**1. Import necessary libraries:** We first import the necessary libraries for this exercise. **pandas** is a library providing high-performance, easy-to-use data structures and data analysis tools. **numpy** is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The **train\_test\_split** function from **sklearn.model\_selection** is used to split the dataset into training and test sets.

1. # Importing the necessary libraries
2. import numpy as np
3. import pandas as pd
4. fromsklearn.model\_selection import train\_test\_split

**2. Load the dataset:** We use the **read\_csv** function from **pandas** to load the Iris dataset from a CSV file. The loaded dataset is a DataFrame object which is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).

1. # Loading the Iris dataset dataset = pd.read\_csv('iris.csv')

**3. Create the matrix of features and dependent variable vector:** We extract the matrix of features (X) and dependent variable vector (y) from the DataFrame. The **iloc** indexer is used to select all rows and all columns except the last one for X, and all rows of the last column for y. The **.values** attribute is used to extract the data as numpy arrays.

1. # Creating the matrix of features (X) and the dependent variable vector (y) X = dataset.iloc[:, :-1].values y = dataset.iloc[:, -1].values

**4. Print X and y:** Finally, we print out the matrix of features and dependent variable vector to verify their creation.

1. # Printing the matrix of features and the dependent variable vector
2. print(X)
3. print(y)

**1. Import the necessary libraries:** We start by importing the necessary libraries for this exercise. Pandas is a library providing high-performance, easy-to-use data structures and data analysis tools. NumPy is a library used for working with arrays. **SimpleImputer** is a class from the **sklearn.impute** module that provides basic strategies for imputing missing values.

1. # Importing the necessary libraries
2. import pandas as pd
3. import numpy as np
4. from sklearn.impute import SimpleImputer

**2. Load the dataset:**The dataset is loaded into a pandas DataFrame using the read\_csv function. This function is widely used in pandas to read a comma-separated values (csv) file into DataFrame.

1. # Load the dataset
2. df = pd.read\_csv('pima-indians-diabetes.csv')

**3. Identify missing data:** We identify missing data in the DataFrame using the isnull function followed by the sum function. This gives us the number of missing entries in each column. These missing entries are represented as NaN.

1. # Identify missing data (assumes that missing data is represented as NaN)
2. missing\_data = df.isnull().sum()

**4. Print the number of missing entries in each column**

1. # Print the number of missing entries in each column
2. print("Missing data: \n", missing\_data)

**5. Configure an instance of the SimpleImputer class:** We create an instance of the **SimpleImputer** class. This class is a part of the **sklearn.impute** module and provides basic strategies for imputing missing values. We configure it to replace missing values (represented as **np.nan**) with the **mean** value of the column.

1. # Configure an instance of the SimpleImputer class
2. imputer = SimpleImputer(missing\_values=np.nan, strategy='mean')

**6. Fit the imputer on the DataFrame:** We fit the imputer on the DataFrame using the fit method. This method calculates the imputation values (in this case, the mean of each column) that will be used to replace the missing data.

1. # Fit the imputer on the DataFrame
2. imputer.fit(df)

**7. Apply the transform to the DataFrame:** We apply the transform to the DataFrame using the transform method. This method replaces missing data with the imputation values calculated by the fit method.

1. # Apply the transform to the DataFrame
2. df\_imputed = imputer.transform(df)

**8. Print the updated matrix of features:** Finally, we print out the updated matrix of features to verify that the missing data has been successfully replaced.

1. print("Updated matrix of features: \n", df\_imputed)

**1. Importing the necessary libraries**. Import pandas for data manipulation, numpy for numerical operations, and the necessary classes from scikit-learn for preprocessing.

1. # Importing the necessary libraries
2. import pandas as pd
3. import numpy as np
4. from sklearn.compose import ColumnTransformer
5. from sklearn.preprocessing import OneHotEncoder, LabelEncoder

**2. Load the dataset.** The Titanic dataset is loaded into a pandas DataFrame from a CSV file.

1. # Load the dataset
2. df = pd.read\_csv('titanic.csv')

**3. Identify the categorical data.** Specify which features in our dataset are categorical. In this case, **'Sex'**, **'Embarked',** and **'Pclass'** are the categorical features.

1. # Identify the categorical data
2. categorical\_features = ['Sex', 'Embarked', 'Pclass']

**4. Implement an instance of the ColumnTransformer clas.** Initialize a ColumnTransformer that will apply a OneHotEncoder to the categorical features. The remainder='passthrough' argument ensures that the non-transformed features are not discarded.

1. # Implement an instance of the ColumnTransformer class
2. ct = ColumnTransformer(
3. transformers=[
4. ('encoder', OneHotEncoder(), categorical\_features)
5. ], remainder='passthrough')

**5. Apply the fit\_transform method**. Fit the **ColumnTransformer** to our DataFrame and transform the data. This applies one-hot encoding to our categorical features, converting them into numerical data suitable for a machine-learning model.

1. # Apply the fit\_transform method on the instance of ColumnTransformer
2. X = ct.fit\_transform(df)

**6. Convert the output into a NumPy array.**Convert the output to a NumPy array: The output of the **ColumnTransformer** is a sparse matrix - convert this to a dense NumPy array for easier manipulation.

1. # Convert the output into a NumPy array
2. X = np.array(X)

**7. Use LabelEncoder to encode binary categorical data.** The 'Survived' feature is our dependent variable. Since it is a binary categorical feature, we use LabelEncoder to transform it into numerical data.

1. # Use LabelEncoder to encode binary categorical data
2. le = LabelEncoder()
3. y = le.fit\_transform(df['Survived'])

**8. Print the transformed feature matrix and dependent variable vector** to verify that our preprocessing steps have been applied correctly.

1. # Print the updated matrix of features and the dependent variable vector
2. print("Updated matrix of features: \n", X)
3. print("Updated dependent variable vector: \n", y)

**1. Import necessary libraries.**Begin by importing all the necessary libraries - **pandas** for data manipulation, **train\_test\_split** for splitting our dataset into training and test sets, and **StandardScaler** for feature scaling.

1. # Import necessary libraries
2. import pandas as pd
3. from sklearn.model\_selection import train\_test\_split
4. from sklearn.preprocessing import StandardScaler

**2. Load the Iris dataset.**Load the Iris dataset using the `pd.read\_csv` function from pandas. The dataset is read directly from a CSV file named 'iris.csv'. This file is assumed to be in the same directory as the script. The dataset is stored in a pandas DataFrame for easier manipulation

1. # Load the Iris dataset using pd.read\_csv
2. iris\_df = pd.read\_csv('iris.csv')

**3. Separate features and target.** The independent variables (features) and dependent variable (target) are separated into **X** and **y**, respectively. It is assumed that the target variable is named **'target'** in the dataset.

1. # Separate features and target
2. X = iris\_df.drop('target', axis=1) # Assuming 'target' is the column name for the target variable
3. y = iris\_df['target']

**4. Split the dataset into an 80-20 training-test set.** Use the **train\_test\_split** function to split our dataset into training and test sets. We specify a **test\_size** of 0.2, which means 80% of the data will be used for training and 20% will be used for testing.

1. # Split the dataset into an 80-20 training-test set
2. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**5. Apply feature scaling.**The **StandardScaler** is applied to standardize the features to have a **mean=0** and **variance=1**. The scaler is fitted on the training set and then used to transform both the training and test sets. This is to prevent information leak from the test set into the training set.

1. # Apply StandardScaler to scale the feature variables
2. scaler = StandardScaler()
3. X\_train = scaler.fit\_transform(X\_train)
4. X\_test = scaler.transform(X\_test)

**6. Print the scaled training and test sets.**Finally, the scaled training and test sets are printed to verify the scaling.

1. # Print scaled training and test sets
2. print("Scaled Training Set:")
3. print(X\_train)
4. print("\nScaled Test Set:")
5. print(X\_test)

**1. Import necessary libraries**: We start by importing the necessary libraries for data preprocessing. This includes **pandas** for data manipulation, **train\_test\_split** from **sklearn.model\_selection** to split our dataset into training and test sets, and **StandardScaler** from **sklearn.preprocessing** to apply feature scaling.

1. import pandas as pd
2. from sklearn.model\_selection import train\_test\_split
3. from sklearn.preprocessing import StandardScaler

**2. Load the dataset:** The Wine Quality Red dataset is loaded into a pandas DataFrame using the pd.read\_csv function. Here, we need to specify the correct delimiter, which in this case is a semicolon.

1. pythonCopy code
2. df = pd.read\_csv('winequality-red.csv', delimiter=';')

**3. Split the dataset into a training set and a test set**: We separate the target variable 'quality' from the features and then split the dataset into an 80-20 training-test set using the **train\_test\_split** function.

1. X = df.drop('quality', axis=1) y = df['quality'] X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**4. Create an instance of the StandardScaler class**: The **StandardScaler** class is used to standardize features by removing the mean and scaling to unit variance.

1. sc = StandardScaler()

**5. Fit the StandardScaler on the training set and transform the data**: The **StandardScaler** is fitted to the training set and then used to transform both the training and test datasets.

1. X\_train = sc.fit\_transform(X\_train) X\_test = sc.transform(X\_test)

**6. Print the scaled datasets**: Finally, we print the scaled training and test datasets to verify the feature scaling process.

1. print("Scaled training set:\n", X\_train) print("Scaled test set:\n", X\_test)

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

df = pd.read\_csv('winequality-red.csv', delimiter=';')

X = df.drop('quality', axis=1)

y = df['quality']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,test\_size=0.2, random\_state=42)

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

print("Scaled training set:\n", X\_train)

print("Scaled test set:\n", X\_test)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

iris\_df = pd.read\_csv('iris.csv')

X = iris\_df.drop('target', axis=1)

y = iris\_df['target']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

print("Scaled Training Set:")

print(X\_train)

print("\nScaled Test Set:")

print(X\_test)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import pandas as pd

import numpy as np

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder, LabelEncoder

df = pd.read\_csv('titanic.csv')

categorical\_features = ['Sex', 'Embarked', 'Pclass']

ct = ColumnTransformer(

transformers=[

('encoder', OneHotEncoder(), categorical\_features)

], remainder='passthrough')

X = ct.fit\_transform(df)

le = LabelEncoder()

y = le.fit\_transform(df['Survived'])

print("Updated matrix of features: \n", X)

print("Updated dependent variable vector: \n", y)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import pandas as pd

import numpy as np

from sklearn.impute import SimpleImputer

df = pd.read\_csv('pima-indians-diabetes.csv')

missing\_data = df.isnull().sum()

print("Missing data: \n", missing\_data)

imputer = SimpleImputer(missing\_values=np.nan, strategy='mean')

imputer.fit(df)

df\_imputed = imputer.transform(df)

print("Updated matrix of features: \n", df\_imputed)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

dataset = pd.read\_csv('iris.csv')

X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, -1].values

print(X)

print(y)

*# Spark Session*

**from** pyspark.sql **import** SparkSession

spark **=** (

SparkSession

**.**builder

**.**appName("Spark Introduction")

**.**master("local[\*]")

**.**getOrCreate()

)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Emp Data & Schema*

emp\_data **=** [

["001","101","John Doe","30","Male","50000","2015-01-01"],

["002","101","Jane Smith","25","Female","45000","2016-02-15"],

["020","102","Grace Kim","32","Female","53000","2018-11-01"]

]

emp\_schema **=** "employee\_id string, department\_id string, name string, age string, gender string, salary string, hire\_date string"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Create emp DataFrame*

emp **=** spark**.**createDataFrame(data**=**emp\_data, schema**=**emp\_schema)

emp**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write our first Transformation (EMP salary > 50000)*

emp\_final **=** emp**.**where("salary > 50000")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

emp\_final**.**write**.**format("csv")**.**save("data/output/1/emp.csv")

*# Small Example for Schema*

**from** pyspark.sql.types **import** StructType, StructField, StringType, IntegerType

schema\_string **=** "name string, age int"

schema\_spark **=** StructType([

StructField("name", StringType(), **True**),

StructField("age", IntegerType(), **True**)

])

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# SELECT columns*

*# select employee\_id, name, age, salary from emp*

emp\_filtered **=** emp**.**select(col("employee\_id"), expr("name"), emp**.**age, emp**.**salary)

*# SHOW Dataframe (ACTION)*

emp\_filtered**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Using expr for select*

*# select employee\_id as emp\_id, name, cast(age as int) as age, salary from emp\_filtered*

emp\_casted\_1 **=** emp\_filtered**.**selectExpr("employee\_id as emp\_id", "name", "cast(age as int) as age", "salary")

emp\_casted**.**printSchema()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Filter emp based on Age > 30*

*# select emp\_id, name, age, salary from emp\_casted where age > 30*

emp\_final **=** emp\_casted**.**select("emp\_id", "name", "age", "salary")**.**where("age > 30")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

schema\_str **=** "name string, age int"

**from** pyspark.sql.types **import** \_parse\_datatype\_string

schema\_spark **=** \_parse\_datatype\_string(schema\_str)

schema\_spark

*# Casting Column*

*# select employee\_id, name, age, cast(salary as double) as salary from emp*

**from** pyspark.sql.functions **import** col, cast

emp\_casted **=** emp**.**select("employee\_id", "name", "age", col("salary")**.**cast("double"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Adding Columns*

*# select employee\_id, name, age, salary, (salary \* 0.2) as tax from emp\_casted*

emp\_taxed **=** emp\_casted**.**withColumn("tax", col("salary") **\*** 0.2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Literals*

*# select employee\_id, name, age, salary, tax, 1 as columnOne, 'two' as columnTwo from emp\_taxed*

**from** pyspark.sql.functions **import** lit

emp\_new\_cols **=** emp\_taxed**.**withColumn("columnOne", lit(1))**.**withColumn("columnTwo", lit('two'))

+-----------+-------------+---+-------+-------+---------+---------+

|employee\_id| name|age| salary| tax|columnOne|columnTwo|

+-----------+-------------+---+-------+-------+---------+---------+

| 001| John Doe| 30|50000.0|10000.0| 1| two|

| 002| Jane Smith| 25|45000.0| 9000.0| 1| two|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Renaming Columns*

*# select employee\_id as emp\_id, name, age, salary, tax, columnOne, columnTwo from emp\_new\_cols*

emp\_1 **=** emp\_new\_cols**.**withColumnRenamed("employee\_id", "emp\_id")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Remove Column*

emp\_dropped **=** emp\_new\_cols**.**drop("columnTwo", "columnOne")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Filter data*

*# select employee\_id as emp\_id, name, age, salary, tax, columnOne from emp\_col\_dropped where tax > 1000*

emp\_filtered **=** emp\_dropped**.**where("tax > 10000")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# LIMIT data*

*# select employee\_id as emp\_id, name, age, salary, tax, columnOne from emp\_filtered limit 5*

emp\_limit **=** emp\_filtered**.**limit(5)

*# Show data*

emp\_limit**.**show(2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

*# Add multiple columns*

columns **=** {

"tax" : col("salary") **\*** 0.2 ,

"oneNumber" : lit(1),

"columnTwo" : lit("two")

}

emp\_final **=** emp**.**withColumns(columns)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Case When*

*# select employee\_id, name, age, salary, gender,*

*# case when gender = 'Male' then 'M' when gender = 'Female' then 'F' else null end as new\_gender, hire\_date from emp*

**from** pyspark.sql.functions **import** when, col, expr

emp\_gender\_fixed **=** emp**.**withColumn("new\_gender", when(col("gender") **==** 'Male', 'M')

**.**when(col("gender") **==** 'Female', 'F')

**.**otherwise(**None**)

)

emp\_gender\_fixed\_1 **=** emp**.**withColumn("new\_gender", expr("case when gender = 'Male' then 'M' when gender = 'Female' then 'F' else null end"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Replace in Strings*

*# select employee\_id, name, replace(name, 'J', 'Z') as new\_name, age, salary, gender, new\_gender, hire\_date from emp\_gender\_fixed*

**from** pyspark.sql.functions **import** regexp\_replace

emp\_name\_fixed **=** emp\_gender\_fixed**.**withColumn("new\_name", regexp\_replace(col("name"), "J", "Z"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Convert Date*

*# select \*, to\_date(hire\_date, 'YYYY-MM-DD') as hire\_date from emp\_name\_fixed*

**from** pyspark.sql.functions **import** to\_date

emp\_date\_fix **=** emp\_name\_fixed**.**withColumn("hire\_date", to\_date(col("hire\_date"), 'yyyy-MM-dd'))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Add Date Columns*

*# Add current\_date, current\_timestamp, extract year from hire\_date*

**from** pyspark.sql.functions **import** current\_date, current\_timestamp

emp\_dated **=** emp\_date\_fix**.**withColumn("date\_now", current\_date())**.**withColumn("timestamp\_now", current\_timestamp())

emp\_dated**.**show(truncate**=False**)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Drop Null gender records*

emp\_1 **=** emp\_dated**.**na**.**drop()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Fix Null values*

*# select \*, nvl('new\_gender', 'O') as new\_gender from emp\_dated*

**from** pyspark.sql.functions **import** coalesce, lit

emp\_null\_df **=** emp\_dated**.**withColumn("new\_gender", coalesce(col("new\_gender"), lit("O")))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Drop old columns and Fix new column names*

emp\_final **=** emp\_null\_df**.**drop("name", "gender")**.**withColumnRenamed("new\_name", "name")**.**withColumnRenamed("new\_gender", "gender")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

*# Convert date into String and extract date information*

**from** pyspark.sql.functions **import** date\_format

emp\_fixed **=** emp\_final**.**withColumn("date\_year", date\_format(col("timestamp\_now"), "z"))

*# Emp Data & Schema*

emp\_data\_1 **=** [

["001","101","John Doe","30","Male","50000","2015-01-01"],

["002","101","Jane Smith","25","Female","45000","2016-02-15"],

["010","104","Lisa Lee","27","Female","47000","2018-08-01"]

]

emp\_data\_2 **=** [

["011","104","David Park","38","Male","65000","2015-11-01"],

["012","105","Susan Chen","31","Female","54000","2017-02-15"],

["020","102","Grace Kim","32","Female","53000","2018-11-01"]

]

emp\_schema **=** "employee\_id string, department\_id string, name string, age string, gender string, salary string, hire\_date string"

*# Create emp DataFrame*

emp\_data\_1 **=** spark**.**createDataFrame(data**=**emp\_data\_1, schema**=**emp\_schema)

emp\_data\_2 **=** spark**.**createDataFrame(data**=**emp\_data\_2, schema**=**emp\_schema)

*# Show emp dataframe (ACTION)*

emp\_data\_1**.**show()

emp\_data\_2**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# UNION and UNION ALL*

*# select \* from emp\_data\_1 UNION select \* from emp\_data\_2*

emp **=** emp\_data\_1**.**unionAll(emp\_data\_2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Sort the emp data based on desc Salary*

*# select \* from emp order by salary desc*

**from** pyspark.sql.functions **import** desc, asc, col

emp\_sorted **=** emp**.**orderBy(col("salary")**.**asc())

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Aggregation*

*# select dept\_id, count(employee\_id) as total\_dept\_count from emp\_sorted group by dept\_id*

**from** pyspark.sql.functions **import** count

emp\_count **=** emp\_sorted**.**groupBy("department\_id")**.**agg(count("employee\_id")**.**alias("total\_dept\_count"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Aggregation*

*# select dept\_id, sum(salary) as total\_dept\_salary from emp\_sorted group by dept\_id*

**from** pyspark.sql.functions **import** sum

emp\_sum **=** emp\_sorted**.**groupBy("department\_id")**.**agg(sum("salary")**.**alias("total\_dept\_salary"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Aggregation with having clause*

*# select dept\_id, avg(salary) as avg\_dept\_salary from emp\_sorted group by dept\_id having avg(salary) > 50000*

**from** pyspark.sql.functions **import** avg

emp\_avg **=** emp\_sorted**.**groupBy("department\_id")**.**agg(avg("salary")**.**alias("avg\_dept\_salary"))**.**where("avg\_dept\_salary > 50000")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP - unionByName*

*# In case the column sequence is different*

emp\_data\_2\_other **=** emp\_data\_2**.**select("employee\_id", "salary", "department\_id", "name", "hire\_date", "gender", "age")

emp\_data\_1**.**printSchema()

emp\_data\_2\_other**.**printSchema()

emp\_fixed **=** emp\_data\_1**.**unionByName(emp\_data\_2\_other)

emp**.**count()

Out[ ]: 20

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Get unique data*

*# select distinct emp.\* from emp*

emp\_unique **=** emp**.**distinct()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Unique of department\_ids*

*# select distinct department\_id from emp*

emp\_dept\_id **=** emp**.**select("department\_id")**.**distinct()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Window Functions*

*# select \*, max(salary) over(partition by department\_id order by salary desc) as max\_salary from emp\_unique*

**from** pyspark.sql.window **import** Window

**from** pyspark.sql.functions **import** max, col, desc

window\_spec **=** Window**.**partitionBy(col("department\_id"))**.**orderBy(col("salary")**.**desc())

max\_func **=** max(col("salary"))**.**over(window\_spec)

emp\_1 **=** emp**.**withColumn("max\_salary", max\_func)

In [18]:

emp\_1**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Window Functions - 2nd highest salary of each department*

*# select \*, row\_number() over(partition by department\_id order by salary desc) as rn from emp\_unique where rn = 2*

**from** pyspark.sql.window **import** Window

**from** pyspark.sql.functions **import** row\_number, desc, col

window\_spec **=** Window**.**partitionBy(col("department\_id"))**.**orderBy(col("salary")**.**desc())

rn **=** row\_number()**.**over(window\_spec)

emp\_2 **=** emp**.**withColumn("rn", rn)**.**where("rn = 2")

(or)

*# Window function using expr*

*# select \*, row\_number() over(partition by department\_id order by salary desc) as rn from emp\_unique where rn = 2*

**from** pyspark.sql.functions **import** expr

emp\_3 **=** emp**.**withColumn("rn", expr("row\_number() over(partition by department\_id order by salary desc)"))**.**where("rn = 2")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Emp Data & Schema*

emp\_data **=** [

["001","101","John Doe","30","Male","50000","2015-01-01"],

["002","101","Jane Smith","25","Female","45000","2016-02-15"],

["020","102","Grace Kim","32","Female","53000","2018-11-01"]

]

emp\_schema **=** "employee\_id string, department\_id string, name string, age string, gender string, salary string, hire\_date string"

dept\_data **=** [

["101", "Sales", "NYC", "US", "1000000"],

["102", "Marketing", "LA", "US", "900000"],

["107", "Customer Service", "Sydney", "Australia", "950000"]

]

dept\_schema **=** "department\_id string, department\_name string, city string, country string, budget string"

In [3]:

*# Create emp & dept DataFrame*

emp **=** spark**.**createDataFrame(data**=**emp\_data, schema**=**emp\_schema)

dept **=** spark**.**createDataFrame(data**=**dept\_data, schema**=**dept\_schema)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Get number of partitions for dept*

dept**.**rdd**.**getNumPartitions()

Out[7]: 8

*# Repartition of data using repartition & coalesce*

emp\_partitioned **=** emp**.**repartition(4, "department\_id")

emp\_partitioned**.**rdd**.**getNumPartitions()

Out[23]: 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Find the partition info for partitions and reparition*

**from** pyspark.sql.functions **import** spark\_partition\_id

emp\_1 **=** emp**.**repartition(4, "department\_id")**.**withColumn("partition\_num", spark\_partition\_id())

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# INNER JOIN datasets*

*# select e.emp\_name, d.department\_name, d.department\_id, e.salary*

*# from emp e inner join dept d on emp.department\_id = dept.department\_id*

df\_joined **=** emp**.**alias("e")**.**join(dept**.**alias("d"), how**=**"inner", on**=**emp**.**department\_id**==**dept**.**department\_id)

In [35]:

df\_joined**.**select("e.name", "d.department\_id", "d.department\_name", "e.salary")**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# LEFT OUTER JOIN datasets*

*# select e.emp\_name, d.department\_name, d.department\_id, e.salary*

*# from emp e left outer join dept d on emp.department\_id = dept.department\_id*

df\_joined **=** emp**.**alias("e")**.**join(dept**.**alias("d"), how**=**"left\_outer", on**=**emp**.**department\_id**==**dept**.**department\_id)

In [38]:

df\_joined**.**select("e.name", "d.department\_name", "d.department\_id", "e.salary")**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write the final dataset*

df\_joined**.**select("e.name", "d.department\_name", "d.department\_id","e.salary")**.**write**.**format("csv")**.**save("data/output/7/emp\_joined.csv")

In [52]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

*# Joins with cascading conditions*

*# Join with Department\_id and only for departments 101 or 102*

*# Join with not null/null conditions*

df\_final **=** emp**.**join(dept, how**=**"left\_outer",

on**=**(emp**.**department\_id**==**dept**.**department\_id) **&** ((emp**.**department\_id **==** "101") **|** (emp**.**department\_id **==** "102"))

**&** (emp**.**salary**.**isNull())

)

In [53]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read a csv file into dataframe*

df **=** spark**.**read**.**format("csv")**.**option("header", **True**)**.**option("inferSchema", **True**)**.**load("data/input/emp.csv")

In [14]:

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# Reading with Schema*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date"

df\_schema **=** spark**.**read**.**format("csv")**.**option("header",**True**)**.**schema(\_schema)**.**load("data/input/emp.csv")

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# Handle BAD records - PERMISSIVE (Default mode)*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date, bad\_record string"

df\_p **=** spark**.**read**.**format("csv")**.**schema(\_schema)**.**option("columnNameOfCorruptRecord", "bad\_record")**.**option("header", **True**)**.**load("data/input/emp\_new.csv")

In [37]:

*# Handle BAD records - DROPMALFORMED*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date"

df\_m **=** spark**.**read**.**format("csv")**.**option("header", **True**)**.**option("mode", "DROPMALFORMED")**.**schema(\_schema)**.**load("data/input/emp\_new.csv")

In [45]:

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# Handle BAD records - FAILFAST*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date"

df\_m **=** spark**.**read**.**format("csv")**.**option("header", **True**)**.**option("mode", "FAILFAST")**.**schema(\_schema)**.**load("data/input/emp\_new.csv")

In [48]:

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# BONUS TIP*

*# Multiple options*

\_options **=** {

"header" : "true",

"inferSchema" : "true",

"mode" : "PERMISSIVE" }

df **=** (spark**.**read**.**format("csv")**.**options(**\*\***\_options)**.**load("data/input/emp.csv"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read Parquet Sales data*

df\_parquet **=** spark**.**read**.**format("parquet")**.**load("data/input/sales\_total\_parquet/\*.parquet")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read ORC Sales data*

df\_orc **=** spark**.**read**.**format("orc")**.**load("data/input/sales\_total\_orc/\*.orc")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Benefits of Columnar Storage*

*# Lets create a simple Python decorator - {get\_time} to get the execution timings*

*# If you dont know about Python decorators - check out : https://www.geeksforgeeks.org/decorators-in-python/*

**import** time

**def** get\_time(func):

**def** inner\_get\_time() **->** str:

start\_time **=** time**.**time()

func()

end\_time **=** time**.**time()

**return** (f"Execution time: {(end\_time **-** start\_time)**\***1000} ms")

print(inner\_get\_time())

In [16]:

@get\_time

**def** x():

df **=** spark**.**read**.**format("parquet")**.**load("data/input/sales\_data.parquet")

df**.**count()

Execution time: 672.5492477416992 ms

In [17]:

@get\_time

**def** x():

df **=** spark**.**read**.**format("parquet")**.**load("data/input/sales\_data.parquet")

df**.**select("trx\_id")**.**count()

Execution time: 348.848819732666 ms

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# BONUS TIP*

*# RECURSIVE READ*

df\_1 **=** spark**.**read**.**format("parquet")**.**option("recursiveFileLookup", **True**)**.**load("data/input/sales\_recursive/")

df\_1**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read Single line JSON file*

df\_single **=** spark**.**read**.**format("json")**.**load("data/input/order\_singleline.json")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read Multiline JSON file*

df\_multi **=** spark**.**read**.**format("json")**.**option("multiLine", **True**)**.**load("data/input/order\_multiline.json")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# With Schema*

\_schema **=** "customer\_id string, order\_id string, contact array<long>"

df\_schema **=** spark**.**read**.**format("json")**.**schema(\_schema)**.**load("data/input/order\_singleline.json")

df\_schema**.**show()

\_schema **=** "contact array<string>, customer\_id string, order\_id string, order\_line\_items array<struct<amount double, item\_id string, qty long>>"

In [21]:

df\_schema\_new **=** spark**.**read**.**format("json")**.**schema(\_schema)**.**load("data/input/order\_singleline.json")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Function from\_json to read from a column*

\_schema **=** "contact array<string>, customer\_id string, order\_id string, order\_line\_items array<struct<amount double, item\_id string, qty long>>"

**from** pyspark.sql.functions **import** from\_json

df\_expanded **=** df**.**withColumn("parsed", from\_json(df**.**value, \_schema))

*# Function to\_json to parse a JSON string*

**from** pyspark.sql.functions **import** to\_json

df\_unparsed **=** df\_expanded**.**withColumn("unparsed", to\_json(df\_expanded**.**parsed))

df\_unparsed**.**select("unparsed")**.**show(truncate**=False**)

*# Get values from Parsed JSON*

df\_1 **=** df\_expanded**.**select("parsed.\*")

In [38]:

**from** pyspark.sql.functions **import** explode

df\_2 **=** df\_1**.**withColumn("expanded\_line\_items", explode("order\_line\_items"))

df\_3 **=** df\_2**.**select("contact", "customer\_id", "order\_id", "expanded\_line\_items.\*")

*# Explode Array fields*

df\_final **=** df\_3**.**withColumn("contact\_expanded", explode("contact"))

df\_final**.**drop("contact")**.**show()

+-----------+--------+------+-------+---+----------------+

|customer\_id|order\_id|amount|item\_id|qty|contact\_expanded|

+-----------+--------+------+-------+---+----------------+

| C001| O101|102.45| I001| 6| 9000010000|

| C001| O101|102.45| I001| 6| 9000010001|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write the data in parquet format*

emp**.**write**.**format("parquet")**.**save("data/output/11/2/emp.parquet")

In [10]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# View data partition information*

**from** pyspark.sql.functions **import** spark\_partition\_id

emp**.**withColumn("partition\_id", spark\_partition\_id())**.**show()

emp**.**write**.**format("csv")**.**option("header", **True**)**.**save("data/output/11/3/emp.csv")

In [12]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write the data with Partition to output location*

emp**.**write**.**format("csv")**.**partitionBy("department\_id")**.**option("header", **True**)**.**save("data/output/11/4/emp.csv")

In [16]:

*# Write Modes - append, overwrite, ignore and error*

emp**.**write**.**format("csv")**.**mode("error")**.**option("header", **True**)**.**save("data/output/11/3/emp.csv")

*# Bonus TIP*

*# What if we need to write only 1 output file to share with DownStream?*

emp**.**repartition(1)**.**write**.**format("csv")**.**option("header", **True**)**.**save("data/output/11/5/emp.csv")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**1.   SQL Query for Retrieving Tables**

This query can be run to retrieve the list of tables present in a database where the database is “My\_Schema”.

With the SELECT command, users can define the columns that they want to get in the query output. This command is also useful to get which column users want to see as the output table. The SELECT statement is applied to pick data from a table. The data retrieved is put in a result table, named the result set. The output data is saved in a result table. This output table is also termed the result set.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** My\_Schema.Tables; |

**2.**[Query for Selecting Columns from a Table](https://bytescout.com/blog/deep-sql-queries-and-examples.html#2)

These are perhaps the most useful SQL query examples. In the example below, we are extracting the “Student\_ID” column or attribute from the table “STUDENT”. The select statement is used to select data from the database.

|  |  |
| --- | --- |
| 1 | **SELECT** Student\_ID **FROM** STUDENT; |

If you want to display all the attributes from a particular table, this is the right query to use:

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** STUDENT; |

**3.**[Query for Outputting Data Using a Constraint](https://bytescout.com/blog/deep-sql-queries-and-examples.html#3)

This SQL query retrieves the specified attributes from the table on the constraint *Employee ID =0000*

|  |  |
| --- | --- |
| 1 | **SELECT** EMP\_ID, **NAME** **FROM** EMPLOYEE\_TBL **WHERE** EMP\_ID = '0000'; |

**4. Query for Outputting Sorted Data Using ‘Order By’**

This query orders the results with respect to the attribute which is referenced using “Order By” – so for example, if that attribute is an integer data type, then the result would either be sorted in ascending or descending order; likewise, if the data type is a String then the result would be ordered in alphabetical order. The order by clause is used to sort the data from the table. The order by clause should always be used in the last of the SQL query.

|  |  |
| --- | --- |
| 1  2 | **SELECT** EMP\_ID, LAST\_NAME **FROM** EMPLOYEE  **WHERE** CITY = 'Seattle' **ORDER** **BY** EMP\_ID; |

The ordering of the result can also be set manually, using “asc ” for ascending and “desc” for descending.

Ascending (ASC) is the default condition for the ORDER BY clause. In other words, if users don’t specify ASC or DESC after the column name, then the result will be ordered in ascending order only.

|  |  |
| --- | --- |
| 1  2 | **SELECT** EMP\_ID, LAST\_NAME **FROM** EMPLOYEE\_TBL  **WHERE** CITY = 'INDIANAPOLIS' **ORDER** **BY** EMP\_ID **asc**; |

**5.   SQL Query for Outputting Sorted Data Using ‘Group By’**

The ‘Group By’ property groups the resulting data according to the specified attribute.

The SQL query below will select Name, Age columns from the Patients table, then will filter them by Age value to include records where Age is more than 40 and then will group records with similar Age value and then finally will output them sorted by Name. The basic rule is that the group by clause should always follow a where clause in a Select statement and must precede the Order by clause.

|  |  |
| --- | --- |
| 1  2 | **SELECT** **Name**, Age **FROM** Patients **WHERE** Age > 40  **GROUP** **BY** **Name**, Age **ORDER** **BY** **Name**; |

Another sample of use of Group By: this expression will select records with a price lesser than 70 from the Orders table, will group records with a similar price, will sort the output by price, and will also add the column COUNT(price) that will display how many records with similar price were found:

|  |  |
| --- | --- |
| 1  2 | **SELECT** COUNT(price), price **FROM** orders  **WHERE** price < 70 **GROUP** **BY** price **ORDER** **BY** price |

Note: you should use the very same set of columns for both SELECT and GROUP BY commands, otherwise you will get an error. Many thanks to *Sachidannad* for pointing it out!

**SQL Queries for Data Manipulation Using Math Functions**

There are a lot of built-in math functions like COUNT and AVG which provide basic functionalities of counting the number of results and averaging them respectively.

**6. Data Manipulation Using COUNT**

This query displays the total number of customers by counting each customer ID. In addition, it groups the results according to the country of each customer. **In count, if users define DISTINCT, then they cal also define the query\_partition\_clause.** This clause is a part of the analytic clause, and other clauses such as order\_by\_clause and windowing\_clause are not permitted.

**Syntax: SELECT COUNT(colname) FROM table name;**

|  |  |
| --- | --- |
| 1 | **SELECT** COUNT(CustomerID), Country **FROM** Customers **GROUP** **BY** Country; |

**7.**[Data Manipulation Using SUM](https://bytescout.com/blog/deep-sql-queries-and-examples.html#7)

SUM calculates the total of the attribute that is given to it as an argument. SUM is an aggregate function and it calculates the sum of all the distinct values. and the sum of all the duplicate values.

|  |  |
| --- | --- |
| 1 | **SELECT** SUM(Salary)**FROM** Employee **WHERE** Emp\_Age < 30; |

**8. Data Manipulation Using AVG**

Simple – an average of a given attribute. Average is also an aggregate function in SQL. The AVG() function computes the average of non-NULL values in a column. It ignores the null values.

|  |  |
| --- | --- |
| 1 | **SELECT** AVG(Price)**FROM** Products; |

**9.   SQL Query for Listing all Views**

This SQL query lists all the views available in the schema.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** My\_Schema.views; |

**10. Query for Creating a View**

A view is a tailored table that is formed as a result of a query. It has tables and rows just like any other table. It’s usually a good idea to run queries in SQL as independent views because this allows them to be retrieved later to view the query results, rather than computing the same command every time for a particular set of results.

|  |  |
| --- | --- |
| 1  2  3  4 | **CREATE** **VIEW** Failing\_Students **AS**  **SELECT** S\_NAME, Student\_ID  **FROM** STUDENT  **WHERE** GPA > 40; |

**11. Query for Retrieving a View**

The standard syntax of selecting attributes from a table is applicable to views as well.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Failing\_Students; |

**12. Query for Updating a View**

This query updates the view named ‘Product List’ – and if this view doesn’t exist, then the Product List view gets created as specified in this query. The view is also called a virtual table. In other words, a view is just a mirrored copy of a table whose data is the result of a stored query.

A view is a legitimate copy of a different table or sequence of tables. A view obtains its information or data from the tables from previously created tables known as base tables. Base tables are real tables. All procedures implemented on a view really modify the base table. Users can use views just like the real or base tables. In view, users can apply various DDL, DML commands such as update, insert into, and delete.

|  |  |
| --- | --- |
| 1  2  3  4 | **CREATE** OR REPLACE **VIEW** [ Product List] **AS**  **SELECT** ProductID, ProductName, Category  **FROM** Products  **WHERE** Discontinued = **No**; |

**13. Query for Dropping a View**

This query will drop or delete a view named ‘V1’. The important thing to remember here is that the DROP VIEW is disallowed if there are any views dependent on the view you are about to drop.

|  |  |
| --- | --- |
| 1 | **DROP** **VIEW** V1; |

**14. Query to Display User Tables**

**A user-defined table** is a representation of defined information in a table, and it can be used as arguments for procedures or **user-defined functions**. Because they’re so useful, it’s useful to keep track of them using the following query. User tables explain the relational tables of the current user.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Sys.objects **WHERE** Type='u' |

**15. Query to Display Primary Keys**

A primary key uniquely identifies all values within a table. A primary key imposes a NOT NULL restriction and a unique constraint in one declaration. In other words, it prevents various rows from having similar values or sequences of columns. It doesn’t allow null values. The primary key can be defined as a single column or the combination of two columns in a table. It is responsible for all the relationships between the tables.

The following SQL query lists all the fields in a **table’s primary key.**

|  |  |
| --- | --- |
| 1 | **SELECT** \* **from** Sys.Objects **WHERE** Type='PK' |

**16. Query for Displaying Unique Keys**

**A Unique Key** allows a column to ensure that all of its values are different. A unique key also recognizes a different tuple uniquely in relation to or table. A table can have more than one unique key. Unique key constraints can take only one NULL value for the column.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Sys.Objects **WHERE** Type='uq' |

**17. Displaying Foreign Keys**

**Foreign keys link** one table to another – they are attributes in one table which refer to the primary key of another table.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Sys.Objects **WHERE** Type='f' |

Primary, Unique, and Foreign are part of the constraints in SQL. Constraints are essential to the scalability, compliance, and sincerity of the data. Constraints implement particular rules, assuring the data adheres to the conditions outlined. For example, these are the laws imposed on the columns of the database tables. These are applied to restrict the kind of data in the table. This assures the efficiency and authenticity of the database.

**18. Displaying Triggers**

A **Trigger** is sort of an ‘event listener’ – i.e, it’s a pre-specified set of instructions that execute when a certain event occurs. The **list of defined triggers** can be viewed using the following query.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Sys.Objects **WHERE** Type='tr' |

**19. Displaying Internal Tables**

**Internal tables** are formed as a by-product of a **user action** and are usually not accessible. The data in internal tables cannot be manipulated; however, the metadata of the internal tables can be viewed using the following query.

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Sys.Objects **WHERE** Type='it' |

**20. Displaying a List of Procedures**

A stored procedure is a **group of advanced SQL queries** that logically form a single unit and perform a particular task. Thus, using the following query you can keep track of them:

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Sys.Objects **WHERE** Type='p' |

**21. Swapping the Values of Two Columns in a table**

In this and subsequent examples, we will use a common company database including several tables that are easily visualized. Our practice DB will include a Customer table and an Order table. The Customers table will contain some obvious columns including ID, Name, Address, zip, and email, for example, where we assume for now that the primary key field for indexing is the *Customer\_ID* field.

With this in mind, we can easily imagine an Orders table that likewise contains the indexed customer ID field, along with details of each order placed by the customer. This table will include the order Number, Quantity, Date, Item, and Price. In our first one of **SQL examples**, imagine a situation where the zip and phone fields were transposed and all the phone numbers were erroneously entered into the zip code field. We can easily fix this problem with the following [SQL statement](https://bytescout.com/blog/postgresql-advanced-queries.html):

|  |  |
| --- | --- |
| 1 | **UPDATE** Customers **SET** Zip=Phone, Phone=Zip |

**22.**[Returning a Column of Unique Values](https://bytescout.com/blog/deep-sql-queries-and-examples.html#22)

Now, suppose that our data entry operator added the same Customers to the Customers table more than once by mistake. As you know, proper indexing requires that the key field contains only unique values. To fix the problem, we will use *SELECT DISTINCT* to create an indexable list of unique customers:

|  |  |
| --- | --- |
| 1 | **SELECT** **DISTINCT** ID **FROM** Customers |

**23. Making a Top 25 with the SELECT TOP Clause**

Next, imagine that our Customers table has grown to include thousands of records, but we just want to show a sample of 25 of these records to demonstrate the column headings and The SELECT TOP clause allows us to specify the number of records to return, like a *Top-25 list*. In this example we will return the top 25 from our Customers table:

|  |  |
| --- | --- |
| 1 | **SELECT** **TOP** 25 **FROM** Customers **WHERE** Customer\_ID<>NULL; |

**24. Searching for SQL Tables with Wildcards**

Wildcard characters or operators like “%” make it easy to find particular strings in a large table of thousands of records. Suppose we want to find all of our customers who have names beginning with “Herb” including Herberts, and Herbertson. The **%** wildcard symbol can be used to achieve such a result. The following [SQL query](https://bytescout.com/blog/postgresql-advanced-queries.html) will return all rows from the Customer table where the **Customer\_name** field begins with “Herb”:

|  |  |
| --- | --- |
| 1 | **SELECT** \* **From** Customers **WHERE** **Name** LIKE 'Herb%' |

**25. Between Monday and Tuesday**

Today is Wednesday, and we arrive at work and discover that our new data entry clerk in training has entered all new orders incorrectly on Monday and Tuesday. We wish to teach our new trainee to find and correct all erroneous records. What’s the easiest way to get all the records from the Orders table entered on Monday and Tuesday? The Between clause makes the task a breeze:

|  |  |
| --- | --- |
| 1  2 | **SELECT** ID **FROM** Orders **WHERE**  **Date** BETWEEN ‘01/12/2018’ AND ‘01/13/2018’ |

**26.**[Finding the Intersection of Two Tables](https://bytescout.com/blog/deep-sql-queries-and-examples.html#26)

Undoubtedly the whole reason that a relational database exists in the first place is to find matching records in two tables! The JOIN statement accomplishes this core objective of SQL and makes the task easy. Here we are going to fetch a list of all records which have matches in the Customers and Orders tables:

|  |  |
| --- | --- |
| 1  2 | **SELECT** ID **FROM** Customers **INNER**  JOIN Orders **ON** Customers.ID = Orders.ID |

The point of INNER JOIN, in this case, is to select records in the Customers table which have matching customer ID values in the Orders table and return only those records. Of course, there are many types of JOIN, such as FULL, SELF, and LEFT, but for now, let’s keep things interesting and move on to more diverse types of advanced SQL commands.

**27. Doubling the Power with UNION**

We can combine the results of two SQL query examples into one naturally with the UNION keyword. Suppose we want to create a new table by combining the Customer\_name and phone from Customers with a list of that customer’s recent orders so that we can look for patterns and perhaps suggest future purchases. Here is a quick way to accomplish the task:

|  |  |
| --- | --- |
| 1  2 | **SELECT** phone **FROM** Customers  **UNION** **SELECT** item **FROM** Orders |

The UNION keyword makes it possible to combine JOINS and other criteria to achieve a very powerful new table generation potential.

**28.**[Making Column Labels More Friendly](https://bytescout.com/blog/deep-sql-queries-and-examples.html#28)

Aliasing column labels give us the convenience of renaming a column label to something more readable. There is a tradeoff when naming columns to make them succinct results in reduced readability in subsequent daily use. In our Orders table, the item column contains the description of purchased products. Let’s see how to alias the item column to temporarily rename it for greater user-friendliness:

|  |  |
| --- | --- |
| 1 | **SELECT** Item **AS** item\_description **FROM** Orders |

**29. Always and Everywhere!**

Wouldn’t it be great if there were a set of conditions you could depend on every time? The complex SQL queries using **ANY** and **ALL** can make this ideal a reality! Let’s look at how the **ALL** keyword is used to include records only when a set of conditions is true for **ALL** records. In the following example, we will return records from the Orders table where the idea is to get a list of high volume orders for a given item, in this case for customers who ordered more than 50 of the product:

|  |  |
| --- | --- |
| 1  2  3  4 | **SELECT** Item **FROM** Orders  **WHERE** id = ALL  (**SELECT** ID **FROM** Orders  **WHERE** quantity > 50) |

**30. Writing Developer Friendly SQL**

An often overlooked but very important element of SQL scripting is adding comments to a script of queries to explain what it’s doing for the benefit of future developers who may need to revise and update your SQL queries.

**A SQL script is a collection of SQL elements and commands accumulated as a file in SQL Scripts. This script file can include many SQL commands or PL/SQL codes. One can utilize SQL Scripts to build, edit, design, execute, and delete files.**

The **—** single line and the **/\*** .. **\*/** multi-line delimiters empower us to add useful comments to scripts, but this is also used in another valuable way. Sometimes a section of code may not be in use, but we don’t want to delete it, because we anticipate using it again. Here we can simply add the comment delimiter to deactivate it momentarily:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | /\* This query below is commented so it won't execute\*/  /\*  SELECT item FROM Orders  WHERE date ALL = (SELECT Order\_ID FROM Orders  WHERE quantity > 50)  \*/    /\* the SQL query below the will be executed  ignoring the text after "--"  \*/    **SELECT** item -- single comment  **FROM** Orders -- another single comment  **WHERE** id  ALL = (**SELECT** ID **FROM** Orders  **WHERE** quantity > 25) |

**31.  SQL queries for Database Management**

So far we have explored SQL query examples for querying tables and combining records from multiple queries. Now it’s time to take a step upward and look at the database on a structural level. Let’s start with the easiest SQL statement of all which creates a new database. Here, we are going to create the DB as a container for our Customers and Orders tables used in the previous ten examples above:

|  |  |
| --- | --- |
| 1 | **CREATE** **DATABASE** AllSales |

**32.**[Adding Tables to Our New DB](https://bytescout.com/blog/deep-sql-queries-and-examples.html#32)

Next, we will actually add the Customers table which we’ve been using in previous examples, and then add some of the column labels which we are already familiar with:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **CREATE** **TABLE** Customers (  ID **varchar**(80),  **Name** **varchar**(80),  Phone **varchar**(20),  ....  ); |

Although most databases are created using a UI such as Access or OpenOffice, it is important to know how to create and delete databases and tables programmatically via code with SQL statements. This is especially so when installing a new web app and the UI asks new users to enter names for DBs to be added during installation.

**33.**[Modifying and Deleting Tables with SQL](https://bytescout.com/blog/deep-sql-queries-and-examples.html#33)

The ALTER statement is used to modify or change the meaning of a table. In the case of the relational tables with columns, ALTER statement is used to update the table to the new or modified rules or definition. **Alter** belongs to the DDL category of Commands. Data definition language can be described as a pattern for commands through which data structures are represented.

Imagine that you decide to send a birthday card to your customers to show your appreciation for their business, and so you want to add a birthday field to the Customers table. In these [SQL examples](https://bytescout.com/blog/postgresql-advanced-queries.html), you see how easy it is to **modify existing tables with the ALTER statement:**

|  |  |
| --- | --- |
| 1 | **ALTER** **TABLE** Customers **ADD** Birthday **varchar**(80) |

If a table becomes corrupted with bad data you can quickly delete it like this:

|  |  |
| --- | --- |
| 1 | **DROP** **TABLE** table\_name |

**34.**[The Key to Successful Indexing](https://bytescout.com/blog/deep-sql-queries-and-examples.html#34)

*An index is a schema element that includes a record for each content that arrives in the indexed column of the database table or cluster and gives a high-speed path to rows. There are many types of indexes such as Bitmap indexes, Partitioned indexes, Function-based indexes, and Domain indexes.*

Accurate indexing requires that the Primary Key column contains only unique values for this purpose. This guarantees that JOIN statements will maintain integrity and produce valid matches. Let’s create our Customers table again and establish the ID column as the Primary Key:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **CREATE** **TABLE** Customers (  ID **int** NOT NULL,  **Name** **varchar**(80) NOT NULL,  **PRIMARY** **KEY** (ID)  ); |

We can extend the functionality of the Primary Key so that it automatically increments from a base. Change the ID entry above to add the *AUTO\_INCREMENT* keyword as in the following statement:

|  |  |
| --- | --- |
| 1 | ID **int** NOT NULL AUTO\_INCREMENT |

**35. Advanced Concepts For Improving Performance**

Whenever practical, is always better to write the column name list into a SELECT statement rather than using the **\*** delimiter as a wildcard to select all columns. SQL Server has to do a search and replace operation to find all the columns in your table and write them into the statement for you (every time the SELECT is executed). For example:

|  |  |
| --- | --- |
| 1 | **SELECT** \* **FROM** Customers |

Would actually execute much faster on our database as:

|  |  |
| --- | --- |
| 1  2 | **SELECT** **Name**, Birthday, Phone,  Address, Zip **FROM** Customers |

Performance pitfalls can be avoided in many ways. For example, avoid the time sinkhole of forcing [SQL Server](https://bytescout.com/blog/postgresql-advanced-queries.html) to check the system/master database every time by using only a stored procedure name, and never prefix it with SP\_. Also setting NOCOUNT ON reduces the time required for SQL Server to count rows affected by INSERT, DELETE, and other commands. Using INNER JOIN with a condition is much faster than using WHERE clauses with conditions. We advise developers to learn SQL server queries to an advanced level for this purpose. For production purposes, these tips may be crucial to adequate performance. Notice that our tutorial examples tend to favor the INNER JOIN.

**36. Conditional Subquery Results**

The SQL operator EXISTS tests for the existence of records in a subquery and returns a value TRUE if a subquery returns one or more records. Have a look at this query with a subquery condition:

|  |  |
| --- | --- |
| 1  2  3 | **SELECT** **Name** **FROM** Customers **WHERE** EXISTS  (**SELECT** Item **FROM** Orders  **WHERE** Customers.ID = Orders.ID AND Price < 50) |

In this example above, the SELECT returns a value of TRUE when a customer has orders valued at less than $50.

**37. Copying Selections from Table to Table**

There are a hundred and one uses for this SQL tool. Suppose you want to archive your yearly Orders table into a larger archive table. This next example shows how to do it.

|  |  |
| --- | --- |
| 1  2  3 | **INSERT** **INTO** Yearly\_Orders  **SELECT** \* **FROM** Orders  **WHERE** **Date**<=1/1/2018 |

This example will add any records from the year 2018 to the archive.

**38. Catching NULL Results**

*The NULL is the terminology applied to describe an absent value. Null does not mean zero. A NULL value in a column of a table is a condition in a domain that seems to be empty. A column with a NULL value is a domain with an absent value. It is essential to recognize that a NULL value is distinct from a zero.*

In cases where NULL values are allowed in a field, calculations on those values will produce NULL results as well. This can be avoided by the use of the IFNULL operator. In this next example, a value of zero is returned rather than a value of NULL when the calculation encounters a field with a NULL value:

|  |  |
| --- | --- |
| 1  2  3 | **SELECT** Item, Price \*  (QtyInStock + IFNULL(QtyOnOrder, 0))  **FROM** Orders |

**39. HAVING can be Relieving!**

The problem was that the SQL WHERE clause could not operate on aggregate functions. The problem was solved by using the HAVING clause. As an example, this next query fetches a list of customers by the region where there is at least one customer per region:

|  |  |
| --- | --- |
| 1  2  3  4 | **SELECT** COUNT(ID), Region  **FROM** Customers  **GROUP** **BY** Region  **HAVING** COUNT(ID) > 0; |

*# Spark Session*

**from** pyspark.sql **import** SparkSession

spark **=** (

SparkSession

**.**builder

**.**appName("Spark Introduction")

**.**master("local[\*]")

**.**getOrCreate()

)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Emp Data & Schema*

emp\_data **=** [

["001","101","John Doe","30","Male","50000","2015-01-01"],

["002","101","Jane Smith","25","Female","45000","2016-02-15"],

["020","102","Grace Kim","32","Female","53000","2018-11-01"]

]

emp\_schema **=** "employee\_id string, department\_id string, name string, age string, gender string, salary string, hire\_date string"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Create emp DataFrame*

emp **=** spark**.**createDataFrame(data**=**emp\_data, schema**=**emp\_schema)

emp**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write our first Transformation (EMP salary > 50000)*

emp\_final **=** emp**.**where("salary > 50000")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

emp\_final**.**write**.**format("csv")**.**save("data/output/1/emp.csv")

*# Small Example for Schema*

**from** pyspark.sql.types **import** StructType, StructField, StringType, IntegerType

schema\_string **=** "name string, age int"

schema\_spark **=** StructType([

StructField("name", StringType(), **True**),

StructField("age", IntegerType(), **True**)

])

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# SELECT columns*

*# select employee\_id, name, age, salary from emp*

emp\_filtered **=** emp**.**select(col("employee\_id"), expr("name"), emp**.**age, emp**.**salary)

*# SHOW Dataframe (ACTION)*

emp\_filtered**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Using expr for select*

*# select employee\_id as emp\_id, name, cast(age as int) as age, salary from emp\_filtered*

emp\_casted\_1 **=** emp\_filtered**.**selectExpr("employee\_id as emp\_id", "name", "cast(age as int) as age", "salary")

emp\_casted**.**printSchema()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Filter emp based on Age > 30*

*# select emp\_id, name, age, salary from emp\_casted where age > 30*

emp\_final **=** emp\_casted**.**select("emp\_id", "name", "age", "salary")**.**where("age > 30")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

schema\_str **=** "name string, age int"

**from** pyspark.sql.types **import** \_parse\_datatype\_string

schema\_spark **=** \_parse\_datatype\_string(schema\_str)

schema\_spark

*# Casting Column*

*# select employee\_id, name, age, cast(salary as double) as salary from emp*

**from** pyspark.sql.functions **import** col, cast

emp\_casted **=** emp**.**select("employee\_id", "name", "age", col("salary")**.**cast("double"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Adding Columns*

*# select employee\_id, name, age, salary, (salary \* 0.2) as tax from emp\_casted*

emp\_taxed **=** emp\_casted**.**withColumn("tax", col("salary") **\*** 0.2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Literals*

*# select employee\_id, name, age, salary, tax, 1 as columnOne, 'two' as columnTwo from emp\_taxed*

**from** pyspark.sql.functions **import** lit

emp\_new\_cols **=** emp\_taxed**.**withColumn("columnOne", lit(1))**.**withColumn("columnTwo", lit('two'))

+-----------+-------------+---+-------+-------+---------+---------+

|employee\_id| name|age| salary| tax|columnOne|columnTwo|

+-----------+-------------+---+-------+-------+---------+---------+

| 001| John Doe| 30|50000.0|10000.0| 1| two|

| 002| Jane Smith| 25|45000.0| 9000.0| 1| two|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Renaming Columns*

*# select employee\_id as emp\_id, name, age, salary, tax, columnOne, columnTwo from emp\_new\_cols*

emp\_1 **=** emp\_new\_cols**.**withColumnRenamed("employee\_id", "emp\_id")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Remove Column*

emp\_dropped **=** emp\_new\_cols**.**drop("columnTwo", "columnOne")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Filter data*

*# select employee\_id as emp\_id, name, age, salary, tax, columnOne from emp\_col\_dropped where tax > 1000*

emp\_filtered **=** emp\_dropped**.**where("tax > 10000")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# LIMIT data*

*# select employee\_id as emp\_id, name, age, salary, tax, columnOne from emp\_filtered limit 5*

emp\_limit **=** emp\_filtered**.**limit(5)

*# Show data*

emp\_limit**.**show(2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

*# Add multiple columns*

columns **=** {

"tax" : col("salary") **\*** 0.2 ,

"oneNumber" : lit(1),

"columnTwo" : lit("two")

}

emp\_final **=** emp**.**withColumns(columns)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Case When*

*# select employee\_id, name, age, salary, gender,*

*# case when gender = 'Male' then 'M' when gender = 'Female' then 'F' else null end as new\_gender, hire\_date from emp*

**from** pyspark.sql.functions **import** when, col, expr

emp\_gender\_fixed **=** emp**.**withColumn("new\_gender", when(col("gender") **==** 'Male', 'M')

**.**when(col("gender") **==** 'Female', 'F')

**.**otherwise(**None**)

)

emp\_gender\_fixed\_1 **=** emp**.**withColumn("new\_gender", expr("case when gender = 'Male' then 'M' when gender = 'Female' then 'F' else null end"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Replace in Strings*

*# select employee\_id, name, replace(name, 'J', 'Z') as new\_name, age, salary, gender, new\_gender, hire\_date from emp\_gender\_fixed*

**from** pyspark.sql.functions **import** regexp\_replace

emp\_name\_fixed **=** emp\_gender\_fixed**.**withColumn("new\_name", regexp\_replace(col("name"), "J", "Z"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Convert Date*

*# select \*, to\_date(hire\_date, 'YYYY-MM-DD') as hire\_date from emp\_name\_fixed*

**from** pyspark.sql.functions **import** to\_date

emp\_date\_fix **=** emp\_name\_fixed**.**withColumn("hire\_date", to\_date(col("hire\_date"), 'yyyy-MM-dd'))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Add Date Columns*

*# Add current\_date, current\_timestamp, extract year from hire\_date*

**from** pyspark.sql.functions **import** current\_date, current\_timestamp

emp\_dated **=** emp\_date\_fix**.**withColumn("date\_now", current\_date())**.**withColumn("timestamp\_now", current\_timestamp())

emp\_dated**.**show(truncate**=False**)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Drop Null gender records*

emp\_1 **=** emp\_dated**.**na**.**drop()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Fix Null values*

*# select \*, nvl('new\_gender', 'O') as new\_gender from emp\_dated*

**from** pyspark.sql.functions **import** coalesce, lit

emp\_null\_df **=** emp\_dated**.**withColumn("new\_gender", coalesce(col("new\_gender"), lit("O")))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Drop old columns and Fix new column names*

emp\_final **=** emp\_null\_df**.**drop("name", "gender")**.**withColumnRenamed("new\_name", "name")**.**withColumnRenamed("new\_gender", "gender")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

*# Convert date into String and extract date information*

**from** pyspark.sql.functions **import** date\_format

emp\_fixed **=** emp\_final**.**withColumn("date\_year", date\_format(col("timestamp\_now"), "z"))

*# Emp Data & Schema*

emp\_data\_1 **=** [

["001","101","John Doe","30","Male","50000","2015-01-01"],

["002","101","Jane Smith","25","Female","45000","2016-02-15"],

["010","104","Lisa Lee","27","Female","47000","2018-08-01"]

]

emp\_data\_2 **=** [

["011","104","David Park","38","Male","65000","2015-11-01"],

["012","105","Susan Chen","31","Female","54000","2017-02-15"],

["020","102","Grace Kim","32","Female","53000","2018-11-01"]

]

emp\_schema **=** "employee\_id string, department\_id string, name string, age string, gender string, salary string, hire\_date string"

*# Create emp DataFrame*

emp\_data\_1 **=** spark**.**createDataFrame(data**=**emp\_data\_1, schema**=**emp\_schema)

emp\_data\_2 **=** spark**.**createDataFrame(data**=**emp\_data\_2, schema**=**emp\_schema)

*# Show emp dataframe (ACTION)*

emp\_data\_1**.**show()

emp\_data\_2**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# UNION and UNION ALL*

*# select \* from emp\_data\_1 UNION select \* from emp\_data\_2*

emp **=** emp\_data\_1**.**unionAll(emp\_data\_2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Sort the emp data based on desc Salary*

*# select \* from emp order by salary desc*

**from** pyspark.sql.functions **import** desc, asc, col

emp\_sorted **=** emp**.**orderBy(col("salary")**.**asc())

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Aggregation*

*# select dept\_id, count(employee\_id) as total\_dept\_count from emp\_sorted group by dept\_id*

**from** pyspark.sql.functions **import** count

emp\_count **=** emp\_sorted**.**groupBy("department\_id")**.**agg(count("employee\_id")**.**alias("total\_dept\_count"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Aggregation*

*# select dept\_id, sum(salary) as total\_dept\_salary from emp\_sorted group by dept\_id*

**from** pyspark.sql.functions **import** sum

emp\_sum **=** emp\_sorted**.**groupBy("department\_id")**.**agg(sum("salary")**.**alias("total\_dept\_salary"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Aggregation with having clause*

*# select dept\_id, avg(salary) as avg\_dept\_salary from emp\_sorted group by dept\_id having avg(salary) > 50000*

**from** pyspark.sql.functions **import** avg

emp\_avg **=** emp\_sorted**.**groupBy("department\_id")**.**agg(avg("salary")**.**alias("avg\_dept\_salary"))**.**where("avg\_dept\_salary > 50000")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP - unionByName*

*# In case the column sequence is different*

emp\_data\_2\_other **=** emp\_data\_2**.**select("employee\_id", "salary", "department\_id", "name", "hire\_date", "gender", "age")

emp\_data\_1**.**printSchema()

emp\_data\_2\_other**.**printSchema()

emp\_fixed **=** emp\_data\_1**.**unionByName(emp\_data\_2\_other)

emp**.**count()

Out[ ]: 20

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Get unique data*

*# select distinct emp.\* from emp*

emp\_unique **=** emp**.**distinct()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Unique of department\_ids*

*# select distinct department\_id from emp*

emp\_dept\_id **=** emp**.**select("department\_id")**.**distinct()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Window Functions*

*# select \*, max(salary) over(partition by department\_id order by salary desc) as max\_salary from emp\_unique*

**from** pyspark.sql.window **import** Window

**from** pyspark.sql.functions **import** max, col, desc

window\_spec **=** Window**.**partitionBy(col("department\_id"))**.**orderBy(col("salary")**.**desc())

max\_func **=** max(col("salary"))**.**over(window\_spec)

emp\_1 **=** emp**.**withColumn("max\_salary", max\_func)

In [18]:

emp\_1**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Window Functions - 2nd highest salary of each department*

*# select \*, row\_number() over(partition by department\_id order by salary desc) as rn from emp\_unique where rn = 2*

**from** pyspark.sql.window **import** Window

**from** pyspark.sql.functions **import** row\_number, desc, col

window\_spec **=** Window**.**partitionBy(col("department\_id"))**.**orderBy(col("salary")**.**desc())

rn **=** row\_number()**.**over(window\_spec)

emp\_2 **=** emp**.**withColumn("rn", rn)**.**where("rn = 2")

(or)

*# Window function using expr*

*# select \*, row\_number() over(partition by department\_id order by salary desc) as rn from emp\_unique where rn = 2*

**from** pyspark.sql.functions **import** expr

emp\_3 **=** emp**.**withColumn("rn", expr("row\_number() over(partition by department\_id order by salary desc)"))**.**where("rn = 2")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Emp Data & Schema*

emp\_data **=** [

["001","101","John Doe","30","Male","50000","2015-01-01"],

["002","101","Jane Smith","25","Female","45000","2016-02-15"],

["020","102","Grace Kim","32","Female","53000","2018-11-01"]

]

emp\_schema **=** "employee\_id string, department\_id string, name string, age string, gender string, salary string, hire\_date string"

dept\_data **=** [

["101", "Sales", "NYC", "US", "1000000"],

["102", "Marketing", "LA", "US", "900000"],

["107", "Customer Service", "Sydney", "Australia", "950000"]

]

dept\_schema **=** "department\_id string, department\_name string, city string, country string, budget string"

In [3]:

*# Create emp & dept DataFrame*

emp **=** spark**.**createDataFrame(data**=**emp\_data, schema**=**emp\_schema)

dept **=** spark**.**createDataFrame(data**=**dept\_data, schema**=**dept\_schema)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Get number of partitions for dept*

dept**.**rdd**.**getNumPartitions()

Out[7]: 8

*# Repartition of data using repartition & coalesce*

emp\_partitioned **=** emp**.**repartition(4, "department\_id")

emp\_partitioned**.**rdd**.**getNumPartitions()

Out[23]: 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Find the partition info for partitions and reparition*

**from** pyspark.sql.functions **import** spark\_partition\_id

emp\_1 **=** emp**.**repartition(4, "department\_id")**.**withColumn("partition\_num", spark\_partition\_id())

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# INNER JOIN datasets*

*# select e.emp\_name, d.department\_name, d.department\_id, e.salary*

*# from emp e inner join dept d on emp.department\_id = dept.department\_id*

df\_joined **=** emp**.**alias("e")**.**join(dept**.**alias("d"), how**=**"inner", on**=**emp**.**department\_id**==**dept**.**department\_id)

In [35]:

df\_joined**.**select("e.name", "d.department\_id", "d.department\_name", "e.salary")**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# LEFT OUTER JOIN datasets*

*# select e.emp\_name, d.department\_name, d.department\_id, e.salary*

*# from emp e left outer join dept d on emp.department\_id = dept.department\_id*

df\_joined **=** emp**.**alias("e")**.**join(dept**.**alias("d"), how**=**"left\_outer", on**=**emp**.**department\_id**==**dept**.**department\_id)

In [38]:

df\_joined**.**select("e.name", "d.department\_name", "d.department\_id", "e.salary")**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write the final dataset*

df\_joined**.**select("e.name", "d.department\_name", "d.department\_id","e.salary")**.**write**.**format("csv")**.**save("data/output/7/emp\_joined.csv")

In [52]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Bonus TIP*

*# Joins with cascading conditions*

*# Join with Department\_id and only for departments 101 or 102*

*# Join with not null/null conditions*

df\_final **=** emp**.**join(dept, how**=**"left\_outer",

on**=**(emp**.**department\_id**==**dept**.**department\_id) **&** ((emp**.**department\_id **==** "101") **|** (emp**.**department\_id **==** "102"))

**&** (emp**.**salary**.**isNull())

)

In [53]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read a csv file into dataframe*

df **=** spark**.**read**.**format("csv")**.**option("header", **True**)**.**option("inferSchema", **True**)**.**load("data/input/emp.csv")

In [14]:

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# Reading with Schema*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date"

df\_schema **=** spark**.**read**.**format("csv")**.**option("header",**True**)**.**schema(\_schema)**.**load("data/input/emp.csv")

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# Handle BAD records - PERMISSIVE (Default mode)*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date, bad\_record string"

df\_p **=** spark**.**read**.**format("csv")**.**schema(\_schema)**.**option("columnNameOfCorruptRecord", "bad\_record")**.**option("header", **True**)**.**load("data/input/emp\_new.csv")

In [37]:

*# Handle BAD records - DROPMALFORMED*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date"

df\_m **=** spark**.**read**.**format("csv")**.**option("header", **True**)**.**option("mode", "DROPMALFORMED")**.**schema(\_schema)**.**load("data/input/emp\_new.csv")

In [45]:

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# Handle BAD records - FAILFAST*

\_schema **=** "employee\_id int, department\_id int, name string, age int, gender string, salary double, hire\_date date"

df\_m **=** spark**.**read**.**format("csv")**.**option("header", **True**)**.**option("mode", "FAILFAST")**.**schema(\_schema)**.**load("data/input/emp\_new.csv")

In [48]:

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*# BONUS TIP*

*# Multiple options*

\_options **=** {

"header" : "true",

"inferSchema" : "true",

"mode" : "PERMISSIVE" }

df **=** (spark**.**read**.**format("csv")**.**options(**\*\***\_options)**.**load("data/input/emp.csv"))

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read Parquet Sales data*

df\_parquet **=** spark**.**read**.**format("parquet")**.**load("data/input/sales\_total\_parquet/\*.parquet")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read ORC Sales data*

df\_orc **=** spark**.**read**.**format("orc")**.**load("data/input/sales\_total\_orc/\*.orc")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Benefits of Columnar Storage*

*# Lets create a simple Python decorator - {get\_time} to get the execution timings*

*# If you dont know about Python decorators - check out : https://www.geeksforgeeks.org/decorators-in-python/*

**import** time

**def** get\_time(func):

**def** inner\_get\_time() **->** str:

start\_time **=** time**.**time()

func()

end\_time **=** time**.**time()

**return** (f"Execution time: {(end\_time **-** start\_time)**\***1000} ms")

print(inner\_get\_time())

In [16]:

@get\_time

**def** x():

df **=** spark**.**read**.**format("parquet")**.**load("data/input/sales\_data.parquet")

df**.**count()

Execution time: 672.5492477416992 ms

In [17]:

@get\_time

**def** x():

df **=** spark**.**read**.**format("parquet")**.**load("data/input/sales\_data.parquet")

df**.**select("trx\_id")**.**count()

Execution time: 348.848819732666 ms

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# BONUS TIP*

*# RECURSIVE READ*

df\_1 **=** spark**.**read**.**format("parquet")**.**option("recursiveFileLookup", **True**)**.**load("data/input/sales\_recursive/")

df\_1**.**show()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read Single line JSON file*

df\_single **=** spark**.**read**.**format("json")**.**load("data/input/order\_singleline.json")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Read Multiline JSON file*

df\_multi **=** spark**.**read**.**format("json")**.**option("multiLine", **True**)**.**load("data/input/order\_multiline.json")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# With Schema*

\_schema **=** "customer\_id string, order\_id string, contact array<long>"

df\_schema **=** spark**.**read**.**format("json")**.**schema(\_schema)**.**load("data/input/order\_singleline.json")

df\_schema**.**show()

\_schema **=** "contact array<string>, customer\_id string, order\_id string, order\_line\_items array<struct<amount double, item\_id string, qty long>>"

In [21]:

df\_schema\_new **=** spark**.**read**.**format("json")**.**schema(\_schema)**.**load("data/input/order\_singleline.json")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Function from\_json to read from a column*

\_schema **=** "contact array<string>, customer\_id string, order\_id string, order\_line\_items array<struct<amount double, item\_id string, qty long>>"

**from** pyspark.sql.functions **import** from\_json

df\_expanded **=** df**.**withColumn("parsed", from\_json(df**.**value, \_schema))

*# Function to\_json to parse a JSON string*

**from** pyspark.sql.functions **import** to\_json

df\_unparsed **=** df\_expanded**.**withColumn("unparsed", to\_json(df\_expanded**.**parsed))

df\_unparsed**.**select("unparsed")**.**show(truncate**=False**)

*# Get values from Parsed JSON*

df\_1 **=** df\_expanded**.**select("parsed.\*")

In [38]:

**from** pyspark.sql.functions **import** explode

df\_2 **=** df\_1**.**withColumn("expanded\_line\_items", explode("order\_line\_items"))

df\_3 **=** df\_2**.**select("contact", "customer\_id", "order\_id", "expanded\_line\_items.\*")

*# Explode Array fields*

df\_final **=** df\_3**.**withColumn("contact\_expanded", explode("contact"))

df\_final**.**drop("contact")**.**show()

+-----------+--------+------+-------+---+----------------+

|customer\_id|order\_id|amount|item\_id|qty|contact\_expanded|

+-----------+--------+------+-------+---+----------------+

| C001| O101|102.45| I001| 6| 9000010000|

| C001| O101|102.45| I001| 6| 9000010001|

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write the data in parquet format*

emp**.**write**.**format("parquet")**.**save("data/output/11/2/emp.parquet")

In [10]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# View data partition information*

**from** pyspark.sql.functions **import** spark\_partition\_id

emp**.**withColumn("partition\_id", spark\_partition\_id())**.**show()

emp**.**write**.**format("csv")**.**option("header", **True**)**.**save("data/output/11/3/emp.csv")

In [12]:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*# Write the data with Partition to output location*

emp**.**write**.**format("csv")**.**partitionBy("department\_id")**.**option("header", **True**)**.**save("data/output/11/4/emp.csv")

In [16]:

*# Write Modes - append, overwrite, ignore and error*

emp**.**write**.**format("csv")**.**mode("error")**.**option("header", **True**)**.**save("data/output/11/3/emp.csv")

*# Bonus TIP*

*# What if we need to write only 1 output file to share with DownStream?*

emp**.**repartition(1)**.**write**.**format("csv")**.**option("header", **True**)**.**save("data/output/11/5/emp.csv")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Feature Files**

The file, in which Cucumber tests are written, is known as feature files

For Example −

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Feature** | **Feature File name** |
| 1 | User Login | userLogin.feature |
| 2 | Share the Post | sharePost.feature |
| 3 | Create Account | createAccount.feature |
| 4 | Delete Account | deleteAccount.feature |

A simple feature file consists of the following keywords/parts −

* **Feature** − Name of the feature under test.
* **Description** (optional) − Describe about feature under test.
* **Scenario** − What is the test scenario.
* **Given** − Prerequisite before the test steps get executed.
* **When** − Specific condition which should match in order to execute the next step.
* **Then** − What should happen if the condition mentioned in WHEN is satisfied.

**Cucumber hook** allows us to better manage the code workflow and helps us to reduce the code redundancy.

@Before public void setUp(){

driver = new FirefoxDriver();

}

@After public void cleanUp(){

driver.close();

}

Before hook − Set up the webdriver and other prerequisites to run the test.

After hook − Close the webdriver and do the cleanup process.

**Tagged Hooks**

We can also indicate if we want before and after hooks to be executed with a specific tag only.

Example − @Before(@Web).

The same concept of tag logical and/or can be applied with hooks as well.

Example − @Before(@dev,@wip), @Before(@dev,~@wip)

**Cucumber - Tags**

Tag can also be defined at a feature level. Once you define a tag at the feature level, it ensures that all the scenarios within that feature file inherits that tag.

**Feature File:**

@SmokeTest

Scenario:

Given user navigates to Facebook

When I enter Username as "<>" and Password as "<>"

Then the user should be redirected to login retry

**Runner file:**

@RunWith(Cucumber.class)

@Cucumber.Options(format = {"pretty", "html:target/cucumber"}, tags = {"~@SmokeTest"})

**Step Definition File:**

public class cucumberTag {

WebDriver driver = null;

@Given("^user navigates to facebook$")

public void goToFacebook() {

driver = new FirefoxDriver();

driver.navigate().to("https://www.facebook.com/");

}

|  |  |  |  |
| --- | --- | --- | --- |
| **ArrayList** | **LinkedList** | **HashMap** | **HashSet** |
| fast random access | slower for random access | key-value pairs, allowing you to access data by a unique key | collection of unique elements |
| lower insertion and deletion | efficient for insertion and deletion | Allows null keys and values | Does not allow duplicates, Uses a HashMap internally, Allows one null value |
|  |  | Not thread-safe | Thread safe |
|  |  | Faster than Hashtable | Slower than HashMap |

|  |  |  |  |
| --- | --- | --- | --- |
| **List** | **Tuple** | **Set** | **Dictionary** |
| allows duplicate | allows duplicate | not allow duplicate | doesn’t allow duplicate keys |
| mutable | immutable | mutable | mutable |
| ordered | ordered | unordered | ordered |

StringBuffer and StringBuilder are both used for creating mutable sequences of characters in Java, as opposed to the immutable String class.

The key difference between them lies in thread safety.

StringBuffer is synchronized, making it thread-safe,

StringBuilder is not, offering better performance in single-threaded environments.