Hadoop & HDFS & MapReduce

Exercise 1 (Questions about HDFS)

1. In HDFS (and in a RDBMS) when is a transaction considered committed?

In HDFS, the concept of a transaction is different from that in an RDBMS because HDFS is not a database system, but a distributed file system designed to store large data sets reliably. HDFS does not support transactions in the database sense. However, operations like file writes can be considered "committed" when all the file's blocks have been successfully written to the configured number of DataNodes and the NameNode has been updated with this information, ensuring the file's durability and availability in the distributed system. For file operations, the "commitment" can be thought of as the point where the file is closed and the HDFS metadata is updated to reflect the new or modified file, making it accessible to readers.

2. What is HDFS (acronym, objectives)?

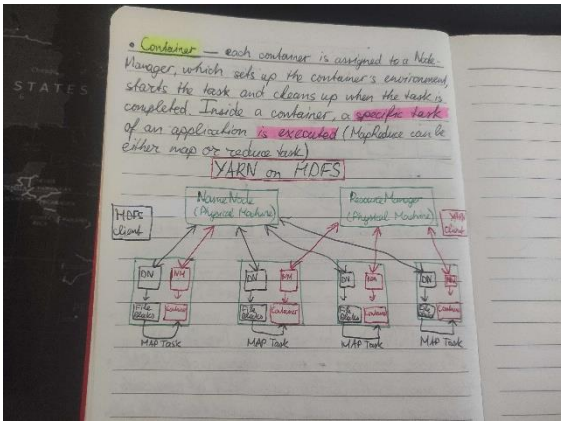
HDFS means Hadoop Distributed File System and its main objective is to manage and store huge data (PB, TB) while optimizing sequential reads. HDFS divides large files into smaller blocks and distributes them across nodes in a cluster.

3. What are the two main components in HDFS? Explain their role.

The two main components in HDFS are: NameNode and DataNode.

- NameNode is the master server in the HDFS architecture that performs the following tasks: Management of the file system's namespace, manages the block placement (creation and replications across DataNodes) and manages the metadata which is stored on disk and in RAM.
- DataNode is the slave noes in the HDFS architecture that contains the actual blocks of data (chunks of data), reads the data from the Disk and communicates with the NameNode by registering with it and sending regular heartbeats, which signals their health and status to the master.

YARN :



4. Explain the difference between fault tolerance and data integrity? How is each implemented in HDFS?

- Fault Tolerance: 3 Copies by default of the same block of data
    - **Definition**: Fault tolerance refers to the ability of a system to continue operating properly in the event of a failure of some of its components. In the context of data storage, it means that the system can handle hardware or software failures without losing data or interrupting service.
    - **Implementation in HDFS**: HDFS achieves fault tolerance primarily through data replication. When data is stored in HDFS, it is divided into blocks, and each block is replicated across multiple DataNodes in the cluster. By default, HDFS creates three replicas for each block. If a DataNode fails, HDFS can use the replicas from other DataNodes to recover the lost data. The system continuously monitors the health of DataNodes and automatically re-replicates the data if the number of replicas falls below the configured replication factor.

- Data Integrity:
    - **Definition:** Data integrity involves ensuring the accuracy, consistency, and reliability of data throughout its lifecycle. It means that the data is not corrupted and remains unaltered unless by authorized operations.
    - **Implementation in HDFS:** HDFS implements data integrity checks through checksums. When a file is written to HDFS, a checksum for each block of the file is computed and stored separately from the actual data. When a block is read, its checksum is recalculated and verified against the stored checksum. If the checksums do not match, it indicates data corruption, and HDFS attempts to fetch another replica of the block from a different DataNode. This mechanism ensures that clients always read correct and uncorrupted data.

5. In HDFS why is the filesystem metadata split in two?

The filesystem metadata is split into two main components for efficiency, scalability, and reliability purposes. These components are NameNode Metadata and Block Metadata (Stored on DataNodes). The primary reasons for this split are:

- **Scalability**: By distributing the block metadata across DataNodes, HDFS can scale to a large number of blocks and files without overwhelming the NameNode. The NameNode focuses on managing the namespace, enabling efficient directory and file lookups.
- **Performance**: Keeping the namespace metadata in memory on the NameNode allows for fast access to the filesystem structure, which is essential for operations like opening files, listing directories, and so on. Meanwhile, distributing block metadata across DataNodes allows parallel processing and data transfers, enhancing the system's overall throughput.

- **Reliability**: Splitting the metadata ensures that a failure in a DataNode affects only the blocks stored on that DataNode and does not impact the overall filesystem's namespace. Replication of data blocks across multiple DataNodes provides fault tolerance for data storage, while separate mechanisms (such as secondary NameNode or CheckpointNode) help in backing up the namespace metadata for recovery purposes.

This design allows HDFS to manage vast amounts of data across a large cluster of machines efficiently, ensuring quick access to file system metadata and the reliability and scalability of data storage.

Exercise 2 (Java Code)

Explain what the code below does. This code is a sample from the NodeManager class in HDFS source code. You must explain what processMisReplicatedBlocks entails in connection to what you know about HDFS.

```
void checkIfClusterIsNowMultiRack(DataNodeDescriptor node) {
        if (!hasClusterEverBeenMultiRack && networktopology.getNumOfRacks() > 1) {
                String message = "DN " + node + " joining cluster has expanded a formerly " + "single-
                rack cluster to be multi-rack. ";
...
        if (...) {
                blockManager.processMisReplicatedBlocks();
        }
        }
}
```

This method is responsible for detecting when an HDFS cluster transitions from a single-rack to a multi-rack configuration upon the addition of a new DataNode and triggers actions to ensure that the data replication across the cluster adheres to the desired fault tolerance and data availability policies. This could involve re-distributing data blocks across the racks to maintain or improve the resilience of the system against rack-level failures.

Exercice 3 (Checksums in HDFS): Not needed.

Exercice 4 (MapReduce: key/value types)

Indicate which key/value pairs must be of the same type below in one (Hadoop) Map/reduce step.

- (k1, v1) → Mapper → (k2, v2)
- (k3, v3) → Combiner → (k4, v4)
- (k5, v5) → Reducer → (k6, v6)

Example of word count: (K1: file, v1: text), (k2: word, v2: 1), (k3: word, v3: 1), (k4: word, v4: [1,1, …]), (k5: word, v5: [1,1, …]), (k6: word, v6: [#word])

Exercise 5 (MapReduce)

1. We suppose that we are given a huge file containing integers. Give a MapReduce program (you must provide the Map / Reduce functions for each step, and the Combiner if it is useful, in pseudocode) to compute:

(a) The same set of integers but display each input only once (elimination of duplicates).

- Map Function:

```
# The function just assigns 1 to each key (number).
map(InputKey file, InputValue numbers) {
        for each number in numbers {
                emit(number, 1);

        }

}
```

- Reduce Function:

```
# The function converts the list of ones into a single one.
reduce(OutputKey number, InputermediateValue list ones) {

        emit(number, 1);

}
```

(b) The greatest integer (just with the map and reduce function, without exploiting the sorting steps in MapReduce jobs).

- Map Function:

```
# The function just assigns 1 (key) to each keys (values = number).
map(InputKey file, InputValue numbers) {
        for each number in numbers {
                emit(1, number);

        }

}
```

- Reduce Function:

```
# The function receives as input the constant key 1, and thanks to the Shuffle
# Phase, all values (numbers) are grouped into a list.
reduce(Integer key, Iterator numbers) {
        max = Min_Integer       # ~-2000000
        for each number in numbers {
                if (number > max) {
                        max = number;
                }
        }

        emit(max);

}
```

2. **We consider as input a file from DBLP (computer science bibliography website) containing a list of papers. Each paper has (among others) a list of authors. We must compute, for each triple of authors (x, y, z), the number of papers that those 3 have co-authored together. When the authors are a superset of (x, y, z) (e.g.: (x, u, v, y, w, z)) this still counts as one co-authored paper. The order of authors does not matter: (z, y, x) is still considered as a co-authored paper by x, y and z.**

How would you compute this with MapReduce?

3. Reverse - engineering:

- function map(key,value):
  emit(key, value);
- function reduce(key, values[]):
  z = 0.0
  for value in values:
  z += value
  emit (key, z / values.length())

(a) What is the job performed here when the input is a list of pairs with type (string, float): write your answer as an SQL query.

The job performed when the input is a list of pairs with (String, Float) type is to return the average value of each key.

```
Select Key, AVG(values)
FROM file
Groupby Key
```

(b) Can we re-use the Reducer as a Combiner?

Yes, we could consider a combiner function which would not only combine (key, value) pairs by their key but also compute the average value for a given mapper, increasing thus the whole efficiency in terms of network traffic. Otherwise, if left like that, the map function via the shuffle function, would send to the reducer function every single key (even duplicates) and then compute the average. By coding the combiner function we would decrease the traffic network.

(c) If you answered yes, would the number of distinct keys impact the efficiency of the combiner (how)? If you answered no, can you change some of the Map and Reduce functions and output format so that the Reducer may be used as Combiner?

The first question has already been answered. If the answer would have been negative, we could have simply changed the code by adding the combiner function with the same coding goal as the reducer, i.e, the combiner functions calculates the average for each key and then sends to the reducer function the pair (key, average) improving the efficiency of the reducer function which, once again needs to compute the average for each key, but in this case, the number of keys has drastically been reduced thanks to the combiner function.

Exercise 6 (MapReduce from SQL)

Decompose into map-reduce steps the following SQL queries. We suppose the relations have at least the attributes mentioned in the queries, we suppose the input of map contains as value an object representing a row of one of the relations involved (the object has one attribute relation, and one attribute per attribute of the relation), and we assume that the key provided as input of map is irrelevant.

1. SELECT a, b
   FROM r1
   WHERE b=3 and c>5

   SQL:

   This query selects records from relation r1 with attributes a, b, and c, where b equals 3 and c is greater than 5.

   Map Function

   The map function processes each row of the relation r1. It filters rows based on the condition b=3 AND c>5. If a row meets this condition, the map function emits a tuple with a key that can be used for sorting or grouping (if necessary) and a value that is the relevant data (a, b) for the final output.

   - **Input**: The input to the map function is a pair (key, value), where the **key** is irrelevant for our purposes (it could be a row identifier or any other piece of data that the MapReduce framework automatically assigns) and the **value** is an object representing a row from the relation r1, with attributes .relation (indicating the source relation, in this case, "r1"), .a, .b, and .c.

   Map Function Pseudocode

   - function map(IrrelevantKey key, RowObject value):

     if value.b == 3 and value.c > 5:

     emit(Key, (value.a, value.b))

2. SELECT a FROM r1
   INTERSECT
   SELECT b AS a FROM r2

   SQL:

   This query returns the common elements from two relations r1 and r2 within the attribute a (for r1) and b (as a) for r2.

   Map Function

During the Map Phase, the map function will transpose the (key, value) pair to return (value, key) and let the shuffle Phase group all common value pairs and then send it to the reduce function to make the final check.

- **Input**: The input to the map function is a pair (key, value), where the **key** is irrelevant for our purpose and the **value** is an object representing a row from the relation, with attributes its corresponding attributes.

Map Function Pseudocode

This key concept is to use the function/method .relation which tells you to which relation belongs to the row.

- function map(Irrelevant key, RowObject value):
  if value.relation == "R_1":
    emit(value.a, "R_1")
  if value.relation == "R_2":
    emit(value.b, "R_1")

- function reduce(Text Key, Iterator list_values):
  key_in_r1 = False
  key_in_r2 = False
  for value in list_values:
    if value == "R_1":
      key_in_r1 = True
    if value == "R_2":
      key_in_r2 = True
  if key_in_r1 and key_in_r2:
    emit(Key)

3. SELECT r1.a, SUM(r1.e)
   FROM r1
   WHERE r1.c AND r1.d >4
   GROUP BY r1.a
   HAVING count(r1.g)>3

SQL:

FROM r1: This specifies that the data is being selected from a table named r1.
WHERE r1.c AND r1.d > 4: This is the filter condition applied to the rows before any grouping or aggregation occurs. It selects rows based on two conditions: **r1.c** is expected to be a Boolean expression and **r1.d > 4**: Only rows where the value of column d is greater than 4 are included.
GROUP BY r1.a: This operation groups the remaining rows based on unique values in column a. Each group will contain rows that have the same value for a.

SELECT r1.a, SUM(r1.e): For each group identified in the previous step, two pieces of information are produced: The value of a (which is the same within each group) and the sum of the values in column e for all rows within the group.
HAVING COUNT(r1.g) > 3: This condition filters the groups created by the GROUP BY operation. It only includes groups where the count of non-null values in column g is greater than 3. Unlike the WHERE clause, which filters rows before the grouping operation, the HAVING clause filters groups after they have been formed based on the GROUP BY operation.

Map Function

During the Map Phase, the map function checks the **WHERE** condition and emits the pair (value.a, RowObject) and lets the shuffle Phase group all common value.a pairs with their corresponding rows, ultimately sending to the reducer function a pair (value.a, list_rows). The reducer function needs to iterate to count the elements of column g, and only then counts the sum of elements in column e, emitting the desired pair.

- **Input**: The input to the map function is a pair (key, value), where the **key** is irrelevant for our purpose and the **value** is an object representing a row from the relation, with attributes its corresponding attributes.

Map Function Pseudocode

- function map(Irrelevant key, RowObject value):
    if value.c and value.d > 4:
        emit(value.a, value)

- function reduce(Text Key, Iterator list_rows):
    sum_e =0
    count_g = 0
    for row in list_rows:
        count_g = count_g + 1
    if count_g > 3:
        for row in list_rows:
            sum_e = sum_e + row.e
        emit(key, sum_e)

4. SELECT r1.a, r2.b, SUM(r1.e)
   FROM r1, r2
   WHERE r1.c = r2.c AND r1.d >4
   GROUP BY r1.a, r2.b
   HAVING count(r1.g)>3

SQL:

FROM r1, r2: This specifies that the data is being selected from 2 tables named r1 and r2.
WHERE r1.c = r2.c AND r1.d > 4: This is the filter condition applied to the rows before any grouping or aggregation occurs. It selects rows based on two conditions: **r1.c** is equal to **r2.c** and **r1.d > 4** where only rows where the value of column d is greater than 4 are included.
GROUP BY r1.a, r2.b: This operation groups the remaining rows based on unique values in column a in r1 and values in column b in r2. Each group will contain rows that have the same value for a and b.

SELECT r1.a, r2.b, SUM(r1.e): For each group identified in the previous step, three pieces of information are produced: The value of a in r1, the value of b in r2 and the sum of the values in column e for all rows within the group.
HAVING COUNT(r1.g) > 3: This condition filters the groups created by the GROUP BY operation. It only includes groups where the count of non-null values in column g is greater than 3. Unlike the WHERE clause, which filters rows before the grouping operation, the HAVING clause filters groups after they have been formed based on the GROUP BY operation.

Map Function

During the Map Phase, the map function checks the **WHERE** condition and emits the pair (value.a, RowObject) and lets the shuffle Phase group all common value.a pairs with their corresponding rows, ultimately sending to the reducer function a pair (value.a, list_rows). The reducer function needs to iterate to count the elements of column g, and only then counts the sum of elements in column e, emitting the desired pair.

- **Input**: The input to the map function is a pair (key, value), where the **key** is irrelevant for our purpose and the **value** is an object representing a row from the relation, with attributes its corresponding attributes.

Map Function Pseudocode

```
def map(key, row):
        if row.relation == 'r1' and row.d > 4:
            emit((row.c, 'r1'), (row.a, row.e, row.g, 1))
        elif row.relation == 'r2':
            emit((row.c, 'r2'), row.b)
def reduce(key, values):
# key es (c, 'r1') o (c, 'r2'), values son los valores emitidos por map
a_values = []
```

```
b_values = []
sum_e = 0
count_g = 0

for value in values:
    if key[1] == 'r1':
        a, e, g, count_indicator = value
        if g > 0:
            count_g += count_indicator

        sum_e += e
        a_values.append(a)

    else:  # key[1] == 'r2'
        b_values.append(value)

# Realizar el join basado en 'c' y combinar 'a' y 'b' con la suma de 'e'

for a in a_values:
        for b in b_values:
            if count_g > 3:
                emit((a, b), sum_e)
```