Rapport: Traitement sur des partitions

Question 1 : Regrouper les films par genre et les compter

1. *f1 = films_extrait.rdd.mapPartitions(count_movies_by_genre).toDF(['genre', 'n'])*
   - `count_movies_by_genre` processes each partition of the `films_extrait` RDD.
   - For every movie, it iterates through its genres, counts occurrences of each genre within the partition, and emits `(genre, count)` pairs.
   - **Partition-level processing**: Each partition processes its data independently before results are aggregated.
2. *f2 = f1.repartition(3, 'genre')*
   - The DataFrame `f1` is repartitioned by the `genre` column using `repartition(3, 'genre')`. This ensures that data related to the same genre is grouped together across fewer partitions (3 partitions in this case).
3. *f3 = f2.rdd.mapPartitions(total_genre).toDF(['genre', 'n'])*
   - `total_genre` processes each partition of `f2`, summing up the counts (`n`) for each genre within the partition. The result is emitted as `(genre, total_count)` pairs.

Question 2 : Numérotation

1. **Counting Rows per Partition**:
   - `notes_extrait.rdd.mapPartitions(part_size_simple).toDF(['size']).collect()` calculates the number of rows in each partition using the `part_size_simple` function.

   The resulting `partition_sizes` list contains the row count for each partition.
2. **Cumulative Offsets Calculation**:
   - `cumulative_sizes` is a list of cumulative row counts up to the beginning of each partition.
   - The first partition always starts with an offset of `0`.
   - For subsequent partitions, the offset is the sum of all row counts in previous partitions.

> Example partition sizes: [100, 150, 200]
> Cumulative sizes: [0, 100, 250] → (100+150=250)

2. **Broadcasting the Offsets**:
   - The `cumulative_sizes` list is broadcasted to all worker nodes using `spark.sparkContext.broadcast`. This ensures that each partition has access to the global offsets without excessive data transfer.
3. **Adding Global Numbering**:
   - `mapPartitionsWithIndex` processes each partition, assigning global numbers to rows.
   - The `partID` argument identifies the current partition's index.
   - For each partition:
     - It retrieves the starting offset (`offsets.value[partID]`).
     - Rows within the partition are numbered sequentially, starting from `offset + 1`.

```
def add_partition_offset(partID, note_iterator: Iterator) ->
Iterator:
    offset = offsets.value[partID]
    i = 1
    for note in note_iterator:
        yield (offset + i, note.nF, note.nU, note.note, not
e.annee)
        i += 1
```

Question 3 : Tri des films par titre

3.1) & 3.2) partition simple

1. **Custom Partitioning by Title (Function `partitioner`)**
   - The `partitioner` function determines the partition index based on the first letter of the film title (`titre`):

```
# a exemple of partition navy
#
def partitioner(titre):
    first_letter = titre[0].upper()
```

```
    if 'A' <= first_letter <= 'E':
        return 0
    elif 'F' <= first_letter <= 'J':
        return 1
    elif 'K' <= first_letter <= 'O':
        return 2
    elif 'P' <= first_letter <= 'T':
        return 3
    else:
        return 4
```

- Then RDD is transformed to include the partition index `p` as a column

2. **Repartitioning by Custom Logic**
   - The `repartitionByRange(5, col('p'))` method ensures that the dataset is physically repartitioned into 5 partitions according to the column `p`.
   - This step redistributes data across the partitions, aligning it with the custom partitioning logic.
3. **Sorting Within Each Partition**
   - `film2.rdd.mapPartitions(sort_partition).toDF(['p', 'nF', 'titre', 'genres'])`

3.3) uniform partition

1. **Calculate Letter Distribution Per Partition**
   - The function `calculate_letter_distribution` calculates the count of titles starting with each letter (`a-z`) within each partition.
   - It iterates through the titles in the partition and increments the count for the first letter of each title.
2. **Aggregate Global Distribution**
   - The local distributions are merged into a single **global letter distribution**:
     - A dictionary `global_distribution` accumulates the counts for each letter.
     - The distribution is then sorted alphabetically by the letter.
3. **Define Partition Boundaries**
   - The `get_boundaries` function calculates the boundaries for new partitions:
     - The total number of titles (`total_count`) is divided equally across the number of partitions (`m = total_count // nb_partitions`).

    - Letters are grouped together into partitions such that the cumulative count of titles in each group is approximately equal to `m`.
    - Each group of letters is assigned a partition ID.
  - exemple of output :

```
Partition 0: a, b, c
Partition 1: d, e, f, g
Partition 2: h, i, j, k, l
Partition 3: m, n, o, p, q, r
```

4. **New Partitioning Logic**

- The `new_partitioner` function uses the calculated boundaries to assign a partition ID to each title:
  - The first letter of the title is compared against the boundary groups to determine its partition ID.
  - Titles with letters outside the defined boundaries are assigned to `1`.
- **Repartition the Dataset :**
  - Titles are repartitioned according to the `new_partitioner` logic and is transformed to include the new partition ID `p`:
  - similar with the method that we applied in Question2

Question 5 : top fréquence

1. **Splitting Titles into Words**
   - The function `split_title` cleans and splits movie titles into individual words:
     - Removes punctuation, numbers, and extra spaces.
     - Converts text to lowercase.
     - Splits the cleaned string into a list of words.
   - The RDD transformation:Produces a DataFrame where each row contains a movie ID (`nF`) and a list of words from its title.
2. **Counting Word Frequencies Locally**
   - The function `count_words` computes the frequency of each word within a single partition:
     - Iterates over the `word_list` of each title.

    - Increments the count for each word in a dictionary.
    - Yields `(word, count)` pairs.
- The RDD transformation:Produces a DataFrame with word frequencies calculated independently for each partition.

3. **Repartitioning by Word**
    - The word-frequency pairs are repartitioned by the `word` column:This ensures that the same word is grouped together in the same partition for aggregation.

---

4. **Aggregating Word Frequencies**
    - The function `count_words_total` combines word frequencies across partitions:
        - Accumulates counts for each word in the partition.
        - Sorts the words by frequency (descending) and retains only the top 5 most frequent words in the partition.
    - The RDD transformation:Produces a DataFrame with the top 5 most frequent words from each partition.

---

5. **Collecting and Finalizing the Top 5 Words**
    - The function `get_words` collects the word-frequency pairs from each partition into a list and these lists are merged across partitions
    - The merged results are sorted globally by frequency, retaining only the top 5 words