فرهاد امان ۹۹۳۱۰۰۶

بخش اول: Hadoop

در این بخش ابتدا الگوریتم **Dijkstra** را با استفاده از شیوه Map-Reduce پیاده‌سازی می‌کنیم.

ابتدا تابع mapper را پیاده‌سازی می‌کنیم در تابع mapper، برای هر راس که اطلاعات مربوط به آن در یک خط قرار گرفته است، فاصله آن راس از مبدا تا راس‌های مجاور این راس را به دست می‌آوریم و سپس راس جدید و فاصله آن تا مبدا را به تابع reducer خود می‌دهیم. همچین باید ساختار گراف را همچنان حفظ کنیم در نتیجه راس فعلی را هم به reducer خواهیم داد.

این کار باعث می‌شود در هر مرحله از اجرای MapReduce، فاصله‌های موقتی از مبدا به سایر راس‌ها بروزرسانی شود. تابع mapper برای هر راس ورودی، فاصله فعلی را به عنوان بخشی از خروجی خود ارسال می‌کند. این اطلاعات به reducer منتقل می‌شود که وظیفه‌ی بروزرسانی فاصله‌ها و انتخاب کوتاه‌ترین فاصله برای هر راس را بر عهده دارد.

در الگوریتم دایکسترا، reducer وظیفه دارد که از میان فواصل ارسال شده برای هر راس، کمترین فاصله را انتخاب کند. این کار با مقایسه فواصل موقتی که از mapper ها ارسال شده‌اند، انجام می‌شود. هر reducer یک راس خاص را در نظر می‌گیرد و برای آن بهترین فاصله تا راس مبدأ را تعیین می‌کند.

عملکرد reducer

1. **جمع‌آوری داده‌ها:** برای هر راس، reducer تمام فواصل ارسال شده از mapperها را دریافت می‌کند.

2. **انتخاب کمترین فاصله:** از میان این فواصل، کمترین فاصله به عنوان فاصله نهایی راس مورد نظر از راس مبدأ انتخاب می‌شود.
3. **ذخیره ساختار گراف:** علاوه بر این، reducer باید اطلاعات مربوط به راس‌های مجاور و وزن یال‌های مرتبط با آن راس را نیز ذخیره کند. این اطلاعات معمولاً از داده‌های اولیه وارد شده به mapper حاصل می‌شود.
4. **بررسی تغییر فاصله:** اگر فاصله نهایی یک راس نسبت به حالت قبلی‌اش تغییر کرده باشد، این نشان‌دهنده آن است که باید یک دور دیگر MapReduce را اجرا کرد تا فواصل بروزرسانی شوند.

```
def mapper(self, _, line):
    # Parse each line from the input
    _, node, data = line.strip().split('"', 2)
    node = node.strip('"')
    data = self.my_eval(data)

    # Emit the node and its data
    yield node, data

    # Process and emit data for adjacent nodes
    if data['Distance'] != float('inf'):
        for key, value in data['AdjacencyList'].items():
            new_data = {'Distance': data['Distance'] + value}
            yield key, new_data
```

```python
def reducer(self, key, values):
    # Initialize variables for the shortest distance and adjacency list
    min_distance = float('inf')
    adjacency_list = None

    # Iterate over values to find the minimum distance
    for value in values:
        min_distance = min(min_distance, value['Distance'])
        if 'AdjacencyList' in value:
            adjacency_list = value['AdjacencyList']

    # Yield the key and updated data
    yield key, {'Distance': min_distance, 'AdjacencyList': adjacency_list}
```

در اینجا می‌توانید کدهای مربوط به mapper و reducer را ببینید.

هنگام اجرای این کدها با یک بار اجرای تسک کار تمام نمی‌شود و باید خروجی هر مرحله به مرحله بعدی داده شود تا زمانی که دیگر خروجی تغییر نکند.

حالا الگوریتم **Pagerank** را با استفاده از Map-Reduce پیاده‌سازی می‌کنیم.

پیاده‌سازی Mapper:

- در مرحله mapper، برای هر راس ورودی، اطلاعات مربوط به راس و لیست گره‌های مجاور آن پردازش می‌شود.
- برای هر گره مجاور، مقداری از PageRank گره فعلی بر اساس تعداد کل گره‌های مجاور تقسیم و به عنوان سهم PageRank به reducer فرستاده می‌شود.
- علاوه بر این، داده‌های ساختاری خود راس نیز به reducer فرستاده می‌شوند تا ساختار گراف حفظ شود.

پیاده‌سازی Reducer:

- در مرحله reducer، کل سهم‌های PageRank دریافتی برای هر گره جمع‌آوری می‌شود.
- با استفاده از فرمول PageRank، مقدار جدید PageRank برای هر گره محاسبه می‌شود
- راس‌های مجاور نیز در کنار مقدار جدید PageRank در فایل خروجی نوشته می‌شوند تا ساختار گراف حفظ شود.
- اگر تغییر در مقدار PageRank هر گره نسبت به مقدار قبلی بیش از مقدار آستانه (epsilon) باشد، فرآیند ادامه می‌یابد.

```
1 usage   Farhad Aman
def mapper(self, _, line):
    # Parse each line from the input
    _, node, data = line.strip().split('"', 2)
    node = node.strip('"')
    data = eval(data)

    # Emit the node and its structure
    yield node, ('node_data', data)

    # Emit PageRank contributions to adjacent nodes
    if data['AdjacencyList']:
        page_rank_contribution = data['PageRank'] / len(data['AdjacencyList'])
        for neighbor in data['AdjacencyList']:
            yield neighbor, ('page_rank_contribution', page_rank_contribution)
```

```
1 usage   Farhad Aman
def reducer(self, node, values):
    total_page_rank_contribution = 0
    node_data = None

    for value_type, value in values:
        if value_type == 'node_data':
            node_data = value
        else:
            total_page_rank_contribution += value

    # Calculate the new PageRank
    new_pagerank = 0.15 + 0.85 * total_page_rank_contribution
    node_data['PageRank'] = new_pagerank

    # Emit the node with updated PageRank and its structure
    yield node, node_data

    # Emit a special key for checking convergence
    if abs(new_pagerank - node_data['PageRank']) > EPSILON:
        yield 'convergence_check', 1
```

2. **چک کردن همگرایی و ادامه فرآیند:**
   - فرآیند MapReduce تا زمانی ادامه می‌یابد که تغییرات در مقادیر PageRank کمتر از مقدار آستانه (epsilon) باشد.
   - در هر دوره، خروجی مرحله قبلی به عنوان ورودی مرحله بعدی استفاده می‌شود.
   - اگر در یک دوره، هیچ یک از گره‌ها تغییری در مقدار PageRank بیش از epsilon نداشته باشند، فرآیند متوقف می‌شود.

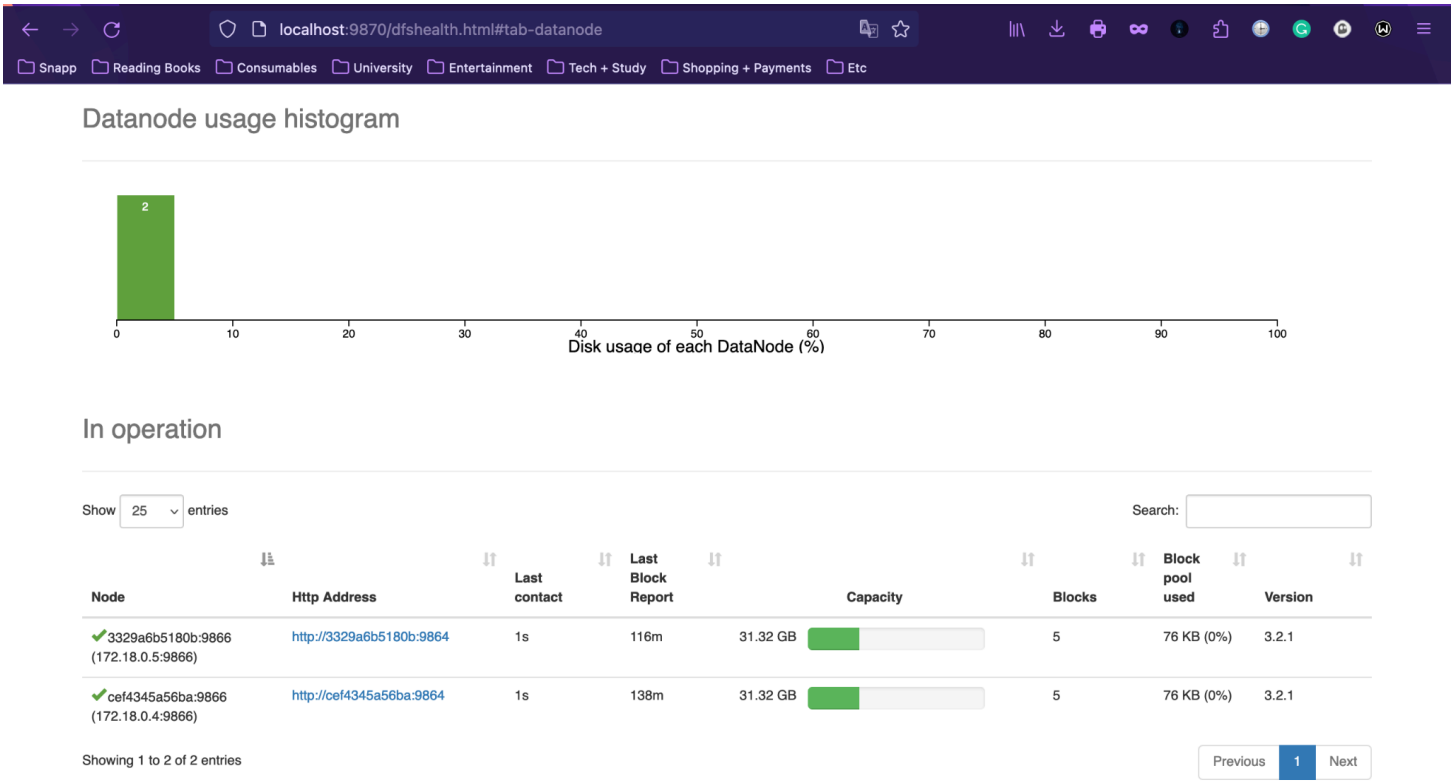
بخش دوم: Spark

```
~ (0.18s)
docker ps
CONTAINER ID   IMAGE                            COMMAND                  CREATED       STATUS                  PORTS
                                                           NAMES
927b0ac3387d   hadoop-spark-resourcemanager     "/run.sh"                3 hours ago   Up 3 hours (healthy)    0.0.0.0:8089->8
088/tcp                                                    resourcemanager
8cb0f3a700af   hadoop-spark-nodemanager1        "/run.sh"                3 hours ago   Up 3 hours (healthy)    0.0.0.0:8042->8
042/tcp                                                    nodemanager1
c3c4b01c49c1   spark-base                       "/bin/sh -c ./start-…"   3 hours ago   Up 3 hours              6066/tcp, 7077/
tcp, 0.0.0.0:7001->7000/tcp, 0.0.0.0:9092->8081/tcp   spark-worker2
5b97fdce3096   spark-base                       "/bin/sh -c ./start-…"   3 hours ago   Up 3 hours              6066/tcp, 7077/
tcp, 0.0.0.0:7100->7000/tcp, 0.0.0.0:9091->8081/tcp   spark-worker1
3329a6b5180b   hadoop-spark-datanode2           "/run.sh"                3 hours ago   Up 3 hours (healthy)    9864/tcp
                                                           datanode2
cef4345a56ba   hadoop-spark-datanode1           "/run.sh"                3 hours ago   Up 3 hours (healthy)    9864/tcp
                                                           datanode1
59efdd3b7efe   hadoop-spark-namenode            "/run.sh"                3 hours ago   Up 3 hours (healthy)    0.0.0.0:8020->8
020/tcp, 0.0.0.0:9870->9870/tcp                            namenode
ada224140789   spark-base                       "/bin/sh -c ./start-…"   3 hours ago   Up 3 hours              6066/tcp, 0.0.0
.0:7077->7077/tcp, 0.0.0.0:9090->8081/tcp                  spark-master
```

1. hadoop-spark-resourcemanager:
   - این کانتینر مدیر منابع (ResourceManager) در معماری YARN است.
   - وظیفه آن مدیریت منابع و تخصیص آن‌ها به برنامه‌های مختلف در خوشه است.
   - این کانتینر وضعیت سلامتی (healthy) دارد و پورت 8089 به پورت 8080 متصل شده است که معمولاً برای دسترسی به وب‌سرور UI مدیر منابع استفاده می‌شود.
2. hadoop-spark-nodemanager1:
   - NodeManager نقش worker در معماری YARN را دارد.
   - مسئولیت آن مدیریت منابع در سطح یک نود و اجرای کانتینرهای برنامه است.
3. spark-workers:
   - این کانتینرها workerهای اسپارک هستند.
   - آن‌ها مسئول پردازش داده‌ها و اجرای تسک‌ها را دارند.
   - پورت‌های 7001، 7000 برای ارتباط داخلی و 8081 برای دسترسی به وب‌سرور worker اسپارک تنظیم شده‌اند.

4. hadoop-spark-datanode1 و hadoop-spark-datanode2:
   - این کانتینرها نقش دیتانودها در HDFS را ایفا می‌کنند.
   - وظیفه آن‌ها ذخیره‌سازی داده‌ها و ارائه داده به برنامه‌های پردازشی است.
   - هر دو کانتینر وضعیت سلامت دارند و پورت 9864 برای ارتباط داخلی با سایر کامپوننت‌های Hadoop استفاده می‌شود.
5. hadoop-spark-namenode:
   - کانتینر namenode در HDFS است.
   - وظیفه آن مدیریت namespace فایل‌ها و دایرکتوری‌ها و کنترل دسترسی به فایل‌ها است.
   - این کانتینر هم وضعیت سلامت دارد و پورت 8020 برای IPC مورد استفاده قرار می‌گیرد.
6. spark-master:
   - این کانتینر نقش مستر در خوشه اسپارک را ایفا می‌کند.
   - مسئولیت آن کنترل workerهای اسپارک و توزیع تسک‌ها بین آن‌ها است.
   - پورت 7077 برای ارتباط با کارگزاران و پورت 8080 برای دسترسی به وب‌سرور UI مستر اسپارک است.

هر کدام از این کانتینرها یک قطعه حیاتی از معماری Hadoop و Spark را تشکیل می‌دهند و برای اجرای موفقیت‌آمیز یک خوشه پردازش داده توزیع‌شده ضروری هستند.



در Apache Spark، شافلینگ یک فرآیند حیاتی است که در آن داده‌ها بر اساس کلیدهای خاصی توزیع و از یک نود به نودهای دیگر منتقل می‌شوند. این کار برای اجرای عملیات موازی مانند گروه‌بندی یا مرتب‌سازی ضروری است. شافلینگ باعث می‌شود داده‌ها در میان نودهای مختلف پراکنده شوند، به طوری که هر نود می‌تواند بخشی از کار را به صورت مستقل انجام دهد. این فرآیند ممکن است بر عملکرد کلی سیستم تأثیر بگذارد، زیرا نیازمند انتقال حجم زیادی از داده‌ها بین نودها است. بنابراین، بهینه‌سازی شافلینگ یک جزء کلیدی در بهبود عملکرد برنامه‌های Spark محسوب می‌شود.

DAG

تعریف: DAG یک گراف جهت‌دار و بدون دور است که نمایشی از توالی عملیات‌های اعمال شده بر روی RDDها است.

کارکرد: هر گره در DAG نمایانگر یک عملیات RDD می‌باشد و یال‌ها جریان داده‌ها را بین این عملیات‌ها نشان می‌دهند. DAG به Spark این امکان را می‌دهد که به طور موثر تعیین کند چگونه داده‌ها باید در طول کلاستر پردازش شوند.

اهمیت: استفاده از DAG به جای مدل‌های سنتی MapReduce، به Spark اجازه می‌دهد تا مراحل مختلف را بدون نیاز به نوشتن داده‌ها به دیسک در هر مرحله، بهینه‌سازی کند.

DAG Scheduler

تعریف: DAG Scheduler بخشی از سیستم زمان‌بندی (scheduling) در Spark است که وظیفه تبدیل کردن یک DAG به یک سری از مراحل (stages) و اجرای آن‌ها را دارد.

کارکرد:

- **تبدیل DAG به مراحل:** DAG Scheduler، DAG را به چندین مرحله تقسیم می‌کند که هر کدام شامل تعدادی از تغییرات RDD هستند که می‌توانند بدون نیاز به داده‌هایی از نودهای دیگر، به صورت موازی اجرا شوند.

- **مدیریت وابستگی‌ها**: DAG Scheduler وابستگی‌های بین مراحل را مدیریت می‌کند تا اطمینان حاصل شود که مراحل در ترتیب صحیح اجرا می‌شوند.
- **کنترل خطا و بازیابی**: در صورت بروز خطا در اجرای یک مرحله، DAG Scheduler قادر است تنها آن بخش از DAG که نیاز به بازیابی دارد را تشخیص دهد و دوباره اجرا کند.

اهمیت: DAG Scheduler به Spark این توانایی را می‌دهد که به طور موثر و کارآمد، محاسبات پیچیده را در مقیاس بزرگ مدیریت کند، بازیابی از خطاها را ساده‌تر کند و در نهایت عملکرد کلی سیستم را بهبود ببخشد.