Spark Optimization Topic

Karthik Kondpak
9989454737

Day 11 — Spark Optimization Topic

🔥 Storage Level — When to Use Cache vs Persist vs Checkpoint

Spark gives three major ways to store intermediate results:

- **cache()**
- **persist()**
- **checkpoint()**

All three improve performance — but each serves a different purpose.

Using the wrong one can slow down the job or overload your executors.

Today you'll learn:

- When to use each
- When to avoid
- Internal working
- Real-time examples

- Spark UI verification

Core Difference (Simple Understanding)

Technique	Purpose	Stored Where?	Lifetime
cache	Reuse DataFrame	Memory	Until eviction
persist	Control storage level	Memory / Disk	Until eviction
checkpoint	Break lineage + fault tolerance	Disk / HDFS	Permanent

🔵 cache() — For Fast Reuse

Cache stores the DataFrame **in memory only** using the default storage level:

`MEMORY_ ONLY`

✔ When to Use cache()

Use cache when:

- DataFrame is reused **multiple times** in the same job

- You want **fastest reads** (in-memory)
- Data fits comfortably in memory

Example:

```
df = spark.read.parquet("/sales")


df.cache()
df.count()      # materialize
df.filter("amount > 1000").show()
df.groupBy("state").count().show()
```

The second and third actions will be much faster.

❌ Avoid cache() when:

- Data is **very large** → may cause memory eviction
- Cluster has limited RAM
- DataFrame used **only once**

persist() — When You Need Control

Persist lets you choose **how** Spark stores the data.

Most important levels:

```
MEMORY_ONLY
MEMORY_AND_DISK
DISK_ONLY
MEMORY_ONLY_SER (serialized)
MEMORY_AND_DISK_SER
```

✔ When to Use persist()

- DataFrame is **too big for memory**
- You want Spark to **spill to disk**
- Multiple expensive transformations depend on the same DataFrame
- Cross-stage reuse

Example:

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

This avoids recomputation even if RAM is full.

❌ **Avoid persist() when:**

- DataFrame used only once
- You don't want disk I/O overhead

🔵 checkpoint() — For Long Lineage & Fault Tolerance

Checkpoint writes the DataFrame to **HDFS / DBFS / Cloud storage**.

It breaks the DAG lineage, creating a new, clean DataFrame.

✔ When to Use checkpoint()

When lineage becomes too long

Examples:

- Repeated joins
- Multiple transformations

- Deep recursive pipelines (graph processing)

Long lineage makes Spark:

- Recursively recompute many steps
- Use heavy driver memory
- Risk stack overflow

Checkpoint solves this.

When building streaming pipelines

Checkpoint is mandatory for:

- Stateful transformations
- Deduplication
- Aggregations with watermarks

When you want high fault tolerance

Unlike cache/persist (lost on executor failure),

✔ Checkpointed data is safe

✔ Stored in stable storage (HDFS / cloud)

✔Survives executor crashes

❌ Avoid checkpoint() when:

- You only need data for short-term reuse
- You don't need DAG breaking
- You want fastest performance (checkpoint is slower)

🔍 How to Check in Spark UI

Go to **Storage Tab**:

- Cached → shows memory usage
- Persisted → shows storage level
- Checkpoint → not shown in Storage Tab (written to filesystem)

Example — Flipkart Orders Pipeline

Situation:

- Order data = 300M rows

- Applied many transformations
- Then joined with multiple dimensions
- Lineage becomes huge

Best approach:

```
spark.sparkContext.setCheckpointDir("/tmp/checkpoint")

orders =
orders_raw.transform(clean_data).transform(apply_logic
)
orders = orders.checkpoint()      # break lineage

final_df = orders.join(product_dim, "product_id")
```

This stabilizes the job and reduces driver memory pressure.

Seekho Bigdata
Data is the New Oil

Let's build your Data Engineering journey together!

📩 Call us directly at: 9989454737

https://seekhobigdata.com/