TP2 : Median computation optimization

0. Setup

0.1 Setup an EMR cluster

- Sign on AWS educate then AWS using your student account.
- Launch an EMR cluster (with as many nodes you want)
- Install Rstudio server on it (use this cookbook)
- Connect to Rstudio server

You will find all the steps to setup your cluster here

Don't forget to turn off your VM !!

0.2 Connect to your spark cluster

▼Use this piece of code to connect to the spark cluster :

```
install.packages("sparklyr")
library(sparklyr)
Sys.setenv(SPARK_HOME="/usr/lib/spark")
sc <- spark_connect(master="yarn-client")
```

0.3 Data

In this practical session, you will reuse the data of the previous labs. But you will not use the full table, just the passenger column. The table is around **42M rows**, you can download the full column but it will take time, or only a few (10k for example) to write your script. Once your script ready, you can test it on the full column.

▼Here is the new code to connect to the database

```
pw <- "YojCqLr3Cnlw6onuzHU3" # Do not change the password !

drv <- dbDriver("PostgreSQL") # loads the PostgreSQL driver.
# (Needed to query the data)

# This is the proper connection.
# The same object will be used each time we want to connect to the
database.
con <- dbConnect(drv, dbname = "postgres",
                 host = "tp-avion.c5bqsgv9tnea.eu-west-
3.rds.amazonaws.com", port = 5432,
                 user = "postgres", password = pw)

rm(pw) # We don't need the password anymore.

dbExistsTable(con, "flight") # Check whether the "flight" table exist.
```

1. Benchmarking

In this part you will benchmark some median computation of the passenger column :

- The R base median function
- The R base quantile function
- The Spark quantile function

```
sdf_quantile([a spark object], [your data], probabilities = c(0.5),
relative.error = 1e-05)
```

- An SQL query

```
DBI::dbGetQuery(sc, "SELECT PERCENTILE(data,0.5)  AS MEDIAN from
[your data]")
```

- A naïve algorithm. The most naïve median computation is in 2 steps
    - Sort the data
    - Get the "middle" element

Q1.1. Which one is the faster with small data ? Big data ? Explain quickly why?

Q1.2. Do all this methods compute the same median ?

2. Median computation with a data stream

Q2.1 Can your median computation be used in a real time application with continuous stream of new data ??

To compute the *exact median* you need all data, and can't only update the previous median based on the new data. But you can structure data to make future computations faster.

The structure you will use is a "heap" (wikipedia page). This structure is a key/values structure, which orders data in a tree, with the min-heap property. For a given node C, if P is the parent node of C, the key of P is smaller then the key of C. There are multiple implementation of this structure, you will use the *Fibonnacci Heap*.

The algorithm is the follow :

- Initialize two heaps :
    - *smaller_heap* which store all the data smaller than the median after n insertions. Because the min-heap properties, this heap have to be "reverse". All inserted data will have its key multiply by -1.
    - *greater_heap* which store all the data greater than the *effective median* after n insertions.
- When a new piece of data arrives, check in which heap it should be inserted :
    - if data > *greater_heap*'s root : data go to *greater_heap*
    - else data go to *smaller_heap*
- If heaps have a size difference greater than 2, balance them
- Compute the median
    - If the size of the heap are not the same take the root element of the bigger one
    - Else compute the mean of the roots

▼To achieve this, you will use the datastructure package (documentation). Here is some code to help you

```
install.package("datasctrutures")
library(datastructures)
# Create a new fibonacci heap with numeric key, you can use character or interger if your keys are characters or intergers
heap <- fibonacci_heap("numeric")

# Insert a new node in the heap
heap <- insert(heap, -1,0, -1,0)
# -> Advice <-
# The node are orderded by their key, and not their value. To simplify the code, you can have the same key value for your node.

# Get the root node of a heap without delete it
root_node <- peek(heap)

# Get the VALUE of the root node
root_value <- root_node[[1]]

# Get the root node of a heap and delete it
root_node <- pop(heap)

# Get the size of a heap
size <- size(heap)
```

Q2.2. Is the Fibonacci heap better than the binomial heap for this algorithm ?

Q2.3. Implement the previous algorithm in a function which take as input a new element and the two heaps, and return the median and the two updated heaps

Q2.4. Benchmark you algorithm and the R base median. Which is the faster ? Why

Q2.5. What the theoretical time complexity of you algorithm to compute the new median when you add one element ?

Q2.6. Same question with your algorithm define in part 1.

3. Distributed median

Q3.1. Explain why the median is not as easy to distribute as the mean

Q3.2. Implement in spark your basic algorithm to compute the media

Q3.3. Is this implementation faster than the other functions on small data? Big data ? (you can try on more data than the passenger column, either by duplicating data or generate brand new data