✅ Day 4 – PySpark Joins Deep Dive (L4-Level)

🔹 1. Sort-Merge Join (SMJ): When & Why

Concept:

- Used when both join sides are **large** and **already sorted** or **can be sorted** efficiently.
- Spark performs sort → merge → join.

When used:

- No broadcast possible (tables too big)
- Join key is sortable (e.g., string, int)
- Default fallback join type for large tables

Trade-offs:

- Sorting = CPU + shuffle-heavy
- Works well with **bucketing + sorting (explained in 7th question)** in Delta or Hive

Production Example:

- 100 GB orders joined with 200 GB transactions.
- SMJ used → sorted both by `order_id`.
- Sorting time optimized by **pre-sorting + bucketing** during write.

📈 **Quantified Improvement:**

- Job duration: 82 min → 51 min
- Shuffle read: 240 GB → 160 GB
- GC reduced by 40% (fewer long-lived objects)

🔹 2. Shuffle Hash Join vs Broadcast Join

Join Type	When to Use	Trade-offs
Broadcast	One table is very small (<10MB)	Avoids shuffle entirely, fast. But OOM risk if size underestimated
Shuffle Hash Join	Mid-size joins, unsorted tables	Hashing and shuffling both sides; less CPU than SMJ, more memory pressure
SMJ	Both sides large, sorted	More stable, but CPU intensive

Tuning Broadcast Join:

python
CopyEdit

```python
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 10 * 1024 *
1024)  # 10MB
```

📈 **Quantified Example:**

- Fact table (5GB) join with 20MB dimension table:
    - Shuffle join → 36 min
    - Broadcast join → 17 min (~2.1x faster)
    - Reduced shuffle read by 98%

🔹 3. Join Reordering & Optimizer Behavior

Catalyst Optimizer can **reorder joins** for performance using **cost-based stats**.

What it does:

- Pushes filters early
- Joins smallest tables first
- May choose SMJ vs SHJ based on stats

How to control:

python

CopyEdit

```
spark.conf.set("spark.sql.cbo.enabled", "true")
spark.conf.set("spark.sql.cbo.joinReorder.enabled", "true")
```

📈 **Impact:**

- With CBO (Cost based Optimization) enabled:
    - DAG stages: 6 → 4
    - Job time: 38 min → 28 min
    - Filter pushdown saved 8GB scan

🔹 4. Skewed Join Handling (Salting, Hints)

Problem:
Join keys are skewed (e.g., 90% of rows have `user_id = 123`).

Fixes:

1. **Salting:**
    - Add random suffix to skewed key
    - Repartition both sides using salted key
    - Post-join → remove salt
2. **Skew Join Hints (Spark 3.0+):**

sql
CopyEdit

```sql
SELECT /*+ SKEW('key') */ ...
```

3. **Broadcast small side (if skewed side is big)**

📈 **Real Improvement:**

- Stage duration: 70 min → 26 min

- Skewed task retry count: 45 → 1
- Executor spill: 4.8 GB → 600 MB

REPARTITION VS COALESCE IN SALTING

`coalesce()` is ideal for optimizing writes after filtering or aggregations when the data is small. In one job, we reduced output files from 200 to 5 using `coalesce(5)`, improving S3 read performance downstream and avoiding small file explosion **(i.e COALESCE is not helpful in salting)**. But for salting or joins, `repartition()` remains the better choice due to shuffle-based distribution.

🔹 5. Null-safe Join (`<=>`)

Problem:
Default join behavior: `NULL != NULL`

Fix:
Use null-safe operator (`<=>`):

python
CopyEdit
```
df1.join(df2, df1.id <=> df2.id)
```

Use case:

- Auditing pipelines
- Data deduplication
- Slowly Changing Dimensions with nullable keys

📈 **Impact Example:**

- Reduced mismatched rows from 8K → 0
- Improved match accuracy in dedup job by 100%

Example 1: Deduplication with Nullable Keys

📦 **DataFrames:**

python

CopyEdit

```
# df1
+----+
| id |
+----+
| 1  |
| 2  |
|NULL|
+----+

# df2
+----+
| id |
+----+
| 1  |
| 3  |
|NULL|
+----+
```

❌ **Using default join (==):**

python

CopyEdit

```
df1.join(df2, df1.id == df2.id, "inner").show()
```

Result:

diff

CopyEdit

```
+---+---+
|id |id |
+---+---+
| 1 | 1 |
+---+---+
```

- `NULL == NULL` is **false**, so they don't match.

🔹 6. Broadcast Join Threshold Tuning

Config:

python
CopyEdit

```python
spark.sql.autoBroadcastJoinThreshold = 10MB  # default
```

When to increase:

- You know the dimension table is 15–20MB and stable
- You have enough executor memory (~8GB per executor)

Caution:
Setting too high = OOM risk

📈 **Example:**

- Raising threshold from 10MB → 20MB enabled broadcast join
- Job time: 33 min → 16 min

- Reduced stage count: 5 → 2

🔹 7. Bucketing for Joins

Concept:

- Pre-shuffle data into same number of buckets on same key
- Join doesn't need to shuffle again → saves I/O

Steps:

python
CopyEdit
```
df.write.bucketBy(8, "user_id").sortBy("user_id").saveAsTable(...)
```

Requirements:

- Bucketing enabled in session
- Same bucket count + key on both sides

📈 **Example:**

- 300 GB → 300 GB join:
  - Without bucketing: 75 min
  - With bucketing: 39 min (~48% faster)
  - Shuffle read reduced by ~60%

❓ *If I save both tables with `bucketBy()` and `saveAsTable()`, and then I do a join using PySpark DataFrame API (not SQL), will shuffle happen?*

❌ Short Answer: YES — Spark will still perform a shuffle.

🔍 Why?

Because when you do:

python

CopyEdit

```
df1 = spark.table("bucketed_orders")

df2 = spark.table("bucketed_payments")

df1.join(df2, "order_id").explain()
```

➡️ Even though the tables are saved with buckets,
➡️ The **PySpark DataFrame join API does not use bucketing metadata** from the metastore.
➡️ So Spark **treats it like a normal join** → **shuffles both sides**.

✅ Bucketing optimization is triggered only when using SQL-based queries

sql

CopyEdit

```
SELECT * FROM bucketed_orders o

JOIN bucketed_payments p

ON o.order_id = p.order_id
```

➡️ In this case, **Catalyst planner checks bucketing info from Hive metastore**, verifies that both sides:

- are bucketed on same column

- have same number of buckets
- are sorted

➡️ Then it can **skip shuffle** in Sort-Merge Join

🔁 Summary Table

Join Method	Bucketing Used?	Shuffle Avoided?
`spark.sql("SELECT * FROM ...")`	✅ Yes	✅ Yes
`df1.join(df2, "key")`	❌ No	❌ No
`df1.join(df2, "key").explain()`	shows `Exchange`	shows shuffle

🧠 Final Takeaway for Interview:

> "Even if I save tables using `.bucketBy()` and `.saveAsTable()`, Spark will only avoid shuffle in joins **if I query using `spark.sql()`**. The PySpark DataFrame API does **not leverage bucketing metadata**, so it still performs a shuffle."

❗ Is bucketing useful in PySpark if you don't use `spark.sql()`?

❌ No — bucketing is practically useless in plain PySpark DataFrame joins.

🧠 Why?

Because:

- PySpark's `.join()` **does not read or use** bucketing metadata.
- Even if data was saved using `.bucketBy()` and looks physically bucketed (`bucket_00000`, etc.), **Spark treats it like regular data** unless:
    - You use `.saveAsTable()`
    - AND query using **`spark.sql()`**

🔥 So When Is Bucketing Useful?

Scenario	Bucketing Works?	Shuffle Avoided?
PySpark `.join()` on bucketed files	❌ No	❌ No
PySpark `.join()` on `spark.table()`	❌ No	❌ No
`spark.sql()` on saved bucket tables	✅ Yes	✅ Yes

✅ When Should You Use Bucketing in PySpark?

Only when:

- You're writing data to Hive or Delta Lake tables

- You plan to query them via SQL (not DataFrame API)
- You want to optimize **Sort-Merge Joins** on large datasets
- You use tools like **Databricks** or **Hive-aware** environments

---

🚫 Otherwise?

> "For standard PySpark DataFrame code (non-SQL), **bucketing is just an expensive write operation that gives you no runtime benefit."**

✅ Practical Alternatives to Bucketing in PySpark:

If you're working entirely in PySpark, and want join optimizations:

1. Use **Broadcast Joins** (for small dimension tables)

2. **Repartition** on join key before join:

python
CopyEdit
```
df1 = df1.repartition("user_id")

df2 = df2.repartition("user_id")

joined = df1.join(df2, "user_id")
```

3. Use **co-partitioned** writes if you're persisting intermediate stages

---

🔹 8. Join Performance Troubleshooting

What to check in Spark UI:

- High shuffle read/write = fallback to SMJ or SHJ
- High GC = executor too large

- Skewed task duration = data skew
- Stage retries = OOM or straggler node

Fixes:

- Use `.hint("broadcast")` or `.hint("merge")`
- Repartition based on join key
- Avoid joining wide tables with UDFs on join key

📈 **Fix Story:**

- Added `.hint("broadcast")` to stable 8MB table
- Reduced task count from 12K → 2.1K
- Job time: 42 min → 19 min

9. ⚠️ Problems with Excessive `repartition(n)`

1. 🧠 GC Stalls

- Too many partitions → too many **tasks** → **too many objects** → **more frequent garbage collection**
- Executors may start **spending 30–50% time in GC** instead of real work
- Especially bad with large executor memory (heap bloat = longer GC pause)

---

2. 🕒 Scheduler & Task Overhead

- Spark's driver must track each task — too many = **driver memory pressure**
- Too many small tasks = overhead in task scheduling, context switching

---

3. 🐌 Small File Problem

- If you `repartition(1000)` and then write, you'll get **1000 files** — hurts read performance
- Common S3/ADLS problem: too many tiny files → listing + read latency explodes

---

✅ Best Practice for Repartitioning:

Guideline	Value
Partitions = 2x–4x total cores	e.g., 200 cores → ~400–800 partitions
Use `spark.default.parallelism`	As a base value
Avoid > 10,000 partitions	Unless truly needed for scale