

# CHAPTER-1

## PYSPARK

### 1.1 Introduction

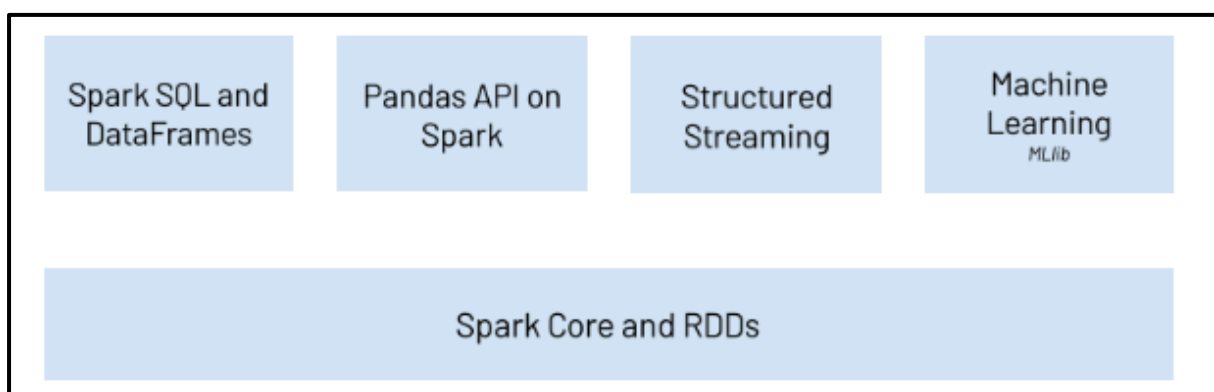
Datasets are becoming huge. Infact, data is growing faster than processing speeds. Therefore, algorithms involving large data and high amount of computation are often run on a distributed computing system. A distributed computing system involves nodes (networked computers) that run processes in parallel and communicate (if, necessary).

- MapReduce – The programming model that is used for Distributed computing is known as MapReduce. The MapReduce model involves two stages, Map and Reduce.
  1. Map – The mapper processes each line of the input data (it is in the form of a file), and produces key – value pairs.
  2. Input data → Mapper → list([key, value])
  3. Reduce – The reducer processes the list of key – value pairs (after the Mapper's function). It outputs a new set of key – value pairs.

Spark – Spark (open source Big-Data processing engine by Apache) is a cluster computing system. It is faster as compared to other cluster computing systems (such as, Hadoop).

It provides high level APIs in Python, Scala, and Java. Parallel jobs are easy to write in Spark. We will cover PySpark (Python + Apache Spark), because this will make the learning curve flatter. To install Spark on a linux system, follow [this](#). To run Spark in a multi – cluster system, follow [this](#). We will see how to create RDDs (fundamental data structure of Spark).

RDDs (Resilient Distributed Datasets) – RDDs are immutable collection of objects. Since we are using PySpark, these objects can be of multiple types. These will become more clear further.



## 1.2 Programs

1.2.1. You have been given a CSV file named `customer_data.csv` containing customer information. The goal is to perform a series of data transformations and clean the data using PySpark. The CSV file has the following schema:

- a. `customer_id` (String): Unique identifier for each customer. `name` (String): Name of the customer.
  - b. `age` (Integer): Age of the customer. `gender` (String): Gender of the customer ('M' for Male, 'F' for Female).
  - c. `country` (String): Country of residence.
  - d. `signup_date` (String): Date when the customer signed up (in the format YYYY-MM-DD).
  - e. `last_purchase_amount` (Float): The amount of the last purchase made by the customer.
  - f. `total_purchases` (Integer): Total number of purchases made by the customer.
- a. Load the CSV file into a PySpark DataFrame.
  - b. Display the first 5 rows of the DataFrame.

OUTPUT :

```
data = ([[111, "Abhishek", 30, 'Male', 'India', '04-05-2023', 30000, 3, 70000, 1000],
         [121, "Vinay", 50, 'Male', 'Pak', '30-05-2024', 45000, 2, 200, 100],
         [131, "Bhuvika", 24, 'Female', 'India', '24-05-2024', 46000, 5, 100, 2000]])
df = spark.createDataFrame(data, ["c_id", "c_name", "age", "gender", "country", "date_of_purchase", "amount", "time", "total_purchase", "last_purchase"])
df.show()
```

c_id	c_name	age	gender	country	date_of_purchase	amount	time	total_purchase	last_purchase
111	Abhishek	30	Male	India	04-05-2023	30000	3	70000	1000
121	Vinay	50	Male	Pak	30-05-2024	45000	2	200	100
131	Bhuvika	24	Female	India	24-05-2024	46000	5	100	2000

- c. Print the schema of the DataFrame to verify the data types.

OUTPUT :

```
df.printSchema()

root
 |-- c_id: long (nullable = true)
 |-- c_name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- country: string (nullable = true)
 |-- date_of_purchase: string (nullable = true)
 |-- amount: long (nullable = true)
 |-- time: long (nullable = true)
 |-- total_purchase: long (nullable = true)
 |-- last_purchase: long (nullable = true)
```

- d. Create a new column age\_group which categorizes customers into:

- ☐ 'Teen' if the age is less than 20
- ☐ 'Young Adult' if the age is between 20 and 35 (inclusive)
- ☐ 'Adult' if the age is between 36 and 50 (inclusive)
- ☐ 'Senior' if the age is above 50

OUTPUT:

```
from pyspark.sql.functions import *
df=df.withColumn("age_group",when(df.age<20,"Teen").when(df.age.between(20,35),"Young Adult").when(df.age.between(36,50),"Adult")
df.show()
```

c_id	c_name	age	gender	country	date_of_purchase	amount	time	total_purchase	last_purchase	age_group
111	Abhishek	30	Male	India	04-05-2023	30000	3	70000	1000	Young Adult
121	Vinay	50	Male	Pak	30-05-2024	45000	2	200	100	Adult
131	Bhuvika	24	Female	India	24-05-2024	46000	5	100	2000	Young Adult

- e. Standardize the gender column to have values 'Male' and 'Female' instead of 'M' and 'F'.

OUTPUT:

```
#e Standardize the gender column to have values 'Male' and 'Female' instead of 'M' and 'F'.
def replace_gender(gender):
    if gender == 'Male':
        return 'M'
    elif gender == 'Female':
        return 'F'
gen = udf(replace_gender, StringType())
df = df.withColumn('gender', gen(df.gender))
df.show()
```

c_id	c_name	age	gender	country	date_of_purchase	amount	time	total_purchase	last_purchase	age_group
111	Abhishek	30	M	India	04-05-2023	30000	3	70000	1000	Young Adult
121	Vinay	50	M	Pak	30-05-2024	45000	2	200	100	Adult
131	Bhuvika	24	F	India	24-05-2024	46000	5	100	2000	Young Adult

- f. Create a new column high\_value\_customer which is 'Yes' if total\_purchases is greater than 100 or last\_purchase\_amount is greater than 1000, otherwise 'No'.

```
'''f Create a new column high_value_customer which is 'Yes' if total_purchases is greater
than 100 or last_purchase_amount is greater than 1000, otherwise 'No'.'''
df=df.withColumn("high_value_customer",when(df.total_purchase>100,"Yes").when(df.last_purchase>1000,"Yes").otherwise("No"))
df.show()
```

c_id	c_name	age	gender	country	date_of_purchase	amount	time	total_purchase	last_purchase	age_group	high_value_customer
111	Abhishek	30	M	India	04-05-2023	30000	3	70000	1000	Young Adult	Yes
121	Vinay	50	M	Pak	30-05-2024	45000	2	200	100	Adult	Yes
131	Bhuvika	24	F	India	24-05-2024	46000	5	100	2000	Young Adult	Yes

- g. Filter out rows where age is less than or equal to 0 or greater than 120.

```
#g Filter out rows where age is less than or equal to 0 or greater than 120.
df=df.filter(df.age.between(0,120))
df.show()
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|c_id|  c_name|age|gender|country|date_of_purchase|amount|time|total_purchase|last_purchase|  age_group|high_value_customer|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 111|Abhishek| 30|    M|  India|    04-05-2023| 30000|  3|       70000|       1000|Young Adult|             Yes|
| 121|  Vinay| 50|    M|   Pak|    30-05-2024| 45000|  2|        200|        100|   Adult|             Yes|
| 131| Bhuvika| 24|    F|  India|    24-05-2024| 46000|  5|        100|       2000|Young Adult|             Yes|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

h. Filter out rows where total\_purchases is less than 0.

```
#h Filter out rows where total_purchases is less than 0.
df=df.filter(df.total_purchase>=0)
df.show()
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|c_id|  c_name|age|gender|country|date_of_purchase|amount|time|total_purchase|last_purchase|  age_group|high_value_customer|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 111|Abhishek| 30|    M|  India|    04-05-2023| 30000|  3|       70000|       1000|Young Adult|             Yes|
| 121|  Vinay| 50|    M|   Pak|    30-05-2024| 45000|  2|        200|        100|   Adult|             Yes|
| 131| Bhuvika| 24|    F|  India|    24-05-2024| 46000|  5|        100|       2000|Young Adult|             Yes|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

```

i. Group the data by country and age\_group, and calculate the following aggregations:

- Total number of customers
- Average last purchase amount
- Total number of purchases

```
'''i Group the data by country and age_group, and calculate the following
aggregations:
```

- ☒ Total number of customers
- ☒ Average last purchase amount
- ☒ Total number of purchases'''

```
aggregated_df = df.groupBy("country", "age_group").agg(
    count("*").alias("total_customers"),
    avg("last_purchase").alias("average_last_purchase_amount"),
    sum("total_purchase").alias("total_number_of_purchases")
)
```

```
aggregated_df.show()
```

```
+-----+-----+-----+-----+-----+
|country|  age_group|total_customers|average_last_purchase_amount|total_number_of_purchases|
+-----+-----+-----+-----+-----+
|  India|Young Adult|             2|             1500.0|             70100|
|   Pak|   Adult|             1|             100.0|             200|
+-----+-----+-----+-----+-----+

```

1.2.2. You are given a DataFrame containing information about employees in a company.

The DataFrame has the following schema:

- employee\_id (String): Unique identifier for the employee
- name (String): Name of the employee
- department (String): Department where the employee works
- salary (Float): The employee's salary
- years\_of\_experience (Integer): The number of years the employee has been working

Queries:

a. Add a column experience\_level: Create a new column that categorizes the employee's experience into "Junior", "Mid", and "Senior":

- "Junior" if the years\_of\_experience is less than 3
- "Mid" if the years\_of\_experience is between 3 and 7 (inclusive)
- "Senior" if the years\_of\_experience is greater than 7

```
'''a . Add a column experience_level: Create a new column that categorizes the
employee's experience into "Junior", "Mid", and "Senior":
❑ "Junior" if the years_of_experience is less than 3
❑ "Mid" if the years_of_experience is between 3 and 7 (inclusive)
❑ "Senior" if the years_of_experience is greater than 7'''
from pyspark.sql.functions import *
df=df.withColumn("experience_level",when(df.experience<3,"Junior").when(df.experience.between(3,7),"Mid").otherwise("Senior"))
df.show()
```

emp_id	emp_name	department	salary	experience	experience_level
111	Abhishek	IT	30000	3	Mid
121	Vinay	Research	45000	2	Junior
131	Bhuvika	HR	46000	5	Mid

b. Add a column bonus\_eligible: Create a new column that indicates whether an employee is eligible for a bonus. An employee is eligible for a bonus if their salary is greater than 70,000 or if they are in the "Senior" experience level.

```
'''b Add a column bonus_eligible: Create a new column that indicates whether
an employee is eligible for a bonus. An employee is eligible for a bonus if their
salary is greater than 70,000 or if they are in the "Senior" experience level.'''
df=df.withColumn("bonus_eligible",when(df.salary>70000,"Yes").when(df.experience_level=="Senior","Yes").otherwise("No"))
df.show()
```

```
+-----+-----+-----+-----+-----+-----+
|emp_id|emp_name|department|salary|experience|experience_level|bonus_eligible|
+-----+-----+-----+-----+-----+-----+
| 111|Abhishek|IT|30000|3|Mid|No|
| 121|Vinay|Research|45000|2|Junior|No|
| 131|Bhuvika|HR|46000|5|Mid|No|
+-----+-----+-----+-----+-----+-----+
```

1.2.3. You are given a DataFrame containing information about students in a university. The DataFrame has the following schema:

- ☐ student\_id (String): Unique identifier for the student
- ☐ first\_name (String): First name of the student
- ☐ last\_name (String): Last name of the student
- ☐ math\_score (Integer): Score in Math subject
- ☐ science\_score (Integer): Score in Science subject
- ☐ english\_score (Integer): Score in English subject

Queries:

a. Add a column full\_name: Combine first\_name and last\_name into a new column named full\_name.

```
'''a . Add a column full_name: Combine first_name and last_name into a new
column named full_name.'''
from pyspark .sql.functions import *
df=df.withColumn("full_name",concat(df.fname,df.lname))
df.show()
```

```
+-----+-----+-----+-----+-----+-----+
|s_id| fname|lname|math_score|science_score|english_score| full_name|
+-----+-----+-----+-----+-----+-----+
| 111|Abhishek|Kumar|90|70|82|AbhishekKumar|
| 121|Vinay|Gowda|40|74|80|VinayGowda|
| 131|Bhuvika|Shetty|79|80|92|BhuvikaShetty|
+-----+-----+-----+-----+-----+-----+
```

b. Add a column average\_score: Calculate the average score of the three subjects (math\_score, science\_score, and english\_score) and add it as a new column named average\_score.

```
''' b. Add a column average_score: Calculate the average score of the three
subjects (math_score, science_score, and english_score) and add it as a new
column named average_score.'''
from pyspark.sql.functions import *
df=df.withColumn("average_score",((df.math_score+df.science_score+df.english_score)/3))
df.show()
```

s_id	fname	lname	math_score	science_score	english_score	full_name	average_score
111	Abhishek	Kumar	90	70	82	AbhishekKumar	80.66666666666667
121	Vinay	Gowda	40	74	80	VinayGowda	64.66666666666667
131	Bhuvika	Shetty	79	80	92	BhuvikaShetty	83.66666666666667

c. Add a column status: Create a new column that indicates whether the student has passed or failed. A student is considered to have passed if their average\_score is greater than or equal to 50.

```
'''c. Add a column status: Create a new column that indicates whether the
student has passed or failed. A student is considered to have passed if their
average_score is greater than or equal to 50.'''
from pyspark.sql.functions import *
df=df.withColumn("status",when(df.average_score>=50,"Passed").otherwise("Failed"))
df.show()
```

s_id	fname	lname	math_score	science_score	english_score	full_name	average_score	status
111	Abhishek	Kumar	90	70	82	AbhishekKumar	80.66666666666667	Passed
121	Vinay	Gowda	40	74	80	VinayGowda	64.66666666666667	Passed
131	Bhuvika	Shetty	79	80	92	BhuvikaShetty	83.66666666666667	Passed

1.2.4. You are given a DataFrame containing sales data for a retail store. The DataFrame has the following schema:

- ☐ transaction\_id (String): Unique identifier for the transaction
- ☐ customer\_id (String): Unique identifier for the customer
- ☐ transaction\_date (Date): The date when the transaction occurred
- ☐ total\_amount (Float): The total amount of the transaction
- ☐ product\_category\_1 (String): Category of the first product purchased
- ☐ product\_category\_2 (String): Category of the second product purchased
- ☐ product\_category\_3 (String): Category of the third product purchased

Queries:

a. Drop columns product\_category\_2 and product\_category\_3 if they have fewer than 100 transactions in total. These columns are considered to have low



sales.

```
'''a . Drop columns product_category_2 and product_category_3 if they have fewer
than 100 transactions in total. These columns are considered to have low
sales'''
# Count non-null values for pc2 and pc3
count_pc2 = df.filter(col("pc2").isNotNull()).count()
count_pc3 = df.filter(col("pc3").isNotNull()).count()

# Drop columns if they have fewer than 100 transactions
if count_pc2 < 100:
    df = df.drop("pc2")
if count_pc3 < 100:
    df = df.drop("pc3")

# Show the resulting DataFrame
df.show()
```

```
⇒ +-----+-----+-----+-----+-----+
|Trans_id|c_id|trans_date| amount|   pc1|
+-----+-----+-----+-----+-----+
|    111|  10|04-05-2023|40000.0|  home|
|    121|  11|20-05-2024|   70.0|fashion|
|    131|  12|12-05-2024|12000.0|  NULL|
+-----+-----+-----+-----+-----+
```

b. Drop columns starting with the prefix "product\_category\_" but not including the column product\_category\_1. These columns are no longer required for analysis.

```
'''b Drop columns starting with the prefix "product_category_" but not including
the column product_category_1. These columns are no longer required for
analysis.'''
df=df.drop("pc2","pc3")
df.show()
```

```
⇒ +-----+-----+-----+-----+-----+
|Trans_id|c_id|trans_date| amount|   pc1|
+-----+-----+-----+-----+-----+
|    111|  10|04-05-2023|40000.0|  home|
|    121|  11|20-05-2024|   70.0|fashion|
|    131|  12|12-05-2024|12000.0|  NULL|
+-----+-----+-----+-----+-----+
```

1.2.5. Write a pyspark SQL query to find the top three salaries in each department from the Employee table. If a department has less than three employees, include the highest salaries for that department. The output should contain the following columns:


- Department (String): The department name.

- Employee (String): The employee name.
- Salary (Integer): The employee salary.

Employee Table Schema:


- Id (Integer): Unique identifier for the employee.
- Name (String): The name of the employee.
- Salary (Integer): The salary of the employee.

```
data = ([[111, "Abhishek", 30000, 3],
         [121, "Vinay", 45000, 2],
         [131, "Bhuvika", 56000, 2]])
df = spark.createDataFrame(data, ["emp_id", "emp_name", "salary", "dept_id"])
df.show()
```



emp_id	emp_name	salary	dept_id
111	Abhishek	30000	3
121	Vinay	45000	2
131	Bhuvika	56000	2

```
df.createOrReplaceTempView("employee")
spark.sql('''SELECT dept_id AS Department, emp_name AS Employee, salary AS Salary
FROM (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS row_num
    FROM Employee
)
WHERE row_num <= 3''').show()
```



Department	Employee	Salary
2	Bhuvika	56000
2	Vinay	45000
3	Abhishek	30000

1.2.6. DepartmentId (Integer): The department ID where the employee works.

Write a pyspark SQL query to find all employees who earn more than their managers. The output should contain the following columns:

- ☐ Employee (String): The employee name.
- ☐ Manager (String): The manager name.


Employee Table Schema:

- ☐ Id (Integer): Unique identifier for the employee.
- ☐ Name (String): The name of the employee.
- ☐ Salary (Integer): The salary of the employee.
- ☐ ManagerId (Integer): The ID of the employee's manager. NULL if the employee has no manager.

## QUERIES

### a. Query to Find All Employees and Their Managers

```
data = ([111,"Abhishek", "Sonu", 30000, 3],
        [121, "Vinay","Riya", 45000, 2],
        [131, "Bhuvika",None ,56000, None])
df = spark.createDataFrame(data,["emp_id","emp_name","manager_name","salary","manager_id"])
df.show()
```




emp_id	emp_name	manager_name	salary	manager_id
111	Abhishek	Sonu	30000	3
121	Vinay	Riya	45000	2
131	Bhuvika	NULL	56000	NULL

## b. Query to Find Employees Earning More Than Their Managers

```
#query
'''Write a pyspark SQL query to find all employees who earn more than their
managers. The output should contain the following columns:
❑ Employee (String): The employee name.
❑ Manager (String): The manager name.'''


df.createOrReplaceTempView("employee")
spark.sql('''SELECT e1.emp_name AS Employee, e2.emp_name AS Manager
FROM employee e1
JOIN employee e2 ON e1.manager_id = e2.emp_id
WHERE e1.salary > e2.salary''').show()
```



Employee	Manager
----------	---------

## c. Query to Find Employees Without Managers

```
#a)Query to Find All Employees and Their Managers
df.createOrReplaceTempView("employee")
spark.sql('''SELECT e1.emp_name AS Employee, e2.emp_name AS Manager
FROM employee e1
LEFT JOIN employee e2 ON e1.manager_id = e2.emp_id''').show()
```



Employee	Manager
Abhishek	NULL
Bhuvika	NULL
Vinay	NULL

## d. Query to Find the Highest Paid Employee

```
#b)Query to Find Employees Earning More Than Their Managers
spark.sql('''SELECT e1.emp_name AS Employee, e2.emp_name AS Manager
FROM employee e1
JOIN employee e2 ON e1.manager_id = e2.emp_id
WHERE e1.salary > e2.salary''').show()
```

```
⇒ +-----+-----+
   |Employee|Manager|
   +-----+-----+
```

- e. Query to Calculate the Average Salary of Employees

```
#c) Query to Find Employees without Managers
spark.sql('''SELECT emp_name AS Employee
FROM employee
WHERE manager_id IS NULL''').show()
```

```
⇒ +-----+
   |Employee|
   +-----+
   | Bhuvika|
   +-----+
```

- f. Query to Count the Number of Employees Managed by Each Manager

```
#d) Query to Find the Highest Paid Employee
spark.sql('''SELECT emp_name AS Employee
FROM employee
ORDER BY salary DESC
LIMIT 1''').show()
```

```
⇒ +-----+
   |Employee|
   +-----+
   | Bhuvika|
   +-----+
```

## g. Query to List All Managers

```
#e) Query to Calculate the Average Salary of Employees
spark.sql('''SELECT AVG(salary) AS Average_Salary
FROM employee''').show()
```

```
⇒ +-----+
| Average_Salary |
+-----+
| 43666.66666666664 |
+-----+
```

## h. Query to List All Employees Grouped by Their Managers

```
#g) Query to Find the Total Salary Paid to Employees Managed by Each Manager

spark.sql('''SELECT manager_name AS Manager, SUM(salary) AS Total_Salary
FROM Employee
GROUP BY manager_name''').show()
```

```
⇒ +-----+-----+
| Manager | Total_Salary |
+-----+-----+
| Sonu    | 30000        |
| NULL    | 56000        |
| Riya    | 45000        |
+-----+-----+
```

1.2.7. You have been given a dataset containing sales information, and you need to perform several sorting operations to analyze the data effectively. The dataset contains the following columns:

- a. TransactionID: Unique identifier for each transaction.
- b. CustomerID: Unique identifier for each customer.
- c. Product: The name of the product sold.
- d. Quantity: The number of units sold.
- e. Price: The price per unit of the product.
- f. TransactionDate: The date when the transaction occurred.

Queries:

- a. Load the Data: Create a PySpark DataFrame from the dataset

```
data = ([[111,10,"TV",4,'04-05-2023',40000.0],
        [121,11,"Phone",1,'20-05-2024',70000.0],
        [131,12,"Headphone",10,'12-05-2024',12000.0]])
df = spark.createDataFrame(data,["Trans_id","c_id","product","quantity","trans_date","price"])
df.show()
```

```

+-----+-----+-----+-----+-----+-----+
|Trans_id|c_id| product|quantity|trans_date| price|
+-----+-----+-----+-----+-----+
|    111|  10|      TV|      4|04-05-2023|40000.0|
|    121|  11|    Phone|      1|20-05-2024|70000.0|
|    131|  12|Headphone|     10|12-05-2024|12000.0|
+-----+-----+-----+-----+-----+

```

- b. Sort by Transaction Date: Sort the DataFrame by TransactionDate in ascending order.

```
#a)Sort by Transaction Date: Sort the DataFrame by TransactionDate in ascending order.
df.orderBy("trans_date").show()
```

```

+-----+-----+-----+-----+-----+-----+
|Trans_id|c_id| product|quantity|trans_date| price|
+-----+-----+-----+-----+-----+
|    111|  10|      TV|      4|04-05-2023|40000.0|
|    131|  12|Headphone|     10|12-05-2024|12000.0|
|    121|  11|    Phone|      1|20-05-2024|70000.0|
+-----+-----+-----+-----+-----+

```

- c. Sort by Price: Sort the DataFrame by Price in descending order.

Sort by CustomerID and Transaction Date: Sort the DataFrame first by CustomerID in ascending order and then by TransactionDate in descending order.

#b) Sort by Price: Sort the DataFrame by Price in descending order.  
`df.orderBy(df.price.desc()).show()`

```

⇒ +-----+-----+-----+-----+-----+
   |Trans_id|c_id|  product|quantity|trans_date|  price|
   +-----+-----+-----+-----+-----+
   |      121|  11|    Phone|      1|20-05-2024|70000.0|
   |      111|  10|      TV|      4|04-05-2023|40000.0|
   |      131|  12|Headphone|     10|12-05-2024|12000.0|
   +-----+-----+-----+-----+-----+

```

- d. Top Transactions: Find the top 5 transactions with the highest total amount spent (Quantity \* Price).

'''c) Sort by CustomerID and Transaction Date: Sort the DataFrame first by CustomerID in ascending order and then by TransactionDate in descending order'''  
`df.orderBy("c_id","trans_date").show()`

```

⇒ +-----+-----+-----+-----+-----+
   |Trans_id|c_id|  product|quantity|trans_date|  price|
   +-----+-----+-----+-----+-----+
   |      111|  10|      TV|      4|04-05-2023|40000.0|
   |      121|  11|    Phone|      1|20-05-2024|70000.0|
   |      131|  12|Headphone|     10|12-05-2024|12000.0|
   +-----+-----+-----+-----+-----+

```



## CHAPTER- 2

# MAP REDUCE

### 2.1 Introduction

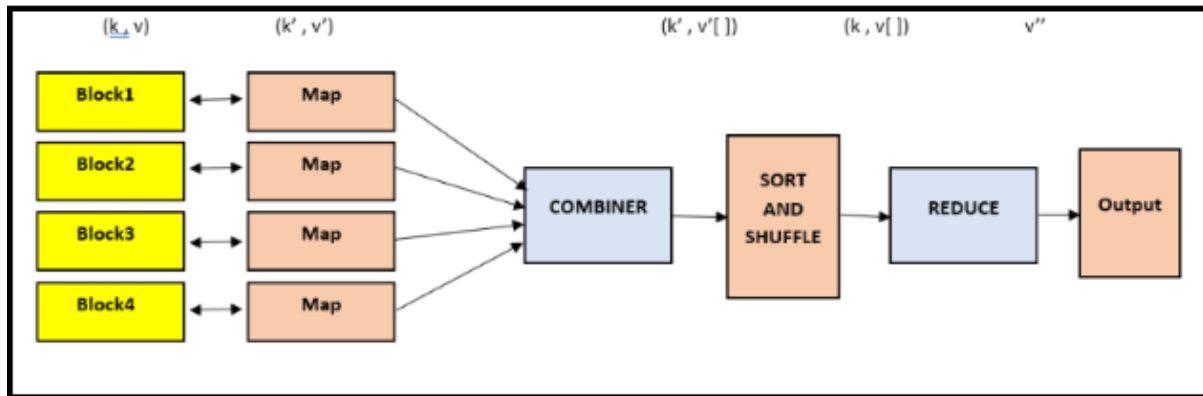
- It is a framework in which we can write applications to run huge amount of data in parallel and in large cluster of commodity hardware in a reliable manner.
- Different Phases of Map-Reduce:- Map-Reduce model has three major and one optional phase.
  1. Mapping
  2. Shuffling and Sorting
  3. Reducing
  4. Combining

**Mapping :-** It is the first phase of MapReduce programming. Mapping Phase accepts key-value pairs as input as  $(k, v)$ , where the key represents the Key address of each record and the value represents the entire record content. The output of the Mapping phase will also be in the key-value format  $(k', v')$ .

**Shuffling and Sorting :-** The output of various mapping parts  $(k', v')$ , then goes into Shuffling and Sorting phase. All the same values are deleted, and different values are grouped together based on same keys. The output of the Shuffling and Sorting phase will be key-value pairs again as key and array of values  $(k, v[ ])$ .

**Reducer :-** The output of the Shuffling and Sorting phase  $(k, v[ ])$  will be the input of the Reducer phase. In this phase reducer function's logic is executed and all the values are Collected against their corresponding keys. Reducer stabilize outputs of various mappers and computes the final output.

**Combining :-** It is an optional phase in the MapReduce phases . The combiner phase is used to optimize the performance of MapReduce phases. This phase makes the Shuffling and Sorting phase work even quicker by enabling additional performance features in MapReduce phases.



## 2.2. Programs

2.2.1. Develop a pyspark program for word-count in Map Reduce.

Let's consider sample input text file contains:

```
Hello world
Hello Spark
Apache Spark is awesome
Word Count example with Spark
```

PROGRAM:

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder.appName("WordCount").getOrCreate()

# Sample input text
text = [
    "Hello world",
    "Hello Spark",
    "Apache Spark is awesome",
    "Word Count example with Spark"
]

# Create RDD from text
rdd = spark.sparkContext.parallelize(text)

# Map phase: Split each line into words and assign count of 1 to
each word
word_counts = rdd.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1))

# Reduce phase: Aggregate the word counts
word_counts = word_counts.reduceByKey(lambda a, b: a + b)

# Display word counts
for word, count in word_counts.collect():
    print(f"{word}: {count}")

# Stop SparkSession
spark.stop()
```

Output:

```
Hello: 2
world: 1
Spark: 3
Apache: 1
is: 1
awesome: 1
Word: 1
Count: 1
example: 1
with: 1
```



### 2.2.2. Implementing Matrix Multiplication with Hadoop Map Reduce

Input text1:

```
0 0 1
0 1 2
1 0 3
1 1 4
```



Input text2:

```
0 0 5
0 1 6
1 0 7
1 1 8
```



OUTPUT:

```
(0,0) 19
(0,1) 22
(1,0) 43
(1,1) 50
```



Program:

```

from pyspark import SparkConf, SparkContext
def parse_matrix(line):
    return list(map(int, line.split()))
def mapper_matrix_A(line):
    elements = parse_matrix(line)
    row = elements[0]
    col = elements[1]
    value = elements[2]
    return [(col, (row, value))]
def mapper_matrix_B(line):
    elements = parse_matrix(line)
    row = elements[0]
    col = elements[1]
    value = elements[2]
    return [(row, (col, value))]
def reducer(key, values):
    A_elements = {}
    B_elements = {}
    for value in values:
        if isinstance(value[1], tuple): # From matrix A
            row, a_val = value[1]
            A_elements[row] = a_val
        else: # From matrix B
            col, b_val = value
            B_elements[col] = b_val
    results = []
    for i in A_elements:
        for j in B_elements:
            results.append(((i, j), A_elements[i] * B_elements[j]))
    return results
def matrix_multiplication(sc, matrix_a_path, matrix_b_path, output_path):
    matrix_A = sc.textFile(matrix_a_path)
    matrix_B = sc.textFile(matrix_b_path)
    mapped_A = matrix_A.flatMap(mapper_matrix_A)
    mapped_B = matrix_B.flatMap(mapper_matrix_B)
    combined = mapped_A.union(mapped_B)
    reduced = combined.groupByKey().flatMap(lambda x: reducer(x[0], x[1]))
    final_result = reduced.reduceByKey(lambda x, y: x + y)
    final_result.saveAsTextFile(output_path)
def main():
    conf = SparkConf().setAppName("MatrixMultiplication")
    sc = SparkContext(conf=conf)

    matrix_a_path = "path/to/matrix_a.txt"
    matrix_b_path = "path/to/matrix_b.txt"
    output_path = "path/to/output"

    matrix_multiplication(sc, matrix_a_path, matrix_b_path, output_path)

    sc.stop()
if __name__ == "__main__":
    main()

```



### 2.2.3. Implement Searching with Hadoop Map Reduce

Input file:

```
Hello world
Welcome to the world of Spark
Apache Spark is a powerful big data processing framework
Search for keywords in this text file
Spark makes big data processing easy
```

PROGRAM:

```
from pyspark import SparkConf, SparkContext
def search_keyword_in_line(keyword, line):
    if keyword in line[1]:
        return [(line[0], line[1])]
    else:
        return []
def main():
    conf = SparkConf().setAppName("KeywordSearch")
    sc = SparkContext(conf=conf)
    input_path = "path/to/your/textfile.txt"
    keyword = "Spark"

    text_file = sc.textFile(input_path)

    # Zip the lines with their line numbers
    lines_with_index = text_file.zipWithIndex().map(lambda x:
(x[1] + 1, x[0]))

    # Search for the keyword in each line
    results = lines_with_index.flatMap(lambda line:
search_keyword_in_line(keyword, line))

    # Collect and print the results
    output = results.collect()
    for (line_num, line) in output:
        print(f"Line {line_num}: {line}")

    # Stop the SparkContext
    sc.stop()
```



Output:

```
Line 2: Welcome to the world of Spark  
Line 3: Apache Spark is a powerful big data  
processing framework  
Line 5: Spark makes big data processing easy
```

## CHAPTER-3

# MongoDB

### 3.1 Introduction

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term 'NoSQL' means 'non-relational'. It means that MongoDB isn't based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON ( similar to JSON format).

SQL databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today's real-world highly growing applications. Modern applications are more networked, social and interactive than ever. Applications are storing more and more data and are accessing it at higher rates.

Relational Database Management System(RDBMS) is not the correct choice when it comes to handling big data by the virtue of their design since they are not horizontally scalable. If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

#### **RDBMS vs MongoDB:**

RDBMS has a typical schema design that shows number of tables and the relationship between these tables whereas MongoDB is document-oriented. There is no concept of schema or relationship.

Complex transactions are not supported in MongoDB because complex join operations are not available.

MongoDB allows a highly flexible and scalable document structure. For example, one data document of a collection in MongoDB can have two fields whereas the other document in the same collection can have four.

MongoDB is faster as compared to RDBMS due to efficient indexing and storage techniques.

There are a few terms that are related in both databases. What's called Table in RDBMS is called a Collection in MongoDB. Similarly, a Row is called a Document and a Column is called

a Field. MongoDB provides a default ‘\_id’ (if not provided explicitly) which is a 12-byte hexadecimal number that assures the uniqueness of every document. It is similar to the Primary key in RDBMS.

### **Features of MongoDB:**

**Document Oriented:** MongoDB stores the main subject in the minimal number of documents and not by breaking it up into multiple relational structures like RDBMS. For example, it stores all the information of a computer in a single document called Computer and not in distinct relational structures like CPU, RAM, Hard disk, etc.

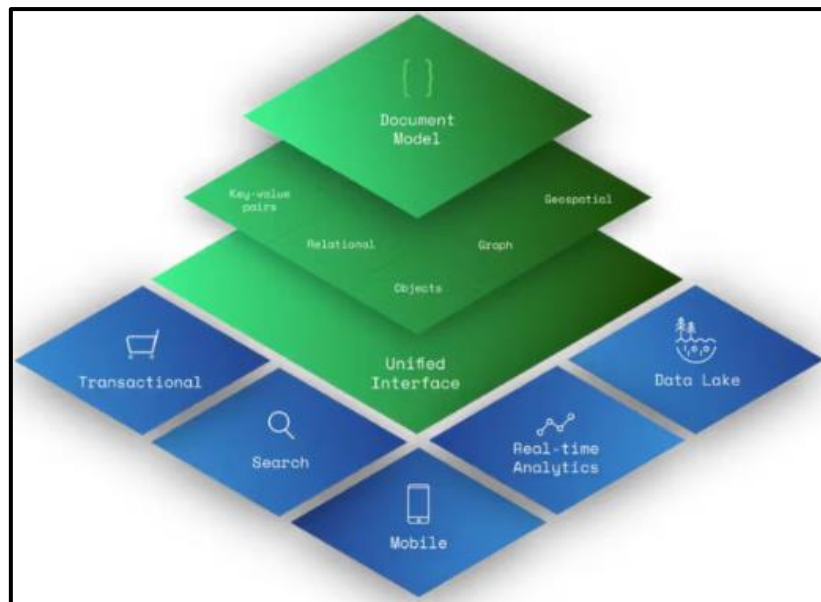
**Indexing:** Without indexing, a database would have to scan every document of a collection to select those that match the query which would be inefficient. So, for efficient searching Indexing is a must and MongoDB uses it to process huge volumes of data in very less time.

**Scalability:** MongoDB scales horizontally using sharding (partitioning data across various servers). Data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. Also, new machines can be added to a running database.

**Replication and High Availability:** MongoDB increases the data availability with multiple copies of data on different servers. By providing redundancy, it protects the database from hardware failures. If one server goes down, the data can be retrieved easily from other active servers which also had the data stored on them.

**Aggregation:** Aggregation operations process data records and return the computed results. It is similar to the GROUPBY clause in SQL. A few aggregation expressions are sum, avg, min, max, etc.





## 3.2 Programs:

3.2.1. You have a collection named students in your MongoDB database. Insert a new student record with the following details

```
mongosh
Current Mongosh Log ID: 669896a985875ba529f00e8f
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.2.6
Using MongoDB:      7.0.11
Using Mongosh:       2.2.6
mongosh 2.2.10 is available for download: https://www.mongodb.com/try/download/shell
For mongosh info see: https://docs.mongodb.com/mongosh-shell/

-----
The server generated these startup warnings when booting
2024-07-17T18:35:33.964+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

test> 
```

```
test> use student_db
switched to db student_db
student_db> db.createCollection("students")
{ ok: 1 }
```

- a. name: "Alice",age: 23,major: "Computer Science"

```
student_db> db.students.insertOne({name:"Alice",age:23,major:"Computer Science"});
{
  acknowledged: true,
  insertedId: ObjectId('669897d3f7612eea34d6c561')
}
```

- b. If a student with \_id: 1 exists, update their details to name: "Bob",age: 24,major: "Mathematics". If no student with \_id: 1 exists, insert this new record.

```
student_db> db.students.updateOne({_id:1},{set:{name:"Bob",age:24,major:"Mathematics"}},{upsert:true});
{
  acknowledged: true,
  insertedId: 1,
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 1
}
```

- c. In the students collection, update the major of the student named "Alice" to "Data Science".

```
student_db> db.students.updateOne({name:"Alice"},{$set : {major:"Data Science"}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

- c. Remove the student record from the students collection where the name is "Bob".

```
student_db> db.students.deleteOne({name:"Bob"});
{ acknowledged: true, deletedCount: 1 }
```

- e. Retrieve all student records from the students collection who are majoring in "Computer Science"

```
student_db> db.students.find({major:"Computer Science"});
```

```
student_db> db.students.find();
[
  {
    _id: ObjectId('669897d3f7612eea34d6c561'),
    name: 'Alice',
    age: 23,
    major: 'Data Science'
  }
]
```

### 3.2.2 Explain the MongoDB aggregation and pipeline with an example

Answer:

**Pipelines:** In MongoDB, a pipeline is a sequence of stages through which documents pass, transforming them step by step. Each stage performs a specific operation (like filtering, grouping, or sorting), allowing for complex data aggregation and transformation processes.

Let's start with our initial dataset:

_id	name	major	gpa	courses
1	Alice	CS	3.9	["Databases", "AI", "Networks"]
2	Bob	Mathematics	3.2	["Calculus", "Statistics"]
3	Carol	CS	3.7	["Databases", "AI"]
4	Dave	Physics	3.5	["Quantum Mechanics", "AI"]
5	Eve	CS	3.8	["Databases", "AI"]

The pipeline:

## Comprehensive Pipelines with Input and Output

### Pipeline 1: Filtering, Unwinding, Grouping, Sorting, and Projecting

```
javascript Copy code
db.students.aggregate([
  { $match: { gpa: { $gte: 3.6 } } },
  { $unwind: "$courses" },
  { $group: { _id: "$courses", averageGPA: { $avg: "$gpa" } } },
  { $sort: { averageGPA: -1 } },
  { $project: { course: "$_id", averageGPA: 1, _id: 0 } }
])
```

Some of the aggregate functions include:

#### i. **\$match**

- **Function:** Filters documents to pass only those that match the specified condition(s).
- **Description:** Acts as a query in the pipeline to narrow down results based on specified criteria, similar to a SQL WHERE clause.

- Filters students with GPA  $\geq 3.6$ .

_id	name	major	gpa	courses
1	Alice	CS	3.9	["Databases", "AI", "Networks"]
3	Carol	CS	3.7	["Databases", "AI"]
5	Eve	CS	3.8	["Databases", "AI"]

#### ii. **\$unwind**

- **Function:** Deconstructs an array field from the input documents to output a document for each element.
- **Description:** Converts an array field in a document into multiple documents, each containing a single element from the array, useful for normalizing data.

### iii. \$group

- **Function:** Groups input documents by a specified identifier expression and applies the accumulator expressions.
- **Description:** Aggregates data based on specified keys and applies accumulator operations like \$sum, \$avg, etc., similar to SQL's GROUP BY.

• Groups by course and calculates the average GPA for each course.

_id	averageGPA
Databases	3.8
AI	3.8
Networks	3.9

- **\$sort**

- **Function:** Sorts all input documents and returns them in order.
- **Description:** Orders the documents based on specified field(s) in ascending or descending order, akin to SQL's ORDER BY.

• Sorts courses by their average GPA in descending order.

_id	averageGPA
Networks	3.9
Databases	3.8
AI	3.8

- **\$project**

- **Function:** Reshapes each document in the stream, such as by including, excluding, or adding new fields.
- **Description:** Specifies the fields to include or exclude in the output documents, similar to SQL's SELECT clause.

- Renames "\_id" to "course" and keeps "averageGPA", excluding "\_id".

course	averageGPA
Networks	3.9
Databases	3.8
AI	3.8

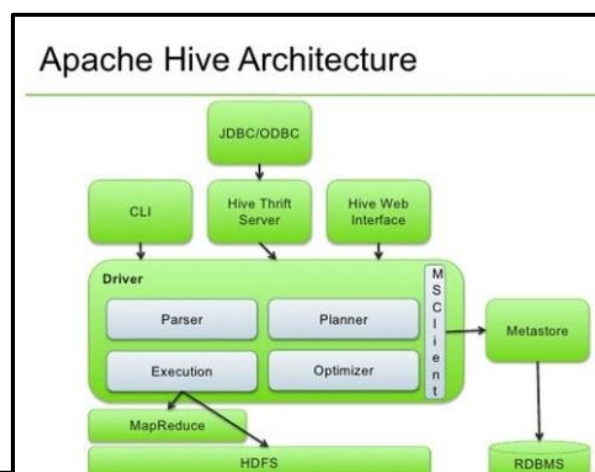
## CHAPTER-4

# APACHE HIVE

### 4.1 Introduction

Apache Hive is a data warehouse and an ETL tool which provides an SQL-like interface between the user and the Hadoop distributed file system (HDFS) which integrates Hadoop. It is built on top of Hadoop. It is a software project that provides data query and analysis. It facilitates reading, writing and handling wide datasets that stored in distributed storage and queried by Structure Query Language (SQL) syntax. It is not built for Online Transactional Processing (OLTP) workloads. It is frequently used for data warehousing tasks like data encapsulation, Ad-hoc Queries, and analysis of huge datasets. It is designed to enhance scalability, extensibility, performance, fault-tolerance and loose-coupling with its input formats.

Initially Hive is developed by Facebook and Amazon, Netflix and It delivers standard SQL functionality for analytics. Traditional SQL queries are written in the MapReduce Java API to execute SQL Application and SQL queries over distributed data. Hive provides portability as most data warehousing applications functions with SQL-based query languages like NoSQL. Apache Hive is a data warehouse software project that is built on top of the Hadoop ecosystem. It provides an SQL-like interface to query and analyze large datasets stored in Hadoop's distributed file system (HDFS) or other compatible storage systems. Hive uses a language called HiveQL, which is similar to SQL, to allow users to express data queries, transformations, and analyses in a familiar syntax. HiveQL statements are compiled into MapReduce jobs, which are then executed on the Hadoop cluster to process the data.



## 4.2 Programs

4.2.1 Create a Hive table named employees with the following columns:

- employee\_id (int)
- first\_name (string)
- last\_name (string)
- email (string)
- salary (double)

Perform following operations:

1. Drop the table named employees if it exists in the Hive database.
  2. Remove all data from the employees table but keep the table structure
  3. Add a new column department (string) to the employees table. Rename the salary column to monthly\_salary.
  4. Display all the tables available in the current Hive database.
  5. Display the structure of the employees table.
- 1) Drop the table named employees if it exists in the Hive database:  
DROP TABLE IF EXISTS employees;

OUTPUT

Table employees dropped if it existed.

2) Create the employees table:

```
CREATE TABLE employees (  
    employee_id INT,  
    first_name STRING,  
    last_name STRING,  
    email STRING,  
    salary DOUBLE  
);
```



OUTPUT

```
Table employees created.
```

3) Remove all data from the employees table but keep the table structure:

```
TRUNCATE TABLE employees;
```

OUTPUT

```
All data from the employees table removed, structure retained.
```

4) Add a new column department (string) to the employees table and rename the salary column to monthly\_salary:

```
ALTER TABLE employees ADD COLUMNS (department STRING);
```

```
ALTER TABLE employees CHANGE salary monthly_salary DOUBLE;
```

OUTPUT

```
Column department added to employees table.  
Column salary renamed to monthly_salary in employees table.
```

5) Display all the tables available in the current Hive database:

```
SHOW TABLES;
```

## OUTPUT

+-----+	
tab_name	
+-----+	
employees	
+-----+	

6) Display the structure of the employees table:

DESCRIBE employees;

+-----+-----+			
col_name	data_type	comment	
+-----+-----+			
employee_id	int		
first_name	string		
last_name	string		
email	string		
monthly_salary	double		
department	string		
+-----+-----+			

---

## CHAPTER-5

### PIG LATIN

#### 5.1. Introduction

The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. It is a textual language that abstracts the programming from the Java MapReduce idiom into a notation. The Pig Latin statements are used to process the data. It is an operator that accepts a relation as an input and generates another relation as an output.

1. It can span multiple lines.
2. Each statement must end with a semi-colon.
3. It may include expression and schemas.
4. By default, these statements are processed using multi-query execution

CONVENTION	DESCRIPTION
( )	The parenthesis can enclose one or more items. It can also be used to indicate the tuple data type. Example - (10, xyz, (3,6,9))
[ ]	The straight brackets can enclose one or more items. It can also be used to indicate the map data type. Example - [INNER   OUTER]
{ }	The curly brackets enclose two or more items. It can also be used to indicate the bag data type Example - { block   nested_block }
...	The horizontal ellipsis points indicate that you can repeat a portion of the code. Example - cat path [path ...]

## 5.2 PROGRAMS

5.2.1. Write a Pig Latin script to find the word count in a given text file.

```
Text = LOAD 'input_text.txt' USING TextLoader AS (line:chararray);
```

```
words = FOREACH text GENERATE FLATTEN(TOKENIZE(line)) AS word;
```

```
grouped_words = GROUP words BY word;
```

```
word_count = FOREACH grouped_words GENERATE group AS word,  
COUNT(words) AS count;
```

```
DUMP word_count;
```

OUTPUT:

Given the content of textfile.txt, the output of the DUMP word\_count; command should look something like this:

```
(Pig,2)  
(great,1)  
(Hello,2)  
(is,1)  
(world,1)
```

5.2.2 Develop a Pig Latin script that processes a dataset containing daily temperature records and finds the maximum temperature recorded for each year.

Example Input:	Output:
2020-01-01 5	2020 8
2020-01-02 6	2021 10
2020-01-03 -2	
2020-02-01 8	
2021-01-01 3	
2021-02-02 10	
2021-03-01 7	

Loading Data:

---

```
records = LOAD 'temperature_data.txt' USING PigStorage(',') AS
```

```
(date:chararray, temperature:int);
```

OUTPUT

```
(2020-01-01, 5)
(2020-01-02, 6)
(2020-01-03, -2)
(2020-02-01, 8)
(2021-01-01, 3)
(2021-02-02, 10)
(2021-03-01, 7)
```

Grouping Records by Year (records\_grouped):

```
records_grouped = GROUP records BY SUBSTRING(date, 0, 4);
```

OUTPUT

```
(2020, {(2020-01-01, 5), (2020-01-02, 6), (2020-01-03, -2),
        (2020-02-01, 8)})
(2021, {(2021-01-01, 3), (2021-02-02, 10), (2021-03-01, 7)})
```

Calculating Maximum Temperature for Each Year (max\_temp):

```
max_temp = FOREACH records_grouped GENERATE group AS
```

```
year, MAX(records.temperature) AS max_temp;
```

OUTPUT

```
(2020, 8)
(2021, 10)
```