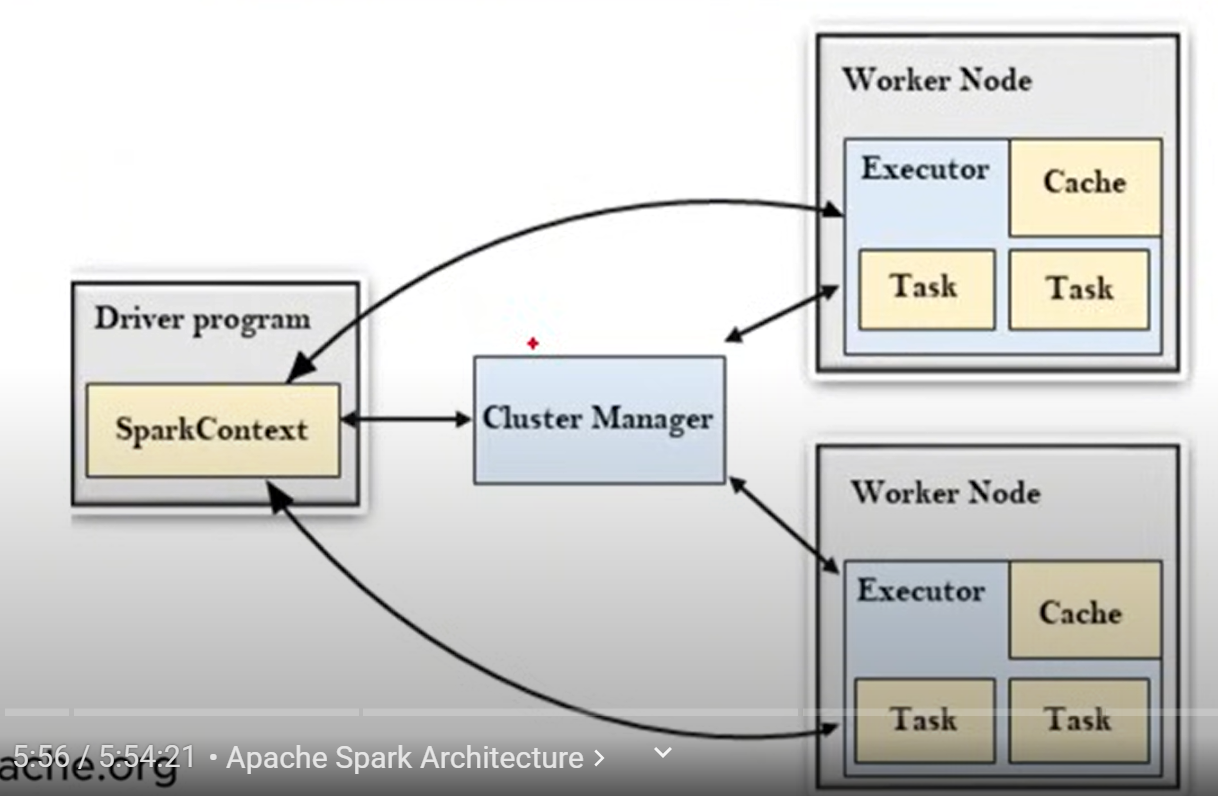
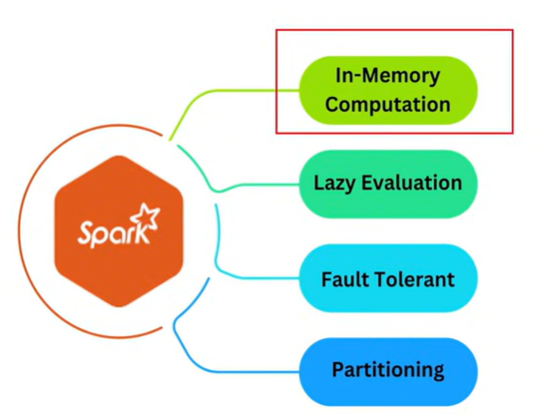
What is SPARK?

It is a distributed computing engine which distributes data to process it.

Spark Architecture





Lazy Evaluation – We have a dataset and we have performed some transformations, it first creates a logical plan because one transformation can be performed faster than other and when we run the logical plan will be executed.

* To upload the data

1. Click on catalogue
2. Create table
3. Browse and upload your data set

* Create a notebook. Create a cluster and attach it to notebook
* Data reading

[code – spark.read.format(‘csv’).option(‘inferSchema’,True).option(‘header’,True).load(‘url of file’)]

* If we don’t remember the path use [dbutils.fs.ls(‘/FileStotr/tables/’)]
* To read json files we replace format as json and put an option(‘multiLine’, True if they are multiline and false if they are single line)
* To change the datatypes there are 2 methods with DDL and structType()
* For DDL - first create definition

my\_ddl\_schema = '''

                   Item\_Identifier STRING,

                   Item\_Weight STRING,

                   Item\_Fat\_Content STRING,

                   Item\_Visibility DOUBLE,

                   Item\_Type STRING,

                   Item\_MRP DOUBLE,

                   Outlet\_Establishment\_Year INT,

                   Outlet\_Size STRING,

                   Outlet\_Location\_Type STRING,

                   Outlet\_Type STRING,

                   Item\_Outlet\_Sales DOUBLE

                   '''

* Inplace of inferschema write .schema(definition variable)
* For structType()
* Create definition
* my\_strct\_schema = StructType([
* StructField('Item\_Identifier',StringType(),True),
* StructField('Item\_Weight',StringType(),True),
* StructField('Item\_Fat\_Content',StringType(),True),
* StructField('Item\_Visibility',StringType(),True),
* StructField('Item\_MRP',StringType(),True),
* StructField('Outlet\_Identifier',StringType(),True),
* StructField('Outlet\_Establishment\_Year',StringType(),True),
* StructField('Outlet\_Size',StringType(),True),
* StructField('Outlet\_Location\_Type',StringType(),True),
* StructField('Outlet\_Type',StringType(),True),
* StructField('Item\_Outlet\_Sales',StringType(),True),
* ])
* To select particular columns [select]
* df.select('Item\_Identifier', 'Item\_Weight', 'Item\_Fat\_Content').display()
* df.select(col('Item\_Identifier'), col('Item\_Weight'), col('Item\_Fat\_Content')).display() #to create column object
* Alias function

df.select(col('Item\_Identifier').alias('Item\_ID')).display()

* Filter/where

1. Filter the data with at content = Regular

df.filter(col('Item\_Fat\_Content')=='Regular').display()

1. Slice the data with item type = Soft Drinks and weight<10

df.filter((col('Item\_Type')=='Soft Drinks') & (col('Item\_Weight')<10)).display()

1. Fetch the data with Tier in (Tier1 or Tier2) and Outlet Size is Null

df.filter((col('Outlet\_Size').isNull()) & (col('Outlet\_Location\_Type').isin('Tier 1','Tier 2'))).display()

* withColumnRenamed – to rename a column

df.withColumnRenamed('Item\_Weight','Item\_Wt').display()

* withColumn – to create a new column and lit – to put a constant value

df = df.withColumn('flag',lit('new')

df.withColumn('Item\_Fat\_Content',regexp\_replace(col('Item\_Fat\_Content'),'Regular','Reg')).display()

if we give the existing column name it will modify the column and if we give a new name it will create a new column

* sort/orderBy

df.sort(col('Item\_Weight').desc()).display()

df.sort(['Item\_Weight','Item\_Visibility'],ascending=[0,1]).display()

// for multiple columns, here 0 and 1 are Boolean values for setting up ascending and descending order

* Limit
* df.limit(10).display()
* drop

#multiple columns

df.drop('Item\_Visibility','Item\_Type').display()

* drop\_duplicates

df.dropDuplicates().display()

there are 2 one is dropDuplicates and other one is drop\_duplicates the one with \_ is used in pandas

* df.drop\_duplicates(subset=['Item\_Type']).display()

for subsets

* for multiple columns

df.drop('Item\_Visibility','Item\_Type').display()

* distinct()

df.distinct().display()

* UNION AND UNION BYNAME
* Union just stacks the dataframes

df1.union(df2).display()

* unionByName – The union byname function in pyspark combines two dataframes with potentially different column orders, matching columns by name rather than position.

df1.unionByName(df2).display()

* STRING FUNCTIONS
* INITCAP() – makes first letter capital

df.select(initcap('Item\_Type')).display()

* UPPER() – Makes every thing to upper case

df.select(upper('Item\_Type')).display()

* LOWER() – To lower case

df.select(lower('Item\_Type')).display()

* DATE FUNCTIONS
* CURRENT\_DATE()

df=df.withColumn('cur\_date',current\_date())

df.display()

* Date\_Add()

df= df.withColumn('week\_after',date\_add('cur\_date',7))

df.display()

* Date\_Sub()

df =df.withColumn('week\_before',date\_sub('cur\_date',7))

df.display()

* Datediff

df=df.withColumn('datediff',datediff('cur\_date','week\_before'))

df.display()

* DATE\_FORMAT

df = df.withColumn('week\_before',date\_format('week\_before','dd-MM-yyyy'))

df.display()

* HANDLING NULLS
* Dropping Nulls

// to drop the records which has all the values as null

df.dropna('all').display()

// to drop records which has atleast one null value

df.dropna(‘any’).display()

// to drop a records which has nulls in a specific column

df.dropna(subset=['Outlet\_Size']).display()

* Filling Nulls

// to fill all the null values

df.fillna('NotAvailable').display()

// to fill null values in a specific column

df.fillna('NotAvailable',subset=['Outlet\_Size']).display()

* SPLIT AND INDEXING
* To split a word based on some delimeter

df.withColumn('Outlet\_Type',split('Outlet\_Type',' ')).display()

* To index it

df.withColumn('Outlet\_Type',split('Outlet\_Type',' ')[1]).display()

* Explode – the explode function takes a column with arrays or maps and turns each element into its own row

df\_exp.withColumn('Outlet\_Type',explode('Outlet\_Type')).display()

* Array\_Contains

Checks whether the array contains an element or not

df\_exp.withColumn('Type1\_flag',array\_contains('Outlet\_Type','Type1')).

display()

* GROUP BY

df.groupBy('Item\_Type').agg(sum('Item\_MRP')).display()

* Group by and aggregation

df.groupBy('Item\_Type').agg(avg('Item\_MRP')).display()

* for multiple columns

df.groupBy('Item\_Type','Outlet\_Size').agg(sum('Item\_MRP').alias('Total\_MRP')).display()

* COLLECT\_LIST

The collect\_list function in pyspark is a way to gather up all the values from a specific column in your data. Imagine you have a bunch of rows, and you want to see all the different things in one of the columns. Collect\_list() takes all those values and puts them into a single list for you

//df\_book.groupBy('user').agg(collect\_list('book')).display()

* PIVOT

The pivot function in pyspark is a tool for reshaping your data. Imagine you have a table where the values in one column are categories and you want to turn those categories into separate columns. Pivot takes the unique values from a column, spreads them out as new column, spreads them out as new columns, and fills in those new columns with aggregated values from another column.

//df.groupBy('Item\_Type').pivot('Outlet\_Size').agg(avg('Item\_MRP')).display()

* When-otherwise

Its like a ternary operator

df = df.withColumn('veg\_flag',when(col('Item\_Type')=='Meat','Non-veg').otherwise('Veg'))

df.display()

* JOINS
* **Inner Join**: Returns only the matching rows from both tables based on the join condition.

df1.join(df2, df1['dept\_id']==df2['dept\_id'],'inner').display()

* **Left Join**: Returns all rows from the left table and matching rows from the right table; unmatched right rows are filled with nulls.

df1.join(df2, df1['dept\_id']==df2['dept\_id'],'left').display()

* **Right Join**: Returns all rows from the right table and matching rows from the left table; unmatched left rows are filled with nulls.

df1.join(df2, df1['dept\_id']==df2['dept\_id'],'right').display()

* **Full Join**: Returns all rows from both tables, with nulls where there is no match on either side.

df1.join(df2, df1['dept\_id']==df2['dept\_id'],'full').display()

* **Anti Join**: Returns rows from the left table that **do not** have a matching row in the right table.

df1.join(df2, df1['dept\_id']==df2['dept\_id'],'anti').display()

* WINDOW FUNCTIONS
* Row Number()

The row number function in PySpark is called row\_number(). It’s a window function that assigns a unique sequential number to each row within a window partition.

//df.withColumn('rowCol',row\_number().over(Window.orderBy('Item\_Identifier'))).display()

* Rank()

The rank function in pyspark is like a special way of numbering items in a list or table. It assigns a unique rank to each item based on its value compared to the others. If there are 4 records with same value rank function will assign 1 to all 4 records but to the next value it will assign 5.

**//df.withColumn('rank',rank().over(Window.orderBy('Item\_Identifie dfr'))).display()**

* Dense rank()

Dense rank will just assign the continuous flow of the numbers for ranking

//df.withColumn('denseRank',dense\_rank().over(Window.orderBy(col('Item\_Identifier').desc()))).display()

* Cumulative sum

It is used to calculate the cumulative sum of the values till specified row

//df.withColumn('cumsum',sum('Item\_MRP').over(Window.orderBy('Item\_Type').rowsBetween(Window.unboundedPreceding,Window.currentRow))).display()

* DATA WRITING MODES
* **APPEND**: Adds new data to the existing data in the target location without modifying the original content.

df.write.format('csv')\

    .mode('append')\

        .save('/FileStore/tables/CSV/data.csv')\

            .save()

* **OVERWRITE**: Replaces the existing data in the target location with the new data.

df.write.format('csv')\

    .mode('overwrite')\

        .save('/FileStore/tables/CSV/data.csv')\

            .save()

* **ERROR(or errorifexists)**: Throws an error if data already exists at the target location.

df.write.format('csv')\

    .mode('error')\

        .save('/FileStore/tables/CSV/data.csv')\

            .save()

* **IGNORE**: Skips writing the new data if the target location already has data, without raising an error.

df.write.format('csv')\

    .mode('ignore')\

        .save('/FileStore/tables/CSV/data.csv')\

            .save()

* PARQUET – columnar format

df.write.format('parquet')\

    .mode('overwrite')\

        .option('path','/FileStore/tables/CSV/data.csv')\

            .save()

* MANAGED VS EXTERNAL
* **Managed Table**: In a managed table, Spark controls both the metadata and the actual data, deleting everything when the table is dropped.
* **External Table**: In an external table, Spark only manages the metadata, and dropping the table leaves the actual data untouched.