A dark blue vertical bar on the left side of the page, with a blue arrow pointing right from its center.

GHCN Data Analysis using Spark

DATA420Assignment 1

Contents

1 About the data	1
2 Data processing	1
2.1 Explore the raw data in HDFS	1
2.1.1 Date structure	1
2.1.2 The daily climate summary	1
2.1.3 Data size	2
2.2 Load data into Spark	2
2.2.1 Define schemas and load data	3
2.2.2 Metadata statistics	3
2.2.3 Merge metadata at a station level	4
2.3.4 Save the enriched stations data in HDFS	7
2.3.5 Try to join the subset of daily and the enriched stations	8
3 Data analysis	8
3.1 Know more about the stations	8
3.1.1 Number of stations in total	8
3.1.2 Stations that have been active in 2020	8
3.1.3 Stations in each of the GSN, HCN, and CRN	8
3.1.4 Enrich countries and states	10
3.1.5 Stations are in the Northern Hemisphere only	10
3.1.6 Stations in the territories of the U.S	10
3.1.7 The pairwise distances between all stations in New Zealand	11
3.2 Explore all the daily climate summaries	11
3.2.1 Data size and block	11
3.2.2 Explore the number of tasks, partitions, blocks and file format	12
3.2.3 Discuss the level of parallelism	13

3.2.4 Count the number of rows in daily	15
3.2.5 Analyze daily focused on all core elements.....	17
3.2.6 Analyze the temperature in New Zealand.....	20
3.2.7 Analyze precipitation all over the world.....	22
3.2.8 Brief summary	25
4 Challenges.....	26
Works Cited	29

1 About the data

Here are some of the weather data contained in the Global Historical Climate Network (GHCN), which is an integrated database of climate summaries from land surface stations around the world. The data covers the last 258 years and is collected from more than 20 independent sources, each of which has been subjected to quality assurance reviews.

2 Data processing

2.1 Explore the raw data in HDFS

2.1.1 Date structure

All the data consists of two parts: the daily climate records and the metadata. The daily climate records from 1763 to 2020 are stored in compressed .csv files in the daily directory. The metadata (station, country, state, and variable inventory) are saved in 4 files respectively.

The data is structured according to the directory tree below:

```
Root(/data/ghcnd/)
|---countries
|
|---daily
|----1763.csv.gz
|----1764.csv.gz
|----1765.csv.gz
|...
|----2019.csv.gz
|----2020.csv.gz
|---inventory
|
|---states
|
|---stations
```

2.1.2 The daily climate summary

In the daily directory, there are 258 files that store the daily climate records from 1763 and 2020 separately. According to Figure 1, the size of the daily data shows an upward trend during the past 258 years. There is no huge difference over the first 100 years. However, the data size increased considerably from 1870 and reached a peak of 226,641KB in 2010. The sharp decrease occurred in 2020 because data is only collected in the first few months in 2020.

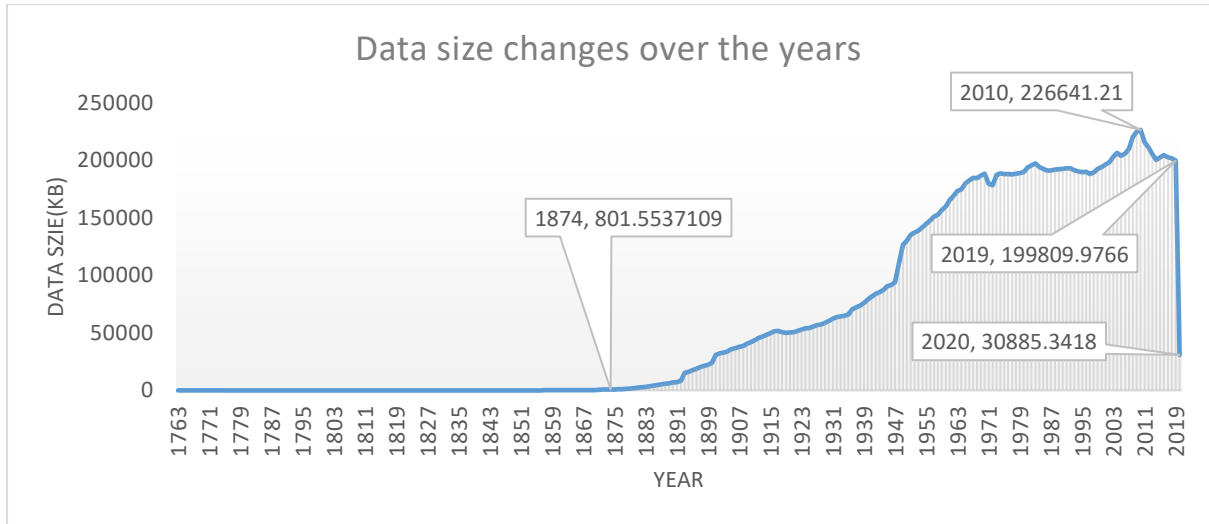


Figure 1 data size changing

2.1.3 Data size

The real size of the data is **15.5 G** in total, but it takes up 124.3 G disk space in the cluster because of the 8 replications.

Table 1 Data size

Data set	File(s) size	Disk space in the cluster
countries	3.6k	28.6k
daily	15.5G	124.0 G
inventory	30.1M	241.2M
states	1.1k	8.5k
stations	9.4M	75.5M

The metadata only contributes less than 40 MB to the total size, which is so tinny when compared with the daily records. The daily records account for almost all the data size. Meanwhile, the daily data is compressed, so the actual size of the uncompressed data will be significantly larger.

2.2 Load data into Spark

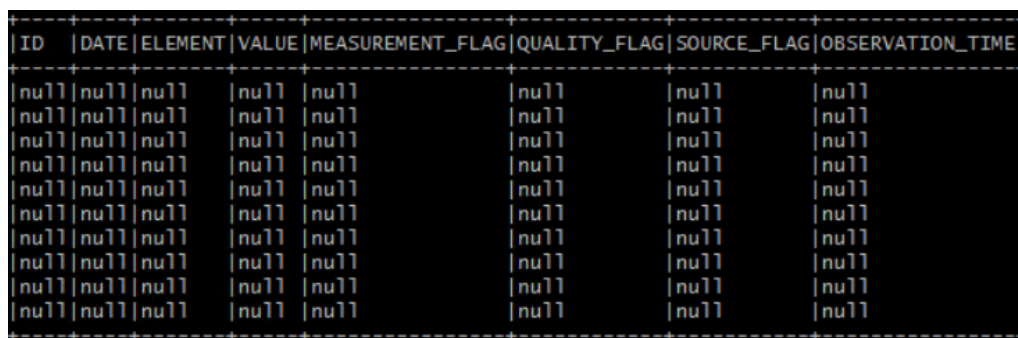
The process in this part shows below:

1. Define schemas
2. Load metadata and analyze statistically
3. Merge metadata at a station level
4. Save the enriched stations data in HDFS
5. Try to join the subset of daily and the enriched stations

2.2.1 Define schemas and load data

Before the schema definition, we check the detailed records for each data frame to understand the data further. As the daily data is saved in CSV files, we can load it into Data Frame directly. While stations, states, countries, and inventory are stored in the fixed-width text formatting, we need to parse the text line and get the attribute values by using *substring()* and *trim()*.

In addition, when I defined the schema for Daily based on the descriptions in this assignment and loaded data into Spark from 2020.csv.gz, I could not get the accurate records, all the cells are null(Figure 2) if I did not specify the date and time format.



ID	DATE	ELEMENT	VALUE	MEASUREMENT_FLAG	QUALITY_FLAG	SOURCE_FLAG	OBSERVATION_TIME
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null

Figure 2 wrongly loaded daily records

The columns of DATE and OBSERVATION_TIME are not formatted in a way that *DateType* and *TimestampType* can parse automatically. Hence, **I decided to read these two fields as strings and may convert them to the corresponding data types later if it's necessary.**

The final schemas definition and loading processes are recorded in the File

Assignmnet1_Processing_Q2Q3.py

Tips: daily.unpersist()--drop spark data frame from cache(memory and disk.)

2.2.2 Metadata statistics

The counts of rows in each metadata table are shown in Table 2, and there are **no duplicate** records in each table as I compared the row counts of each table before and after distinct().

Table 2 Metadata statistics

Table	Count of rows
countries	219
inventory	687141
states	74
stations	115081

Besides, we filter the WMO_ID column in the station table and find out that there are **106993** stations without WMO ID.

2.2.3 Merge metadata at a station level

2.2.3.1 Left join stations with countries

In order to join stations and countries, we extracted the first two characters of *ID*(station code) to get the country code as a new column(*FIPS*). Then stations could left join the countries on *stations.FIPS == countries.CODE*.

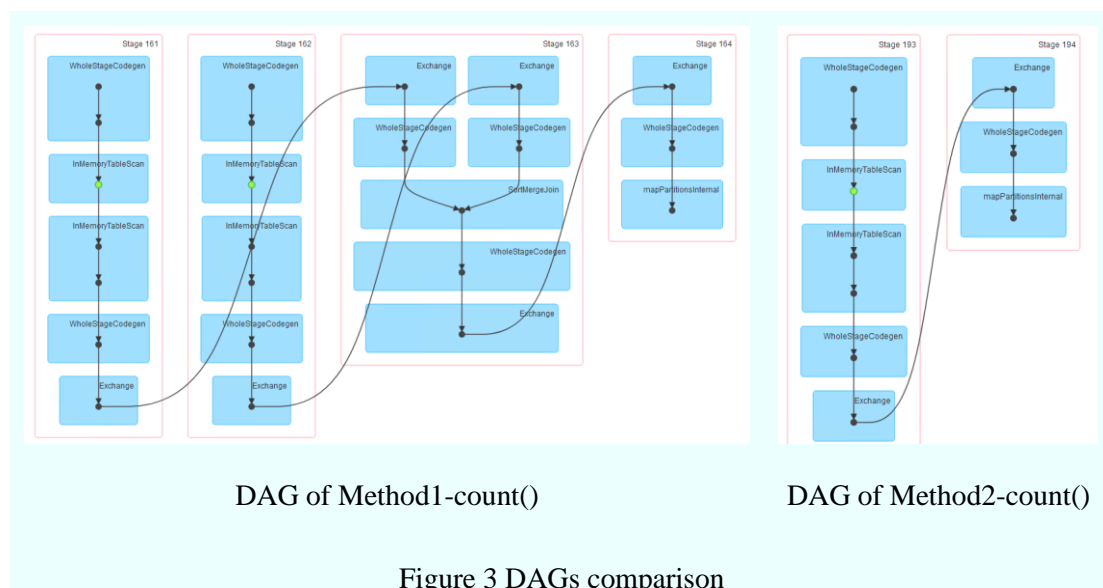
2.2.3.3 Left join stations and states

According to *readme.txt*, the code in states is the postal code of US state/territory or Canadian province where the station is located. As for joining stations with states allowing for the fact that state codes are only provided for stations in the US, there are two ways to achieve it.

Method1: Filter the US stations to get the subset stations and inner join the subset and the states. Then left join the stations with the output above and remove the extra attributes.

Method2: Left join station with states, and then update the STATE_CODE values to black when FIPS is not equal to “US”.

I chose Method2 as it is easy to understand. What's more, there is only one join operation and the state table is small. When we calculated the row count for stations, 3 shuffles occurred in Method1 and there was only 1 shuffle in Method2. We could see the details in Figure 3.



We could also determine that there are **61867** US stations.

2.2.3.3 Generate new attributes of stations based on inventory

Before analyzing the inventory, I checked the LAST YEAR, FIRSTYEAR, and ELEMENT columns to make sure there are no null values there.

3-1 The first and last year

By grouping ID, we can get the min FIRSTYEAR as OPEN(the first year that the station was active) and the max LAST YEAR as CLOSE(the last year that the station was active). The result is saved in `stations_active`.

3-2 The counts of totally different elements, core elements, and other elements

The five core elements are:

- PRCP = Precipitation (tenths of mm)
- SNOW = Snowfall (mm)
- SNWD = Snow depth (mm)
- TMAX = Maximum temperature (tenths of degrees C)
- TMIN = Minimum temperature (tenths of degrees C)

Here I tried two ways to get the counts of totally different elements, core elements, and other elements for each station based on inventory.

Method1: using `collect_set()` and UDF

- Defined a Python list `core_elements_list = ['PRCP', 'SNOW', 'SNWD', 'TMAX', 'TMIN']`
- Generated a new dataset called `stations_elementSet` by grouping ID and using `collect_set()` as this function returns a set of objects with duplicate elements eliminated. `stations_elementSet` has two columns: ID and `collect_set(ELEMENT)`
- Defined two functions to get the sizes of intersection and difference for two lists respectively and registered the user-defined Python functions as SQL functions.
- Created a new column called `core_elements_list` using the Python list.

Tips: `F.array([F.lit(x) for x in core_elements_list])` -- using `lit()` to change a python list to literal values for a column

- `F.size("collect_set(ELEMENT)")` is the count of different elements that each station has collected; the `core_elements_list` and `collect_set(ELEMENT)` are the input parameters for the UDFs to get the counts of core elements and other elements of each station collected.

Method2: using the `filter()` and `join()`

1. Selected distinctive ID and ELEMENT in inventory and got the count of ELEMENT

- for each ID using group by() and count(). The output is saved in *stations_elements*.
2. Filtered *ELEMENT* to get the inventory records with core elements named *inventory_with_coreelements*.
 3. Did the same operation on *inventory_with_coreelements* as step1 to get the number of core elements of each station. The result called *stations_core_elements*.
 4. Left join *stations_elements* and *stations_core_elements* and get the count of other elements using subtraction like $F.col('ELEMENTS') - F.col('CORE_ELEMENTS')$.

We showed the top five records of the outputs of the two methods. I guessed the second method should be more efficient as it firstly filtered inventory by element type. However, the consuming time is almost the same, but more shuffles occurred in Method2.

The reason is that there are two groupBy() and one join() operations which triggered shuffles in Method 2. Besides, filtering-first is more efficient in general, but it depends on the distribution of *ELEMENT*, which is associated with the file format.

The result of 3-2 is saved in *stations_elements*.

3-3 Left join stations with stations_elements and stations_active

We left joined stations with *stations_elements* and *stations_active*. So far, the attributes of stations are shown in Table 3:

Table 3 Attributes of enriched stations(new attributes in bold blue)

No.	Name	Annotation
1	STATION_ID	The station identification code
2	LATITUDE	The latitude of the station
3	LONGITUDE	The longitude of the station
4	ELEVATION	The elevation of the station
5	STATION_NAME	The name of the station
6	GSN_FLAG	whether the station is part of the GSN
7	HCN/CRN_FLAG	whether the station is part of the HCN/CRN
8	WMO_ID	The WMO number for the station
9	FIPS	The FIPS country code
10	COUNTRY_NAME	The country name where this station is located
11	STATE_NAME	The state name where this station is located
12	STATE_CODE	The postal code for the state (for U.S. stations only)
13	ELEMENTS	The number of different elements the station collected

14	CORE_ELEMENTS	The number of core elements
15	OTHER_ELEMENTS	The number of other elements
16	OPEN	The first year that this station was active
17	CLOSE	The last year that this station was active

4 Other analysis

There are **20266** stations collecting all five core elements and **314** different stations that only collected temperature.

In addition, there are 115081 stations but only 115072 of them occurred in inventory. The rest of the stations(9) are shown in Table 4.

Table 4 The stations without elements in *inventory*

No.	STATION_ID
1	GMM00010210
2	GME00124150
3	GME00122338
4	USC00144840
5	GME00122686
6	GME00123742
7	PKM00041529
8	GMM00010686
9	GME00125002

2.3.4 Save the enriched stations data in HDFS

The data can be formed in a human-readable format like JSON or CSV file, but that doesn't mean that's the best way to store the data(Hartono). As for the AVRO, ORC, and Parquet, the ORC file has the highest compression rate and its efficiency of reading is higher than parquet(Carter Shanklin) in general. ORC's column-oriented data stores are optimized for read-heavy analytical workloads. Therefore, we saved the enriched station data with the ORC format and we could compare the sizes of files for the same data with different formats in Figure 4.

```
[xzh216@canterbury.ac.nz~]$ hadoop fs -du -h /user/xzh216/Assign1/output
10.8 M  43.3 M  /user/xzh216/Assign1/output/stations_enriched.csv
3.7 M   14.9 M  /user/xzh216/Assign1/output/stations_enriched.orc
3.8 M   15.0 M  /user/xzh216/Assign1/output/stations_enriched.parquet
```

Figure 4 size of data files in different formats

All the outputs will be saved in the directory “/user/xzh216/Assign1/output”.I did not repartition the data before storage, but we will discuss it in Section 3.2.4 and 3.2.6.

2.3.5 Try to join the subset of daily and the enriched stations

We loaded the first 1000 rows of daily data in 2020 and left joined it with the enriched stations. There are 318 different stations in the subset of daily. By filtering the *STATION_ID*, we did not find any stations in the subset of daily that are not in stations.

Besides, I tried functions of `dropDuplicates()` and `exceptAll()` to find the unexpected stations. The function `subtract()` is also available. All three methods will reach the same conclusion.

There are 2,928,664,523 rows of all the daily data, and they will be stored in 108 partitions by default. The duration would be at least 2.9×10^6 times as long as that merely joining 1000 rows, and this cannot be tolerated, not to mention more shuffles and other resources.

However, in this case, the daily is huge and the station with 115018 rows is tiny (less than 2GB). We could broadcast the stations and join it with the daily data so that the small table will be distributed to each working node in the cluster. By doing this, the joining operations are done locally, the shuffles will be canceled basically, and we'll get huge performance gains.

3 Data analysis

In the beginning, we increased the resources up to 4 executors, 2 cores per executor, 4 GB of executor memory, and 4 GB of master memory.

3.1 Know more about the stations

3.1.1 Number of stations in total

We load the enriched station data from “/user/xzh216/Assign1/output/” and there are **115081** different stations in total.

3.1.2 Stations that have been active in 2020

Stations that have been active in 2020 mean the *CLOSE* is not less than 2020 and *OPEN* is not larger than 2020 (see the meaning of *CLOSE* and *OPEN* in Table 3 in Section 2.2.3). We could filter the stations based on the query condition. The result shows **32850** stations have been active in 2020.

3.1.3 Stations in each of the GSN, HCN, and CRN

How many stations are in each of the GSN, HCN, and CRN?

There are two possible values for GSN FLAG: Blank = not a GSN station or WMO Station number not available; GSN = GSN station. So, it is easy to filter rows using the given

condition (*F.col("GSN_FLAG")=="GSN"*) to get stations in the GCOS Surface Network (GSN) and there are **991** GSN stations totally.

HCN/CRN_FLAG has three possible values: Blank, HCN and CRN. We can filter stations using *F.col("HCN/CRN_FLAG")=="HCN"* and *F.col("HCN/CRN_FLAG")=="CRN"* to get the station number in HCN and in CRN respectively.

Alternatively, we can get the rows that the values of HCN/CRN_FLAG is not blank by using *where (F.col("HCN/CRN_FLAG").isNull())&(F.col("HCN/CRN_FLAG")!= "")*. Then we could group the count of *STATION_ID* by *HCN/CRN_FLAG*. By doing this, we could also get the number of stations in HCN and in CRN conveniently.

In short, there are **1218** stations that are the members of HCN, and **233** stations are in CRN. **113630** stations are neither in HCN nor in CRN.

Are there any stations that are in more than one of these networks?

Functions *when()* and *otherwise()* are the pair to set values for a newly generated column. I created three columns named *GSN*, *HCN*, and *CRN*. When the value of "*GSN_FLAG*" is equal to "*GSN*", the value of *GSN* is 1 otherwise it is 0. Similarly, we can set values for *HCN* and *CRN*. Then we could sum up the values of three new columns together to generate a new column called *SUM*. By filtering *SUM* given the condition *F.col("SUM")>1*, we could get the stations that are in more than one of these networks.

There are **14** stations in more than one of the networks and the station are shown in Figure 5.

According to the output, it is obvious that these stations are both in GSN and HCN. There are no stations in all the three networks

STATION_ID	STATION_NAME	GSN_FLAG	HCN/CRN_FLAG	SUM
USW00003870	GREER	GSN	HCN	2
USW00014771	SYRACUSE HANCOCK INTL AP	GSN	HCN	2
USW00023044	EL PASO INTL AP	GSN	HCN	2
USW00024128	WINNEMUCCA MUNI AP	GSN	HCN	2
USW00024144	HELENA RGNL AP	GSN	HCN	2
USW00024213	EUREKA WFO WOODLEY ISLAND	GSN	HCN	2
USW00014742	BURLINGTON INTL AP	GSN	HCN	2
USW00093729	CAPE HATTERAS AP	GSN	HCN	2
USW00094008	GLASGOW INTL AP	GSN	HCN	2
USW00012836	KEY WEST INTL AP	GSN	HCN	2
USW00012921	SAN ANTONIO INTL AP	GSN	HCN	2
USW00014922	MINNEAPOLIS/ST PAUL AP	GSN	HCN	2
USW00023051	CLAYTON MUNI AIR PK	GSN	HCN	2
USW00093193	FRESNO YOSEMITE INTL AP	GSN	HCN	2

Figure 5 The stations in more than one of the networks

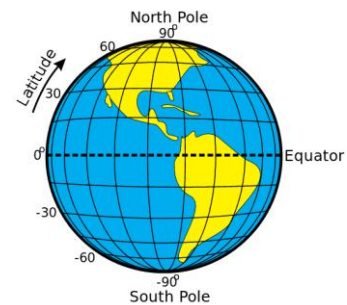
3.1.4 Enrich countries and states

In order to count the total number of stations in each country, we should group the count of *STATION_ID* by *FIPS* based on stations and get a new data frame called *countries_ex*. By left joining *countries* with *countries_ex*, we attach the column *count(STATION_ID)* to *countries* and rename the new one as *StationCount*. The enriched country data is saved in the output directory with the ORC format.

The enriched states could be got and stored in a similar way. However, I discovered that there are **21** states without stations as the value of *StationCount* is *null*.

3.1.5 Stations are in the Northern Hemisphere only

If the station is in the Northern Hemisphere, the value of its latitude would be positive. Hence, by filtering rows with the `F.col("LATITUDE")>0`, we knew there are **89745** stations in the Northern Hemisphere. By the way, there are **25336** stations in the Southern Hemisphere.



3.1.6 Stations in the territories of the U.S

Base on the enrich country data we have got in section 3.1.4, I used `countries.NAME.like('%United States%')` to find the countries whose name contains string “United States” and the result is shown in Figure 6. But the US is included in the list, we need to remove it to get the territories of the U.S and there are nine territories in this data set. The territories data frame is named *territories_us*. According to *territories_us*, we summed up the *SationsCount* and found that there are **316** stations in total in the territories of the U.S. Additionally, there are 61867 stations in ICUS.

CODE	NAME	StationCount
AQ	American Samoa [United States]	20
CQ	Northern Mariana Islands [United States]	11
GQ	Guam [United States]	21
JQ	Johnston Atoll [United States]	4
LQ	Palmyra Atoll [United States]	3
MQ	Midway Islands [United States]	2
RQ	Puerto Rico [United States]	211
US	United States	61867
VQ	Virgin Islands [United States]	43
WQ	Wake Island [United States]	1

Figure 6 Countries whose name contains “United States”

Besides, we could also inner join *stations* with *territories_us*, and count the rows to get the

count of stations in total in the territories of the U.S. But join will cause shuffle, and I will recommend the first way.

3.1.7 The pairwise distances between all stations in New Zealand

I defined a native python function to compute the geographical distance (Houston, Pete) and registered it as a Spark SQL function(Jessica) called *StationsDistance_1_udf*.

```
def StationsDistance_1(s_lon,s_lat,t_lon,t_lat)
```

s_lon,s_lat,t_lon,t_lat are the latitude and longitude of the two stations separately

When it comes to stations in New Zealand, we could filter the stations by ("*FIPS*")='NZ' and only select the three columns *STATION_ID*, *LATITUDE*, and *LONGITUDE* to get a new data frame named *stations_NZ*. There are 15 stations in New Zealand.

On the basis of *stations_NZ*, we got two copies and rename the columns because we need to cross join the two tables and treat *LATITUDE* and *LONGITUDE* as input parameters of the UDF. After cross join, it is necessary to remove the rows in which the two station IDs are the same, which is meaningless as the distance is always zero. So far, there would be 210(15*14) records in the data frame. Now we could call the UDF *StationsDistance_1_udf* to create a new column for the distances between two stations.

Additionally, as the distance from A to B and that from B to A are the same, 15 stations will generate 105 distances. Hence, we dropped the duplicated distances and the got the final 105 records. The result will be saved in the directory “/user/xzh216/Assign1/output/” with the name “*StationsDistance_NZ.orc*”

It is easy to find the two stations geographically furthest apart if we sort the specified column(*distance*) in descending order.

The two stations with ID 'NZ000093994' and 'NZ000939450' are geographically furthest apart with a distance of **2950.70**km.

Tips: dropDuplicates(["distance"]) [] is essential

3.2 Explore all the daily climate summaries

3.2.1 Data size and block

HDFS stores each file as blocks and distribute it across the Hadoop cluster. The default size of a block in HDFS is 128 MB(Hadoop 2.X+). The command *hdfs getconf -confKey "dfs.blocksize"* returns the same block size(134217728B=128M)

Table 5 Stored information of daily files

File	Size	# of blocks	Avg.block size
2020.csv.gz	30.2 M	1	30.2M
2015.csv.gz	198.0M	2	99.0M
2010.csv.gz	221.3 M	2	110.66M

In general, “when Spark reads a file from HDFS, it creates **a single partition for a single input split**. Input split is set by the Hadoop InputFormat used to read this file. If it is TextInputFormat in Hadoop, which would return us a single partition for a single block of HDFS while the split between partitions would be done on line split, not the exact block split”(Degget).

Based on these results in Table5, it is not possible for Spark to load and apply transformations in parallel for the year 2020. Supposing we treated the default block size as the split size, there is one partition only for data in 2020.

There are two blocks for data in 2010, it seems that the file could be split into 2 partitions so that each executor can operate on a single part, enabling parallelization. However, we will discuss the practicability later.

3.2.2 Explore the number of tasks, partitions, blocks and file format

Here we had two jobs:

Job1: load and count the number of observations in daily for the year 2015

Job2: load and count the number of observations in daily for the year 2020

There are **34,899,014** observations in daily for the year 2015 and **5,215,365** observations in 2020.

Through the web user interface, we could ascertain that there are two stages with two tasks in total in Job 1 and the number of tasks executed by each stage is one. For Job 2, we could reach the same results. According to Table 5, there are two blocks for data in 2015 while only one for that in 2020. Obviously, the number of tasks executed did not correspond to the number of blocks in each input. There is only one task in the first stage of Job1, so parallelization did not happen although there are two blocks for data in 2015.

By searching on the internet, I got the reason. “When it comes to the compressed file, we would get a single partition for a single file **as compressed text files are not splittable**” (Degget).

Now let us verify the viewpoint by Job3.

Job3: Load and count the number of observations in daily from 2015 to 2020.

The number of observations in daily from 2015 to 2020 is **178,918,901** and the number of partitions is **6**.

For Job3, there are 2 stages, with 6 tasks in Stage 1 and 1 task in Stage 2. We loaded six compressed text files and there are 6 tasks in the data loading stage.

As we applied for 4 executors and 2 cores per executor, the 6 tasks ran parallel.

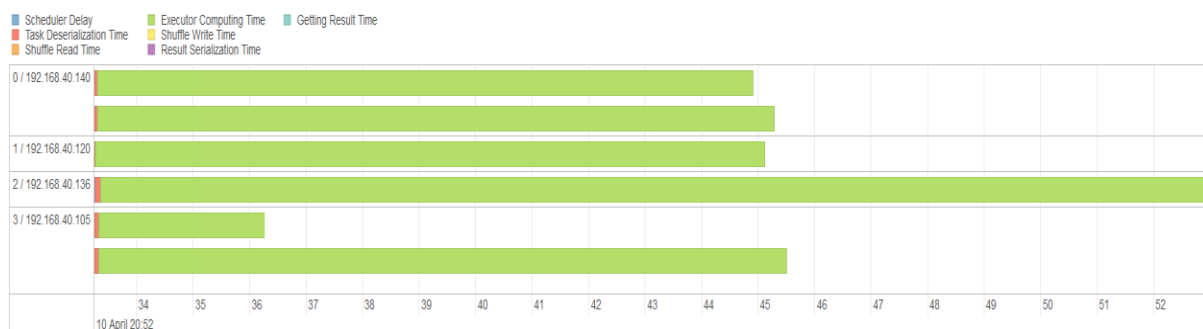


Figure 7 Event Timeline of Job3

Based on the findings so far, we could get the conclusion that Spark does not split a compressed CSV file(.gzip) even if the file size is larger than 128M, and we can get one partition for one compressed file.

Besides, I also discovered something interesting when I loaded all the data from a dictionary. We will discuss it in Section 3.2.4.

3.2.3 Discuss the level of parallelism

As we used 4 executors and 2 cores per executor, the max level of parallelism is 8 when loading and applying transformations to daily. But we had only 6 tasks in the first stage of Job3, so the level of parallelism is 6(Figure 7).

Data is split into partitions so that each executor can operate on a single part, enabling parallelization(Felipe). So, we can develop proper strategies to increase the level of parallelism in spark.

To sum up, there are four aspects we can pay attention to in my opinion:

1: Tuning development

- To avoid creating duplicates RDDs and reuse the same RDD as much as possible.

- If an RDD is used several times in the DAG, we should cache it or persist it in the disk, so further operations associated with this RDD could use it without recalculation.
- It is crucial to choose appropriate operations to reduce the times of shuffles(operations like groupBy, join reduceByKey、 join、 distinct、 repartition will cause shuffle) and the amount of data shuffled because shuffles are fairly expensive operations. But I have to say that “loading large unsplittable files and using the reduce or aggregate action to aggregate data into the driver are exceptions”(Sandy Ryza). So, when shuffle operations are unavoidable, try to use pre-aggregate action.

2: Tuning resource allocation

The two main resources that Spark (and YARN) think about are CPU and memory. In our cluster, there are 32 alive workers with 8 cores at each worker, so we have 256 CPU cores in total. The total memory in use is 883.2 GB.

In theory, the more resources are allocated, the faster the job will be executed. However, resources in the whole cluster are shared. It is unfair to allocate much more resources to a special application and too many resources will lead to waste. If we apply for resources when spark-submit, we should think about the total number of applications in the cluster and the priority of our application.

3: Tuning the number of tasks/partitions

Reasonable partitions enable us to utilize the cores available in the cluster and avoid excessive overhead in managing small tasks. There are two types of RDD:

a): An RDD with parent RDD:

The number of tasks in a stage is the same as the number of partitions in the first RDD in the stage. The number of partitions in an RDD is the same as the number of partitions in the RDD on which it depends(except coalesce, union, cartesian, etc)

“The most straightforward way to tune the number of partitions is experimentation: Look at the number of partitions in the parent RDD and then keep multiplying that by 1.5 until performance stops improving”(Sandy Ryza).

b): An RDD without parent RDD

RDDs produced by textFile or Hadoop files have their partitions determined by the underlying MapReduce InputFormat that’s used.

Based on this situation, here are some methods(Sandy Ryza).

- Use the repartition transformation, which will trigger a shuffle. Repartition works well especially when it comes to a data skew problem.
- Configure the InputFormat to create more splits.
- Write the input data to HDFS with smaller block size. But more blocks mean that more metadata are recorded on the NameNode in Hadoop.

In addition, the recommended number of tasks is 2 to 3 times the number of CPU cores ($\text{num_executors} * \text{num_cores_perexecutor}$).

4: Tuning data formats

We have discovered in Section 3.2.3 that the conclusion that Spark does not split a compressed CSV file. It is better to use an extensible binary format like Avro, Parquet, ORC, Thrift, or Protobuf to stored data on Hadoop and stick to the chosen format.

Back to our case, we allocated 8 CPU cores but there are only 6 tasks and **2 cores were waste**. we could extract the files and store them with Parquet or ORC format which is splittable and higher compression so that we can increase the level of parallelism without consuming more space in the disk and we could also repartition the data after loading.

3.2.4 Count the number of rows in daily

All the data stored in HDFS under `hdfs:///data/ghcnd` has a replication factor of 8 but there are 32 live nodes in the cluster. Hence, data is **not available locally on every node** in our cluster.

There are **256** CPU cores but around 80 users in the cluster. Therefore, I applied for 32 CPU cores (`start_pyspark_shell -e 8 -c 4 -w 1-m 1`) quickly to load and count the number of rows in daily.

Using the *command* `hdfs fsck /data/ghcnd/daily -blocks`, we could know that there are 258 compressed files in the daily directory with the size 16,639,100,391B(=15.5G) and the block number is **327**(Figure 8). After checking the size of each file in the directory, there are 69 files whose size is larger than 128M but less than 256M, and that is why the number of total blocks is 327.

```

Connecting to namenode via http://node0:9870/fsck?ugi=xzh216&blocks=1&path=%2Fdata%2Fghcnd%2Fdaily
FSCK started by xzh216 (auth:SIMPLE) from /192.168.40.10 for path /data/ghcnd/daily at Fri Apr 03 1
8:05 NZDT 2020

Status: HEALTHY
Number of data-nodes: 32
Number of racks: 1
Total dirs: 1
Total symlinks: 0

Replicated Blocks:
Total size: 16639100391 B
Total files: 258
Total blocks (validated): 327 (avg. block size 50884099 B)
Minimally replicated blocks: 327 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 4
Average block replication: 8.0
Missing blocks: 0
Corrupt blocks: 0

```

Figure 8 Block report of daily

There are **2,928,664,523** rows in daily. All the observations were saved in the data frame called *daily*. The number of partitions is **108**.

Here I found two interesting phenomena.

Firstly, load() triggers a job when there are many fragmented files in the dictionary.

In Section 3.2.2, loading and counting the observation number for one or six files was one job in Spark, but when I tried to load all files and count the number of rows in daily, there were two jobs. The codes are compared below:

```

path_2015to2020 = "hdfs:///data/ghcnd/daily/20{[1][5-9]},20}.csv.gz"
daily_2015to2020= (
  spark.read.format("com.databricks.spark.csv")
    .option("header", "false")
    .option("inferSchema", "false")
    .schema(schema_Daily)
    .load(path_2015to2020)
)
daily_2015to2020.count()

```

Code segment 1: load 6 files and count-1 job

```

path = "hdfs:///data/ghcnd/daily/*.csv.gz"
daily= (
  spark.read.format("com.databricks.spark.csv")
    .option("header", "false")
    .option("inferSchema", "false")
    .schema(schema_Daily)
    .load(path)
)
daily.count() #2928664523

```

Code segment 2: load all files and count-2 jobs

I checked the execution process by using my application console for segment2.

Spark Jobs (?)					
User: xzh216					
Total Uptime: 19 min					
Scheduling Mode: FIFO					
Completed Jobs: 2					
Event Timeline					
Completed Jobs (2)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2020/04/03 14:50:10	52 s	2/2	109/109
0	Listing leaf files and directories for 258 paths: hdfs://node0-9000/data/ghcnd/daily/1763.csv.gz, ... load at NativeMethodAccessorImpl.java:0	2020/04/03 14:49:55	3 s	1/1	258/258

Figure 9 Jobs' information

Consequently, we could confirm that `load()` triggers a job when there are many fragmented files in the dictionary.

Secondly, repartition occurs during the process of Spark Session `read()` and `load()`

there is only one stage in Job 0 with 258 tasks. As we have 258 compressed files in the directory, so we could understand why there are 258 tasks. However, the number of partitions is 108 rather than 258 and the tasks in the first stage of Job1 is also 108.

```
path = "hdfs:///data/ghcnd/daily/*.csv.gz"
daily= (
    spark.read.format("com.databricks.spark.csv")
    .option("header", "false")
    .option("inferSchema", "false")
    .schema(schema_Daily)
    .load(path)
)
daily.rdd.getNumPartitions() #108    load is a job
```

Code segment 2: spark read- DF

```
path = "hdfs:///data/ghcnd/daily/*.csv.gz"
rdd_daily = sc.textFile(path)
rdd_daily.getNumPartitions() #258
daily_4 = spark.read.csv(rdd_daily)
daily_4.rdd.getNumPartitions() #258
```

Code segment 3: sc read -rdd

According to Segment 3, `SparkContext` reads files [`sc.textFile(path)`] to get a signal RDD and it creates a single partition for a single compressed file. By contrast, `SparkSession read()` and `load()` return a data frame directly. Repartition occurs during this process.

Additionally, I also tried to load each file one by one using `spark.read()` from HDFS and *union* them together. Undoubtedly, the number of partitions for the union result is 258. I learned a lot from the attempt. Tips below are the gains.

Tips:

1: use pip to install the *hdfs* package in the user's directory

We have no permission to install packages in “`/usr/local/anaconda3/lib/python3.7/site-packages/`”, but we could install the package in our own directory and append this path to the system.

2: connect to a Hadoop Filesystem

The command “`echo $HADOOP_HOME`” and “`cat /opt/hadoop/hadoop/etc/hadoop/core-site.xml`” enable us to get the hostname: *node0*. The port for Hadoop 2.X is 50070 and for 3.X is **9870**. The source code is in *Assignmnet1_Analysis_Q4.py*(line 69-94).

3.2.5 Analyze daily focused on all core elements

The number of partitions of *daily.rdd* is 108, and we allocated 32 CPU cores for the application. The reasonable task number would be 64 to 96.

The first option is to allocate more resources while the 256 CPU cores in our cluster are for all the 80 users. It is embarrassed to occupy more resources. Consequently, I tried to repartition *daily* (More specifically, it is better to repartition small files).

How many observations are there for each of the five core elements?

--Is repartition necessary?

According to the question above, we could filter *daily* to obtain the subset of observations containing the five core elements. But in order to find a proper partition number, I decided to count the subset as an example. There will three stages of the job and extra shuffle is inevitable.

- The first stage is loading data and there are 108 tasks;
- The second stage is filtering core elements from *daily* then counting the observations of core elements in each partition. The number of tasks is the number of partitions.
- The final stage is summing the counts together.

One shuffle occurs between the first two stages because of **repartition** and the other shuffle is because **DataFrame's count operation uses groupBy**(read the source code).

In fact, the duration is around 3 mins for the first stage and 8 or 9 seconds for the second stage no matter how many partitions of *daily.rdd*. we need to compare the durations for the last stage as it concerns partitions number.

Table 6 Duration for count()-Stage3

# of partitions	Duration
96	47m
108	22m
128	38m
256	66m

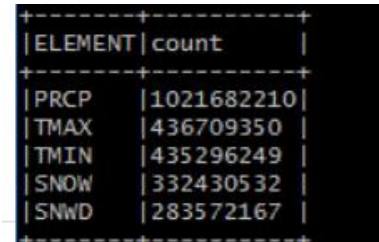
The four attempts are executed sequentially. Ignoring the changes in the Internet speed during the 20 mins, **it is not necessary to repartition *daily*** according to Table 6.

--Is data persistence necessary?

Besides, the subset(*daily_with_coreelements*) is the intermediate result for further analysis. There are **2,509,690,508** observations containing core elements, which accounts for **85.69%** of the daily data. It does not fit in memory, but we can persist it at the *MEMORY_AND_DISK* level. "In Python, stored objects will always be serialized with the Pickle library, so it does not

matter whether you choose a serialized level(Spark 2.4.5 Documentation).” It is better to allocate more memory for better performance. The important thing is that we must **trigger** the caching as it is a lazy evaluation.

I grouped the count of *ID* by *ELEMENT* based on *daily_with_coreelements*, then sorted the counts in descending order. The counts of observations for each of the five core elements are shown in Figure 10. The **PRCP** owns the most observations in *daily*.

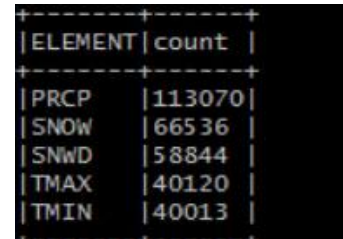


ELEMENT	count
PRCP	1021682210
TMAX	436709350
TMIN	435296249
SNOW	332430532
SNWD	283572167

Figure 10. # of Obs. for core elements

The duration of the job before the persistence is 2.2mins, but it is 22 seconds only after that. **So, it is better to persist *daily_with_coreelements*.**

There are 318583 observations in *inventory* with core elements and the counts of observations for each of the five core elements are shown in Figure 11. Element **PRCP** still has the **most** observations, which is consistent with the result we got above. But the orders are different.



ELEMENT	count
PRCP	113070
SNOW	66536
SNWD	58844
TMAX	40120
TMIN	40013

Figure 11. # of Obs. for core elements

In Processing Q3, we determined that there are 20266 stations that collect all five core elements. But how many different stations are there for each of the five core elements? The results are shown in Table 7. The results focusing on orders are consistent with each other.

Table 7 numbers of different stations for each of the five core elements

In <i>daily</i>		In <i>inventory</i>	
Element	Counts of different stations	Element	Counts of different stations
PRCP	113074	PRCP	113070
SNOW	66562	SNOW	66536
SNWD	59050	SNWD	58844
TMAX	40145	TMAX	40120
TMIN	40042	TMIN	40013
Total:318873		Total:318583	

How many observations of TMIN do not have a corresponding observation of TMAX?

Firstly, we should make sure all values of *VALUES* in *daily* is not null, so we could focus on '*ID*', '*DATE*' and '*ELEMENT*' to **shrink the data**.

As '*TMAX*' and '*TMIN*' are core elements, I extracted observations(*daily_with_tempelements*) of '*TMAX*' and '*TMIN*' from *daily_with_coreelements* which had been persisted. We also grouped the new created column *Temp* (*F.collect_list('ELEMENT').alias('Temp')*) by *ID* and *DATE*. There is no duplicate *<ID,DATE>*, so *collect_set* is better. Now, we got a data frame called *daily_temp*.

Based on *daily_temp*, we generated two columns using *F.array_contains()* to represent if the observation is *TMAX* or *TMAX*. The snapshot of intermediate result(*daily_temp_elements*) we got so far is shown in Figure 12.

ID	DATE	Temp	Has_TMAX	Has_TMIN
ACW00011604	19490316	[TMAX, TMIN]	true	true
ACW00011604	19490401	[TMAX, TMIN]	true	true
ACW00011604	19490409	[TMAX, TMIN]	true	true
AE000041196	19440621	[TMAX, TMIN]	true	true
AE000041196	19440622	[TMAX, TMIN]	true	true
AE000041196	19440717	[TMAX, TMIN]	true	true
AE000041196	19440722	[TMAX, TMIN]	true	true
AE000041196	19450107	[TMAX, TMIN]	true	true
AE000041196	19450207	[TMAX, TMIN]	true	true
AE000041196	19450411	[TMAX, TMIN]	true	true
AE000041196	19450611	[TMAX, TMIN]	true	true
AE000041196	19450814	[TMAX, TMIN]	true	true
AE000041196	19550426	[TMAX, TMIN]	true	true
AE000041196	19550908	[TMAX, TMIN]	true	true
AE000041196	19551112	[TMAX, TMIN]	true	true
AE000041196	19560119	[TMAX, TMIN]	true	true
AE000041196	19560128	[TMAX, TMIN]	true	true
AE000041196	19560309	[TMAX, TMIN]	true	true
AE000041196	19560423	[TMAX, TMIN]	true	true
AE000041196	19560704	[TMAX]	true	false
AE000041196	19560727	[TMAX, TMIN]	true	true

Figure 12 The snapshot of *daily_temp_elements*

By filtering *daily_temp_elements* using (*F.col("Has_TMIN") & (F.col("Has_TMAX") == False)*) we could determine that there are **8428801** observations of TMIN do not have corresponding observation of TMAX and **27526 different** stations contributed to these observations.

There are **18270703** observations of TMIN only or TMAX only. We could also filter the size of *Temp* to reach the same result. Besides, **30177different** stations contributed to these observations.

3.2.6 Analyze the temperature in New Zealand

In order to get all observations of TMIN and TMAX for all stations in New Zealand, we should know the stations in New Zealand and the observations of TMIN and TMAX in *daily*.

In Section 3.1.8, we determined that there are 15 stations in New Zealand. Now we could read the ORC file *stations_enriched.orc* from the output directory and get the IDs of stations in New Zealand only (*stations_NZ_ID*).

Besides, the *daily_with_tempelements* we got in Section 3.2.5 are the 872,005,599 observations of TMIN and TMAX.

How many observations of TMIN and TMAX are there for all stations in New Zealand?

I tried two ways to get observations of TMIN and TMAX for all stations in New Zealand.

Method1: filter *daily_with_tempelements* when *ID* in *stations_NZ_ID* and then count it.

Method2: broadcast *stations_NZ_ID* and join it with *daily_with_tempelements* then count it.

Tips:

Convert data frame in PySpark to Python list: *stations_NZ_list=[row.STATION_ID for row in stations_NZ_ID.collect()]*

Method1 cost 36 seconds while the duration of Method 2 was only 22 seconds, which means broadcasting is more efficient in this case. There are **458892** observations of TMIN and TMAX for all stations in New Zealand.

In addition, we added 2 new columns *YEAR* and *MONTH* by substring *DATE*, and the result was *daily_with_tempelements_NZ*.

How many years are covered by the observations?

It is easy to find that the max(*YEAR*) and min(*YEAR*) are 2020 and 1940 respectively. Furthermore, the min(*MONTH*) in 1940 is March and the max(*MONTH*) in 2020 is March. So, **80** years are covered by the observations.

Repartition the result and then save it to my output directory

The file "*daily_with_tempelements_NZ.orc*" is only 1.305M but there are 81 blocks. This means that plenty of space was waste and more memory was occupied in the name node. The best way is to repartition the data frame *daily_with_tempelements_NZ* so that only one block is required in HDFS and the file size is smaller(1.156M). I also saved a CSV file to plot time series in R.

Copy the output from HDFS to local and count the number of rows in the part files

As I repartitioned *daily_with_tempelements_NZ* to 1 before we saved the data frame, there is

only one file. The command below returns us **458892**, which is consistent with what we have got before.

command: `wc -l /users/home/xzh216/Assign1/Result/daily_with_tempelements_NZ.csv/part-00000-00127051-eee8-4454-a64c-507e7478a397-c000.csv`

Plot the time series of TMIN and TMAX on the same axis for each station in NZ

The period lasted for 80 years so we could plot the time series of TMIN and TMAX for the 15 stations with a frequency of one value every year. The time series for each station could be seen in R shiny.

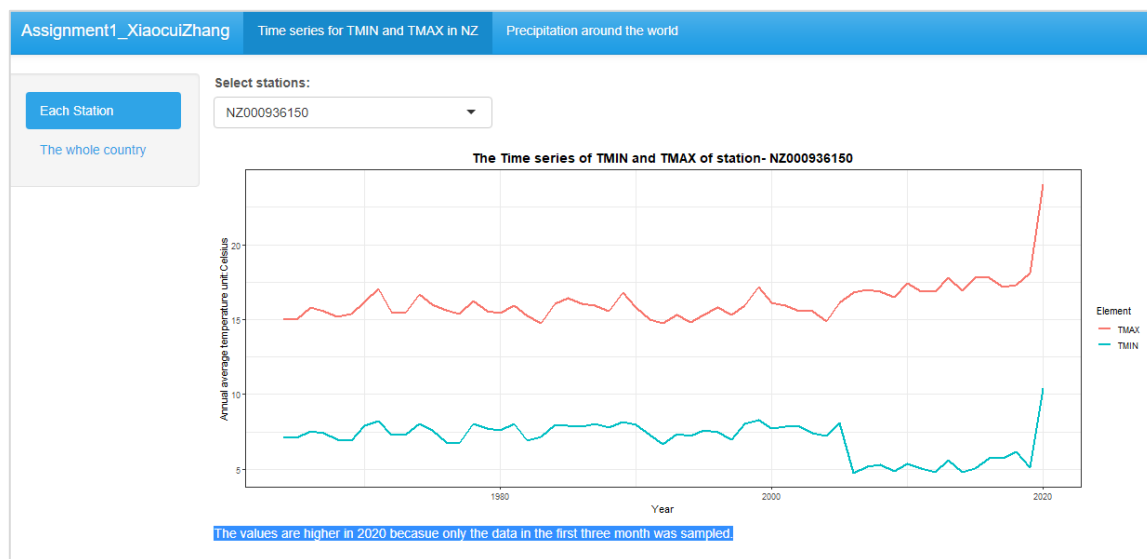


Figure 13 The snapshot of time series

3.2.7 Analyze precipitation all over the world

Before we computed the average rainfall in each year for each country, we can filter daily to get the observations of PRCP. Besides, we could also extract the first four characters of *DATE* to generate a new column *YEAR* for the following analysis. The result is named *daily_PRCs*. There are **1,021,682,210** observations of PRCP.

Country name and code are attributes of *Stations* data, which is smaller, so it is reasonable to broadcast *stations* and join it with *daily_PRCs* before grouping `mean("VALUE")` by '*FIPS*', '*COUNTRY_NAME*' and '*YEAR*'. I also counted the observations for each country of each year (*Counts_observations*), added all values (*Sum_Rainfall*), and took the value of *Avg_Rainfall_daily* to multiply by 365 to estimate the annual precipitation (*Avg_Rainfall_yearly*).

After sorting the daily average rainfall, we could determine that **Equatorial Guinea** had the

highest daily average precipitation (436.1mm) in 2000.(Figure 14).

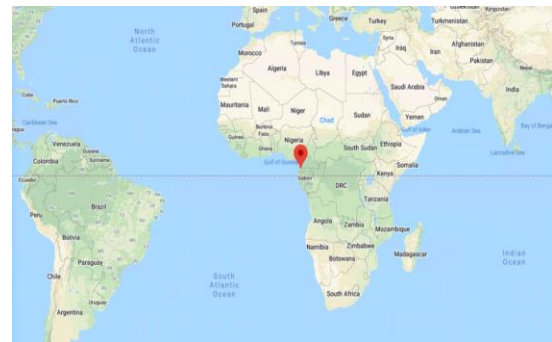
FIPS	COUNTRY_NAME	YEAR	Avg_Rainfall_daily	Counts_observations	Sum_Rainfall	Avg_Rainfall_yearly
EK	Equatorial Guinea	2000	4361.0	1	4361	1591765.0
DR	Dominican Republic	1975	3414.0	1	3414	1246110.0
LA	Laos	1974	2480.5	2	4961	905382.5
BH	Belize	1978	2244.714285714286	7	15713	819320.7142857143
NN	Sint Maarten	1979	1967.0	10	19670	717955.0
CS	Costa Rica	1974	1820.0	2	3640	664300.0
BH	Belize	1979	1755.5454545454545	11	19311	640774.0909090909
NS	Suriname	1973	1710.0	3	5130	624150.0
UC	Curacao	1978	1675.0384615384614	26	43551	611389.0384615384
BH	Belize	1977	1541.7142857142858	7	10792	562725.7142857143

only showing top 10 rows

Figure 14 The top ten average rainfall of daily around a year (unit: tenths of mm)

“Equatorial Guinea is a small African country located just north of the Equator. The climate is tropical, hot and humid all year round, with a slightly cooler period from June to September when the south-west currents prevail(Pélissier, René)”. It sounds reasonable that this country had the highest daily average precipitation. But the annual rainfall varied from 1,930 mm at Malabo to 10,920 mm at Ureka, Bioko(Wikimedia). If the daily average rainfall is 436.10mm, the annual

rainfall would be 159176.5mm, which is much larger than 10920 mm. Hence, our result is not sensible.



I checked the PRCP observations of Equatorial Guinea in 2020 and discovered that there was only one row on June,22 and this observation failed gap check (the value of *QFLAG* is G). Perhaps it is better to remove the observation with a quality problem. If I did that, the result would be **Belize in 1978** with daily average precipitation of 195.88 mm. I do not think this result is reliable either, or maybe we need further advice from meteorologists.

FIPS	COUNTRY_NAME	YEAR	Avg_Rainfall_daily	Counts_observations	Sum_Rainfall	Avg_Rainfall_yearly
BH	Belize	1978	1958.8333333333333	6	11753	714974.1666666666
CS	Costa Rica	1974	1820.0	2	3640	664300.0
BH	Belize	1979	1755.5454545454545	11	19311	640774.0909090909
NS	Suriname	1973	1710.0	3	5130	624150.0
BH	Belize	1977	1541.7142857142858	7	10792	562725.7142857143
TS	Tunisia	1973	1162.0	4	4648	424130.0
BM	Burma	2006	1152.0	2	2304	420480.0
EK	Equatorial Guinea	2001	1100.0	1	1100	401500.0
AE	United Arab Emirates	1995	1018.0	4	4072	371570.0
HO	Honduras	1978	934.6585365853658	41	38321	341150.3658536585

Figure 15 The top ten daily average rainfall after the quality filter (unit: tenths of mm)

By the way, I also found that there are **218** countries in the result and **Pitcairn Islands** [United

Kingdom](PC) had no average rainfall because it had no observations of PRCP. It had 9968 TMAX observations and 9962 TMIN observations only.

The result *daily_Avg_Rainfall* is saved in "hdfs:///user/xzh216/Assign1/output/".

Based on *daily_Avg_Rainfall*, we could keep on calculating the cumulative rainfall for each country through functions in *pyspark.sql.Window*. The result *daily_Cum_Rainfall* is also sorted in the output directory. I also repartitioned the two small results before we save them to the output directory.

However, the counts of observations for each country of each year are varied; the numbers of stations in each country are different, and the timespans that countries collecting records are also different. In my opinion, **it is not reasonable** to sum the daily average precipitation together through the whole period then make a choropleth for comparison (Figure 16).

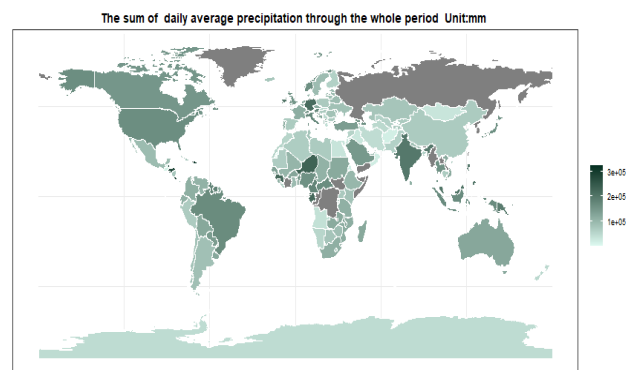


Figure 16 The sum of daily average precipitation

Instead, we recommend using the choropleth to show the annual average precipitation year by year and the results are more sensible.

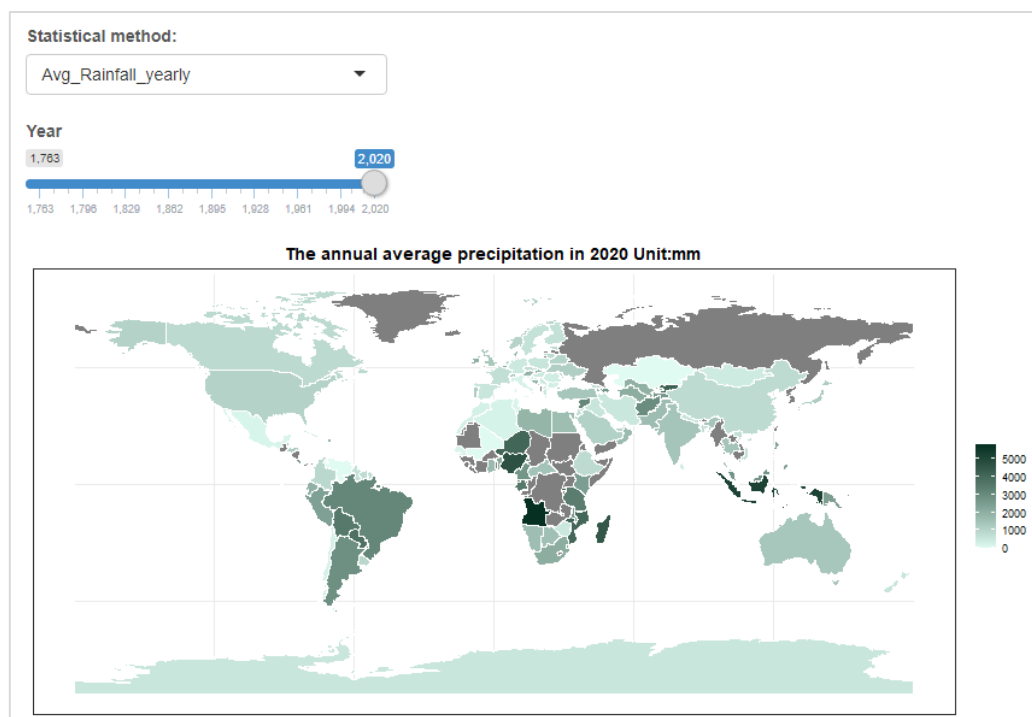


Figure 17 The annual average precipitation in 2020

3.2.8 Brief summary

Counts of observations

Table 8 observations counts summary

Count	Element(s) of observations
458892	TMIN or TMAX in NZ
8428801	TMIN without TMAX in all countries
18270703	TMIN only or TMAX only in all countries
872005599	TMIN or TMAX in all countries
1021682210	PRCP in all countries
2509690508	core in all countries
2928664523	any elements in all countries

Results in HDFS

We have saved seven results in the output directory so far, but some of the results are saved with different file formats for comparison or plotting.

```
[xzh216@canterbury.ac.nz~]$ hdfs dfs -ls /user/xzh216/Assign1/output
Found 12 items
drwxr-xr-x - xzh216 xzh216 0 2020-04-01 16:51 /user/xzh216/Assign1/output/StationsDistance_NZ.orc
drwxr-xr-x - xzh216 xzh216 0 2020-03-31 22:43 /user/xzh216/Assign1/output/countries.orc
drwxr-xr-x - xzh216 xzh216 0 2020-04-12 22:37 /user/xzh216/Assign1/output/daily_Avg_Rainfall.csv
drwxr-xr-x - xzh216 xzh216 0 2020-04-12 22:37 /user/xzh216/Assign1/output/daily_Avg_Rainfall.orc
drwxr-xr-x - xzh216 xzh216 0 2020-04-12 23:10 /user/xzh216/Assign1/output/daily_Cum_Rainfall.csv
drwxr-xr-x - xzh216 xzh216 0 2020-04-12 23:10 /user/xzh216/Assign1/output/daily_Cum_Rainfall.orc
drwxr-xr-x - xzh216 xzh216 0 2020-04-12 21:07 /user/xzh216/Assign1/output/daily_with_tempelements_NZ.csv
drwxr-xr-x - xzh216 xzh216 0 2020-04-12 21:03 /user/xzh216/Assign1/output/daily_with_tempelements_NZ.orc
drwxr-xr-x - xzh216 xzh216 0 2020-03-31 22:55 /user/xzh216/Assign1/output/states.orc
drwxr-xr-x - xzh216 xzh216 0 2020-03-31 19:58 /user/xzh216/Assign1/output/stations_enriched.csv
drwxr-xr-x - xzh216 xzh216 0 2020-03-31 19:58 /user/xzh216/Assign1/output/stations_enriched.orc
drwxr-xr-x - xzh216 xzh216 0 2020-03-31 19:58 /user/xzh216/Assign1/output/stations_enriched.parquet
[xzh216@canterbury.ac.nz~]$
```

Figure 18 results in HDFS

RDD Lineage

The rectangular forms in light blue are the main data frames, I used the DF names to represent primary RDDs in this picture. That is why we persisted *daily_with_coreelements* and *daily_PRCPs*.

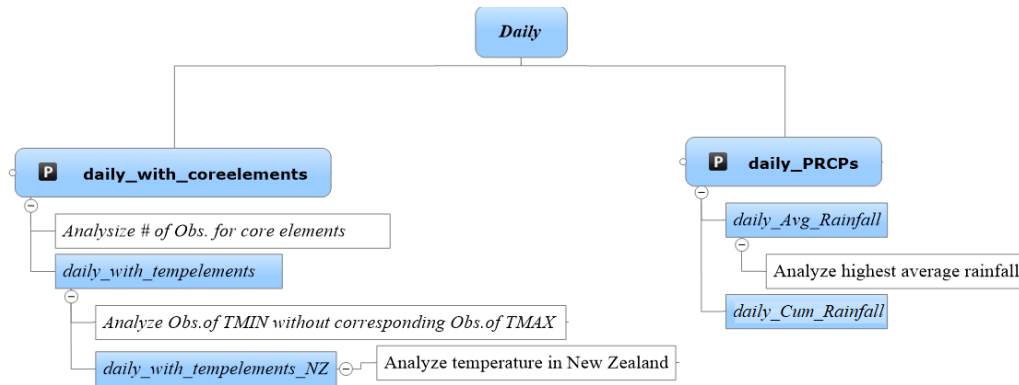


Figure 19 schematic RDDs Lineage

4 Challenges

There are 2,928,664,523 rows in daily, and 2,919,648,731 rows that did not fail any quality assurance check, which takes up **99.96%** collectively. This means **9,015,792** rows in daily failed the quality assurance checks.

The observations without good quality

We attempted to explore the distribution of the 9,015,792 rows and the result is illustrated in Figure 20.

QUALITY_FLAG	obs_bad_count
I	6328103
D	1226489
L	329631
S	300290
K	215752
O	174224
G	137229
X	135629
Z	96725
M	27496
N	17649
W	13308
R	11984
T	1283

Figure 20 counts of rows failed the quality assurance checks

The evaluation of the data quality focusing on stations

Firstly, we decided to find out the subset observations that did not fail any quality assurance check from daily. Then, we could analyses how many observations with good quality there are for each station and evaluated the proportions of observations without quality issues. The result is called *Evaluations_quality*.

ID	obs_all_count	obs_good_count	obs_bad_count	good_perc	bad_perc
ASN00008194	3496	3496	0	1.0	0.0
ASN00010501	17263	17263	0	1.0	0.0
ASN00009031	39814	39814	0	1.0	0.0
ALE00100939	36143	36143	0	1.0	0.0
ASN00009048	2923	2923	0	1.0	0.0
ASN00002033	14217	14217	0	1.0	0.0
ASN00009727	4383	4383	0	1.0	0.0
ASN00010276	1065	1065	0	1.0	0.0
ASN00009734	515	515	0	1.0	0.0
ASN00006077	3380	3380	0	1.0	0.0

Figure 21 snapshot of the result of the evaluation

The value of *good_perc* is the proportions of observations without quality issues at the corresponding station.

There are no stations appearing *daily* that are not in *stations*, but there are 8 stations that are not in *daily*. The 8 stations are listed in Table 9.

Table 9 The stations without elements in *daily*

No.	STATION ID
1	GME00122338
2	GME00122686
3	GME00123742
4	GMM00010686
5	GME00125002
6	GMM00010210
7	GME00124150
8	PKM00041529

Among the **115073** existing stations, only one station (**USC00144840**) failed all the quality assurance checks. By filtering *Evaluations_quality* using “*good_perc==1*”, we confirmed that only **49185** stations whose data quality is excellent, which consists of **42.74%** of the 115073 stations.

Furthermore, these 49185 stations covered **433,463,954** observations which make up **14.80%** of all observation in *daily*.

The evaluation of the data quality focusing on stations

By joining the *Evaluations_quality* with *stations*, we could group the data by *COUNTRY_NAME* and calculate the proportions of observations without quality issues for each country.

According to the diagram below, we could determine that **7** countries do not have any quality problems of the data.

COUNTRY_NAME	good_counts	all_counts	good_perc
Dominica	3655	3655	1.0
Burundi	27941	27941	1.0
Uganda	123329	123329	1.0
Niue [New Zealand]	7795	7795	1.0
Antigua and Barbuda	13936	13936	1.0
Tokelau [New Zealand]	7394	7394	1.0
Guatemala	5324	5324	1.0
United Arab Emirates	146004	146006	0.9999863019327973
Brunei	39283	39284	0.9999745443437532
Luxembourg	281617	281628	0.9999609413836692
Seychelles	63703	63706	0.999529086742223
New Zealand	753222	753259	0.999508801089665
Norfolk Island [Australia]	36357	36359	0.999449929866058
Liberia	18108	18109	0.999447788392511
Bahrain	33908	33910	0.999410203479799
Serbia	300291	300312	0.999300727243666
Equatorial Guinea	13212	13213	0.999243169605692
Iraq	25547	25549	0.999217190496693
Qatar	33956	33959	0.999116581760358
Netherlands	16707181	16708735	0.9990699475454
French Polynesia	385205	385242	0.999039564741123
El Salvador	89753	89762	0.998997348543928
Wallis and Futuna [France]	104950	104961	0.998951991692152

Figure 22 counts of observations belong to each country

The evaluation of the data quality focusing on years

Besides the analysis of data quality focusing on stations and countries, we could also investigate data quality temporally. The percentage of observations without quality issues fluctuated significantly at the beginning and tended to be steady gradually even though some valleys occurred.

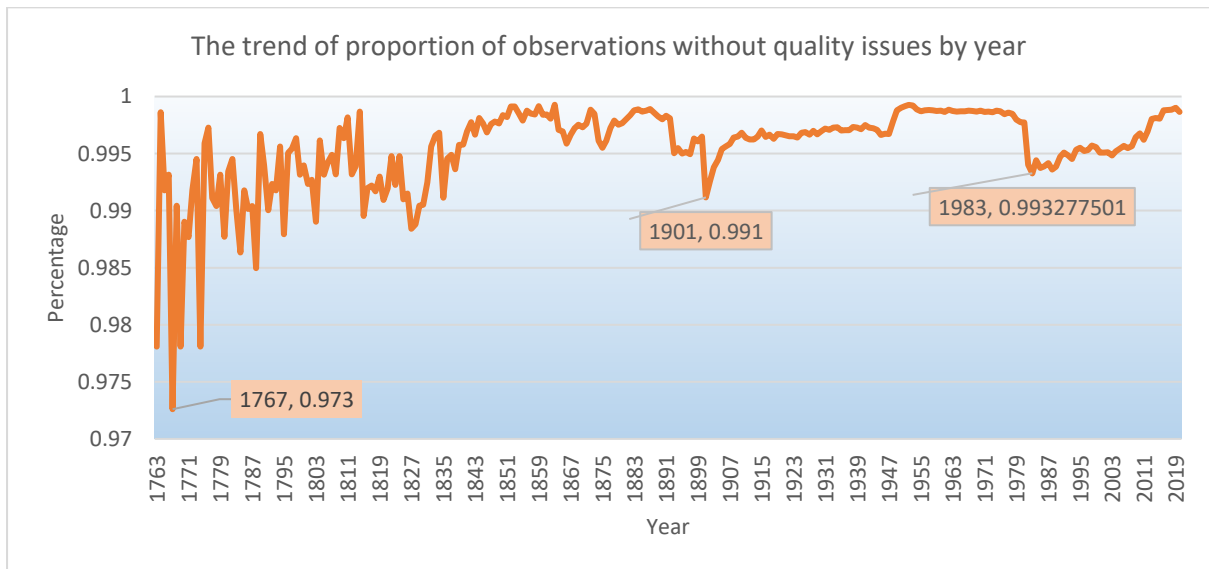


Figure 23 The trend of data quality by year

Works Cited

- Hartono, Oswin Rahadiyan. "Parquet, Avro or ORC?" *Medium*, Medium, 4 Nov. 2019, medium.com/@oswin.rh/parquet-avro-or-orc-47b4802b4bcb.
- Carter Shanklin. "ORCFile in HDP 2: Better Compression, Better Performance." *Cloudera Blog*, 11 Sept. 2019, blog.cloudera.com/orcfile-in-hdp-2-better-compression-better-performance/.
- Houston, Pete. "Calculate Distance of Two Locations on Earth Using Python." *Medium*, Medium, 31 Mar. 2019, medium.com/@petehouston/calculate-distance-of-two-locations-on-earth-using-python-1501b1944d97.
- Jessica Walkenhorst. "How to Convert Python Functions into PySpark UDFs." *Tales of One Thousand and One Data*, 7 Feb. 2019, walkenho.github.io/how-to-convert-python-functions-into-pyspark-UDFs/.
- Degget. "How Does Spark Partition(Ing) Work on Files in HDFS?" *Stack Overflow*, 1 Dec. 1964, stackoverflow.com/questions/29011574/how-does-spark-partitioning-work-on-files-in-hdfs.
- Felipe. "Spark Concepts Overview: Clusters, Jobs, Stages, Tasks, Etc." *Queirozf.com*, 23 Dec. 2019, queirozf.com/entries/spark-concepts-overview-clusters-jobs-stages-tasks-etc.
- Sandy Ryza. "How-to: Tune Your Apache Spark Jobs (Part 1)." *Cloudera Blog*, 9 Aug. 2019, blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-1/.
- Sandy Ryza. "How-to: Tune Your Apache Spark Jobs (Part 2)." *Cloudera Blog*, 9 Aug. 2019, blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-2/.
- Spark 2.4.5 Documentation "RDD Programming Guide." *RDD Programming Guide - Spark 2.4.5 Documentation*, spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence.
- Péllissier, René, and Ronald James Harrison-Church. "Climate." *Encyclopædia Britannica*, Encyclopædia Britannica, Inc., 6 Sept. 2019, www.britannica.com/place/Equatorial-Guinea/Climate.
- "Equatorial Guinea." *Wikipedia*, Wikimedia Foundation, 11 Apr. 2020, en.wikipedia.org/wiki/Equatorial_Guinea#Climate.