صبا سحبان ۹۹۳۱۰۹۶

گزارش تمرین عملی ۳

بخش اول

دایکسترا:

هدف پروژه پیدا کردن کوتاه ترین مسیر از یک ند به سایر ند ها با استفاده از الگوریتم دایکسترا است. حال میخواهیم دایکسترا را با mapreduce پیاده سازی کنیم.

Mapper.py:

```
def mapper(input_graph: Dict) -> List:
    output = []

    for node in input_graph.items():
        nid = node[0]
        distance = node[1][0]
        neighbors = node[1][1]
        path = node[1][2]

        if len(path) > 0 and path[len(path) - 1] != nid:
            path.append(nid)
        elif len(path) == 0:
            path.append(nid)

        output.append([nid, distance, neighbors, path])
        path = node[1][2]
        if len(neighbors) > 0:
            for neighbor in neighbors:
                if neighbor[0] is nid:
                    continue
                neighbor_path = path[:] + [neighbor[0]]
                neighbor_distance = distance + neighbor[1]
                        output.append([neighbor[0], neighbor_distance, None,
neighbor_path])

    return output
input_stream = sys.stdin.read().strip().replace('\n', '')

input_graph = ast.literal_eval(input_stream)

result = mapper(input_graph)

print(shuffle_and_sort(result))
```

این تابع Mapper یک گراف ورودی را دریافت می‌کند که به صورت یک دیکشنری نمایش داده شده است. برای هر نود در گراف ورودی، اطلاعات مربوط به نود شامل شناسه نود، فاصله از منبع، همسایگان و مسیر کنونی به آن نود استخراج می‌کند.

سپس، برای هر نود، تصمیمات زیر را انجام می‌دهد:
- اگر مسیری وجود داشته باشد و آخرین نود در مسیر نود فعلی نباشد، نود فعلی را به مسیر اضافه می‌کند.
- در صورتی که مسیر خالی باشد، مسیر را با نود فعلی مقداردهی اولیه می‌کند.
- سپس، اطلاعات نود فعلی را به لیست خروجی (`output`) اضافه می‌کند.
- در صورت وجود همسایگان برای نود، مسیر و فاصله به همسایگان را محاسبه کرده و به لیست خروجی اضافه می‌کند.

در نهایت، داده‌های خروجی ایجاد شده توسط تابع Mapper به وسیله تابع `shuffle_and_sort` مرتب و آماده‌سازی می‌شوند تا برای مراحل بعدی الگوریتم Dijkstra در چارچوب Hadoop آماده شوند.

Reducer.py:

```
import sys
from typing import Dict


def reducer(input_graph: Dict) -> Dict:
   output = {}
   for node in input_graph.items():
       min_distance = float('inf')
       nid = node[0]
       neighbors = []
       path = []
       for possibility in node[1]:
           distance = possibility[0]
           if possibility[1] is not None:
               neighbors = possibility[1]
           if distance < min_distance:
               min_distance = distance
               path = possibility[2]
       output[nid] = [
           min_distance,
           neighbors,
           path
       ]
   return output


input_stream = sys.stdin.read().strip().replace('\n', '')

input_graph = ast.literal_eval(input_stream)

result = reducer(input_graph)

# Emit
print(result)
```

این تابع Reducer یک گراف ورودی را دریافت می‌کند که به صورت یک دیکشنری نمایش داده شده است. برای هر نود در گراف ورودی، تابع Reducer کوتاه‌ترین فاصله را از منبع تا هر نود، همسایگان و مسیر کوتاهترین فاصله تا هر نود را محاسبه می‌کند.

- ابتدا، یک دیکشنری خروجی (`output`) برای نگهداری اطلاعات محاسباتی ایجاد می‌کند.
- سپس، برای هر نود در گراف ورودی، تمامی احتمالات و مسیرهای ممکن برای رسیدن به هر نود را بررسی می‌کند.
- با استفاده از اطلاعات احتمالات محاسبه شده برای هر نود، کوتاه‌ترین فاصله را به همراه همسایگان و مسیر مربوطه برای رسیدن به هر نود محاسبه می‌کند.
- سپس این اطلاعات را در دیکشنری خروجی (`output`) ذخیره می‌کند، که شامل اطلاعات کوتاه‌ترین فاصله، همسایگان و مسیر به نودها است.

در نهایت، این دو الگوریتم را اگر تعداد کافی انجام دهیم(حداقل به اندازه قطر گراف) به جواب میرسیم.

[1]:
```
import dijkstra_hadoop as dh
```

•[20]:
```
graph_list = [
    [1, 2, 10],
    [1, 3, 5],
    [2, 3, 2],
    [2, 4, 1],
    [3, 5, 2],
    [3, 2, 3],
    [3, 4, 9],
    [4, 5, 4],
    [5, 1, 7],
    [5, 4, 6],
]
```

[22]:
```
dh.dijkstra(graph_list)
```

[22]:
```
{1: [0, [[2, 10], [3, 5]], [1]],
 2: [8, [[3, 2], [4, 1]], [1, 3, 2]],
 3: [5, [[5, 2], [2, 3], [4, 9]], [1, 3]],
 4: [9, [[5, 4]], [1, 3, 2, 4]],
 5: [7, [[1, 7], [4, 6]], [1, 3, 5]]}
```

پیج رنک:

mapper.py

```python
import sys

def process_node(node, value, adjacency_list=''):
    print(f'{node}\tNODE\t{value}\t{adjacency_list}')

    if adjacency_list:
        neighbors = adjacency_list.split(',')
        rank = value / len(neighbors)

        for neighbor in neighbors:
            print(f'{neighbor}\tVALUE\t{rank}\t{node}')

def mapper():
    for line in sys.stdin:
        parts = line.strip().split()

        node = parts[0]
        value = float(parts[1])
        adjacency_list = parts[2] if len(parts) > 2 else ''

        process_node(node, value, adjacency_list)

if __name__ == "__main__":
    mapper()
```

گزارش توابع Mapper برای پردازش داده‌های گرافی در فرآیند MapReduce

این کد Python یک تابع Mapper را به عنوان بخشی از فرآیند MapReduce برای پردازش داده‌های گرافی ارائه می‌دهد. این تابع مسئولیت تبدیل داده‌های ورودی را به فرمت مناسبی برای فرآیند MapReduce دارد.

عملکرد تابع Mapper

1. ورودی و خواندن داده‌ها: تابع `mapper` از ورودی استاندارد (`sys.stdin`) داده‌ها را به عنوان یک خط دریافت می‌کند و آن‌ها را به صورت خطوطی جدا شده دریافت می‌کند.

2. پردازش هر خط داده: برای هر خط در داده‌های ورودی، تابع خط را به صورت پاره‌هایی جدا کرده و مقادیر مربوط به نود، مقدار آن و لیست همسایگان را استخراج می‌کند.

3. فراخوانی تابع پردازش نود: با استفاده از اطلاعات استخراج شده از هر خط ورودی، تابع `process_node` را صدا می‌زند تا نود فعلی را با اطلاعات مربوطه (شامل شناسه نود، مقدار آن و لیست همسایگان) پردازش کند.

4. پردازش همسایگان و چاپ: در صورت وجود همسایگان برای نود فعلی، مقدار مقسوم بر پردازش شده برای همسایگان را محاسبه کرده و برای هر یک از همسایگان، یک خط خروجی تولید می‌کند که شامل شناسه همسایه، مقدار مقسوم بر و شناسه نود فعلی است.

5. تولید خروجی: تابع Mapper اطلاعات پردازش شده را به صورت مجزا چاپ می‌کند و این خروجی‌ها به عنوان خروجی برای فرآیند MapReduce در نظر گرفته می‌شوند.

این تابع Mapper از ورودی‌های خط به خط داده‌های گرافی را پردازش می‌کند و خروجی‌هایی در قالب جفت‌های کلید و مقدار تولید می‌کند که برای مرحله‌ی بعدی یعنی Reducer در فرآیند MapReduce قابل استفاده است.

Reducer.py:

```
#!/usr/bin/env python

import sys

def calculate_page_rank(alpha, graph_size, sum_ranks):
   return alpha / graph_size + (1 - alpha) * sum_ranks

def  process_node(current_page,  current_adjacency_list,  sum_ranks,  alpha,
graph_size):
   new_rank = calculate_page_rank(alpha, graph_size, sum_ranks)
   print(f'{current_page}\t{new_rank}\t{current_adjacency_list}')
   return None, '', 0.0

def reducer(graph_size):
   alpha = 0.8
   current_page = None
   current_adjacency_list = ''
   sum_ranks = 0.0

   for line in sys.stdin:
       page_id, node_type, *other_parts = line.strip().split('\t')

       if page_id != current_page:
           if current_page is not None:
                           current_page,  current_adjacency_list,  sum_ranks  =
process_node(
                       current_page, current_adjacency_list, sum_ranks, alpha,
graph_size
               )

           if node_type == 'NODE':
               current_page = page_id
                current_adjacency_list = other_parts[1] if len(other_parts) >
1 else ''
               sum_ranks = 0.0
```

```
        elif node_type == 'VALUE':
            sum_ranks += float(other_parts[0])

    if current_page is not None:
        process_node(current_page, current_adjacency_list, sum_ranks, alpha,
graph_size)

if __name__ == "__main__":
    with open('input.txt', 'r') as file:
        graph_size = sum(1 for line in file)
    reducer(graph_size)
```

تابع Reducer برای محاسبه PageRank در چارچوب MapReduce

این کد Python یک تابع Reducer را ارائه می‌دهد که برای محاسبه مقدار PageRank در چارچوب MapReduce استفاده می‌شود. این تابع Reducer مسئولیت ادغام و پردازش داده‌های گرافی را دارد تا مقدار PageRank برای هر نود محاسبه شود.

عملکرد تابع Reducer

1. ورودی و خواندن داده‌ها: این تابع از ورودی استاندارد (`sys.stdin`) داده‌ها را خط به خط دریافت می‌کند و آن‌ها را به عنوان خطوط جدا شده در نظر می‌گیرد.

2. پردازش خطوط و ادغام داده‌ها: برای هر خط داده، اطلاعات مربوط به شناسه نود، نوع نود و سایر بخش‌های داده را استخراج می‌کند.

3. محاسبه مقدار PageRank: با استفاده از اطلاعات استخراج شده از هر خط داده، تابع `process_node` را فرا می‌خواند تا مقدار PageRank برای نود جاری محاسبه شود و در خروجی چاپ شود.

4. محاسبات برای هر نود و چاپ مقدار PageRank: برای هر خط داده، تصمیمات زیر را انجام می‌دهد:
- اگر شناسه نود جاری با شناسه نود قبلی متفاوت بود، مقدار PageRank جدید برای نود قبلی محاسبه می‌شود و در خروجی چاپ می‌شود.
- اگر نوع نود "NODE" بود، اطلاعات مربوط به نود جدید استخراج می‌شود.
- اگر نوع نود "VALUE" بود، مقدار PageRank جدید با توجه به اطلاعات موجود و مقدار خوانده شده برای نود محاسبه می‌شود و به مجموع مقادیر PageRank اضافه می‌شود.

5. خاتمه و چاپ نتایج: پس از پایان گردش داده‌ها، اطلاعات آخرین نود و مقدار مربوطه PageRank به تابع `process_node` فراخوانی می‌شود تا محاسبات نهایی انجام شود و خروجی نهایی چاپ شود.

نتیجه

این تابع Reducer از داده‌های گرافی ورودی استفاده کرده و مقادیر PageRank مربوط به هر نود را محاسبه کرده و خروجی نهایی را برای استفاده در مراحل بعدی الگوریتم PageRank در چارچوب MapReduce تولید می‌کند.

```
sabasahban@sabas-MacBook-Pro pageRanking % cat input.txt | python mapper.py | sort | python reducer.py
A       0.36666666666666664     B,C
B       0.36666666666666664     A
C       0.36666666666666664
```

بخش دوم

کلاستر هدوپ را با اسکریپت داده شده ران میکنیم. و میبینیم که کانتینرها در داکرکامپوز ساخته میشوند.
نمایش کانتینرهای ایجاد شده:

```
sabasahban@sabas-MacBook-Pro CC_HW3 % docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED         STATUS                            PORTS
                    NAMES
42b7dc766a06   docker-historyserver      "/run.sh"                8 seconds ago   Up 3 seconds (health: starting)   0.0.0.0:8188->8188/tcp
                    historyserver
b72a456a810f   docker-resourcemanager    "/run.sh"                8 seconds ago   Up 4 seconds (health: starting)   0.0.0.0:8089->8088/tcp
                    resourcemanager
8725b62802b4   docker-nodemanager1       "/run.sh"                8 seconds ago   Up 3 seconds (health: starting)   0.0.0.0:8042->8042/tcp
                    nodemanager1
f5c8504ed060   spark-base                "/bin/sh -c ./start-…"   8 seconds ago   Up 4 seconds                      6066/tcp, 7077/tcp, 0.0.0.0:7001->7000/tcp, 0.0.0.0:9092
->8081/tcp          spark-worker2
ac1734e38181   docker-datanode1          "/run.sh"                8 seconds ago   Up 4 seconds (health: starting)   9864/tcp
                    datanode1
a057e5522ff7   spark-base                "/bin/sh -c ./start-…"   8 seconds ago   Up 5 seconds                      6066/tcp, 7077/tcp, 0.0.0.0:7100->7000/tcp, 0.0.0.0:9091
->8081/tcp          spark-worker1
9ead10ea943a   docker-datanode2          "/run.sh"                8 seconds ago   Up 5 seconds (health: starting)   9864/tcp
                    datanode2
352b20c3dd47   docker-namenode           "/run.sh"                8 seconds ago   Up 6 seconds (health: starting)   0.0.0.0:8020->8020/tcp, 0.0.0.0:9870->9870/tcp
                    namenode
d6f0540ef277   spark-base                "/bin/sh -c ./start-…"   8 seconds ago   Up 6 seconds                      6066/tcp, 0.0.0.0:7077->7077/tcp, 0.0.0.0:9090->8081/tcp
                    spark-master
08af164393d6   docker-jupyter-notebook   "/bin/sh -c 'jupyter…"   8 seconds ago   Up 6 seconds                      6066/tcp, 7077/tcp, 0.0.0.0:4040->4040/tcp, 0.0.0.0:8888
->8888/tcp, 8081/tcp   jupyter-notebook
```

وظیفه هر کدام از کانتینرهای هدوپ:

بله، الگوهای مختلف کانتینرها در محیط Hadoop را توضیح می‌دهم. هر کدام از این کانتینرها نقش‌ها و مسئولیت‌های خاصی در سیستم Hadoop دارند:

1. historyserver:

- سرور تاریخچه (History Server) در Hadoop است که اطلاعاتی مانند گزارش‌های اجرایی و تاریخچه کارهای انجام شده را نگهداری می‌کند.
- ارائه اطلاعاتی از اجرای کارهای MapReduce و امکاناتی مانند نمایش گزارشات و اطلاعات اجرایی.

2. resourcemanager:

- ResourceManager در Hadoop، مسئولیت مدیریت منابع و تخصیص آن‌ها به برنامه‌ها و کارهای اجرایی در cluster را دارد.
- تخصیص منابع بین برنامه‌های کاربردی بر اساس نیازهای منابع و نظارت بر NodeManagers.

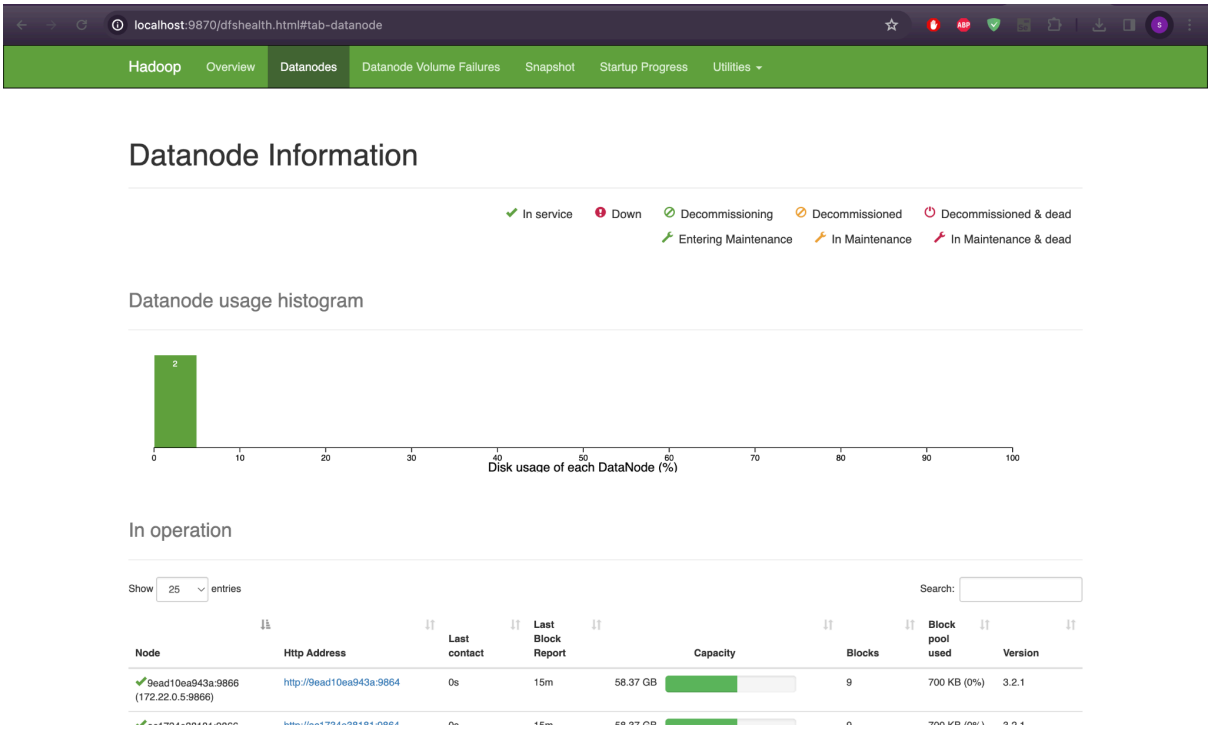
3. nodemanager1, datanode1, datanode2:
- NodeManager و DataNodeها نقش‌های اجرایی در Hadoop دارند.
- DataNodeها: ذخیره و مدیریت بلوک‌های داده‌ای در HDFS. داده‌ها به شکل توزیع شده روی چندین ند ذخیره میشود.
- NodeManagerها: اجرای وظایف بر روی نودها و ارتباط با ResourceManager.

4. namenode:
- NameNode در Hadoop، مسئولیت مدیریت فایل‌سیستم HDFS را بر عهده دارد.
- نگهداری Metadata در مورد مکان دقیق داده‌ها و ارتباط با DataNodeها برای مدیریت و دسترسی به داده‌ها.

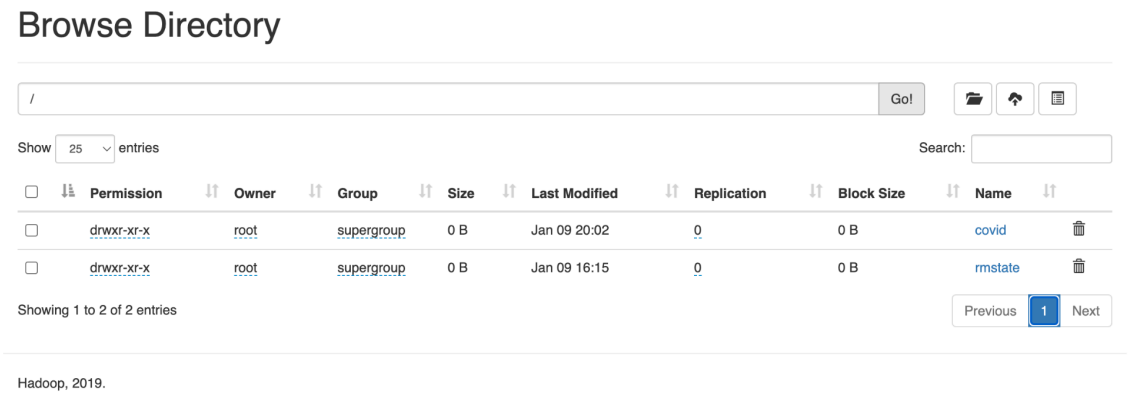
5. spark-base, spark-master, spark-worker1, spark-worker2:
- این کانتینرها به عنوان بخش‌هایی از Spark ecosystem عمل می‌کنند.
- Spark Master و Workerها مسئول اجرای برنامه‌های Spark و ایجاد محاسبات موازی بر روی cluster را دارند.

6. docker-jupyter-notebook:
- این کانتینر یک محیط Jupyter Notebook ارائه می‌دهد که به عنوان یک محیط تعاملی برای اجرای کدهای Python و دیگر زبان‌های برنامه‌نویسی استفاده می‌شود.
- برای تحلیل داده، توسعه و تست کدها در محیط Hadoop و Spark استفاده می‌شود.

نمایش UI:



نمایش فایل سیستم:



سوالات notebook:

توضیح DAG scheduler:

DAG (Directed Acyclic Graph) Scheduler در Apache Spark یکی از مؤلفه‌های اصلی آن است که مسئول مدیریت و برنامه‌ریزی اجرای عملیات‌هایی است که توسط کد برنامه نویسی شده تحت Spark انجام می‌شود. این Scheduler وظیفه مدیریت اجرای تسک‌ها (Tasks) را بر اساس یک نمایش گراف جهت‌دار و بدون حلقه (DAG) بر عهده دارد.

تعریف یک گراف جهت‌دار انجام عملیات‌هایی که باید انجام شوند و وابستگی‌های بین آن‌ها را مشخص می‌کند. این گراف شامل گره‌ها (نشان دهنده عملیات‌های مورد نیاز) و یال‌ها (نشان دهنده وابستگی بین عملیات‌ها) است.

DAG Scheduler در Spark ابتدا تمامی کدهای ارسالی برای اجرا را به صورت یک نمایش گرافی (DAG) از ترکیب مراحل و اعمالی که باید انجام شود، تحلیل می‌کند. این تحلیل شامل ترتیب اعمال و ارتباط بین آن‌ها در گراف است.

با توجه به گراف حاصل، DAG Scheduler وظیفه تقسیم گراف به بخش‌های کوچک‌تر و قابل اجرا (Stages) را دارد. این Stages شامل ترکیبی از عملیات‌هایی است که می‌توانند به صورت موازی اجرا شوند.

هر Stage شامل Taskهایی است که بر روی داده‌هایی که به صورت RDD یا DataFrame در Spark وجود دارند، اجرا می‌شوند. به عبارت دیگر، DAG Scheduler کار را به Task Scheduler منتقل کرده و Task Scheduler وظیفه اجرای Taskهای مربوطه را بر روی executorها در cluster بر عهده دارد.

به طور کلی، DAG Scheduler در Apache Spark مسئول تجزیه و تحلیل اعمالی است که باید انجام شوند و تبدیل آن‌ها به مراحل قابل اجرا (Stages) با Taskهای متناظر، برای اجرا در cluster است.

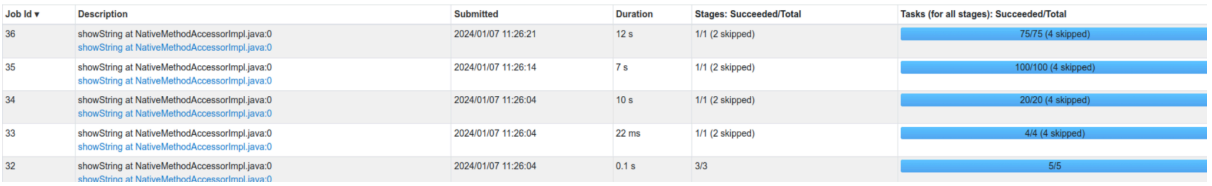
گزارش نحوه شافلینگ:
شافل یا Shuffling یکی از مراحل مهم در فرآیند پردازش داده در Apache Spark است که در هنگام اجرای عملیات‌هایی مانند join و groupBy اتفاق می‌افتد. در این فرآیند، داده‌ها بین اجزای مختلف کلاستر Spark جابجا می‌شوند تا داده‌های مرتبط با یکدیگر در یک نود یا پارتیشن مشخص قرار بگیرند.

زمانی که عملیاتی مانند join انجام می‌شود و دو دیتافریم مختلف که از طریق شبکه توزیع شده‌اند، ترکیب می‌شوند، اطلاعات مربوط به کلیدهای مشترک بین این دیتافریم‌ها نیاز به ترتیب‌بندی دارند. به همین دلیل، داده‌های مرتبط با یکدیگر باید در یک نود یا پارتیشن جمع‌آوری شده و گروه‌بندی شوند تا فرآیند ترکیب و ادغام آن‌ها صورت گیرد.

Shuffling به عنوان یک مکانیزم مهم در Spark، در بهبود عملکرد و اجرایی برنامه‌ها نقش اساسی دارد. این عملیات می‌تواند موجب افزایش ترافیک شبکه، مصرف منابع محاسباتی، و افزایش زمان اجرا شود. همچنین، بهینه‌سازی‌های مختلفی در Spark برای کاهش بار Shuffling و بهبود عملکرد در این بخش صورت گرفته است.

اطلاعات مربوط به Shuffle ممکن است در رابط‌های کاربری مانیتورینگ Spark مانند Application Master UI یا Spark UI مشاهده شود. این رابط‌های کاربری اطلاعاتی از جمله تعداد تسک‌های ShuffleMapTask و

ReduceTask، حجم داده‌های Shuffled، و سایر جزئیات مرتبط با عملکرد و انجام Shuffling در کلاستر Spark را نمایش می‌دهند.



Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
36	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/01/07 11:26:21	12 s	1/1 (2 skipped)	75/75 (4 skipped)
35	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/01/07 11:26:14	7 s	1/1 (2 skipped)	100/100 (4 skipped)
34	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/01/07 11:26:04	10 s	1/1 (2 skipped)	20/20 (4 skipped)
33	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/01/07 11:26:04	22 ms	1/1 (2 skipped)	4/4 (4 skipped)
32	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/01/07 11:26:04	0.1 s	3/3	5/5