08 Spark Internals and Introduction to Higher Level APIs

> In Spark, a higher-level API lets you describe what you want to do with data, not how to execute it.
>
> APIs like DataFrame and Spark SQL provide schema and metadata, allowing Spark to automatically optimize execution.
>
> This makes code simpler, faster, and more maintainable than low-level RDD APIs.

1. DataFrames

- Distributed data with **schema and metadata**
- Like an RDD but with structure (column names, data types)
- Not persistent — disappears when Spark session ends
- Faster because Spark can optimize using schema (Catalyst, Tungsten)

2. Spark SQL

- Data stored on disk + metadata in **metastore**
- This creates a **persistent Spark table**
- Best for analytics and ETL since SQL is highly optimized
- Uses the same engine as DataFrames but persists metadata

3. Datasets (language specific)

- Strongly typed API available only in Scala and Java
- Not used much in industry
- Python developers rarely interact with Datasets

Low level API : RDD (for understanding only)

- Raw distributed data **without schema or metadata**
- Spark cannot optimize well because it knows nothing about structure
- Mostly avoided in real projects
- Important only to understand Spark internals (partitions, shuffles)

Key Concept

RDD = raw data

DataFrame = structured data with schema

Spark SQL = stored data with schema (persistent tables)

Higher-level APIs perform better **because Spark has more information** (schema, datatypes) → better optimization.

Dataframe (Hands-on):

1- read data from a file and create a dataframe

2- do transformations

3- write the results back to your storage

demo

```
spark = (
    SparkSession.builder
    .master("spark://spark-master:7077")
    .appName("dataframe_apis")
    .getOrCreate()
)

orders_df = spark.read \
.format("csv") \
.option("header","false") \
```

```
.option("inferSchema",'true') \ -- not preferred 1- sometimes dont infer right schema, performance issues
.load("/data/orders_300mb.csv")

df = spark.read.format("csv").option("header", "false").option("inferSchema","true").load("/data/orders_300mb.csv")
df.show()
# add headers to the file
df = spark.read.format("csv").option("header", "true").option("inferSchema","true").load("/data/orders_300mb.csv")

# total sales of delivered orders by state
df_filtered = df.filter("status == 'DELIVERED' ")
df2 = df_filtered.groupBy("state").sum("sales")
df2.show()
#df2.write.mode("overwrite").format("csv").save("/data/first_etl")
#df2.write.mode("overwrite").format("noop").save()

# shortcut method to read csv
orders_df = spark.read.csv("/data/orders_header.csv", inferSchema = True , header = True)
```

Enforcing schema

```
# infer schema based on sampling ratio, 1 means full table
df = spark.read.csv("/data/orders_300mb.csv", header = True , inferSchema=True , samplingRatio = 0.1)

# enforcing schema

orders_schema = 'order_id int,customer_id int ,product_id int, price float,order_date date,order_status string,state string,quantity int'

df = spark.read.csv("/data/orders_300mb.csv", header = True , schema = orders_schema)
```

```
note → if you give wrong datatype it wont fail, just have null values
orders_schema = 'order_id int,customer_id int ,product_id int, price float,or
der_date date,order_status string,state string,quantity int'
df = spark.read.csv("/data/orders_300mb.csv", header = True , schema = o
rders_schema)
```

Lazy evaluation :

Spark does **not execute transformations immediately**.

It records them and executes only when an **action** is called.

Transformations are lazy and they build the **logical plan**. (DAG)

Examples: `select` , `filter` , `groupBy` , `withColumn` .

Actions trigger **actual execution**. Every action creates one job.

Examples: `show` , `count` , `write` , `collect` .

Transformations are lazy, actions are eager.

number of jobs = number of actions

demo:

```
use previous code
show number of jobs with and without inferSchema
```

Initial number of partitions in a dataframe

When a DataFrame is created, Spark **immediately decides how many partitions** the data will be split into. This decision happens **at read time**, before any transformation.

For file formats like CSV, JSON, Parquet:

- Spark divides data into partitions based on **file size**

- Each partition is roughly **128 MB by default**

This is controlled by:

```
spark.sql.files.maxPartitionBytes (default ~128 MB)
spark.conf.get("spark.sql.files.maxPartitionBytes")
```

spark.sparkContext.defaultParallelism = number of cores =number of tasks that can run in parallel

Number of partitions will be → max(number of cores , filesize/128mb)

demo

```
orders_schema = 'order_id int , customer_id int ,product_id int, amount dou
ble , order_date string, order_status string , state string'
orders_df = spark.read.csv("/data/orders_500mb.csv" , schema = orders_s
chema , header = True , inferSchema=False)
orders_df.rdd.getNumPartitions()

#show with 40mb, 300mb, 500mb, 1gb files
```

Transformations fall into 2 category : Narrow and Wide

Narrow transformations

Each output partition is derived from **exactly one input partition**.

Data is processed **locally within the same executor**, so no data movement happens.

Multiple narrow transformations can be **pipelined into a single stage**.

Examples: `select` , `filter` , `withColumn` , `map`

Wide transformations

Each output partition depends on **data from multiple input partitions**.

Spark must **redistribute data across executors**, which triggers a shuffle.

Wide transformations **break stages** and are the most expensive operations.

Examples: `groupBy` , `join` , `orderBy` , `distinct`

Every wide transformation creates a new stage .

number of stages = 1 + number of wide transformations.

after each stage data is written to the disk (shuffle write) and then next stage read that data(shuffle read) for further processing

after wide transformation Spark creates 200 partitions by default. (spark.sql.shuffle.partitions)

Number of tasks in each stage = number of partitions

number of tasks that can run in parallel = number of cores

Internals of `groupBy` in Spark

- `groupBy` is a **wide transformation**. It **always triggers a shuffle** because rows with the same key must come to the same executor.
- Execution happens in **two phases**:
  1. **Partial aggregation (map-side)**

     Spark aggregates data locally within each partition to reduce shuffle size.
  2. **Final aggregation (reduce-side)**

     Shuffled data is merged and final results are computed.
- During shuffle:

    - data is **partitioned by key** (hash partitioning by default)
    - records are **transferred over the network**
    - each shuffle partition becomes one task in the next stage
- The number of output partitions after `groupBy` is controlled by:

```
spark.sql.shuffle.partitions (default = 200)
```

Internals of Join

- A join in Spark is a **wide transformation** because data with the same join key must be brought together on the same executor.
- When Spark cannot avoid it, a **shuffle happens on both datasets** involved in the join.
- During shuffle:
    - data is **partitioned by the join key** (hash partitioning by default)
    - records are **transferred over the network**
    - each shuffle partition becomes a task in the next stage
- After shuffle:
    - each executor partition contains matching keys from **both datasets**
    - Spark performs the join **partition by partition**
- The number of partitions after shuffle is controlled by:

```
spark.sql.shuffle.partitions

#demo
orders_schema = 'order_id int, customer_id int, product_id int , price flo
at, order_date date, order_status string, state string, quantity int'
df = spark.read.csv("/data/orders_1gb.csv",header=True , schema = or
ders_schema)
product_schema = 'product_id int, product_name string, cost_price int'
df_products= spark.read.csv("/data/products.csv" , header=True, sche
ma = product_schema)
```

```
df_joined = df.join(df_products, "product_id" , "inner")
df_joined.write.mode("overwrite").format("noop").save()
```

Key intuition

- GroupBy → shuffle **one dataset**
- Join → shuffle **two datasets**

This makes joins **more expensive than groupBy** in most cases.

Key performance implications (important to say out loud)

- `groupBy` is **one of the most expensive operations** in Spark.
- High cardinality keys → more memory usage.
- Skewed keys → uneven task durations.
- Poor partition sizing → high GC or long runtimes.

Internals of joins :

demo :

```
show on excel first how partitions process..
shuffle between executors ..same key goes to same partition
show spark UI

orders_schema = 'order_id int , customer_id int ,product_id int, amount dou
ble , order_date string, order_status string , state string'
df = spark.read.csv("/data/orders_500mb.csv" , schema = orders_schema ,
header = True , inferSchema=False)
df2 = df1.groupBy("state","customer_id").agg(sum("sales").alias("total_sale
s"))
df3= df2.groupBy("state").avg("total_sales")
df3.write.mode("overwrite").format("noop").save()
```

repartition vs coalesce

repartition

`repartition()` is used to **increase or decrease partitions** with a **full shuffle**.

- Redistributes data evenly across partitions
- Uses hash partitioning when a column is provided
- Expensive operation due to shuffle

```
df2 = df.repartition(8)
df2 = df.repartition(8,"state")

#demo
orders_schema = 'order_id int , customer_id int ,product_id int, amount dou
ble , order_date string, order_status string , state string'
df = spark.read.csv("/data/orders_500mb.csv" , schema = orders_schema ,
header = True , inferSchema=False)
#df2 = df1.groupBy("state","customer_id").agg(sum("sales").alias("total_sal
es"))
#df2.groupBy("state").count()
df2 = df.repartition(8) # df.repartition(8,"state")
#df2.rdd.getNumPartitions()
df2.filter("order_status='DELIVERED'").write.mode("overwrite").format("no
op").save()

Note : when you see the UI , the data is filtered first and then repartition is
done.
```

Use it when:

- You want better parallelism
- Before joins or aggregations
- You need even data distribution
- let say after filter very less data in each partition that time you want to reduce the partitions

coalesce

`coalesce()` is used to **only reduce partitions** without a shuffle.

- Merges existing partitions
- Faster and cheaper than repartition
- Can cause uneven partition sizes

```
df2 = df.coalesce(4) #coalesce(3)
```

Use it when:

- Reducing partitions before write
- Avoiding small files
- You do not care about data distribution

Key differences

repartition	coalesce
Shuffle happens	No shuffle
Can increase partitions	Only decreases partitions
Even distribution	Can be skewed
Expensive	Cheap

Takeaway

> repartition reshuffles data,
> coalesce just merges partitions.