Big Data Management

Project Report : Formatted and Exploitation Zones - P2

Tejaswini Dhupad
Chun Han Li

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

June, 2022

TABLE OF CONTENTS

Please find here with the *link* to our Github repository.

1. Introduction

After the P1 implementation of the project, we pivoted the business idea. Wherein along with the business idea[1] the implementation structure of the project also changed. Please refer to the modified business idea diagram in the Appendix A. The implementation is divided into two parts -

1. Schedule generation[2]
2. Data Partitioning[3]

Now with our business idea, we are trying to utilize the idle time of computers(resources) to solve some complex computation problems. Here we have data from both provider(IdleHost[4]) and tenant(IdleTenant[5]). Therefore, scheduling is an important part of the implementation. To maximize the utilization of all resources[6] for solving the research problem , we design a scheduling algorithm with multiple constraints, which are partially implemented by Spark.

2. Dataset

We don't have any real dataset for the users, so we generated all the data by ourselves in Python.
We have mainly 2 types of data format:

1. Tenant data (i.e Task data)
2. Provider data (i.e. Resource data)

The Task data is as shown below :

	A	B	C	D	E	F	G	H
1		ID	name	deadline	type	data_size_GB	RAM_requirement_GB	working_units
2	1	T0	Violet	24-05-2022 02:00	CPU	70	16	410
3	2	T1	Sam	21-05-2022 10:00	GPU	80	32	280
4	3	T2	Alexia	21-05-2022 08:00	CPU	50	16	420

where,
type - defines the type of resources needed (CPU/GPU)
data_size_GB - defines the total data size of a task in unit GB
working_unit - defines a quantified workload unit wrt task.
For example : If we are using the same computer to run 100 working-units task, it would take twice as long as it would for running 50 working-units .

The Resource data is as shown below :

	A	B	C	D	E	F	G	H	I	J	K	L
1		ID	name	available_start_time	available_end_time	type	disk_size	CPU_name	CPU_RAM	GPU_name	GPU_RAM	working_units_per_hour
2	1	R0	Richardso	27-05-2022 18:00	28-05-2022 09:00	GPU	380	AMD_Naples_Epyc	64	Nvidia_TITAN_X	256	25
3	2	R1	Reed	17-05-2022 10:00	17-05-2022 21:00	GPU	440	AMD_Rome_Epyc	16	GeForce_GTX_1080	1024	15
4	3	R2	Brown	22-05-2022 12:00	23-05-2022 07:00	CPU	450	Intel_Itanium_2	128	GeForce_RTX_3080	64	10

where,
available_start_time - defines the time when the resource will be available
available_end_time- defines the end time of the resource availability
type - defines the kind of resource provided (CPU/GPU)
working_unit_per_hour - defines the computational power of the resource(i.e.machine)
For example : A working_unit_per_hour=10 computer can finish a working_units=80 in 8 hours.
More detail about how the data generated can be seen in ***data_gen.ipynb*.**

---

[1] The business idea shifted from the both IdleTenant and IdleHost to be paying parties and IdleCompute being the facilitator to-now only the IdleTenant as the fee paying party if they want to utilize the computation and IdleHost would be donating his/her resources(computation powers) at will without any monetary benefit, as a goodwill gesture.
[2] The schedule generation from the available data from the both IdleHost and IdleTenant is implemented by team **Team-BDMA11-C2**(Tejaswini Dhupad and Chun Han Li)
[3] The data partitioning i.e. chunking(dividing) the dataset and sending it to different machines(resources) for computation based on the schedule generated is implemented by the team **BDMA11-B1**(Adam Broniewski and Vlada Kylynnyk)
[4] IdleHost refers to the people who would be donating their computation powers to IdleCompute
[5] IdleTenant refers to any research organization that wants to perform computation
[6] Resources refer to the computation capabilities/powers provided by the IdleHost

3. Persistent landing to Data formatters

In the persistent landing we had our data in the HDFS in *parquet* [7]format. In the folder at location - "hdfs://10.4.41.85:27000/user/bdm/dataset/" we have stored our files for the data that we are collected from both the IdleHost and IdleTenant through the web form that they would fill during the registration process. The user data is stored in the monthly format.

☐	-rw-r--r--	bdm	supergroup	32 KB	Jun 09 11:32	3	128 MB	provider_2022-07.parquet
☐	-rw-r--r--	bdm	supergroup	102.92 KB	Jun 09 11:32	3	128 MB	tenant_2022-05.parquet

Fig.1. Data representation of the persistent landing(P1)

We read the data from P1 and extract the data we need for scheduling using PySpark, as shown below for both *Tenant* and *Provider* data.

```
read_pfile.createOrReplaceTempView("p_view") #Creating view for provider data
p_file = spark.sql("SELECT * FROM p_view ").rdd
data1 = p_file.map(lambda line:
(line['ID'],line['available_start_time'],line['available_end_time'],line['type'],line['d
isk
_size'],line['CPU_RAM_GB'],line['GPU_RAM_GB'],line['working_units_per_hour'] ))
```

4. Formatted Zone

After this, we save the dataset to Formatted Zone (HDFS) at location "hdfs://10.4.41.85:27000/user/bdm/formatted_data/" using the Pickle [8]module, as our data is in list format. The screenshot for saved data can be seen below:

☐	-rw-r--r--	bdm	supergroup	16.75 KB	Jun 13 18:41	3	128 MB	2022-05_provider
☐	-rw-r--r--	bdm	supergroup	268.44 KB	Jun 13 18:41	3	128 MB	2022-05_task

5. Exploitation Zone - Implementing Scheduling Algorithm

Now we read the data from the Formatted Zone for our Exploitation Zone, and then we implement the scheduling algorithm using PySpark(partially) and Python(for the last step).

A. Read Data and Convert to RDD : As our data is in a pickled state we will need to unpickle the data to perform the further operations. Hence, we transform the data to RDD in PySpark, as shown below:

```
data1 = sc.parallelize(data1_read) #from list to RDD, data1 is RDD
data2 = sc.parallelize(data2_read)
```

B. Combining Instances : We are combining the instances from different sources i.e. combining the Task data with Resource data. As shown below we are doing this process in PySpark with a Cartesian function.

```
cartesian_Rkey = data1.cartesian(data2)
```

In this way by using PySpark we are able to easily combine the required fields from two different sources(i.e. Data files), and now all the needed data are in one RDD.

---

[7] Apache Parquet is a free and open-source column-oriented data storage format in the Apache Hadoop ecosystem
[8] The pickle module implements binary protocols for serializing and de-serializing a Python object structure

Scheduling Algorithm :

Assumption: 1. We can't have a situation where an IdleHost who registers for donating his/her resources and for some reasons withdraws. Then this situation can't be tackled currently with our scheduling algorithm.

2. We can't have a situation where an IdleTenant who wants to utilize the computing powers/capabilities decides to withdraw from the contract. Then this situation can't be tackled currently with our scheduling algorithm.

3. We also can't have backup resources for a particular task, as we are only considering the one-to-one relationship between them i.e. every task will have only one resource allocation. So, in the case wherein if a particular IdleHost is withdrawn, we will need to rerun the algorithm again for schedule generation.

Scheduling problem is an optimization problem[9].

Maximize : Number of tasks finished on time.

Constraints:

1. Resources type match i.e. CPU==CPU and GPU==GPU
2. Resource RAM > Task RAM requirements
3. Resource Disk Size > Task File Size

Please refer to Appendix B for the actual(pictorial) representation of the algorithm.

Steps of Implementation :

I. Call the filter function to filter out all the pairs that do not meet the constraints.

```
Rkey_rdd1=cartesian_Rkey.filter(lambda
t:( (t[0][3]==t[1][2])&(t[0][4]>t[1][3])&(t[0][5]>t[1][4]) ) if t[1][2]=='CPU'\
                        else ( (t[0][3]==t[1][2])&(t[0][4]>t[1][3])&(t[0][6]>t[1][4]))  )
```

II. Pairs are sorted according to their task deadlines, and the sorted tasks represent our priorities for further assignment of tasks.

```
Rkey_rdd2 = Rkey_rdd1.sortBy(lambda x: x[1][1], ascending=True).sortBy(lambda x:
x[0][7], ascending=False)
```

III. Next, we want to know for every resource, and list all tasks that this resource can perform. We are using reduceByKey, since the tasks are already sorted, all the task lists in each resource should also be sorted after reduceByKey.

```
Rkey_rdd4 = Rkey_rdd3.reduceByKey(lambda a, b: a + '|' + b)
RT = Rkey_rdd4.collect()
```

For example, these 'T2059','T298','T366'…… tasks that resource 'R9' can perform.

```
[('R9',
  'T2059|T298|T366|T3128|T4108|T3154|T3259|
```

IV. Generate an available time table for each resource. The columns are all the resources, and every row is one hour, the value is the status for that resource in that hour.For example, as shown below, resource 'R8' is working on 01/05/2022 4h to 6h.

[9] In mathematics, computer science and economics, an **optimization problem** is the problem of finding the *best* solution from all feasible solutions.

	A	B	C	D
1		R8	R52	R65
2	2022/5/1 00:00	offline	offline	offline
3	2022/5/1 01:00	offline	offline	offline
4	2022/5/1 02:00	offline	offline	offline
5	2022/5/1 03:00	offline	offline	offline
6	2022/5/1 04:00	working	offline	offline
7	2022/5/1 05:00	working	offline	offline

V. Scheduling, this is the last step of our algorithm. Now we have the available time table, and also the task waiting list (generated by Spark, sorted by deadline). We do the scheduling as shown below,

```
#Allocate the works
for i in timetable:  #every resource
    for j in range(len(timetable[i])): #every one hour
        if timetable[i][j] == 'working':
            if len(RT_list[i]) == 0: #nothing in the task waiting list
                print(i+' have nothing to do')
                break
            else: #something in the waiting list
                run_task = RT_list[i][0] #choose the first one, the most urgent one (because it's sorted)
                if work_remain[run_task] > 0: #the task haven't finished
                    timetable[i][j] = run_task #assign the task for that resource in that hour
                    work_remain[run_task] -= resource_power[i] #task remain after running one hour
                    if work_remain[run_task] <= 0: #that task is done
                        print(run_task+' Done')
                        for k in RT_list:  #remove all that finished task from the waiting list of ALL resources
                            if run_task in RT_list[k]:
                                RT_list[k].remove(run_task)
                else:
                    print('error')

timetable.to_csv('scheduling-07.csv')
```

The reason why we can not use Spark to implement the last step is because of the step “remove all that finished task from the waiting list of ALL resources” in the code. “ALL resources” imply the first for loop, which means if we remove something, it's going to affect the next for loop execution, therefore, the for loop must execute sequentially. It’s not a for loop that repeats the same action many times, so cannot be parallelized, which also means we have no reason to use Spark.

VI. Output the scheduling file and save it onto HDFS.

Scheduling file :

A	B	C	D	E	F	G	H	I	J	K	L	M	N
	R9	R25	R35	R49	R73	R92	R94	R113	R74	R88	R95	R105	R109
2022/7/8 16:00	offline	offline	T1218	T3565	T2349	T843	T623	offline	T420	T1561	T1158	T2725	offline
2022/7/8 17:00	offline	offline	T1218	T3565	T2349	T843	T623	offline	T420	T1561	T1158	T2725	offline
2022/7/8 18:00	offline	offline	T1218	T3565	T2349	T843	T623	offline	T420	T1561	T1158	T2725	offline
2022/7/8 19:00	offline	offline	T1218	T3565	T2349	T843	T623	offline	T420	T1561	T1158	T2725	offline
2022/7/8 20:00	offline	offline	T1218	T3565	T2349	T843	T623	offline	T420	T1561	T1158	T2725	offline

In this file, what all resources should do every hour is recorded in great detail.

Save it to disk (HDFS)

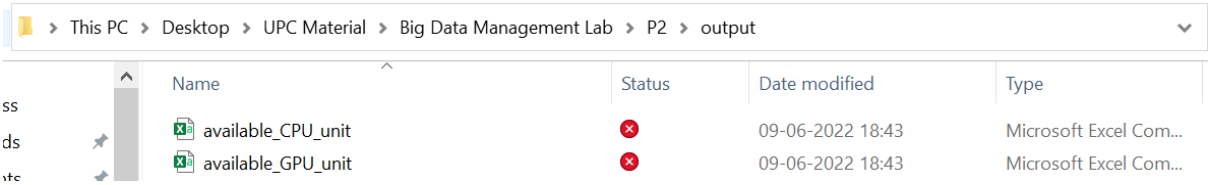
-rw-r--r--	bdm	supergroup	1.26 MB	Jun 09 18:25	3	128 MB	scheduling-05.csv	
-rw-r--r--	bdm	supergroup	1.93 MB	Jun 09 18:25	3	128 MB	scheduling-06.csv	
-rw-r--r--	bdm	supergroup	3.25 MB	Jun 09 18:25	3	128 MB	scheduling-07.csv	

The code of this scheduling algorithm is ***schedulingAlgo.ipynb*****.**

6. Visualization

For data visualization, we needed to process the files. As the generated schedule only gave us the available time for the resources, we can't use this data to answer the KPIs that we have determined, hence, as mentioned in the 'Data Processing for Visualization' section in the schedulingAlgo.ipynb we are processing the input files wrt the generated schedule and get some meaningful data out of it. After this, we are saving these processed files at location - "`/home/bdm/P2/viz`" on the VM.

We were able to establish the connection between the Tableau and the HDFS, we were not able to view the files, hence, we downloaded these processed files to a local folder (as shown below) and then uploaded them in Tableau for visualization.

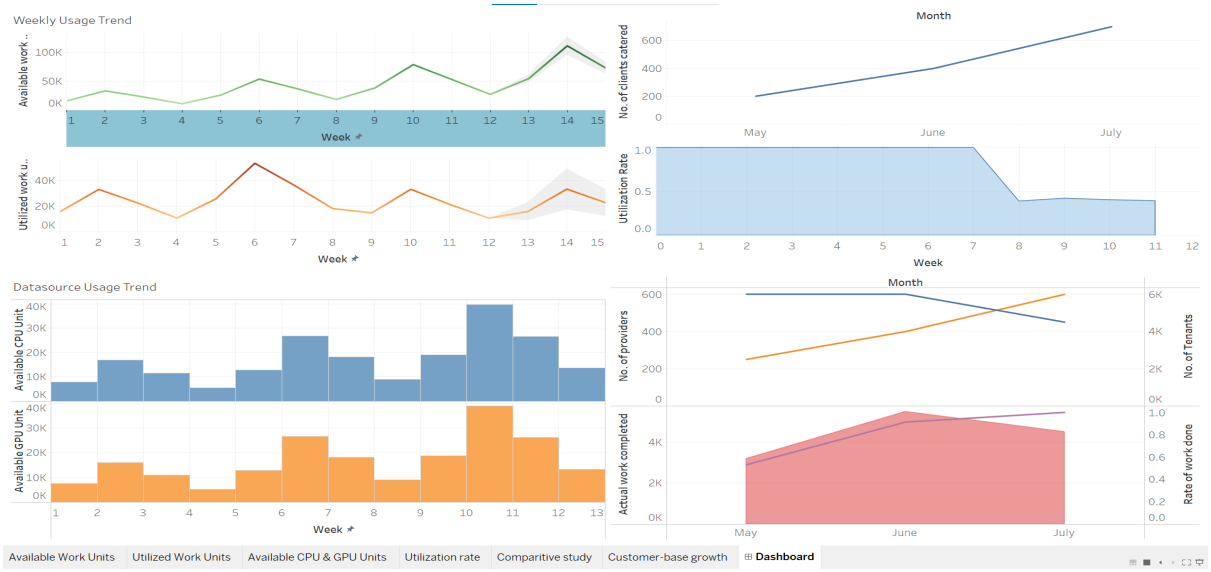


From the business perspective, it is important to know the status of the resource availability and usage. Hence with the help of visualization we are trying to answer the company KPI's as below :

1. How is the weekly availability of the work units?
2. How is the weekly utilization of the work units?
3. How is the weekly availability of the resources i.e. CPU/GPU units?
4. How is the utilization rate of the resources?
5. How many providers and tenants subscribed to IdleCompute in the last 3 months? How is the trend going?
6. How is the comparative study between the actual work completed and rate at which the work is done?
7. How as a company is our customer base growing in the last 3 months?

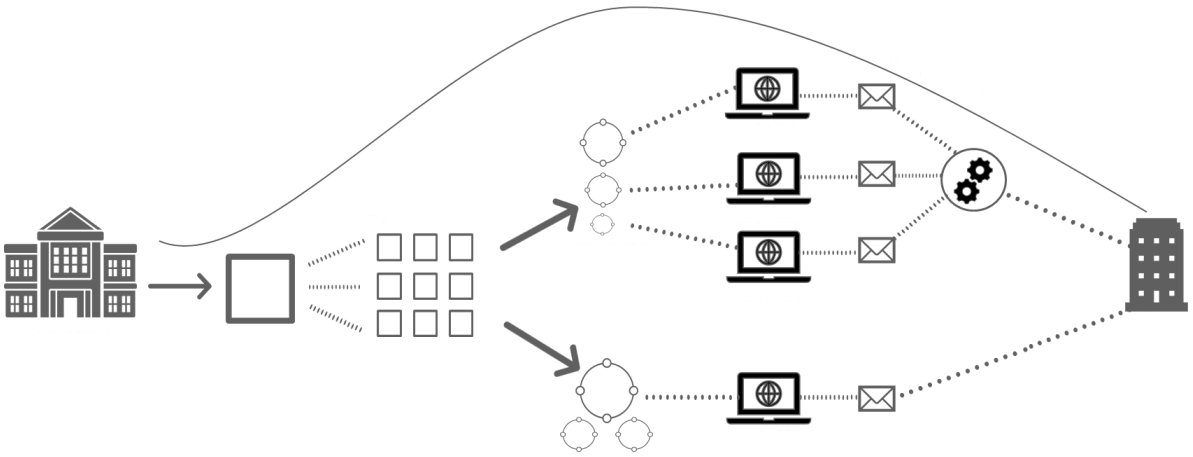
As mentioned above, to answer the KPI's mentioned above we use descriptive analysis in Tableau. There is a feature in the Tableau for forecasting hence, while actually accessing the availability and the utilization of the work units, we have also forecasted the values for the next 3 weeks.

The complete visualization of the data is represented on the dashboard as shown below:

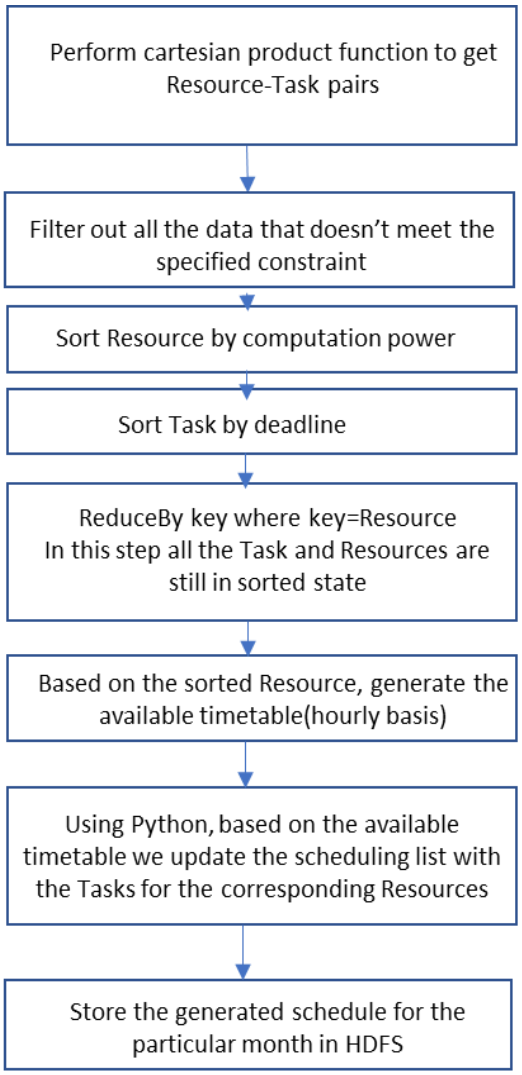


The whole visualization can be viewed in the submitted file named ***P2_viz.twbx***.

Appendix A :



Appendix B :



Appendix C :

Go to this site:

https://virtech.fib.upc.edu

OpenNebula login with:

user: masterBD13

password: dWqkPqLj13

Once in the dashboard, setup VM (press green +, name machine, keep default settings)

password: 123

In the local terminal: to connect and interact with the VM. IP address can be found in the OpenNebula dashboard. This will turn this instance of your local terminal into the VM terminal.

Make sure the VM status is "running" in OpenNebula.

ssh bdm@host_ip_address

ssh bdm@10.4.41.85

password: 123

In VM terminal: Start HDFS

/home/bdm/BDM_Software/hadoop/sbin/start-dfs.sh

In local internet browser: Check HDFS status in browser (Chrome)

http://10.4.41.85:9870

Once, in VM you would be able to view all the source files that we created for the project :

/home/bdm/python/data_gen.ipynb → Data generator file

/home/bdm/P2/schedulingAlgo.ipynb → Scheduling Algorithm

/home/bdm/python/PyHdfsConnect_upload.py → Uploading the files in HDFS

As the scheduling file is a Jupyter notebook you can directly run each cell.

For the PyHdfsConnect_upload.py file, please use the below command in the terminal:

python PyHdfsConnect_upload.py

****

For running the code some dependencies needs to be installed as listed below :

import HDFS

import pyspark