Question: Write PySpark code to implement SCD Type II on a customer master data frame.

Submitted by Shramana Sinha, 23f1002703

1. Dataproc Cluster Setup

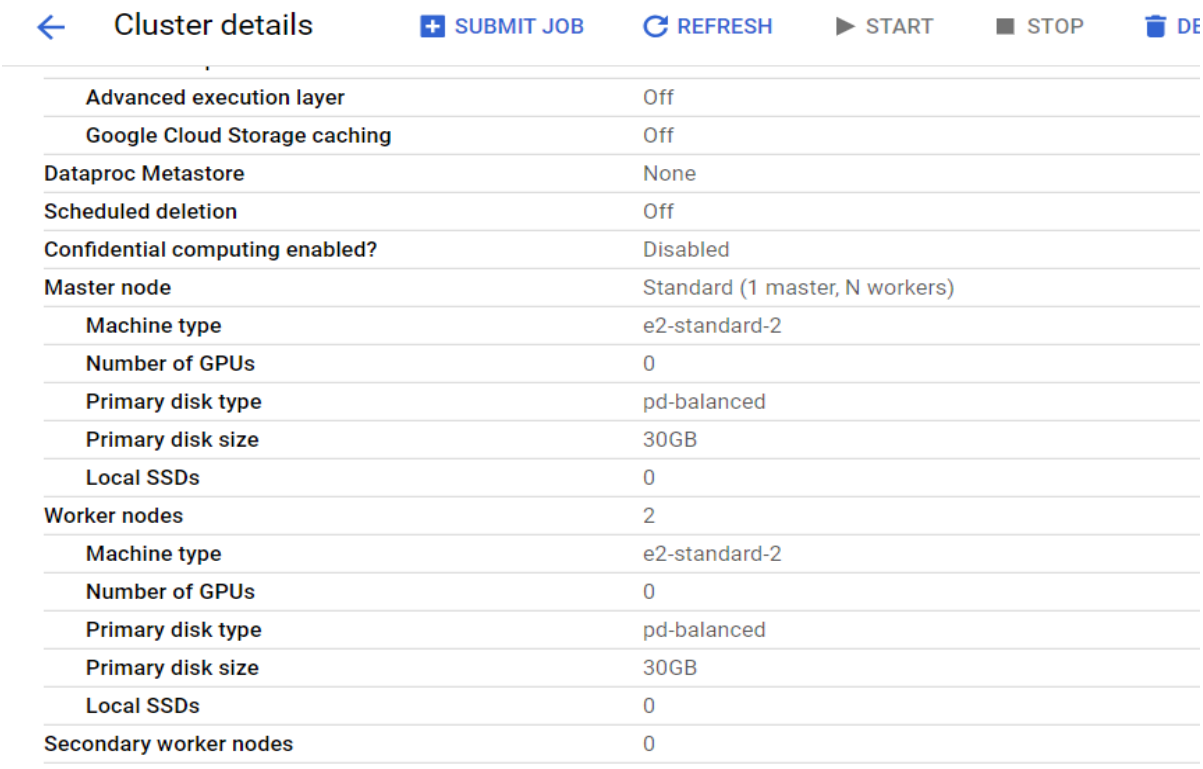
A Dataproc cluster on Compute Engine was created to run the PySpark job, with the following configuration:

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 input files were created based on the example data from the lecture. The following files were uploaded to the cluster's Master Node using SSH via the cloud console:

Files:

- `main.py` – The PySpark script implementing SCD Type II logic.
- `source.csv` – Source dataset containing customer data with `customer_id` as the primary key.
- `target.csv` – Target dataset containing all columns from the source dataset, along with `effective_start_date`, `effective_end_date`, and `record_id` (its own primary key), as required by the SCD Type II implementation.

File Transfer Command: To Transfer the files from local file system to hdfs

```
Unset
hadoop fs -put -f source.csv source.csv
hadoop fs -put -f target.csv target.csv
```

```
sinhashrutaba@cluster-8d8f-m:~$ ls
main.py  source.csv  target.csv
sinhashrutaba@cluster-8d8f-m:~$ hdfs dfs -ls
sinhashrutaba@cluster-8d8f-m:~$ hadoop fs -put -f source.csv source.csv
sinhashrutaba@cluster-8d8f-m:~$ hadoop fs -put -f target.csv target.csv
sinhashrutaba@cluster-8d8f-m:~$ hdfs dfs -ls
Found 2 items
-rw-r--r--   2 sinhashrutaba hadoop        174 2025-02-16 15:29 source.csv
-rw-r--r--   2 sinhashrutaba hadoop        745 2025-02-16 15:30 target.csv
```

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

```
sinhashrutaba@cluster-8d8f-m:~$ hdfs dfs -cat source.csv
customer_id,name,city
1,Name1,City1
2,Name2,City2
3,Name3,City3
4,Name4,City4
5,Name5,City5
6,Name6,City6
7,Name7,City7
8,Name8,City8
9,Name9,City9
10,Name10,City10sinhashrutaba@cluster-8d8f-m:~$
```

Screenshot 3: The source.csv

```
sinhashrutaba@cluster-8d8f-m:~$ hdfs dfs -cat target.csv
record_id,customer_id,name,city,effective_start_date,effective_end_date
1,1,Name1,City1,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
2,2,Name2,City2,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
3,3,Name3,City3,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
4,4,Name4,City4,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
5,5,Name5,City5,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
6,6,Name6,City6,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
7,7,Name7,City7,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
8,8,Name8,City8,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
9,9,Name9,City9,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Z
10,10,Name10,City10,1735-01-01T00:00:00.000Z,9999-12-31T00:00:00.000Zsinhashrutaba@cluster-8d8f-m:~$
```

Screenshot 4: The target.csv

3. Input Data Preparation

The `source.csv` in the hdfs was modified to:

- Delete a row (row with customer id 5).
- Add a new row (row with customer id 11).
- Change details for an existing row (row with customer id 3).

This allowed testing the correctness of the SCD Type II implementation.

```
sinhashrutaba@cluster-8d8f-m:~$ hdfs dfs -cat source.csv
customer_id,name,city
1,Name1,City1
2,Name2,City2
3,Name3,City
4,Name4,City4
6,Name6,City6
7,Name7,City7
8,Name8,City8
9,Name9,City9
10,Name10,City10
11,Name11,City11
```

Screenshot 5: The modified source.csv

4. Spark Job Execution

The spark job was submitted to the cluster.

Command:

```
Unset
spark-submit main.py
```

5. Code Explanation

Import Required Libraries

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql.window import Window
```

Data Loading and Preparation

The source and target data is read from CSV files into DataFrames. The maximum existing record ID of the target column is determined, the current timestamp is captured, and the far-future date for active records is defined.

```python
source_df = spark.read.csv(source_path, header=True, inferSchema=True)
target_df = spark.read.csv(target_path, header=True, inferSchema=True)

max_id = target_df.agg(max(col(unique_id_col))).collect()[0][0]
curr_timestamp = current_timestamp()
max_date = lit("9999-12-31").cast("timestamp")
```

Detect Changes

This step creates a condition to identify changes in tracked columns. It uses a logical OR (|) to combine conditions for each tracked column, effectively detecting if any of these columns have changed between the source and target datasets.

```python
change_condition = None
for column in tracked_columns:
        if change_condition is None:
            change_condition = target_df[column] != source_df[column]
        else:
```

```
        change_condition = change_condition | (
            target_df[column] != source_df[column]
        )
```

Identify Records

Identifying Active Records: This filters the target DataFrame to get only the currently active records. Active records will have their end date set to a far-future date (`max_date`).

```python
active_records = target_df.filter(col(end_date_col) == max_date)
```

Detecting Changed Records: This step uses an inner join between active records and the source data on business keys. The join ensures we're only looking at records that exist in both datasets. The `filter(change_condition)` then applies the previously defined change condition to identify which of these records have actually changed.

```python
changed_records = active_records.join(source_df, business_keys,
"inner").filter(change_condition)
```

Identifying Deleted Records: A left anti-join is used here. This join type returns records from the left dataset (active_records) that do not have a match in the right dataset (source_df). This effectively identifies records that exist in the target but are no longer present in the source, i.e., deleted records. The end date for these records is set to the current timestamp to mark them as no longer active.

```python
deleted_records = active_records.join(source_df.select(*business_keys),
business_keys, "leftanti").withColumn(end_date_col, curr_timestamp)
```

Identifying Unchanged Records: This join operation first combines the business keys of changed and deleted records, then performs a left anti-join with the target DataFrame. This effectively selects all records from the target that are neither changed nor deleted, i.e., the unchanged records.

```python
unchanged_records = target_df.join(
changed_records.select(*business_keys).unionByName(deleted_records.select(*busi
ness_keys)),business_keys,"leftanti")
```

Manage Versions

Expiring Changed Records: This inner join selects active records that match the business keys of changed records. These are the current versions of records that have changes in the source data. Their end date is set to the current timestamp to mark them as expired.

```python
records_to_expire = active_records.join(changed_records.select(*business_keys),
business_keys, "inner").withColumn(end_date_col, curr_timestamp)
```

Creating New Versions of Changed Records: This operation creates new versions for changed records. It joins the source data with the changed records on business keys to get the updated information. The start date is set to the current timestamp, and the end date is set to the far-future date. A new unique ID is assigned using a window function to ensure uniqueness across all records.

```python
w = Window.orderBy(monotonically_increasing_id())
new_changed_records = source_df.join(changed_records.select(*business_keys),
business_keys, "inner").select(*source_df.columns,
curr_timestamp.alias(start_date_col),
max_date.alias(end_date_col)).withColumn(unique_id_col, row_number().over(w) +
max_id)
```

Adding New Records: A left anti-join is used to identify records in the source that don't exist in the target. These are completely new records. They are assigned a start date of the current timestamp, an end date of the far-future date, and a new unique ID.

```python
new_records = source_df.join(target_df.select(*business_keys), business_keys,
"leftanti").select(*source_df.columns, curr_timestamp.alias(start_date_col),
max_date.alias(end_date_col)).withColumn(unique_id_col, row_number().over(w) +
max_id)
```

Final Dataset Assembly

This step combines all the processed record types into a single DataFrame. The `unionByName` operation is used to ensure that columns align correctly across all the different DataFrames being combined.

```python
final_df = (unchanged_records.unionByName(records_to_expire)
    .unionByName(new_changed_records)
    .unionByName(new_records)
    .unionByName(deleted_records))
```

6. Output Verification

The output was displayed in the terminal and written to a CSV file. The output dataset accurately reflected the changes made to the source data:

- The deleted row in the source dataset (row with customer id 5) was correctly marked as expired in the output.
- The new row from the source dataset (row with customer id 11) was added with current timestamps.
- For the modified row (row with customer id 3), the older version was marked as expired, and a new version with updated details was added.

```
+-----------+---------+------+------+--------------------------+--------------------------+
|customer_id|record_id|name  |city  |effective_start_date      |effective_end_date        |
+-----------+---------+------+------+--------------------------+--------------------------+
|1          |1        |Name1 |City1 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|2          |2        |Name2 |City2 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|3          |3        |Name3 |City3 |1735-01-01 00:00:00       |2025-02-16 15:38:18.956131|
|4          |4        |Name4 |City4 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|5          |5        |Name5 |City5 |1735-01-01 00:00:00       |2025-02-16 15:38:18.956131|
|6          |6        |Name6 |City6 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|7          |7        |Name7 |City7 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|8          |8        |Name8 |City8 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|9          |9        |Name9 |City9 |1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|10         |10       |Name10|City10|1735-01-01 00:00:00       |9999-12-31 00:00:00       |
|3          |11       |Name3 |City  |2025-02-16 15:38:18.956131|9999-12-31 00:00:00       |
|11         |12       |Name11|City11|2025-02-16 15:38:18.956131|9999-12-31 00:00:00       |
+-----------+---------+------+------+--------------------------+--------------------------+
```

Screenshot 6: The output of the execution