⚙️ Spark Execution Cheatsheet for ELT

This cheatsheet summarizes the Spark concepts we covered, tailored for your work on data ingestion in a **Databricks Cluster** environment. Understanding these concepts is the key to writing efficient PySpark code that runs fast on large-scale data.

I. Cluster Architecture: Who Does What?

The Spark architecture is a separation of duties, ensuring maximum parallelism.

Component	Role in the System	Key Property
Cluster	The entire pool of physical machines (Nodes).	Provides the distributed environment.
Node	An individual computer within the cluster (Server/VM).	A physical machine with its own CPU and RAM.
Driver	The single process that runs your PySpark script. It builds the logical plan (**DAG**) and coordinates all work.	**The Head Chef.** Resides on one Node.
Executor	Multiple processes that run on the worker Nodes. They execute the **Tasks** and hold the data **Partitions**.	**The Workers.** Do the actual data crunching.
Task	The smallest unit of work. **One Task processes exactly one Partition.**	The instruction manual for an Executor.

II. Practical Configuration: Sizing Executors

In your Databricks cluster, the configuration of **Executors** is critical to performance. It's determined by the resources of a single **Worker Node**.

Goal: Maximize parallelism without overwhelming the Node's resources.

A. Executor Calculation Example

If a standard **Worker Node** has **64 GB RAM** and **16 CPU Cores:**

1. **Cores per Executor (CPU Constraint):** The best practice is to reserve 1-2 cores for system overhead and use 4-5 cores per Executor process.

$$\frac{16 \text{ Cores}}{4 \text{ Cores per Executor}} = 4 \text{ Executors per Node}$$

2. **Memory per Executor (RAM Constraint):** Divide the remaining usable RAM by the number of Executors. (Assume 4 GB of RAM is reserved for the OS/system).

$$\frac{64 \text{ GB} - 4 \text{ GB (System)}}{4 \text{ Executors}} = \mathbf{15} \text{ GB per Executor}$$

Configuration: You would configure Spark to use 4 Executors, each with 15 GB of memory.

B. Partitions: The Unit of Work

Concept	Definition	Importance for Speed
Partition 🧩	A logical chunk of rows from your JSON files.	**Parallelism:** The total number of Partitions determines how many Tasks can be run simultaneously by your Executors.

III. The Execution Flow: Transformations and Actions

Spark runs your PySpark code lazily, only executing when an **Action** is called.

A. Lazy vs. Eager Operations

Operation Type	Definition	Key Feature
Transformation	An operation that creates a new DataFrame (e.g., `filter`, `select`, `join`).	**LAZY.** The Driver only adds the step to the logical plan (DAG).
Action 🍽️	An operation that returns a result or writes data (e.g., `count`, `show`, `write`).	**EAGER.** Forces the Driver to execute all preceding Transformations.

B. Narrow vs. Wide Transformations

This is the most critical distinction for performance optimization.

Transformation Type	Characteristic	Examples	Performance Cost
Narrow	Data required for a result is entirely within the current Partition. **No data needs to leave the Node.**	`select()`, `filter()`, `union()`, `withColumn()`	**Fast.** Very cheap to execute.
Wide	Data required for a result is scattered across multiple Partitions/Nodes.	`groupBy()`, `join()`, `orderBy()`, `distinct()`	**Slow.** Triggers a **Shuffle**.

IV. Detailed: Wide Transformation and the Shuffle Boundary

A **Wide Transformation** acts as a **Shuffle Boundary**, breaking the overall job into multiple **Stages** of execution. This is where most of your job's time will be spent.

The Shuffle Process: Two Stages

Stage 1: Pre-Shuffle (The Map Phase / The Prep)

1. **Tasks Start:** The **Driver** assigns the initial set of **Tasks** to the **Executors**.
2. **Local Work:** Each of the 4 Executors processes its Partitions (e.g., reads the JSON data and applies any *Narrow* transformations like column renaming).
3. **Intermediate Write:** When the Executor hits the `groupBy` command, it writes the necessary records (e.g., all 'East' region records) to its **local disk**.
4. **Data Send:** The Executor sends this intermediate data across the network to the specific **target Executor** that will finalize the aggregation.

Shuffle Boundary (The Move) 🌊

- This is the time spent waiting for all the data to travel across the network between the different worker **Nodes**.

Stage 2: Post-Shuffle (The Reduce Phase / The Final Cook)

1. **New Partitions:** The data is now ready in its new location, forming a new set of **Partitions** (e.g., one Partition now contains ALL 'East' records).
2. **New Tasks:** The **Driver** starts the second **Stage**, creating a new set of **Tasks** for the final aggregation (the final `count`).
3. **Final Aggregation:** The **Executors** pick up these new Tasks and calculate the single, correct result (e.g., the total count for the 'East' region).

🏆 Optimization Strategy (for Joins)

To prevent the Shuffle, use the **Broadcast Join** when joining a large DataFrame (your JSON data) with a small DataFrame (a reference table).

```
from pyspark.sql.functions import broadcast

df_sales.join(
    broadcast(df_small_lookup_table), # <-- Tells Spark to copy this to all Executors'
    on='key_column'
)
```

This is a **Narrow Transformation** because the small table is now locally available on every Executor, avoiding the need for network data exchange.