12 PySpark Tricks

Karthik Kondpak

Swipe for more

1. Use selectExpr for Efficient Column Transformations

Instead of using multiple withColumn, use selectExpr for inline transformations.

```
df = df.selectExpr("id", "upper(name) as name",
"salary * 1.1 as updated_salary")
```

2. Use dropDuplicates Instead of distinct for Specific Columns

To remove duplicates based on certain columns, use dropDuplicates.

```
df = df.dropDuplicates(["name", "age"])
```

Swipe for more

https://www.seekhobigdata.com/

3. Optimize groupBy with agg

When performing aggregations, always use agg instead of multiple groupBy calls.

```
df.groupBy("department").agg({"salary": "avg",
"bonus": "sum"}).show()
```

4. Broadcast Smaller DataFrames for Joins

If one DataFrame is significantly smaller, use broadcast for better performance.

```
from pyspark.sql.functions import broadcast

df_result = df_large.join(broadcast(df_small), "id")
```

Swipe for more

5. Filter Early to Improve Performance

Push down filters as early as possible to minimize data shuffling.

```
df_filtered = df.filter(df.age > 30)
```

6. Use withColumn Efficiently

Instead of adding multiple columns one by one, use select or selectExpr for better performance.

```
df = df.withColumn("new_column",
df["existing_column"] * 10)
```

Swipe for more

7. Use cache and persist Wisely

Cache DataFrames that will be used multiple times to avoid recomputation.

```
df.cache()  # Stores the DataFrame in memory

df.persist()  # Default stores in memory, can specify different storage levels
```

8. Use explode to Work with Nested Data

If a column contains arrays, use explode to flatten them.

```
from pyspark.sql.functions import explode

df_exploded = df.withColumn("exploded_column",
explode(df["array_column"]))
```

Swipe for more

9. Use coalesce for Efficient Repartitioning

If you have too many small partitions, use coalesce to reduce them efficiently.

```
df = df.coalesce(5)

# Reduces partitions but avoids full shuffle
```

10. Use repartition for Evenly Distributed Data

When dealing with skewed data, use repartition to balance partitions.

```
df = df.repartition(10, "department")
```

Swipe for more

11. Use rdd.mapPartitions for Efficient Row-Level Operations

When working with large datasets, use mapPartitions instead of map for better performance.

```
df.rdd.mapPartitions(lambda partition:
some_function(partition))
```

12. Optimize Writing with partitionBy

When writing large datasets, partition them to improve query performance.

```
df.write.mode("overwrite").partitionBy("year",
"month").parquet("output_path")
```

Swipe for more

If you
find this
helpful like
and share

https://www.seekhobigdata.com/

+91 99894 54737