Question: Write a Spark program to count the number of user clicks between time intervals.

Submitted by Shramana Sinha, 23f1002703

1. Dataproc Cluster Setup

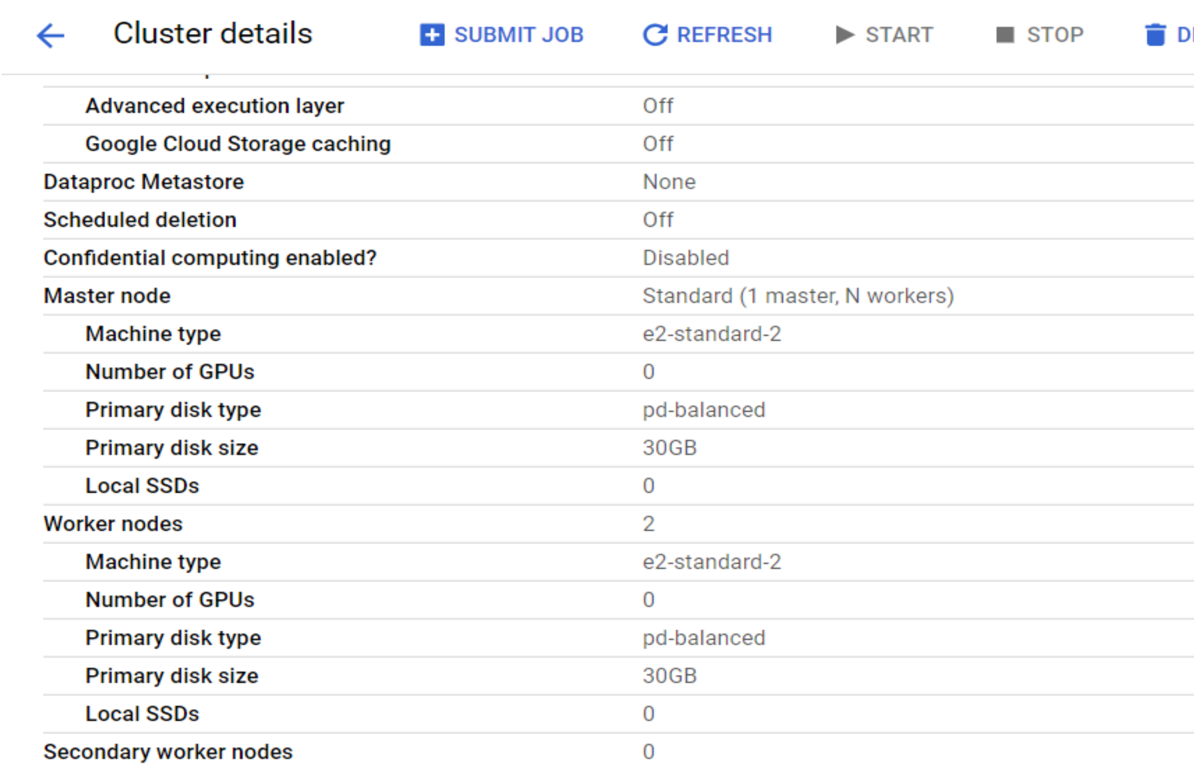
A Dataproc cluster on Compute Engine was created to run the PySpark job, with the following configuration:

Manager Node:

- Machine Series: E2
- Machine Type: e2-standard-2
- Primary Disk Size: 30 GB

Worker Nodes:

- Number of Nodes: 2
- Machine Series: E2
- Machine Type: e2-standard-2
- Primary Disk Size: 30 GB



Advanced execution layer	Off
Google Cloud Storage caching	Off
Dataproc Metastore	None
Scheduled deletion	Off
Confidential computing enabled?	Disabled
Master node	Standard (1 master, N workers)
Machine type	e2-standard-2
Number of GPUs	0
Primary disk type	pd-balanced
Primary disk size	30GB
Local SSDs	0
Worker nodes	2
Machine type	e2-standard-2
Number of GPUs	0
Primary disk type	pd-balanced
Primary disk size	30GB
Local SSDs	0
Secondary worker nodes	0

Screenshot 1: Dataproc Cluster Configuration

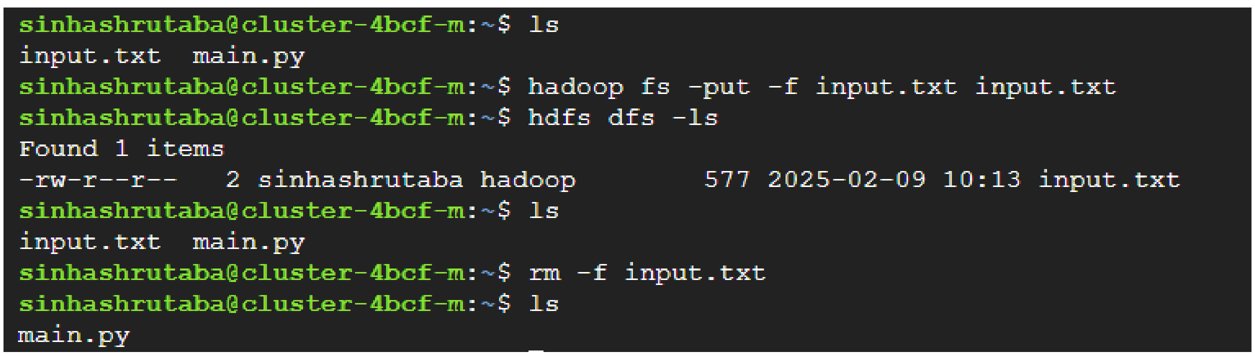
2. File Upload

The required files were uploaded to the cluster's Master Node, using SSH on the cloud console.

- **Files:**
  - `main.py` (PySpark script)
  - `input.txt` (Dataset containing user click data)

- **File Transfer Command:** To Transfer the local file system to hdfs

```
Unset
hadoop fs -put -f input.txt input.txt
```



```
sinhashrutaba@cluster-4bcf-m:~$ ls
input.txt  main.py
sinhashrutaba@cluster-4bcf-m:~$ hadoop fs -put -f input.txt input.txt
sinhashrutaba@cluster-4bcf-m:~$ hdfs dfs -ls
Found 1 items
-rw-r--r--   2 sinhashrutaba hadoop        577 2025-02-09 10:13 input.txt
sinhashrutaba@cluster-4bcf-m:~$ ls
input.txt  main.py
sinhashrutaba@cluster-4bcf-m:~$ rm -f input.txt
sinhashrutaba@cluster-4bcf-m:~$ ls
main.py
```

Screenshot 2: Terminal showing the files in Master Node's local system and hdfs

3. PySpark Script Execution

The PySpark script `main.py` was executed on the cluster's Master Node.

Command:

```
Unset
python3 main.py
```

4. Code Explanation

Import Required Libraries

```
Python
from pyspark.sql import SparkSession
```

```
from pyspark.conf import SparkConf
from pyspark.sql.functions import col, udf
from pyspark.sql.types import StructType, StructField, StringType
```

Create Spark Session

A Spark session is created with a configuration setting to enable the use of `ObjectHashAggregateExec`, ensuring the use of HashAggregate as per the discussed example of the lecture.

```python
conf = SparkConf().set("spark.sql.execution.useObjectHashAggregateExec",
"true")
spark = (
    SparkSession.builder.appName("TimeIntervalClickCounter")
    .config(conf=conf)
    .getOrCreate()
)
```

Define Schema and Read Input File

A schema is defined for reading the input text file. The dataset consists of three columns: `Date`, `Time`, and `UserID`. The data is read from `input.txt` as a tab-separated file with the specified schema.

```python
schema = StructType([
    StructField("Date", StringType(), True),
    StructField("Time", StringType(), True),
    StructField("UserID", StringType(), True),
])
df = spark.read.csv("input.txt", sep="\t", header=True, schema=schema)
```

Define and Apply Hashing Function

A Python function `hash_time` is defined to categorize time into four intervals: `0-6`, `6-12`, `12-18`, and `18-24`. This function extracts the hour portion from the `Time` column and assigns it to one of these intervals. The function is registered as a User-Defined Function (UDF) using `udf()`. Then the function is applied to the data, creating a new column `time_range` which is then used to group and count the number of user clicks for different time intervals.

```python
def hash_time(time):
    hour = int(time.split(":")[0])
    if 0 <= hour < 6:
        return "0-6"
    elif 6 <= hour < 12:
        return "6-12"
    elif 12 <= hour < 18:
        return "12-18"
    else:
        return "18-24"
hash_time_udf = udf(hash_time, StringType())
df_hashed = df.withColumn("time_range", hash_time_udf(col("Time")))
df_counted = df_hashed.groupBy("time_range").count()
```

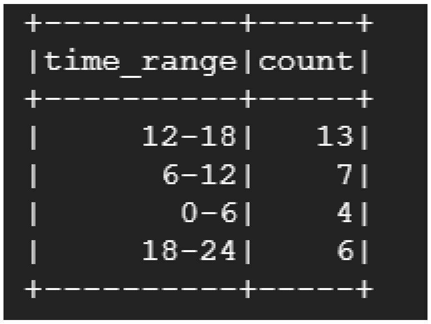
Display and Save Results

The transformed dataset is displayed using `show()`, and `explain(True)` is used to confirm that the query plan utilizes `HashAggregate`. Finally, the results are saved to the `output` directory.

```python
df_counted.show()
df_counted.explain(True)
df_counted.write.mode("overwrite").csv("output")
```

5. Output Verification

The expected output was generated, categorizing clicks into time intervals.



time_range	count
12-18	13
6-12	7
0-6	4
18-24	6

Screenshot 3: Output Results Screenshot

Additionally, `df_counted.explain(True)` confirmed the use of `HashAggregate`.

```
== Parsed Logical Plan ==
'Aggregate ['time_range], ['time_range, count(1) AS count#18L]
+- Project [Date#0, Time#1, UserID#2, hash_time(Time#1)#6 AS time_range#7]
   +- Relation [Date#0,Time#1,UserID#2] csv

== Analyzed Logical Plan ==
time_range: string, count: bigint
Aggregate [time_range#7], [time_range#7, count(1) AS count#18L]
+- Project [Date#0, Time#1, UserID#2, hash_time(Time#1)#6 AS time_range#7]
   +- Relation [Date#0,Time#1,UserID#2] csv

== Optimized Logical Plan ==
Aggregate [time_range#7], [time_range#7, count(1) AS count#18L]
+- Project [pythonUDF0#34 AS time_range#7]
   +- BatchEvalPython [hash_time(Time#1)#6], [pythonUDF0#34]
      +- Project [Time#1]
         +- Relation [Date#0,Time#1,UserID#2] csv

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[time_range#7], functions=[count(1)], output=[time_range#7, count#18L])
   +- Exchange hashpartitioning(time_range#7, 1000), ENSURE_REQUIREMENTS, [plan_id=75]
      +- HashAggregate(keys=[time_range#7], functions=[partial_count(1)], output=[time_range#7, count#29L])
         +- Project [pythonUDF0#34 AS time_range#7]
            +- BatchEvalPython [hash_time(Time#1)#6], [pythonUDF0#34]
               +- FileScan csv [Time#1] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex
(1 paths)[hdfs://cluster-4bcf-m/user/sinhashrutaba/input.txt], PartitionFilters: [], PushedFilters: [], ReadSchem
a: struct<Time:string>
```

Screenshot 4: Execution Plan Showing HashAggregate Usage

6. Comparison with Lecture Video

This implementation follows the hashing example from the [referenced lecture]:

- Data Preparation (Time → time_range via UDF)
- Partial Aggregation (Per-executor counting)
- Shuffle (Exchange by time_range)
- Final Aggregation (Global counts)