Spark SQL Optimization

1: Optimize JOIN Queries in Spark SQL

🔷 Topic: Join Optimization

📌 Explanation:

Join operations are expensive in Spark due to shuffling data across nodes. You can optimize joins by:

- Using **broadcast joins** for small tables.
- `Controlling` **`join types`** `(broadcast, sort-merge, shuffle hash).`
- Ensuring **partitioning** and **data skew** are handled.

📊 Use Case:

You have a users table with 10M records and a country_codes table with only 200 records. You want to join them to get the country name for each user.

❌ Unoptimized Spark SQL:

```
users_df.createOrReplaceTempView("users")
countries_df.createOrReplaceTempView("countries")
```

```
# Regular join without optimization
spark.sql("""
SELECT u.user_id, u.name, c.country_name
FROM users u
JOIN countries c
ON u.country_code = c.country_code
""")
```

⚠️ Problem:

This causes a **shuffle join** (both datasets are shuffled), which is inefficient because countries is small and doesn't need to be shuffled.

✅ Optimized Spark SQL: Use Broadcast Join

```
# Broadcast the smaller table
from pyspark.sql.functions import broadcast

optimized_df = users_df.join(

    broadcast(countries_df),
    users_df.country_code == countries_df.country_code,
    "inner"
)

optimized_df.select("user_id", "name", "country_name").show()
```

✅ Alternate (SQL syntax):

```
spark.sql("""
SELECT /*+ BROADCAST(c) */
  u.user_id, u.name, c.country_name
FROM users u
JOIN countries c
ON u.country_code = c.country_code
```

```
""")
```

🔍 Why This Works:

- **Broadcast join** sends the small table (countries) to all executors.
- Avoids shuffle — faster performance, especially with large users dataset.
- Works best when one table is small (<10 MB by default) — configurable.

📈 Performance Improvement:

Metric	Unoptimized	Optimized (Broadcast)
Shuffle Read Size	High	Low
Join Time	~12s	~3s
Stage Failures	Possible	Fewer

Perfect! Let's continue with the next Spark SQL optimization case studies — following the same structure of **topic explanation**, **use case**, and **optimized vs. unoptimized code**.

✅ Case Study 2: Predicate Pushdown in Spark SQL

🔷 Topic: Predicate Pushdown

📌 Explanation:

Predicate Pushdown means applying filters at the **data source level**, so that Spark reads only the relevant data. This:

- Reduces I/O and network usage.
- Speeds up query execution.
- Works best with formats like **Parquet**, **ORC**, and JDBC sources.

📊 Use Case:

You're querying a **Parquet** dataset of 1 TB of sales data, but only need records from **January 2024**.

❌ Unoptimized Spark SQL:

```
# Reading full file then filtering
df = spark.read.parquet("s3://data-lake/sales/")
filtered = df.filter("sale_date >= '2024-01-01' AND sale_date < '2024-
02-01'")
filtered.select("sale_id", "amount").show()
```

⚠️ Problem:

If schema inference is triggered or data is cached early, Spark may read the **entire 1 TB**, then filter in memory — wasting I/O and time.

✅ Optimized Spark SQL (Pushdown Enabled):

```
# Apply filter during read itself (pushdown)
df = spark.read \
    .option("basePath", "s3://data-lake/sales/") \
    .parquet("s3://data-lake/sales/year=2024/month=01/")

df.select("sale_id", "amount").show()
```

Or, using a path and partition filter:

```
df = spark.read.parquet("s3://data-lake/sales/")
filtered = df.filter("year = 2024 AND month = 1")
filtered.select("sale_id", "amount").show()
```

🔍 Why This Works:

- Pushdown filters **before** loading data into Spark.
- Works best when the data is **partitioned** by date fields.
- Spark reads only year=2024/month=01/, skipping the rest.

📈 Performance Improvement:

Metric	Unoptimized	Optimized (Pushdown)
Data Read	1 TB	~80 GB (Jan only)
Read Time	~90s	~12s
CPU Usage	High	Low

✅ Case Study 3: Caching and Persistence

🔷 Topic: Caching and Persistence

📌 Explanation:

If a DataFrame is reused multiple times in a pipeline or across queries, caching avoids recomputation.

Use .cache() or .persist(StorageLevel) to store it in memory or disk.

📊 Use Case:

You run 5 analytics queries on a heavy transformation of a 100M row DataFrame.

❌ Unoptimized:

```
# Expensive transformation computed 5 times
transformed = df.withColumn("net_price", df.price * (1 - df.discount))
transformed.filter("category = 'electronics'").count()
transformed.groupBy("category").agg({"net_price": "avg"}).show()
# ... and 3 more actions
```

✅ Optimized:

```
from pyspark import StorageLevel

# Cache after first transformation
transformed = df.withColumn("net_price", df.price * (1 -
```

```
df.discount)).cache()

# Run queries
transformed.filter("category = 'electronics'").count()
transformed.groupBy("category").agg({"net_price": "avg"}).show()
# ... other queries
```

Or use `.persist(StorageLevel.MEMORY_AND_DISK)`if memory is tight.

🔍 Why This Works:

- Without caching: Spark recomputes lineage for each action.
- With caching: Transformation is computed once, reused efficiently.

📈 Performance Improvement:

Metric	Unoptimized	Optimized (Cached)
Total Time (5 queries)	~180s	~60s
CPU Load	High	Lower
Memory Usage	Low	Higher (intentional)

✅ Case Study 4: Skew Join Optimization in Spark SQL

🔷 Topic: Skew Join Handling

📌 Explanation:

Data skew occurs when one or more keys in a join have **disproportionately more rows** than others. This causes:

- One executor to do most of the work.
- Long-running stages and uneven load.
- Possible out-of-memory errors.

📊 Use Case:

You are joining a transactions table (2B records) with a `merchant` table (500K records). But 60% of the transactions belong to a single merchant.

❌ Unoptimized Spark SQL:

```
# Skewed join on merchant_id
transactions_df.join(merchants_df, "merchant_id").select("txn_id",
"merchant_name").show()
```

⚠️ Problem:

- Most merchant_ids are balanced.
- One merchant_id (say, M12345) appears 1.2B times.
- This causes a **hot partition**, poor performance, and executor OOM.

✅ Optimized Spark SQL:

Option 1: Salting the skewed key (manual skew fix)

```
from pyspark.sql.functions import col, concat_ws, lit, rand

# Add salt to transactions
salted_txns = transactions_df.withColumn("salt", (rand() *
10).cast("int"))
salted_txns = salted_txns.withColumn("skewed_key", concat_ws("_",
col("merchant_id"), col("salt")))

# Duplicate skewed keys in merchants 10 times (salt replication)
replicated_merchants = merchants_df \
    .filter(col("merchant_id") == "M12345") \
    .withColumn("salt", explode(array([lit(i) for i in range(10)]))) \
    .withColumn("skewed_key", concat_ws("_", col("merchant_id"),
col("salt")))
# Normal keys remain same
normal_merchants = merchants_df.filter(col("merchant_id") != "M12345")
# Union replicated and normal
final_merchants =
rep lic at ed_ mer ch ant s. uni on ByN am e(n orm a l_m erc ha nts .wi th Col um
ey", col("merchant_id")))
# Final join
result = salted_txns.join(final_merchants,
"skewed_key").select("txn_id", "merchant_name")
```

Option 2: Set Spark skew optimization config (automatic for > Spark 3.0):

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true")
```

🔍 Why This Works:

- **Salting** splits the skewed key into multiple smaller keys, balancing load.
- **Adaptive Skew Join** (in Spark 3.0+) automatically detects and splits large partitions at runtime.

📈 Performance Improvement:

Metric	Unoptimized	Optimized (Salting)
Runtime	~200s	~60s
Stage Failure Risk	High	Low
Executor Memory Load	Imbalanced	Balanced

✅ Case Study 5: File Format + Partition Pruning in Spark SQL

🔷 Topic: File Format and Partition Pruning

📌 Explanation:

Two key performance boosters:

- **Columnar Formats**: Use **Parquet/ORC** instead of CSV/JSON.
- **Partition Pruning**: Read only needed partitions based on query filters.

📊 Use Case:

Reading product catalog data partitioned by `category` and `brand`.

❌ Unoptimized Spark SQL:

```
# CSV read, no pruning
df = spark.read.csv("/mnt/products/")
df.filter("category = 'Electronics' AND brand = 'Samsung'").show()
```

⚠️ Problem:

- CSV is row-based — slow read, poor compression.
- No pruning — reads entire folder structure.

✅ Optimized Spark SQL:

```
# Use Parquet and filters
df = spark.read.parquet("/mnt/products/")

# Use partition pruning with filters
result = df.filter("category = 'Electronics' AND brand = 'Samsung'")
result.select("product_id", "price").show()
```

Or, directly specify paths:

```
# Read specific partitions (best for large data)
df =
spa rk. re ad. par qu et( "/ mnt /p rod uc ts/ cat e gor y=E le ctr oni cs /br an
)
```

🔍 Why This Works:

- **Parquet** is a compressed, columnar format — faster read and scan.
- Spark **prunes directory partitions** at read time using filter predicates.

📈 Performance Improvement:

Metric	CSV + No Pruning	Parquet + Pruning
Load Time	~70s	~6s
Disk Read	Full Dataset	Only Needed Parts
Compression Ratio	Low	High

Great! Let's continue with the next **two Spark SQL optimization case studies**:

✅ Case Study 6: Delta Lake Optimization in Spark SQL

🔷 Topic: Delta Lake Optimization (ZORDER, Data Skipping, Vacuum, Compaction)

📌 Explanation:

Delta Lake is a storage layer that brings **ACID transactions** and **schema enforcement** to Spark. But to make queries faster and scalable, you must use:

- **ZORDER**: Optimizes data layout for faster filtering on specific columns.
- **Data Skipping**: Leverages statistics to avoid scanning unnecessary files.
- **Vacuum**: Cleans up stale files.
- **Compaction**: Merges many small files into large ones for performance.

📊 Use Case:

You manage a Delta table `/delta /ev en ts/` with 5 years of IoT event data. Most queries filter on device_id and `event_date.`

❌ Unoptimized Delta Lake Usage:

```
# Query without ZORDER or compaction
df = spark.read.format("delta").load("/delta/events/")
df.filter("device_id = 'D1002' AND event_date = '2023-08-01'").count()
```

⚠️ Problem:

- Query scans many small files (~millions).
- No data clustering → slow scans even if partitions exist.

✅ Optimized Delta Lake Usage:

Step 1: Compaction (CoalescingFiles)

```
# Coalesce into fewer files
(
  spark.read.format("delta").load("/delta/events/")
  .repartition(10)  # Tune as needed
  .write.option("dataChange", "false")
  .format("delta")
  .mode("overwrite")
  .save("/delta/events/")
)
```

Step 2: Z-Ordering on Filter Columns

```
OPTIMIZE delta.`/delta/events/` ZORDER BY (device_id, event_date)
```

Note: OPTIMIZE & `ZORDER` are **Databricks-only** features (or Delta Lake OSS 2.0+ with Photon).

Step 3: Vacuum Old Files

```
VACUUM delta.`/delta/events/` RETAIN 168 HOURS
```

🔍 Why This Works:

- **ZORDER** clusters column values across files to reduce file scans.
- **Data skipping** uses min/max stats to skip irrelevant files.
- **Vacuum** deletes obsolete files — keeps storage clean.
- **Compaction** improves read performance and parallelism.

📈 Performance Improvement:

Metric	Unoptimized	Optimized (ZORDER + Compact)
Query Time	~120s	~8s
Files Scanned	~800K	~100
Disk IO	High	Minimal

✅ Case Study 7: Aggregation Optimization in Spark SQL

🔷 Topic: Aggregation Tuning

📌 Explanation:

Aggregations can be costly, especially on large datasets. You can optimize them via:

- **Partial aggregation** (map-side combine)
- **Approximate aggregations** (approx_count_distinct)
- **Efficient group keys** (avoid high-cardinality)

📊 Use Case:

You analyze user behavior on an e-commerce site with 5 billion click records and need to:

- Count distinct users per region

- Calculate total time spent per session

❌ Unoptimized:

```
# Heavy exact aggregation
df. gro up By( "re gi on"
). agg(
    countDistinct("user_id").alias("unique_users"),
    sum("session_time").alias("total_time")
).show()
```

✅ Optimized:

Option 1: Use Approximate Aggregation for Large Data

```
from pyspark.sql.functions import approx_count_distinct

df.groupBy("region").agg(
    approx_count_distinct("user_id").alias("unique_users"),
    sum("session_time").alias("total_time")
).show()
```

Option 2: Enable Map-Side Combine

```
spark.conf.set("spark.sql.codegen.aggregate.mapSideCombine", "true")
```

Option 3: Use Bucketed Tables if grouping by same column repeatedly

```
CREATE TABLE user_logs_bucketed
USING PARQUET
CLUSTERED BY (region) INTO 32 BUCKETS
```

```
AS SELECT * FROM user_logs
```

🔍 Why This Works:

- Approximate methods use probabilistic data structures (like HyperLogLog++).
- Map-side combine reduces shuffle size by aggregating locally first.
- Bucketing helps Spark avoid full shuffle during grouping.

📈 Performance Improvement:

Metric	Unoptimized	Optimized
Aggregation Time	~90s	~12s
Memory Usage	High	Lower
Shuffle Data	Large	Reduced

✅ Summary of All Optimization Techniques So Far

Case Study	Key Technique
Join Optimization	Broadcast joins
Predicate Pushdown	Filtering during read
Caching and Persistence	Avoid recomputation
Skew Join Optimization	Salting / Adaptive skew join
File Format & Partition Pruning	Use Parquet + prune with filters
Delta Lake Optimization	ZORDER, Vacuum, Compaction
Aggregation Optimization	Map-side, approximate, bucketing

Seekho Bigdata

Data is the New Oil

Let's build your Data Engineering journey together!

📩 Call us directly at: 9989454737

🌐 https://seekhobigdata.com/