SPARK

- A distributed computing engine, which distributes our data to process it.

Ex (1GB DATA) ← IPC → increase resources but upto a limit

Cluster (Connect all your computer classes machines → tell them to act as 1 machine)

Architecture (where it distributes data among machines) →
- Connect them together
- no saturation point / no limit
- cluster of machines

(NODE ↔ Computer in spark language)

Driver Program ↔ Cluster Manager ↔ Worker Node, Worker Node

Driver Program:
will orchestrate the tasks (break down the code into smaller tasks & send that info to →
(3) Asses/break ~~the~~ code into stages/jobs/tasks & hand it back to Cluster Manager
(4) Requests (C-M) to create (2 worker Nodes) based on info

Cluster Manager:
- will create worker nodes based on info & those → (Manages Resources)
(1) Create a driver node on our cluster
(2) when we submit our code, it will go to driver node

Worker Node:
- will actually process our data transformations/ processing/ everything
(5) All breakdown info which was provided to (CM) from (D-P) → will go to these 2 workers
(6) Execute all transformations

Why spark ?

- Hadoop uses disk memory which took lot of processing time
  Spark uses "in memory computation" (RAM) → FASTER
- fault tolerance
- Partioning
- Lazy Evaluation

HADOOP — Disk Based memory
- Like a Notebook, you write results down, then look at them again next time.
  Writing / Reading takes time.

SPARK — In memory
- Like a whiteboard (RAM)
  you can write, read, erase, update instantly w/o going back to your notebook
  RAM is designed for temp storage & ultrafast access
  - Can Run on your PC / Single machine / No point
  - Can run on cluster of machines → On Prenise / cloud
    each machine has its own RAM, CPU.
    spark distributes data across nodes.

On prenise
- hardwares & servers are physically located at your company
  Your (IT dept) manages network, security, storage electricty, cooling.
  ~~No~~ scaling is slower, limitations, expensive

Cloud
- Computing resources are hosted & managed by cloud provider
  Expensive

Lazy Evaluation

Ex: DATAFRAME

And we perform some transformations.

- Transf 1 (selecting col's)
- Trans 2 (adding 1 row)
- Transf 3 (deleting 2 row)

Spark will store all these into a Logical Plan & optimize it..

↳ Once we trigger the action → it will execute the logical plan. performing all transactions

- Remembers | builds a plan |
  Combine steps together
  Choose the fastest path (DAG Optimization)

When we records a plan

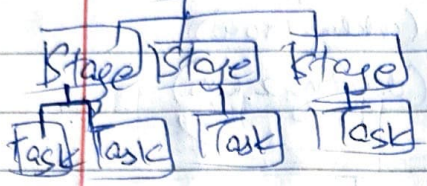
1. When we write transformations (filter, group) → 1 direction, No loops
2. Spark records them as a plan → | Directed Acyclic Graph |
   DAG is basically a map (spark's map) of all steps it needs to take.
3. Spark's optimizer → looks at the plan & asks "Is there any better/faster way to get same result.

DAY2

7/10/25

Hierarchal Structure to execute Jobs

JOB → the code which we submit in every cell



SPARK APIs

↳ allow us to write code in our native language (depending on what developer is comfortable with)

interface

A set of rules/definitions/tools, that allows different software systems to interact with each other.

Ex: B/w your code & (client) & website (server)

Python + Spark → Pyspark library

Modern way of making table

↳ Delta Lake is default storage format for tables in databricks.

↳ It takes raw data → stores it in optimized, reliable format

∴ Creating a Table → meaning creating a Delta Table

Other way

↳ Create a Volume (secure storage locker)

(we get a temporary df in memory

↳ upload CSV → & then read

DA73

8/10/25

Code

- spark.read.format("csv")
- .option("inferSchema", True)

  Automatically detects datatypes (int, float, string)
- .option("header", True)

  Treats first row as column names
- .load('/Volumes/workspace ---- path')

To trigger a action

- df.show() displays text based preview
  df.display() // it will actually execute this time // display beautiful, interactive table letting you to sort/filter/charts etc.

For JSON

.option('multiLine', False)

Schema → df.printSchema()

Create

```
my_ddl_schema = '''
                Item_weight STRING,
                Item_MRP INT,
                '''

df = spark.read.format('csv')\
        .schema(my_ddl_schema)\
        .option('header', True)\
        .load('')
```

Select

```
from pyspark.sql.functions import col
df_sel = df.select('Item_Weight', 'Item_MRP').display
```

OR

```
df_sel = df.select(col('Item_Weight'), col('Item_MRP')).display
```

Alias

```
df.select(col('Item_weight').alias('G')).display()
```

Filter

```
df.filter(col('Item_Fat') == 'Regular').display
df.filter(col('Item_type') == 'Coke' & (col('Item_MRP') <= 10))
df.filter((col('Size').isNull()) & col('Outlet').isin('Tier 1', 'Tier 2'))
```

9 Oct
Day 4

Rename

```
df.withColumnRenamed('Weightt', 'Weight')
```

Create / Modify column

```
df = df.withColumn('Flag', lit("new"))
```

↳

Flag
new
new
new

```
df.withColumn("multiply", col("Weight") * col("MRP")).display()
```

Replace / Edit content

```
df.withColumn('Fat', regexp_replace(col('Fat'),
    "Regular", "Reg"))
  .withColumn('Fat', regexp_replace(col('Fat'),
    'low fat', 'LF'))
```

10 Oct

Day 5

Type Casting

```
df = df.withColumn("Weight", col("Weight").cast(StringType())
                                          .cast(IntegerType()))
```

Sort / OrderBy

```
df.sort("Weight")
df.sort("Weight")
df.sort(col("Weight").desc())

df.filter(col("Weight").isNotNull())
  .sort(col("Weight").asc())

df.sort(col("Weight").asc(), col("MRP").desc())
```

Limit

```
df.limit(10).display()
```

Oct 11

Day 6

- Drop

df.drop(col('MRP')).display()

- df.dropDuplicates().display

→ df.dropDuplicates(subset=['MRP']).display()

List

OR

→ df.distinct().display()

---

date1 = [(1,'Gurtej'), (2,'Harsh')]

schema1 = """ id INT, name STRING """

df1 = createDataFrame(date1, schema1)

date2

scheme2

df2

→ df1.union(df2).display()

df1.unionByName(df2)

df1.unionByName(df2, allowMissingColumns=True)

Oct 12

Day 7

String

df.select(initcap('Item_type')).display()

lower('')

upper('').alias('upper_Item_type')

df.withColumn('', date_format

Date

df.withColumn('Current Date', current_date())

df.withColumn('Week After', date_add('Current Date', 7))

df.withColumn("Week_Before", date_sub('Current_Date', 7))

OR

date_add('Curent_Date', -7))

df.withColumn('date_diff', datediff('Curent_Date', 'Week_After'))

modify existing

df.withColumn('week_before', date_format('week_before', 'dd-MM-yyyy'))

yyyy-MM-dd

ORIGINAL

drop Na → that rows which had

df.dropNa('all') → remove null from all col'n

df.dropNa('any') → any col'n

→ df.dropNa(subset=['Order_Type'])

Filling Nulls

df.fillna('NA')

df.fillna('NA', subset=['Outlet_type'])

Split Indexing

① df.withColumn('Outlet', split('Outlet', ' ')) (delimeter)

↓ ['Outlet_A', 'Type_A']

② df.withColumn('Name', col('Outlet').getItem(0))

.withColumn('Type', col('Outlet').getItem(1))

or

.withColumn('Outlet', split('Outlet', ' ')[1]) (only give Type)

df.withColumns('flag', array_contains('Outlet', 'Type1'))

groupBy

```
df.groupBy('Item').agg(sum('MRP'))
                       avg
df.groupBy('Item', 'Size').agg(sum('MRP').
df.groupBy('Item_type', 'Size').agg(sum('MRP'), avg('MRP'))
                                    alias(...)
```

14 Mock

Day 9

SPARK

collect list

df.groupBy('user').agg(collect_list('book'))

pivot

df.groupBy('Type').pivot('Size').agg(avg('MRP'))

when otherwise

df.withColumn('Veg', when(col('Type')=='Meat', 'Non-Veg').otherwise('Veg'))

Inner join

df1.join(df2, df1.dept_id==df2.dept_id, 'inner').drop(df2.dept_id)

df1.join(df2, on="dept_id", how="inner")

left → All from left | Both col'ns

left semi → from left | but matching | And only col'ns from left | Inner join + left col'ns

left anti → only what is not in right

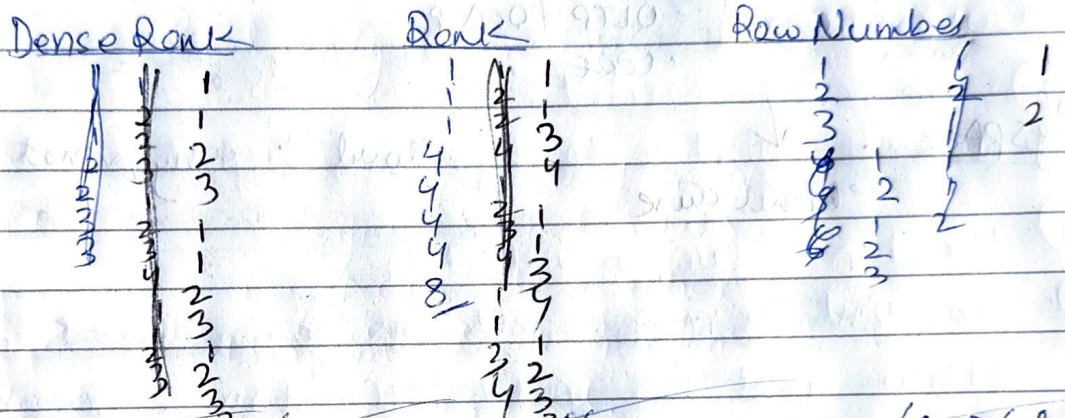
16th Oct

Day 11

Window functions

from pyspark.sql.window import Window

- df.withColumn("row_col", row_number().over(Window.orderBy("Item"))).display()



~~df.withColumn("rank", rank().over(Window.orderBy(col("").desc())))~~

df.withColumn("1--", row_number().over(Window.partitionBy("Item").orderBy("MRP")))

18th Oct

Day 13

STORAGE

Data writing — Azure → Data Lake, AWS → S3

provide a dedicated location

```
df.write.format("csv")
  .save("/FileStore/tables/csv/date-csv")
```

→ Append

```
df.write.mode("append")
  .format("csv").save("")
```

- Schema must match (same cols, data types)

```
data1 = [(1, "TV", 500)]
schema1 = "id INT, product STRING, price INT"
df1 = spark.createDataFrame(data1, schem.1)
df1.write.mode("overwrite").parquet("/mnt/date")
```

```
data2
schema2
df2
df2.write.mode("append").parquet("/mnt/date")
```

```
df_all = spark.read.parquet("/mnt/date")
df_all.show()
```

1 Dataframe — multiple files

/mnt/data/

part 0000.parquet (contains 1, TV, 500)

part 0001.parquet (contains 2, Cell, 100)

CSV : human readable → Row Based

Parquet : Binary, not human readable
Faster, for large scale datasets
Columnar storage format
↳ allows for more efficient querying
→ But no transactional control
can't edit → only rewrite
Read Only, Immutable

Fetches only necessary cols

Delta : Its a layer built on top of Parquet that adds
↳ ACID transactions
↳ optimised file management
↳ Time travel
Because of json log

A Real DBase

json_log → A folder containg files representing versions of your tables.
Every time you update, delete
↳ delta creates new log file
never overwrites old files
like Git → version tracking

Writing modes
↳ append, overwrite, ignore, error

SQL in Pyspark

- We read data in pyspark using dataframe
  ↳ A python object
- for SQL engine to read the data we need to create ~~temporary view~~ table.

df.createTempView("sales")
or df.createOrReplaceTempView("sales")

- But when notebook session ends → temp view disappears

→ If u want permanent table
  ↳ (2) df.write.saveAsTable("bigmart.sales_table")
  (1) spark.sql("Create Database if not exists bigmart")

- Convert SQL Query to dataframe

dfsql = spark.sql("select * from view where ....")