	Min / Max Stats	Result
File 1	`date [2024-01-01 .. 2024-02-28]` →	**SKIPPED**
File 2	`date [2024-03-01 .. 2024-03-31]` →	**READ ✓**
File 3	`date [2024-04-01 .. 2024-05-31]` →	**SKIPPED**
File 4	`date [2024-06-01 .. 2024-07-31]` →	**SKIPPED**
File 5	`date [2024-08-01 .. 2024-09-30]` →	**SKIPPED**

→ 4 of 5 files skipped = **80% less data scanned**

Min/max stats in the Delta log enable this optimization

```
ALTER TABLE table_name SET TBLPROPERTIES
('delta.dataSkippingStatsColumns' =
'col1,col2')
```

```
ALTER TABLE table_name SET TBLPROPERTIES
('delta.dataSkippingNumIndexedCols' =
'40')
```

```
ALTER TABLE table_name ALTER COLUMN
long_str_col AFTER last_indexed_col
```

Adaptive Target File Size

Put file size on autopilot

No more manual target file size tuning

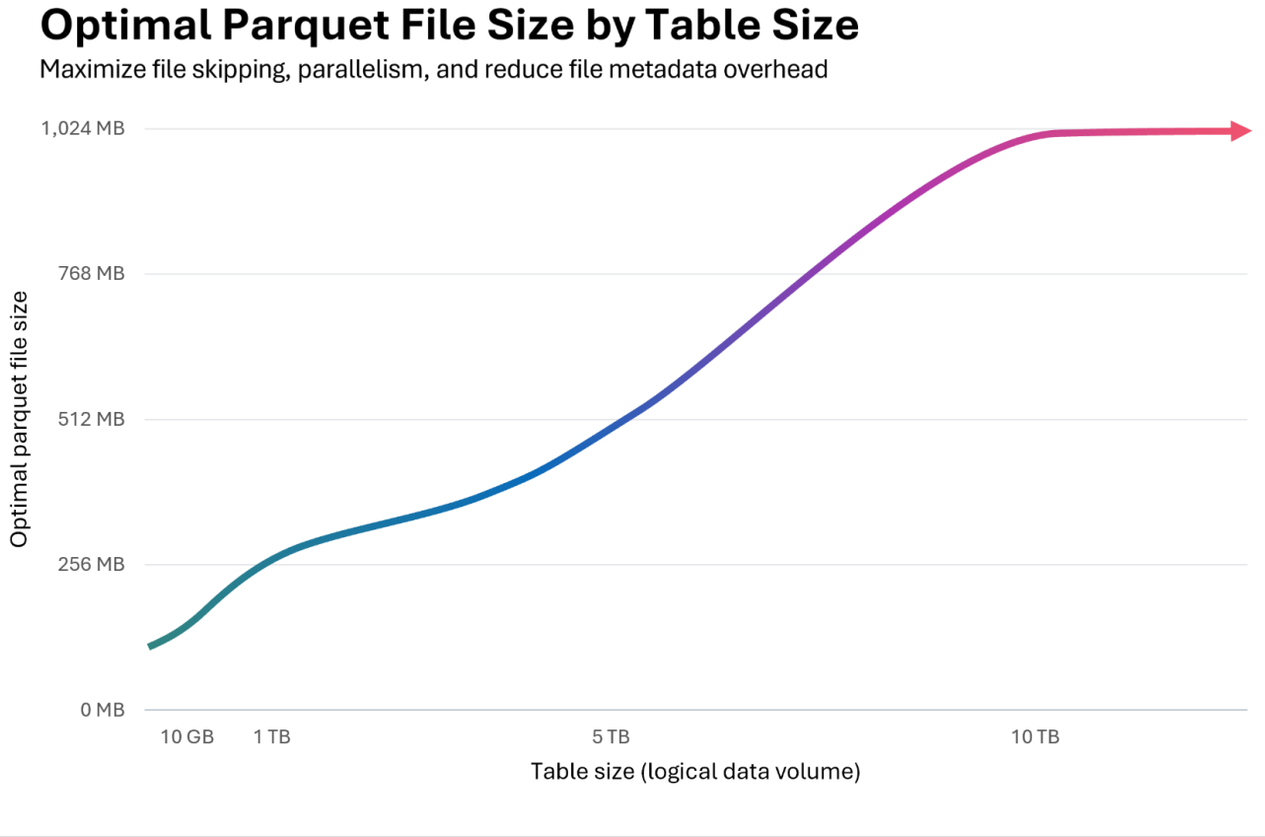
Key features automatically adapt based on table heuristics like table size:

- Optimize
- Auto Compaction
- Optimize Write

Evaluated on initial write operations and start of optimize. Logged as table property.

Better perf & lower write amplification:

- 30-60% faster compaction when a more optimal size is selected
- Up to 8x better file skipping and clustered reads
- Lower MERGE/UPDATE rewrite costs
- Better read & write parallelism




```
1  SET spark.microsoft.delta.targetFileSize.adaptive.enabled = TRUE
```

©Microsoft Corporation

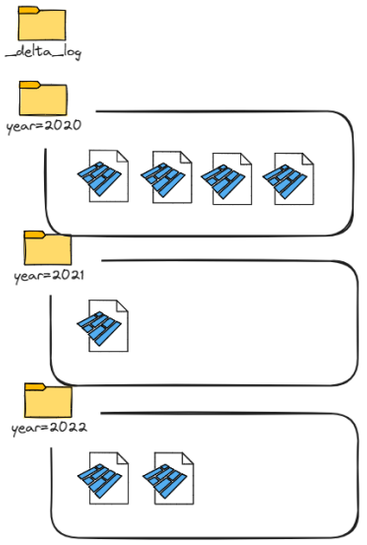
2B — Physical / logical data organization

Partitioning vs. Z-Order vs. Liquid Clustering

DATA organization strategies

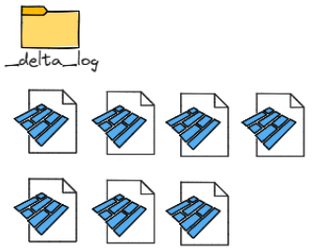
Partitioning

- Physical storage
- Enables more concurrent writes
- Don't use for tables under 1TB
- Easy to cause perf regressions



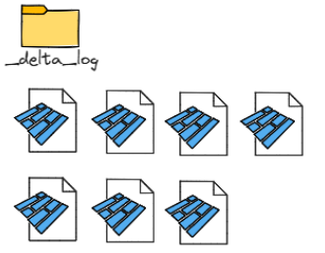
Z-Order

- Clusters data by key for fast selective lookups
- Very costly to maintain not incremental, but you can manually cluster by a predicate
- Transient, no table feature



Liquid Clustering

- Clusters data by key for fast selective lookups
- ~~Costly to maintain. Semi-incremental.~~ Fast in Runtime 2.0
- Table feature



©Microsoft Corporation

2B — Level of write amplification

Optimize Write

Pre-Write Compaction

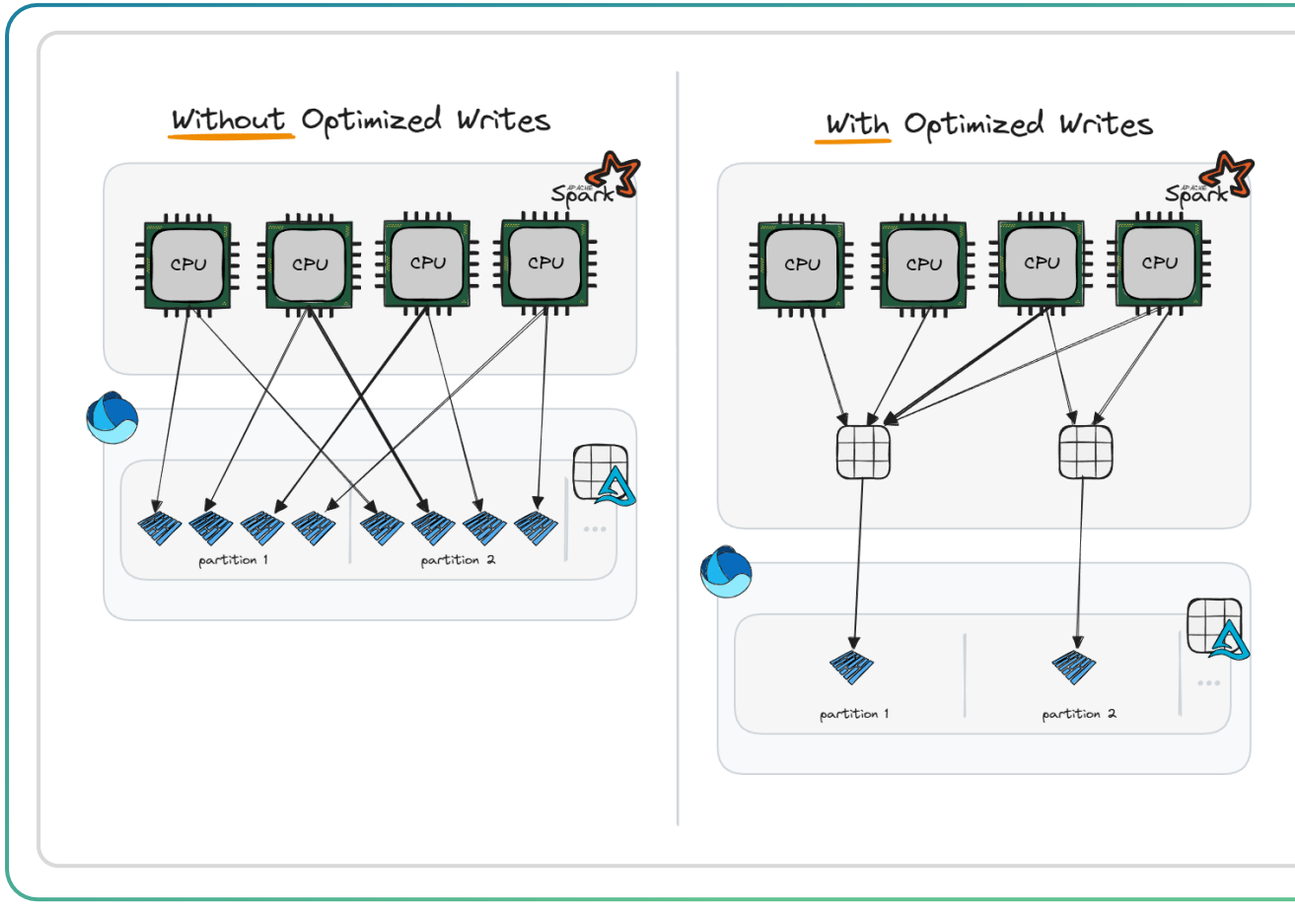
Pre-write compaction is generally less costly than post-write

Always enable for workloads that generate small files:

- Streaming jobs
- Small batch jobs
- Partitioned tables

⚠ **Don't use without Adaptive Target File Size enabled!**

- Having a suboptimal (too large) binSize results in slow writes
- OW is incremental, but binSize is key!




```
SET spark.databricks.delta.optimizeWrite.enabled = TRUE # UNSET is the best default
SET spark.databricks.delta.optimizeWrite.binSize = '128M' # value is a MB ByteString!!
SET spark.databricks.delta.optimizeWrite.partitioned.enabled = TRUE
```

©Microsoft Corporation

Fast Optimize

Faster & More Idempotent

Reduces write-amplification and improves performance via skipping running optimize on suboptimal bins.

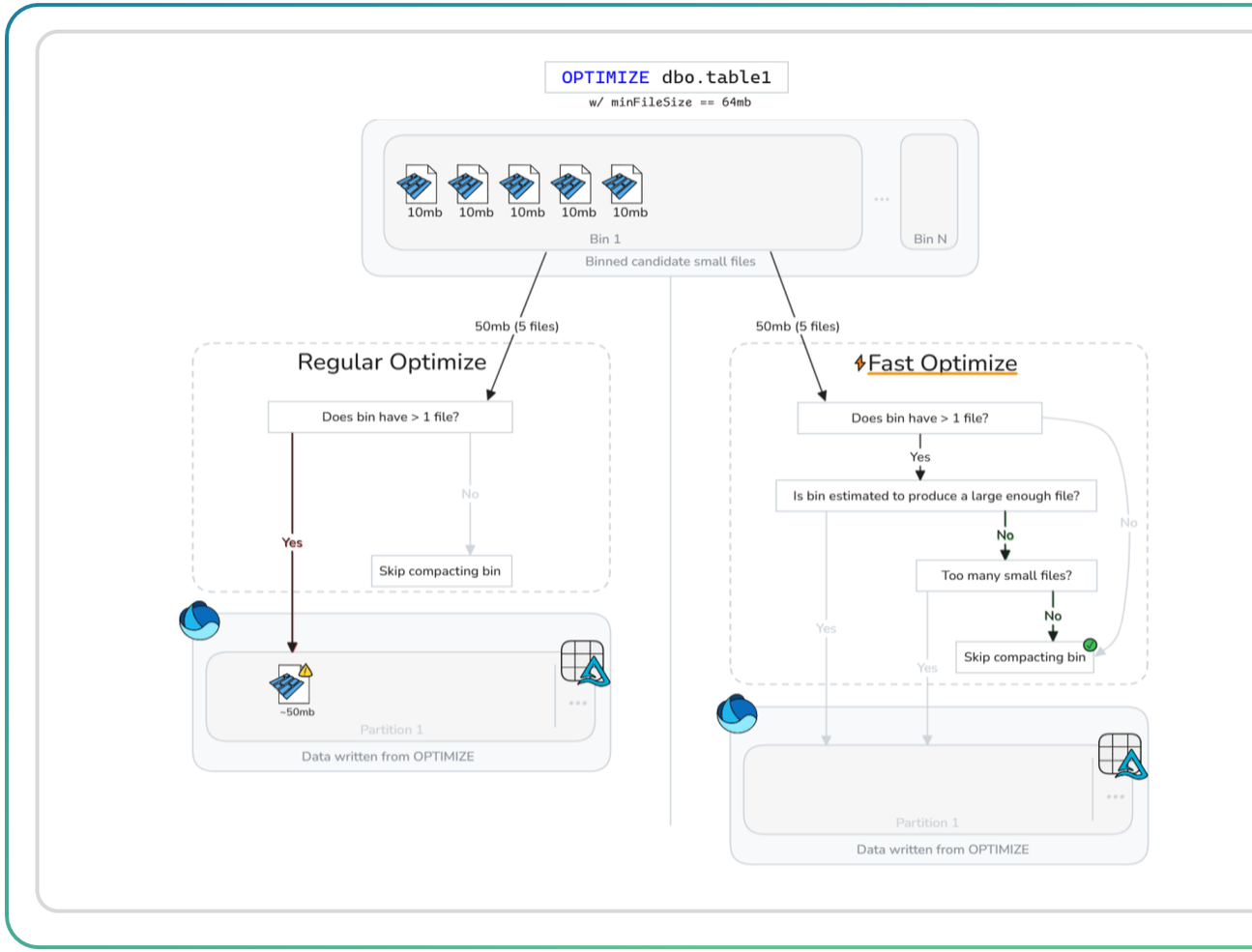
Additional checks:

- Is the bin of small files estimated to produce a large enough file that won't be compacted again?
- Too many small files?

Optimize can be run more frequently and will idempotently only compact bins when estimated to optimal.

Faster Optimize jobs

Less write amplification




```
SET spark.microsoft.delta.optimize.fast.enabled = TRUE
```

©Microsoft Corporation

Fast Optimize

Faster & More Idempotent

Reduces write-amplification and improves performance via skipping running optimize on suboptimal bins.

Additional checks:

- Is the bin of small files estimated to produce a large enough file that won't be compacted again?
- Too many small files?

Optimize can be run more frequently and will idempotently only compact bins when estimated to optimal.

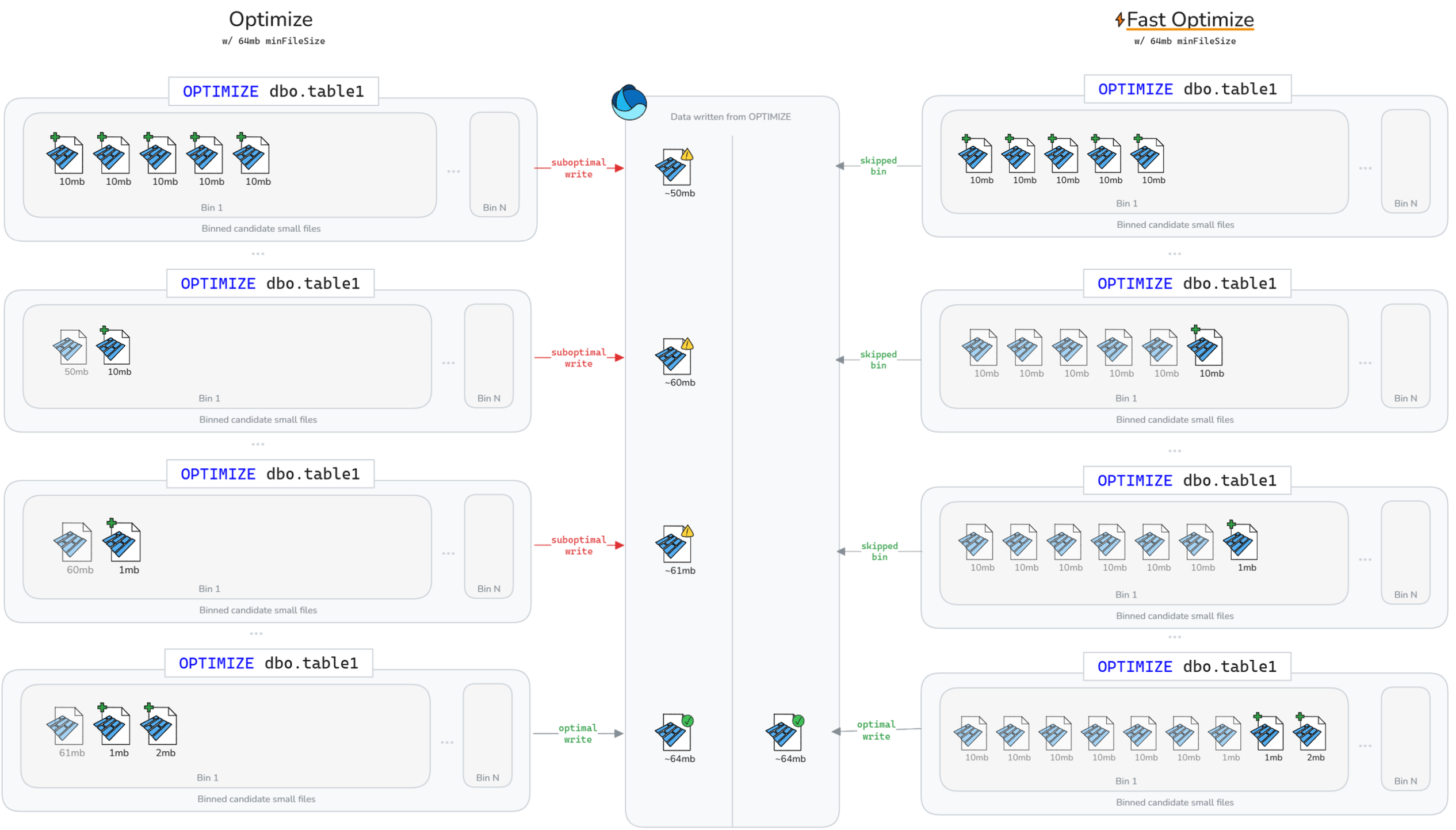
Faster Optimize jobs

Less write amplification

```
SET spark.microsoft.delta.optimize.fast.enabled = TRUE
```



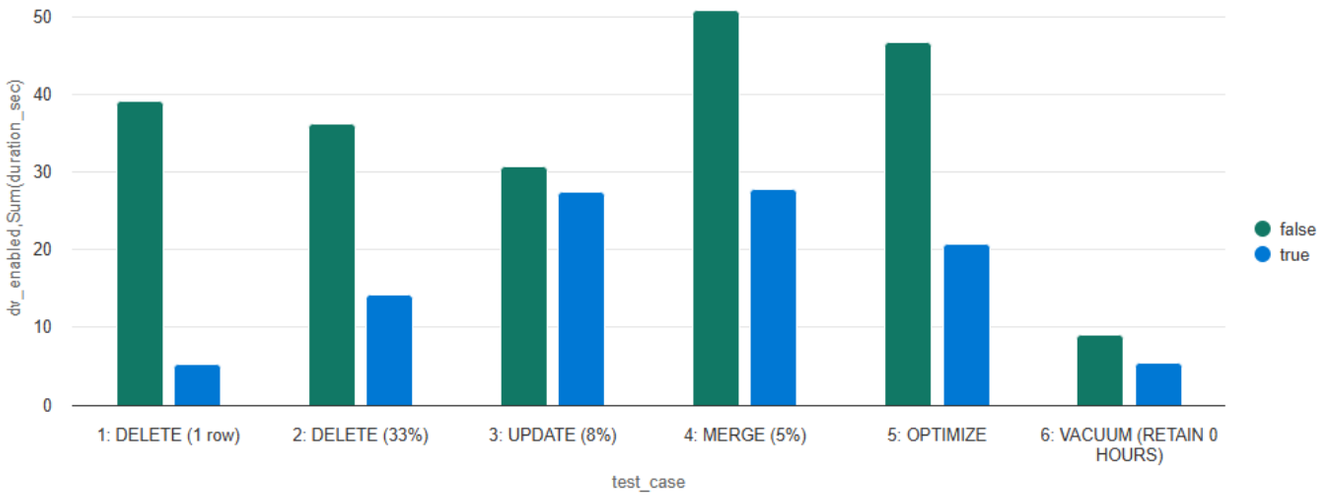
©Microsoft Corporation


Optimize
w/ 64mb minFileSize
Fast Optimize
w/ 64mb minFileSize
OPTIMIZE dbo.table1
10mb
10mb
10mb
10mb
10mb
Bin 1
Bin N
Binned candidate small files
suboptimal write
Data written from OPTIMIZE
~50mb
skipped bin
OPTIMIZE dbo.table1
50mb
10mb
Bin 1
Bin N
Binned candidate small files
suboptimal write
~60mb
skipped bin
OPTIMIZE dbo.table1
10mb
10mb
10mb
10mb
10mb
10mb
Bin 1
Bin N
Binned candidate small files
OPTIMIZE dbo.table1
60mb
1mb
Bin 1
Bin N
Binned candidate small files
suboptimal write
~61mb
skipped bin
OPTIMIZE dbo.table1
10mb
10mb
10mb
10mb
10mb
10mb
1mb
Bin 1
Bin N
Binned candidate small files
OPTIMIZE dbo.table1
61mb
1mb
2mb
Bin 1
Bin N
Binned candidate small files
optimal write
~64mb
~64mb
optimal write
OPTIMIZE dbo.table1
10mb
10mb
10mb
10mb
10mb
10mb
1mb
1mb
2mb
Bin 1
Bin N
Binned candidate small files

Deletion Vectors

Write Faster and Rewrite Less Data

- Mechanism that tracks deleted rows without rewriting data files
- Stores delete information as lightweight bitmaps instead of generating new Parquet files
- Speeds up DELETE, UPDATE, and MERGE operations by avoiding costly file rewrites
- Introduces minor read-time overhead since readers must apply the deletion bitmap
- Run PURGE to force merge DVs, but OPTIMIZE does merge DVs when deletes exceed 5% of total



Enable on all <u>new</u> tables

```
1  spark.databricks.delta.properties.defaults.enableDeletionVectors = TRUE
```

Enable on <u>individual</u> OR <u>existing</u> tables

```
1  ALTER TABLE dbo.table SET TBLPROPERTIES ('delta.enableDeletionVectors' = 'true')
```

©Microsoft Corporation

OPTIMIZE, VACUUM, ~~PURGE~~

Table maintenance essentials

OPTIMIZE — Compacts Small Files Into Larger Ones

- Merges small files into target-size files (~1 GB default), reducing file listing overhead and improving read parallelism
- Supports ZORDER BY for co-locating related data; supports WHERE to limit scope
- Non-destructive: old files remain until VACUUM removes them

VACUUM — Removes Orphaned Files Beyond Retention

- Deletes data files no longer referenced by the Delta log AND older than retention threshold
- Default retention: 168 hours (7 days) — aligns with time travel window

PURGE — Rewrites Files to Remove Deletion Vectors

- Optimize automatically does this when > 5% DV rows!
- Physically removes soft-deleted rows marked by Deletion Vectors
- Use after heavy UPDATE/DELETE workloads when DV density degrades read performance
- Order of operations: OPTIMIZE → PURGE (if DVs) → VACUUM

Maintenance Operations

```sql
-- Compact small files
OPTIMIZE schema_name.table_name;


-- Compact all files
OPTIMIZE schema_name.table_name FULL;


-- Compact with Z-Order (if applicable)
OPTIMIZE schema_name.table_name
  ZORDER BY (customer_id, order_date);

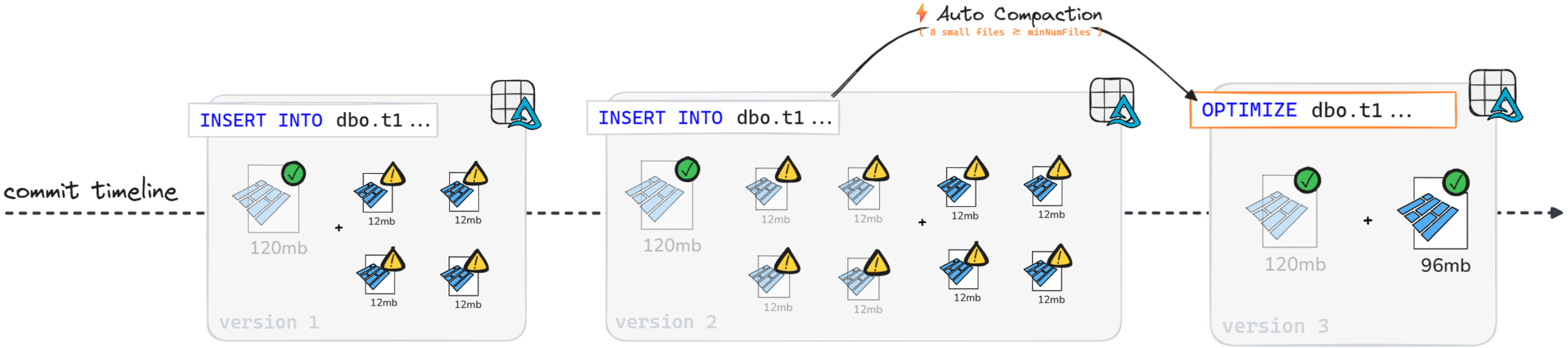

-- Compact small files based on predicate
OPTIMIZE schema_name.table_name
    WHERE date_col > '2026-01-01';


-- Compact all files based on predicate
OPTIMIZE schema_name.table_name FULL
    WHERE date_col > '2026-01-01';
```

©Microsoft Corporation

Auto Compaction

Less Manual Maintenance ➔ Fully Automated




```
1  SET spark.databricks.delta.autoCompact.enabled = TRUE
2  SET spark.databricks.delta.autoCompact.minNumFiles = 50
```

```
1  SET spark.microsoft.delta.autoCompact.onCheckpointOnly.enabled = TRUE
```

Runtime 2.0 ONLY

©Microsoft Corporation

Module 3
Query Optimization Patterns

Module 4
Advanced Debugging

Debugging Storage

Table diagnostics

input_file_name() — Identify Which Files Serve Your Query

- Returns the physical file path for each row — essential for diagnosing small file problems and partition pruning

DESCRIBE DETAIL — Table-Level Physical Metadata

- Shows: format, location, partition columns, table size, number of files, created/modified timestamps
- Use to verify file count, total size, and whether table properties are set correctly

DESCRIBE HISTORY — Full Transaction Log

- Shows every operation (WRITE, MERGE, OPTIMIZE, VACUUM) with timestamp, user, and metrics
- Use to audit maintenance cadence, identify missed OPTIMIZEs, and check VACUUM recency

DESCRIBE EXTENDED — Column-Level Info & Properties

- Shows column data types, nullability, and table properties including Delta features enabled

Delta metadata inspection

```
1  -- Physical metadata: size, file count, location
2  DESCRIBE DETAIL schema_name.table_name;
3
4
5  -- Transaction history: operations, metrics, etc.
6  DESCRIBE HISTORY schema_name.table_name;
7
8
9  -- Column stats and table properties
10 DESCRIBE EXTENDED schema_name.table_name;
```

©Microsoft Corporation

Debugging Execution

Table diagnostics

EXPLAIN — See the physical plan

- Read the textual plan

Spark UI — Visualize the physical plan

- See planned operators, stages, execution metrics, and the textual plan

setJobDescription() — Add *your engineering context* to the Spark UI

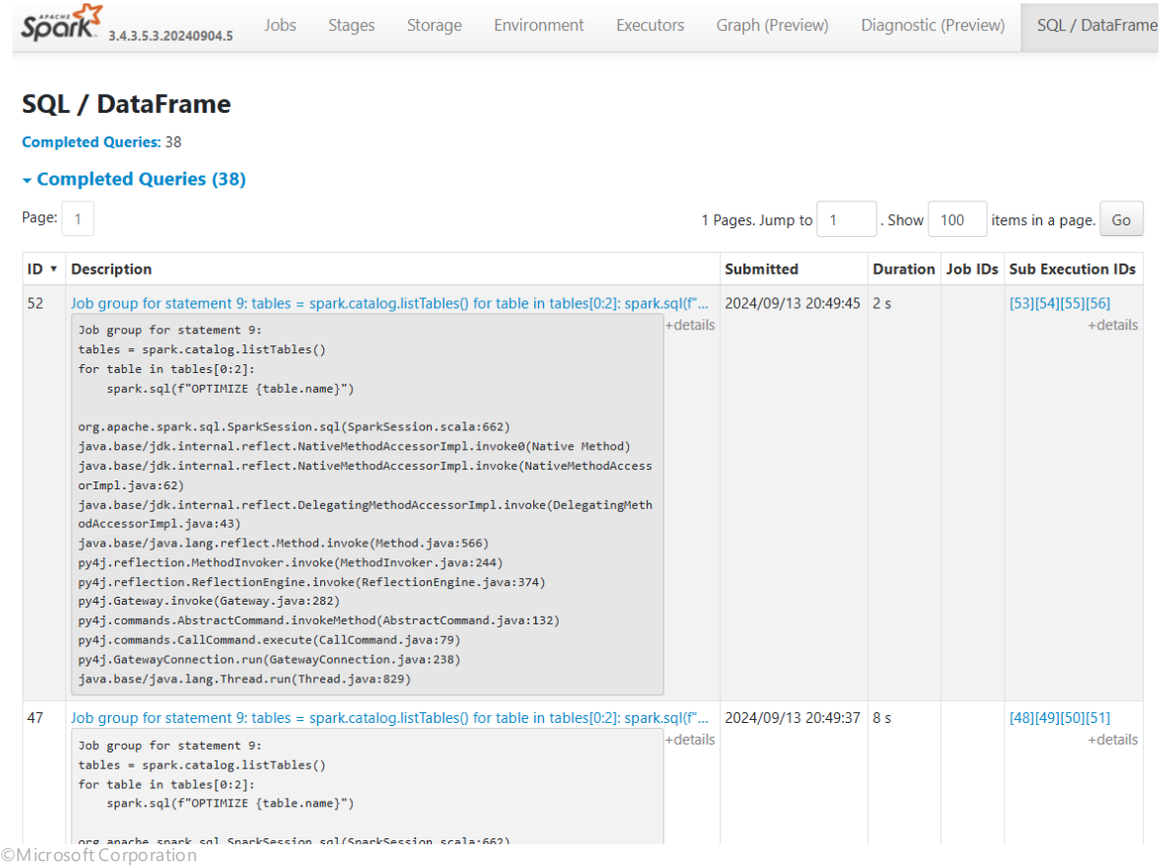
- Use a context manager (optional) to add friendly labels to Spark jobs.

©Microsoft Corporation

setJobDescription()

Coding for Better Observability

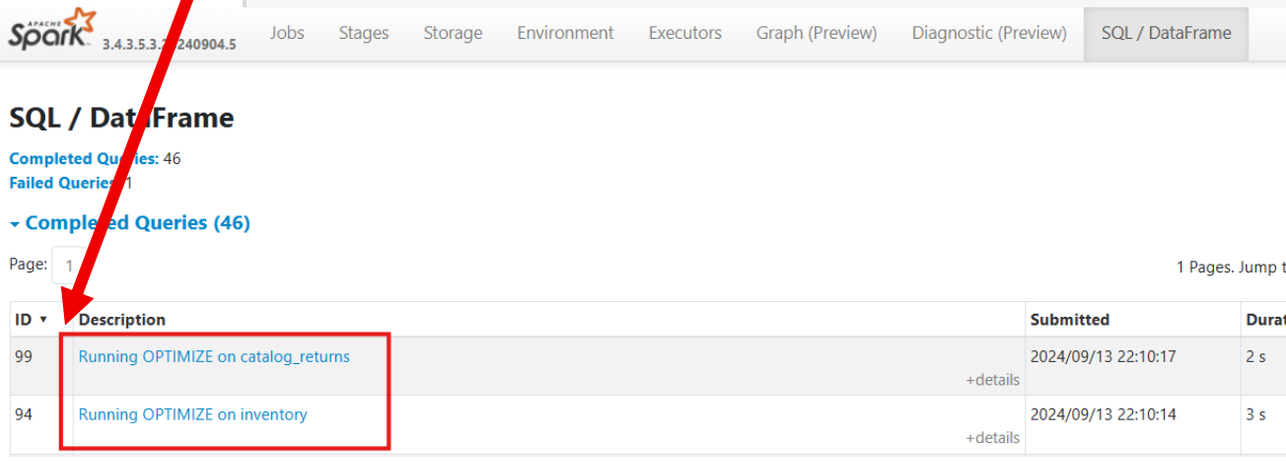
Ever open the Spark UI and have no idea how to find a specific job? 🤔




```
tables = spark.catalog.listTables()
for table in tables:
    # set the description of all following spark
    spark.sparkContext.setJobDescription(
       f"Running OPTIMIZE on {table.name}"
    )

    spark.sql(f"OPTIMIZE {table.name}")

    # disable the user defined description
    spark.sparkContext.setJobDescription(None)
```



©Microsoft Corporation

Microsoft

Thank You

© Copyright Microsoft Corporation. All rights reserved.