FIT5202 Data processing for Big data¶

Activity: Parallel Search¶

For this tutorial, we will focus on parallel search in Big Data. Thus, the following sections will be done:

1. Review Data partitioning strategies
2. Implement distinct searching functionalities using RDDs:
3. Implement distinct searching functionalities using Spark SQL module: you will use the Spark API to use dataframes and Spark SQL to perform the search functionality as in section 1.

Also, you will need to visualise the parallelism on searching in these APIs and RDD implementation. Furthermore, you will need to look at the Query execution plan done by the Spark Optimizer Engine and understand how internally Spark executes or plans a searching function.

Let's get started.

Table of Contents¶

SparkContext and SparkSession ¶

In [1]:

```
# Import SparkConf class into program
from pyspark import SparkConf

# local[*]: run Spark in local mode with as many working processors as logical cores
on your machine
# If we want Spark to run locally with 'k' worker threads, we can specify as
"local[k]".
master = "local[*]"
# The `appName` field is a name to be shown on the Spark cluster UI page
app_name = "Parallel Search"
# Setup configuration parameters for Spark
spark_conf = SparkConf().setMaster(master).setAppName(app_name)

# Import SparkContext and SparkSession classes
from pyspark import SparkContext # Spark
from pyspark.sql import SparkSession # Spark SQL

# Method 1: Using SparkSession
spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()
sc = spark.sparkContext
sc.setLogLevel('ERROR')

# # Method 2: Getting or instantiating a SparkContext
# sc = SparkContext.getOrCreate(spark_conf)
# sc.setLogLevel('ERROR')
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-3b546fb2ecbe> in <module>
      1 # Import SparkConf class into program
----> 2 from pyspark import SparkConf
      3 
      4 # local[*]: run Spark in local mode with as many working processors as 
logical cores on your machine
      5 # If we want Spark to run locally with 'k' worker threads, we can specify as 
"local[k]".

ModuleNotFoundError: No module named 'pyspark'
```

Data Partitioning ¶

In this first part of the tutorial, we will do a quick review of a few data partitioning strategies which we will need to know for the rest of the topics in this tutorial.

Data partitioning is the fundamental step for parallel search algorithms as parallelism in query and search processing is achieved through data partitioning. In this activity, we will consider the following **three** partitioning strategies:

1. Round-robin data partitioning¶

Round-robin data partitioning is the simplest data partitioning method in which each record in turn is allocated to a processing element (simply processor). Since it distributes the data evenly among all processors, it is also known as "equal-partitioning".

2. Range data partitioning¶

Range data partitioning records based on a given range of the partitioning attribute. For example,the student table is partitioned based on "Last Name" based on the alphabetical order (i.e. A ~ Z).

3. Hash data partitioning¶

Hash data partitioning makes a partition based on a particular attribute using a hash function. The result of a hash function determines the processor where the record will be placed. Thus, all records within a partition have the same hash value.

RDD partitioning ¶

By default, Spark partitions the data using **Random equal partitioning** unless there are specific transformations that uses a different type of partitioning</strong> In the code below, we have defined two functions to implement custom partitioning using **Range Partitioning** and **Hash Partitioning**.

In [ ]:

```
from pyspark.rdd import RDD

#A Function to print the data items in each RDD
#WARNING: this function is only for demo purpose, it should not be used on large 
dataset
```

```
def print_partitions(data):
    if isinstance(data, RDD):
        numPartitions = data.getNumPartitions()
        partitions = data.glom().collect()
    else:
        numPartitions = data.rdd.getNumPartitions()
        partitions = data.rdd.glom().collect()

    print(f"####### NUMBER OF PARTITIONS: {numPartitions}")
    for index, partition in enumerate(partitions):
        # show partition if it is not empty
        if len(partition) > 0:
            print(f"Partition {index}: {len(partition)} records")
            print(partition)
```

In [ ]:

```
#Sample data used for demonstrating the partitioning
list_tutors = [(1,'Aaditya'),(2,'Chinnavit'),(3,'Neha'),(4,'Huashun'),(5,'Mohammad'),
                (10,'Peter'),(11,'Paras'),(12, 'Tooba'),(3, 'David'),(18,'Cheng'),
(9,'Haqqani')]

#Define the number of partitions
no_of_partitions = 4
```

In [ ]:

```
(1,'Aaditya'),(2,'Chinnavit')
(3,'Neha'),(4,'Huashun')
(5,'Mohammad'),(10,'Peter')
(11,'Paras'),(12, 'Tooba')
(3, 'David'),(18,'Cheng')
(9,'Haqqani')
```

Default Partitioning in Spark RDD ¶

In [ ]:

```
# random equal partition
rdd = sc.parallelize(list_tutors, no_of_partitions)
```

In [ ]:

```
print("Number of partitions:{}".format(rdd.getNumPartitions()))
print("Partitioner:{}".format(rdd.partitioner))
print_partitions(rdd)
Number of partitions:4
Partitioner:None
####### NUMBER OF PARTITIONS: 4
Partition 0: 2 records
[(1, 'Aaditya'), (2, 'Chinnavit')]
Partition 1: 4 records
[(3, 'Neha'), (4, 'Huashun'), (5, 'Mohammad'), (10, 'Peter')]
Partition 2: 2 records
[(11, 'Paras'), (12, 'Tooba')]
Partition 3: 3 records
[(3, 'David'), (18, 'Cheng'), (9, 'Haqqani')]
```

TODO: How do you think the data is divided across the partitions by default when no partitoner is specified?

Hash Partitioning in RDD ¶

Hash partitioning uses the formula `partition = hash_function() % numPartitions` to determine which partition data item falls into.

In [ ]:
```
#Hash Function to implement Hash Partitioning
#Just computes the sum of digits
#Example : hash_function(12) produces 3 i.e. 2 + 1
def hash_function(key):
    total = 0
    for digit in str(key):
        total += int(digit)
    return total
```
In [ ]:
```
# hash partitioning
hash_partitioned_rdd = rdd.partitionBy(no_of_partitions, hash_function)
print_partitions(hash_partitioned_rdd)
####### NUMBER OF PARTITIONS: 4
Partition 0: 1 records
[(4, 'Huashun')]
Partition 1: 5 records
[(1, 'Aaditya'), (5, 'Mohammad'), (10, 'Peter'), (18, 'Cheng'), (9, 'Haqqani')]
Partition 2: 2 records
[(2, 'Chinnavit'), (11, 'Paras')]
Partition 3: 3 records
[(3, 'Neha'), (12, 'Tooba'), (3, 'David')]
```
Note: Look at how the data is partitioned. For example, Partition 0 has 1 record, [(4, 'Huashun')]. Here is the step-wise breakdown:

- hash_function(4) = 4
- Partition for the key of 4 is determined by `hash_function(4)%numPartitions` i.e. 4%4=0
- Similarly, for (18,'Cheng'), partition is given by `hash+function(18)%numPartitions` i.e. 9%4=1

Range Partitioning in RDD ¶

This strategy uses a range to distribute the items to respective partitions when the keys fall within the range.

In [ ]:
```
no_of_partitions=4

#Find the size of the elements in RDD
chunk_size = len(list_tutors)/no_of_partitions
#Define a range of values by key to distribute across partitions
#Here for simplicity, we are defining the range i.e. keys from 1-4 to fall in first
partition, 5-9 in second partition and so on
range_arr=[[1,4],[5,9],[10,14],[15,19]]

def range_function(key):
    for index,item in enumerate(range_arr):
        if key >=item[0] and key <=item[1]:
            return index
```
In [ ]:
```
# range partition
range_partitioned_rdd = rdd.partitionBy(no_of_partitions, range_function)
print_partitions(range_partitioned_rdd)
####### NUMBER OF PARTITIONS: 4
Partition 0: 5 records
[(1, 'Aaditya'), (2, 'Chinnavit'), (3, 'Neha'), (4, 'Huashun'), (3, 'David')]
```

```
Partition 1: 2 records
[(5, 'Mohammad'), (9, 'Haqqani')]
Partition 2: 3 records
[(10, 'Peter'), (11, 'Paras'), (12, 'Tooba')]
Partition 3: 1 records
[(18, 'Cheng')]
```

Parallel Search using RDDs ¶

Now we will implement basic search functionalities and visualise the parallelism embedded in Spark to perform these kind of queries.

In this tutorial, you will use a csv dataset **bank.csv**. However, for this tutorial we won't analyse the case study but only perform some search queries with this data

In [ ]:

```
# Using Spark, we can read and load a csv file
# Read csv file and load into an RDD object
bank_rdd = sc.textFile('bank.csv')

# If you want to specify the number of partitions, you can add the number as a second
argument
# bank_rdd = sc.textFile('bank.csv', 10)

## Exploring the data file, we can see that it contains different types of
information
## Some useful information is printed below
print(f"Total partitions: {bank_rdd.getNumPartitions()}")
print(f"Number of lines: {bank_rdd.count()}")

## Each element of the RDD is a line from the file
bank_rdd.take(4)
```

Search in RDDs based on multiple conditions¶

We will focus on only four attributes from the data: age, education, marital and balance for filtering conditions. However, we will display additional information as well.

In [ ]:

```
# 1. Split each line separated by comma into a list
bank_rdd1 = bank_rdd.map(lambda line: line.split(','))
# 2. Remove the header
header = bank_rdd1.first()
bank_rdd1 = bank_rdd1.filter(lambda row: row != header)   #filter out header

# Indices for each attribute we will use
# Filter: age, education, marital, balance = 0, 3, 2, 5
# Display additional: day, month, deposit = 9, 10, 16

# 3. Search the records with balance between 1000 and 2000
bank_rdd1 = bank_rdd1.filter(lambda x: int(x[5])>1000 and int(x[5])<2000)
# 4. Also search the records with primrary or secondary education and age less than
30
bank_rdd1 = bank_rdd1.filter(lambda x: x[3] in ['primary','secondary'] and
int(x[0])<30)
# 5. Also filter with those who are married
bank_rdd1 = bank_rdd1.filter(lambda x: x[2]=='married' )
# 6. Display the previous attributes plus the information of day, month and deposit
bank_rdd1 = bank_rdd1.map(lambda field: (field[0],field[2],field[3],field[5],
                                         field[9],field[10],field[16]))
```

```
# Print how many final records
print(bank_rdd1.count())
```

In [ ]:

```
# Let's see how the data was divided and the data for each partition
numPartitions = bank_rdd1.getNumPartitions()
print(f"Total partitions: {numPartitions}")

# glom(): Return an RDD created by coalescing all elements within each partition into
a list
# WARNING: glom().collect() only works for a small dataset
partitions = bank_rdd1.glom().collect()
for index,partition in enumerate(partitions):
    print(f'------ Partition {index}:')
    for record in partition:
        print(record)
```

TODO: Verify the parallelism in the Spark UI and explore the content. How many jobs have been executed so far?

Searching max/min value of an attribute in an RDD¶

This task will aim to find the record in the dataset that contains the highest value for a given attribute. In this case the attribute chosen is "balance".

In [ ]:

```
# Read csv but now with 4 partitions
bank_rdd_4 = sc.textFile('bank.csv',4)

# Split and remove the header
bank_rdd_4 = bank_rdd_4.map(lambda line: line.split(','))
header = bank_rdd_4.first()
bank_rdd_4 = bank_rdd_4.filter(lambda row: row != header)   #filter out header

# Display the first 3 records
bank_rdd_4.take(3)
```

In [ ]:

```
# Using the RDD function max(), it can be obtained in a single line
result_max_balance = bank_rdd_4.max(key=lambda x: x[5]) # Get max by value in index 5
(balance)
# Print the record obtain with highest balance
print(result_max_balance)
```

In [ ]:

```
# Get record with balance 10576
bank_record = bank_rdd_4.filter(lambda x: x[5]=='10576').collect()
print(bank_record)
```

1. Lab Task: Compare the `result_max_balance` record with the record above (`bank_record`). Was the record obtained previously correct i.e. `result_max_balance`? **Explain what happened.**

In [ ]:

```
result_max_balance = bank_rdd_4.max(key=lambda x: x[5])
```

2. Lab Task: As you noticed in the previous result, the record returned originally (result_max_balance) was incorrect.

Fix the code below that uses the `max()` function to get the record with the correct maximum balance.

To learn more about functions in RDDs, you can look into the next 2 sites:¶

1. http://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations

2. https://s3.amazonaws.com/assets.datacamp.com/blog_assets/PySpark_Cheat_Sheet_Python.pdf

DataFrames in Spark ¶

A DataFrame is a distributed collection of data organized into named columns. It is equivalent to a table in relational database or a dataframe in R/Python but with richer optimizations under the hood. For more information visit : https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html

Creating DataFrames¶

SparkSession provides an easy method `createDataFrame` to create Spark DataFrames. Data can be loaded from csv, json, xml and other sources like local file system or HDFS. More information on : https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

To display the schema, i.e. the structure of the DataFrame, you can use **printSchema()** method.

In [ ]:
```
df = spark.createDataFrame([(1,'Aaditya','A'),(2,'Chinnavit','C'),(3,'Neha','N'),
(4,'Huashun','H'),(5,'Mohammad','M'),
                             (10,'Prajwol', 'P'),(1,'Paras','P'),(1, 'Tooba','T'),(3,
'David','D'),(4,'Cheng','C'),(9,'Haqqani','H')],
                            ['Id','Name','Initial'])

#display the rows of the dataframe
df.show(5)
#view the schema
df.printSchema()
```

Another way to create a DataFrame is use the **spark.read.csv** file to load the data from csv to a DataFrame

In [ ]:
```
df = spark.read.csv("bank.csv",header=True)
```

TODO: Display first 10 rows of the above dataframe. Try out other dataframe methods:

- `df.columns, df.count()`
- `df.describe('column_name').show()`
- **Selecting:**`df.select('column_name').show(), df.select('column_name').distinct().show()`
- **Filtering:**`df.filter(df.column_name == 123).show()`

Partitioning in DataFrames ¶

In [ ]:
```
df = spark.createDataFrame([(1,'Aaditya','A'),(2,'Chinnavit','C'),
(3,'Neha','N'),(4,'Huashun','H'),(5,'Mohammad','M'),
                             (10,'Prajwol', 'P'),(1,'Paras','P'),(1,
'Tooba','T'),(3, 'David','D'),(4,'Cheng','C'),(9,'Haqqani','H')],
                            ['Id','Name','Initial'])
```

In [ ]:
```
# Round-robin data partitioning
df_round = df.repartition(5)
# Range data partitioning
df_range = df.repartitionByRange(3,"Name")
# Hash data partitioning
column_hash = "Id"
df_hash = df.repartition(column_hash)
```

In [ ]:
```
print_partitions(df_round)
```
In [ ]:
```
print_partitions(df_range)
```
In [ ]:
```
print_partitions(df_hash)
```
In [ ]:
```
# Read csv file and load into a dataframe
df = spark.read.csv("bank.csv",header=True)
```
TODO: How many partitions the dataframe have?

3. Lab Task: Implement Range and Hash Partitioning techniques for the new dataset and display the partitions. COMPLETE THE CODE BELOW.

In [ ]:
```
## We can specify how many partitions or what kind of partitioning we want for
a dataframe
# Round-robin data partitioning
df_round =
# Range data partitioning
df_range =
# Hash data partitioning
column_hash =
df_hash =
```

In [ ]:
```
## Print the number of partitions for each dataframe
print(f"----- NUMBER OF PARTITIONS df_round:
{df_round.rdd.getNumPartitions()}")
print(f"----- NUMBER OF PARTITIONS df_range:
{df_range.rdd.getNumPartitions()}")
print(f"----- NUMBER OF PARTITIONS df_hash: {df_hash.rdd.getNumPartitions()}")

## Verifying the number of partitions for the dataframe with hash partitioning
it would indicate 200.
## One important thing is that by default, when the number of partitions are
not indicated,
## The default number of partitions is 200

## However, most of the partitions for df_hash are empty.
```

TODO: Complete the above code to show the values for each partition

In [ ]:
```
## You can verify the partitioning and the query plan when an action is
performed with the function explain()
# Query plan for df_round
df_round.explain()
# Query plan for df_range
df_range.explain()
# Query plan for df_hash
df_hash.explain()
```
Parallel Search using Spark Dataframe ¶

We will perform the same filtering criteria as in section 1. This time the logic won't be implemented by us but just declare by using the functions of the Spark Dataframe API to perform the same queries. Thus, we should obtain the same results as before. Furthermore, now you will need to see in the Spark UI, the RDD DAG Visualisation and the Execution Plan of the queries performed with the function explain() as we did previously.

4. Lab Task: Complete the code in the given cell below to implement the given conditions.

In [ ]:

```
# Using the Spark Dataframe API we can obtain the dataframe for a csv file
# We already created dataframes with different types of partitioning
# Choose one of them to work with and perform the queries made in section 1
from pyspark.sql.functions import col

bank_df = df_round

## The functions to filter in dataframes are similar to the functions in RDD.
Thus, the steps are:
# 1. Search the records with balance between 1000 and 2000
bank_df = bank_df.filter(col("balance")>1000)\
            .filter(col("balance")<2000)
# TODO:
# 2. Also in the same dataframe, search the records with primary or secondary
education and age less than 30
bank_df =

# TODO:
# 3. Also filter with those who are married
bank_df =


# TODO:
# 4. Display the previous attributes plus the information of day, month and
deposit
bank_df =



# 5. Display the records
bank_df.show()
```

In [ ]:

```
#### Query and partition information
print_partitions(bank_df)
#### Execution Plan for query with multiple filter conditions
bank_df.explain()
```

TODO: Repeat the same query with different partitioning strategies **(Round-Robin, Range and Hash)** and compare its query execution plan plus its information in the **Spark UI.**

EXAMPLE: Obtain also the `max/min` as you did in RDDs but now using **Spark DataFrame**. Does it return the same value as in section 1? Also, check its execution plan and the information in Spark UI.

In [ ]:

```
#### SOLUTION:
bank_max_balance =
df_round.selectExpr("int(balance)").selectExpr("max(balance)").collect()
print(bank_max_balance)
```

Parallel Search using SQL language in Spark ¶

Spark SQL¶

Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning). [Read More].

A view can be created from a dataframe in order to use SQL queries to search data. In this section, you will use SQL statements to perform search queries in the views that will be registered from the dataframes we created in the previous section.

In [ ]:

```
# register the original DataFrame as a temp view so that we can query it using
SQL
df.createOrReplaceTempView("df_sql")
filter_sql = spark.sql('''
  SELECT age,education,balance,day,month,deposit
  FROM df_sql
  WHERE balance between 1000 and 2000
  AND education in ('secondary','primary')
  AND age < 30
  AND marital = 'married'
''')
# filter_sql.explain()
filter_sql.collect()
```

EXAMPLE: Obtain also the `max/min` as you did in RDDs and DataFrames, but now using **Spark SQL**. Does it return the same value as in previous cases? Also, check its execution plan and the information in Spark UI.

In [ ]:

```
#### SOLUTION:
max_sql = spark.sql('''
  SELECT MAX(CAST(balance AS INT)) as max_balance
  FROM df_sql
''')
# Check the result obtained
max_sql.collect()
```

Congratulations on finishing this activity!

Having practiced today's activities, we're now ready to embark on a trip of the rest of exiciting FIT5202 activities! See you next week!