READ MODE · ANSWER FIRST

How Spark Joins Work

An interactive, visual explanation of How Spark Joins Work, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

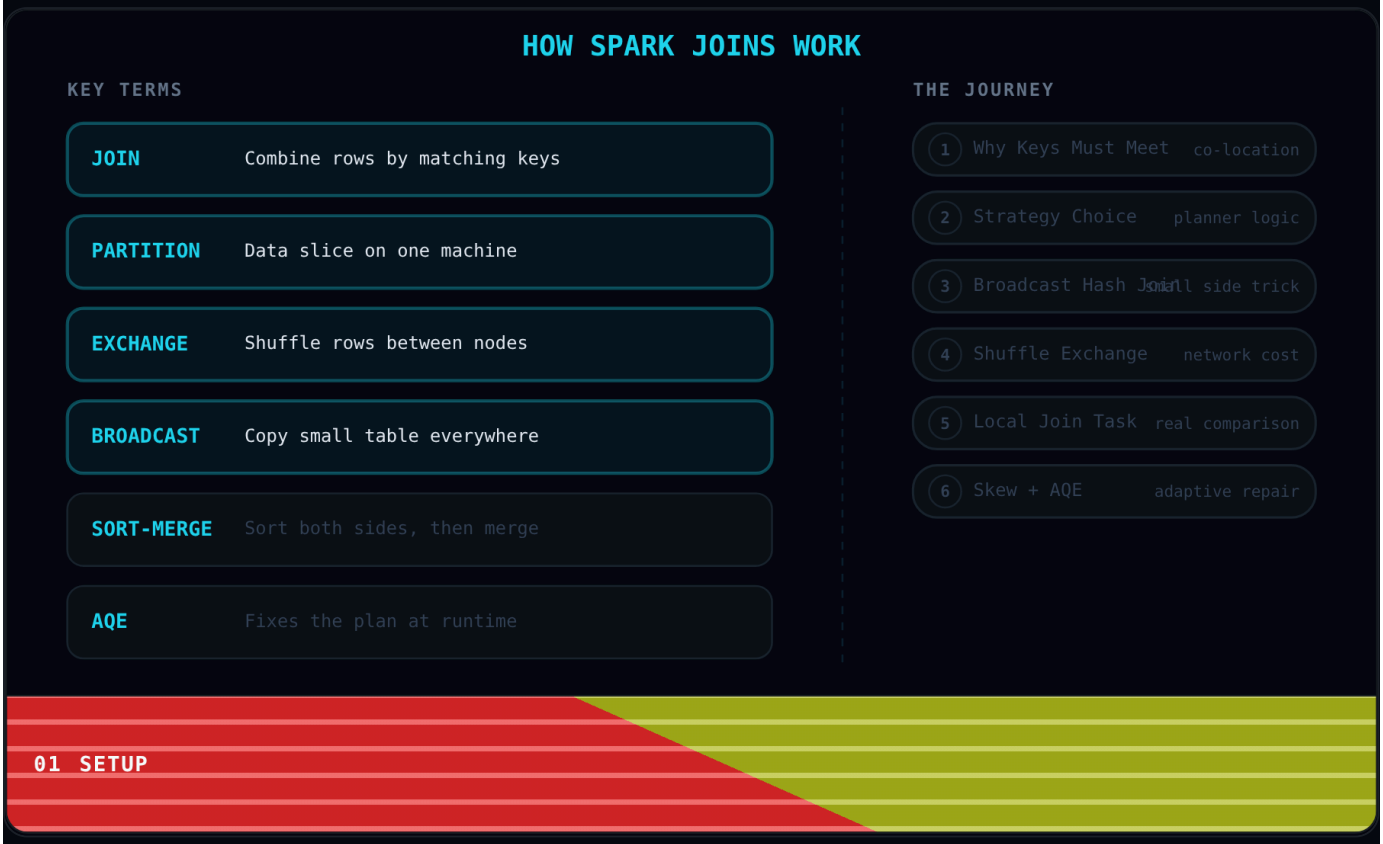
The whole story in 6 lines

How Spark Joins Work becomes easier to reason about when every stage is connected as one system.

1. The expensive part of a distributed join is usually data movement, not the final comparison itself.
2. Broadcast is fast when one side is tiny. Otherwise Spark usually pays for a shuffle and joins partition by partition.
3. Broadcast avoids the double shuffle. The tradeoff is executor memory: every worker now keeps a copy of the small side.
4. This exchange is usually the dominant cost of a large join: serialization, network transfer, spill risk, and task imbalance all show up...
5. The distributed exchange gets the attention, but correctness still depends on the local task respecting duplicates and outer-join null...
6. AQE does not invent a new join. It repairs the physical plan after Spark has seen the real partition sizes instead of trusting static...

STAGE 1 OF 8

Setup



Welcome. This explainer walks through how Apache Spark executes a join, step by step. We will start with six key terms you will see throughout.

A join combines rows from two tables wherever their keys match. Think of it like matching student IDs between an enrollment list and a grade sheet.

A partition is a slice of data that lives on one machine. Spark splits large tables into many partitions so work can happen in parallel.

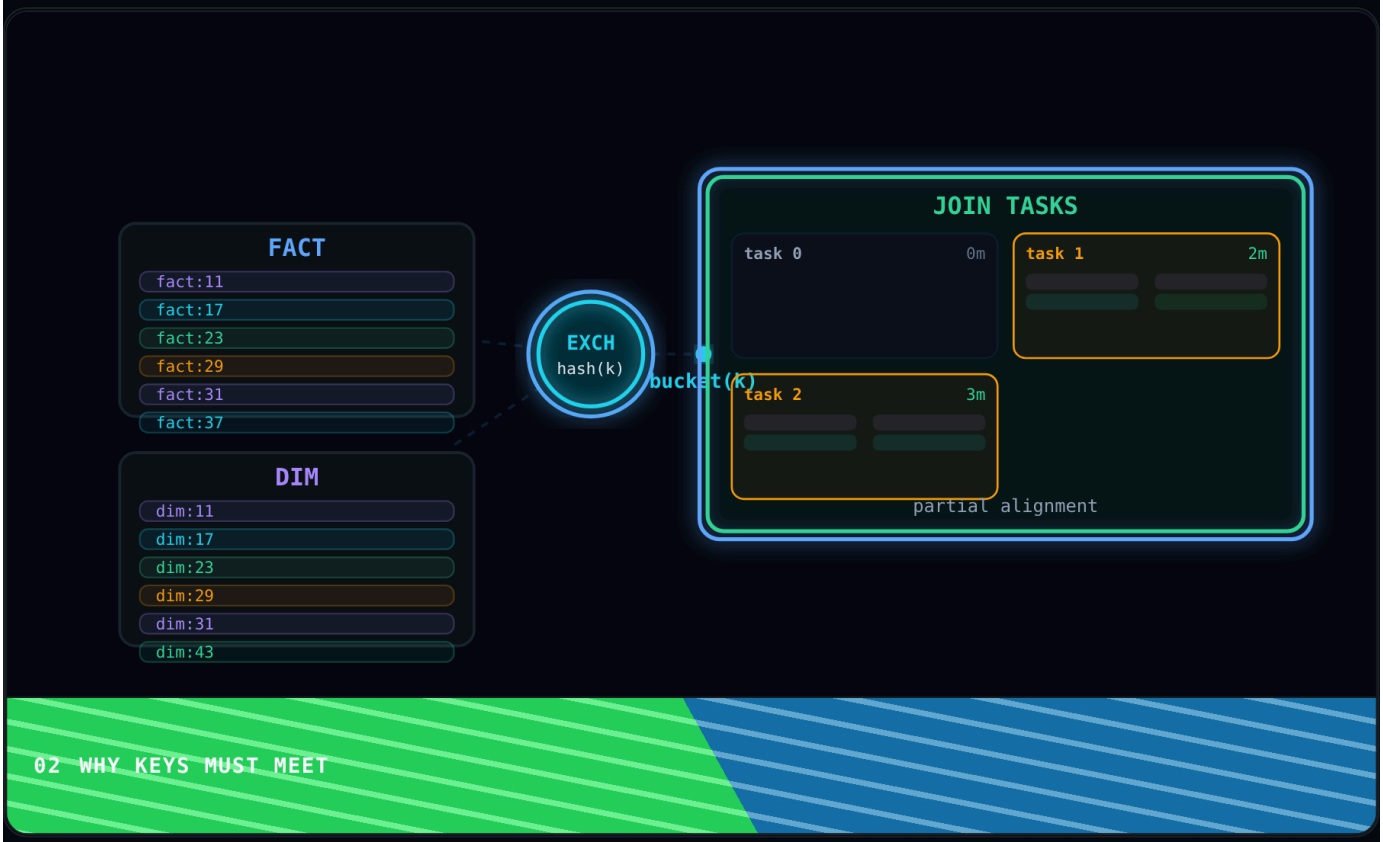
An exchange is a network shuffle. Spark moves rows between machines so that matching keys end up on the same executor.

Now let us preview the road ahead. We will cover six stages: why keys must meet, how Spark picks a strategy, broadcast joins, shuffle exchanges, local join tasks, and adaptive execution.

Now let us start with the most fundamental idea: why matching keys need to be on the same machine before any comparison can happen.

STAGE 2 OF 8

Why Keys Must Meet



02 WHY KEYS MUST MEET

Here is the core question: how do two huge tables, scattered across a cluster, find their matching rows? The answer starts with the fact table on the left.

The dimension table holds the rows we want to match against. Right now these keys live on completely different machines than the fact rows. No comparison is possible yet.

Spark inserts an exchange boundary. This is the moment where rows leave their original partitions and travel across the network, grouped by join key.

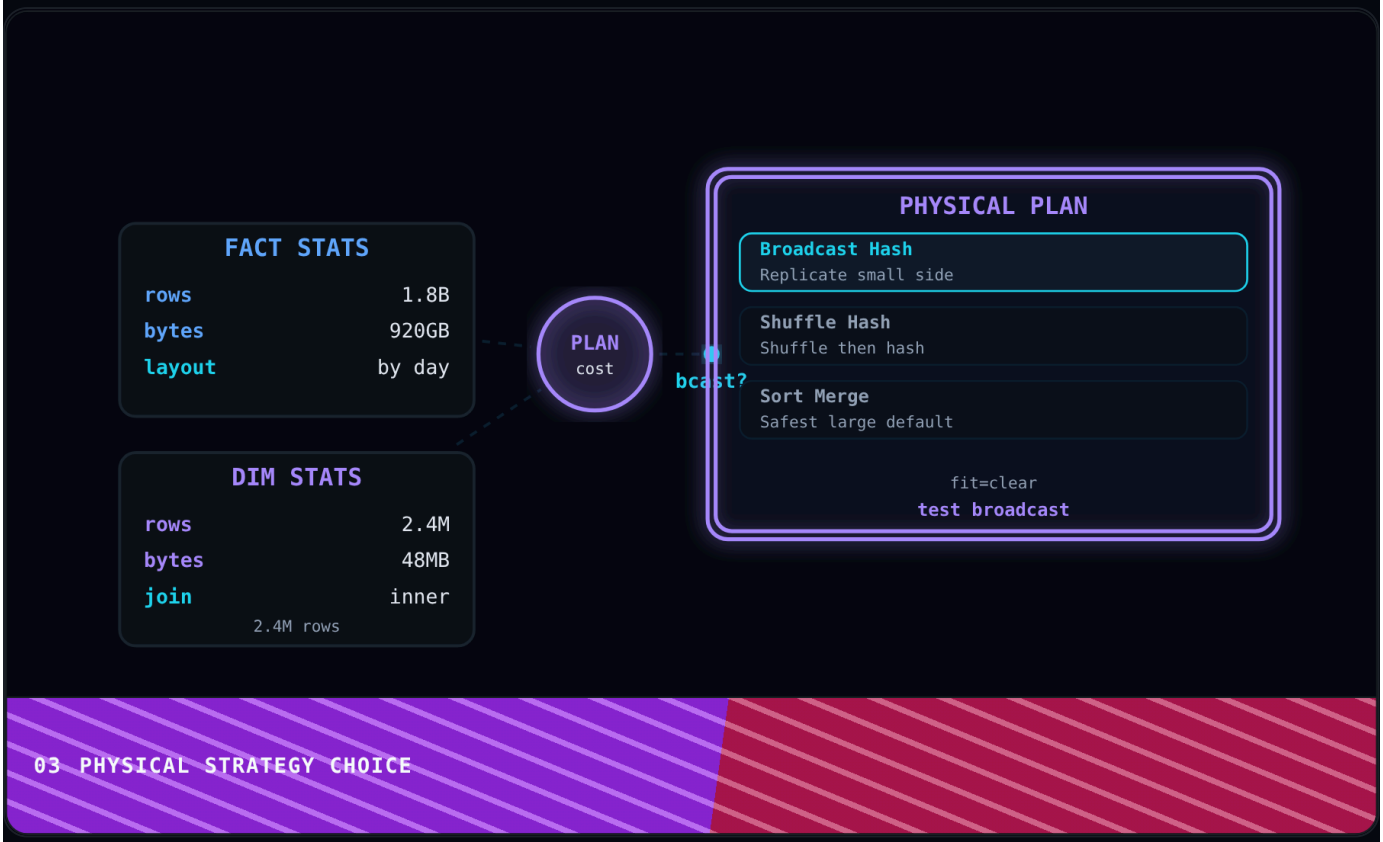
Both sides are re-bucketed using the same hash function. Watch how rows with identical keys land in the same bucket. Try changing the Workers control in the sidebar to see how bucket assignment shifts.

Matching keys now share a task slot. Each bucket contains all the left rows and all the right rows for its key range. Switch the Skew control to Hot Key and notice how one bucket gets overloaded.

Local join tasks can finally run. No more network traffic is needed. The key takeaway: the expensive part of a distributed join is moving data, not comparing it. Next, we will see how Spark decides which join strategy to use.

STAGE 3 OF 8

Physical Strategy Choice



03 PHYSICAL STRATEGY CHOICE

Now that we understand why keys must meet, let us look at how Spark chooses the best way to make that meeting happen. It starts with a logical join and table statistics.

The planner reads the fact side first. It needs to know how many rows and how many bytes it is dealing with. These numbers set the baseline cost for every strategy.

Next it checks the dimension side. The critical question: is it small enough to broadcast? Change the Dim Size control in the sidebar and watch the threshold check shift.

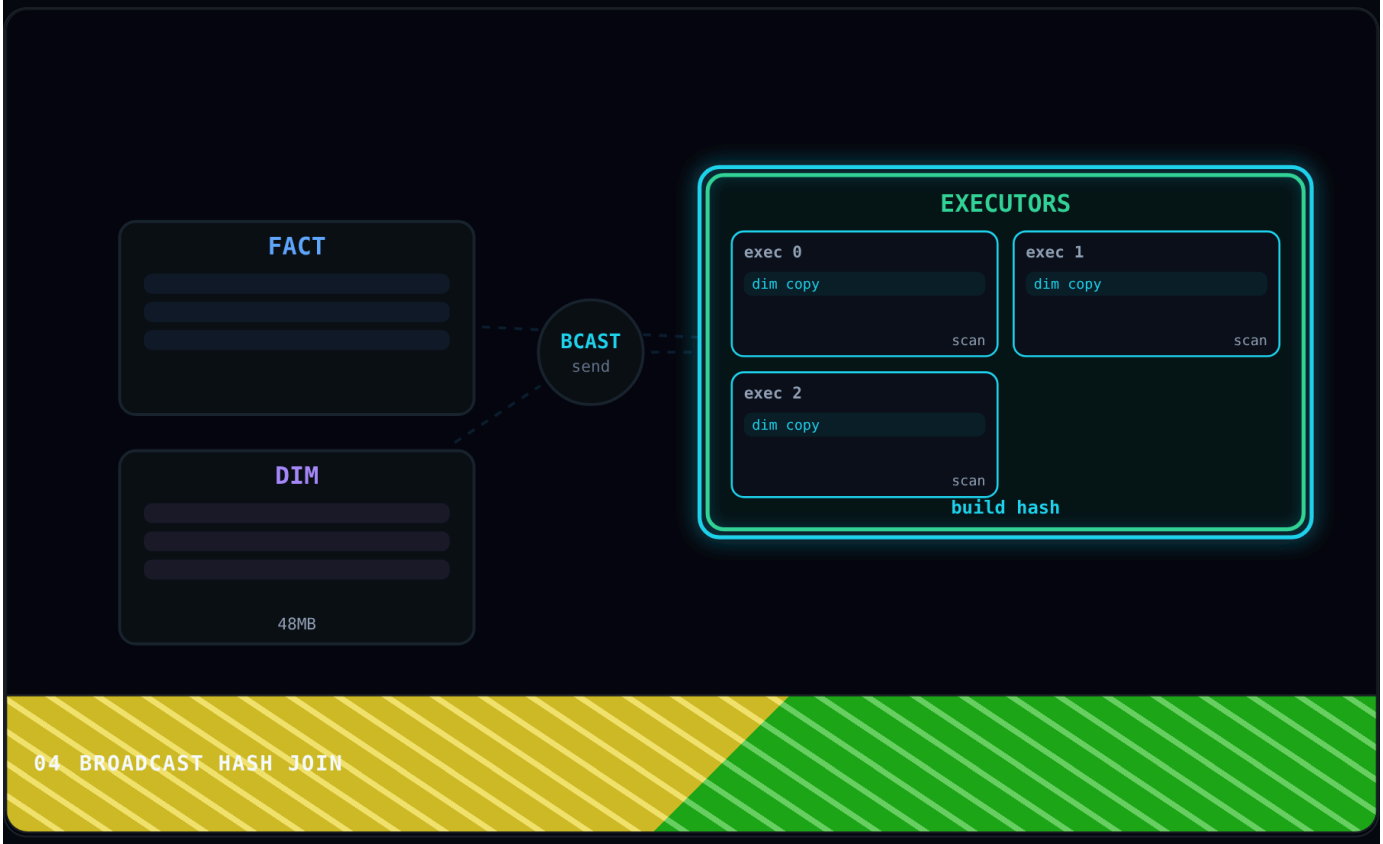
If broadcast is off the table, the planner evaluates shuffle-based strategies instead. A shuffle hash join builds a local hash table. A sort-merge join sorts both sides.

The candidate costs are ranked side by side. Toggle the Stats control off. Notice how stale statistics push the planner away from broadcast even when the dimension is small.

Spark emits one concrete physical operator. That single decision determines how much network traffic, memory, and CPU the join will cost. Next, we will see the fastest option: the broadcast hash join.

STAGE 4 OF 8

Broadcast Hash Join



We just saw how the planner picks a strategy. When it chooses broadcast, here is what actually happens. The driver starts with a physical plan that says "broadcast the small side."

The dimension table is serialized into one compact payload. This is the data that every executor will receive. Watch the payload size change when you switch the Dim Size control.

The driver ships that payload to every executor in the cluster. This is the one network trip. After this, the large side never moves.

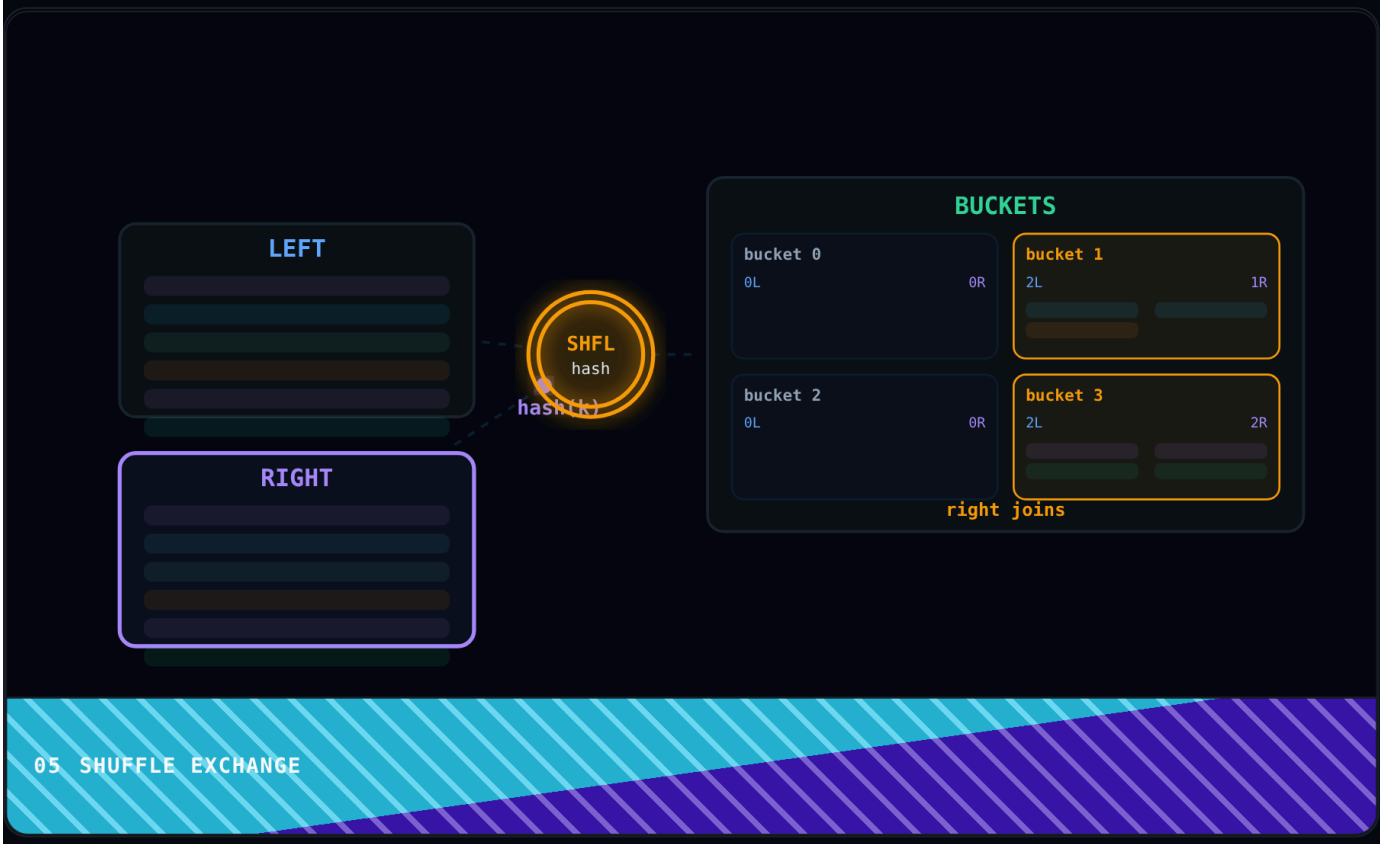
Each executor unpacks the broadcast and builds a local hash table keyed on the join column. This hash table fits entirely in memory. Add more Workers in the sidebar and notice each one gets its own copy.

Fact partitions stream past the hash table. For each fact row, the executor looks up the join key in the hash table. A match means an output row. No match means the row is skipped.

The joined output appears without ever shuffling the large side. That is the broadcast tradeoff: one small copy per executor versus zero network cost for the big table. Next, what happens when the dimension is too large to broadcast.

STAGE 5 OF 8

Shuffle Exchange



05 SHUFFLE EXCHANGE

Now that we have seen broadcast, let us look at what happens when the dimension is too big. Remember from Stage 1 that keys must meet on the same machine. Without broadcast, both sides must shuffle.

Spark attaches shuffle writers to the left side. Every row gets hashed by the join key, and the hash determines which output partition the row lands in.

The right side does the same using the identical hash function. This is critical. Both sides must agree on the partitioner or keys will end up in different buckets.

Rows travel across the network into their hash buckets. This is the most expensive step. Every row from both tables crosses a network boundary. Change the Partitions control to see how finer partitioning redistributes the load.

Each bucket now contains all matching keys from both sides. Switch the Skew control to Hot Key. Notice how one bucket balloons while others stay small. That imbalance will slow down the entire stage.

The shuffle is done. Aligned partition pairs are ready for local join tasks. The key cost was network serialization and transfer. Next, we will see what happens inside one of these local tasks.

Local Join Task

06 LOCAL JOIN TASK

We saw how the shuffle aligns partitions. Now let us zoom into one executor task that receives a left partition and a right partition. This is where actual row comparison happens.

The task prepares its local state. Switch the Algorithm control in the sidebar. In sort-merge mode, both sides get sorted by key. In hash mode, the right side builds an in-memory hash table.

Keys are compared inside this single executor. Sort-merge walks two sorted streams with a cursor. Hash mode probes the hash table for each left row. Either way, no network traffic is involved.

Duplicate keys create a cross product. If key 42 appears twice on the left and twice on the right, that is four output rows. Increase the Dup Keys control to see the output grow.

Switch the Join Type to Left. Notice the null-padded rows that appear for unmatched left keys. An inner join would simply skip those rows. This is where join semantics determine the final shape of the output.

The task emits its slice of the final result. Every partition produces output independently. The key insight: correctness depends on handling duplicates and nulls correctly at this local level. Now let us step back and see the whole picture together.

STAGE 7 OF 8

Skew + AQE

07 SKEW + AQE

Remember the hot bucket from Stage 4? After the shuffle finishes, Spark now has real partition sizes instead of estimates. These runtime metrics are AQE's starting point.

The metrics reveal that partition 2 holds far more data than the others. Spark flags it as a skewed outlier. Without intervention, this one task will take longer than all the rest combined.

AQE decides whether it can safely rewrite the plan. Toggle the AQE control off in the sidebar. Without AQE, Spark is stuck with the original skewed plan. Turn it back on to see the fix.

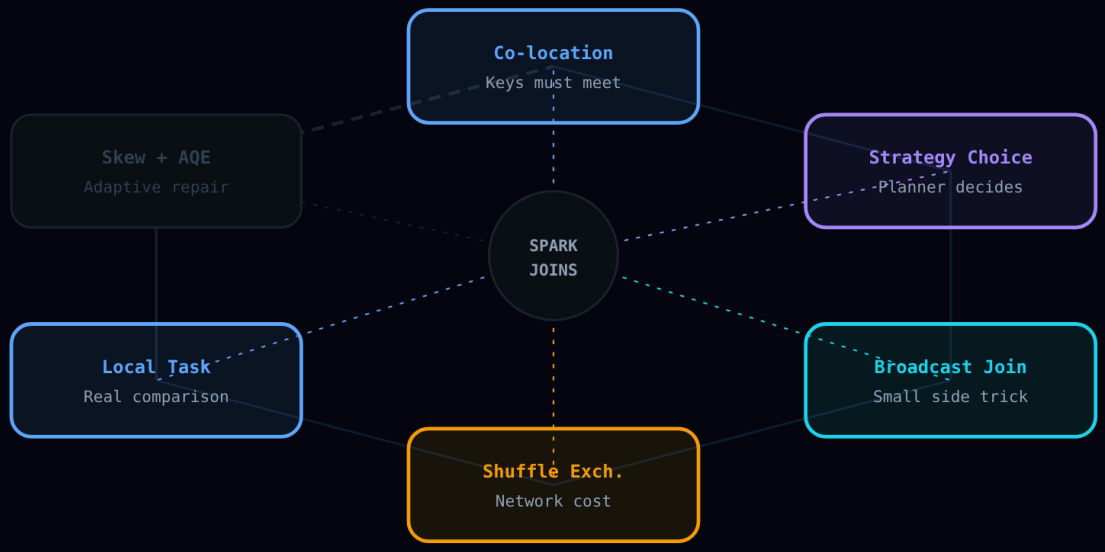
The hot partition is split into smaller chunks. Spark creates follow-up tasks and replicates the opposite side of the join into each chunk. More tasks, but each one finishes quickly.

The repaired tasks finish faster and more evenly. The wall-clock time for the stage drops because no single task dominates. Switch the Skew control to Balanced and notice AQE has nothing to fix.

The final output is logically identical. AQE changed only the physical execution, not the result. That is the whole story of Spark joins: strategy selection, data movement, local execution, and adaptive repair.

STAGE 8 OF 8

Recap



08 RECAP

We started with the most basic requirement: matching join keys must be on the same machine. Without co-location, no comparison is possible. That is the foundation everything else builds on.

Then we learned how the Catalyst planner chooses a physical strategy. It reads table statistics, checks the broadcast threshold, and picks the cheapest path. The choice depends on table sizes, join type, and stat freshness.

When one side is small, broadcast hash join avoids shuffling the large table entirely. One copy per executor, zero network cost for the big side.

When broadcast is not an option, both sides pay for a shuffle exchange. Rows travel across the network into hash buckets so matching keys land together.

Inside each executor, the local join task does the real work: sorting and merging, or hashing and probing. Duplicate keys fan out, outer joins add null-padded rows.

Finally, AQE watches runtime metrics and repairs the plan when skew shows up. It splits hot partitions without changing the logical result.

That is the full pipeline. Strategy selection, data movement, local execution, and adaptive repair. Every Spark join you write follows this path.