Spark Optimization

Cluster and Code Level

By Dhandapani Yedappalli Krishnamurthi Sep 4, 2025

Spark Optimization Guide

1. Cluster-Level Optimizations

These ensure Spark has the right resources and settings.

a) Cluster Sizing

- Right number of **executors**, **cores**, and **memory**.
- Formula:

Executor Memory=Total Node Memory−OSReserveExecutorsperNodeExecutor **Memory**=ExecutorsperNodeTotal Node Memory−OSReserveCores per **Executor**≈5Cores per Executor≈5

💡 Too many cores per executor → excessive GC (Garbage Collection).
💡 Too few cores → under-utilization.

---

b) Dynamic Allocation

- Enables Spark to scale executors up/down depending on workload.

```
--conf spark.dynamicAllocation.enabled=true
--conf spark.shuffle.service.enabled=true
--conf spark.dynamicAllocation.minExecutors=2
--conf spark.dynamicAllocation.maxExecutors=50
```

---

c) Shuffle Optimizations

Shuffles are costly (network + disk I/O).

- Use **spark.sql.shuffle.partitions** wisely.
  - Default = 200 (too high for small jobs).
  - Tune according to dataset size.

```
spark.conf.set("spark.sql.shuffle.partitions", 50)
```

---

d) Caching and Storage

- Cache frequently reused DataFrames.
- Decide storage level:

```
df.persist(StorageLevel.MEMORY_AND_DISK)
```

---

e) File Format & Compression

- Prefer **Parquet/ORC** over CSV/JSON → columnar, compressed, splittable.
- Enable **snappy/zstd compression**.

---

f) Cluster Hardware & I/O

- Co-locate Spark with HDFS for locality.
- Prefer **SSD** for shuffle-heavy workloads.
- Enable **compression** during shuffle (saves network).

---

2. Code-Level Optimizations

These are in how you write Spark transformations/actions.

a) Avoid Wide Transformations

- Wide transformations (e.g., `groupByKey`) cause shuffle.
- Prefer **reduceByKey / aggregateByKey**.

❌ Bad:

```
rdd.groupByKey().mapValues(sum)
```

✅ Better:

```
rdd.reduceByKey(lambda x, y: x + y)
```

---

b) Partitioning

- Repartition/join on the same key → reduces shuffle.

```
df1 = df1.repartition("id")
df2 = df2.repartition("id")

df_join = df1.join(df2, "id")
```

---

c) Broadcast Joins

- For small dimension tables → avoid shuffling large fact tables.

```
from pyspark.sql.functions import broadcast

fact.join(broadcast(dim), "id")
```

---

d) Column Pruning

- Select only needed columns before joins/aggregations.

```
df = df.select("id", "amount")
```

---

e) Predicate Pushdown

- Let Spark filter early. Works best with Parquet/ORC.

```
df = spark.read.parquet("s3://data/").filter("year = 2024")
```

---

f) Avoid Collect / Count / Show on Large Data

- They pull data to driver → risk of OOM.
- Use `limit()` for sampling.

---

g) UDF Alternatives

- Prefer **built-in Spark SQL functions** over Python UDFs (slow, no optimization).
- If needed, use **Pandas UDFs** (vectorized).

---

h) Cache Smartly

- Cache only when reused multiple times, and unpersist after use.

```
df.cache()
# use df multiple times
df.unpersist()
```

---

i) Skew Handling

- If one key has too much data → skew.
    - Techniques: **Salting keys**, **skew join hints**.

```
df1.join(df2.hint("skew"), "id")
```

---

3. Diagram: Spark Optimization Flow

```
               +-----------------------+
                |   Cluster Tuning        |
                |   (Executors, Memory) |
                +-----------------------+
                           ↓
                +-----------------------+
                |   Shuffle Optimizing  |
                |   (Partitions, Joins) |
                +-----------------------+
                           ↓
                +-----------------------+
                |   Code-level Best     |
                |   Practices           |
                +-----------------------+
                           ↓
                +-----------------------+
                |   Final Output        |
                |   Faster Jobs         |
                +-----------------------+
```

---

4. Example: Optimized Code

Suppose we need to join **sales (fact)** with **customers (dim)**:

```
# Without optimization
sales = spark.read.parquet("s3://data/sales/")
```

```
customers = spark.read.csv("s3://data/customers.csv",
header=True)

result = sales.join(customers,
"cust_id").groupBy("region").agg({"amount": "sum"})
```

Optimized:

```
# Optimize partitions & format
sales =
spark.read.parquet("s3://data/sales/").repartition("cust_id"
)
customers = spark.read.csv("s3://data/customers.csv",
header=True)

# Column pruning
customers = customers.select("cust_id", "region")

# Broadcast small dimension
result = sales.join(broadcast(customers), "cust_id") \
              .groupBy("region") \
              .sum("amount") \
              .persist()

result.write.mode("overwrite").parquet("s3://data/output/")
```

- PySpark DataFrame:

```
df.explain()               # basic physical plan

df.explain(True)           # extended (parsed/analyzed/optimized +
physical)
```

```
df.explain("formatted")    # (Spark 3.x) nicely formatted plan
```

- Scala / Spark shell:

```
df.explain()                // simple

df.explain(true)            // extended

println(df.queryExecution.executedPlan)   // the physical SparkPlan
object as string
```

- Spark SQL:

```
spark.sql("SELECT ...").explain(true)
```

---

1) Create / prepare a query or DataFrame

Example (PySpark):

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

sales = spark.read.parquet("/path/sales")
dim = spark.read.csv("/path/customers.csv", header=True)

q = sales.filter("amount >
100").join(dim.select("id","region"),
"id").groupBy("region").sum("amount")
```

2) Print the basic physical plan

```
q.explain()
```

What you get: a compact physical plan tree (operators & their order). Good for a fast look.

3) Print the extended plan (parsed → analyzed → optimized → physical)

```
q.explain(True)
# or
q.explain("extended")
```

What you get: full stack of plans:

- Parsed logical plan
- Analyzed logical plan (with resolved attributes)
- Optimized logical plan (after Catalyst optimizations)
- Physical plan (how Spark will execute)

4) Use formatted and codegen modes (Spark 3.x)

```
q.explain("formatted")   # friendlier tree, indentation and
column stats
q.explain("codegen")     # shows whole-stage codegen
Java-like code (where applicable)
```

If a mode fails (version differences), fall back to `explain(True)`.

5) (Scala) Inspect the `queryExecution` internals for programmatic access

```
println(df.queryExecution.logical)         // logical plan
object
println(df.queryExecution.analyzed)        // analyzed plan
println(df.queryExecution.optimizedPlan)   // optimized
logical plan
```

```
println(df.queryExecution.executedPlan)    // physical
SparkPlan
```

Use this when you need the plan inside an application / unit tests.

6) (PySpark) access internal Java plan (advanced / internal API)

```
# internal — may depend on version; use for deeper
introspection only
print(q._jdf.queryExecution().executedPlan().toString())
```

Caution: internal API and Py4J usage can vary across Spark versions.

7) See the actual runtime execution & metrics in the Spark UI

1. Run your job/query (the app must be running).
2. Open the driver UI: `http://<driver-host>:4040` (local) or Yarn / ApplicationMaster / Spark History Server for cluster modes.
3. Click the **SQL** tab (if present) → find the query → view its details.
4. The UI shows the **Physical/Executed Plan** and the DAGScheduler stages and task metrics (shuffle read/write, time, etc.).
   This is where you see *actual* runtime statistics (not visible from `explain()` alone).

8) RDD lineage (if using RDDs)

```
rdd = df.rdd
print(rdd.toDebugString())   # shows lineage and
partitioning
```

9) If you want cost-based info: enable CBO & gather stats

```
spark.conf.set("spark.sql.cbo.enabled", True)
spark.sql("ANALYZE TABLE my_table COMPUTE STATISTICS FOR ALL
COLUMNS")
# then run explain(...) to see plans influenced by stats/CBO
```

10) Helpful settings & tips

- If plan output is truncated, bump the debug fields:

```
spark.conf.set("spark.debug.maxToStringFields", 100)
```

- `explain()` only shows *what* will be executed (plan). To see task-level metrics you must actually run the job and consult the Spark UI or HistoryServer.
- For small dimension joins, confirm Spark used a **BroadcastHashJoin** in physical plan (look for `BroadcastHashJoin` node).
- For heavy shuffles, check `Exchange` / `SortMergeJoin` nodes in the physical plan.

Example of a sample physical plan (illustrative)

```
== Physical Plan ==
*(2) HashAggregate(keys=[region#12],
functions=[sum(amount#5)])
+- Exchange hashpartitioning(region#12, 200)
   +- *(1) Project [region#12, amount#5]
      +- *(1) Filter (isnotnull(amount#5) AND (amount#5 >
100))
         +- *(1) FileScan parquet [id,region,amount] ...
```

Interpretation: you can read it top→bottom: aggregate → shuffle → project → filter → scan.

Quick checklist to debug why a physical plan looks bad

- Are large joins not broadcasted? → check join type and sizes.
- Is there an unexpected `Exchange` (shuffle)? → consider `repartition()` or join hints.
- Is whole-stage codegen present? → look for `WholeStageCodegen` nodes or use `explain("codegen")`.
- Are statistics available for CBO? → run `ANALYZE TABLE` and enable `spark.sql.cbo.enabled`.
