permonth and per year query:

```
SELECT date_format(timestamp_column, 'yyyy-MM') as month_year, count(*) as count
FROM table_name
GROUP BY month_year
```

From pyspark.sql.types import *

The pyspark.sql.types module in PySpark provides a collection of data types that can be used to define the schema of a DataFrame. Here's a brief explanation of some of the data types available in this module:

StringType: Represents string values
IntegerType: Represents integer values
LongType: Represents long integer values
DoubleType: Represents double precision floating-point values
FloatType: Represents single precision floating-point values
DecimalType: Represents decimal values with fixed precision and scale
TimestampType: Represents timestamp values
DateType: Represents date values
BooleanType: Represents boolean values
You can use these data types to define the schema of a DataFrame using the StructType and StructField classes.

For example:

ETL=>Ingestion transfermatuon load reporting:

you can create a DataFrame using the createDataFrame method.

```
cust_data='dbfs:/filestor/data/cust.csv',
emp_data='dbfs:/filestore/data/emp.csv',
dept_data='dbfs:/filestore/data/dept.csv'

=>cust_schema=StructType([StructField('cust_no',IntegerType(),Nullabale=true)])

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("salary", DoubleType(), True)
])
```

RDD:

```
from pysark.sql.tyoes import *
from pyspark.sql import sparksession

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("salary", DoubleType(), True)
])


from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Example").getOrCreate()

data = [("Alice", 25, 50000.0), ("Bob", 30, 60000.0), ("Charlie", 35, 70000.0)]
rdd = spark.sparkContext.parallelize(data)

df = spark.createDataFrame(rdd, schema)
df.show()
```

----

 from pyspark.sql.functions import *
statement imports all the functions available in the pyspark.sql.functions module in PySpark.

This allows you to use these functions in your PySpark SQL queries without having to prefix them with f..

Here are some common functions that can be used with this import statement:

col: returns a Column based on the given column name
lit: creates a Column with a literal value
when: evaluates a condition and returns one of two values based on the result
concat: concatenates two or more columns into a single column
substring: extracts a substring from a column based on a starting index and length
upper: converts a string column to uppercase
lower: converts a string column to lowercase
trim: removes leading and trailing whitespace from a string column
round: rounds a numeric column to a specified number of decimal places
count: counts the number of rows in a DataFrame or the number of non-null values in a specific column
sum: computes the sum of values in a numeric column
avg: computes the average of values in a numeric column
min: finds the minimum value in a numeric or string column
max: finds the maximum value in a numeric or string column

By importing all the functions using from pyspark.sql.functions import *, you can use these and many other functions in your PySpark SQL queries. However, it's important to note that importing all functions can lead to naming conflicts if you have functions with the same name as those in the module. In such cases, it may be better to import functions individually to avoid any potential issues.

```
from pyspark.sql.functions import *
```

each examples
Sure, here are some examples of how to use some of the functions available in pyspark.sql.functions:

col: Returns a Column based on the given column name.
sql

```
from pyspark.sql.functions import col

# Create a DataFrame with a column named "name"
data = [("Alice", 25), ("Bob", 30), ("Charlie", 35)]
df = spark.createDataFrame(data, ["name", "age"])

# Use "col" to reference the "name" column in a PySpark SQL query
df.select(col("name")).show()
---------------------------------------------------------------------------------------------
```

when: Evaluates a condition and returns one of two values based on the result.

```
from pyspark.sql.functions import when

# Create a DataFrame with columns named "age" and "is_adult"

data = [(25, True), (17, False), (30, True)]
df = spark.createDataFrame(data, ["age", "is_adult"])

# Use "when" to create a new column named "adult_status"
# If "is_adult" is True, set "adult_status" to "Adult"
# If "is_adult" is False, set "adult_status" to "Minor"

df.select(
    "age",
    "is_adult",
    when(col("is_adult"), "Adult").otherwise("Minor").alias("adult_status")
).show()
----------------------------------------------------------------------------------------------------
```

concat: Concatenates two or more columns into a single column.
sql

```
from pyspark.sql.functions import concat
```

```
# Create a DataFrame with columns named "first_name" and "last_name"

data = [("Alice", "Smith"), ("Bob", "Johnson"), ("Charlie", "Brown")]
df = spark.createDataFrame(data, ["first_name", "last_name"])

# Use "concat" to create a new column named "full_name"
# Concatenate the "first_name" and "last_name" columns with a space in between

df.select(concat(col("first_name"), lit(" "), col("last_name")).alias("full_name")).show()
-----------------------------------------------------------------------------------------------
substring: Extracts a substring from a column based on a starting index and length.
sql

from pyspark.sql.functions import substring

# Create a DataFrame with a column named "name"

data = [("Alice Smith"), ("Bob Johnson"), ("Charlie Brown")]
df = spark.createDataFrame(data, ["name"])

# Use "substring" to create a new column named "last_name"
# Extract the last name from the "name" column, assuming it is separated by a space

df.select(substring(col("name"), instr(col("name"), " ") + 1, length(col("name"))).alias("last_name")).show()
------------------------------------------------------------------------------------------------------------------------
ETL-CONCEPT:
------------
#ETL--concepts etc

from pyspark.sql.types import *
from pyspark.sql.functions import col

data_shopping_cust = 'dbfs:/FileStore/Shopping_CustomerData.csv'
data_shopping_index = 'dbfs:/FileStore/Shopping_ShoppingIndexData.csv'

cust_schema = StructType([StructField('CustomerAge', IntegerType(), nullable=True)])
index_schema = StructType([StructField('Kolkata', IntegerType(), nullable=True)])

df = spark.read.option('header', 'true').csv(path=data_shopping_cust, schema=cust_schema).show()
df1 = spark.read.option('header', 'true').csv(path=data_shopping_index, schema=index_schema).show()
-----------------------------------------------------------------------------------------------------------------
partitions:
-----------
from pyspark.sql.functions import year

df = spark.read.format('csv').option('header', 'true').load('dbfs:/FileStore/Shopping_CustomerData.csv')
df = df.withColumn('year', year('AnnualIncome',))
df.write.partitionBy('year').mode('overwrite').parquet('/dbfs:/FileStore/output.csv')
---------------------------------------------------------------------------------------------------------------
Using hash:
----------
df_partion = spark.read.format('csv').option('header', 'true').load('dbfs:/FileStore/Shopping_CustomerData.csv')
df_partion  = df.repartition(4)
df_partion .write.mode('overwrite').parquet('File//path/to/output')
df_partion .show()
-------------------------------------------------------------
partition:
----------
from pyspark.sql.types import *
```

```
# Write the DataFrame to disk, partitioned by "Channel_Name" and "Genre"
df.write.mode("overwrite").partitionBy("Channel_Name",
"Genre").csv("dbfs:/FileStore/output")

# Read the data back from disk
df2 = spark.read.csv("dbfs:/FileStore/output", header=True)

# Show the data
df2.show()
-----------------------------------------------------------------------
# Print the current number of partitions
print('Number of current partitions:', str(df.rdd.getNumPartitions()))

# Reduce the number of partitions to 5 using coalesce
df_reduced = df.coalesce(5)
print('Number of partitions after reducing using coalesce:',
str(df_reduced.rdd.getNumPartitions()))

# Increase the number of partitions to 10 using repartition
df_increased = df.repartition(10)
print('Number of partitions after increasing using repartition:',
str(df_increased.rdd.getNumPartitions()))
----------------------------------------------------------------------------------------------------------
Broad Cast Join:
----------------

from pyspark.sql.functions import broadcast

large_df = spark.read.parquet("path/to/large/dataframe")
small_df = spark.read.parquet("path/to/small/dataframe")

# Mark the small_df for broadcast join
small_df_b = broadcast(small_df)

# Perform the join operation
joined_df = large_df.join(small_df_b, "join_column")
------------------------------------------------------------------------------------------------------
```