```
Chapter 1: Introduction to PySpark

1.1 What is PySpark?
PySpark is a tour-de-force in the world of big data processing. As an interface

Its roots can be traced back to the early 2000s, when researchers at UC Berkeley

In terms of its technical advantages, PySpark offers a highly flexible and scala

PySpark also boasts a rich set of libraries and APIs for big data processing, in

In terms of its community and development, PySpark is an open-source platform th

With PySpark, one can unleash the full potential of Apache Spark on large-scale

1.2 PySpark vs. MapReduce

PySpark and MapReduce, both widely used big data processing frameworks, offer a

MapReduce, as its name suggests, is a data processing framework that uses a map-

PySpark, on the other hand, was developed as a high-level interface for Apache S

When comparing the two frameworks, it is clear that PySpark offers several benef

MapReduce was a pioneering solution for big data processing in the early 2000s,

1.3 PySpark vs. SQL

PySpark and SQL, both widely used technologies for data processing and analysis,

SQL, short for Structured Query Language, is a standard language used for managi

PySpark, on the other hand, was developed as a high-level interface for Apache S

When comparing PySpark and SQL, it is important to consider the benefits that Py

In addition, PySpark offers improved scalability and performance compared to SQL

While SQL is a well-established and relatively simple technology, its limitation

While both PySpark and SQL have their strengths and weaknesses, organizations wi
```

```
  File "<ipython-input-1-c0712ae4515e>", line 1
    Chapter 1: Introduction to PySpark
            ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

Lesson 1: Getting started with PySpark DataFrames

SQL

```
#  Create a SparkSession
#  This step is not necessary in SQL, as you can directly issue SQL commands to

#  Read a CSV file into a Spark DataFrame
CREATE TEMPORARY TABLE data USING csv OPTIONS (path "data.csv", header "true", i

#  Show the first 5 rows of the DataFrame
SELECT * FROM data LIMIT 5;

#  Print the schema of the DataFrame
DESCRIBE data;

#  Get the number of rows in the DataFrame
SELECT COUNT(*) FROM data;

#  Get the number of columns in the DataFrame
#  This step is not necessary in SQL, as you can directly look at the output of

#  Describe the summary statistics of the DataFrame's columns
SELECT MIN(age), MAX(age), AVG(age), SUM(age) FROM data;

#  Check if the DataFrame has any null values
SELECT COUNT(*) FROM data WHERE age IS NULL;
```

Python

```
# Create a SparkSession
spark = SparkSession.builder.appName("SQLtoPySpark").getOrCreate()

# Read a CSV file into a PySpark DataFrame
df = spark.read.csv("data.csv", header=True, inferSchema=True)

# Show the first 5 rows of the DataFrame
df.show(5)

# Print the schema of the DataFrame
df.printSchema()

# Get the number of rows in the DataFrame
df.count()

# Get the number of columns in the DataFrame
len(df.columns)

# Describe the summary statistics of the DataFrame's columns
df.describe().show()

# Check if the DataFrame has any null values
df.filter(df.column.isNull()).count()
```

Lesson 2: Filtering and selecting data in PySpark

SQL

```
#  Filter the DataFrame to only include rows where the column "age" is greater t
SELECT * FROM data WHERE age > 30;

#  Select only the "name" and "age" columns from the DataFrame
SELECT name, age FROM data;

#  Select only the rows where the "income" column is greater than 100000
SELECT name, age, income FROM data WHERE income > 100000;

# Filter the DataFrame to only include rows where the column "age" is greater th
SELECT * FROM data WHERE age > 30;

#  Select only the "name" and "age" columns from the DataFrame
SELECT name, age FROM data;

#  Select only the rows where the "income" column is greater than 100000
SELECT name, age, income FROM data WHERE income > 100000;
```

Python

```
# Filter the DataFrame to only include rows where the column "age" is greater th
df_filtered = df.filter(df.age > 30)

# Show the first 5 rows of the filtered DataFrame
df_filtered.show(5)

# Select only the "name" and "age" columns from the DataFrame
df_selected = df.select(["name", "age"])

# Show the first 5 rows of the selected DataFrame
df_selected.show(5)

# Select only the rows where the "income" column is greater than 100000
df_selected = df.filter(df.income > 100000).select(["name", "age", "income"])

# Show the first 5 rows of the selected DataFrame
df_selected.
```

Lesson 3: Grouping and aggregating data in PySpark

SQL

```
# Group the DataFrame by the "age" column and count the number of occurrences of
SELECT age, COUNT(*) FROM data GROUP BY age;

# Group the DataFrame by the "income" column and calculate the average age for e
SELECT income, AVG(age) FROM data GROUP BY income;

# Group the DataFrame by the "age" column and calculate the total income for eac
SELECT age, SUM(income) FROM data GROUP BY age;
```

Python

```
# Group the DataFrame by the "age" column and count the number of occurrences of

grouped = df.groupBy("age").count()

# Show the first 5 rows of the grouped DataFrame

grouped.show(5)

# Group the DataFrame by the "income" column and calculate the average age for e

grouped = df.groupBy("income").agg(avg("age"))

# Show the first 5 rows of the grouped DataFrame

grouped.show(5)

# Group the DataFrame by the "age" column and calculate the total income for eac

grouped = df.groupBy("age").agg(sum("income"))

# Show the first 5 rows of the grouped DataFrame

grouped.show(5)
```

Lesson 4: Joining DataFrames in PySpark

SQL

```sql
-- Join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
JOIN customer ON data.customer_id = customer.customer_id;

-- Inner join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
JOIN customer ON data.customer_id = customer.customer_id
WHERE data.age > 30;

-- Left join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
LEFT JOIN customer ON data.customer_id = customer.customer_id;

-- Right join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
RIGHT JOIN customer ON data.customer_id = customer.customer_id;

-- Outer join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
FULL OUTER JOIN customer ON data.customer_id = customer.customer_id;
```

Python

```
# Join the data table with the customer table on the customer_id column
joined = df_data.join(df_customer, df_data["customer_id"] == df_customer["custom

# Show the first 5 rows of the joined DataFrame
joined.show(5)

# Inner join the data table with the customer table on the customer_id column
inner_joined = df_data.join(df_customer, df_data["customer_id"] == df_customer["

# Show the first 5 rows of the inner joined DataFrame
inner_joined.show(5)

# Left join the data table with the customer table on the customer_id column
left_joined = df_data.join(df_customer, df_data["customer_id"] == df_customer["c

# Show the first 5 rows of the left joined DataFrame
left_joined.show(5)

# Right join the data table with the customer table on the customer_id column
right_joined = df_data.join(df_customer, df_data["customer_id"] == df_customer["

# Show the first 5 rows of the right joined DataFrame
right_joined.show(5)

# Outer join the data table with the customer table on the customer_id column
outer_joined = df_data.join(df_customer, df_data["customer_id"] == df_customer["

# Show the first 5 rows of the outer joined DataFrame
outer_joined.show(5)
```

Lesson 5: Advanced Joining in PySpark

SQL

```
-- Join the data table with the customer table on multiple columns (customer_id
SELECT data.*, customer.age
FROM data
JOIN customer ON data.customer_id = customer.customer_id AND data.name = custome

-- Left join the data table with the customer table on multiple columns (custome
SELECT data.*, customer.age
FROM data
LEFT JOIN customer ON data.customer_id = customer.customer_id AND data.name = cu

-- Right join the data table with the customer table on multiple columns (custom
SELECT data.*, customer.age
FROM data
RIGHT JOIN customer ON data.customer_id = customer.customer_id AND data.name = c

-- Outer join the data table with the customer table on multiple columns (custom
SELECT data.*, customer.age
FROM data
FULL OUTER JOIN customer ON data.customer_id = customer.customer_id AND data.nam
```

Python

```
# Join the data table with the customer table on multiple columns (customer_id a
joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer["custo

# Show the first 5 rows of the joined DataFrame
joined.show(5)

# Left join the data table with the customer table on multiple columns (customer
left_joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer["

# Show the first 5 rows of the left joined DataFrame
left_joined.show(5)

# Right join the data table with the customer table on multiple columns (custome
right_joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer[

# Show the first 5 rows of the right joined DataFrame
right_joined.show(5)

# Outer join the data table with the customer table on multiple columns (custome
outer_joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer[

# Show the first 5 rows of the outer joined DataFrame
outer_joined.show(5)
```

SQL

```
SELECT departments.department, employees.name, employees.salary
FROM
(SELECT department, AVG(salary) as avg_salary
      FROM employees
      GROUP BY department) departments
JOIN employees
ON departments.department = employees.department
WHERE departments.avg_salary > 75000 AND employees.salary > 80000;
```

Python

```
from pyspark.sql.functions import avg

df = spark.read.csv("employees.csv", header=True)

# Compute average salary by department
avg_salary = df.groupBy("department").agg(avg("salary").alias("avg_salary"))

# Filter departments with average salary greater than $75,000
departments = avg_salary.filter(avg_salary["avg_salary"] > 75000)

# Join with employees table and filter employees with salary greater than $80,00
result = departments.join(df, "department").filter(df["salary"] > 80000)

result.show()
```

SQL

```
WITH department_salary AS (
  SELECT department, AVG(salary) as avg_salary
  FROM employees
  GROUP BY department
),
high_salary_department AS (
  SELECT department
  FROM department_salary
  WHERE avg_salary > 75000
)
SELECT employees.department, employees.name, employees.salary, department_salary
FROM employees
JOIN high_salary_department
ON employees.department = high_salary_department.department
JOIN department_salary
ON employees.department = department_salary.department
WHERE employees.salary > 80000;
```

Python

```
from pyspark.sql.functions import avg

df = spark.read.csv("employees.csv", header=True)

# Compute average salary by department
department_salary = df.groupBy("department").agg(avg("salary").alias("avg_salary

# Filter departments with average salary greater than $75,000
high_salary_department = department_salary.filter(department_salary["avg_salary"

# Join with employees table and filter employees with salary greater than $80,00
result = df.join(high_salary_department, "department").join(department_salary, "

result.show()
```

Optimising Joins in PySpark

(1) Broadcast Join: When joining a small DataFrame with a large DataFrame, you can use broadcast join to improve performance. Broadcast join broadcasts the smaller DataFrame to each worker node so that it can be used by the join operation without having to be transferred across the network. You can use the broadcast function from the pyspark.sql.functions module to achieve this.

```
# Broadcast join

from pyspark.sql.functions import broadcast

# Use broadcast join when joining a small DataFrame with a large DataFrame
broadcast_joined = df_large.join(broadcast(df_small), "id")

# Show the first 5 rows of the broadcast joined DataFrame
broadcast_joined.show(5)
```

(2) Sort Merge Join: Use sort merge join when joining two large DataFrames on a common key. This is because sort merge join sorts both DataFrames on the join key and then performs the join. Sort merge join is particularly effective when both DataFrames are sorted on the join key.

```
# Sort merge join
sort_merge_joined = df_large1.join(df_large2, "id").sortWithinPartitions("id")

# Show the first 5 rows of the sort merge joined DataFrame
sort_merge_joined.show(5)
```

(3) Shuffle Hash Join: Use shuffle hash join when joining two large DataFrames that have a large number of unique join keys. This is because shuffle hash join uses a hash partitioner to distribute the data evenly among worker nodes. Shuffle hash join is effective when the join keys are well distributed among the partitions.

```
# Shuffle hash join
shuffle_hash_joined = df_large1.join(df_large2, "id").repartition(1000, "id").so

# Show the first 5 rows of the shuffle hash joined DataFrame
shuffle_hash_joined.show(5)
```

(4) Tune Parallelism: When joining large DataFrames, it's important to set the right number of partitions and parallelism level to achieve the best performance. You can use the repartition method to control the number of partitions and the spark.default.parallelism configuration property to control the parallelism level.

```
# Repartition DataFrames
df_large1 = df_large1.repartition(1000, "id")
df_large2 = df_large2.repartition(1000, "id")

# Set parallelism level
spark.conf.set("spark.default.parallelism", 1000)
```

(5) Predicate Pushdown: Predicate pushdown is a technique to improve the performance of joins by reducing the amount of data that needs to be processed. Predicate pushdown works by pushing the filtering conditions down to the source system to reduce the amount of data that is read into Spark. You can use the filter method in PySpark to apply filtering conditions before the join operation.

```
# Apply filtering conditions before join
filtered_df_large1 = df_large1.filter(df_large1.column_a > 10)
filtered_df_large2 = df_large2.filter(df_large2.column_b < 20)

# Join filtered DataFrames
filtered_joined = filtered_df_large1.join(filtered_df_large2, "id")

# Show the first 5 rows of the filtered joined DataFrame

filtered_joined.show(5)
```

(6) Use Bucketing: Bucketing is a technique used to optimize joins by partitioning data into smaller groups, or buckets, based on a hash value of one or more columns. When two DataFrames are bucketed on the same columns, Spark can efficiently perform joins between the buckets without having to shuffle the data.

```
# Bucket the DataFrames

bucketed_df_large1 = df_large1.write.bucketBy(1000, "id").sortBy("id").saveAsTab
bucketed_df_large2 = df_large2.write.bucketBy(1000, "id").sortBy("id").saveAsTab

# Load the bucketed DataFrames into Spark

bucketed_df_large1 = spark.table("bucketed_df_large1")
bucketed_df_large2 = spark.table("bucketed_df_large2")

# Join the bucketed DataFrames

bucketed_joined = bucketed_df_large1.join(bucketed_df_large2, "id")

# Show the first 5 rows of the bucketed joined DataFrame

bucketed_joined.show(5)
```

(7) Use Broadcast Variables: Broadcast variables allow you to cache a read-only variable on each node in a cluster, reducing the amount of data that needs to be transferred over the network during a join. When joining a large DataFrame with a small DataFrame, you can use a broadcast variable to cache the small DataFrame on each node, improving the performance of the join.

```
# Convert the small DataFrame to a broadcast variable
broadcast_df_small = sc.broadcast(df_small.collect())

# Join the large DataFrame with the broadcast variable
broadcast_joined = df_large.rdd.map(lambda x: (x.id, x)).leftOuterJoin(broadcast

# Convert the result back to a DataFrame
broadcast_joined_df = broadcast_joined.toDF(df_large.columns)

# Show the first 5 rows of the broadcast joined DataFrame
broadcast_joined_df.show(5)
```

(8) Use Caching: Caching is a technique used to store the result of a query or computation in memory, allowing you to reuse it multiple times without having to recompute it each time. You can use caching to improve the performance of joins by reducing the amount of data that needs to be read from disk.

```
# Cache the large DataFrames
df_large1.cache()
df_large2.cache()

# Join the cached DataFrames
cached_joined = df_large1.join(df_large2, "id")

# Show the first 5 rows of the cached joined DataFrame
cached_joined.show(5)
```

More complex examples: SQL to Pyspark to Optimised Pyspark

Example 1

SQL

```
SELECT c.id, c.name, c.age, c.city, c.country, o.order_id, o.product, o.quantity
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE c.age > 30 AND c.country = 'USA' AND o.price > 100
GROUP BY c.id, o.product
HAVING COUNT(o.product) > 1
ORDER BY o.price DESC
LIMIT 10;
```

PySpark equivalent:

```
from pyspark.sql import SparkSession

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

customers = spark.read.csv("customers.csv", header=True)
orders = spark.read.csv("orders.csv", header=True)

join_query = customers.join(orders, customers.id == orders.customer_id) \
    .filter((customers.age > 30) & (customers.country == "USA") & (orders.price
    .groupBy(customers.id, orders.product) \
    .agg({"*": "count"}) \
    .filter("count(1) > 1") \
    .select(customers.id, customers.name, customers.age, customers.city, custome
            orders.order_id, orders.product, orders.quantity, orders.price, orde
    .orderBy(orders.price.desc()) \
    .limit(10)

join_query.show()
```

More optimized PySpark implementation:

```
from pyspark.sql import SparkSession

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

customers = spark.read.csv("customers.csv", header=True).filter("age > 30").filt
orders = spark.read.csv("orders.csv", header=True).filter("price > 100")

join_query = customers.join(orders, customers.id == orders.customer_id) \
    .groupBy(customers.id, orders.product) \
    .agg({"*": "count"}) \
    .filter("count(1) > 1") \
    .select(customers.id, customers.name, customers.age, customers.city, custome
            orders.order_id, orders.product, orders.quantity, orders.price, orde
    .orderBy(orders.price.desc()) \
    .limit(10)

join_query.show()
```

More complex examples: SQL to Pyspark to Optimised Pyspark

Example 2 - subqueries

SQL

```
SELECT *
FROM table1 t1
JOIN (
    SELECT col1, MAX(col2) as max_col2
    FROM table2
    GROUP BY col1
) t2
ON t1.col1 = t2.col1
JOIN (
    SELECT col3, SUM(col4) as sum_col4
    FROM table3
    GROUP BY col3
) t3
ON t1.col3 = t3.col3
WHERE t1.col5 = 'some value'
```

Here's the equivalent PySpark code using DataFrames and SQL functions:

```
from pyspark.sql import functions as F

t1 = spark.table("table1")
t2 = spark.table("table2").groupBy("col1").agg(F.max("col2").alias("max_col2"))
t3 = spark.table("table3").groupBy("col3").agg(F.sum("col4").alias("sum_col4"))

t1.join(t2, on="col1", how="inner").join(t3, on="col3", how="inner")\
    .filter(col("col5") == "some value").select("*").show()
```

Here's an optimized version of the PySpark code that leverages broadcast joins:

```
from pyspark.sql import functions as F

t1 = spark.table("table1")
t2 = spark.table("table2").groupBy("col1").agg(F.max("col2").alias("max_col2"))
t3 = spark.table("table3").groupBy("col3").agg(F.sum("col4").alias("sum_col4"))

t1.createOrReplaceTempView("t1")
t2.createOrReplaceTempView("t2")
t3.createOrReplaceTempView("t3")

spark.sql("""
SELECT *
FROM t1
JOIN t2
ON t1.col1 = t2.col1
JOIN t3
ON t1.col3 = t3.col3
WHERE t1.col5 = 'some value'
""").show()
```

Example 3 - with subqueries and window functions

SQL

```
# Advanced SQL query with subqueries and window functions
SELECT
  customer_id,
  order_date,
  order_total,
  -- Subquery to calculate the average order total for each customer
  (SELECT AVG(order_total)
   FROM orders
   WHERE customer_id = o.customer_id) avg_order_total,
  -- Window function to calculate the running total of order_total for each cust
  SUM(order_total) OVER (PARTITION BY customer_id ORDER BY order_date) running_t
FROM orders o
```

PySpark

```
from pyspark.sql.functions import avg, sum
from pyspark.sql.window import Window

# Define window function
window = Window.partitionBy("customer_id").orderBy("order_date")

# Calculate running total using window function
df_running_total = df_orders.select(
  "customer_id",
  "order_date",
  "order_total",
  sum("order_total").over(window).alias("running_total")
)

# Join with average order total
df_result = df_running_total.join(
  df_orders.groupBy("customer_id").agg(avg("order_total").alias("avg_order_total
  on="customer_id"
)
```

More optimized PySpark

```
from pyspark.sql.functions import avg, sum
from pyspark.sql.window import Window
from pyspark.sql.functions import broadcast

# Calculate average order total using SparkSQL
avg_order_total = df_orders.groupBy("customer_id").agg(avg("order_total").alias(

# Broadcast average order total data
broadcastAvgOrderTotal = broadcast(avg_order_total)

# Define window function
window = Window.partitionBy("customer_id").orderBy("order_date")

# Calculate running total using window function
df_running_total = df_orders.select(
  "customer_id",
  "order_date",
  "order_total",
  sum("order_total").over(window).alias("running_total")
)

# Join with average order total using broadcast variable
df_result = df_running_total.join(
  broadcastAvgOrderTotal,
  on="customer_id"
)
```

Example 4 - using CTEs

SQL

```
WITH recursive_cte (id, name, parent_id, level) AS (
  SELECT id, name, parent_id, 1
  FROM employees
  WHERE parent_id IS NULL
  UNION ALL
  SELECT e.id, e.name, e.parent_id, r.level + 1
  FROM employees e
  JOIN recursive_cte r ON e.parent_id = r.id
)
SELECT id, name,
  CASE WHEN level = 1 THEN 'Manager'
       WHEN level = 2 THEN 'Team Lead'
       ELSE 'Employee' END AS role
FROM recursive_cte;
```

PySpark code:

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window

employees_df = spark.createDataFrame([
    (1, 'John', None),
    (2, 'Jane', 1),
    (3, 'Jim', 2),
    (4, 'Joan', 3),
    (5, 'Jake', 4)
], ['id', 'name', 'parent_id'])

window_spec = Window.orderBy('level')

result_df = employees_df.select('id', 'name', 'parent_id') \
    .union(employees_df.filter(F.col('parent_id').isNull()) \
           .withColumn('level', F.lit(1)) \
           .select('id', 'name', 'parent_id', 'level'))

for i in range(2, 20):
    result_df = result_df.union(result_df \
        .filter(F.col('level') == i - 1) \
        .join(employees_df, result_df.id == employees_df.parent_id) \
        .select(employees_df.id, employees_df.name, employees_df.parent_id, F.li

result_df = result_df.withColumn('role', F.when(F.col('level') == 1, 'Manager')
                                 .when(F.col('level') == 2, 'Team Lead') \
                                 .otherwise('Employee'))
result_df.show()
```

Optimized PySpark code:

```
from pyspark.sql import functions as F

employees_df = spark.createDataFrame([
    (1, 'John', None),
    (2, 'Jane', 1),
    (3, 'Jim', 2),
    (4, 'Joan', 3),
    (5, 'Jake', 4)
], ['id', 'name', 'parent_id'])

result_df = employees_df.filter(F.col('parent_id').isNull()) \
    .withColumn('level', F.lit(1)) \
    .select('id', 'name', 'parent_id', 'level')

for i in range(2, 20):
    result_df = result_df.union
```

Example 5 - using pivots

SQL

```
WITH data_cte AS (
    SELECT
        order_id,
        product,
        order_date,
        CASE
            WHEN product = 'book' THEN 'books'
            WHEN product = 'music' THEN 'music'
            ELSE 'others'
        END AS category
    FROM orders
),
pivot_cte AS (
    SELECT
        order_id,
        order_date,
        SUM(CASE WHEN category = 'books' THEN 1 ELSE 0 END) AS books,
        SUM(CASE WHEN category = 'music' THEN 1 ELSE 0 END) AS music,
        SUM(CASE WHEN category = 'others' THEN 1 ELSE 0 END) AS others
    FROM data_cte
    GROUP BY order_id, order_date
)
SELECT
    order_date,
    SUM(books) AS total_books,
    SUM(music) AS total_music,
    SUM(others) AS total_others
FROM pivot_cte
GROUP BY order_date
ORDER BY order_date DESC;
```

Equivalent PySpark code:

```
from pyspark.sql.functions import when, sum
from pyspark.sql.window import Window

data = (
    spark.table("orders")
    .select(
        "order_id",
        "product",
        "order_date",
        when(col("product") == "book", "books").when(col("product") == "music",
    )
)

pivot = (
    data.groupBy("order_id", "order_date")
    .agg(
        sum(when(col("category") == "books", 1).otherwise(0)).alias("books"),
        sum(when(col("category") == "music", 1).otherwise(0)).alias("music"),
        sum(when(col("category") == "others", 1).otherwise(0)).alias("others"),
    )
)

result = (
    pivot
    .groupBy("order_date")
    .agg(
        sum(col("books")).alias("total_books"),
        sum(col("music")).alias("total_music"),
        sum(col("others")).alias("total_others"),
    )
    .sort(desc("order_date"))
)
```

Example 6 - using rank() function

SQL

```
WITH ranked_data AS (
  SELECT *, RANK() OVER (PARTITION BY col1 ORDER BY col2 DESC) AS rank_col
  FROM data_table
)
SELECT *
FROM ranked_data
WHERE rank_col <= 5
```

PySpark equivalent:

```
from pyspark.sql.functions import rank, dense_rank

ranked_data = data_table.select("*", dense_rank().over(Window.partitionBy("col1"

top_5 = ranked_data.filter(ranked_data["rank_col"] <= 5)
top_5.show()
```

More optimized PySpark implementation:

```
from pyspark.sql.functions import dense_rank

ranked_data = data_table.select("col1", "col2").sort(col("col2").desc()).groupBy

top_5 = ranked_data.filter(ranked_data["rank_col"] <= 5)
top_5.show()
```

Double-click (or enter) to edit

Colab paid products - Cancel contracts here

0s completed at 12:55