To process 25 GB data in spark

1) How many CPU cores are required?

2) How many executors are required?

3) How much each executor memory is required?

4) What is total memory for all executor is required?

① 25 GB → 25 x 1024 MB = 25600 MB

1 Core ⇒ 1 Task

Spark - maxBytesPerPartition → 128 MB

No. of Partitions = 25600 / 128 = 200 → 200 Task.

No. of CPU cores = No. of Partition = 200

↳ by default Spark creates each partition for each block (128M)

We can increase the Number of Bytes per partition.

② How many executors will be required to process 25 GB of data?

Based on research's ideal number of maximum core for each executor is

2 - 5 → 4

Executor → 2 - 5 cores

Avg CPU cores for each executor = 4

Total.no of executors = 200 / 4

③ How much each executor memory is required to process 25 GB of data.

CPU Core for each executor = 4

Memory for each executor = 4 x 512 MB

= 2 GB

Expected memory for each core =

4 x default Partition Size

4 x 128 MB

= 512 MB

(4) What is the total memory required to process 25 GB?

Total No of executor = 50

Memory for each = 2 GB.

Total memory for all executors = 100 GB

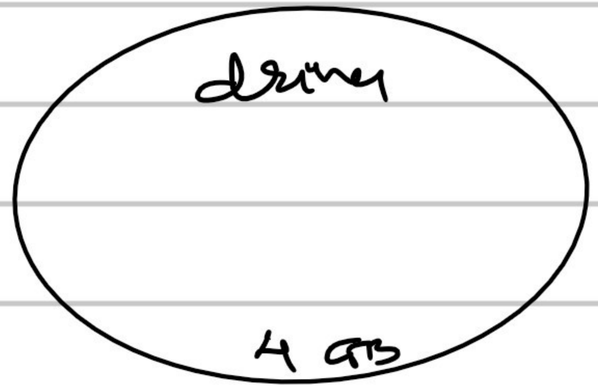
Executor memory is not less than 1.5 times of Spark reserved memory — Single Core executor memory should not be less than 450 MB

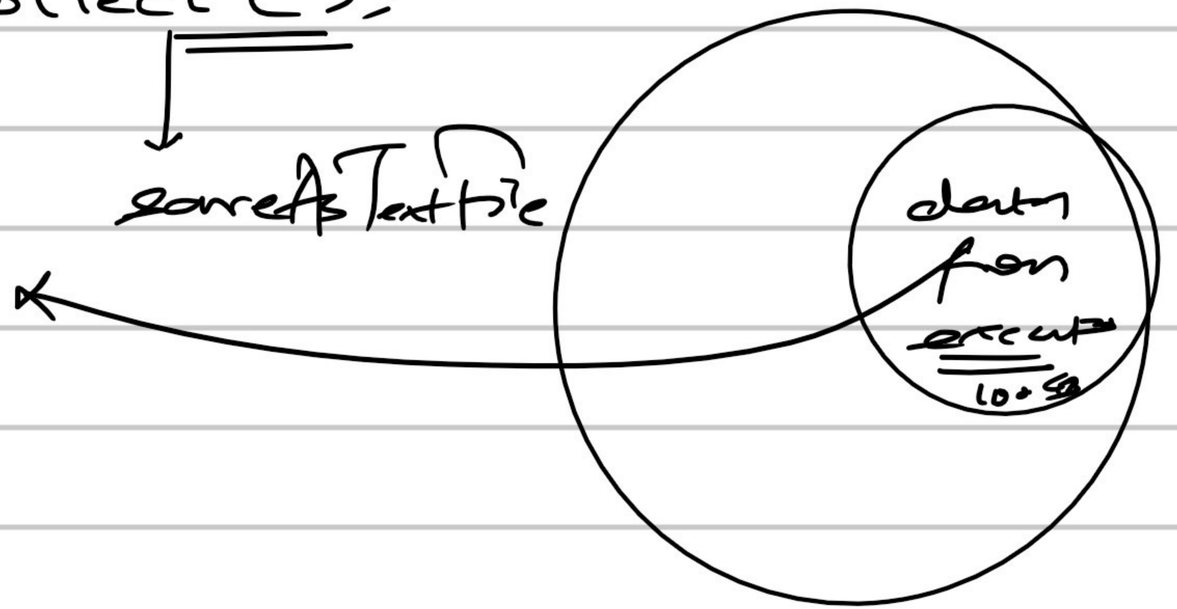
1.5 x 300 m = 450 MB

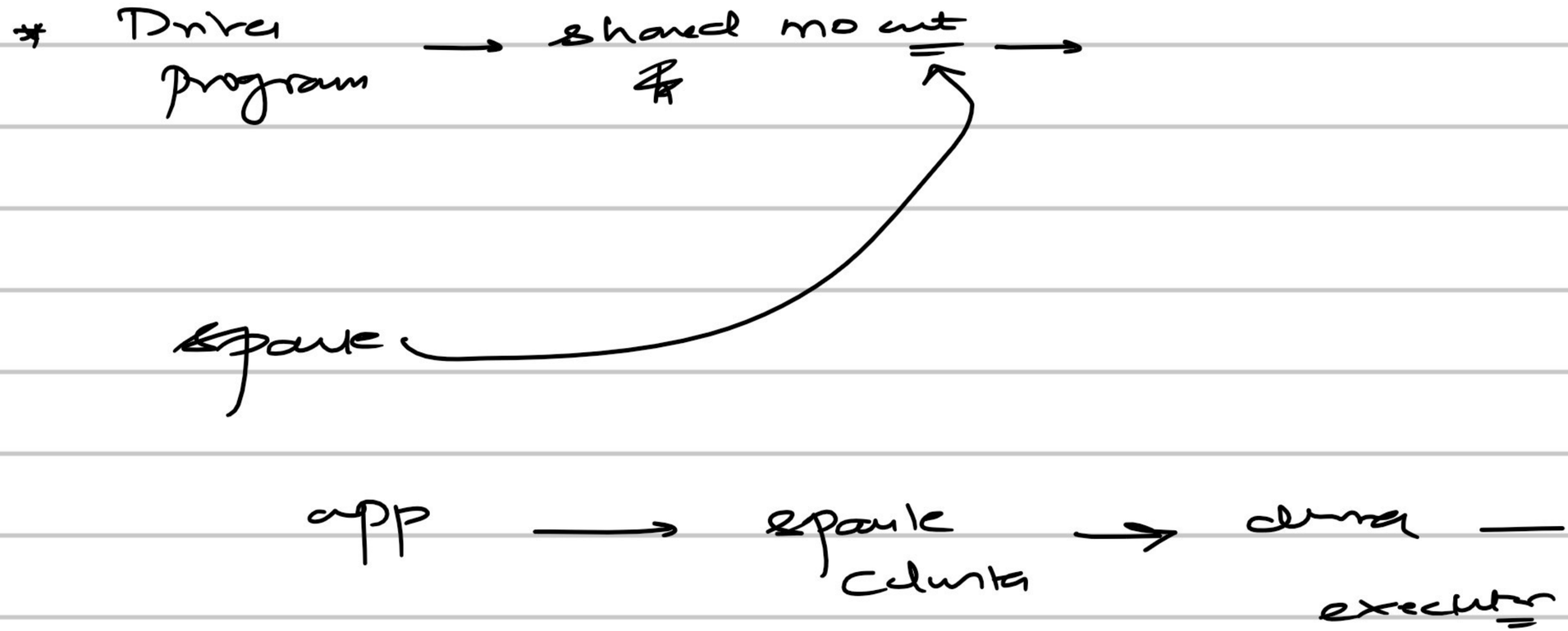
⑤ How much memory & core to be allocated for driver ?

driver → 2 cores → 2 GBs.

Out-of-memory → collect();



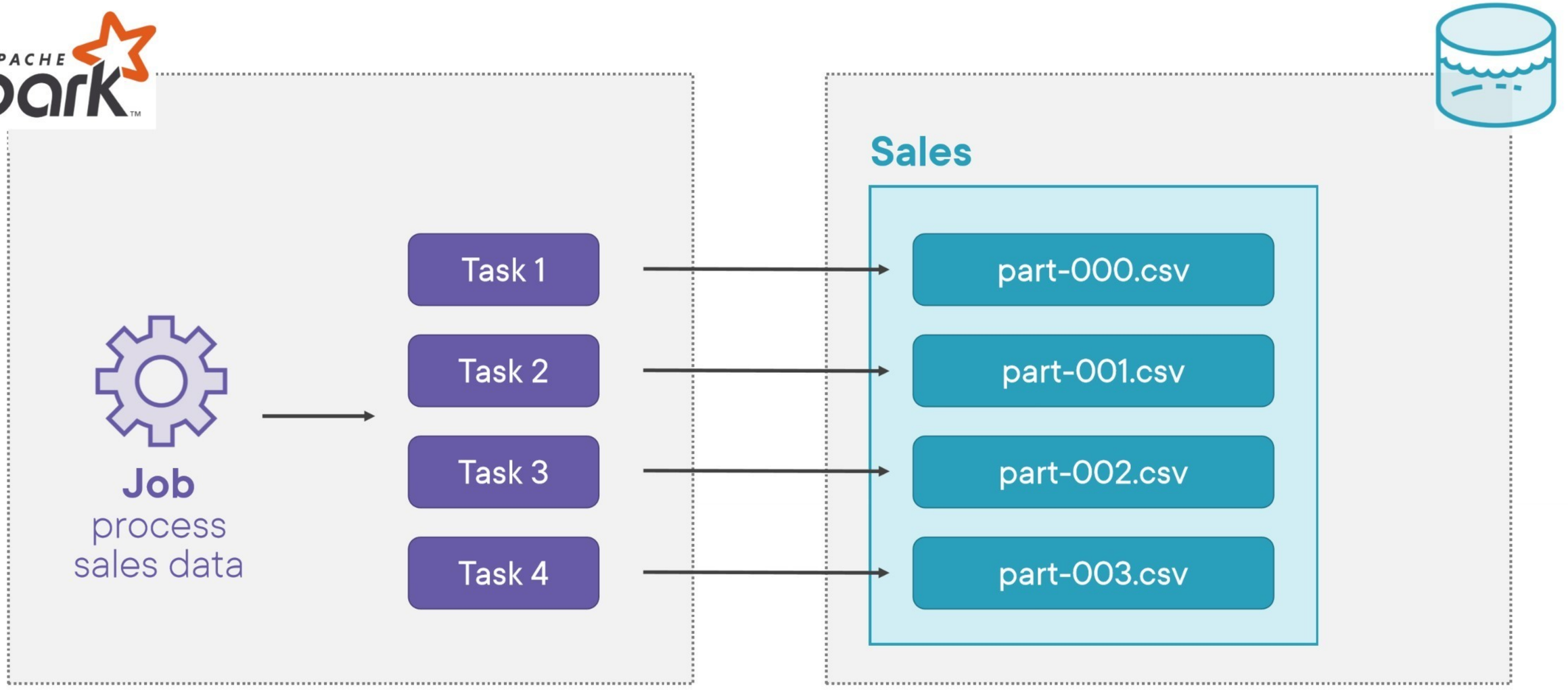



Driver Program
shared mount
Spark
app
Spark Context
driver executor

Need for Delta Lake with Spark

Data Lake is a central repository to store all types of data at any scale, in the form of files

How Spark Writes to Data Lake?



ACID Guarantees on Transactions

Each transaction in a Database provides ACID guarantees

ACID Guarantees on Transactions

Atomicity

All or no changes are written

Id	Name	Salary
1	A	10000
2	B	17000
3	C	12000

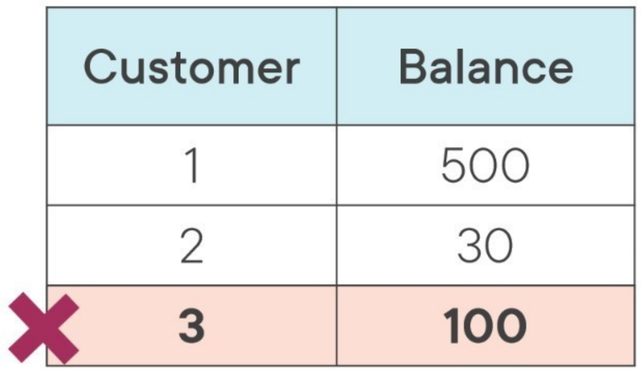
4	D	11000
5	E	14000

ACID Guarantees on Transactions

Consistency

Data always remain
in valid state

Withdraw 200
where Customer = 3



Customer	Balance
1	500
2	30
3	**100**

Constraint
Balance >= 0

ACID

Guarantees on Transactions

Isolation

Transaction must run isolated from other processes

Writer

4	D
5	E

Reader

Id	Name
1	A
2	B
3	C

ACID Guarantees on Transactions

Durability

Once committed, data persists even if system fails

Id	Name
1	A
2	B
3	C

Data Lake does NOT provide
ACID guarantees

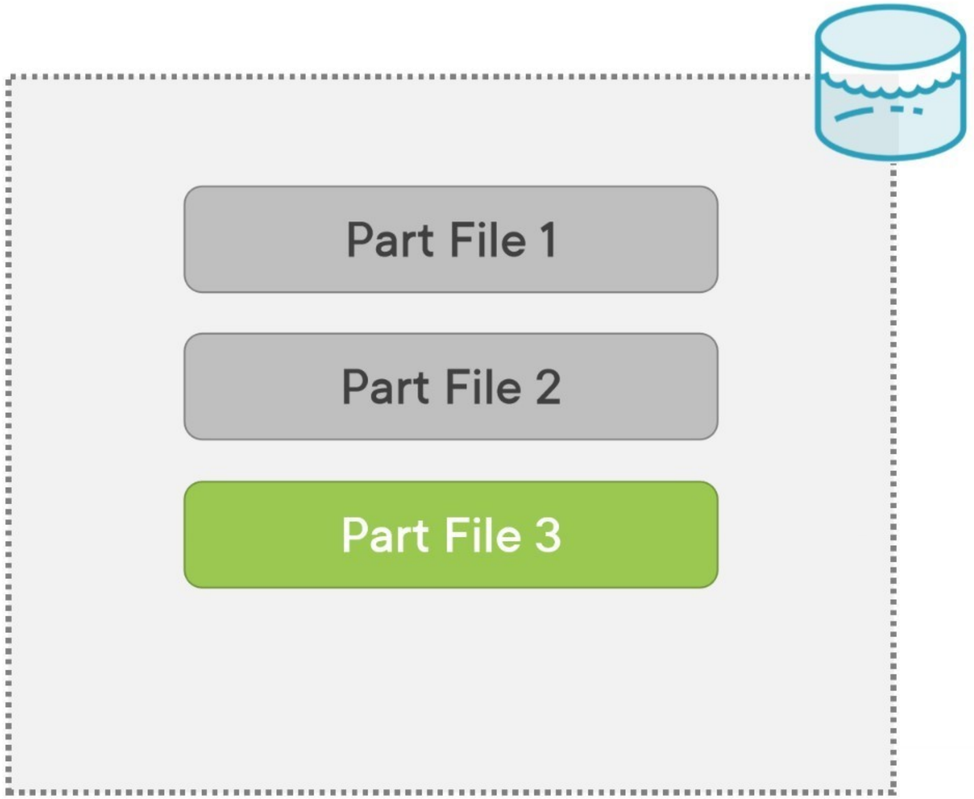
1 – Job Failure in Appending Data

Writer

Append 2 more part files using 2 tasks

Job failed with runtime error – Part file 4 could not be written

Part File 4 ✗



Reader

Read the folder (reads part files 1, 2 & 3)

Reads inconsistent data

Breaks Atomicity and Consistency!

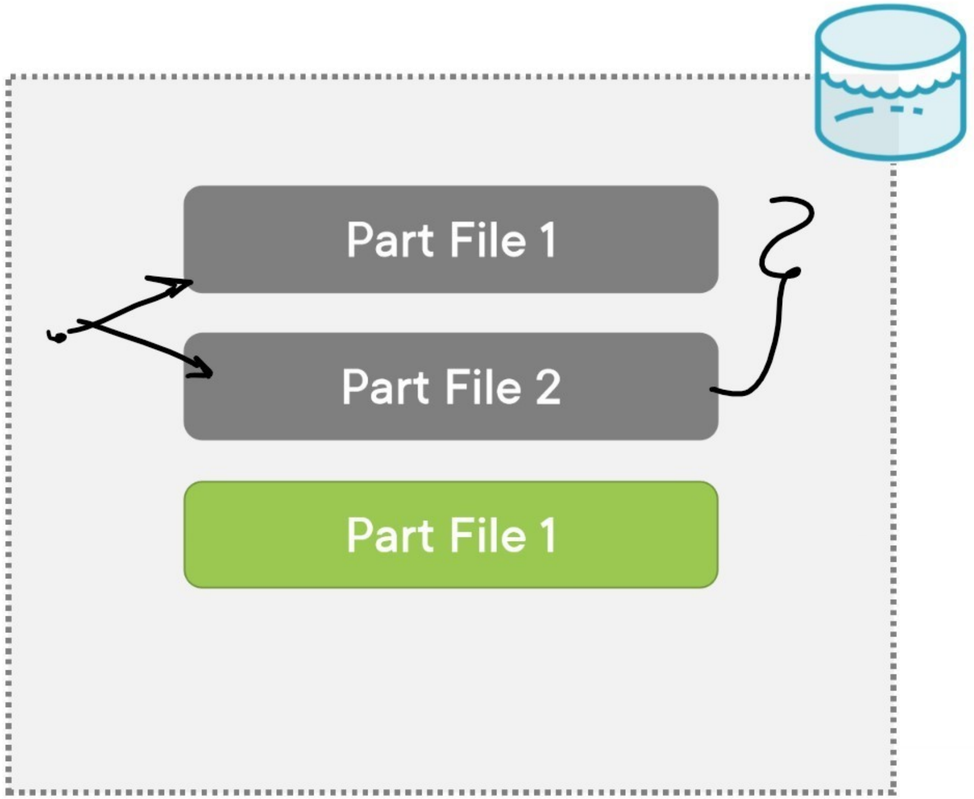
2 – Job Failure in Overwriting Data

Writer

Overwrite existing =
Delete existing
+ Write 2 new part files

Job failed with runtime error – New part file 2 could not be written

Part File 2 ✖



Reader

Read the folder
(reads part file 1 only)

Reads inconsistent data
+ Previous data is lost

Breaks Atomicity, Consistency and Durability!

3 – Simultaneous Read / Write

Writer

Write 2 more part files using 2 tasks

Only Part file 3 is written. Part 4 is still getting processed

Part File 4

Part File 1

Part File 2

Part File 3

Reader

Read the folder while writing is in progress (reads part files 1, 2 & 3)

Reads inconsistent data

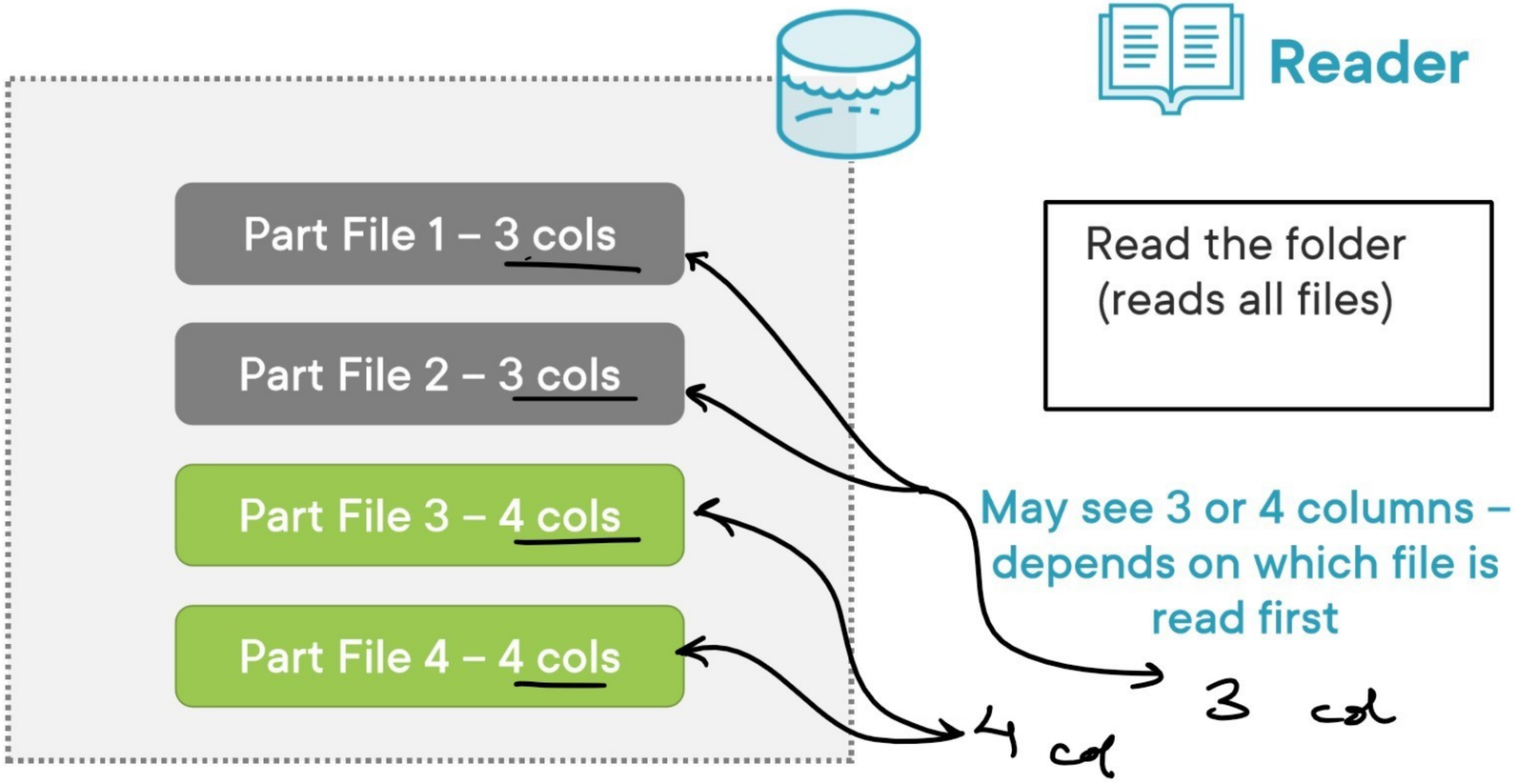
Breaks Consistency and Isolation!

4 – Appending Data with New Schema

Writer

Write 2 more part files with 4 output columns

New output files have 4 columns. No schema validation before writing



Breaks Consistency!

Challenges with Data Lake

Data reliability issues

- Data corruption because of failures – no rollback!
- No data validation
- Consistency issues while reading data

Handling Batch and Streaming data together is tough

No updates / deletes / merge on files

- Difficult to implement GDPR / CCPA compliance

Data quality issues

- Schema isn't verified before writing
- Cannot apply checks on data

Query performance issues

Difficult to maintain historical versions of data

Delta Lake can help us solve these challenges!

But is Delta Lake the only option?

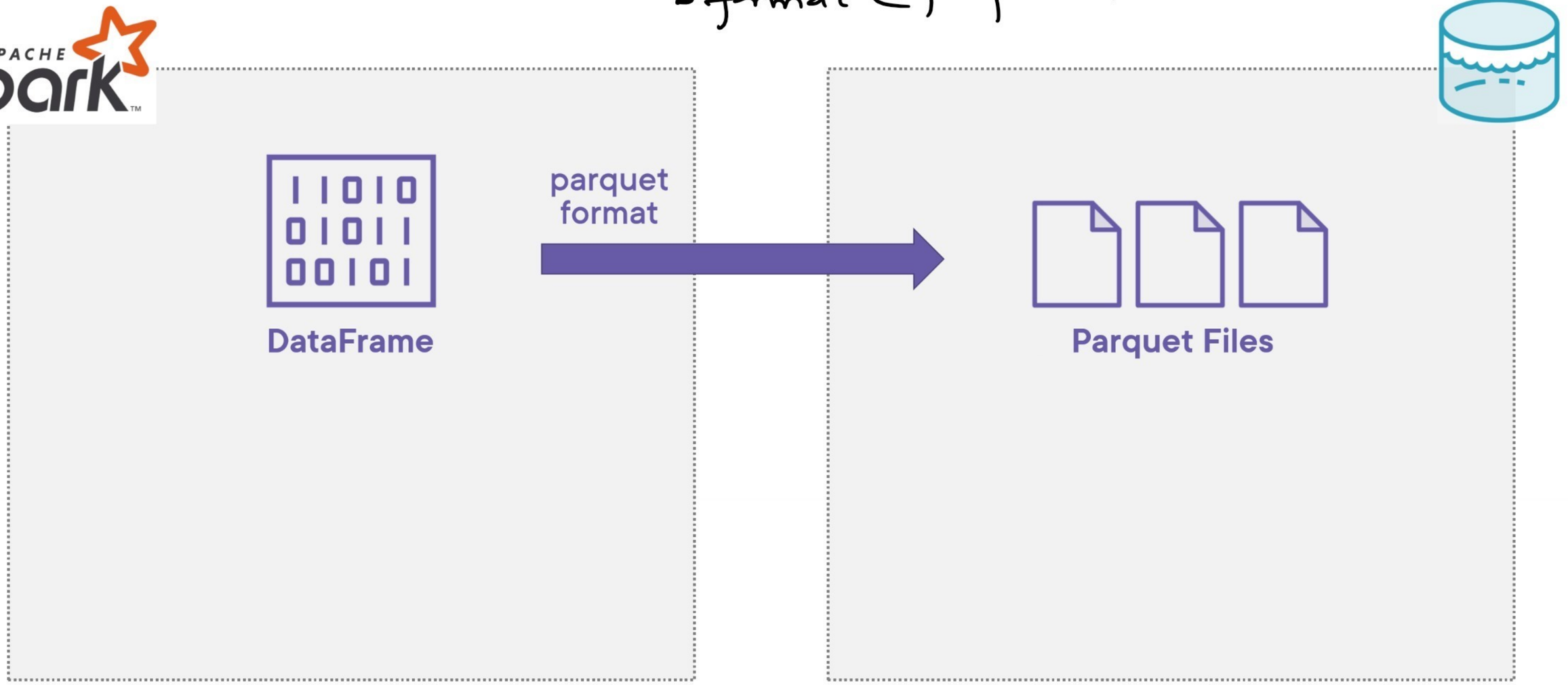
Apache Iceberg
Apache Hudi

How Delta Lake Works?

Delta Lake is an open-source storage layer that brings reliability to Data Lakes

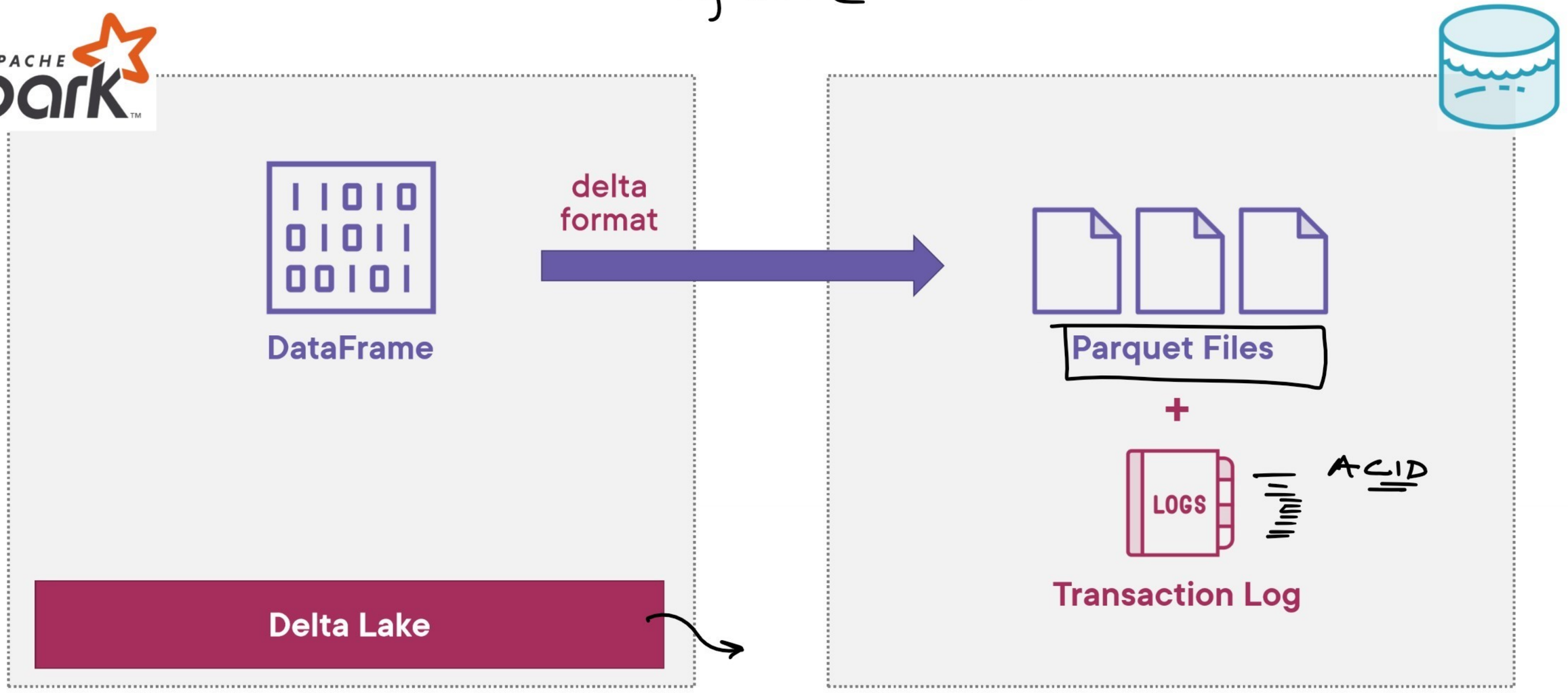
Writing Data in Parquet Format

• format ("parquet");



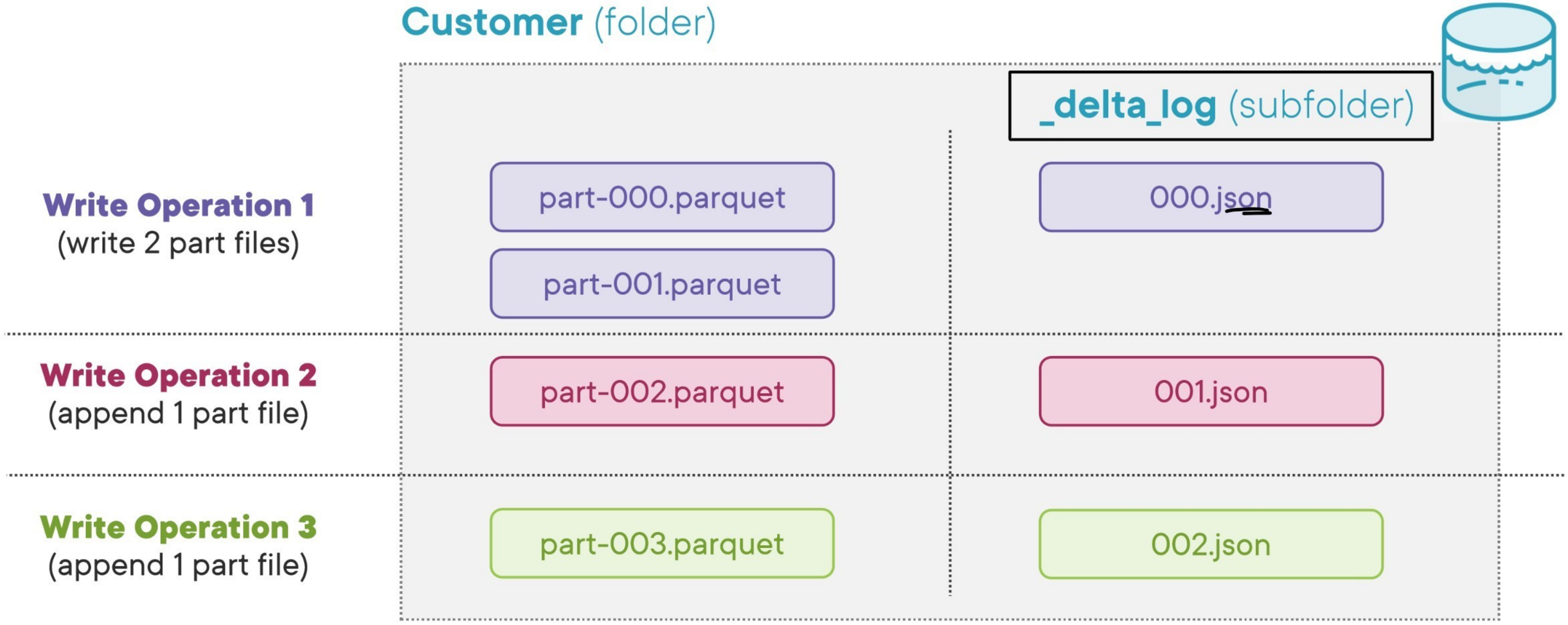
Writing Data in Delta Format

• format("delta");



```
(
 dataframe
      .write
      .format("delta")    # like other formats - csv, parquet etc.
      .save(filepath)
)
```

Save DataFrame in Delta Format



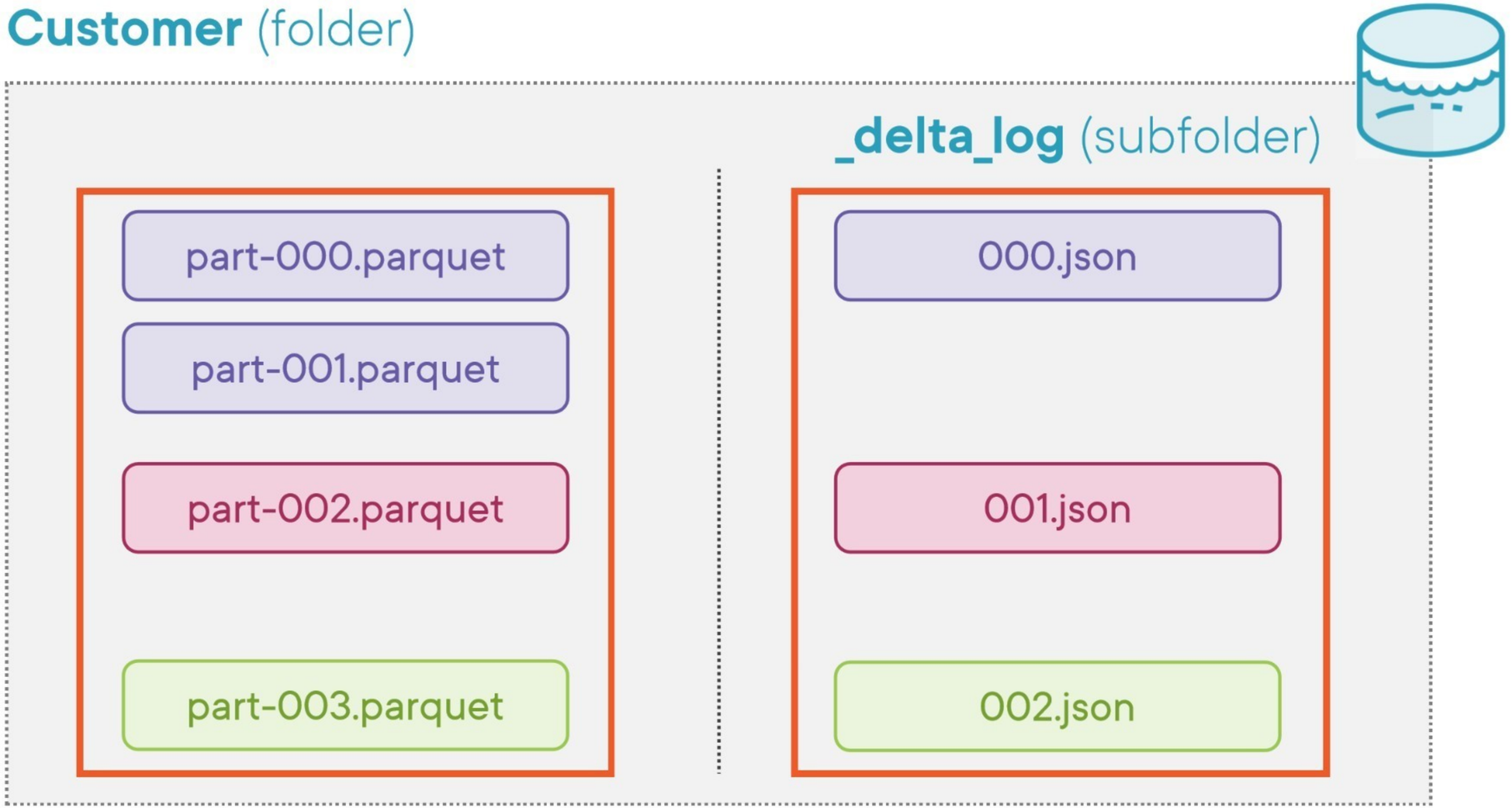
For each write operation:

- Part files are written first
- A transaction log file is added to _delta_log folder in JSON format

Read Operation

Reads log files - 000.json, 001.json, 002.json

Reads 4 part files, based on log information



For each read operation:

- Transaction log files are read first
- Part files are then read based on log files

Customer (folder)

_delta_log (subfolder)

Create Operation
(with 2 part files)

Id	Name
1	A
2	B

Part 1

Id	Name
3	C
4	D

Part 2

Operation	File Name
Add	Part 1
Add	Part 2

000.json

Insert Operation
(append 1 part file)

Id	Name
5	E
6	F

Part 3

Operation	File Name
Add	Part 3

001.json

Update Operation
(Change name from
A to AA where Id = 1)

Id	Name
1	AA
2	B

Part 4

Add modified row

Copy
unchanged rows

Operation	File Name
Remove	Part 1
Add	Part 4

002.json

Customer (folder)

Read all records

~~**Part 1**~~

Part 2

Part 3

Part 4

Id	Name
1	A
2	B

Part 1

Id	Name
3	C
4	D

Part 2

Id	Name
5	E
6	F

Part 3

Id	Name
1	AA
2	B

Part 4

_delta_log (subfolder)

Operation	File Name
Add	Part 1
Add	Part 2

000.json

Operation	File Name
Add	Part 3

001.json

Operation	File Name
Remove	Part 1
Add	Part 4

002.json

Customer (folder)

Read all records

Id	Name
3	C
4	D
5	E
6	F
1	AA
2	B

Id	Name
1	A
2	B

Part 1

Id	Name
3	C
4	D

Part 2

Id	Name
5	E
6	F

Part 3

Id	Name
1	AA
2	B

Part 4

_delta_log (subfolder)

Operation	File Name
Add	Part 1
Add	Part 2

000.json

Operation	File Name
Add	Part 3

001.json

Operation	File Name
Remove	Part 1
Add	Part 4

002.json

Delta Lake Features

Provides ACID Guarantees

- No data corruption because of failures
- Data consistency while reading data

Perform inserts / updates / deletes

Schema enforcement and evolution

Protect data using Time Travel

Handle batch & streaming data together

Apply data quality checks

Performance improvements using statistics

...and much more

ACID Guarantees on Delta Lake

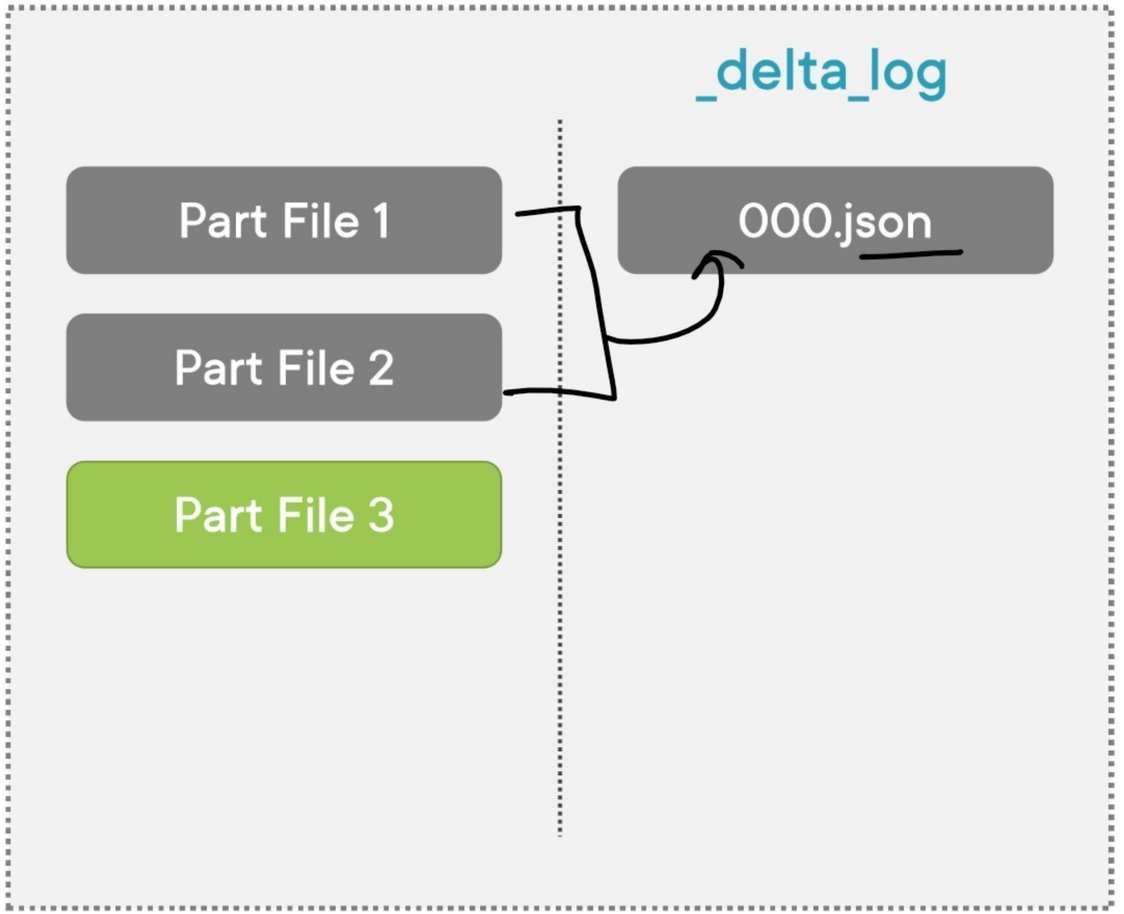
1 – Job Failure in Appending Data

Writer

Append 2 more part files using 2 tasks

Job failed with runtime error – Part file 4 could not be written

Part File 4 ✖



Reader

Reads log file (000.json)

Only reads Part files 1 & 2

Reads consistent data

2 – Job Failure in Overwriting Data

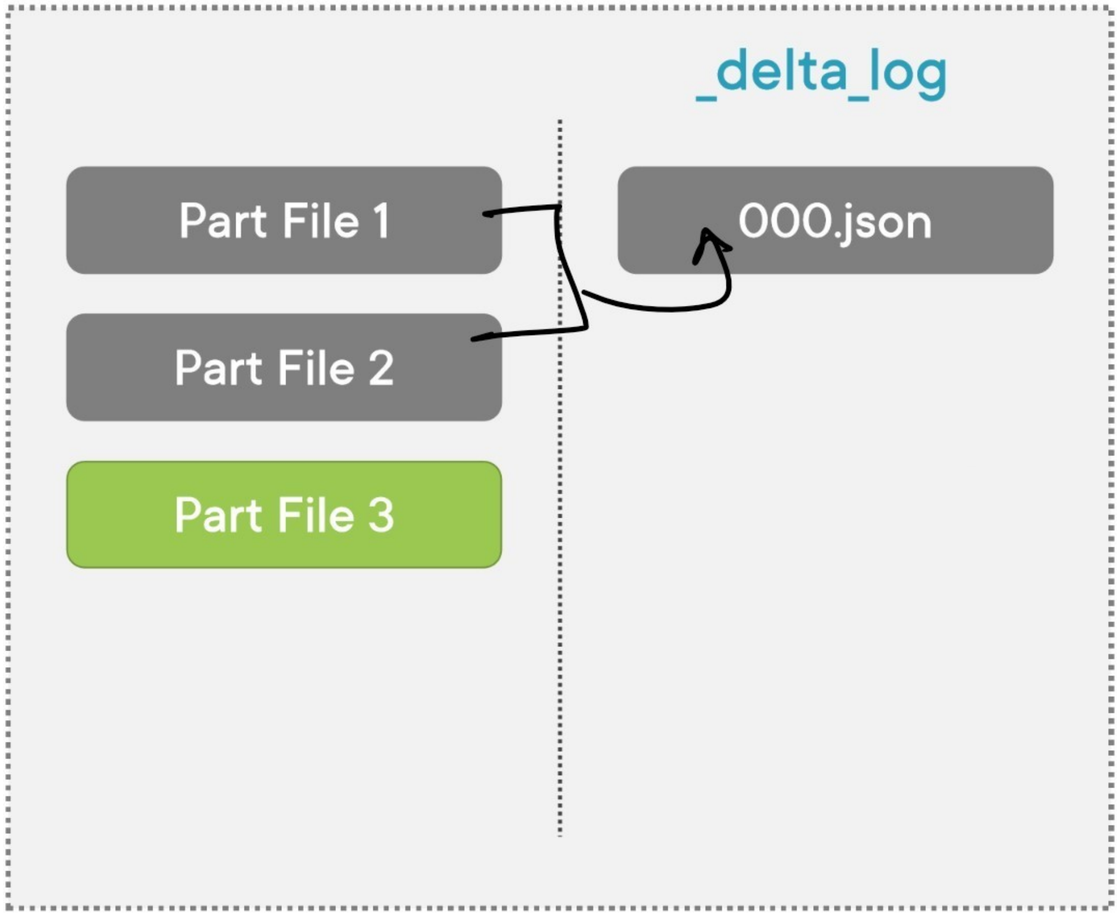
Writer

Overwrite existing data by writing 2 new part files

Existing files are not deleted!

Job failed with runtime error – New part file 4 could not be written

Part File 4 ✖



Reader

Reads log file (000.json)

Only reads Part files 1 & 2

Reads consistent data
+ Previous data is not lost

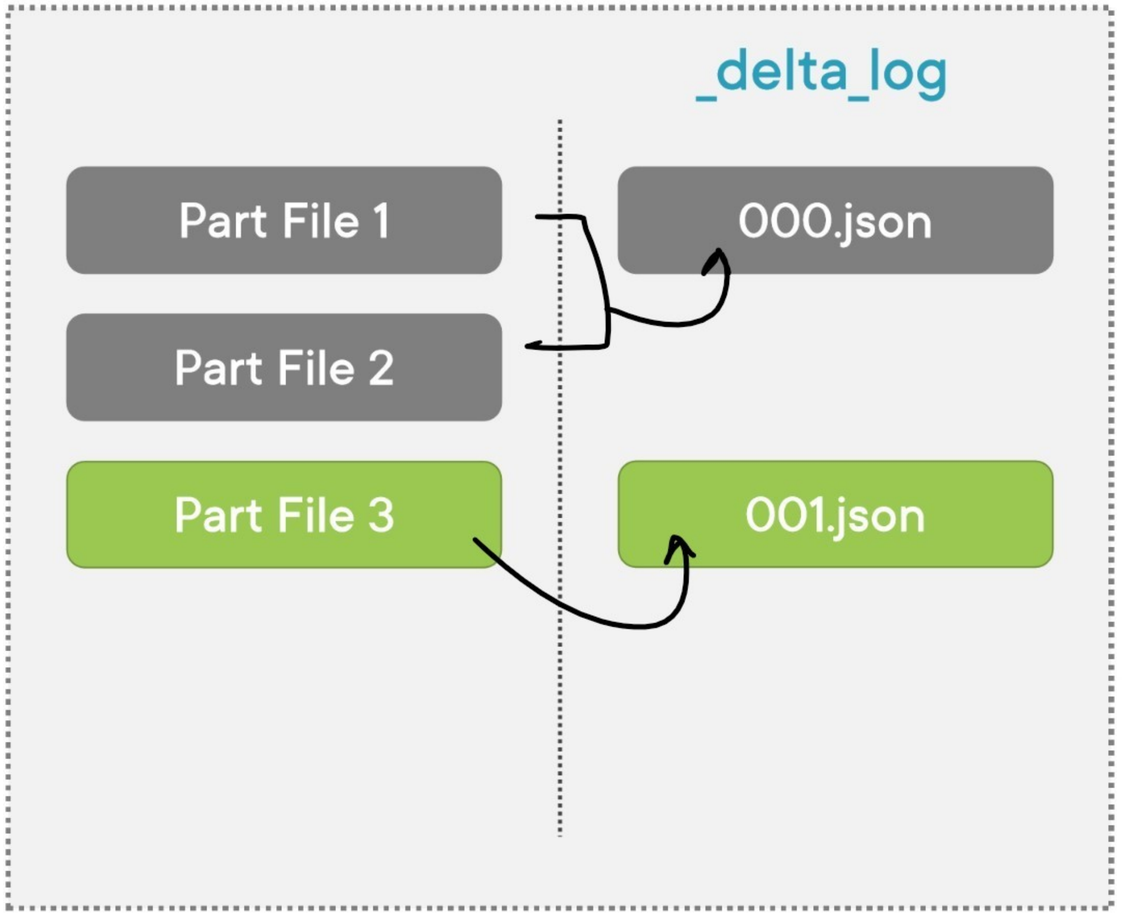
3 – Simultaneous Read / Write

Writer

Write 2 more part files using 2 tasks

Only Part file 3 is written. Part 4 is still getting processed

Part File 4



Reader

Reads log file while writing is in progress

(reads 000.json)

Only reads Part files 1 & 2

No dirty reads while writing is in progress

Creating Delta Tables

Sample Transaction Log Entry


```
{"commitInfo":{"timestamp":1640067393982,"userId":"1472626815582251",
    "operation":"WRITE","operationParameters":{"mode":“Overwrite"},
    "operationMetrics":{"numFiles":"2","numOutputRows":“200","numOutputBytes":“14629"}}}
```

```
{"metaData":{"schemaString":"{
   \"fields\":[
              {\"name\":\"Day\",    \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"RideId\", \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"Amount\", \"type\":\"double\",  \"nullable\":true, \"metadata\":{}}
            ]
}"}}
```

```
{"add":{"path":"part-000.parquet","size":7284 }}
{"add":{"path":"part-001.parquet","size":7345 }}
```

Sample Transaction Log Entry


```
{"commitInfo":{"timestamp":1640067393982,"userId":"1472626815582251",
    "operation":"WRITE","operationParameters":{"mode":"Overwrite"},
    "operationMetrics":{"numFiles":"2","numOutputRows":“200","numOutputBytes":“14629"}}}
```

```
{"metaData":{"schemaString":"{
   \"fields\":[
              {\"name\":\"Day\",    \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"RideId\", \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"Amount\", \"type\":\"double\",  \"nullable\":true, \"metadata\":{}}
            ]
}"}}
```

```
{"add":{"path":"part-000.parquet","size":7284 }}
{"add":{"path":"part-001.parquet","size":7345 }}
```

Sample Transaction Log Entry

```
{"commitInfo":{"timestamp":1640067393982,"userId":"1472626815582251",
    "operation":"WRITE","operationParameters":{"mode":"Overwrite"},
    "operationMetrics":{"numFiles":"2","numOutputRows":“200","numOutputBytes":“14629"}}}
```

```
{"metaData":{"schemaString":"{
   \"fields\":[
              {\"name\":\"Day\",    \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"RideId\", \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"Amount\", \"type\":\"double\",  \"nullable\":true, \"metadata\":{}}
            ]
}"}}
```

```
{"add":{"path":"part-000.parquet","size":7284 }}
{"add":{"path":"part-001.parquet","size":7345 }}
```

Sample Transaction Log Entry

```
{"commitInfo":{"timestamp":1640067393982,"userId":"1472626815582251",
    "operation":"WRITE","operationParameters":{"mode":"Append"},
    "operationMetrics":{"numFiles":"2","numOutputBytes":“14629","numOutputRows":"200"}}}
```

```
{"metaData":{"schemaString":"{
   \"fields\":[
              {\"name\":\"Day\",    \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"RideId\", \"type\":\"integer\", \"nullable\":true, \"metadata\":{}},
              {\"name\":\"Amount\", \"type\":\"double\",  \"nullable\":true, \"metadata\":{}}
            ]
}"}}
```

```
{"add":{"path":"part-000.parquet","size":7284 }}
{"add":{"path":"part-001.parquet","size":7345 }}
```

Inserting Data to Delta Table

Options to Insert Data

INSERT Command
(SQL)

Append DataFrame
(PySpark / Scala)

Performing DML Operations

Applying Table Constraints

Table Constraints

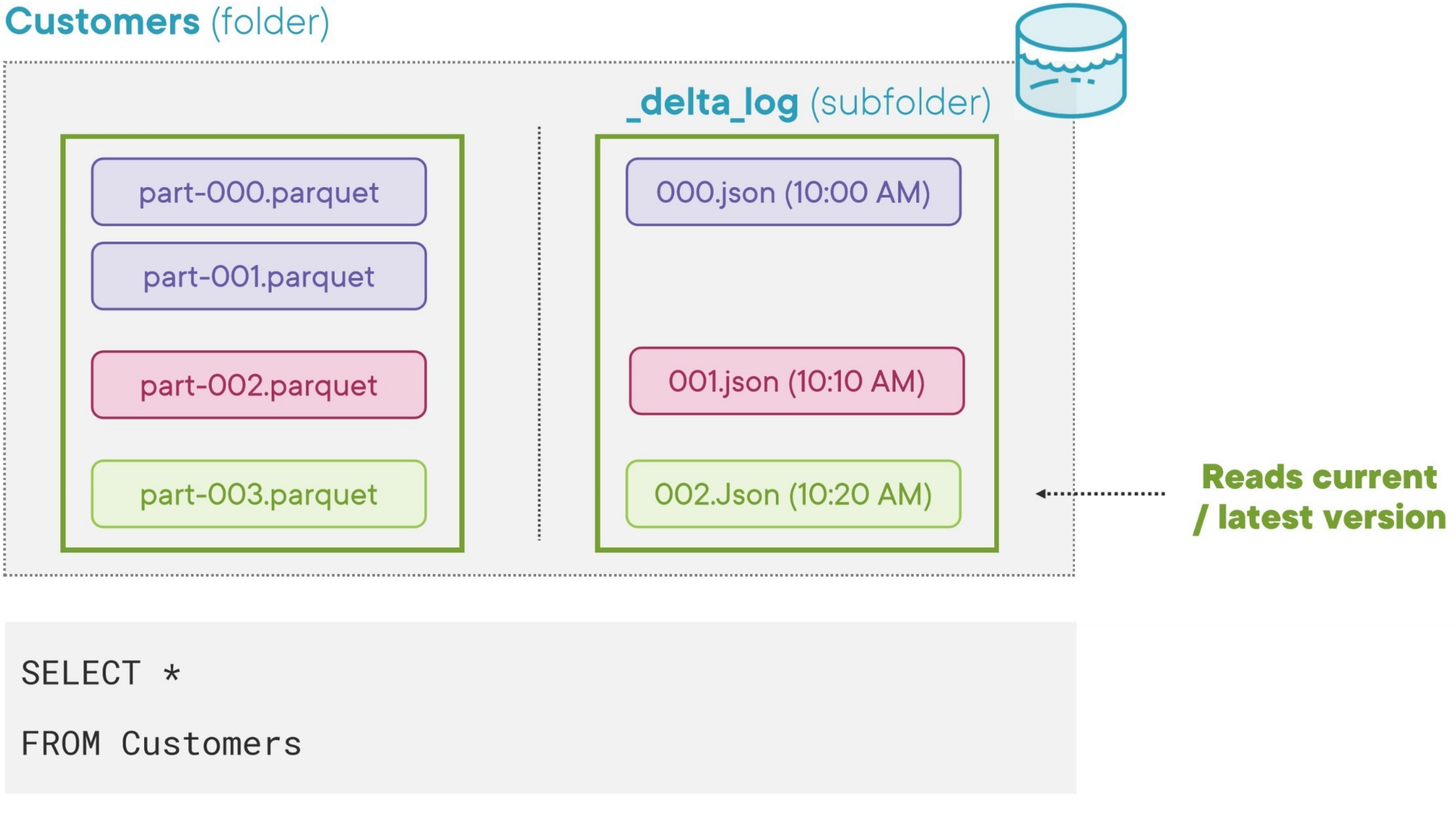
NOT NULL

To prevent column from having NULL values

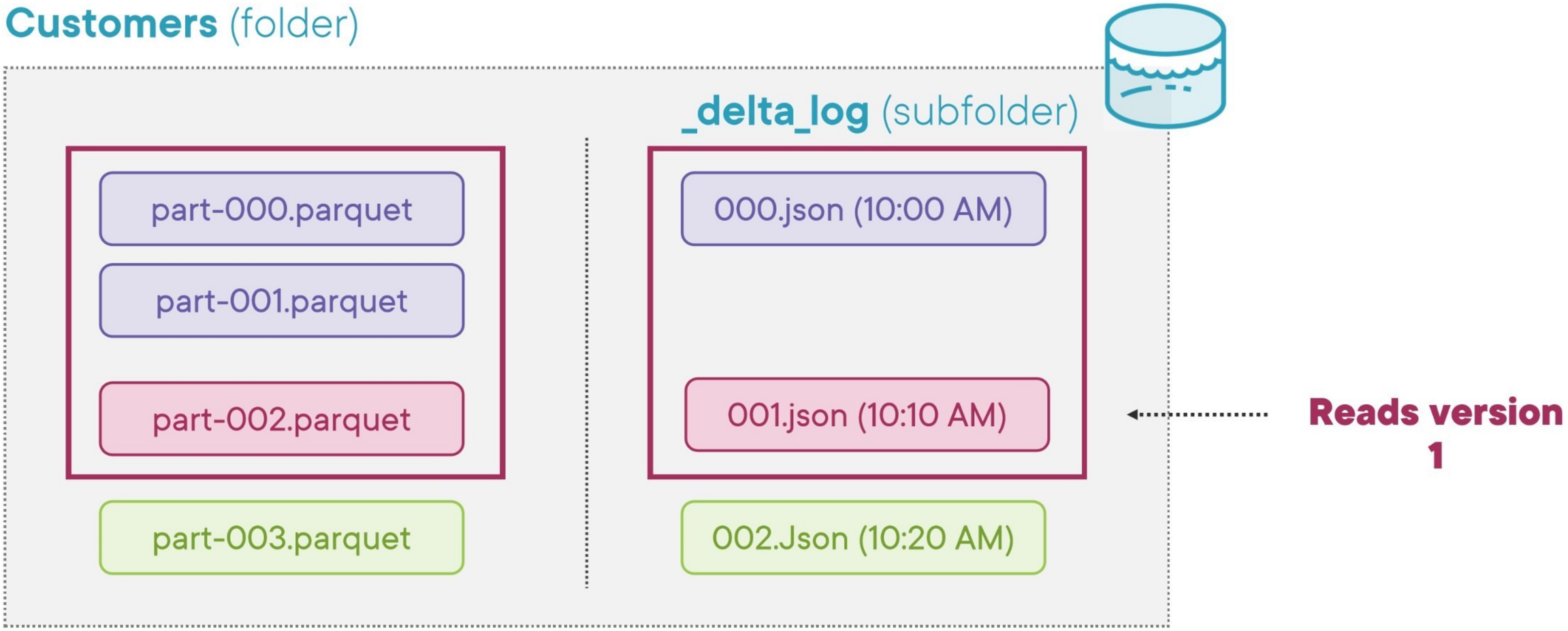
CHECK

Define conditions to enforce on data in the table

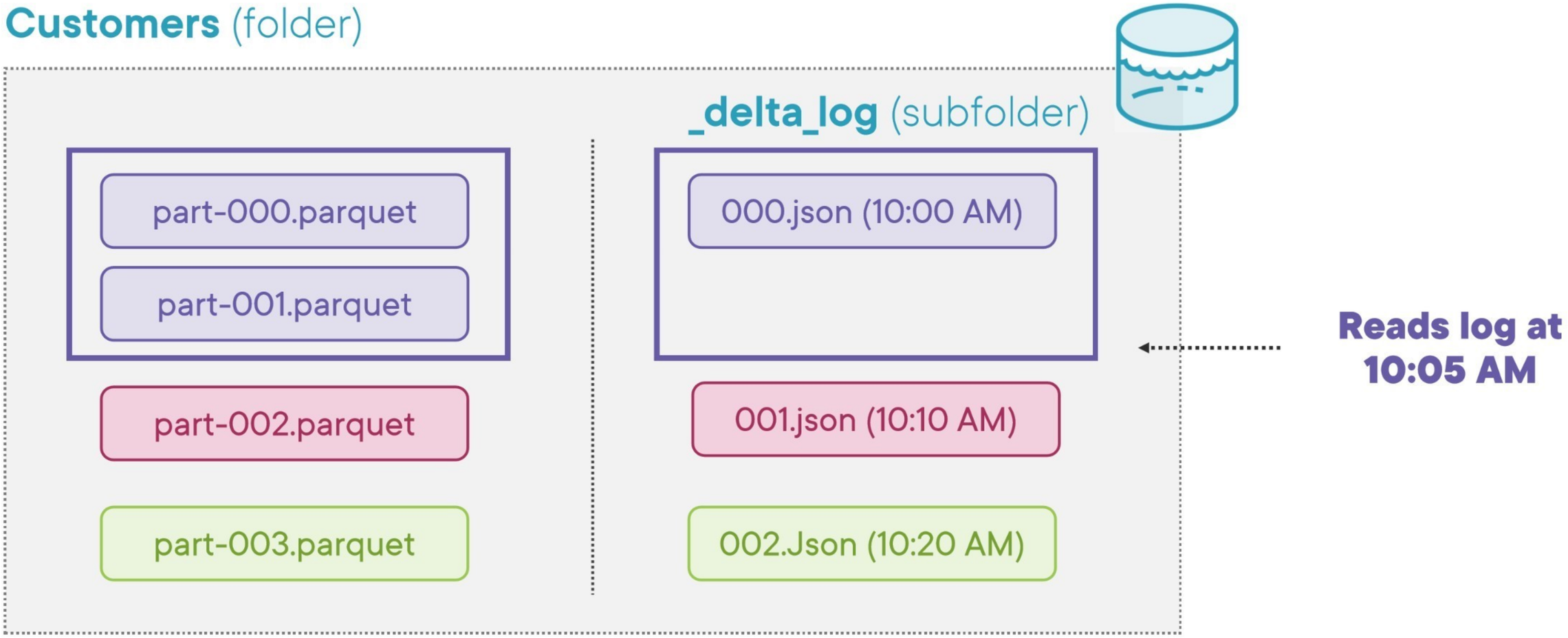
Accessing Data with Time Travel




```
SELECT *

FROM Customers
```




```
SELECT *

FROM Customers VERSION AS OF 1
```




```
SELECT *

FROM Customers TIMESTAMP AS OF '2022-03-01 10:05'
```

Time Travel allows to access/restore previous snapshot of data, even if data has been modified or deleted

Summary

Delta Lake is storage layer bringing reliability to Lakes

- Stores data in parquet format + Transaction log
- Provides ACID guarantees

Operations

- Write operation: First write the files, then the log
- Read operation: First read the log, then the files

Create Delta Table by storing data in delta format

Various options to add data to Delta Table

- Append DataFrame (Python) & Insert (SQL)

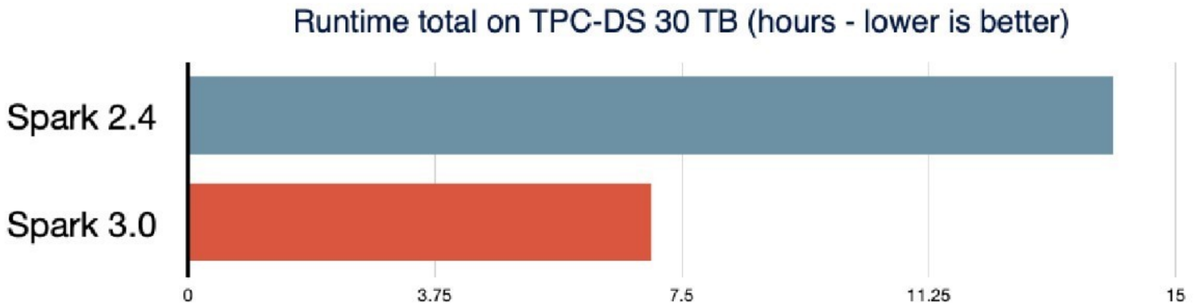
Perform DML operations - Update, Delete & Merge

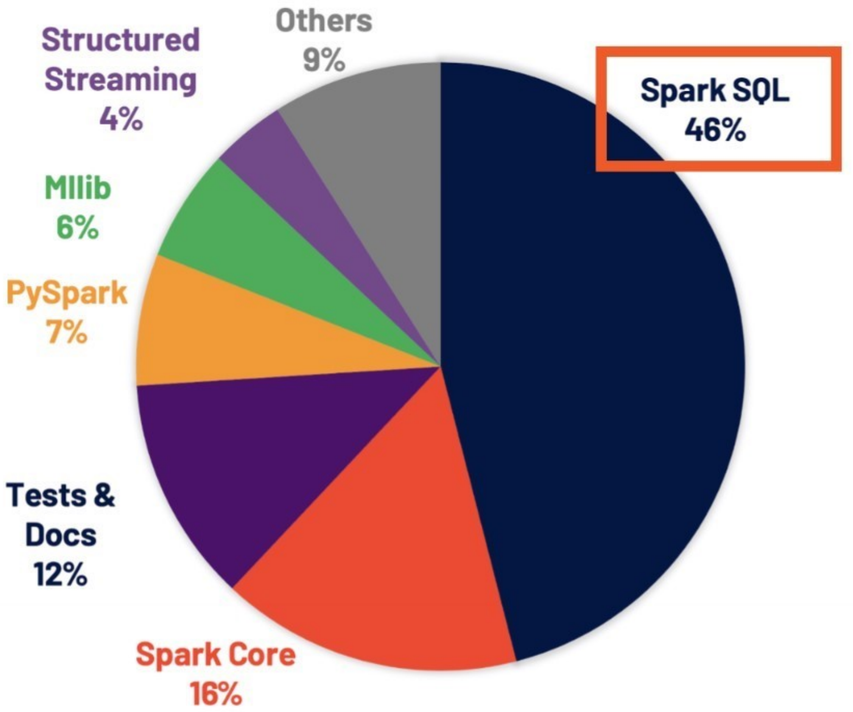
Apply table constraints – Not Null & Check

Access data using time travel

- Query using version or timestamp, or restore table

Apache Spark 3





*Images Source: databricks.com/blog

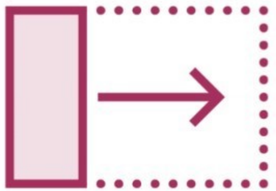
Apache Spark 3 Features

Performance

API Enhancements

SQL Compliance

Data Sources



Extensibility

Monitoring

1

Performance Improvements

Adaptive Query Execution (AQE) framework

- Reoptimizes query plan at runtime based on stats
- a) Dynamically coalescing shuffle partitions
- b) Dynamically switching join strategy
- c) Handling data skew in joins

Dynamic Partition Pruning (DPP)

- Improves on Partition Pruning technique

SQL join hints

- Spark has multiple join strategies
- Join Hint allows to enforce a particular join strategy
- Join Hints available for each join strategy

Faster query compilation

2

APIs, SQL and Monitoring

30+ new built-in functions

PySpark enhancements

- Pandas API enhancement & support
- Better error handling

Deep Learning improvements

- Project Hydrogen aims to unify data processing & deep learning

ANSI SQL compliance

- spark.sql.ansi.enabled *(disabled by default)*

Monitoring

- Better UI for Spark Structured Streaming
- Observable Metrics

3

Data Sources, Extensibility & Ecosystem

Built-in Data Sources

- New data sources like Apache Iceberg
- Performance improvements to existing sources like Parquet, Kafka, Delta Lake etc.

Extensibility

- Catalog API to use external catalog for managing tables (instead of Hive)

Ecosystem

- Support for Hadoop 3+, Hive 3+, JDK 11+

Overview

Adaptive Query Execution

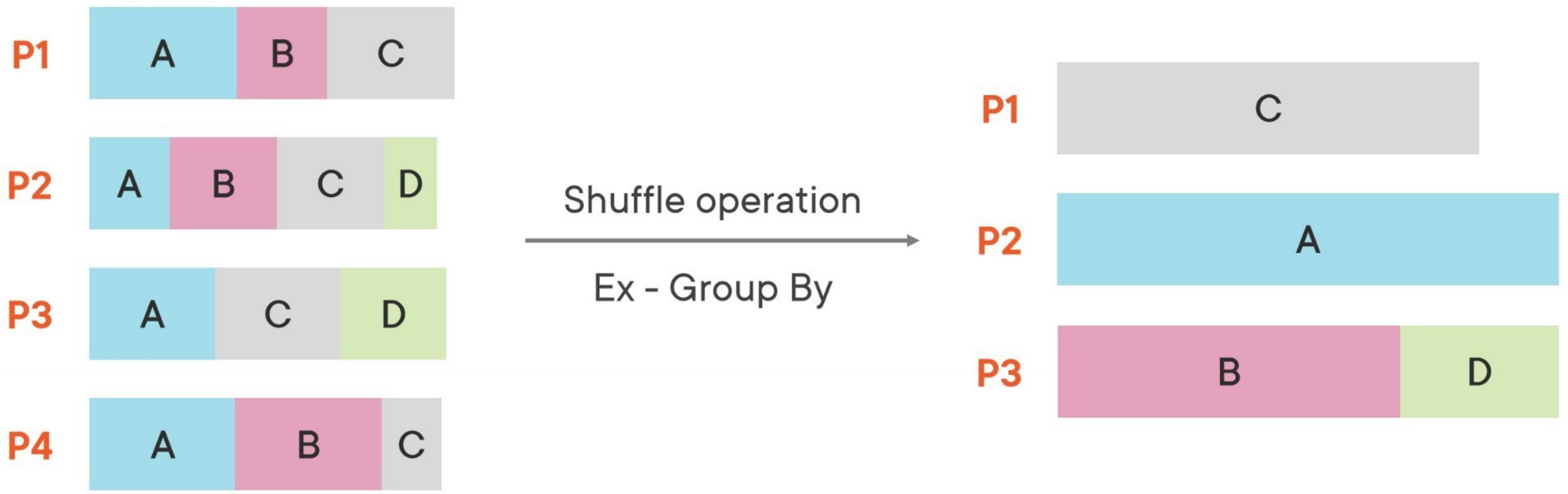
- Dynamically coalescing shuffle partitions
- Dynamically switching join strategy
- Handling data skew in joins

Dynamic Partition Pruning

Adaptive Query Execution: Dynamic Coalescing

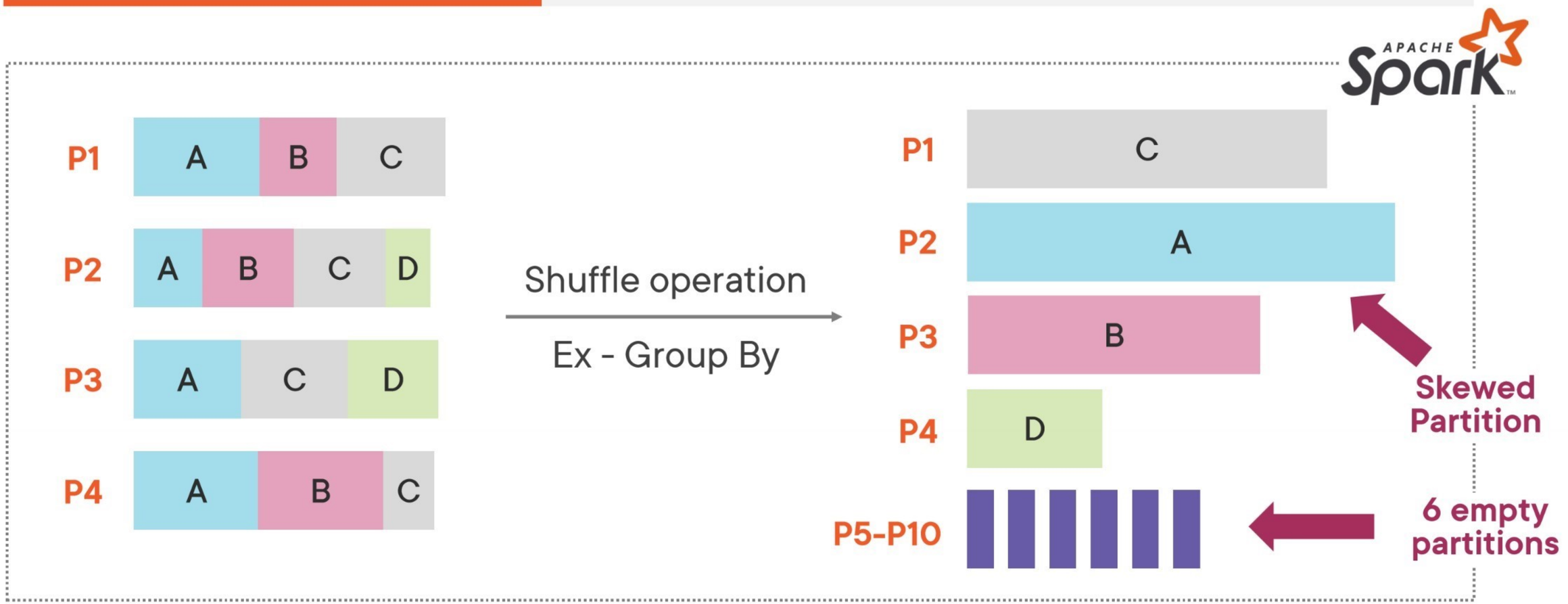
Shuffling Data

spark.sql.shuffle.partitions = 3 [default = 200]



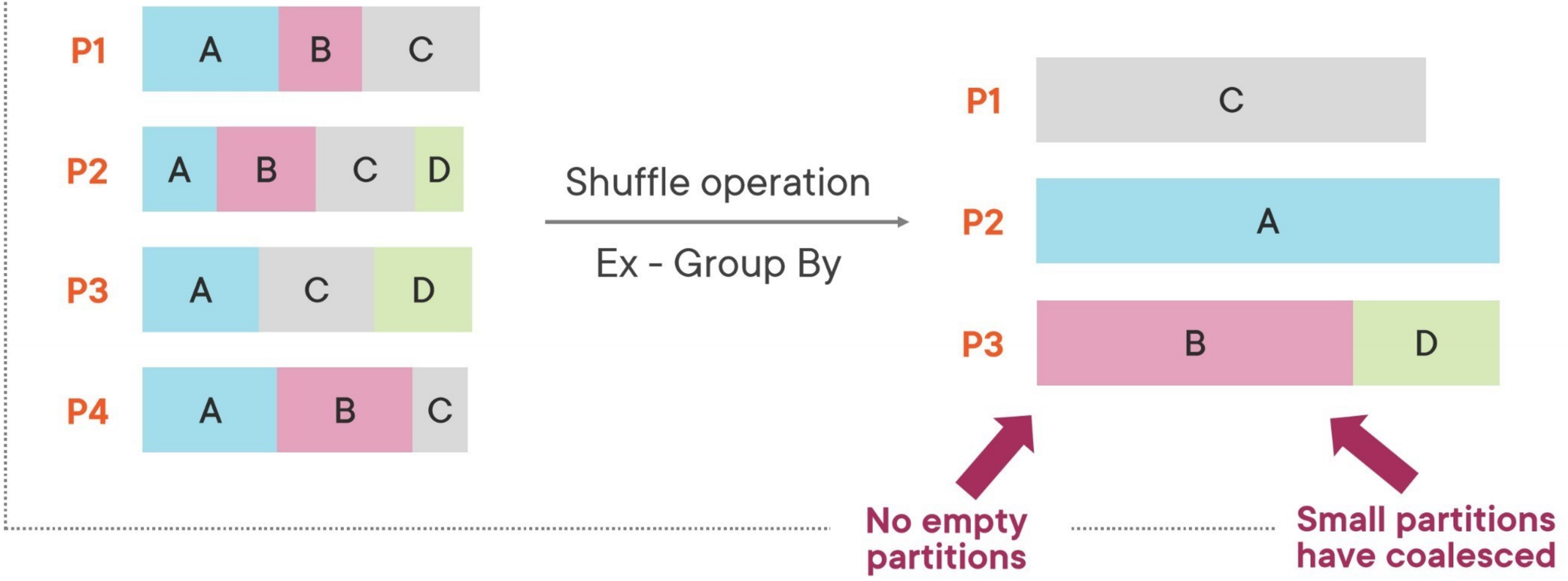
Shuffling Data

spark.sql.shuffle.partitions = 10 [default = 200]

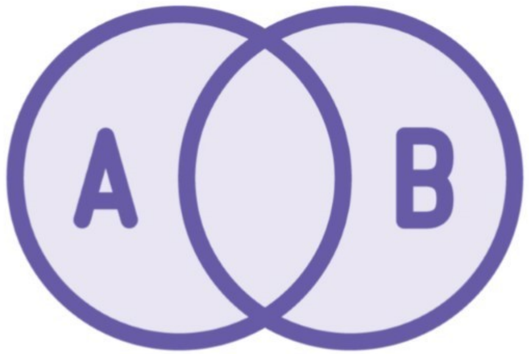


Shuffling With AQE

spark.sql.shuffle.partitions	= 10	[default = 200]
spark.sql.adaptive.enabled	= true	[default = true]
spark.sql.adaptive.coalescePartitions.enabled	= true	[default = true]



Dynamic Coalescing Shuffle Partitions



Having empty or lot of small partitions:

- Too many tasks are created
- Reduces parallelism and consumes time/resources

AQE dynamically coalesces shuffle partitions

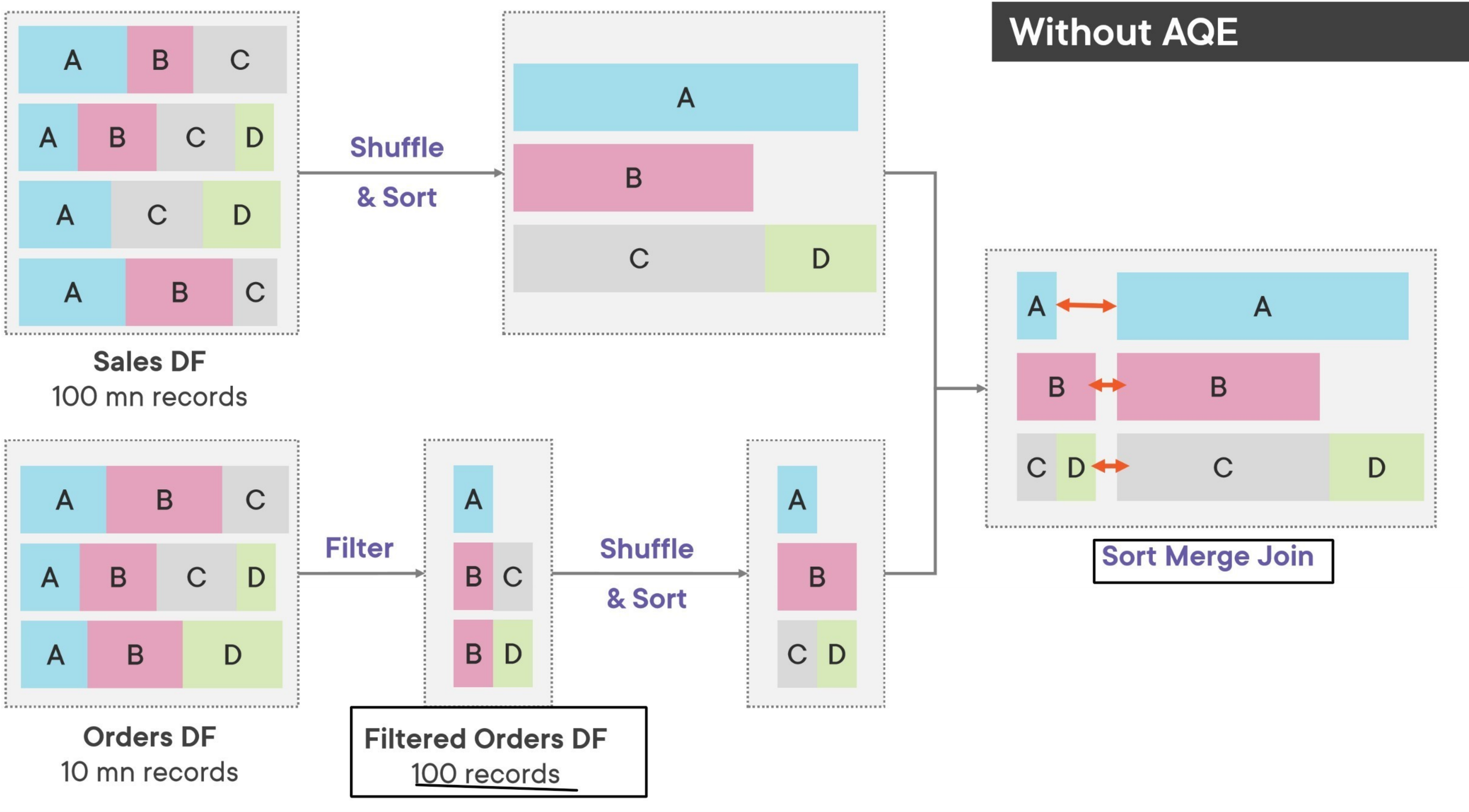
- Removes empty partitions
- Combines small partitions to produce optimal sized shuffle partitions

Datasets are big → Sort merge

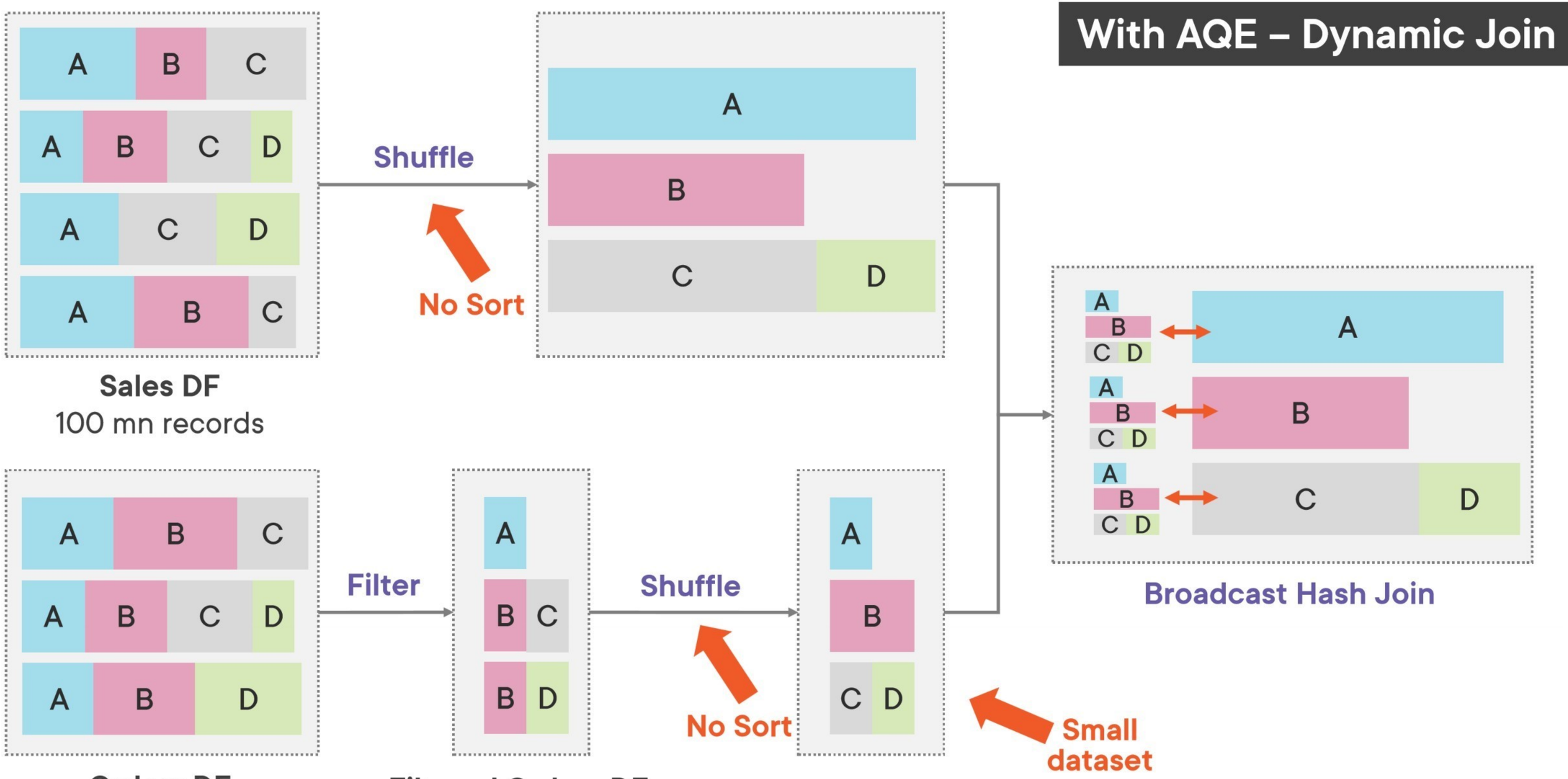
One dataset is small and one is big → Broadcast

Adaptive Query Execution: Dynamic Join

Without AQE


A B C
A B C D
A C D
A B C
Shuffle
& Sort
A
B
C D
Sales DF
100 mn records
A A
B B
C D C D
Sort Merge Join
A B C
A B C D
A B D
Filter
A
B C
B D
Filtered Orders DF
100 records
Shuffle
& Sort
A
B
C D
Orders DF
10 mn records

With AQE – Dynamic Join



AQE – Dynamic Switching Join Strategy
OR
Manual Broadcast

Which is better?

Dynamically Switching Join Strategies

For large datasets, Shuffle Sort Merge Join is performed

For Broadcast Hash Join, one dataset must be small

- Small dataset should be less than setting spark.sql.autoBroadcastJoinThreshold
- Default threshold is 10 MB

If highly selective filter is applied on large dataset

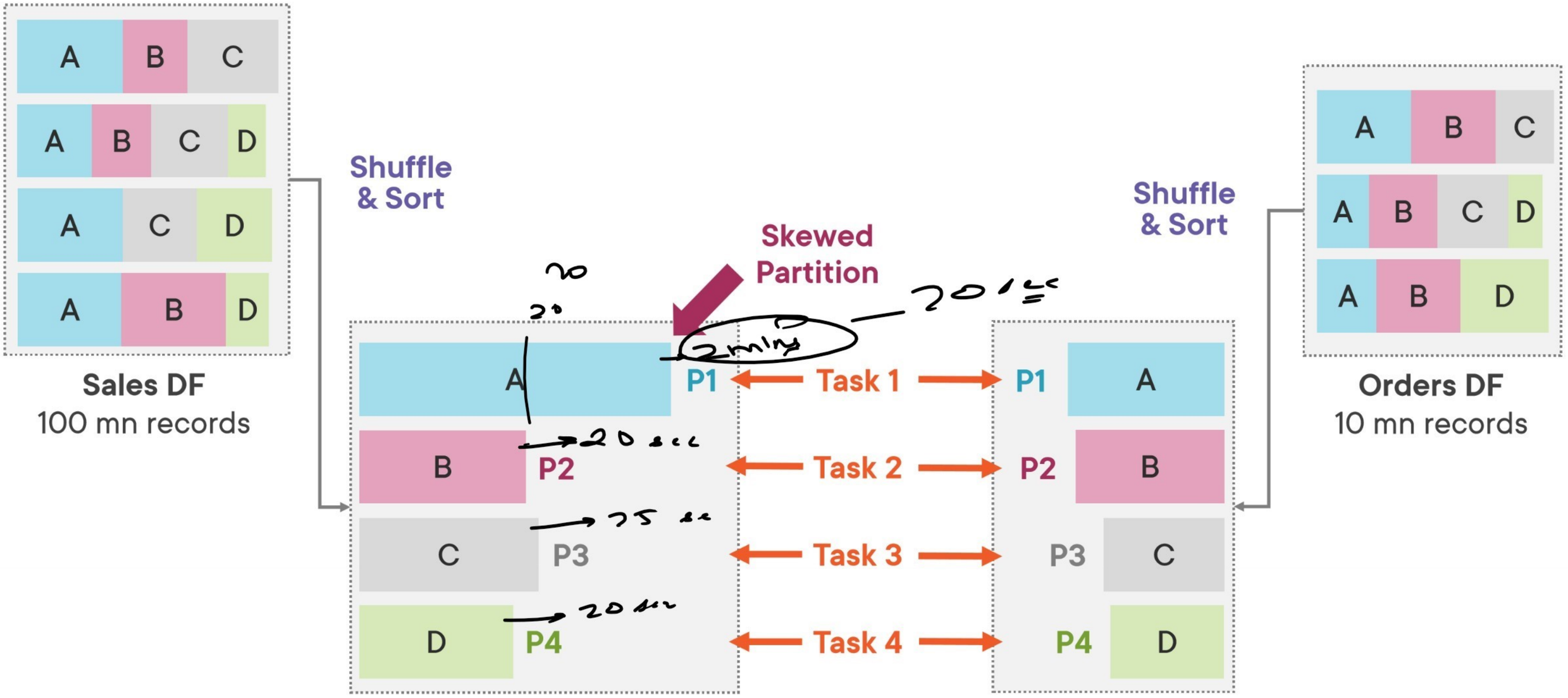
- It may become smaller than broadcast threshold
- Since execution plan is ready, still performs Shuffle Sort Merge Join (if joined with large dataset)

If enabled, AQE dynamically switches from Sort Merge Join to Broadcast Hash Join at runtime

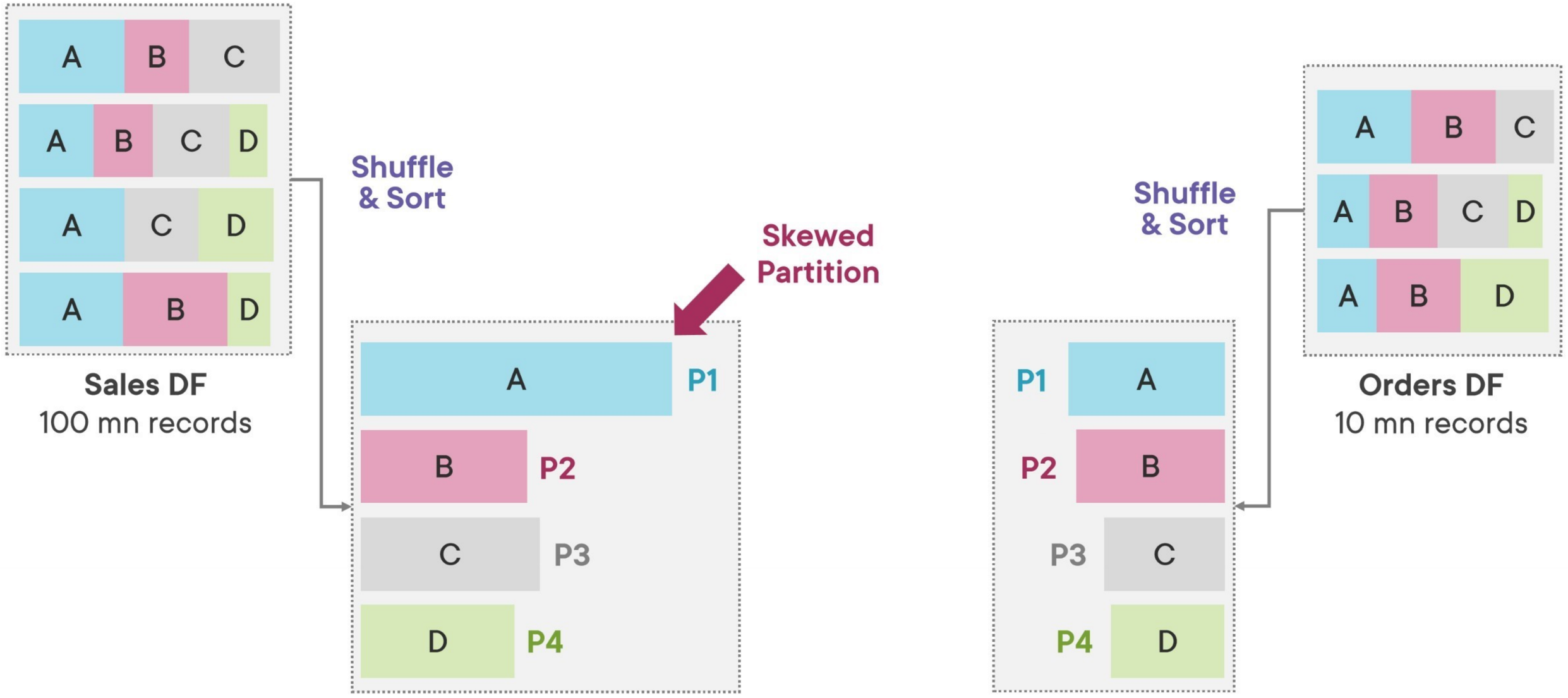
- AQE checks for dataset size after shuffle
- If size of a dataset is now less than broadcast threshold, switches to Broadcast Hash Join

Adaptive Query Execution: Handling Skew

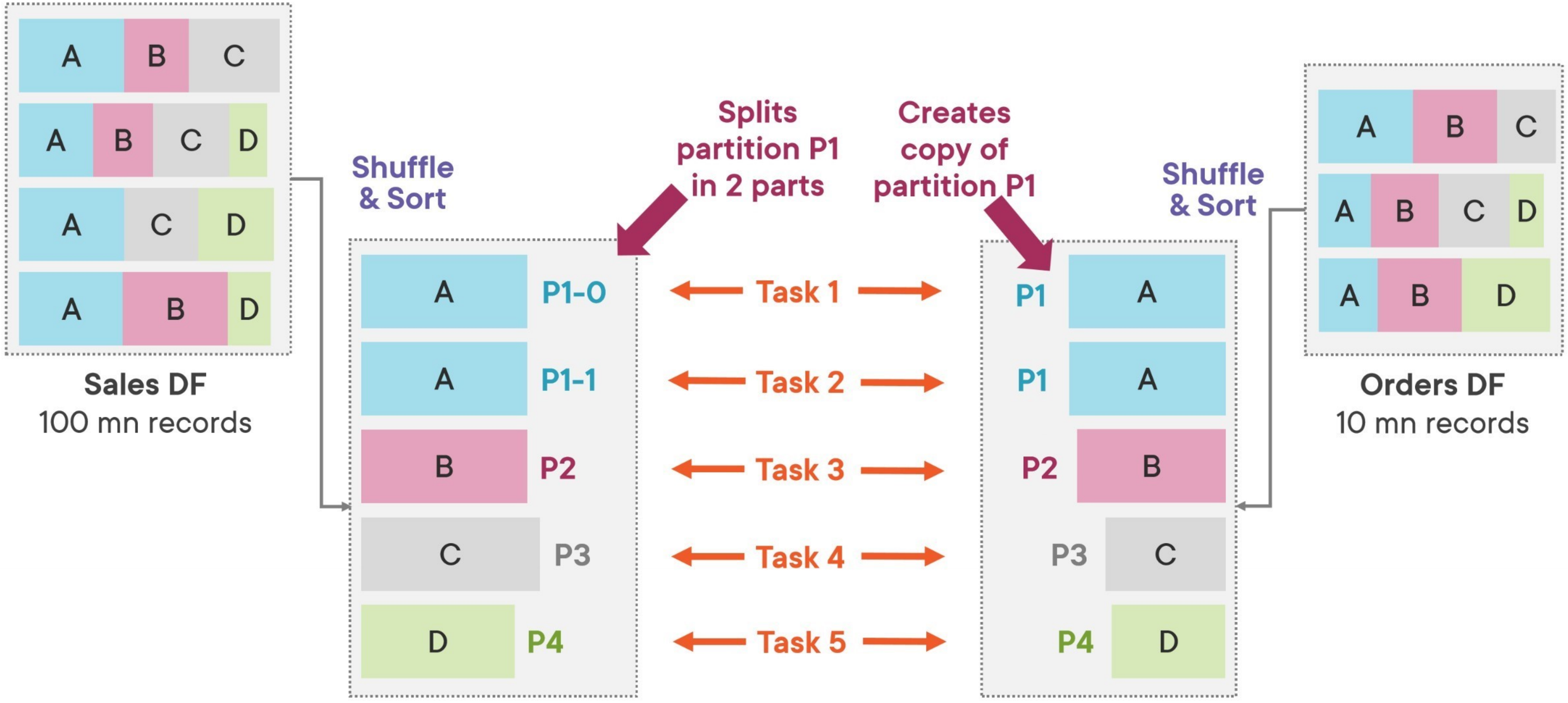
Without AQE


A B C
A B C D
A C D
A B D
Sales DF
100 mn records
Shuffle & Sort
Skewed Partition
A P1
B P2
C P3
D P4
Task 1
Task 2
Task 3
Task 4
P1 A
P2 B
P3 C
P4 D
Shuffle & Sort
A B C
A B C D
A B D
Orders DF
10 mn records

With AQE: Handling Skew



With AQE: Handling Skew



Dynamically Optimizing Skew Joins

Data Skew

- One partition has much more data than others

In join operations:

- After shuffle, data may be unevenly distributed among partitions
- Data skew can impact query performance
- Tasks processing larger partitions will take more time than ones handling smaller partitions

If enabled, AQE dynamically optimizes data skew in joins

- AQE checks for partition sizes after shuffle
- Splits skewed partitions into smaller sub-partitions
- Creates copy of corresponding partition on other side
- Number of tasks increase, but each one will almost take same time to finish

Dynamic Partition Pruning

Disk Partitions

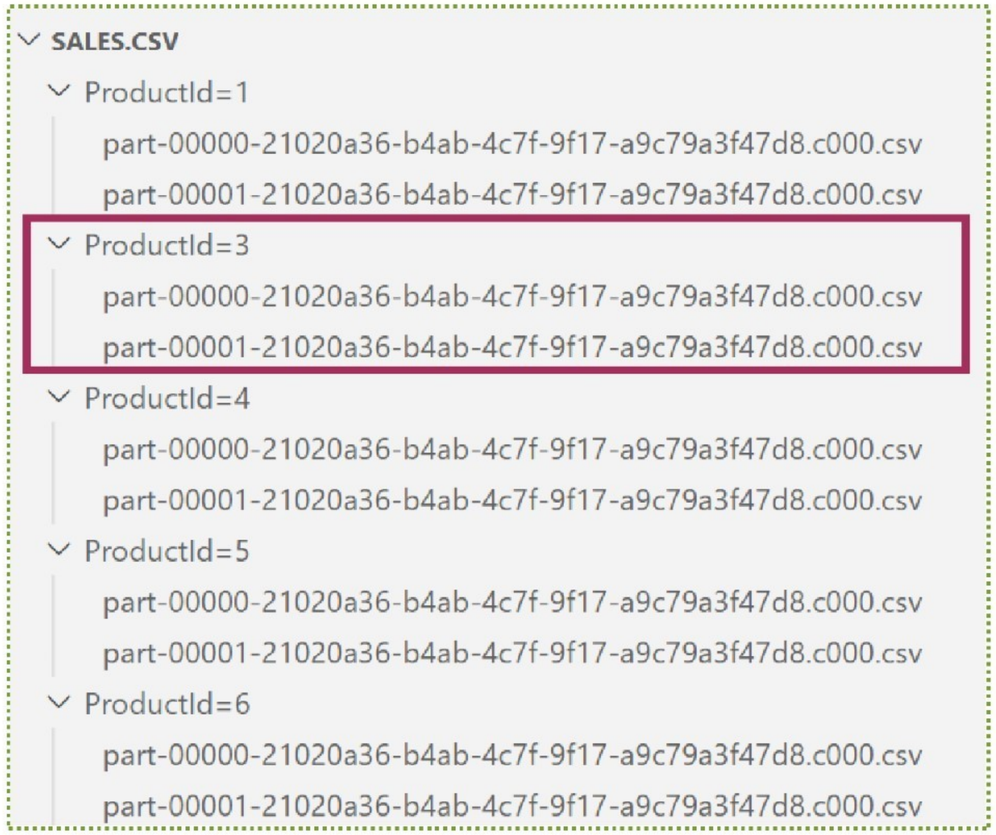
```
(
   salesDF
      .write
      .partitionBy("ProductId")
      .saveAsTable("Sales")
)
```

Partition Pruning

```
SELECT *
FROM Sales
WHERE ProductId = 3
```

Sales

100 mn records - Partitioned



Products

1000 records – Non-partitioned

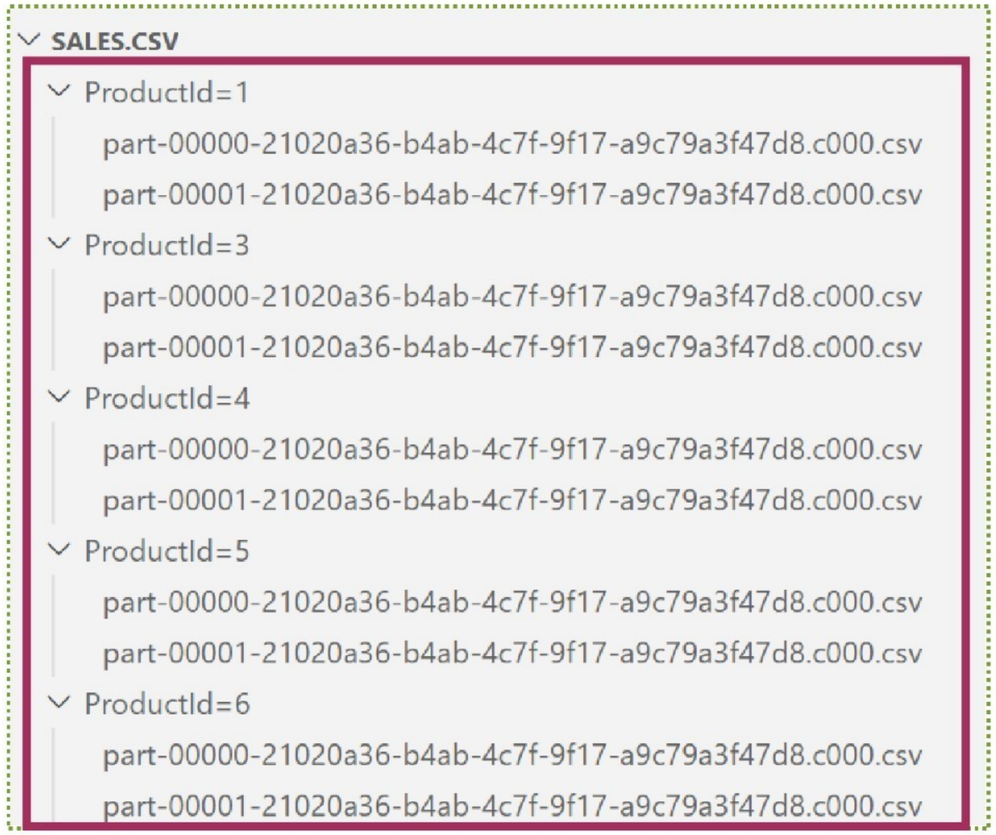
Partition Pruning Works!

```
SELECT /*+  BROADCASTJOIN(p) */ *

FROM Sales s
   JOIN Products p
   ON s.ProductId = p.ProductId

WHERE s.ProductId = 3
```

Small Table

Sales

100 mn records - Partitioned

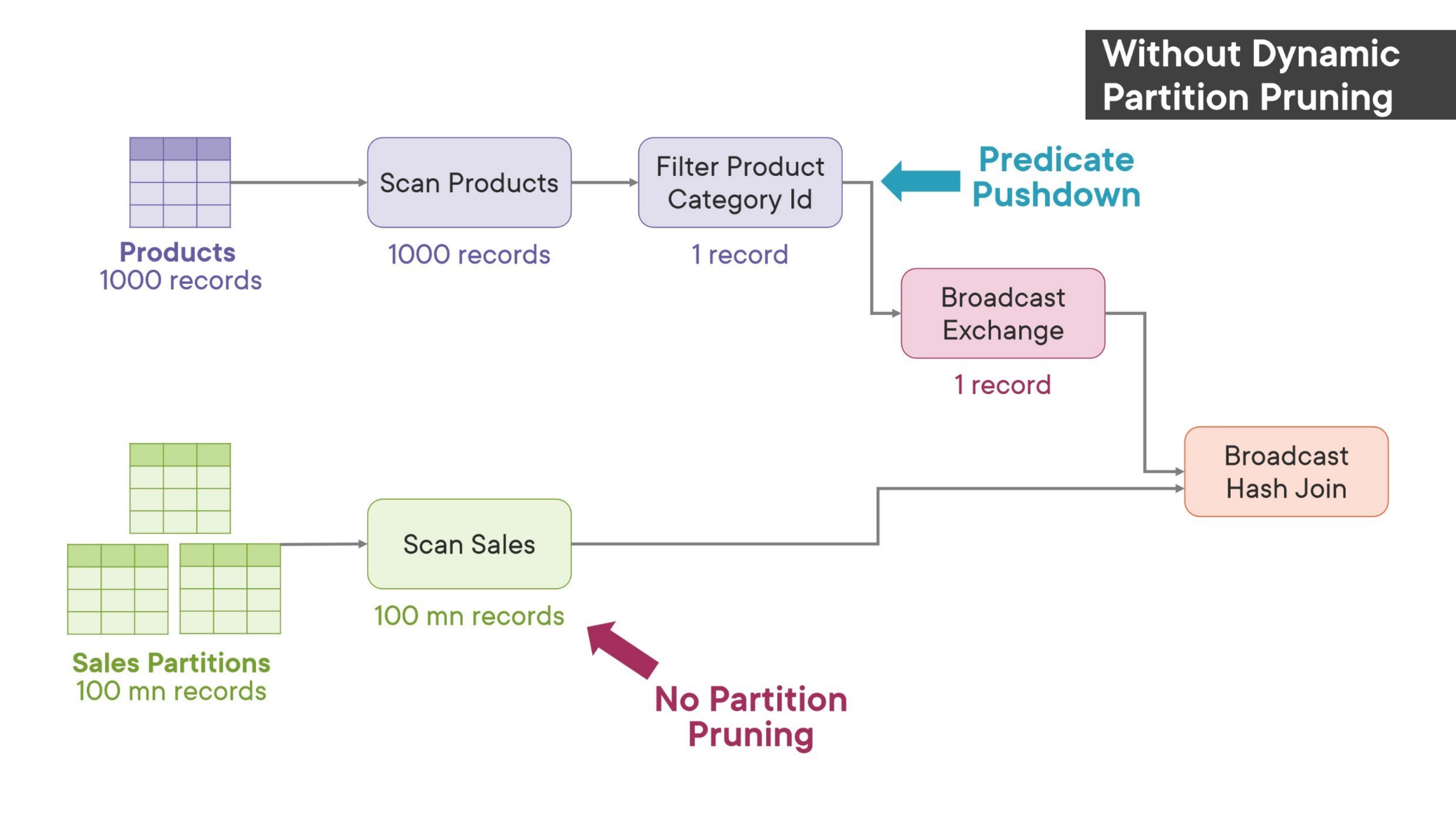


Products

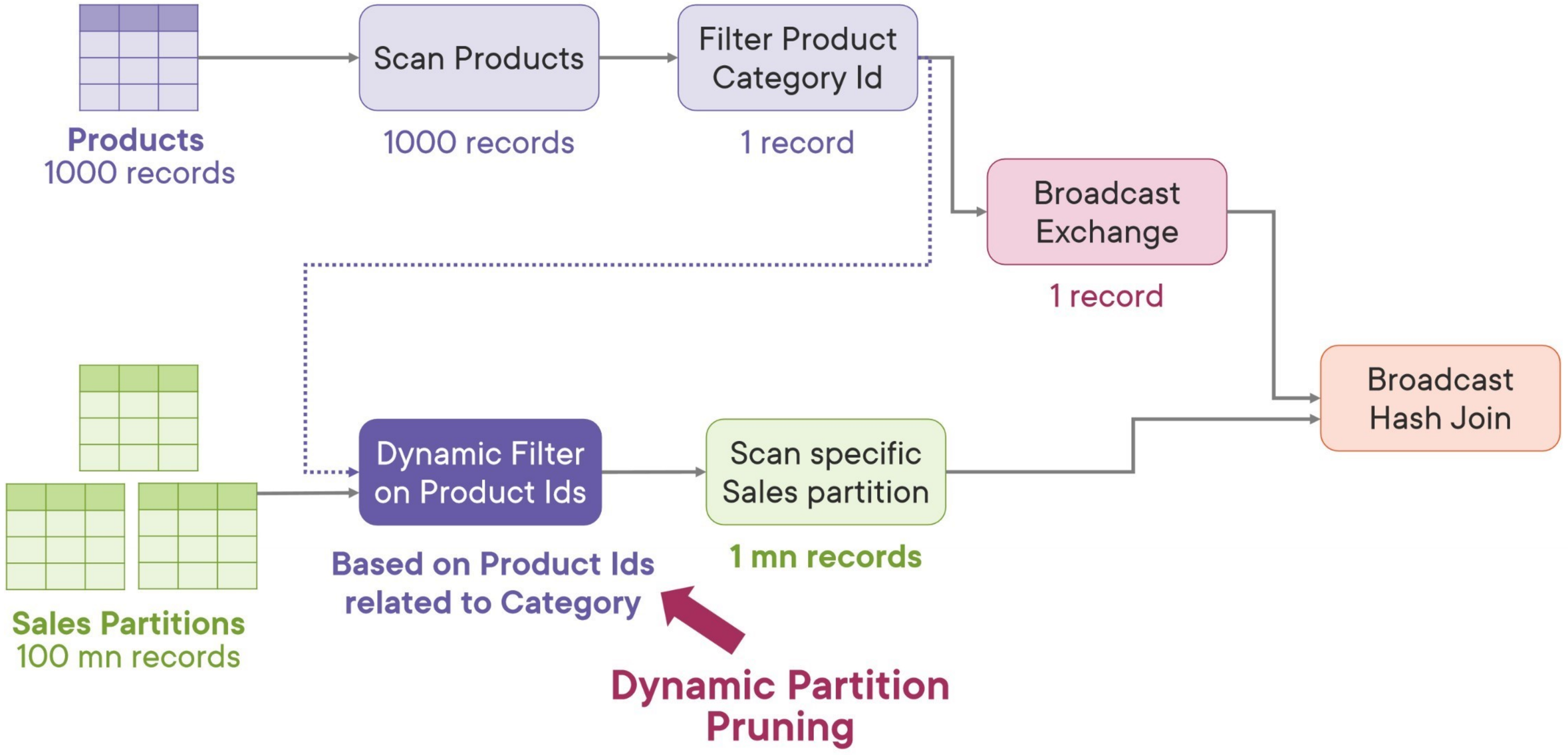
1000 records – Non-partitioned

Partition Pruning Does Not Work!

```
SELECT /*+  BROADCASTJOIN(p) */ *

FROM Sales s
   JOIN Products p
   ON s.ProductId = p.ProductId

WHERE p.ProductCategoryId = 3
```


Without Dynamic Partition Pruning
Products
1000 records
Scan Products
1000 records
Filter Product Category Id
1 record
Predicate Pushdown
Broadcast Exchange
1 record
Broadcast Hash Join
Sales Partitions
100 mn records
Scan Sales
100 mn records
No Partition Pruning

With Dynamic Partition Pruning



Conditions

Dynamic Partition Pruning must be enabled

- spark.sql.optimizer.dynamicPartitionPruning.enabled

Large table must have disk partitions

During join, small table should be broadcasted

Summary

Spark 3 has several optimizations

Adaptive Query Execution

- Reoptimizes query plan at runtime based on stats

1. AQE: Dynamically coalescing shuffle partitions

- After shuffle, empty shuffle partitions are removed & small ones are merged

2. AQE: Dynamically switching join strategy

- At runtime, if one dataset becomes small, AQE can switch from Shuffle Sort to Broadcast Hash join

3. AQE: Handling data skew in joins

- After shuffle, if any partition is skewed, AQE can split that partition to multiple smaller partitions

Dynamic Partition Pruning

- During join, Spark can dynamically skip disk partitions at runtime that are not required by the query