Adaptive Query Execution in Spark

Dano Lee

Adaptive Query Execution

Spark Catalyst Optimizer

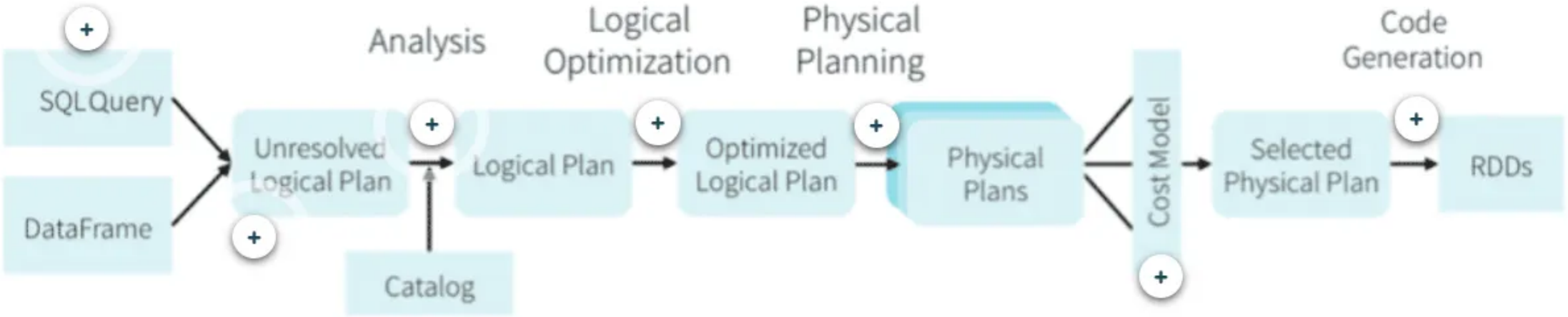
Spark 1.x, Rule

Spark 2.x, Rule + Cost

Spark 3.0, Rule + Cost + Runtime

databricks

Catalyst Optimizer



Measuring the Cost of Physical Plans

- The cost of a physical plan is measured based on several factors, including:
  - **Data Size**: The number of records and size of each partition.
  - **Operation Type**: Some operations are more expensive than others (e.g., shuffling data across the network is costly).
  - **Data Distribution**: How data is distributed across the nodes can influence the cost.
  - **Join Strategy**: The cost varies significantly depending on the join strategy used (broadcast join, shuffle join, etc.).
  - **Number of Stages**: More stages typically involve more overhead in terms of task scheduling and execution.

Example of Cost Consideration

- Consider a scenario where **a join operation needs to be performed between two datasets**. Catalyst may generate multiple physical plans for this join, such as:
  - **Broadcast Join**: If one dataset is small enough to fit into memory, Spark may decide to broadcast it to all nodes, avoiding a costly shuffle operation.
  - **Sort-Merge Join**: If both datasets are large, Spark may decide to use a sort-merge join, which involves sorting and shuffling data across the network.
- The cost model will estimate the costs of these plans based on factors like the size of the datasets, the number of partitions, and the amount of network communication required. It will then choose the plan with the lowest estimated cost.

Adaptive Query Execution

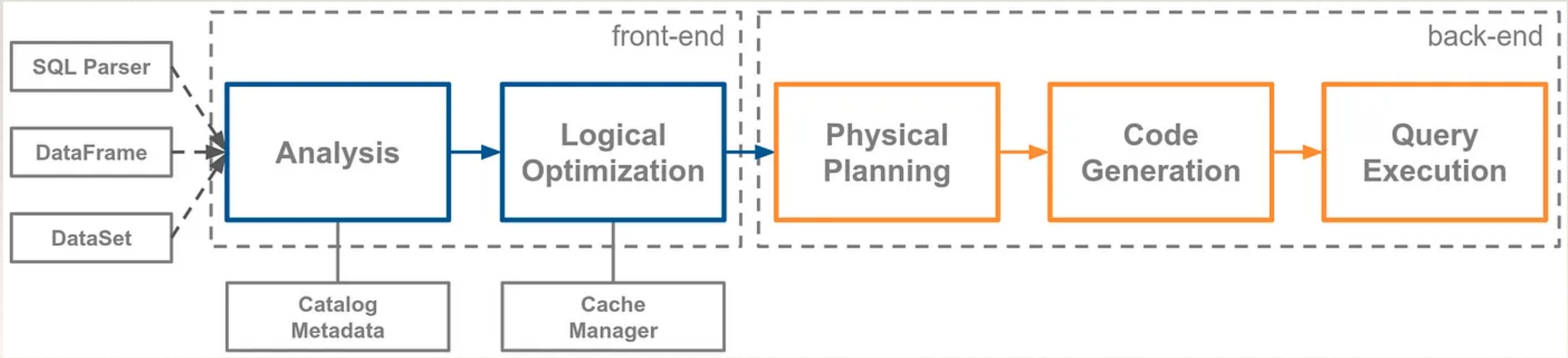
- an advanced optimization engine that **dynamically tunes the execution plan** for a given query, **based on the runtime statistics of the data**. It can improve query performance by optimizing the data processing pipelines.
- Traditional query execution engines follow **a fixed plan for processing the data, regardless of the data's distribution or statistics**. This approach can result in suboptimal performance when dealing with large, complex datasets.
- To overcome this issue, Spark introduced the AQE optimizer, which dynamically adapts the execution plan based on the characteristics of the data.
- Improves execution efficiency, especially in scenarios with data skew or unpredictable data sizes.

Adaptive Query Execution

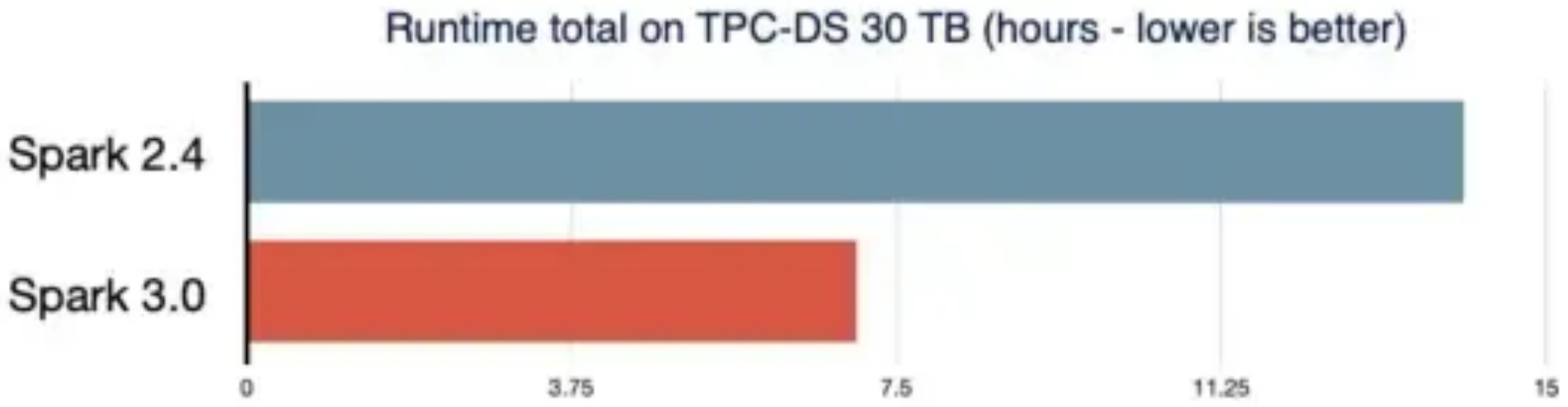
- Modern distributed systems are designed upon the **principle of decoupling compute and storage layers**, that is storing persistent data on remote storage and making use of local storage only for transient data.
- Spark was built following this idea (and recently Hadoop 3 has adopted it). While this design brings benefits in terms of scalability and availability (as well as cost reductions), **it also hinders the query optimization process, as unpredictable data arrivals make cost calculation less reliable**.

Adaptive Query Execution

- Very basically, a logical plan of operations (coming from the parsing a SQL sentence or applying a lineage of transformations over a DataFrame or a Dataset) is (1) resolved against the catalog, (2) optimized on a rule-basis (constant folding, predicate and projection pushdowns…) and (3) translated into a physical plan of Spark operators (actually into multiple plans and then selected on a cost basis).



```python
from pyspark.sql.functions import sum
from pyspark.sql.types import IntegerType

# Enable AQE
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.optimizerEnabled", "true")

# Read data from CSV
df = spark.read.csv("data.csv", header=True, inferSchema=True)

# Group by column 'A' and sum column 'B'
df = df.groupBy("A").agg(sum("B").alias("B"))

# Filter rows where 'B' is greater than 100
df = df.filter(df.B > 100)

# Cache the resulting DataFrame
df.cache()

# Count the number of rows in the DataFrame
print(df.count())

# Unpersist the DataFrame
df.unpersist()
```

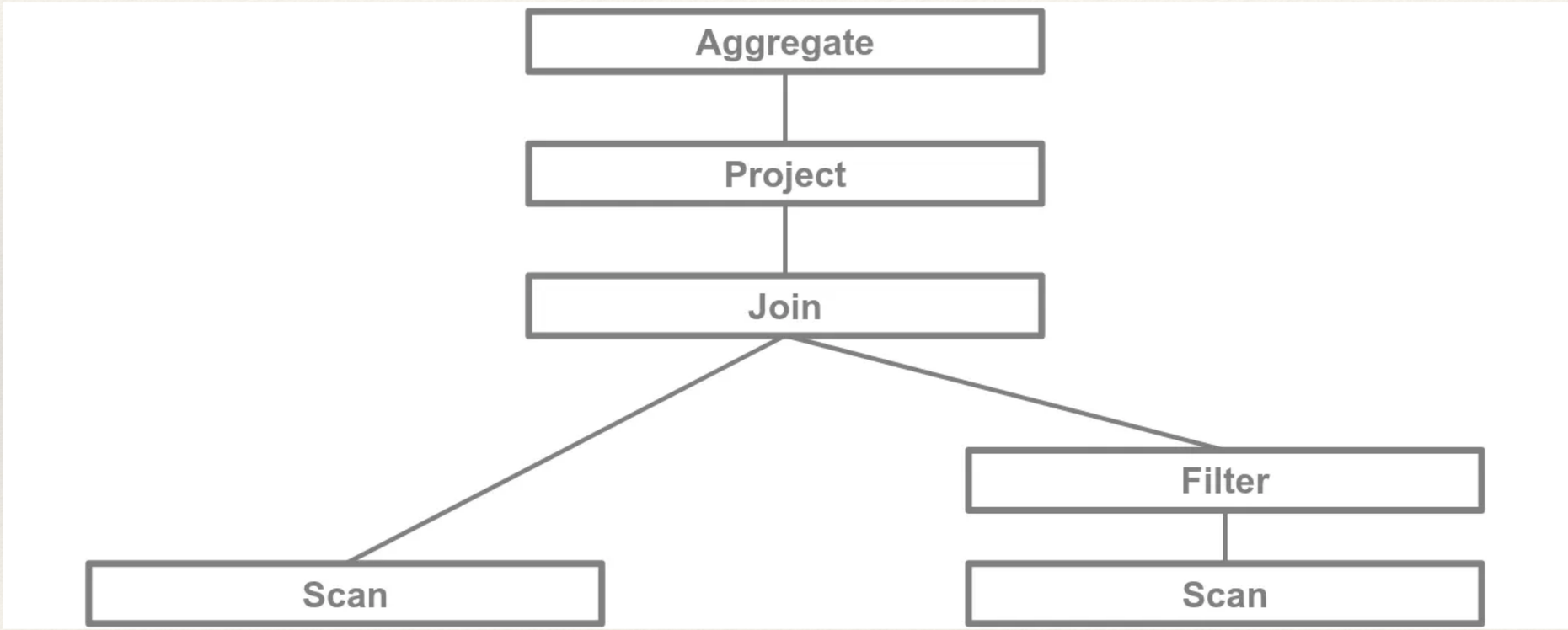

Runtime total on TPC-DS 30 TB (hours - lower is better)
Spark 2.4
Spark 3.0
0
3.75
7.5
11.25
15

source

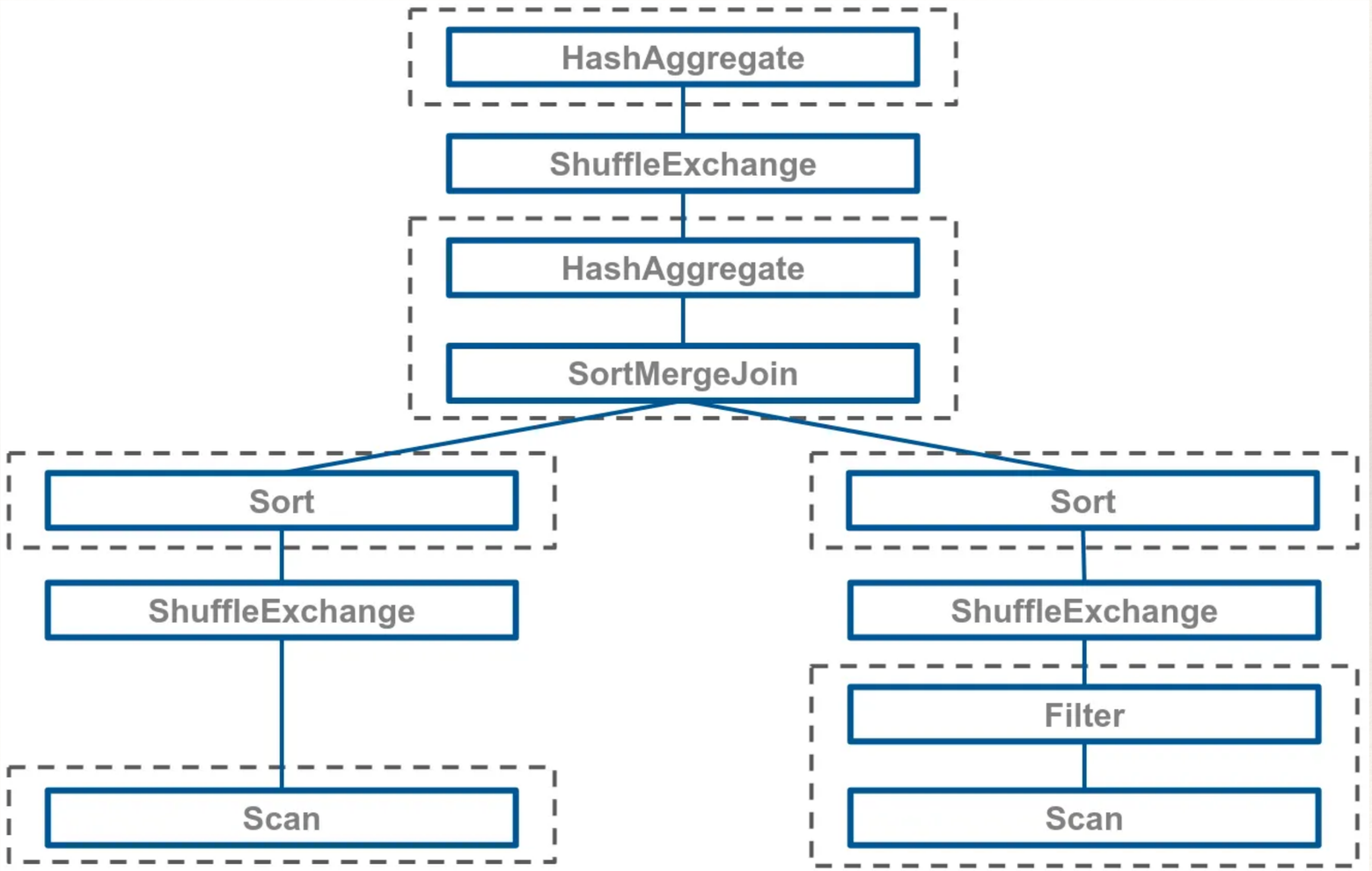
Adaptive Query Execution

```
select a1, avg(b1)
from A join B on A.pk =
B.pk
where b1 <= 10000
group by a1
```

SQL Logical Plan



Spark Physical Plan



Adaptive Query Execution

- The downside of this implementation is that once the physical execution plan is determined it cannot be altered at runtime. The plan is translated into a Direct Acyclic Graph (where nodes are RDDs and edges are dependencies) and this representation is submitted to the DAGScheduler. It breaks the graph down at shuffle boundaries and passes a TaskSet to a TaskScheduler, which takes care of execution on physical resources.
- When adaptive execution is enabled, **stages get earlier split by breaking down the logical plan into independent subgraphs** called **query stages**.

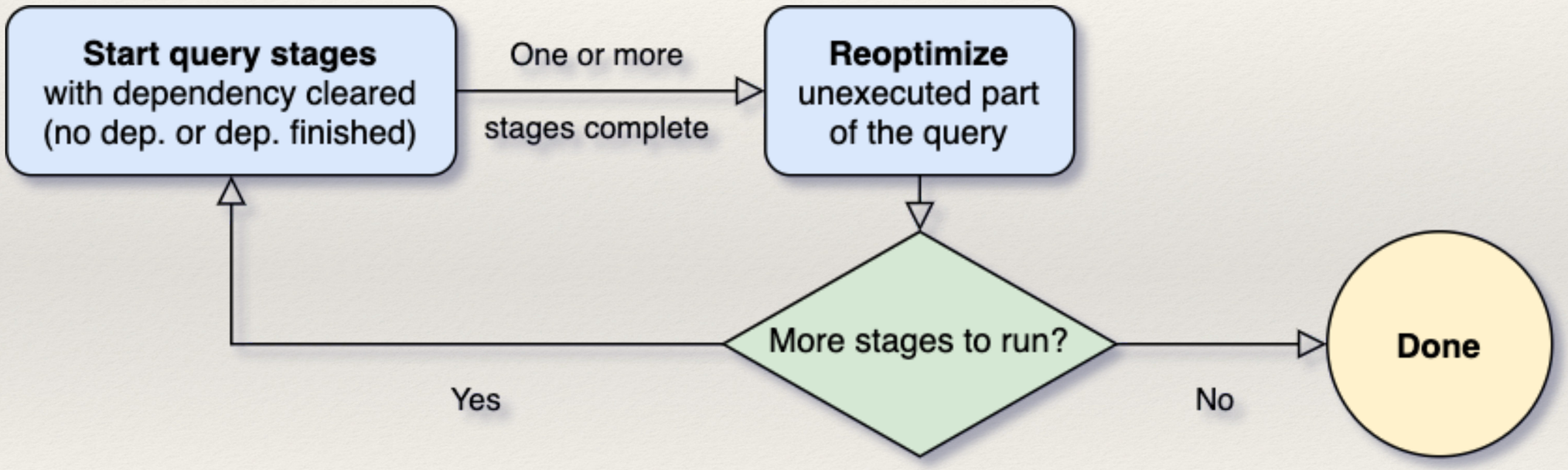
Query Stages in AQE

- Query stages **materialize their processing before further proceeding through down-stream stages in the execution plan.**
  - This allows to **individually submit map stages, collect their MapOutputStatistics objects**, and **analyze them for further tuning of subsequent steps** .

Query Stages in AQE

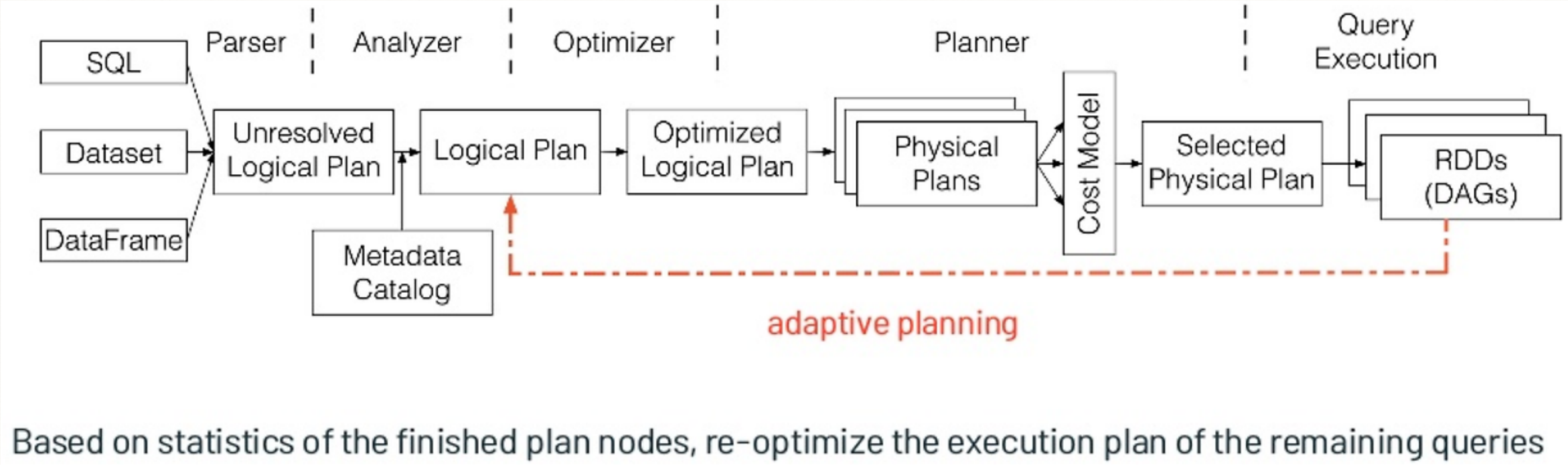
- Two types of query stages can be found in AQE plans:
  - **Shuffle query stages**, which materialize their output to shuffle files.
  - **Broadcast query stages**, which materialize their output to an array in Driver memory.
- To bear with this new feature, **DAGScheduler now supports the submission of a single map stage**. Spark SQL engine also include modifications at planning and execution phases.

Query Stages in AQE



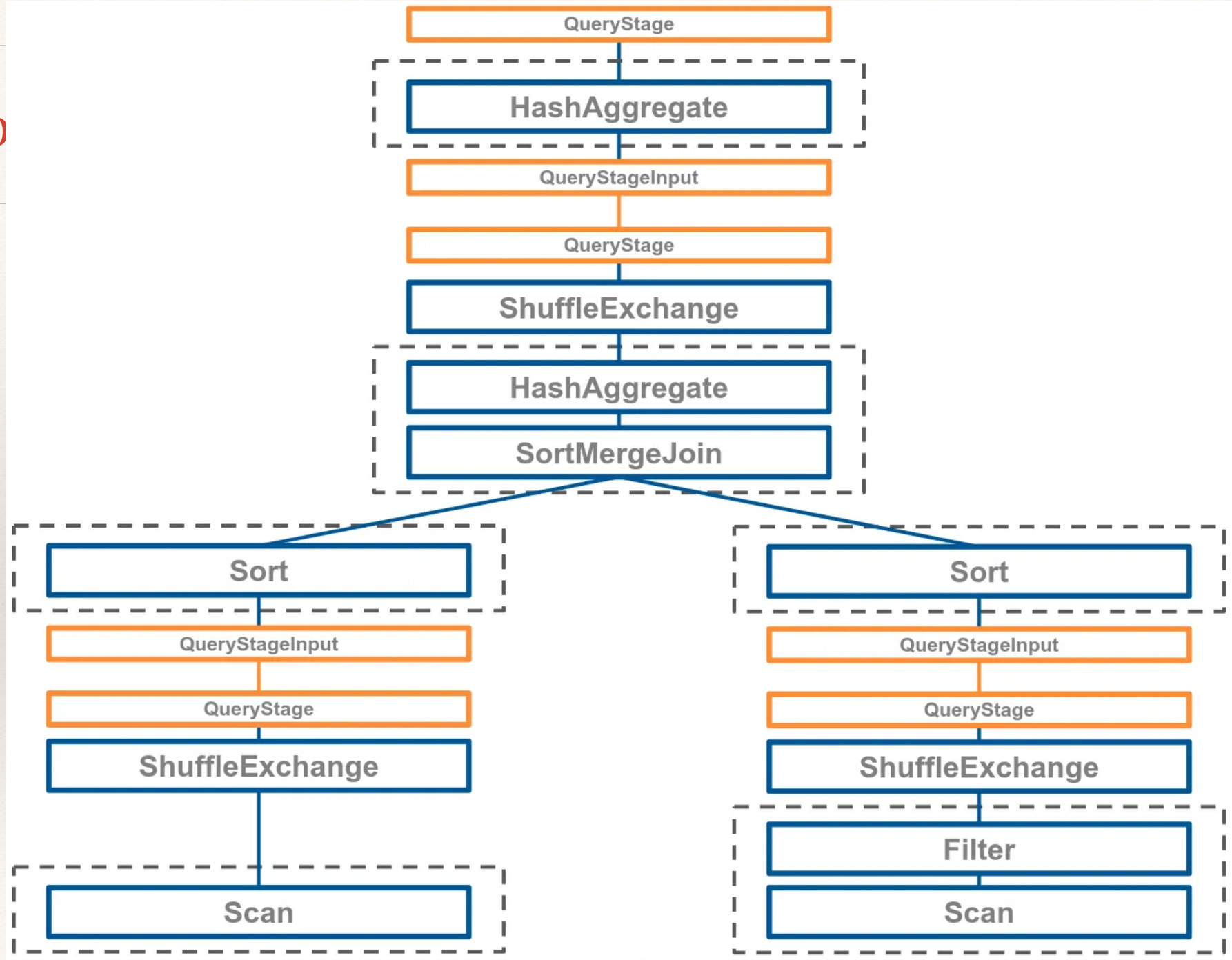
source

Query Stages in AQE



source

AQE p nput nodes



Query Stages in AQE

- During the execution phase, any QueryStage on the tree has a reference to its child stages and executes them recursively. After all children of a QueryStage are completed, runtime shuffle-write statistics are collected and used for further refinement. Spark then re-launches logical optimization and physical planning phases, and dynamically updates the query plan according to this fresh information.

AQE is disabled by default

```
spark.sql.adaptive.enabled = false
```

When to use AQE

- Parallelism on Reducers
- Join Strategy
- Skewed Joins

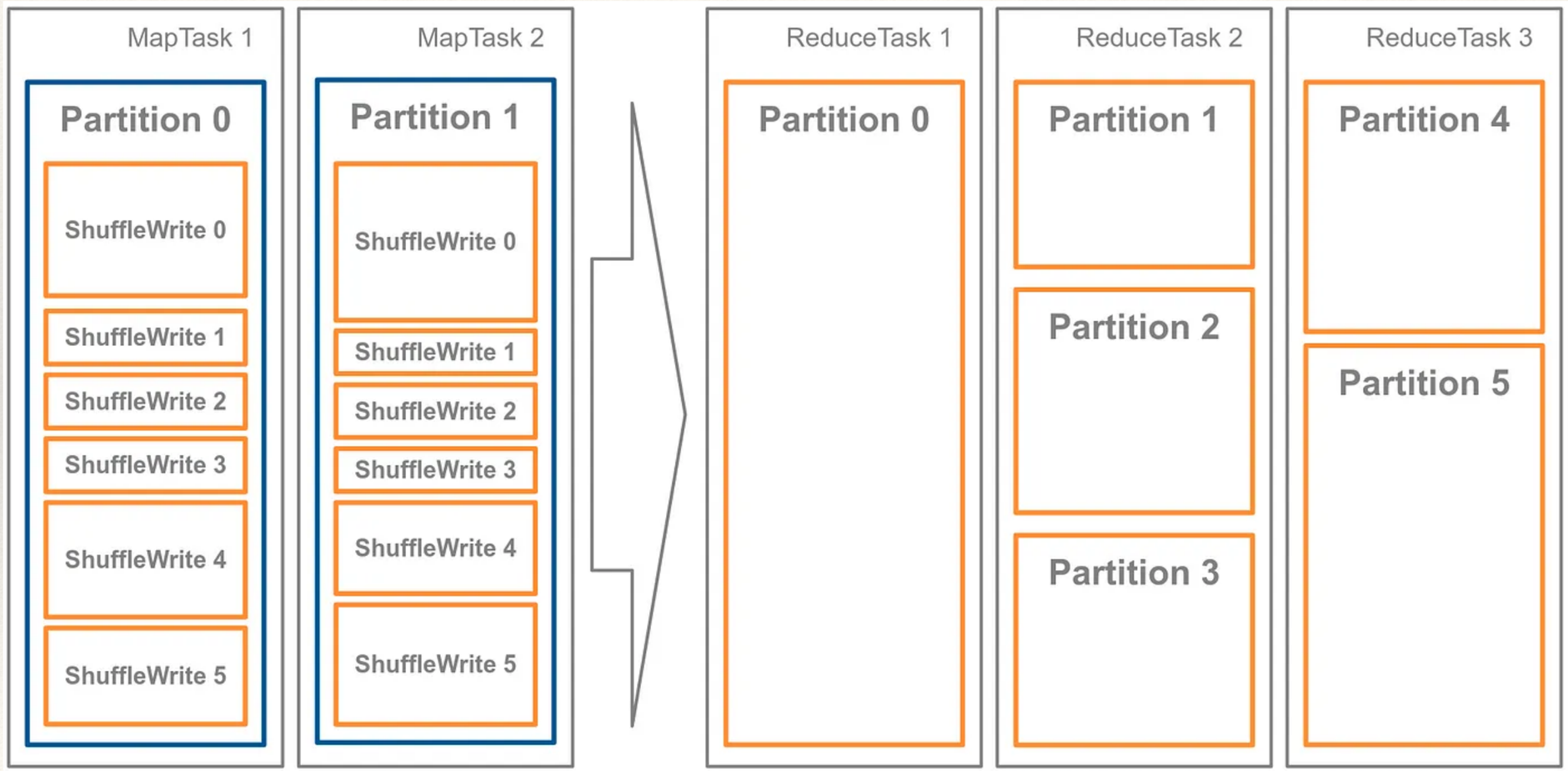
Parallelism on Reducers

- The level of parallelism after shuffling data for join or aggregation operations has shown to be critical when it comes to complex query performance.
  - If the number of reduce tasks is too low, partitions may go too large to fit in executor memory, forcing to spill data on disk and thus requiring more I/O operations.
  - On the other hand, more scheduling overhead is introduced as the number of reducers get larger, which will start degrading performance at a certain point.

Parallelism on Reducers

- CoallesceShufflePartitions rule enables Spark to dynamically lower the number of post-shuffle partitions by eventually merging adjacent small ones. By setting the optimal figure accordingly to runtime data sizes, load balancing improves and the number of requested I/O operations also decreases.

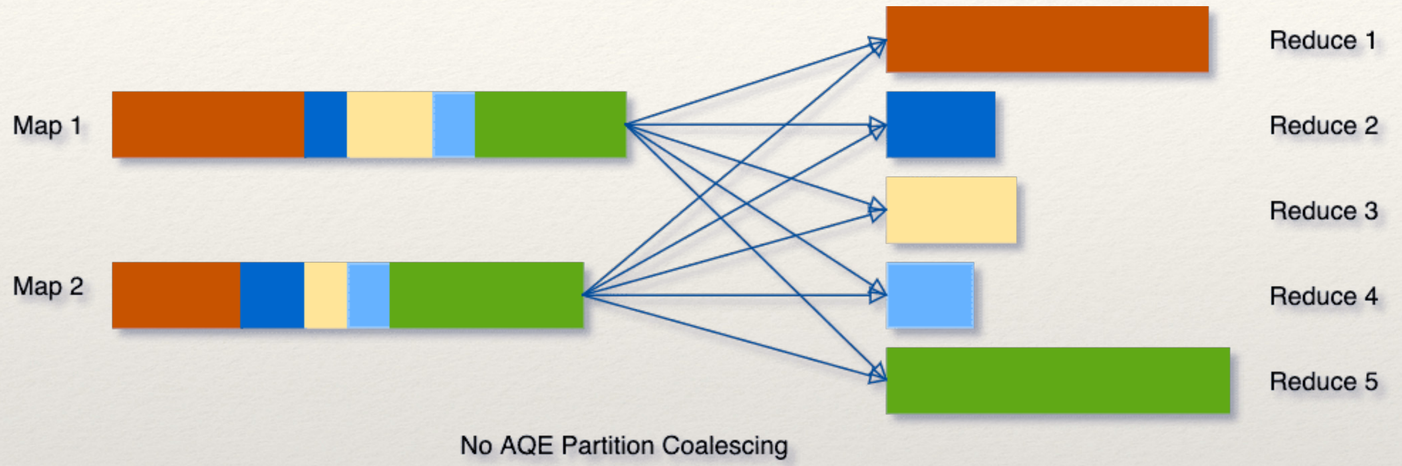
Partition merging after shuffling

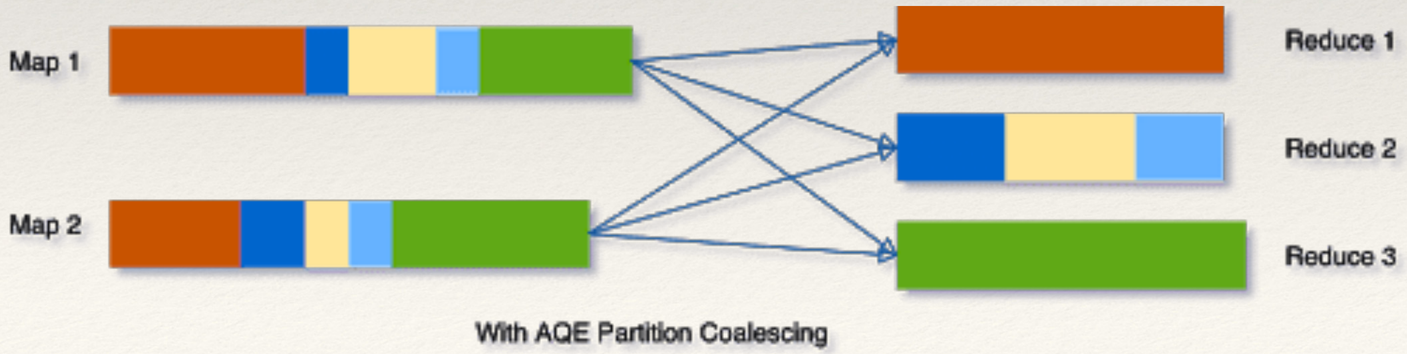


Parallelism on Reducers

- CoallesceShufflePartitions rule enables Spark to dynamically lower the number of post-shuffle partitions by eventually merging adjacent small ones. By setting the optimal figure accordingly to runtime data sizes, load balancing improves and the number of requested I/O operations also decreases.
- This feature coalesces the post shuffle partitions based on the map output statistics when both spark.sql.adaptive.enabled and spark.sql.adaptive.coalescePartitions.enabled configurations are true.
- This feature simplifies the tuning of shuffle partition number when running queries. You do not need to set a proper shuffle partition number to fit your dataset. Spark can pick the proper shuffle partition number at runtime once you set a large enough initial number of shuffle partitions via spark.sql.adaptive.coalescePartitions.initialPartitionNum configuration.

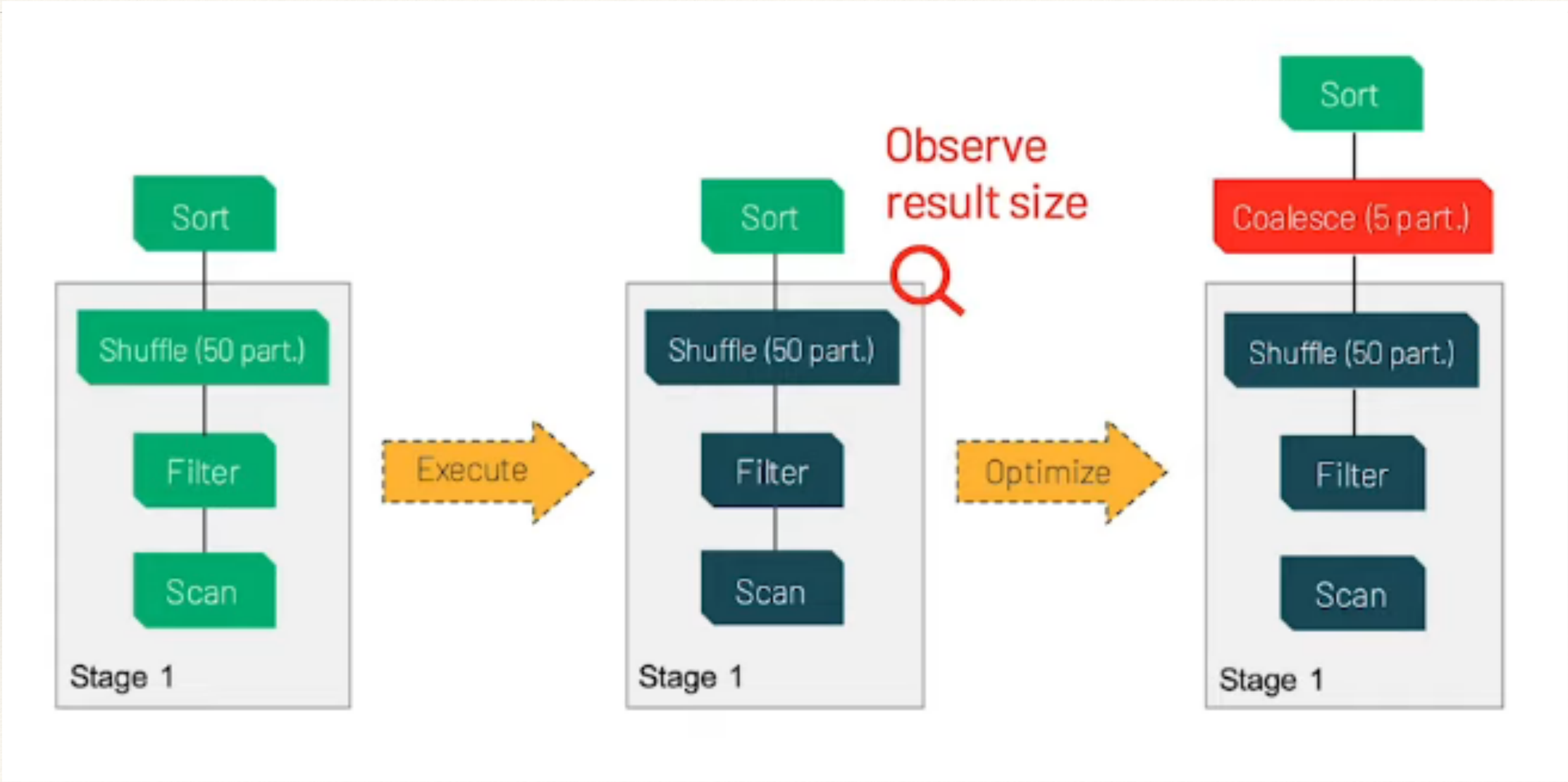
Parallelism on Reducers





source

Reducing Shuffle Partitions



source

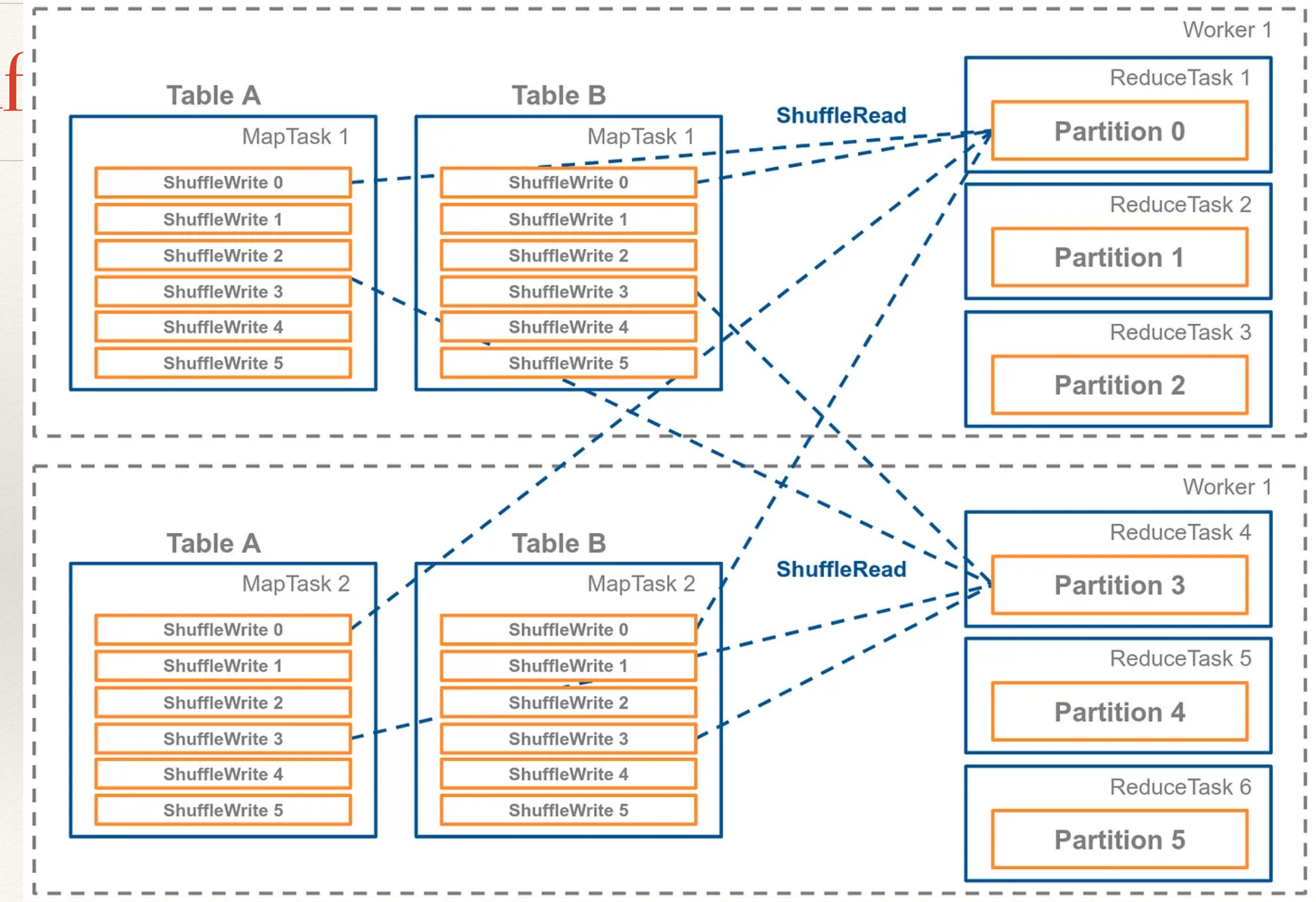
Parallelism on Reducers

```
spark.sql.shuffle.partitions = 200  (Spark 2.x)
spark.sql.adaptive.coalescePartitions.enabled = true
spark.sql.adaptive.coalescePartitions.minPartitionNum
spark.sql.adaptive.coalescePartitions.initialPartitionNum = 200
spark.sql.adaptive.advisoryPartitionSizeInBytes = 64MB
```

Join Strategy

- Another key decision affecting the performance of complex queries is the selection of a physical join strategy. Actually, Sparks implements three physical join operators:
  - Two operators first shuffle the data according to the join keys and then apply a join algorithm.
  - (1) **ShuffleHashJoin** performs a hash join, that is create a hash table (build) using content from one side and then scan the other one (probe).
  - (2) **SortMergeJoin** sorts both sides by the join attribute and merge the data by iterating over the elements and finding rows with the same key. Traffic exchange due to shuffle-join operations is shown in Fig. 6. Every reducer have to read one file from every mapper which implies transferring a lot of data over the network when tasks happen to run in different workers.

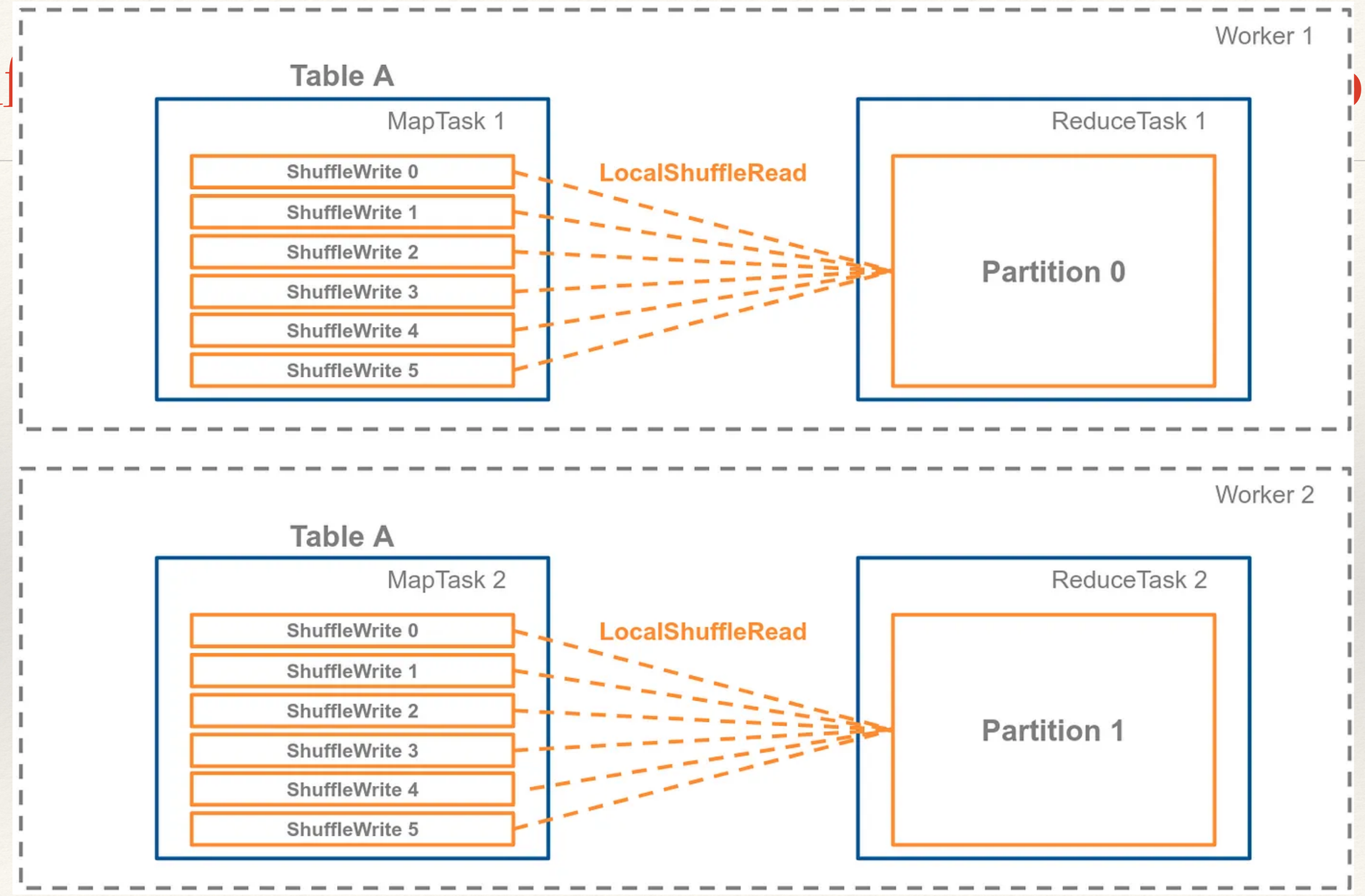
Shuf erator


Worker 1
Table A
MapTask 1
ShuffleWrite 0
ShuffleWrite 1
ShuffleWrite 2
ShuffleWrite 3
ShuffleWrite 4
ShuffleWrite 5
Table B
MapTask 1
ShuffleWrite 0
ShuffleWrite 1
ShuffleWrite 2
ShuffleWrite 3
ShuffleWrite 4
ShuffleWrite 5
ShuffleRead
ReduceTask 1
Partition 0
ReduceTask 2
Partition 1
ReduceTask 3
Partition 2
Worker 1
Table A
MapTask 2
ShuffleWrite 0
ShuffleWrite 1
ShuffleWrite 2
ShuffleWrite 3
ShuffleWrite 4
ShuffleWrite 5
Table B
MapTask 2
ShuffleWrite 0
ShuffleWrite 1
ShuffleWrite 2
ShuffleWrite 3
ShuffleWrite 4
ShuffleWrite 5
ShuffleRead
ReduceTask 4
Partition 3
ReduceTask 5
Partition 4
ReduceTask 6
Partition 5

Join Strategy

- (3) **BroadcastHashJoin** is preferable over the previous strategies when one of the sides is reasonably small to be broadcasted to and stored in each executor. this join implementation avoids to exchange data across nodes for co-partitioning. The number of reduce tasks equals that of the mappers, having to read each reducer the output from a single mapper. Usually, both tasks are scheduled on the same working node to avoid unnecessary network transfers.

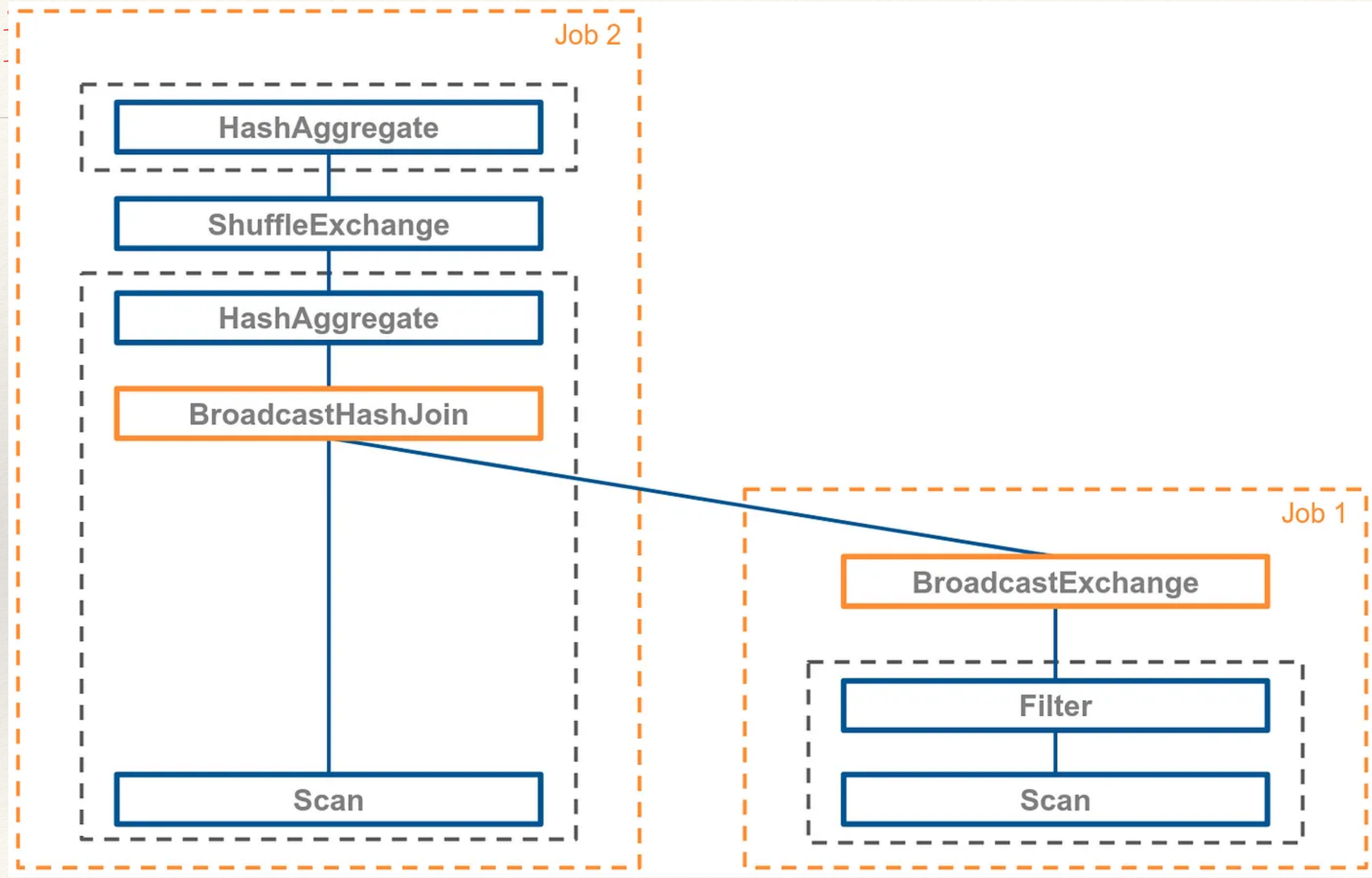
Shu[illegible]perator


Worker 1
Table A
MapTask 1
ShuffleWrite 0
ShuffleWrite 1
ShuffleWrite 2
ShuffleWrite 3
ShuffleWrite 4
ShuffleWrite 5
LocalShuffleRead
ReduceTask 1
Partition 0
Worker 2
Table A
MapTask 2
ShuffleWrite 0
ShuffleWrite 1
ShuffleWrite 2
ShuffleWrite 3
ShuffleWrite 4
ShuffleWrite 5
LocalShuffleRead
ReduceTask 2
Partition 1

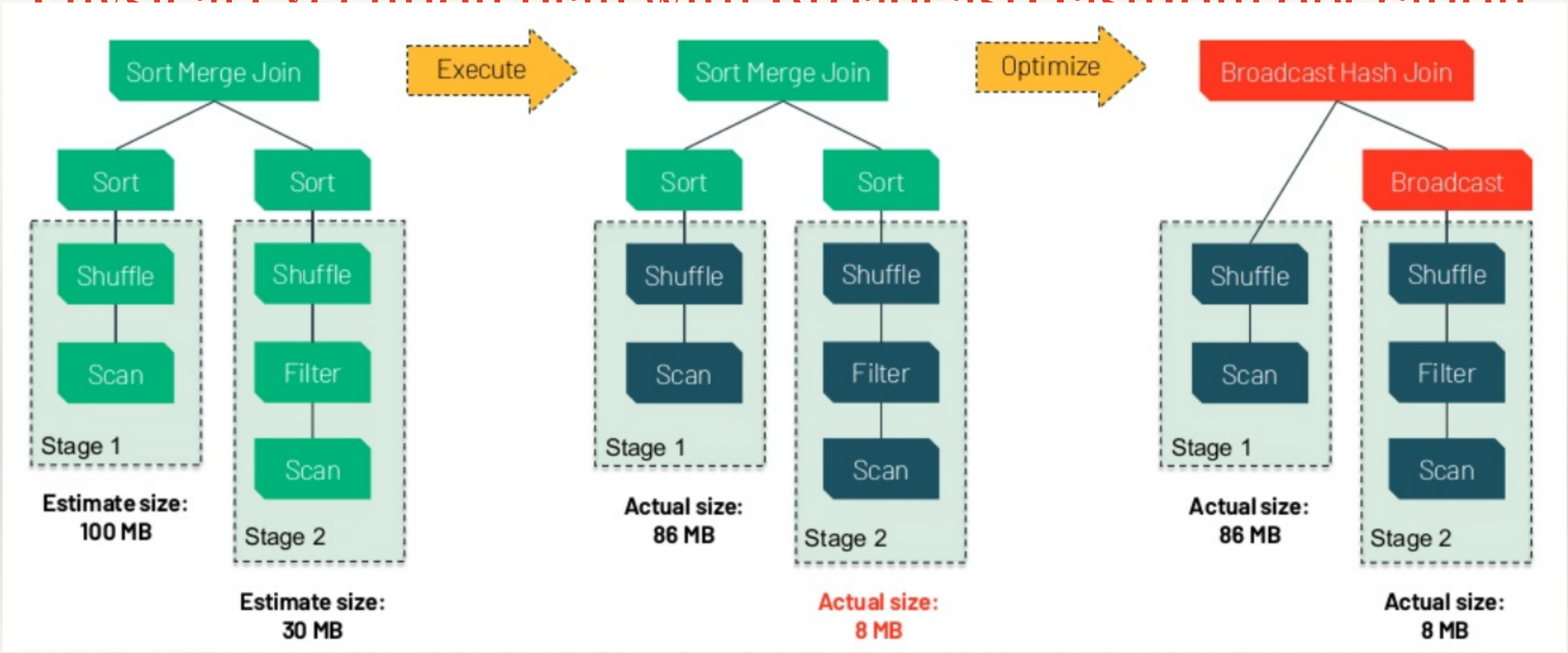
Join Strategy

- This rule enables Spark to avoid shuffle by dynamically replacing a Sort-Merge join for a Broadcast-Hash join implementation, when size conditions are met on any of its sides. OptimizeLocalShuffleReader rule may further optimize the plan by replacing regular shuffle reads for localized ones in order to reduce network usage (see Figs. 6,7), depending on configuration.

Phys eration



Physical execution plan with BroadcastHashJoin operation



source

Join Strategy

```
spark.sql.autoBroadcastJoinThreshold = 10MB
spark.sql.adaptive.localShuffleReader.enabled
```

Skewed Joins

- Another critical aspect in query performance is data distribution across partitions, which defines the size of the scheduled tasks. In case of unevenly distributed shards, tasks operating on a heavier amount of data will slow down the whole stage and block down-stream stages.
- This inconvenience was usually tackled by increasing the parallelism level or somehow re-engineering the join keys to increase cardinality.

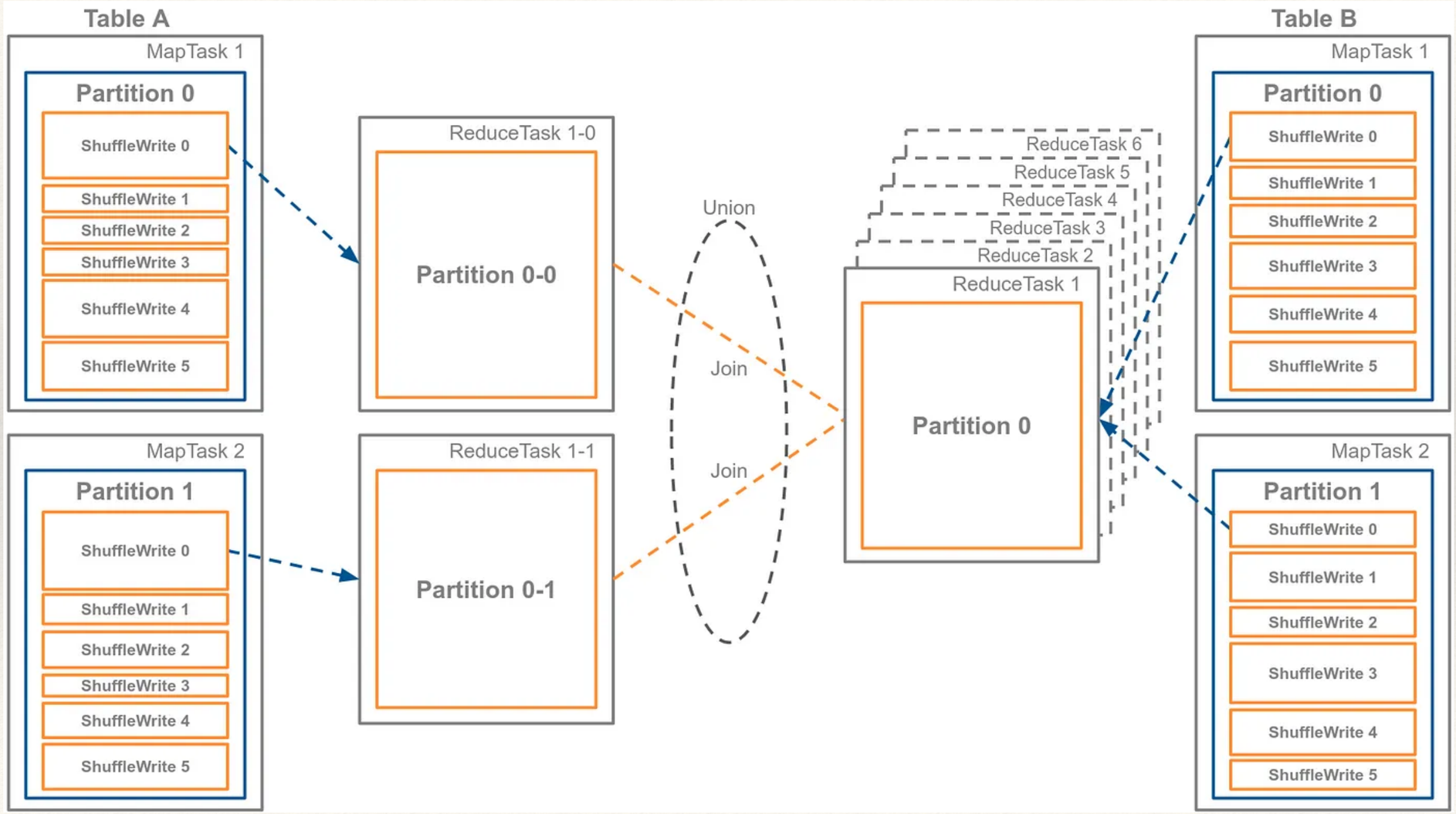
Skewed Joins

- OptimizeSkewedJoin rule enables Spark to achieve better load balancing by following a simple rule. Being parameters F the skewed factor, S the skewed size and R the skewed row count, a partition is considered skewed iff:
  - Its size is larger than S and larger than the median partition size multiplied by F.
  - Its row count is langer than R and larger than the median partition row count multiplied by F.

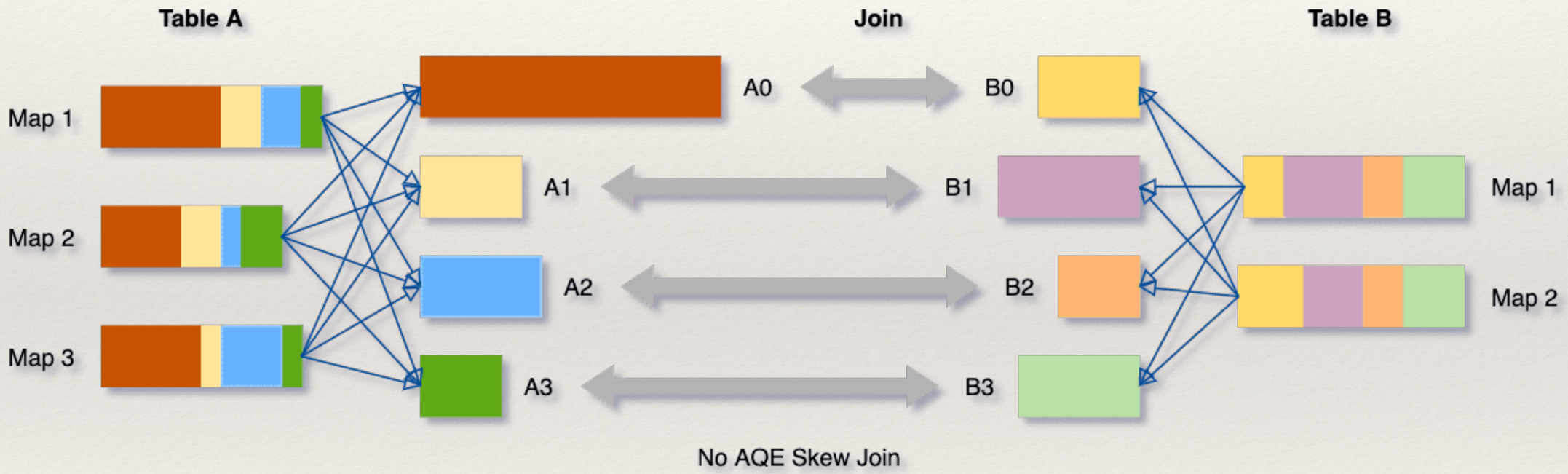
Skewed Joins

- When a partition is deemed skewed based on map-side statistics, multiple reducer tasks are scheduled to operate on it. This way, each of these reducers pulls the output written by a single mapper and performs the join operation on this partition chunk (instead of just one single reducer fetching files from every mapper).
- This schema is illustrated in Fig. 9, where partition 0 in table A is processed by two reducers to later union their outputs. OptimizeLocalShuffleReader rule may further optimize the physical plan.

Handling of a skewed join

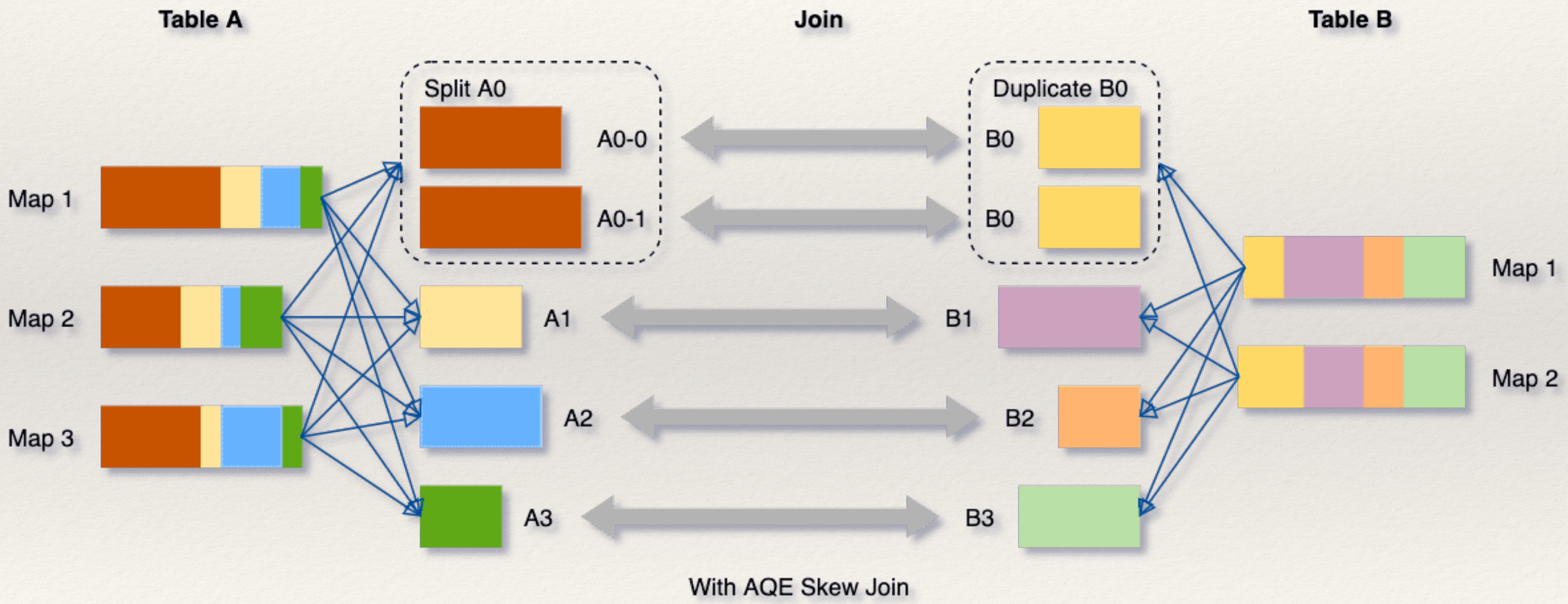


Handling of a skewed join



source

Handling of a skewed join



source

Skewed Joins

```
spark.sql.adaptive.skewJoin.enabled = true
spark.sql.adaptive.skewJoin.skewedPartitionFactor = 10
spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes = 256MB
spark.sql.adaptive.localShuffleReader.enabled
```

When not to use AQE

- Small jobs with minimal shuffle stages.
- Workloads with predictable and well-understood performance.
- Cases where manual tuning is more effective.

SQL Performance Tuning

https://spark.apache.org/docs/latest/sql-performance-tuning.html

Why is Join Followed by a Shuffle?

- A join often requires data from different partitions to be combined. For example, when performing a join between two DataFrames on a specific key, the rows with matching keys need to be brought together.
- In a distributed environment like Spark, **the data for the same key might reside on different partitions** (or even different nodes). To perform the join, **Spark must shuffle data between partitions so that all data with the same key is on the same partition**.
- This shuffling process involves redistributing the data across the cluster, which is why joins are often followed by a shuffle.

Does Shuffling Always Happen After a Join?

- Not necessarily. Whether a shuffle occurs after a join depends on several factors:
  - **Skewed Data**: If data is skewed, additional shuffling might be needed to handle the imbalance. AQE (Adaptive Query Execution) can optimize for this.
  - **Pre-existing Partitioning**: If the two DataFrames/RDDs being joined are already partitioned in the same way (e.g., by the join key), Spark can avoid a shuffle.
  - **Broadcast Joins**: When one of the DataFrames is small enough, Spark can perform a broadcast join, **which avoids shuffling by sending the smaller DataFrame to all worker nodes**. This is much faster than shuffling large datasets.
  - **Hinting and Optimizations**: With AQE enabled, Spark may dynamically adjust join strategies based on runtime statistics, potentially reducing or avoiding shuffles.

Transformations Causing a Shuffle

- 1. **groupBy and groupByKey**
  - What it does: Groups data by a specific key.
  - Why it causes a shuffle: Data with the same key needs to be colocated in the same partition to apply aggregation functions.
- 2. **reduceByKey and aggregateByKey**
  - What it does: Aggregates data by key using a specified associative and commutative reduce function.
  - Why it causes a shuffle: Similar to groupByKey, data with the same key is shuffled to the same partition for aggregation.
- 3. **join (including inner, outer, left, right joins)**
  - What it does: Combines rows from two DataFrames/RDDs based on a matching key.
  - Why it causes a shuffle: Data with the same key from both DataFrames/RDDs needs to be brought together, often requiring data movement across partitions.

Transformations Causing a Shuffle

- 4. **distinct**
  - What it does: Removes duplicate rows from a DataFrame/RDD.
  - Why it causes a shuffle: Spark needs to group the data to identify and remove duplicates, which typically involves a shuffle.
- 5. **repartition and coalesce**
  - What it does: Changes the number of partitions of a DataFrame/RDD.
  - Why it causes a shuffle:
    - repartition triggers a full shuffle because it redistributes data across all partitions evenly.
    - coalesce minimizes shuffling by merging partitions but can still trigger a shuffle if it results in more partitions being combined than the original layout.

Transformations Causing a Shuffle

- 6. **sortBy and sortByKey**
  - What it does: Sorts data within each partition or globally across the entire DataFrame/RDD.
  - Why it causes a shuffle: Sorting usually requires all data to be ordered, often necessitating data movement across partitions to get the global order.
- 7. **cogroup**
  - What it does: Groups data from two RDDs by key and returns each group as a pair of collections.
  - Why it causes a shuffle: Like a groupBy operation, data with the same key needs to be colocated in the same partition, requiring a shuffle.

Transformations Causing a Shuffle

- 8. **union** (when combining multiple RDDs/DataFrames)
  - What it does: Merges two or more RDDs/DataFrames.
  - Why it causes a shuffle: When the resulting RDD/DataFrame needs to be repartitioned (for example, to achieve even distribution), a shuffle may occur.