

Cluster Topography & Execution Hierarchy

Architecture

```
# Cluster Submission Architecture & Execution Model
Application
├── Driver (Coordinates DAGScheduler, TaskScheduler, SparkContext)
│   └── Job (Triggered by an Action: .count(), .collect(), .save())
│       └── Stage (Boundary determined by Shuffle Exchanges)
│           └── Task (Individual computation executed on 1 Partition)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Driver:** Orchestrates execution, translates logical DataFrame plans into physical execution stages, and monitors executor health.
- > **DAGScheduler:** Evaluates operator graphs, splits stages along wide shuffle boundaries, and optimizes task scheduling.
- > **TaskScheduler:** Dispatches individual partition tasks to worker executor slots based on data locality preferences (NODE_LOCAL, PROCESS_LOCAL).
- > **Executor:** Dedicated JVM process per worker node executing assigned tasks in parallel threads and caching partition blocks.

Golden Rule: 1 Task = 1 CPU Core = 1 Data Partition. Cluster throughput depends on balancing task counts to total available executor cores.

Dependency Graph: Narrow vs Wide Transformations

Execution DAG

```
# Narrow Transformations (1:1 Partition Mapping - NO Network Shuffle)
df_narrow = (df
    .filter("amount > 100")      # Local row filter
    .select("id", "amount")      # Local projection
    .withColumn("tax", F.col("amount") * 0.18)) # Local arithmetic

# Wide Transformations (M:N Shuffle Exchange - Stage Boundary Created!)
df_wide = (df_narrow
    .groupBy("id").agg(F.sum("amount")) # Hash shuffle across nodes
    .distinct()                        # Deduplication shuffle
    .join(df_other, on="id"))          # SortMerge shuffle
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Narrow Dependencies:** Partitions are processed independently in parallel within the same executor thread without network transfers (pipelined).
- > **Wide Dependencies:** Require data redistribution across nodes based on hash partitioning; forces a stage completion boundary.
- > **Shuffle Exchange:** Wide transformations serialize data into shuffle blocks on disk, initiating high network I/O across executors.
- > **Inspection:** Execute `df.explain()` to verify stage boundaries; look for `Exchange hashpartitioning` operators.

Architect Principle: Structure transformations so narrow operations filter and prune data prior to invoking wide shuffle operations.

Catalyst Query Optimizer & Whole-Stage CodeGen

Catalyst / Tungsten

```
# Inspect Catalyst Execution Pipeline
df.explain(mode="formatted")

# 4 Stages of Catalyst:
1. Unresolved Logical Plan: Verifies SQL syntax & AST
2. Analyzed Logical Plan: Resolves column names against Catalog
3. Optimized Logical Plan: Pushes down predicates & prunes columns
4. Physical Plan: Selects algorithms (HashJoin vs SortMerge)
   └── Whole-Stage CodeGen: Emits Java Bytecode (clean for loops)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Predicate Pushdown:** Pushes filter conditions directly into storage file scans, skipping entire row groups before data enters JVM memory.
- > **Column Pruning:** Reads only projected columns from columnar Parquet storage, avoiding deserialization of unused wide table fields.
- > **Cost-Based Optimizer (CBO):** Evaluates table statistics to choose optimal join orders and select broadcast joins automatically.
- > **Whole-Stage CodeGen:** Collapses multiple physical operators into a single Java bytecode loop, eliminating virtual function invocation costs.

Plan Indicator: In `df.explain()`, an asterisk `*` confirms that the operator is compiled with Whole-Stage CodeGen!

JVM Unified Memory Management (Heap & Off-Heap)

Memory Pools

```
# Total Executor Memory Breakdown (JVM + Off-Heap)
Total Memory = Executor Heap (spark.executor.memory)
              + Memory Overhead (spark.executor.memoryOverhead)
              + Off-Heap Memory (spark.memory.offHeap.size)

JVM Heap Pools:
├── Reserved Memory: 300 MB (Hardcoded internal Spark engine safety)
├── User Memory: (JVM - 300MB) * (1 - 0.6) = ~40% (UDFs, metadata, hashes)
├── Unified Spark Memory: (JVM - 300MB) * 0.6 = ~60%
│   ├── Storage Pool: ~50% (Data caching & broadcast variables)
│   └── Execution Pool: ~50% (Shuffles, joins, sorts, aggregations)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Dynamic Borrowing:** Execution and Storage pools share a unified boundary; execution can evict cached storage blocks if memory is constrained.
- > **Memory Overhead:** Off-heap native memory for Py4J communication, Netty shuffle network buffers, and OS page cache (typically 10% of heap).
- > **Off-Heap Tungsten:** `spark.memory.offHeap.enabled = true` stores data outside Java GC in native C-style memory, eliminating GC pauses.
- > **OOM Prevention:** If execution memory overflows the unified pool, data spills to local NVMe SSD disks (Spill Memory / Spill Disk).

Tuning Knob: `spark.memory.fraction = 0.6` controls the size of the combined execution/storage pool relative to total JVM heap.

Shuffle Mechanics & Spill Bottlenecks

Engine Bottlenecks

```
# Shuffle Tuning Knobs
# 1. Shuffle write buffer (reduces disk write syscalls)
spark.conf.set("spark.shuffle.file.buffer", "1m") # Default 32k

# 2. Max size of remote fetch requests in flight
spark.conf.set("spark.reducer.maxSizeInFlight", "96m") # Default 48m

# 3. Dedicated shuffle partition count
spark.conf.set("spark.sql.shuffle.partitions", 400)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Shuffle Write:** Map tasks partition records into in-memory append-only buffers, sorting by partition hash before flushing to local disk files.
- > **Shuffle Read:** Reduce tasks open parallel Netty HTTP connections to pull their respective partition chunks across the cluster network.
- > **Spill (Memory):** Size of uncompressed deserialized records overflowing the execution pool that could not be held in RAM.
- > **Spill (Disk):** Size of compressed serialized records written to disk; high spill signals severe memory starvation and disk I/O thrashing.

Diagnostic Alert: When Spark UI shows Spill (Memory) > 0, increase partition counts or scale up executor RAM to eliminate disk thrashing.

Production Executor Sizing Formula

Cluster Math

```
# Node Hardware: 10 Nodes, 16 Cores, 64 GB RAM per Node
1. Cores per Node for OS/Daemons = 1 Core (15 Cores left for Spark)
2. Optimal Cores per Executor = 5 Cores (Avoids GC thrashing & HDFS bottleneck)
3. Executors per Node = 15 / 5 = 3 Executors
4. Total Executors = (10 Nodes * 3) - 1 (Driver) = 29 Executors
5. RAM per Node for OS = 1 GB (63 GB left for Spark)
6. RAM per Executor = 63 GB / 3 = 21 GB
7. Memory Overhead = max(384MB, 0.10 * 21GB) = ~2.1 GB
8. Executor Heap = 21GB - 2.1GB = 18.9 GB (Set: --executor-memory 19g)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **5 Cores Golden Ratio:** Allocating >5 cores per executor leads to severe JVM Garbage Collection pause degradation and storage I/O contention.
- > **OS Core Headroom:** Leaving 1 core per VM prevents Kubernetes kubelet and Linux background daemons from starving executor threads.
- > **Overhead Calculation:** 10% overhead ensures sufficient memory for off-heap buffers, avoiding YARN/K8s container exit code 137 kills.
- > **Driver Reservation:** Deducting 1 executor accounts for the ApplicationMaster/Driver node in cluster deployment mode.

Parallelism Rule: Configure `spark.default.parallelism = total_executor_cores * 2 to 3` (For 145 cores = ~300-450 partitions).

Databricks Photon Native Vectorized Engine

Photon C++

```
# Enable Photon Vectorized Acceleration in Databricks
spark.conf.set("spark.databricks.photon.enabled", "true")

# Vectorized Batch Size & Feature Toggles
spark.conf.set("spark.databricks.photon.columnarExecution", "true")
spark.conf.set("spark.databricks.photon.scan.parquet", "true")
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Hand-Optimized C++:** Replaces Java bytecode execution paths entirely with native C++ execution compiled for modern CPU architectures.
- > **SIMD Vectorization:** Processes columnar arrays in parallel CPU SIMD registers (Single Instruction, Multiple Data) on every clock cycle.
- > **Zero JVM GC Overhead:** Eliminates Java heap allocations for intermediate records, preventing JVM GC pause spikes during heavy aggregations.
- > **Accelerated Operations:** Accelerates Parquet scans, hash joins, sorting, higher-order functions, and string parsing by up to 5x.

FinOps ROI: Photon instances cost slightly more DBUs but deliver 3x-5x throughput, lowering the net total compute cost per pipeline run.

Spark UI Diagnostic & Triage Playbook

Diagnostics

```
# Spark UI Metric Diagnosis Framework

1. Stages Tab & Timeline: Look for straggler tasks (Max >> 75th percentile).
2. Executors Tab: Check GC Time percentage (Alert if GC Time > 10%).
3. Task Metrics: Inspect Shuffle Spill (Memory) and Spill (Disk).
4. SQL / Plan Tab: Verify SortMergeJoin vs BroadcastHashJoin selection.
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Task Duration Skew:** If 199 tasks finish in 2 seconds but 1 task takes 10 minutes, you have severe key data skew in the shuffle stage.
- > **GC Time > 10%:** Signals JVM heap pressure; reduce executor cores or enable off-heap memory to avoid pause cascades.
- > **Shuffle Spill Indicator:** Spill indicates partitions are larger than available execution RAM; increase `spark.sql.shuffle.partitions`.
- > **DAG Inspection:** Use the SQL tab DAG to find missing partition pruning filters or unexpected CartesianProduct operators.

Triage Playbook: Always check the Stages tab first, identify the longest stage, then inspect task duration distribution to find bottlenecks.

Storage Levels & Checkpoint Decision Matrix

Persistence

```
# persist() keeps DAG lineage for node failure recovery
df_cached = df.persist(StorageLevel.MEMORY_AND_DISK_SER)

# checkpoint() writes to ADLS and TRUNCATES lineage graph!
spark.sparkContext.setCheckpointDir("abfss://chk@adls.dfs.core.windows.net/dag")
df_checkpointed = df.checkpoint(eager=True)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **persist():** Retains the full logical DAG lineage; if an executor fails, Spark recomputes only the missing partition from source.
- > **checkpoint():** Writes physical data to durable cloud storage and completely **truncates the lineage graph**.
- > **StackOverflow Prevention:** Mandatory in iterative machine learning loops (ALS, GraphX) and complex streaming joins to avoid DAG overflow.
- > **Storage Level Selection:** Use `MEMORY_AND_DISK_SER` in production to minimize JVM GC pauses while avoiding OOM spills.

Architect Rule: Use `persist()` for reusing intermediate DataFrames `checkpoint()` to break huge recursive lineages.

Adaptive Query Execution (AQE) 3 Superpowers

Adaptive Tuning

```
# Enable Adaptive Query Execution Core & Features
spark.conf.set("spark.sql.adaptive.enabled", "true")

# 1. Dynamic Partition Coalescing (Merges small post-shuffle partitions)
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")

# 2. Dynamic Join Switching (SortMergeJoin → BroadcastHashJoin)
spark.conf.set("spark.sql.adaptive.autoBroadcastJoinThreshold", 67188864) # 64MB

# 3. Dynamic Skew Join (Splits oversized partitions automatically)
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Runtime Re-Optimization:** AQE inspects concrete stage output statistics at runtime shuffle boundaries to dynamically re-plan downstream stages.
- > **Partition Coalescing:** Combines thousands of small post-shuffle partitions into target ~128MB partitions, preventing task scheduling overload.
- > **Dynamic Join Switching:** If intermediate filtered tables turn out to be <64MB, AQE converts SortMergeJoin to BroadcastHashJoin on the fly.
- > **Automatic Skew Join:** Detects skewed partitions (>5x median size) and splits them into parallel sub-partitions to eliminate stragglers.

Production Note: AQE is enabled by default in Databricks Runtime 13+; always verify that skew join is active for heavy ETL pipelines.

Data Skew Root Causes & Solutions

Skew Remediation

```
# Strategy: Separate Nulls & Extreme Hot-Keys
df_non_null = df_fact.filter(F.col("cust_id").isNotNull())
df_nulls = df_fact.filter(F.col("cust_id").isNull())

# Join clean data with dimension, then re-attach nulls without shuffle
df_joined = (df_non_null
    .join(df_dim, on="cust_id", how="left")
    .unionByName(df_nulls.selectExpr("* ", "null as dim_col")))
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **Root Cause:** High-frequency keys (e.g. `NULL` customer IDs, default dummy values like -1) hash into a single partition executor.
- > **Null Isolation:** Null values never match in inner/left joins; isolating nulls bypasses the shuffle hash collision entirely.
- > **Broadcast Join:** If one side of the join is <8GB on large clusters, broadcasting it completely bypasses the shuffle exchange.
- > **Synthetic Salting:** Appending random integer suffixes (0-9) spreads skewed keys across 10 independent executors in parallel.

Detection: Run `df.groupBy("key").count().orderBy(F.desc("count")).show(10)` to identify high-cardinality skew keys.

JVM Garbage Collection (G1GC) & Shuffle Math

Tuning Knobs

```
# Production G1GC Flags for Databricks / Spark
-XX:+UseG1GC
-XX:InitiatingHeapOccupancyPercent=35
-XX:G1ReservePercent=15
-XX:MaxGCPauseMillis=200

# Optimal Shuffle Partitions Calculation Formula:
# Rule: Target 100MB ~ 200MB uncompressed per shuffle partition
Total Shuffle Input Data = 64 GB (= 65,536 MB)
Target Partition Size = 128 MB
Optimal Partitions = 65,536 MB / 128 MB = 512 Partitions
Config: spark.conf.set("spark.sql.shuffle.partitions", 512)
```

CODE BREAKDOWN & ENGINE MECHANICS:

- > **G1GC Collector:** Partitions heap into 2048 regions; prioritizes garbage collection in regions containing mostly dead objects.
- > **InitiatingHeapOccupancyPercent (35%):** Starts concurrent marking earlier, preventing stop-the-world Full GC pause cascades.
- > **MaxGCPauseMillis (200ms):** Sets soft target pause duration for the JVM collector to balance throughput and latency.
- > **Shuffle Partition Math:** Oversized partitions (>500MB) cause OOM spills; undersized partitions (<10MB) introduce scheduling overhead.

AQE Advisory Size: Set `spark.sql.adaptive.advisoryPartitionSizeInBytes` (128MB) for automatic post-shuffle partition sizing.