

Part 9. Apache Spark

스파크 개요

Apach Spark란?

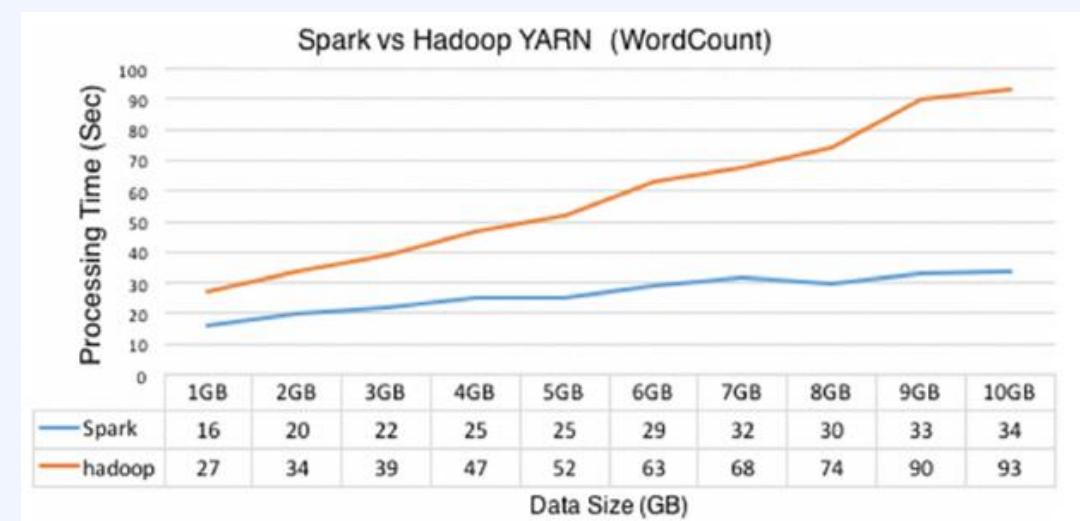
Apache Spark란

데이터 센터나 클라우드에서 대규모 분산 데이터 처리를
하기 위해 설계된 통합형 엔진

Apache Spark의 주요 설계 철학

1. 속도

- a. 디스크 I/O를 주로 사용하는 하둡 맵리듀스 처리 엔진과 달리, 중간 결과를 메모리에 유지하기 때문에 훨씬 더 빠른 속도로 같은 작업을 수행 가능.
- b. 질의 연산을 방향성 비순환 그래프(DAG - Directed acyclic graph)로 구성.
 - DAG의 스케줄러와 질의 최적화 모듈은, 효율적인 연산 그래프를 만들고 각각의 태스크로 분해하여, 클러스터의 워커 노드 위에서 병렬 실행될 수 있도록 함.
- c. 물리적 실행 엔진인 텅스텐은 전체 코드 생성 기법을 사용해, 실행을 위한 간결한 코드를 생성.
 - a. 텅스텐 프로젝트 : <https://www.databricks.com/kr/glossary/tungsten>



Apache Spark의 주요 설계 철학

2. 사용 편리성

- a. 클라이언트 입장에서, 스파크 코드가 단일 pc, 혹은 분산 환경에서 실행되는지 구별하기 어려울 정도로 추상화가 잘 되어 있음.
- b. Dataframe, Dataset과 같은 고수준 데이터 추상화 계층 아래에, RDD (Resilient distributed dataset)라 불리는 단순한 자료구조를 구축해 단순성을 실현함.
- c. 연산의 종류로, 트랜스포메이션(transformation)과 액션(action)의 집합과 단순한 프로그래밍 모델을 제공.
- d. 여러 프로그래밍 언어(Scala, Java, Python, Kotlin, R 등)을 제공하여, 사용자들이 각자 편한 언어로 스파크 애플리케이션을 작성할 수 있음.

Apache Spark의 주요 설계 철학

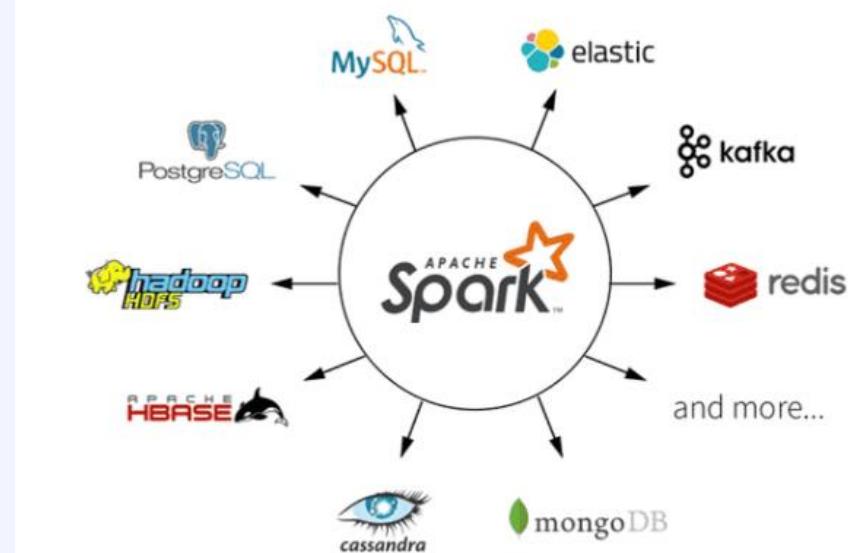
3. 모듈성

- a. 스파크에 내장된 다양한 컴포넌트(SparkSQL, Structured Streaming, MLlib 등)들을 사용해, 다양한 타입의 워크로드에 적용이 가능함.
- b. 특정 워크로드를 처리하기 위해 하나의 통합된 처리 엔진을 가짐.
 - 하둡 맵리듀스의 경우, 배치 워크로드에는 적합하나 SQL 질의, 스트리밍, 머신 러닝 등 다른 워크로드와 연계해 사용하기에는 어려움 존재.
 - 이런 워크로드를 다루기 위해 엔지니어들은 하둡과 함께, Apache Hive (SQL 질의), Storm(스트리밍), Mahout(머신러닝) 등 다른 시스템과의 연동이 필요.
 - 이들은 각각 자신만의 API, 설정 방식 등을 가지고 있기 때문에, 모듈성이 떨어지고 개발자가 배우기 어려움.

Apache Spark의 주요 설계 철학

4. 확장성

1. 저장과 연산을 모두 포함하는 하둡과는 달리, 스파크는 빠른 병렬 연산에 초점.
2. 수많은 데이터 소스(하둡, 카산드라, Hbase, 몽고DB, hive, RDBMS, AWS S3 등)로부터 데이터를 읽어 들일 수 있음.
3. 여러 파일 포맷(txt, csv, parquet, roc, hdfs 등)과 호환 가능.
4. 이 외에 많은 서드파티 패키지 목록 사용 가능



로컬 환경에 스파크 설치 및 워드 카운트 예제 실행 (MAC)

1. 로컬 스파크 구축

1) brew를 이용한 Java 설치

- Spark 구동 시 java는 꼭 설치 되어 있어야 함.
- brew install : https://brew.sh/index_ko
- brew install 후 터미널에서 아래의 순서로 명령어 입력

```
$ brew update
```

```
$ brew tap adoptopenjdk/openjdk # adoptopenjdk/openjdk 레포지토리 추가
```

```
$ brew search jdk # 다운로드 가능한 jdk 버전 확인
```

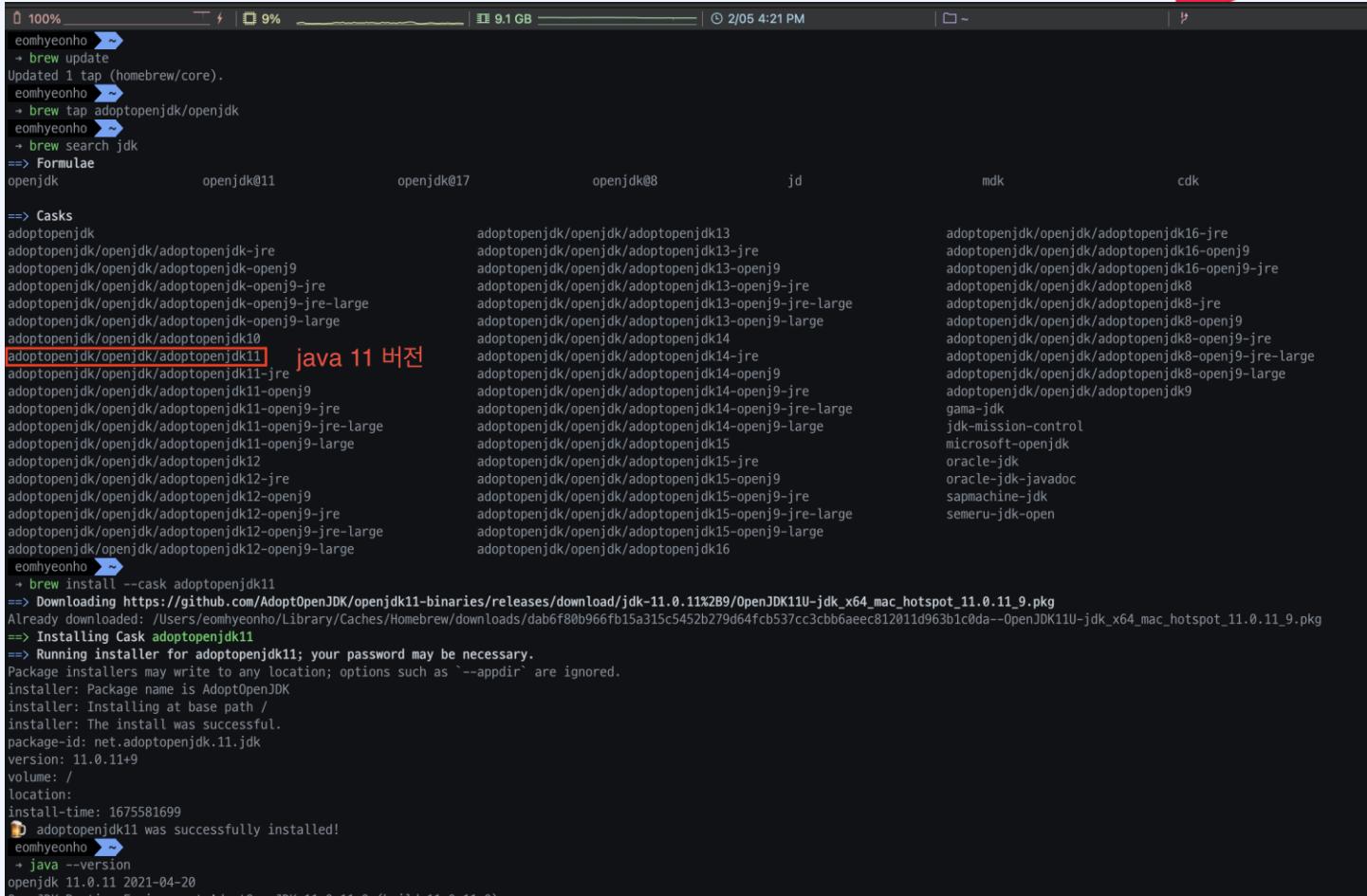
```
$ brew install --cask adoptopenjdk11 # java 11 다운로드
```

```
$ java --version # 자바 버전 확인
```

1. 로컬 스파크 구축

1) brew를 이용한 Java 설치

- screenshot



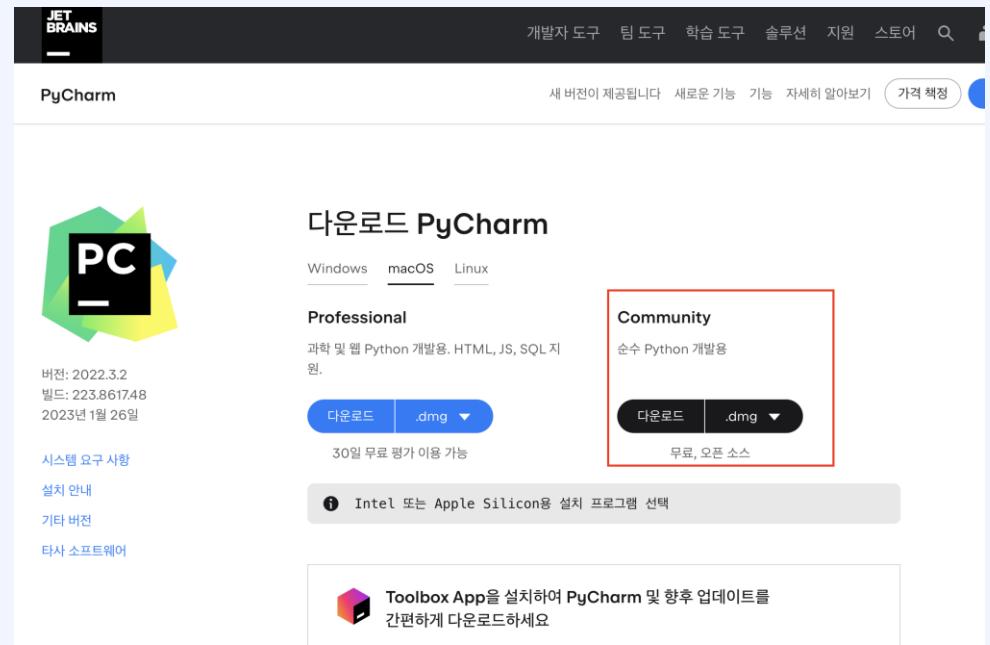
```
brew update
brew tap adoptopenjdk/openjdk
brew search jdk
Formulae
openjdk      openjdk@11      openjdk@17      openjdk@8      jdk      mdk      cdk
--> Casks
adoptopenjdk
adoptopenjdk/openjdk/adoptopenjdk-jre
adoptopenjdk/openjdk/adoptopenjdk-openj9
adoptopenjdk/openjdk/adoptopenjdk-jre-large
adoptopenjdk/openjdk/adoptopenjdk-openj9-large
adoptopenjdk/openjdk/adoptopenjdk@10
adoptopenjdk/openjdk/adoptopenjdk@11 [java 11 버전]
adoptopenjdk/openjdk/adoptopenjdk@11-jre
adoptopenjdk/openjdk/adoptopenjdk@11-openj9
adoptopenjdk/openjdk/adoptopenjdk@11-jre-large
adoptopenjdk/openjdk/adoptopenjdk@11-openj9-large
adoptopenjdk/openjdk/adoptopenjdk@12
adoptopenjdk/openjdk/adoptopenjdk@12-jre
adoptopenjdk/openjdk/adoptopenjdk@12-openj9
adoptopenjdk/openjdk/adoptopenjdk@12-jre-large
adoptopenjdk/openjdk/adoptopenjdk@12-openj9-large
eomhyeonho >
+ brew install --cask adoptopenjdk11
==> Downloading https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk-11.0.11+289/OpenJDK11U-jdk_x64_mac_hotspot_11.0.11_9.pkg
Already downloaded: /Users/eomhyeonho/Library/Caches/Homebrew/downloads/dab6f80b966fb15a315c5452b279d64fc5b37cc3bb6aeec812011d963b1c0da--OpenJDK11U-jdk_x64_mac_hotspot_11.0.11_9.pkg
==> Installing Cask adoptopenjdk11
==> Running installer for adoptopenjdk11; your password may be necessary.
Package installers may write to any location; options such as `--appdir` are ignored.
installer: Package name is AdoptOpenJDK
installer: Installing at base path /
installer: The install was successful.
package-id: net.adoptopenjdk.11.jdk
version: 11.0.11+9
volume: /
location:
install-time: 1675581699
adoptopenjdk11 was successfully installed!
eomhyeonho >
+ java --version
openjdk 11.0.11 2021-04-20
Java(TM) SE Runtime Environment 11.0.11+9 (build 11.0.11+9)
Java HotSpot(TM) 64-Bit Server VM 11.0.11+9 (build 11.0.11+9)
```

1. 로컬 스파크 구축

2) Python 가상 환경 구성, pyspark 라이브러리 설치

1. 아래 링크에서 PyCharm Community Edition 설치

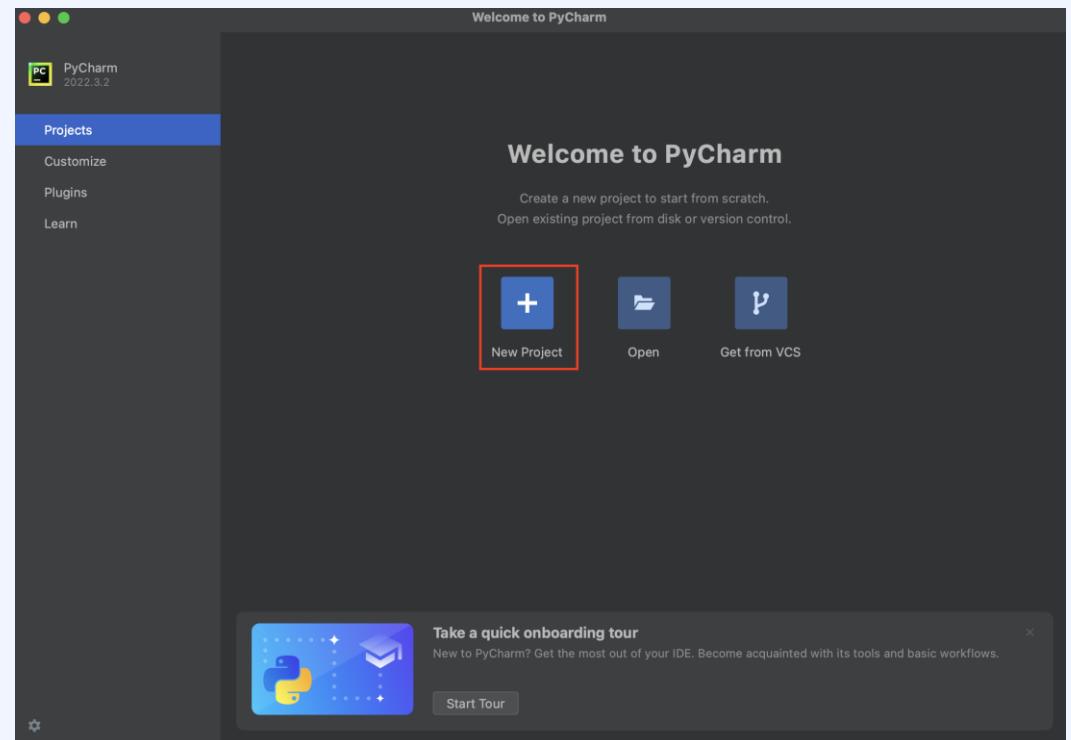
<https://www.jetbrains.com/ko-kr/pycharm/download/#section=mac>



1. 로컬 스파크 구축

2) Python 가상 환경 구성, pyspark 라이브러리 설치

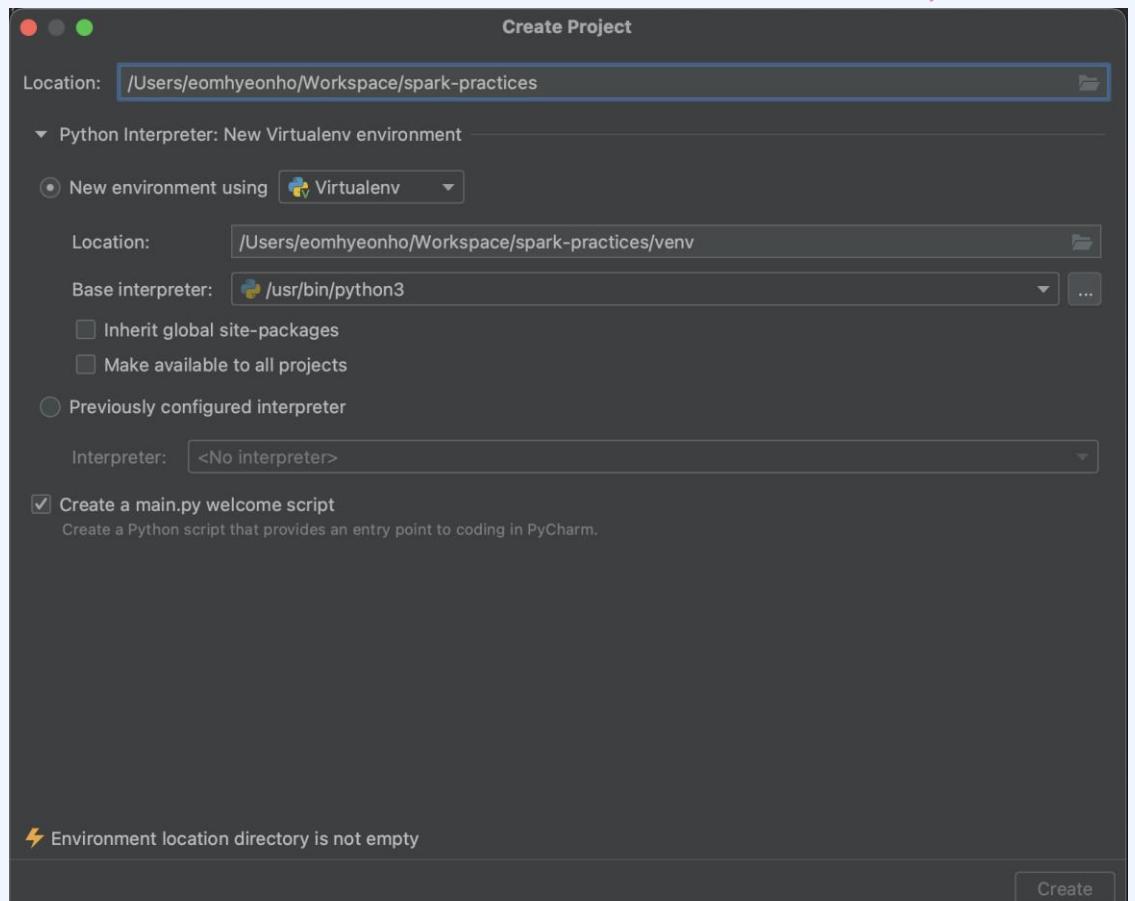
2. Pycharm 다운로드 완료 후 New Project 선택



1. 로컬 스파크 구축

2) Python 가상 환경 구성, pyspark 라이브러리 설치

3. 프로젝트의 Location 설정 및 virtualenv를 사용해,
이 프로젝트를 위한 Python 가상 환경 설정



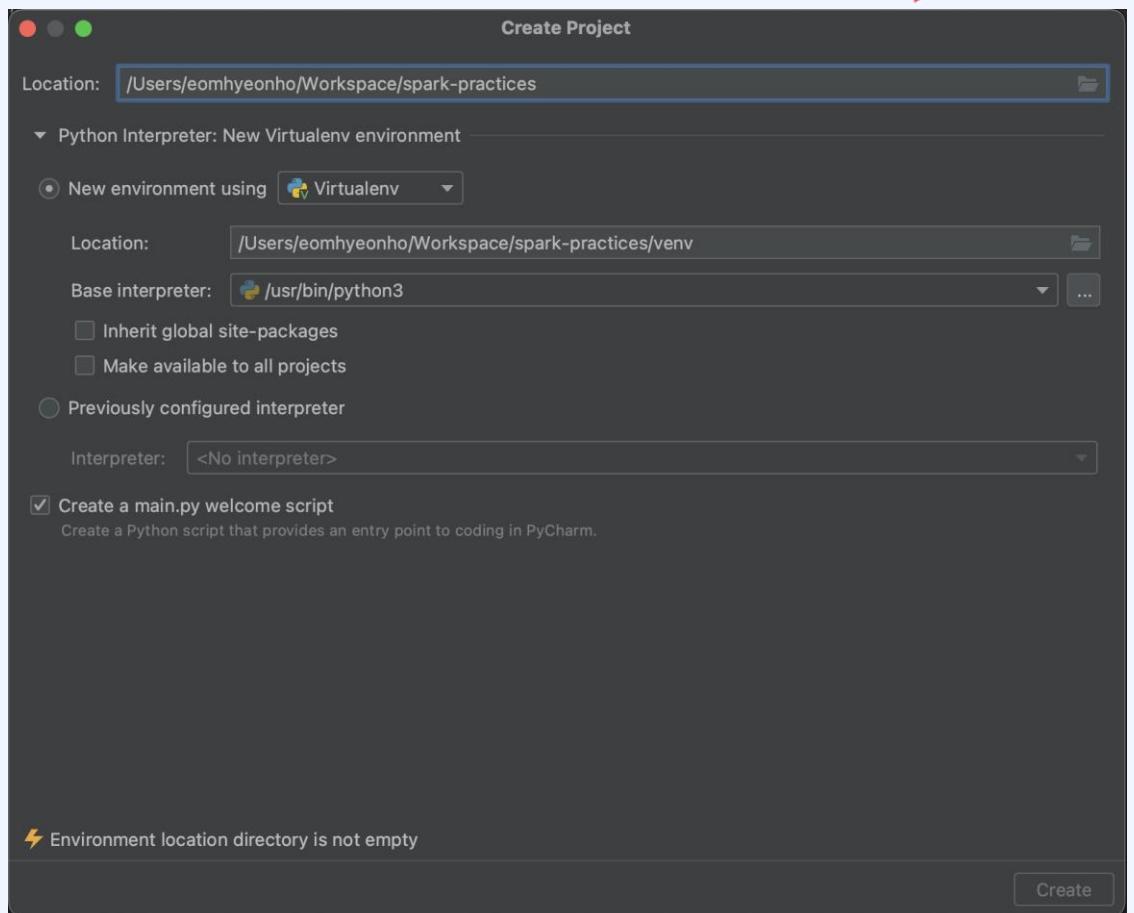
1. 로컬 스파크 구축

2) Python 가상 환경 구성, pyspark 라이브러리 설치

3. 프로젝트의 Location 설정 및 virtualenv를 사용해,
이 프로젝트를 위한 Python 가상 환경 설정

location)

/Users/{macbook username}/Workspace/spark-practices
(꼭 위 형식으로 경로를 지정할 필요 x)

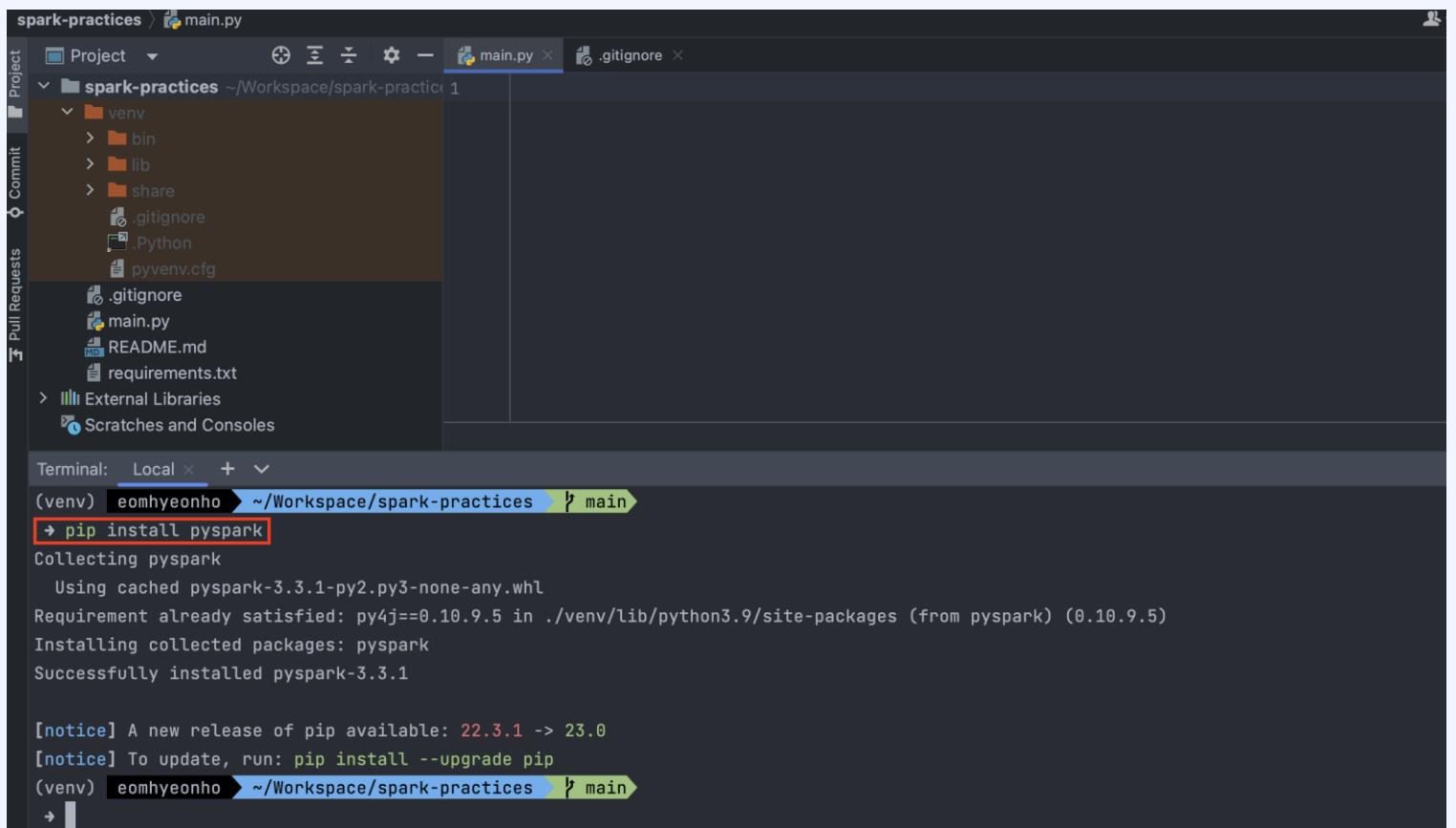


1. 로컬 스파크 구축

2) Python 가상 환경 구성, pyspark 라이브러리 설치

4. 터미널에서 pyspark 설치

```
$ pip install pyspark
```



```
spark-practices / main.py
Project: spark-practices ~/Workspace/spark-practices 1
  venv
    bin
    lib
    share
      .gitignore
      Python
      pyvenv.cfg
    .gitignore
    main.py
    README.md
    requirements.txt
  External Libraries
  Scratches and Consoles

Terminal: Local × + ▾
(venv) eomhyeonho ➜ ~/Workspace/spark-practices ↵ main>
→ pip install pyspark
Collecting pyspark
  Using cached pyspark-3.3.1-py2.py3-none-any.whl
Requirement already satisfied: py4j==0.10.9.5 in ./venv/lib/python3.9/site-packages (from pyspark) (0.10.9.5)
Installing collected packages: pyspark
Successfully installed pyspark-3.3.1

[notice] A new release of pip available: 22.3.1 -> 23.0
[notice] To update, run: pip install --upgrade pip
(venv) eomhyeonho ➜ ~/Workspace/spark-practices ↵ main>
→
```

1. 로컬 스파크 구축

2) Python 가상 환경 구성, pyspark 라이브러리 설치

5. 터미널에 spark-shell (scala 기반 REPL),
pyspark (python 기반 REPL) 입력 후
정상 작동 확인.

```
(venv) eomhyeonho ~/Workspace/spark-practices ➜ main
→ spark-shell
23/02/05 16:27:52 WARN Utils: Your hostname, eomhyeonhoui-MacBookPro
23/02/05 16:27:52 WARN Utils: Set SPARK_LOCAL_IP if you need to bind
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
23/02/05 16:28:00 WARN NativeCodeLoader: Unable to load native-hadoop
Spark context Web UI available at http://192.168.0.11:4040
Spark context available as 'sc' (master = local[*], app id = local-1675582144372).
Spark session available as 'spark'.
Welcome to

    ___
   / _ \_  ___  _ _ / / _ \
  \ \ \_ \ \_ \_ \_ / \ \_ \
 /_ / . _ / \_ / / \_ \  version 3.3.1
 /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 11.0.12)
Type in expressions to have them evaluated.
Type :help for more information.
```

scala>

```
(venv) eomhyeonho ~/Workspace/spark-practices ➜ main
→ pyspark
Python 3.9.6 (default, Oct 18 2022, 12:41:40)
[Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
23/02/05 16:29:02 WARN Utils: Your hostname, eomhyeonhoui-MacBookPro.local resolves
23/02/05 16:29:02 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(ne
23/02/05 16:29:03 WARN NativeCodeLoader: Unable to load native-hadoop library for yo
Welcome to

    ___
   / _ \_  ___  _ _ / / _ \
  \ \ \_ \ \_ \_ \_ / \ \_ \
 /_ / . _ / \_ / / \_ \  version 3.3.1
 /_/

Using Python version 3.9.6 (default, Oct 18 2022 12:41:40)
Spark context Web UI available at http://192.168.0.11:4040
Spark context available as 'sc' (master = local[*], app id = local-1675582144372).
SparkSession available as 'spark'.
>>> 
```

>>>

2. 워드 카운트 예제

Link : https://github.com/startFromBottom/fc-spark-practices/tree/main/1_word_count

스파크 애플리케이션의 구성 요소

1. 애플리케이션 구성 요소

- 1) 클러스터 매니저(Cluster Manager)
- 2) 드라이버(Driver)
- 3) 실행기 (Executor)
- 4) 스파크 세션 (Session)
- 5) 잡 (Job)
- 6) 스테이지 (Stage)
- 7) 태스크 (task)

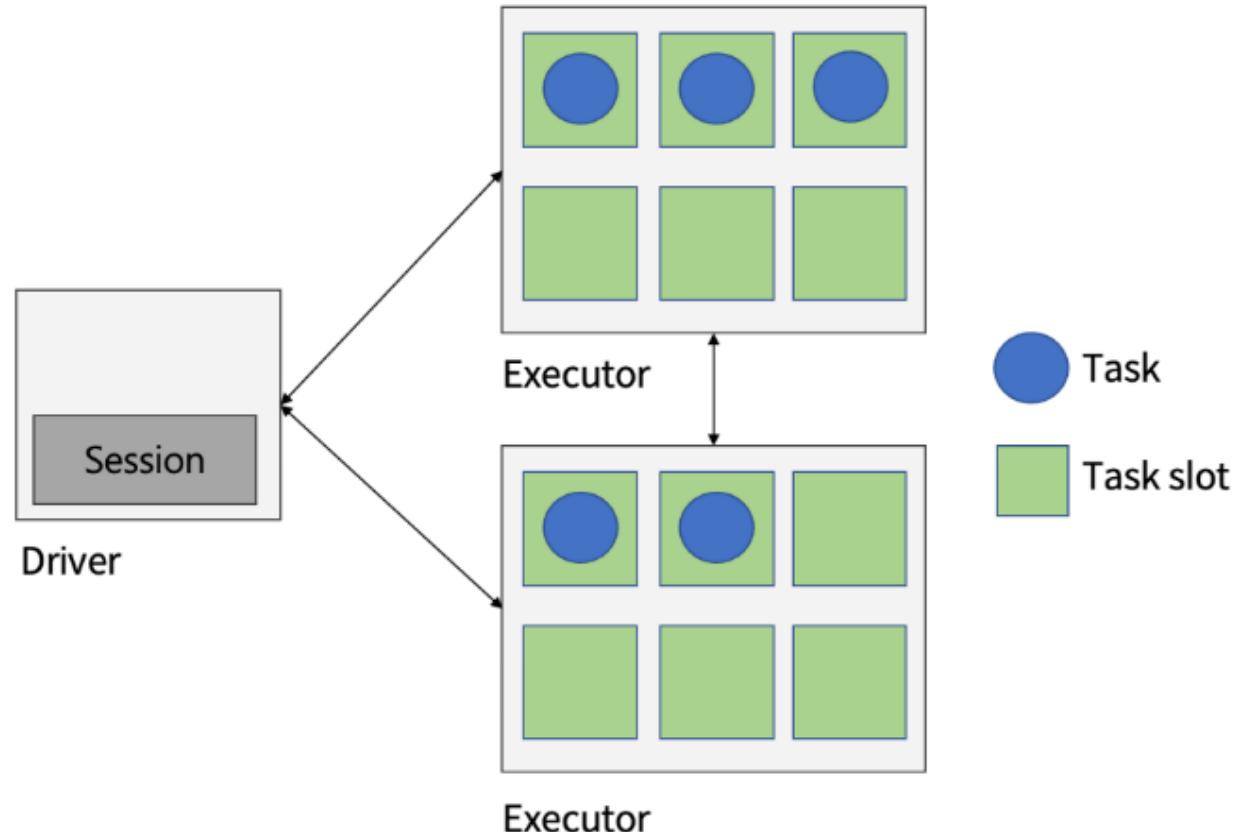


Fig 1 - Driver, Session, Executor, Task, Task slot간의 관계.

1) 클러스터 매니저

주요 역할 : 애플리케이션의 리소스 관리

- ex)
 - 드라이버가 요청한 실행기 프로세스 시작.
 - 실행 중인 프로세스를 중지하거나 재시작.
 - 실행자 프로세스가 사용할 수 있는 최대 CPU 코어 개수 제한 등.
- 종류
 - Standalone
 - Apache Mesos
 - Hadoop Yarn
 - Kubernetes

2) 드라이버(Driver)

주요 역할 : 스파크 애플리케이션의 실행을 관리하고 모니터링.

- ex)
 - 클러스터 매니저에 메모리 및 CPU 리소스를 요청.
 - 애플리케이션 로직을 스테이지와 태스크로 분할.
 - 여러 실행자에 태스크를 전달.
 - 태스크 실행 결과 수집.
- 1개의 스파크 애플리케이션에는 1개의 드라이버만 존재.
- 드라이버 프로세스가 어디에 있는지에 따라, 스파크에는 크게 두 가지 모드가 존재.
 - 클러스터 모드 – 드라이버가 클러스터 내의 특정 노드에 존재.
 - 클라이언트 모드 – 드라이버가 클러스터 외부에 존재.

3) 실행기 (Executor)

주요 역할 : 스파크 드라이버가 요청한 태스크들을 받아서 실행하고 그 결과를 드라이버로 반환.

- JVM 프로세스
- 각 프로세스는 드라이버가 요청한 태스크들을 여러 태스크 슬롯 (스레드)에서 병렬로 실행.

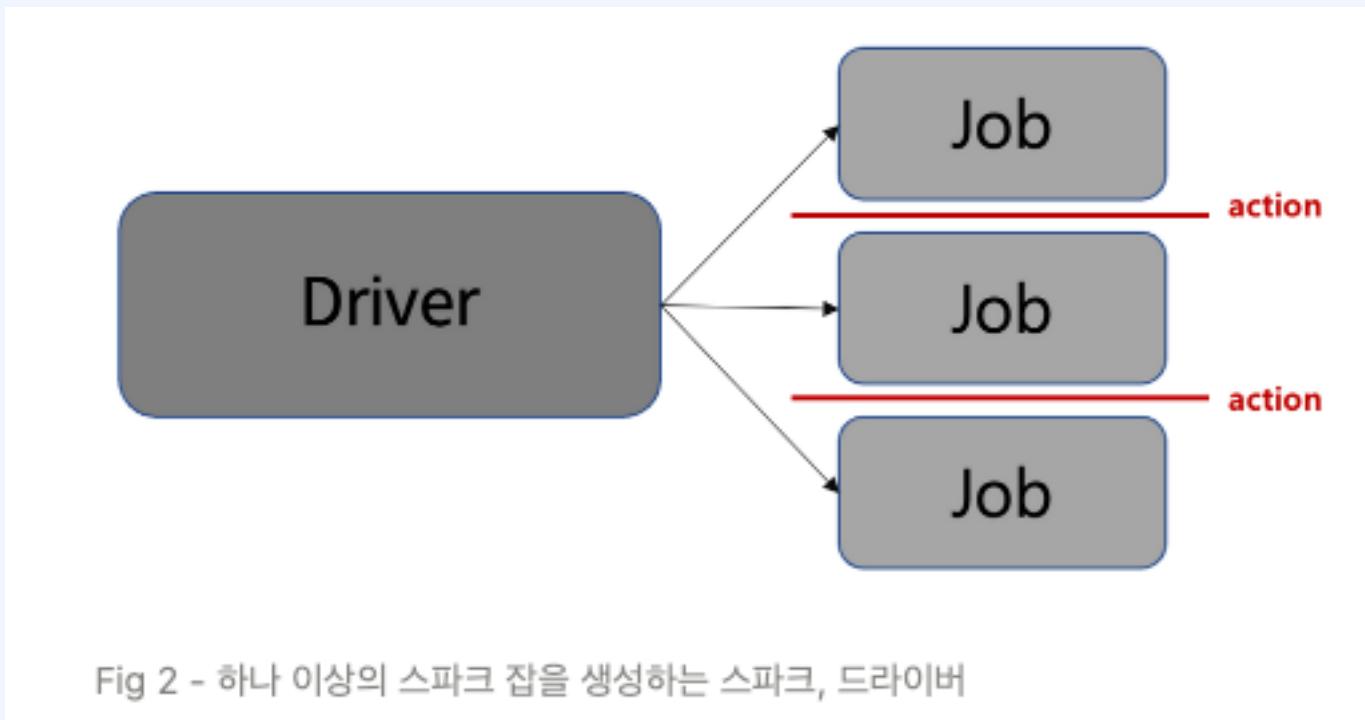
4) 스파크 세션(Spark Session)

주요 역할 : 스파크 코어 기능들과 상호 작용할 수 있는 진입점 제공,
그 API로 프로그래밍을 할 수 있게 해주는 객체.

- spark-shell에서는 기본적으로 제공
- 스파크 애플리케이션에서는 사용자가 SparkSession 객체를 생성해 사용해야 함.

5) 잡 (Job)

- 스파크 액션(ex : save(), collect())에 대한 응답으로 생성되는 여러 태스크로 이루어진 병렬 연산.



6) 스테이지(stage)

- 스파크 각 잡은 스테이지라 불리는 서로 의존성을 가지는 다수의 태스크 모음으로 나뉨

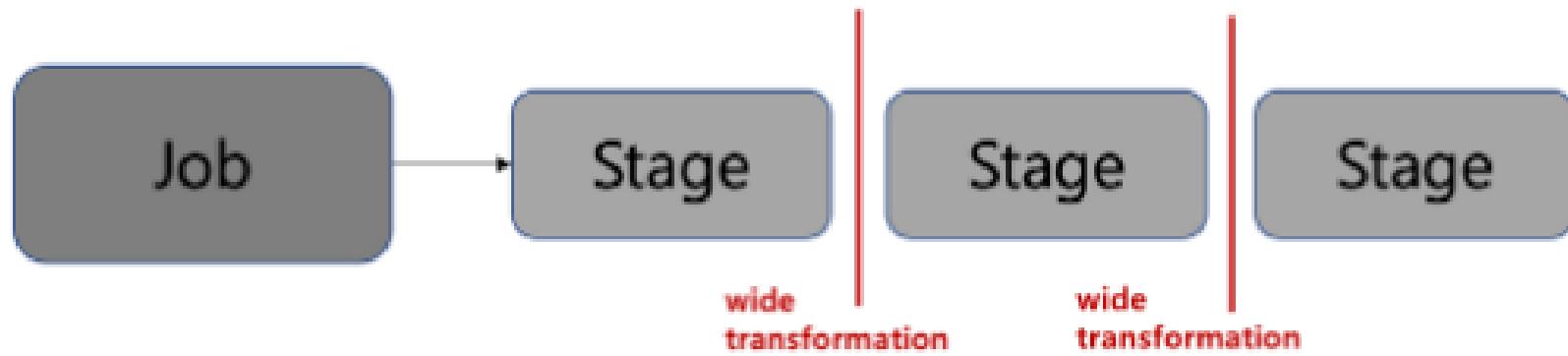


Fig 3. - 하나 이상의 스테이지를 생성하는 스파크 잡.

7) 태스크(task)

- 스파크 각 잡별 실행기로 보내지는 작업 할당의 가장 기본적인 단위
- 개별 task slot에 할당 되고, 데이터의 개별 파티션을 가지고 작업,

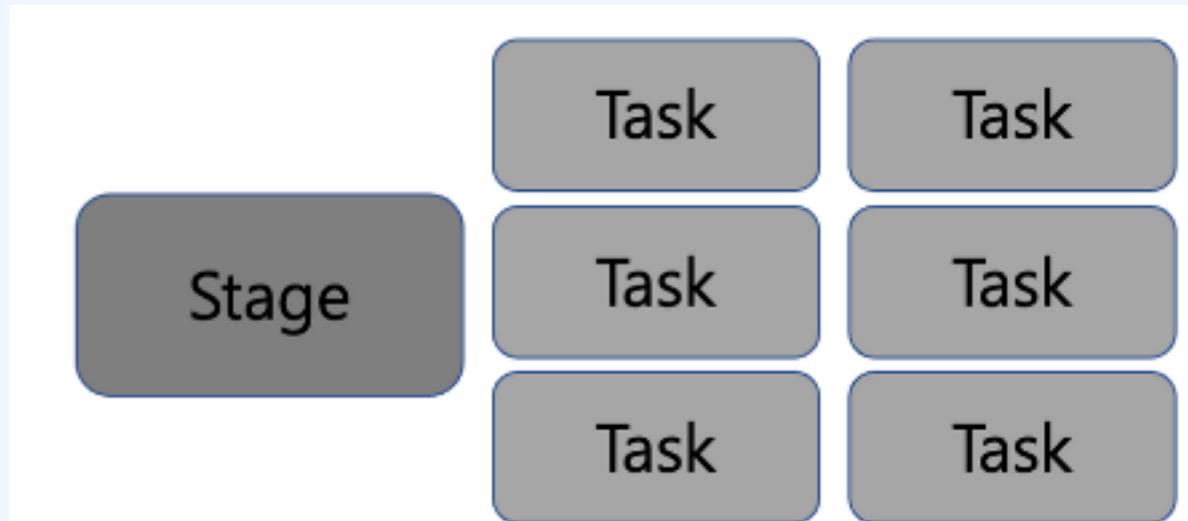


Fig 4. - executor에서 분산 처리 되는 하나 이상의 task를 생성하는 stage.

Transformation, Action, Lazy evaluation 의 개념

1. 스파크 연산의 종류

스파크 연산은 크게

트랜스포메이션(Transformation), 액션(Action)으로 구별됨.

2. Transformation

- immutable(불변)인 원본 데이터를 수정하지 않고,
하나의 RDD나 Dataframe을 새로운 RDD나 Dataframe으로 변형.
 - (input, output) 타입 : (RDD, RDD), (DataFrame, DataFrame)인 연산
 - ex) map(), filter(), flatMap(), select(), groupby(), orderby() 등.
 - RDD ? DataFrame ?
- Narrow, Wide 두 종류가 존재.

2.1. Narrow transformation

- **Narrow transformation**
 - input : 1개의 파티션
 - output : 1개의 파티션
 - 파티션 간의 데이터 교환이 발생하지 않음.
 - ex) filter(), map(), coalesce()

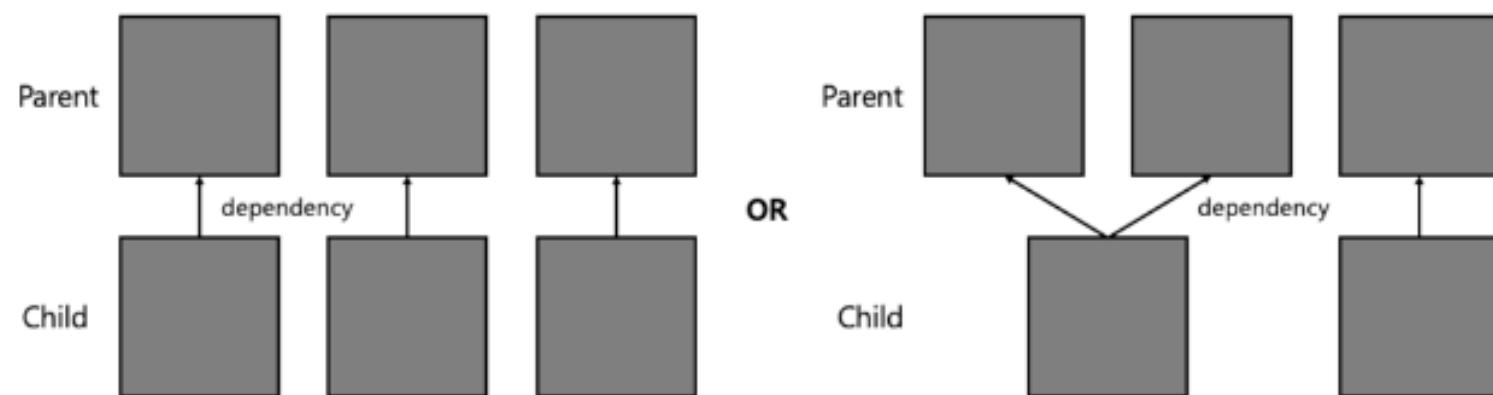


Fig 1. narrow transformation에서 파티션 간 의존 관계

2.2. Wide transformation

- Wide transformation
 - 연산 시 파티션끼리 데이터 교환 발생.
 - ex) groupby(), orderby(), sortByKey(), reduceByKey()
 - 단, join의 경우 두 부모 RDD/Dataframe이 어떻게 파티셔닝 되어 있느냐에 따라 narrow일 수도, wide일 수도 있음.

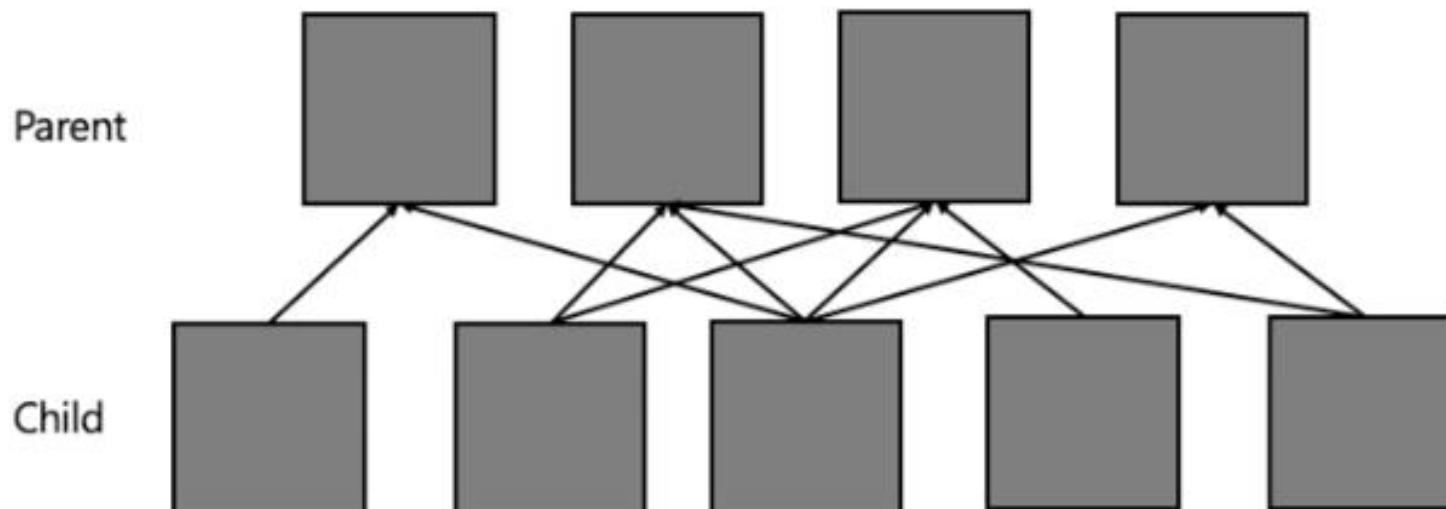


Fig 2. wide transformation에서 파티션 간 의존 관계

3. Action

- immutable(불변)인 인풋에 대해, Side effect (부수 효과)를 포함하고, 아웃풋이 RDD 혹은 Dataframe이 아닌 연산.
 - ex)
 - count() -> 아웃풋 : int
 - collect() -> 아웃풋 : array
 - save() -> 아웃풋 : void

4. Lazy evaluation ?

- 모든 transformation은 즉시 계산되지 않고 계보(lineage)라 불리는 형태로 기록.
- transformation이 실제 계산되는 시점은 action이 실행되는 시점.
- action이 실행될 때, 그 전까지 기록된 모든 transformation들의 자연 연산이 수행됨.

장점)

- 스파크가 연산 쿼리를 분석하고, 어디를 최적화할지 파악하여, 실행 계획 최적화가 가능.
(eager evaluation이라면, 즉시 연산이 수행되기 때문에 최적화의 여지가 없다.)
- 장애에 대한 데이터 내구성을 제공.
 - 장애 발생 시, 스파크는 기록된 lineage를 재실행 하는 것만으로 원래 상태를 재생성 할 수 있음.

RDD

Spark RDD란?

1. RDD란?

Resilient Distributed Dataset
스파크의 기본 추상화 객체.

2. 주요 구성 요소

1. 의존성 정보

- 어떤 입력을 필요로 하고 현재의 RDD가 어떻게 만들어지는지 스파크에게 가르쳐줌
- 새로운 결과를 만들어야 하는 경우, 스파크는 이 의존성 정보를 참고하고 연산을 재반복해 RDD를 재생성할 수 있음.

2. 파티션 (지역성 정보 포함)

- 스파크에서 작업을 나눠 실행기(Executor)들에 분산해 파티션별로 병렬 연산할 수 있는 능력을 제공.

3. 연산 함수 : Partition \Rightarrow Iterator[T]

1. RDD에 저장되는 데이터를 Iterator[T] (반복자) 형태로 변환.

RDD 실습 – 로그 집계 파이프라인 만들기 (map, filter, reduce, group by)

예제 링크)

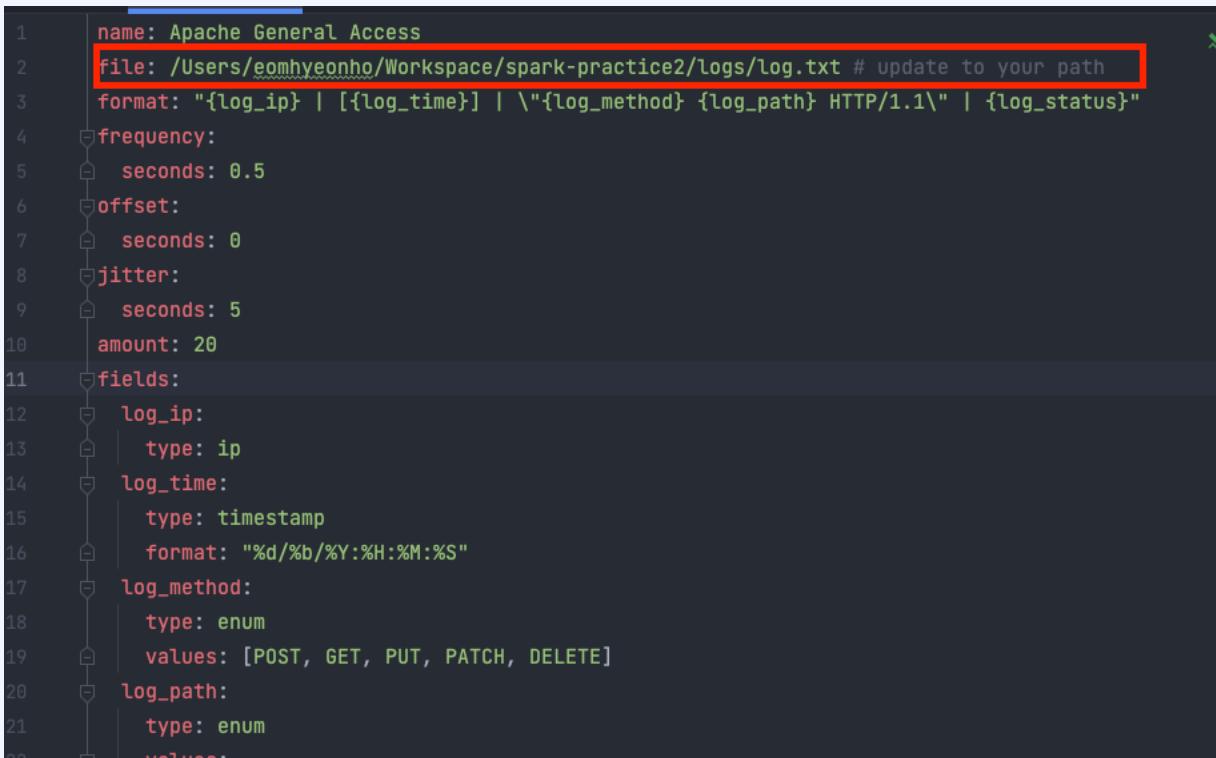
- https://github.com/startFromBottom/fc-spark-practices/blob/main/2_rdd_examples/log_rdd_ex.py

1. 참고 자료)

Random log data 생성하는 방법)

- Pycharm 프로젝트 내의 Terminal에서 아래 명령어 입력
\$ pip install log-generator
\$ log-generator pattern/log.yml
- Yaml file 속성을 수강생의 pc 기준으로 경로를 수정하기.
(경로에 빈 log.txt파일까지 만들어놔야 로그가 정상적으로 생성됨)

참고 링크 : <https://pypi.org/project/log-generator/>



```
name: Apache General Access
file: /Users/eomhyeonho/Workspace/spark-practice2/logs/log.txt # update to your path
format: "{log_ip} | [{log_time}] | \"{log_method} {log_path} HTTP/1.1\" | {log_status}"
frequency:
  seconds: 0.5
offset:
  seconds: 0
jitter:
  seconds: 5
amount: 20
fields:
  log_ip:
    type: ip
  log_time:
    type: timestamp
    format: "%d/%b/%Y:%H:%M:%S"
  log_method:
    type: enum
    values: [POST, GET, PUT, PATCH, DELETE]
  log_path:
    type: enum
    values:
```

RDD 실습 - join

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/2_rdd_examples/join_ex.py

RDD 실습 – 실전 예제 (F사의 강의별 랭킹 데이터 만들기)

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/2_rdd_examples/practical_ex.py

1. 문제 정의)

Q) 인터넷 강의 전문 사이트인 F사에서, 수강생들의 수강 데이터를 바탕으로, 현재 열려 있는 코스들의 스코어를 매기고자 함.

1개의 코스에는 1개 이상의 챕터가 존재하고, 아래와 같이 스코어를 계산.

한 수강생의 코스 스코어)

- 한 수강생이 코스의 90% 이상(ex: 1개 코스에 10개의 챕터가 있을 때, 9개 이상의 챕터를 수강) 이수했을 때는 10점.
- 50% 이상, 90% 미만 이수 시 4점
- 25% 이상, 50% 미만 이수 시 2점
- 25% 미만 이수 시 0점

코스의 전체 스코어)

- 한 수강생 당 계산된 코스 스코어들의 평균 값

2. 입력 데이터 1) 수강생의 챕터별 수강 데이터

user_id	chapter_id	timestamp
14	96	2022.02.24 11:00:00
14	97	2022.02.24 12:23:00
13	96	2022.02.24 13:00:00
13	96	2022.02.25 10:00:00
13	96	2022.02.25 18:00:00
14	99	2022.02.25 20:00:00
13	100	2022.02.25 23:45:22

- 데이터 경로)
2_rdd_examples/data/views-*.csv
(csv 데이터에는 timestamp 컬럼은 넣지 않았습니다.)
- 한 수강생은 한 개 이상의 챕터를 수강 가능.
- 점수 계산 시, 한 수강생이 한 챕터를 여러 번 수강한 경우는 1번으로 고려.
- 왼쪽의 표는 실제 테스트 입력 데이터

2. 입력 데이터 2) 각 코스의 챕터 데이터

course_id	chapter_id
1	96
1	97
1	98
2	99
3	100
3	101
3	102
3	103
3	104
3	105
3	106
3	107
3	108
3	109

- 데이터 경로)
`2_rdd_examples/data/chapters.csv`
- 한 코스에는 한 개 이상의 챕터가 존재 가능.
- 왼쪽의 표는 실제 테스트 입력 데이터

2. 입력 데이터 3) 각 코스의 제목 데이터

course_id	title
1	Flutter Programming
2	Marketing 101
3	Apache Spark guide

- 데이터 경로)
`2_rdd_examples/data/titles.csv`
- 한 코스당 한 개의 제목 존재.
- 왼쪽의 표는 실제 테스트 입력 데이터

3. 출력 데이터 예)

title	Score
Marketing 101	10
Flutter programming	3
Apache Spark guide	0

- Score 기준으로 내림차순 정렬
- 왼쪽의 표는 앞 장의 입력 데이터들 기준으로 계산된 실제 출력 데이터 결과.

SparkSQL, DataFrame, Dataset이란?

SparkSQL, DataFrame, Dataset이란?

1. RDD API의 문제점

1. 스파크가 RDD API 기반의 연산, 표현식을 검사하지 못해 최적화할 방법이 없음.

- RDD API 기반 코드에서 어떤 일이 일어나는지 스파크는 알 수 없음.
- Join, filter, group by 등 여러 연산을 하더라도 스파크에서는 람다 표현식으로만 보임.
- 특히 PySpark의 경우, 연산 함수 Iterator[T] 데이터 타입을 제대로 인식하지 못함.
스파크에서는 단지 파이썬 기본 객체로만 인식.

2. 스파크는 어떠한 데이터 압축 테크닉도 적용하지 못함.

- 스파크는 위에 제네릭 형태로 표현한 타입 T에 대한 정보를 전혀 얻을 수 없음.
- 그 타입의 객체 안에서 어떤 타입의 컬럼에 접근하다고 해도, 스파크는 알 수 없음.
- 결국 바이트 뭉치로 직렬화 해 사용할 수 밖에 없음.

--> 스파크가 연산 순서를 재정렬 해 효과적인 질의 계획으로 바꿀 수 없음

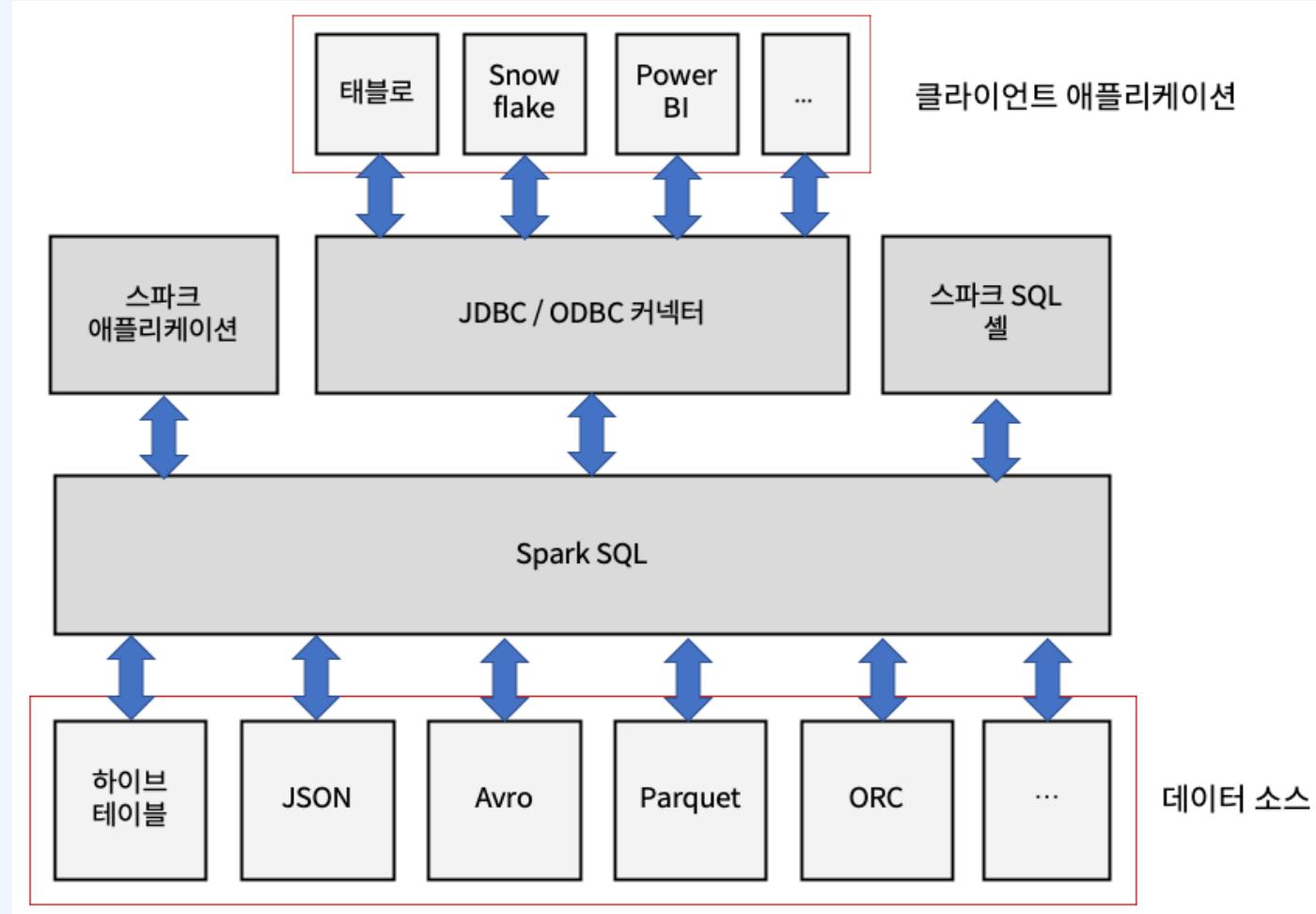
2. SparkSQL이란?

- 구조화된 데이터를 처리하기 위한 스파크 모듈
- DataFrame, Dataset이라 불리는 추상화를 제공하고, 분산 SQL 쿼리 엔진의 역할도 수행.
- 이전 페이지에서 언급한 RDD의 문제점들을 해결할 수 있는지? -> Yes

2.1 SparkSQL의 역할

1. SQL 같은 질의 수행
2. 스파크 컴포넌트들을 통합하고, Dataframe, Dataset가 java, scala, python, R 등 여러 프로그래밍 언어로 정형화 데이터 관련 작업을 단순화할 수 있도록 추상화 해줌
3. 정형화된 파일 포맷(JSON, CSV, txt, avro, parquet, orc 등)에서 스키마와 정형화 데이터를 읽고 쓰며, 데이터를 임시 테이블로 변환
4. 빠른 데이터 탐색을 할 수 있도록 대화형 스파크 SQL 셀을 제공.
5. 표준 데이터베이스 JDBC/ODBC 커넥터를 통해, 외부의 도구들과 연결할 수 있는 중간 역할 제공.
6. 최종 실행을 위해 최적화된 질의 계획과 JVM을 위한 최적화된 코드를 생성

2.1 SparkSQL의 역할



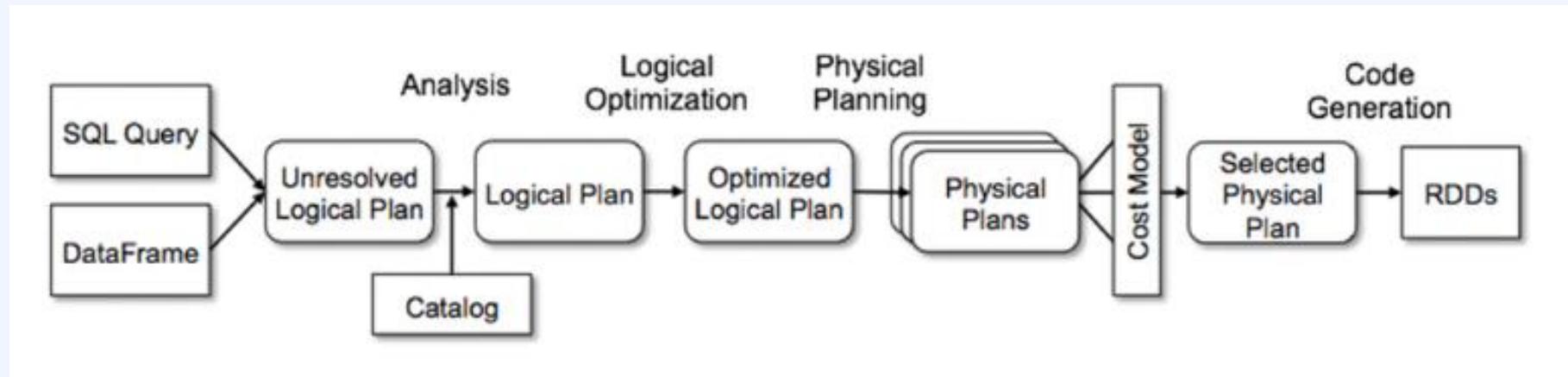
3. SparkSQL의 장점

1. 성능

- RDD와 달리, SparkSQL 사용 시 스파크는 연산, 표현식, 데이터 타입 정보를 모두 알 수 있음.
-> 스파크는 연산 순서를 재정렬해, 더 효과적인 질의 계획으로 변경 가능
- 카탈리스트 옵티마이저(Catalyst Optimizer)

카탈리스트 옵티マイ저(Catalyst Optimizer) ?

- 연산 쿼리를 받아 실행 계획으로 변환. 아래의 그래프에서, 크게 네 단계의 변환 과정을 거쳐 RDD 생성.
 - 분석
 - 논리적 최적화
 - 물리 계획 수립
 - 코드 생성. → 실습 강의에서 확인!



<https://www.databricks.com/glossary/catalyst-optimizer>

3. SparkSQL의 장점

2. 표현성

- Ex) 이름별 나이의 평균 구하기
- RDD 예시의 경우는, 이 랍다 표현식이 어떻게 키를 집계하고 평균 계산을 하는지 직관적으로 알기 어려움.
- 반면 DataFrame API 예시의 경우는 스파크가 무엇을 하는지가 명확하게 보임.

```
# rdd example)
dataRDD = sc.parallelize([('영희', 20), ('철수', 30), ('민수', 22),
                         ('갓나다랑', 10)])
agesRDD = \
    (dataRDD.map(lambda x: (x[0], (x[1], 1)))
     .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])))
     .map(lambda x: (x[0], x[1][0] / x[1][1])))
```

```
# dataframe example)
data_df = ss.createDataFrame([('영희', 20), ('철수', 30), ('민수', 22),
                             ('갓나다랑', 10)], ["name", "age"])

avg_df = data_df.groupby("name").agg(avg("age"))
```

3. SparkSQL의 장점

3. 일관성

- 프로그래밍 언어들을 통틀어 일관성을 가지고 있음.
- Ex) python, scala, java로 작성한 스파크 코드의 형태도 거의 비슷함

```
import org.apache.spark.sql.functions.avg
import org.apache.spark.sql.SparkSession

class Practice {
    def main(args: Array[String]): Unit = {

        val ss = SparkSession
            .builder()
            .appName( name = "AuthorsAges")
            .getOrCreate()

        val dataDF = ss.createDataFrame(Seq(("영희", 20), ("철수", 30), ("민수", 22),
            ("가나다라", 10))).toDF( colNames = "name", "age")

        val avgDF = dataDF.groupBy( col1 = "name").agg(avg( columnName = "age"))

        avgDF.show()
    }
}
```

scala

```
from pyspark import SparkContext, RDD
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg

if __name__ == "__main__":
    ss: SparkSession = SparkSession.builder \
        .master("local") \
        .appName("AuthorsAges") \
        .getOrCreate()

    data_df = ss.createDataFrame([(("영희", 20), ("철수", 30), ("민수", 22),
        ("가나다라", 10)], ["name", "age"]))

    avg_df = data_df.groupby("name").agg(avg("age"))
    avg_df.show()
```

python

3. SparkSQL의 장점

3. 일관성

- 프로그래밍 언어들을 통틀어 일관성을 가지고 있음.
- Ex) python, scala, java로 작성한 스파크 코드의 형태도 거의 비슷함

```
import org.apache.spark.sql.functions.avg
import org.apache.spark.sql.SparkSession

class Practice {
    def main(args: Array[String]): Unit = {

        val ss = SparkSession
            .builder()
            .appName( name = "AuthorsAges")
            .getOrCreate()

        val dataDF = ss.createDataFrame(Seq(("영희", 20), ("철수", 30), ("민수", 22),
            ("가나다라", 10))).toDF( colNames = "name", "age")

        val avgDF = dataDF.groupBy( col1 = "name").agg(avg( columnName = "age"))

        avgDF.show()
    }
}
```

scala

```
from pyspark import SparkContext, RDD
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg

if __name__ == "__main__":
    ss: SparkSession = SparkSession.builder \
        .master("local") \
        .appName("AuthorsAges") \
        .getOrCreate()

    data_df = ss.createDataFrame([(("영희", 20), ("철수", 30), ("민수", 22),
        ("가나다라", 10)], ["name", "age"]))

    avg_df = data_df.groupby("name").agg(avg("age"))
    avg_df.show()
```

python

4. Dataframe API – 개요

- 구조, 포맷 등 몇몇 특정 연산 등에 있어, 판다스 데이터 프레임에 영향을 많이 받음.
 - Pandas doc : <https://pandas.pydata.org/>
- 이름 있는 컬럼과 스키마를 가진 분산 인메모리 테이블처럼 동작.
- 사람이 보는 형태로는 스파크 Dataframe은 아래처럼 하나의 표의 형태로 보임.

Id (int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List<String>)
1	Stitch	Um	https://aaa.1	1/4/2022	9999	[twitter, Linkedin]
2	Denny	Lee	https://bb.1	2/3/2023	4200	[Instagram, web]
3	Das	Kim	https://cdef.1	2/5/2023	1356	[LinkedIn, Instagram]

4.1 Dataframe API – 데이터 타입

- 기본 타입
 - Byte, Short, Integer, Long, Float, Double, String, Boolean, Decimal
- 정형화 타입
 - Binary, Timestamp, Date, Array, Map, Struct, StructField
- Spark doc : <https://spark.apache.org/docs/latest/sql-ref-datatypes.html>
- 실제 데이터를 위한 스키마를 정의할 때 어떻게 이런 타입들이 연계되는지를 아는 것이 중요

4.2 Dataframe API – 스키마(Schema)

- 스파크에서, 스키마는 Dataframe을 위해 컬럼 이름과 연관된 데이터 타입을 정의한 것.
- 외부 데이터 소스에서 구조화된 데이터를 읽어 들일 때 사용.
- 읽을 때 스키마를 가져오는 방식과 달리, 미리 스키마를 정의하는 것은 여러 장점이 존재.
 1. 스파크가 데이터 타입을 추측해야 되는 책임을 덜어줌.
 2. 스파크가 스키마를 확정하기 위해, 파일의 많은 부분을 읽어 들이려고 별도의 잡을 만드는 것을 방지.
 3. 데이터가 스키마와 맞지 않는 경우, 조기에 문제 발견 가능.

4.2 Dataframe API – 스키마(Schema)

- 스키마 정의 방법
 1. 프로그래밍 스타일
 2. DDL (Data definition language)

```
# programming
schema_programming = StructType([StructField("author", StringType(), False),
                                 StructField("title", StringType(), False),
                                 StructField("pages", StringType(), False), ])

# DDL
schema_ddl = "author STRING, title STRING, pages INT"
```

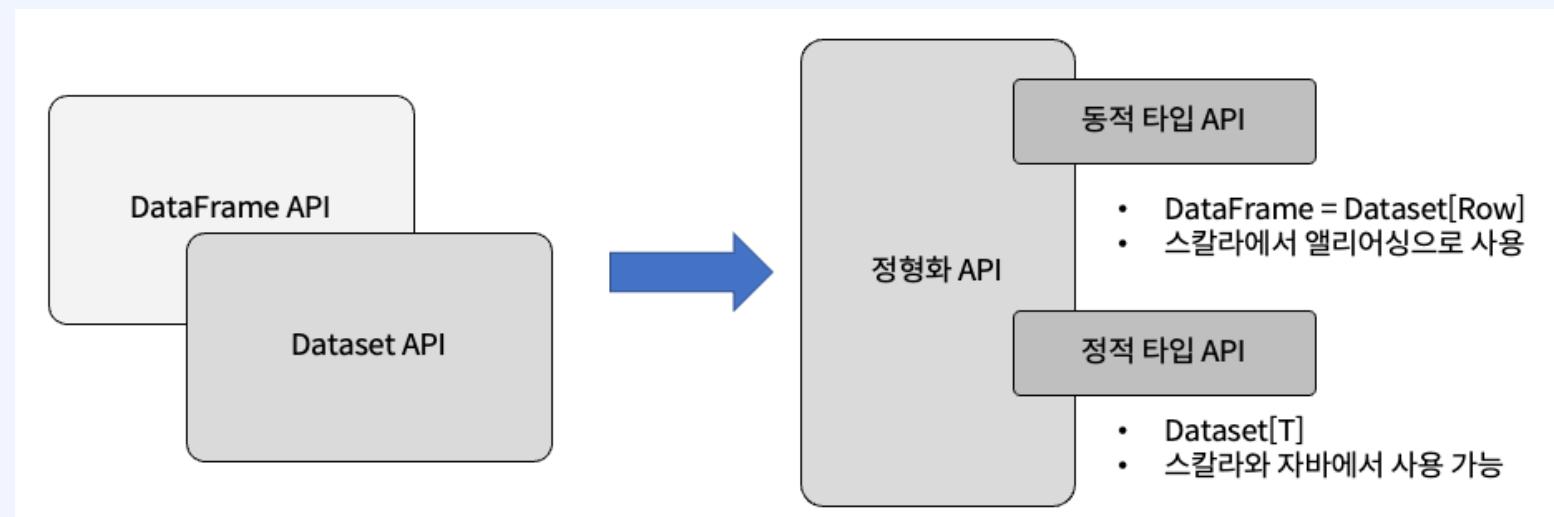
4.3 Dataframe API – 기타

- 컬럼과 표현식
- 행 (Row)
- Transformation, action 연산들
- DataframeReader, DataFrameWriter
- Etc..

→ 실습 강의에서 확인!

5. Dataset API - 개요

- 스파크 2.0에서, 개발자들이 한 종류의 API만 알면 되도록, DataFrame, Dataset API를 하나로 합침.
- Dataset은 정적 타입(typed) API와 동적 타입(untyped) API의 두 특성을 모두 가짐.
- Java, Scala(타입 안전을 보장하는 언어)에서만 사용이 가능하고, Python, R(타입 안전을 보장하지 않는 언어)에서는 사용이 불가능, DataFrame API만 사용 가능.



5.1 Dataset API – 특징

- Scala에서는 case class를, Java에서는 JavaBean 클래스를 사용해 Dataset이 쓸 스키마를 지정할 수 있음.
- DataFrame API에서 지원하는 다양한 트랜스포메이션, 액션 연산들은 Dataset에서도 비슷하게 사용이 가능.
- Dataset이 사용되는 동안에는, 하부의 스파크 SQL 엔진이 JVM 객체의 생성, 변환, 직렬화, 역직렬화를 담당.
- Dataset Encoder의 도움을 받아 Java Off-heap 메모리 관리

6. DataFrame vs Dataset

- 가장 큰 차이점은 오류가 발견되는 시점

	SQL	DataFrame	Dataset
문법 오류	런타임 시점	컴파일 시점	컴파일 시점
분석 오류	런타임 시점	런타임 시점	컴파일 시점

SparkSQL, DataFrame, Dataset 실습

- 로그 집계 파이프라인 만들기 (DataFrame API)

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/3_sparksqlexamples/log_dataframe_ex.py

SparkSQL, DataFrame, Dataset 실습

- 로그 집계 파이프라인 만들기 (SQL API)

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/3_sparksql_examples/log_sql_ex.py

SparkSQL, DataFrame, Dataset 실습

- Join

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/3_sparksqlexamples/join_ex.py

SparkSQL, DataFrame, Dataset 실습

- 실전 예제 1 (샌프란시스코 소방서 공공 데이터)

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/3_sparksqlexamples/sf_fire_calls_ex.py

데이터 링크

- <https://github.com/databricks/LearningSparkV2/blob/master/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv>

SparkSQL, DataFrame, Dataset 실습

- udf (사용자 정의 함수)

예제 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/3_sparksq...

RDD, SparkSQL, DataFrame 비교

1. RDD, Dataframe, 언제 사용해야 하는가?

1. RDD를 사용해야 하는 경우

1. RDD에 의존적인 서드파티 패키지 / 라이브러리를 사용.
2. Dataframe, Dataset(for Java, Scala only)를 사용함으로써 얻을 수 있는
코드 최적화, 효과적인 공간 사용, 성능 상의 이득을 포기할 수 있는 경우.

- 코드 최적화, 성능 상의 이득)

RDD 코드는 Spark engine에서 별도의 최적화를 하나도 적용하지 않고,

작성한 코드 그대로 실행. 개발자가 spark engine보다 잘 짤 수도 있지만, 대부분은 아님.

- 효과적인 공간 사용)

RDD 사용 시 기본적으로 Serialization(직렬화) 시 Java 내장 Serializer를 사용하는데 매우 비효
율적이고, 공간이 낭비된다.

3. 스파크가 어떻게 질의를 수행할지 정확하게 지정해주고 싶은 경우.
4. legacy 코드가 RDD로 구성되어있고, 큰 문제 없이 잘 돌아가고 있을 때.

2. DataFrame을 사용하는 경우

- 앞 장의 언급된 특별한 이유 외에는 거의 무조건 DataFrame 사용이 추천된다.
- 특히 Pyspark의 경우, RDD와 DataFrame의 성능 차이가 크다고 함.
- 성능 문제 외에도, Dataframe code는 스파크가 무엇을 하는지 가 코드 상으로 더 명확함!

1. Spark cluster ?

- spark application을 구동하기 위한 분산 클러스터.
- spark cluster 설정 예시
 - 10개의 worker node.
 - 각 worker node당 capacity.
 - 8 CPU cores.
 - 32 GB RAM
 - cluster의 capacity.
 - 80 CPU cores.
 - 320 GB RAM.

스파크 심층 분석

목차

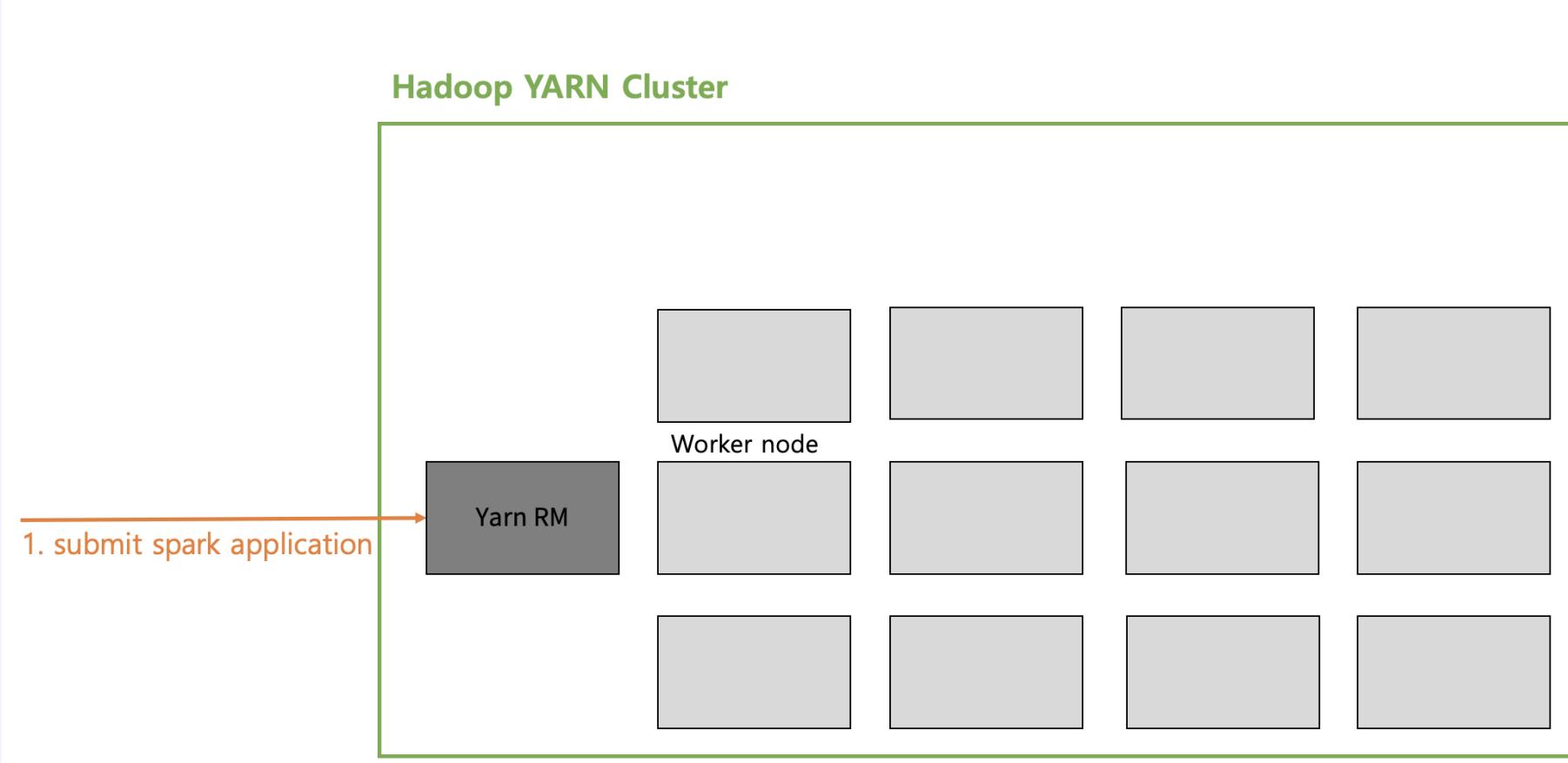
1. 스파크 클러스터, 런타임 아키텍처에 대한 이해.
2. spark-submit 주요 파라미터 확인.
3. Deploy mode - cluster, client mode.
4. 스파크(action, stage, shuffle, task, slot 개념) 실습.
5. Join의 종류.
6. 스파크에서의 메모리 할당.
7. 스파크 메모리 관리.
8. Repartition / Coalesce 대한 이해.
9. Caching, Persistence에 대한 이해.
10. Shared Variable (Accumulator, Broadcast variable)에 대한 이해.
11. 스파크 Query Plan.
12. Dynamic Resource Allocation.
13. Spark Scheduler에 대한 이해.
14. (Spark 3.0 +) AQE - Adaptive Query Execution에 대한 이해.
15. (Spark 3.0 +) DPP - Dynamic Partition Pruning에 대한 이해.

1. 스파크 클러스터, 런타임 아키텍쳐에 대한 이해

1. Spark cluster ?

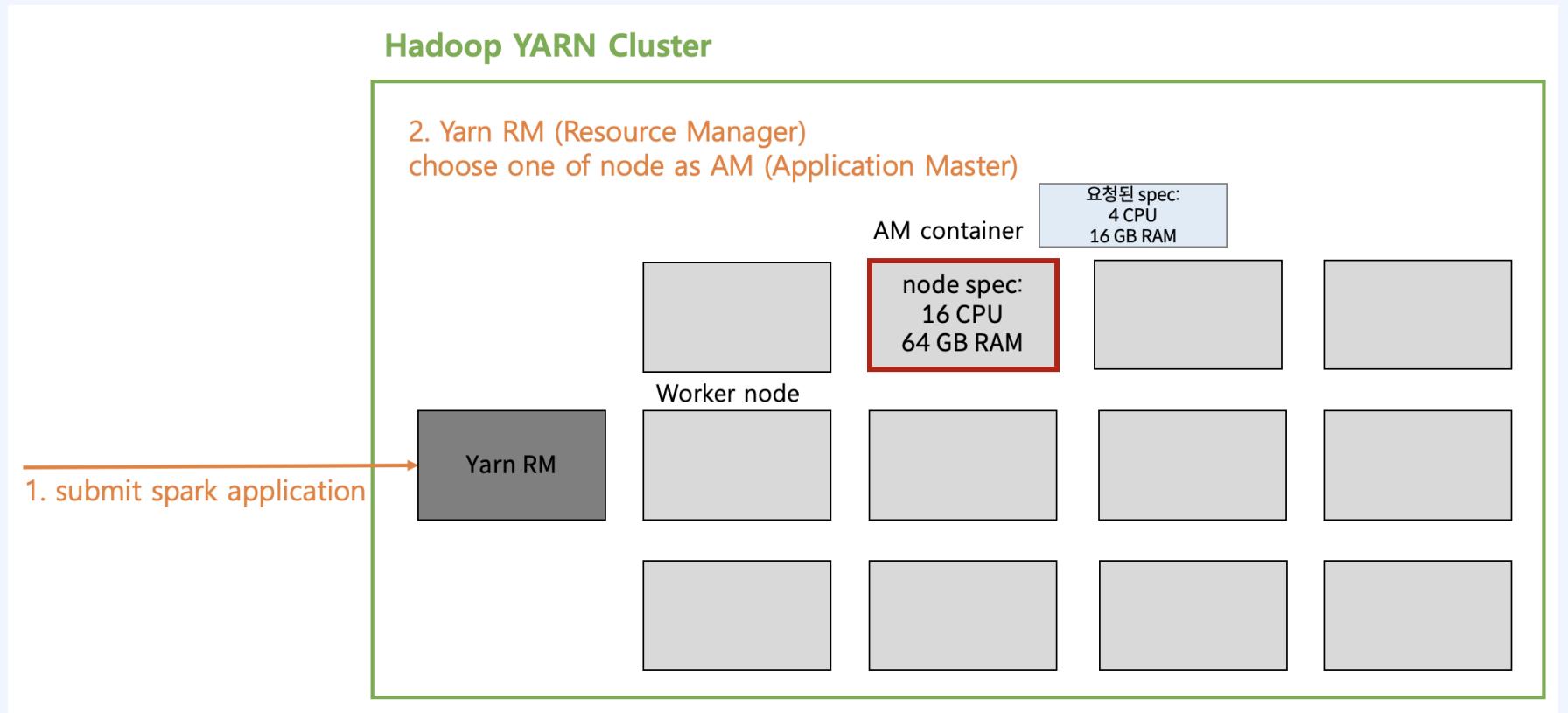
- spark application을 구동하기 위한 분산 클러스터.
- spark cluster 설정 예시
 - 10개의 worker node.
 - 각 worker node당 capacity.
 - 8 CPU cores.
 - 32 GB RAM
 - cluster의 capacity.
 - 80 CPU cores.
 - 320 GB RAM.

2. 런타임 아키텍처



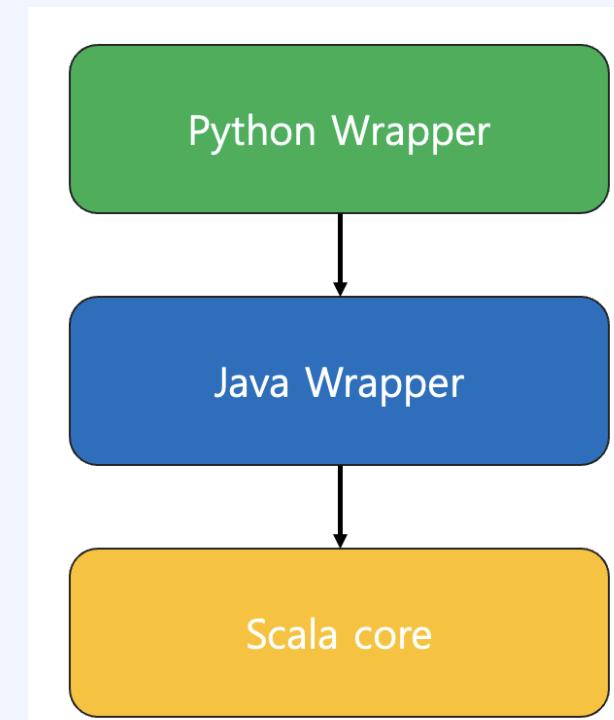
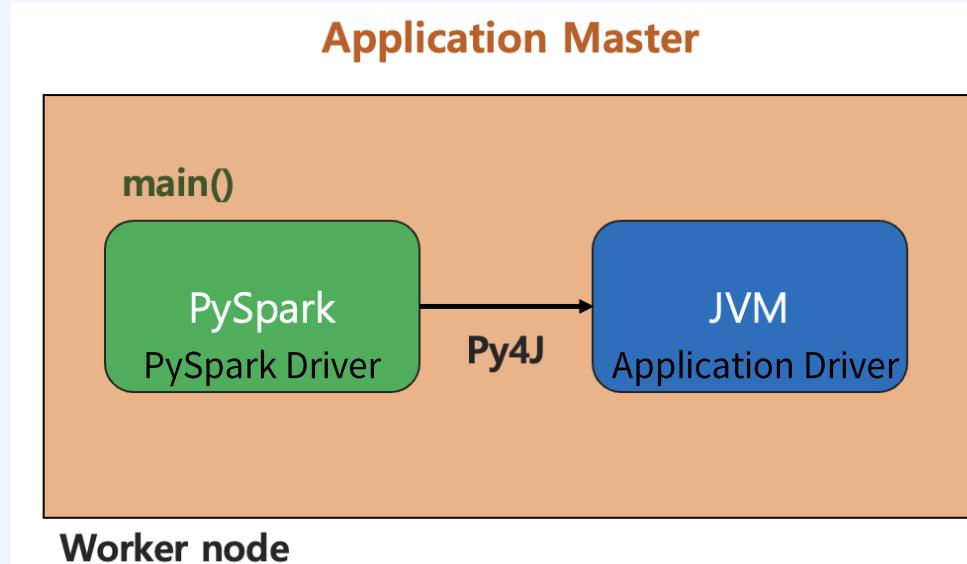
2. 런타임 아키텍처

- Worker node의 spec \geq Yarn RM이 할당한 spec



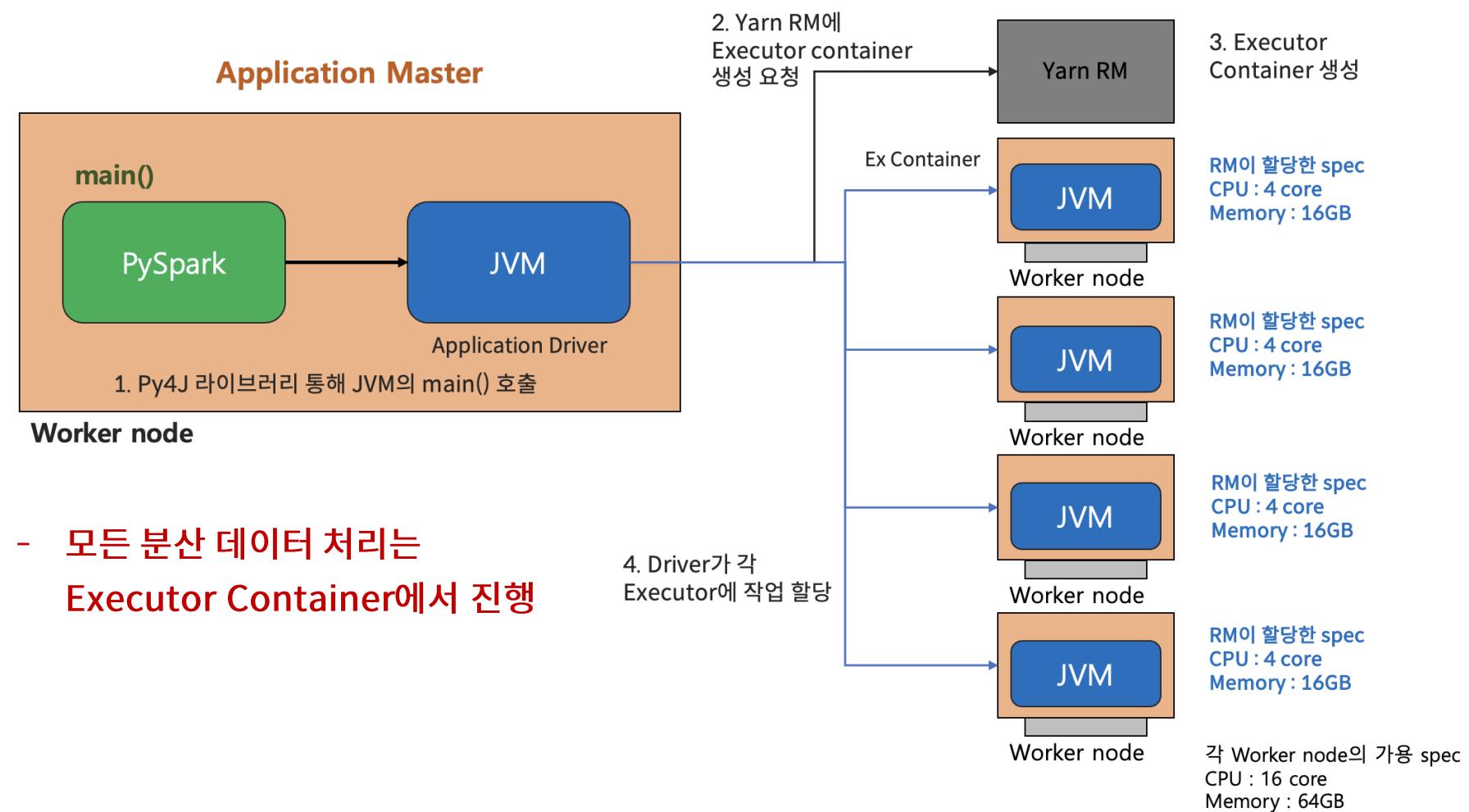
2.1 런타임 아키텍처 - Application Master(AM) container

- AM (Application Master) Container 내에 main() 존재.
- PySpark 사용 시, Python wrapper는 Py4J라는 라이브러리를 사용해, JVM 내의 Java wrapper를 호출.(Py4J : Python application이 Java application 호출할 수 있게 함.)
- 그 후 JVM 내의 Scala application이 실행됨.
- Application Driver는 실제 데이터를 분산 처리하지 않음.
(실제 데이터는 Executor Node에서 처리됨.)

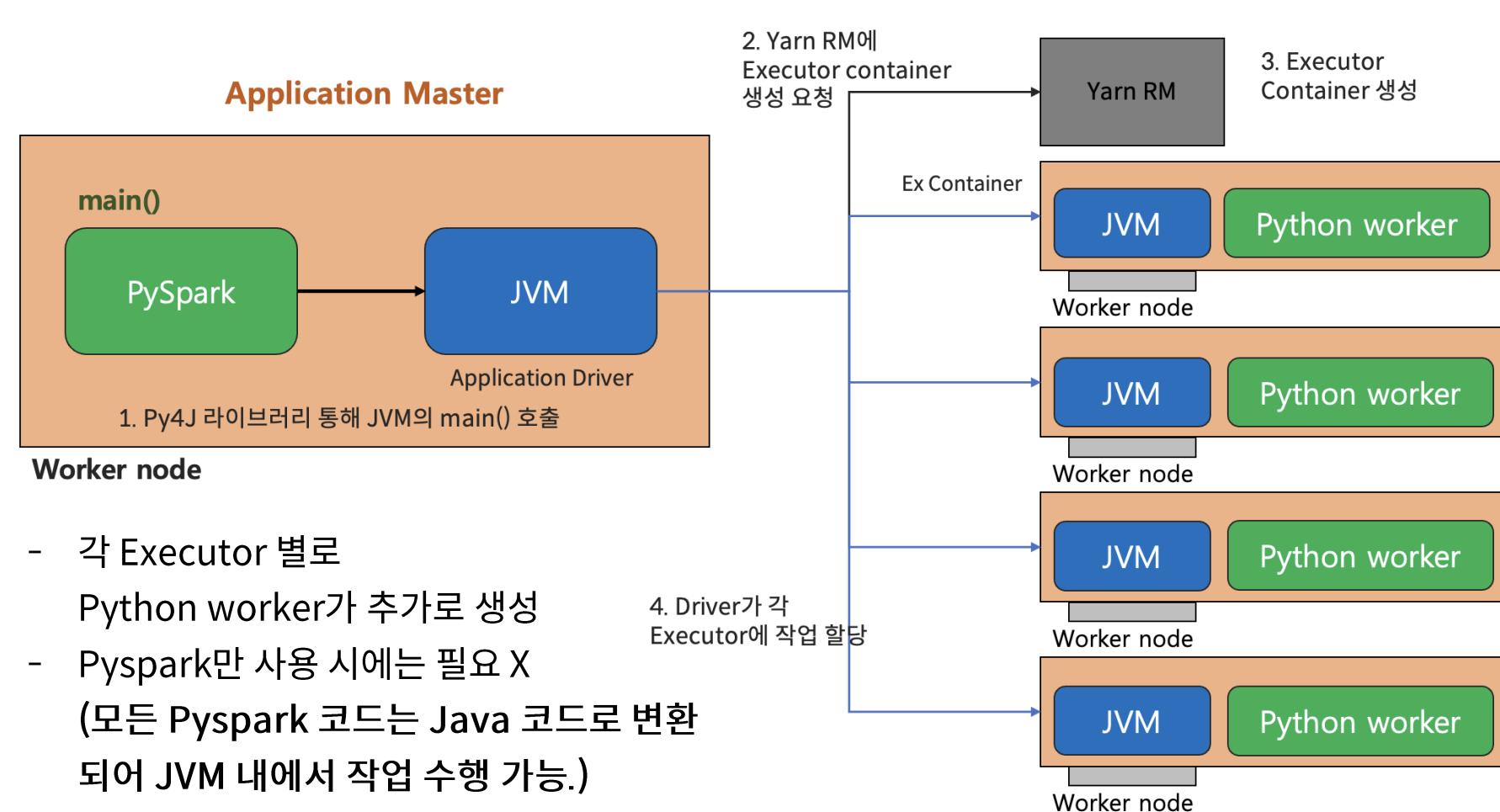


(스파크의 코드 구조)

2.2 런타임 아키텍처 - AM Container + Yarn RM + Executor Container



2.3 런타임 아키텍처 - Pyspark에 포함되지 않은 Python 라이브러리 및 UDF 사용 시



Spark-submit 주요 파라미터 확인

1. spark-submit 커맨드

- spark cluster에 Spark application을 배포하기 위한 명령어.
- local 환경에서는 단순히 파이썬 파일을 실행시켜도 되지만,
원격 cluster에 application을 배포하기 위해서는 spark-submit 명령어를
사용해야 함.

2. spark-submit 명령어

명령어 구조

spark-submit

--class <main-class> : Pyspark에서는 사용 불가, Java, Scala에서만 가능.

--master <master-url> : ex) yarn, local[3] (3: thread count)

--deploy-mode <deploy-mode> : client 또는 cluster

<application-jar or python script>

--conf ... (additional spark options) : ex) spark.executor.memoryOverhead = 0.20

--driver-cores 2

--driver-memory 8G

--num-executors 4

--executor-cores 4

--executor-memory 16G

..

- 다른 설정들 : <https://spark.apache.org/docs/latest/configuration.html>

3. spark-submit 예시)

Pyspark)

```
spark-submit --master yarn --deploy-mode cluster --driver-cores 2 --driver-memory 8G  
--num-executors 4 --executor-cores 4 --executor-memory 16G hello-spark.py
```

Scala or Java)

```
spark-submit --class fastcampus.learning-spark.HelloSpark  
--master yarn --deploy-mode cluster --driver-cores 2  
--driver-memory 8G --num-executors 4 --executor-cores 4 --executor-memory 16G hello-spark.jar
```

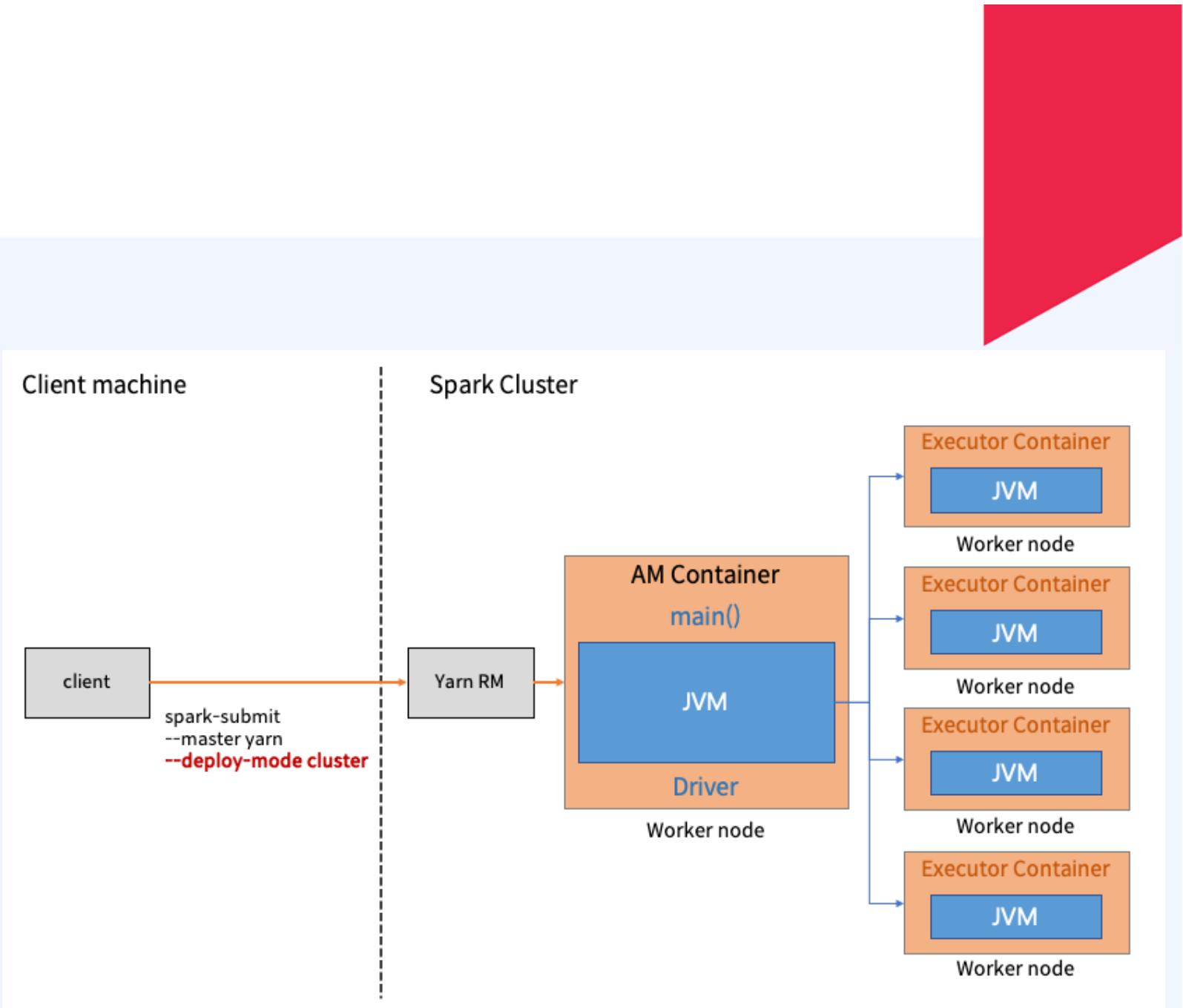
Deploy mode – cluster, client mode

1. Deploy mode

- 크게 cluster, client 두 가지 모드가 존재.
- Cluster mode
 - 명령어 : spark-submit --master yarn --deploy-mode cluster
- Client mode
 - 명령어 : spark-submit --master yarn --deploy-mode client

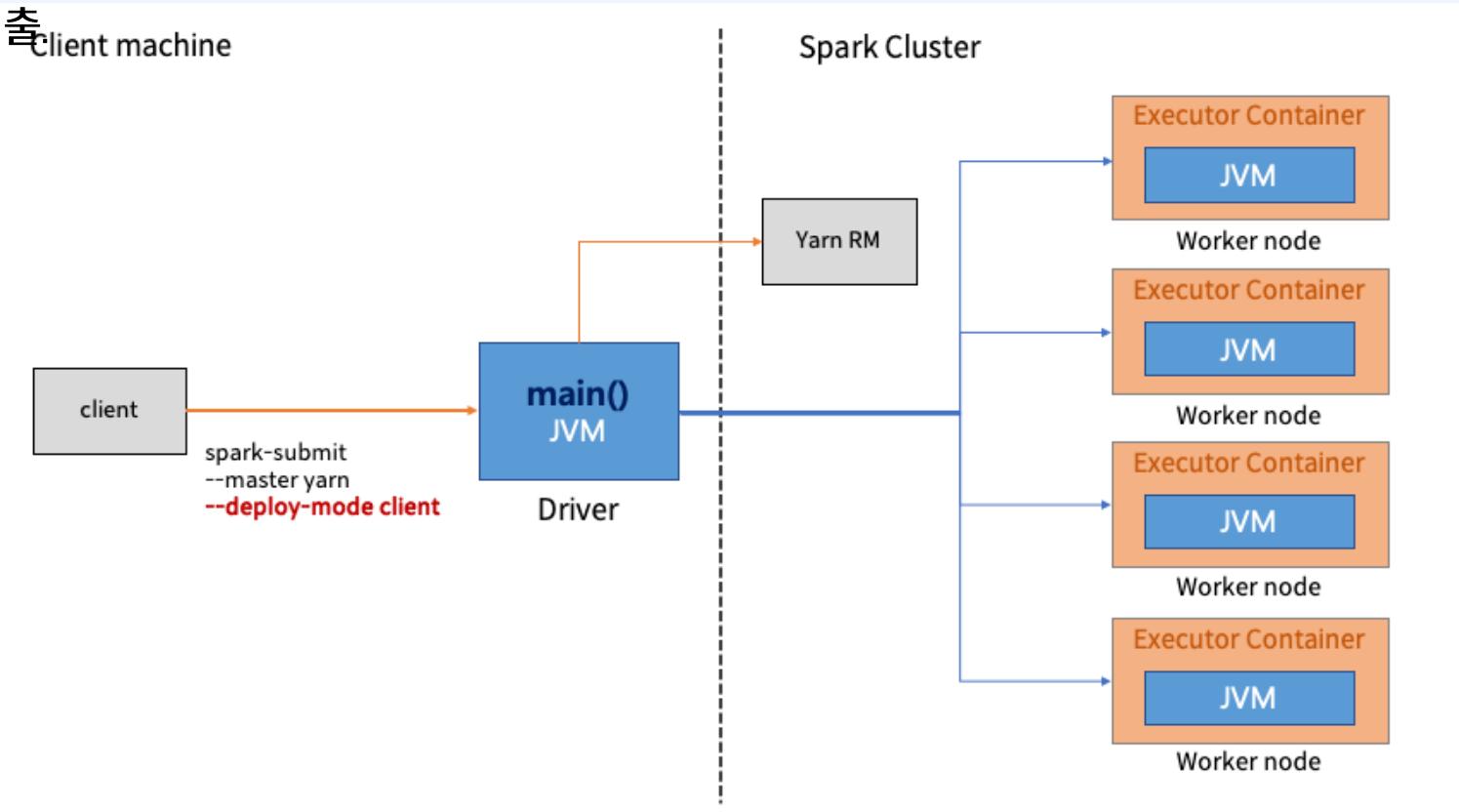
2. Cluster mode

1. client가 Yarn RM에 잡을 제출.
 2. Yarn RM은 AM 컨테이너에서 Driver를 구동하도록 함.
 3. Driver는 Yarn RM에 executor container를 위한 자원 요청.
 4. Yarn RM은 executor container들을 실행하고, Driver에 executor 관리를 위임.
- AM, Executor container 모두 Spark cluster의 worker node에서 실행.



3. Client mode

1. client는 잡을 Yarn RM이 아닌,
client 머신에 있는 driver JVM에 제출
2. Driver는 Yarn RM에 executor
container를 위한 자원 요청.
3. Yarn RM은 executor container
들을 실행하고, Driver에
executor 관리를 위임.

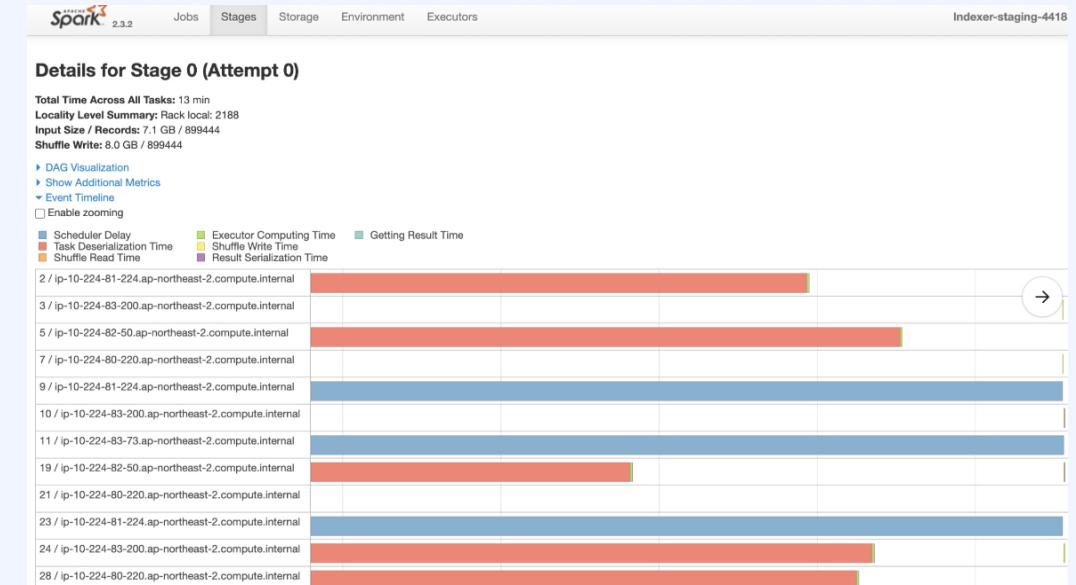
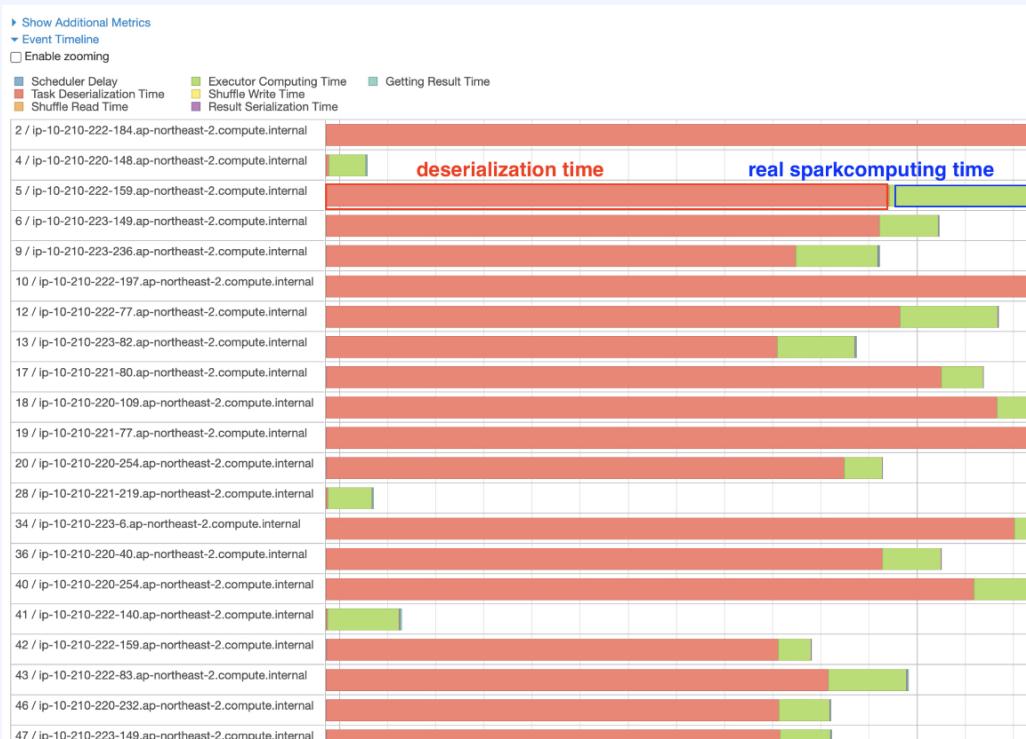


4. Cluster mode, Client mode 비교

- 가장 큰 차이점은 **Driver application의 위치**.
 - Cluster mode : Driver가 Spark cluster의 worker node에 위치.
 - Client mode : Driver가 Spark cluster가 아닌 client 머신에 위치.
- 두 모드에서 Executor는 모두 Spark cluster에 위치.
- **production 환경에서는 거의 무조건 cluster mode를 사용!**
 - Why?
 1. Driver, Executor, Yarn RM 모두 Spark cluster에 위치하기 때문에, client 머신과의 의존 관계 없음.
 - ex) client 머신에 장애가 발생해도, spark cluster와는 무관.
 2. 성능.
 - client 대비, driver, executor가 network 상에서 가까이 있을 가능성 높음. -> network latency 감소.
- Client mode를 사용하는 경우.
 - interactive workload.
 - driver가 local에 있기 때문에, 커뮤니케이션이 쉬움.
 - ex) spark-shell, notebooks.
 - spark 잡의 로그를 쉽게 보고자 할 때.

5. 실무 사례 소개 (Production 환경에서 client mode를 사용했던 경우)

- production 환경에서 client mode 모드 시, Task Deserialization time 및 Scheduler Delay 이 비정상적으로 커지는 경우 존재.



5. 실무 사례 소개 (Production 환경에서 client mode를 사용했던 경우)

- client mode에서 cluster mode로 변환 시 stage 별 lead time 변화.
(데이터를 불러오는 맨 앞 stage들에서, Task deserialization time 감소 확인.)

Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write		Submitted	Duration	Tasks:
2023/01/06 19:55:52	6 s	2297/2297	43.7 GB				+details	2023/01/06 20:40:49	4 s	
2023/01/06 19:55:42	4 s	2297/2297	43.7 GB				+details	2023/01/06 20:40:33	7 s	
2023/01/06 19:55:37	0.3 s	2055/2055		15.9 KB	2.1 KB		+details	2023/01/06 20:40:28	0.3 s	
2023/01/06 19:55:36	1 s	912/912				2.1 KB	+details	2023/01/06 20:40:27	1 s	
2023/01/06 19:55:24	10 s	2297/2297	43.7 GB	5.8 GB			+details	2023/01/06 20:40:14	10 s	
2023/01/06 19:55:05	4 s	2055/2055		8.2 GB	8.5 GB		+details	2023/01/06 20:39:48	4 s	
2023/01/06 19:54:19	44 s	2297/2297			35.5 GB	8.5 GB	+details	2023/01/06 20:38:51	54 s	
2023/01/06 19:51:06	3.2 min	2297/2297	77.0 GB			35.5 GB	+details	2023/01/06 20:34:46	4.0 min	
2023/01/06 19:50:44	8 s	AS-IS(client mode) 1/1					+details	2023/01/06 20:34:25	4 s	
2023/01/06 19:49:06	1.6 min	137/137					+details	2023/01/06 20:33:47	36 s	
2023/01/06 19:47:08	1.9 min	97/97					+details	2023/01/06 20:32:47	57 s	
2023/01/06 19:46:50	18 s	7/7					+details	2023/01/06 20:32:34	12 s	
2023/01/06 19:46:34	16 s	12/12					+details	2023/01/06 20:32:28	6 s	

스파크 - action, stage, shuffle, task, slot 확인 실습

1. 실습 코드 링크)

https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/spark_jobs_ex.py

2. code snippet

Job 1

```
df = ss.read.option("header", "true") \
    .option("inferSchema", "true") \
    .csv("data/titanic_data.csv")
```

Job 2

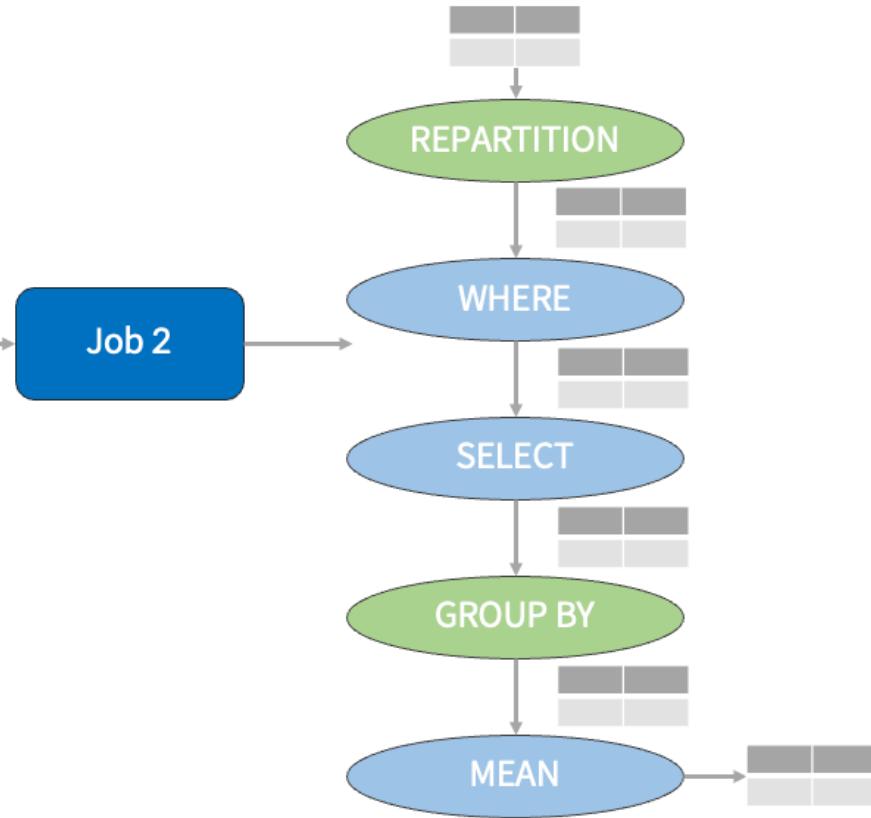
```
df = df.repartition(5) \
    .where("Sex = 'male'") \
    .select("Survived", "Pclass", "Fare") \
    .groupby("Survived", "Pclass") \
    .mean()

print(f"result ===> {df.collect()}")
```

1. Action : Job을 나누는 기준
 - read(), collect()
2. Wide transformation : Stage를 나누는 기준
 - repartition()
 - groupby()
3. Narrow transformation
 - where()
 - select()

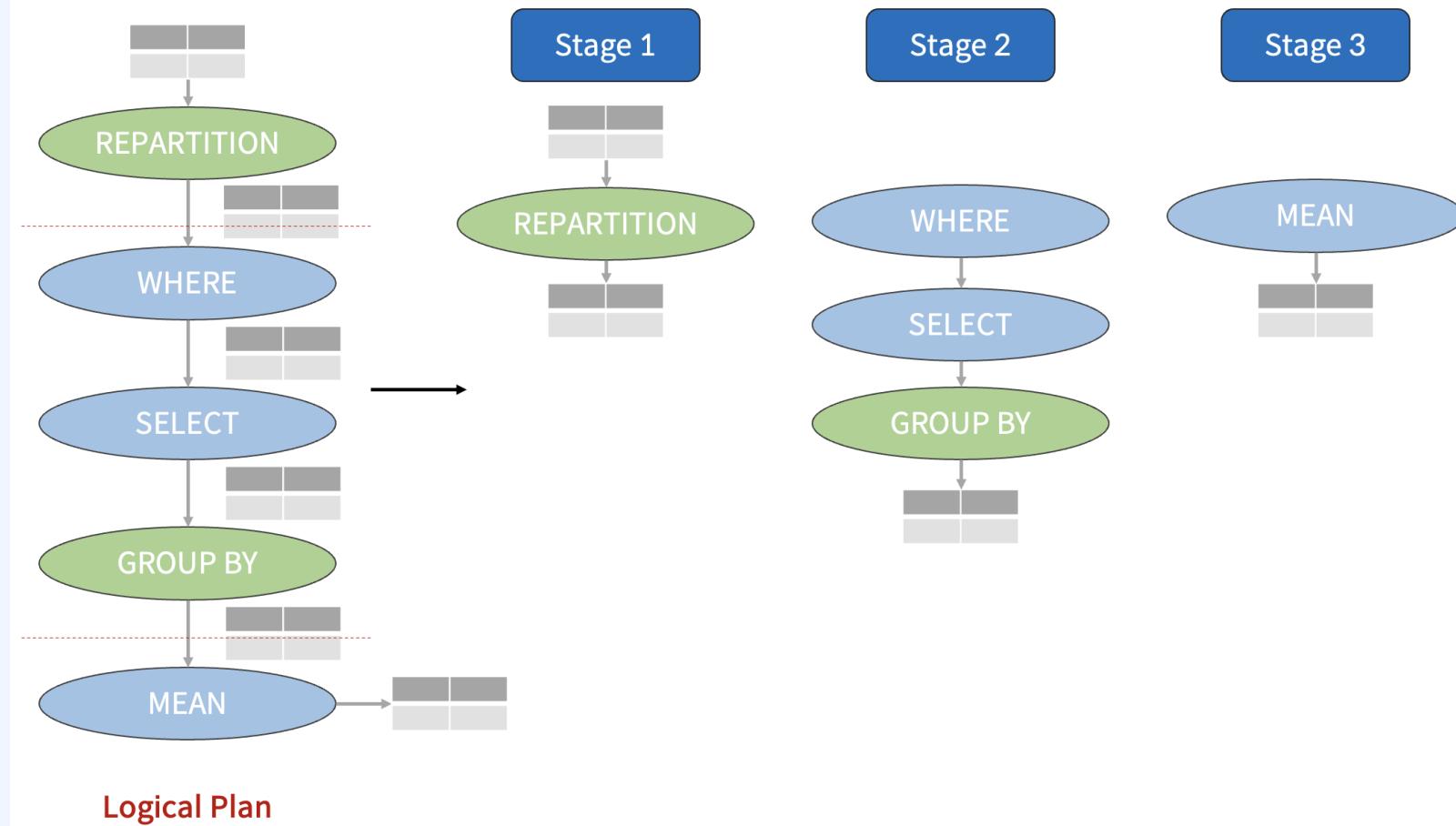
3. Logical Plan

```
df = df.repartition(5) \
    .where("Sex == 'male'"") \
    .select("Survived", "Pclass", "Fare") \
    .groupby("Survived", "Pclass") \
    .mean()
print(f"result ===> {df.collect()}"")
```

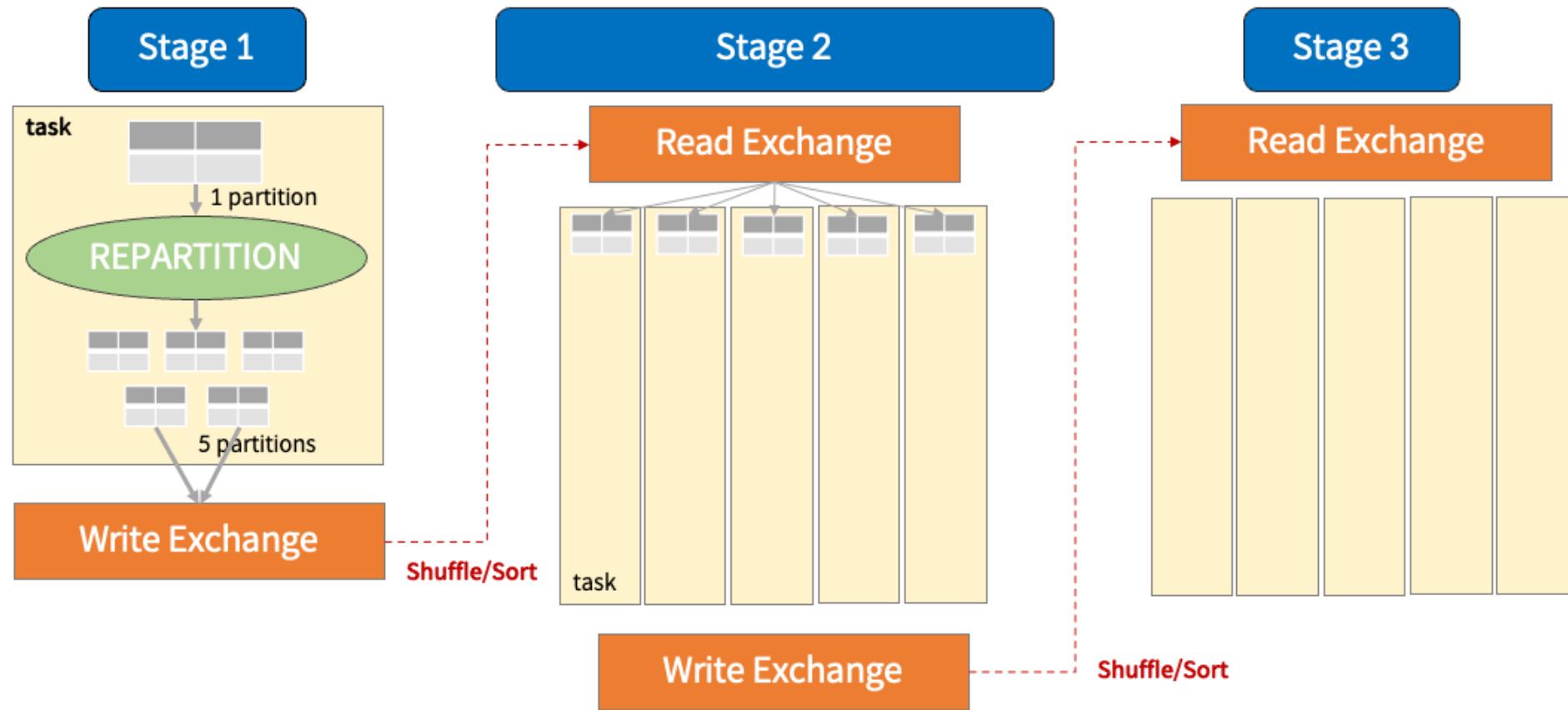


Logical Plan

3.1 Logical Plan 기준 Stage 구성



3.2 Logical Plan 기준 Stage - Deep Dive



4. 용어 정리 (Job / Stage / Shuffle / Task)

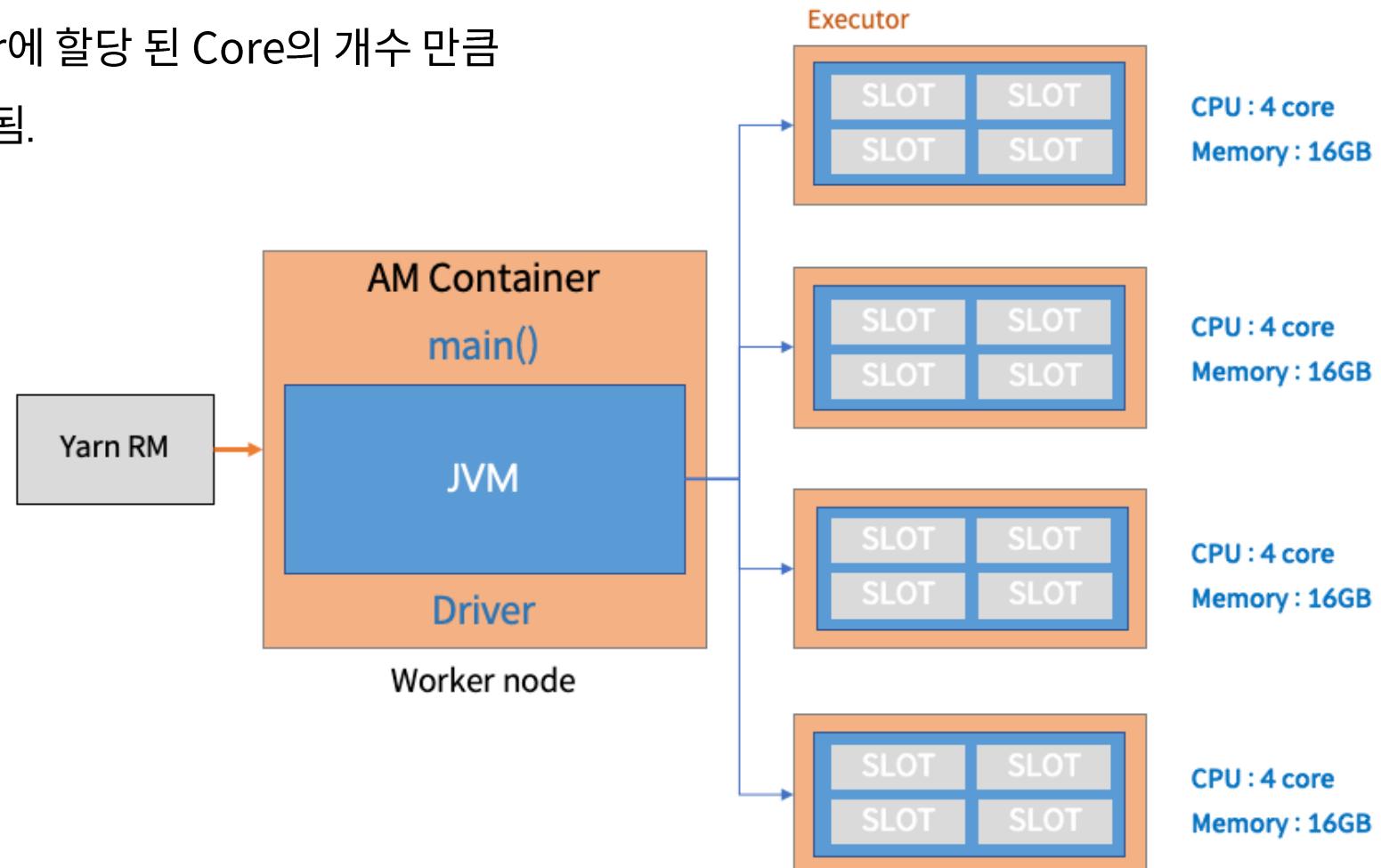
- **Job**
 - Spark는 각 action을 기준으로 1개의 Job을 생성.
 - 1개의 Job에는 여러 개의 transformation이 존재할 수 있는데, 스파크 엔진은 이 transformation들을 바탕으로 logical plan을 생성.
- **Stage**
 - 만들어진 logical plan을 바탕으로, Job을 wide dependency transformation 단위로 쪼개 Stage를 생성.
 - 만약 wide 가 없다면, Job은 1개의 stage만 가지게 되고, N개의 wide transformation이 있다면, Stage는 N + 1 개가 됨.

4. 용어 정리 (Job / Stage / Shuffle / Task)

- **Shuffle/Sort**
 - Stage 간의 데이터는 Shuffle / Sort 연산을 통해 복사 후 전달됨.
- **Task**
 - Spark Job에서 가장 작은 작업 단위.
 - 각 stage는 1개 이상의 병렬 task로 수행.
 - 만들어지는 task의 개수는 그 stage에서 사용되는 partition의 개수와 동일.
 - task의 개수가 어떻게 세팅되느냐에 따라 stage의 처리 시간이 크게 차이나기 때문에, 매우 중요.
 - 특정 task를 특정 executor에 할당하는 것은 driver의 역할.
할당 후에는 각 executor에서 task 수행을 담당.

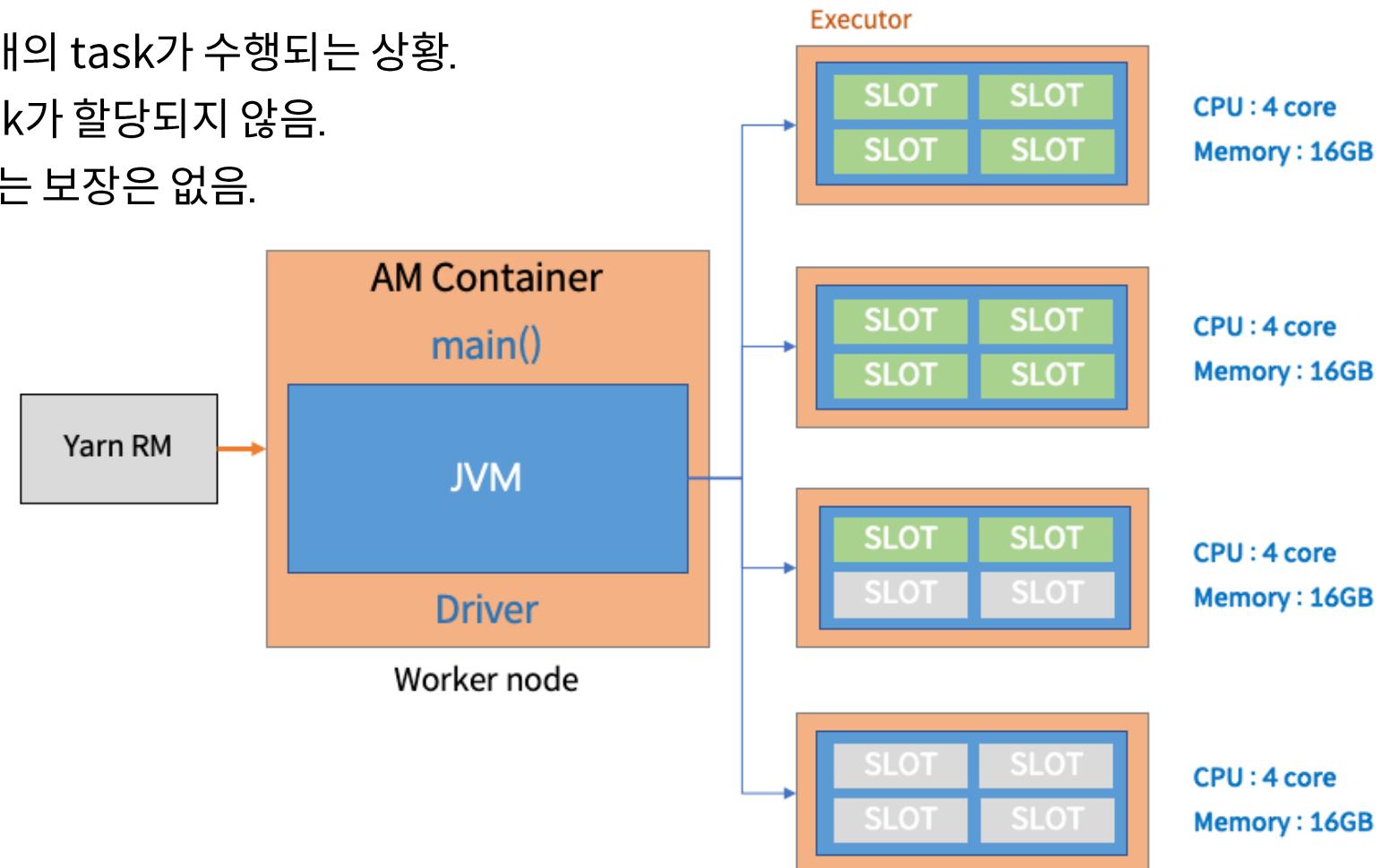
5. slot

각 Executor에 할당 된 Core의 개수 만큼
Slot이 할당됨.



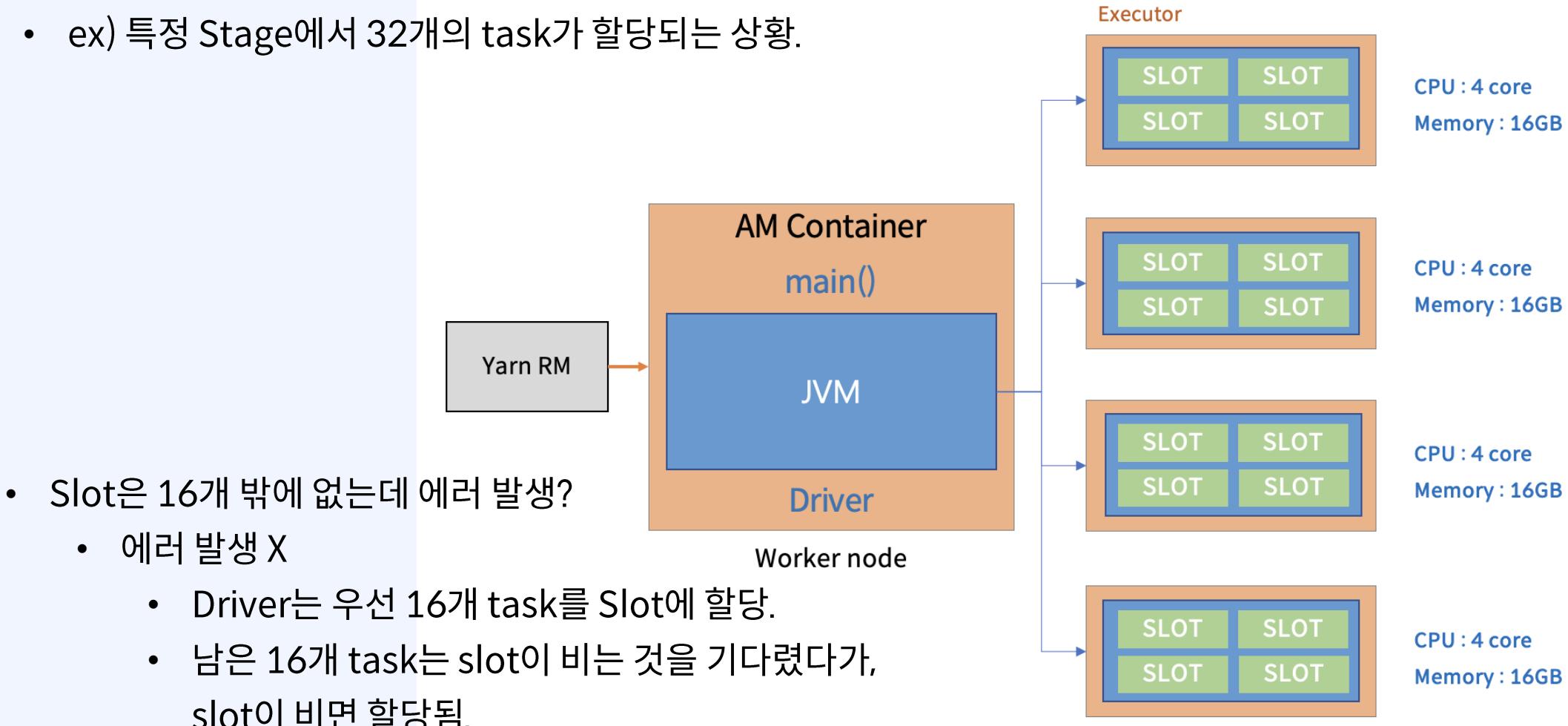
5.1 slot example 1)

- ex) 특정 Stage에서 10개의 task가 수행되는 상황.
 - 일부 SLOT에는 task가 할당되지 않음.
- 꼭 이 예시처럼 할당된다는 보장은 없음.



5.2 slot example 2)

- ex) 특정 Stage에서 32개의 task가 할당되는 상황.



6. Action - collect

- collect() action 수행 시, executor의 각 task는 driver로 데이터를 전송.
- 모든 task가 성공했을 때 driver는 이 job이 성공했다고 여김.
- 만약 특정 task가 실패했다면, driver는 그 task를 재시도.
 - 그 task를 다른 executor에서 실행할 수 있음.
- retry도 실패한다면, driver는 exception을 발생시키고 job은 실패.

```
df = df.repartition(5) \
    .where("Sex = 'male'") \
    .select("Survived", "Pclass", "Fare") \
    .groupby("Survived", "Pclass") \
    .mean()
print(f"result ===> {df.collect()}")
```

Join 의 종류

1. 실습 코드 링크)

- https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/join_broadcast_hash_join_ex.py
- https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/join_sort_merge_join_ex.py

2. Join

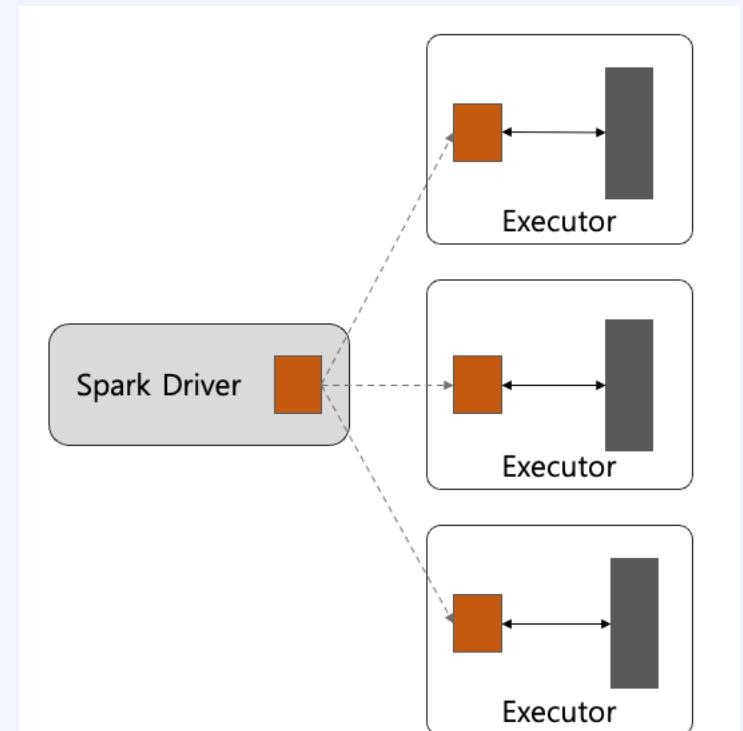
- RDD나 Dataframe의 형태로 되어 있는 두 데이터를 공통적으로 일치하는 키를 기준으로 병합하는 연산.
- 앞의 실습에서는 inner, left outer, right outer, full outer 방법에 대해 확인.
- Join 연산들은 Executor 사이의 방대한 데이터 이동이 발생할 수 있음.
Spark 성능의 핵심 요소!

3. Join의 종류

- Broadcast Hash Join (BHJ)
- Sort Merge Join (SMJ)
- Shuffle Hash Join (SHJ)
- Broadcast Nested Loop Join (BNLJ)
- Categorical Product Join

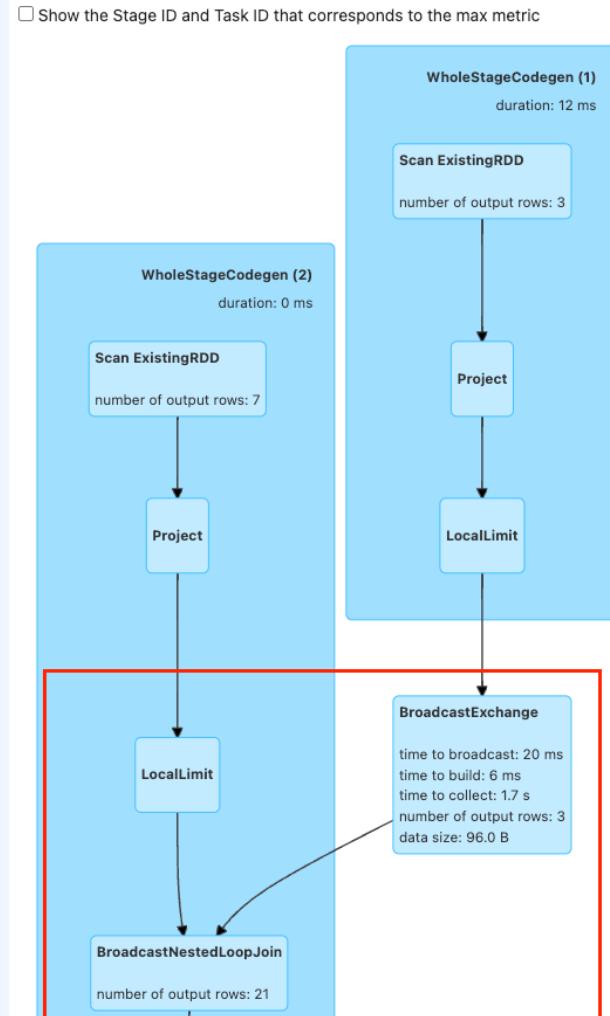
3.1 Join의 종류 - 1) Broadcast Hash Join (BHJ)

- map-side only join 이라고도 함.
- 큰 데이터셋과, 작은 데이터셋을 Join할 때 사용.
- Broadcast variable을 사용해, 더 작은 쪽의 데이터가 Driver에 의해 모든 Executor에 복사되고, 각각의 Executor에 나뉘어 있는 큰 데이터와 Join됨.
- 데이터 교환, 즉 Shuffle이 거의 발생하지 않으므로 효율적.
- 관련 파라미터)
`spark.sql.autoBroadcastJoinThreshold (default : 10MB)`
 - 데이터셋의 크기가 threshold 이하라면 자동으로 BHJ를 수행.



3.1 Join의 종류 - 1) Broadcast Hash Join (BHQ)

- in spark UI.

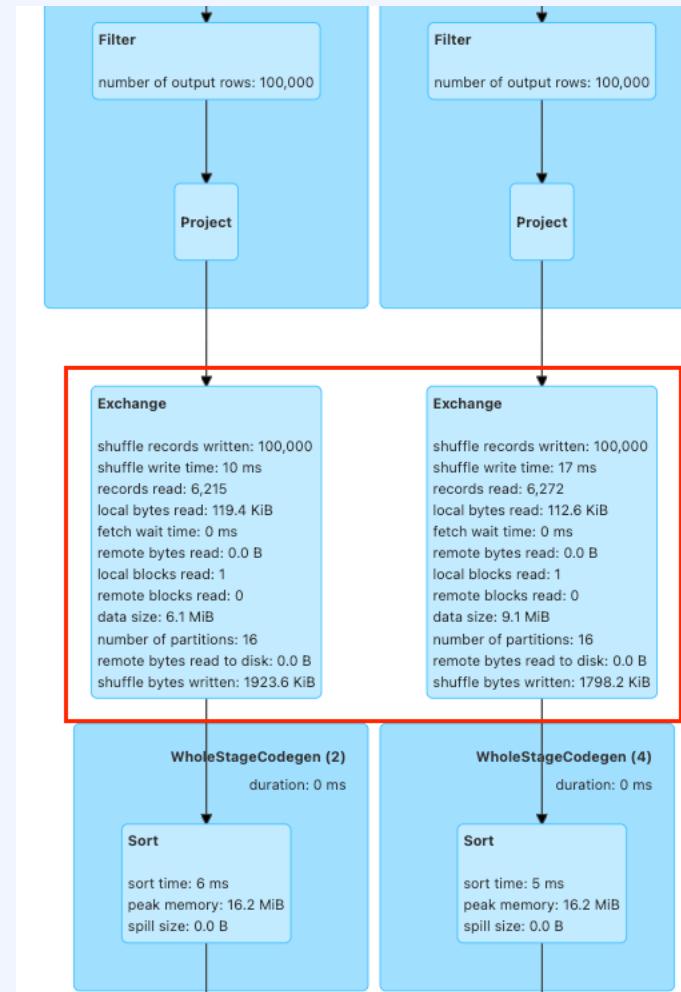


3.2 Join의 종류 - 2) Sort Merge Join(SMJ)

- 정렬 가능하고 겹치지 않으면서, 공통의 파티션에 저장이 가능한 공통 키를 기반으로, 큰 두 종류의 데이터셋을 합칠 수 있는 방법.
- 해시 가능한 공통 키를 가지면서, 공통 파티션에 존재하는 두 가지의 데이터셋을 사용.
 - 각 데이터셋의 동일 키를 가진 데이터셋의 모든 row가, 동일 executor의 동일 파티션에 존재하도록 Shuffle 작업이 필요.(데이터 저장 방식 등에 따라 생략될 수 있음.)
- Sort, Merge 두 단계가 존재.
 - Sort
 - 각 데이터 셋을 Join 연산에 사용한 키를 기준으로 정렬.
 - Merge
 - 각 데이터 셋에서 키 순서대로 데이터를 순회하며, 키가 일치하는 Row끼리 병합.
- 관련 파라미터 :
 - `spark.sql.join.preferSortMergeJoin`

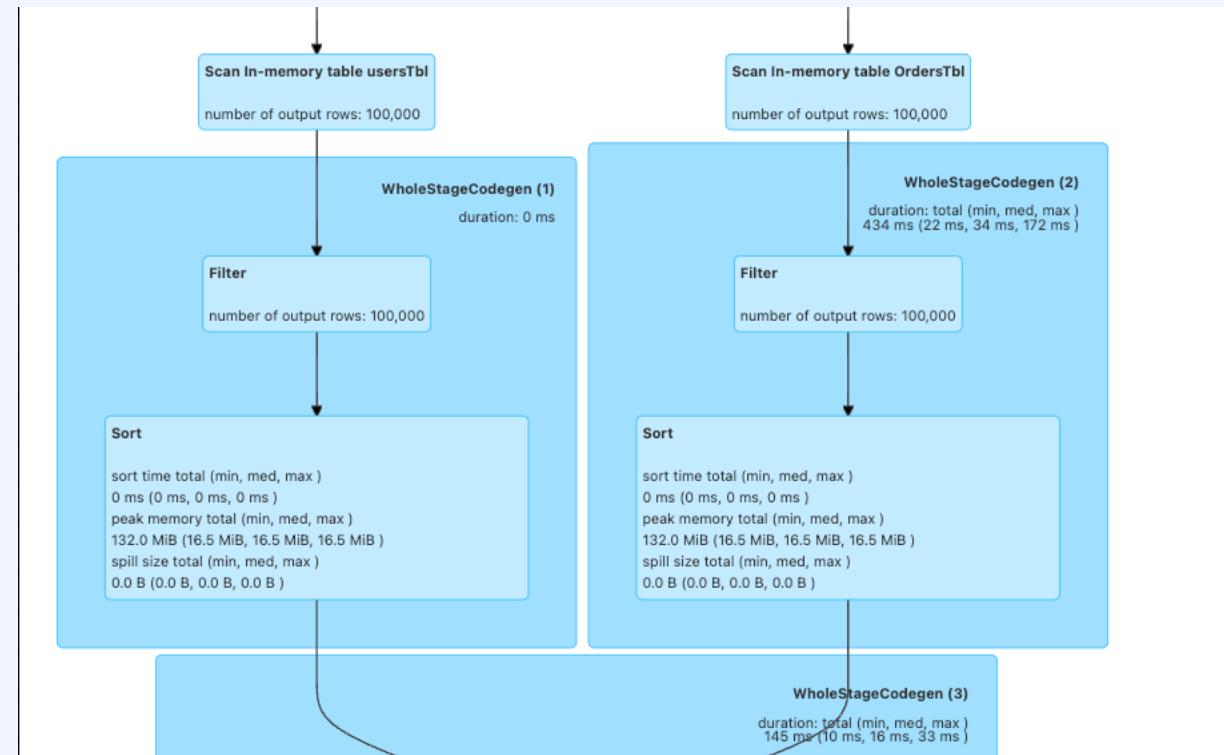
3.2 Join의 종류 - 2) Sort Merge Join(SMJ)

- in Spark UI (최적화 전)



3.2 Join의 종류 - 2) Sort Merge Join(SMJ)

- in Spark UI (최적화 후)
 - 데이터 사전 정렬
 - Bucketing.
- Exchange 과정 없어짐.
- 정렬에 소요되는 시간 : 0ms.



스파크에서의 메모리 할당

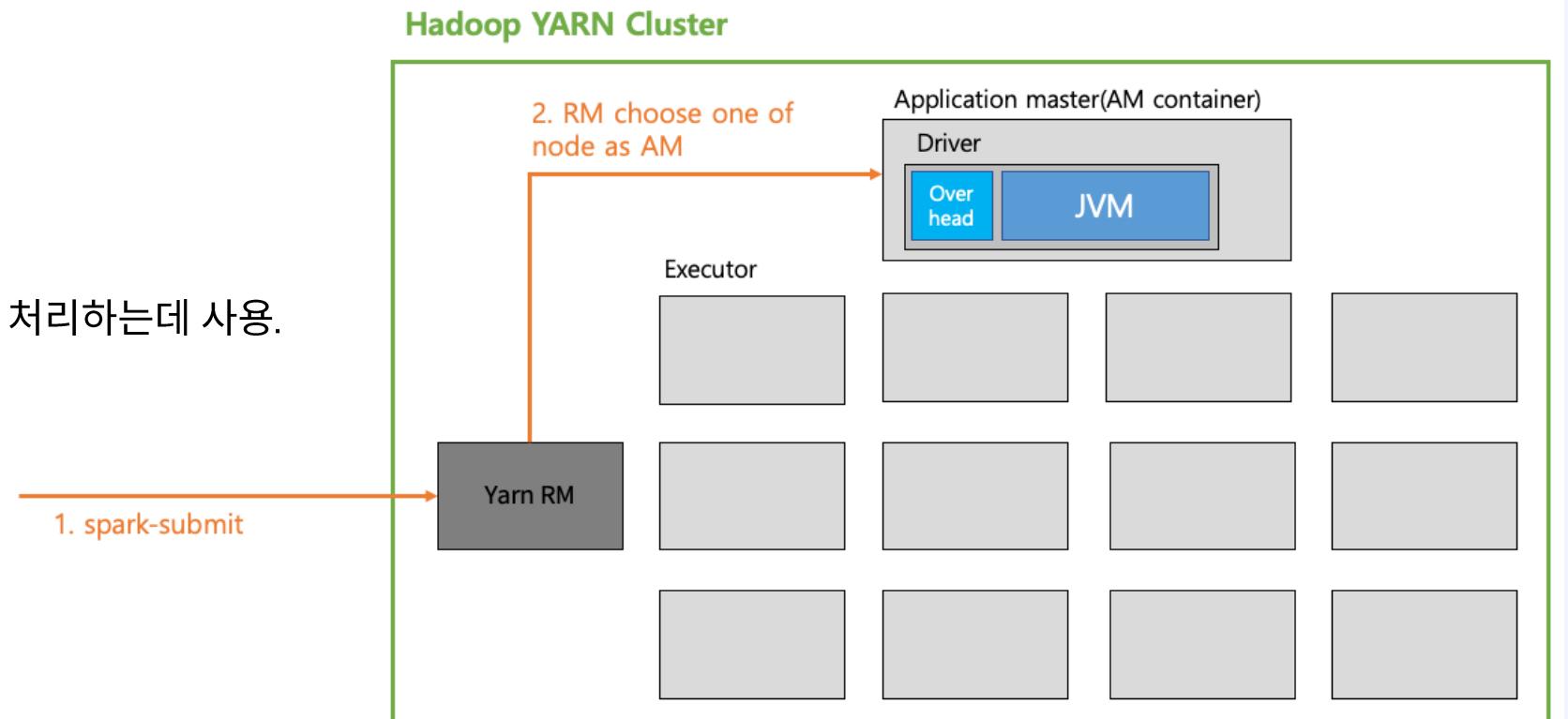
1. Driver 메모리 할당

관련 spark 파라미터)

1. JVM memory(spark.driver.memory = 1GB)
2. Overhead(spark.driver.memoryOverhead = 0.1) -> max (10% or 384MB)

Overhead ?

- JVM heap이 아닌 공간
- 컨테이너 프로세스,
JVM 이 아닌 프로세스를 처리하는데 사용.

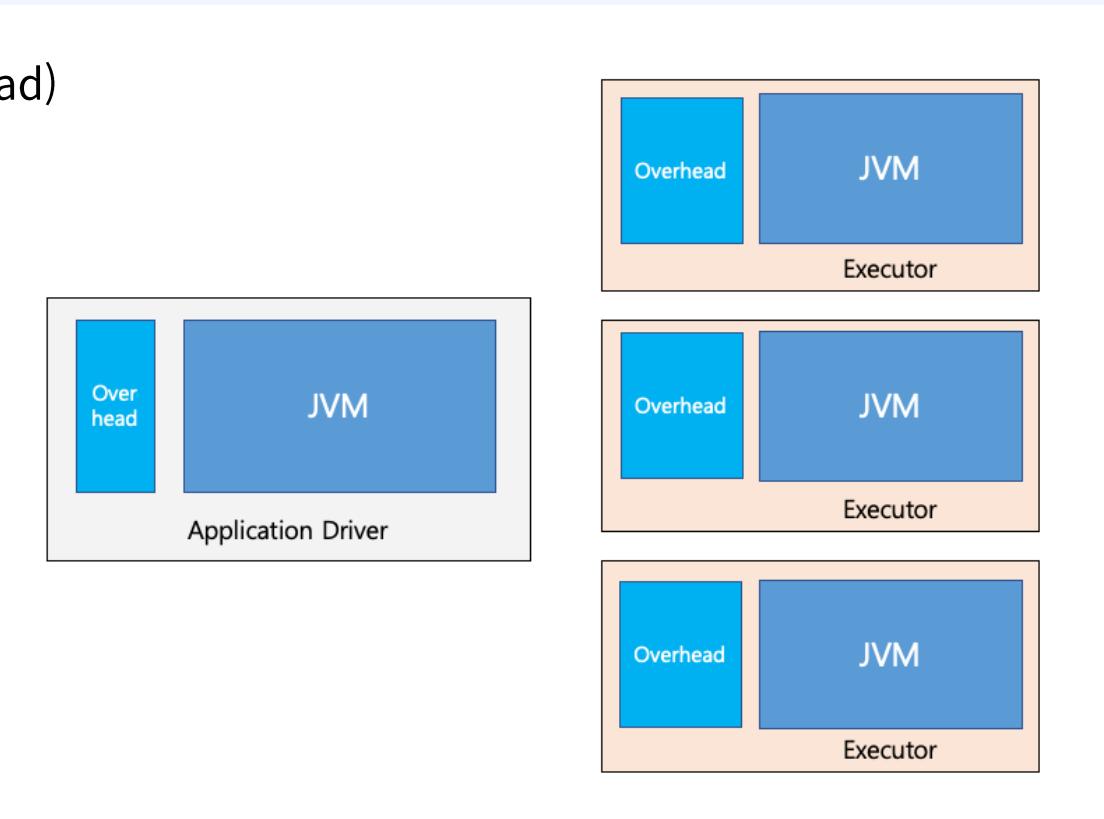


2. Executor 메모리 할당

관련 spark 파라미터)

1. Heap Memory (spark.executor.memory)
2. Overhead Memory (spark.executor.memoryOverhead)
3. Offheap Memory (spark.memory.offHeap.size)
4. Pyspark Memory (spark.executor.pyspark.memory)

Offheap, Pyspark Memory가
양수 값으로 세팅이 된다면, Overhead 영역에 포함됨.
(JVM 영역과는 분리)



2.1 Executor 메모리 할당 예시

Ex)

`spark.executor.memory = 8GB`

`spark.executor.memoryOverhead = 0.1 (8 x 0.1 = 0.8GB)`

`spark.memory.offHeap.size = 0`

`spark.executor.pyspark.memory = 0`

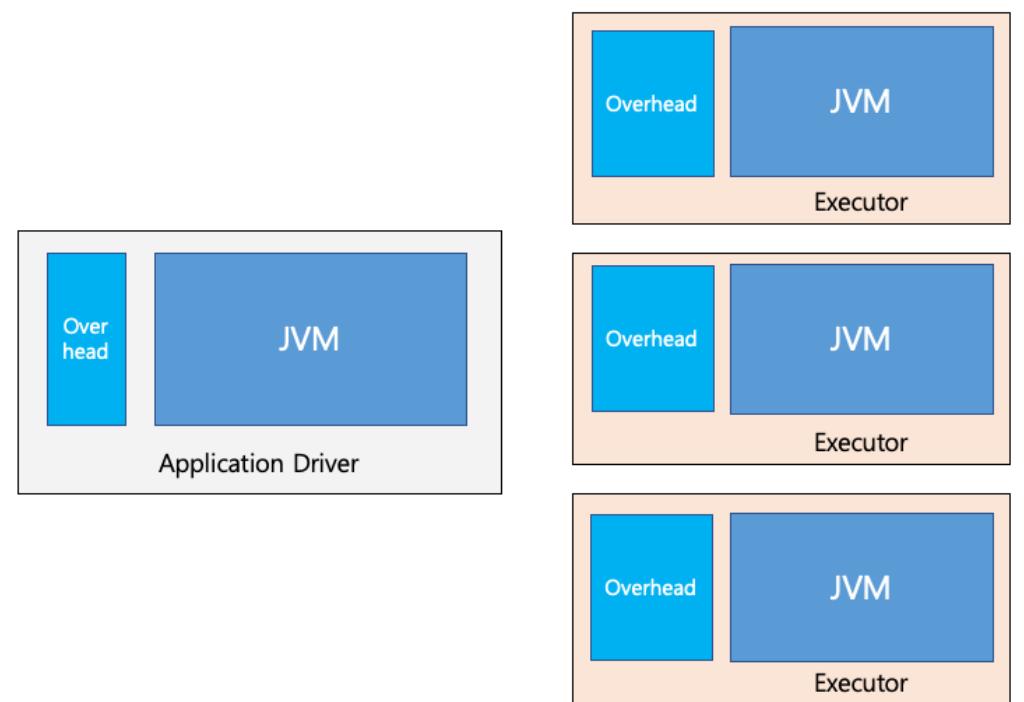
-> Spark driver는 Yarn RM에 Executor 하나 당

8.8GB의 메모리를 요청함.

-> 요청한 만큼 그대로 받아올 수가 있는가?

(만약 Executor가 돌아가는 Worker node의 물리 메모리가
8.8GB보다 작다면?)

-> X



2.3 Overhead Memory

- Non-JVM 프로세스들이 처리되는 공간.
- Non-JVM 프로세스의 예)
 - network buffer
 - 셔플 데이터 교환
 - 원격 스토리지로부터 파티션 데이터를 읽어오기 등.

2.3 Pyspark executor memory

- Scala나 Java spark를 사용하는 경우에는 고려할 필요 없음.
- Pyspark를 사용하는 경우,
spark.executor.pyspark.memory 파라미터를 따로 지정하지 않으면,
Spark는 Python의 메모리 사용을 제한하지 않으며,
Python 외에 다른 non-JVM 프로세스와 공유하는 오버헤드 메모리 공간을
초과하지 않도록 하는 것은 application의 역할!

참고 링크

Spark configuration)

<https://spark.apache.org/docs/latest/configuration.html>

Yarn configuration)

<https://www.ibm.com/docs/en/spectrum-scale-bda?topic=tuning-yarn>

스파크 메모리 관리

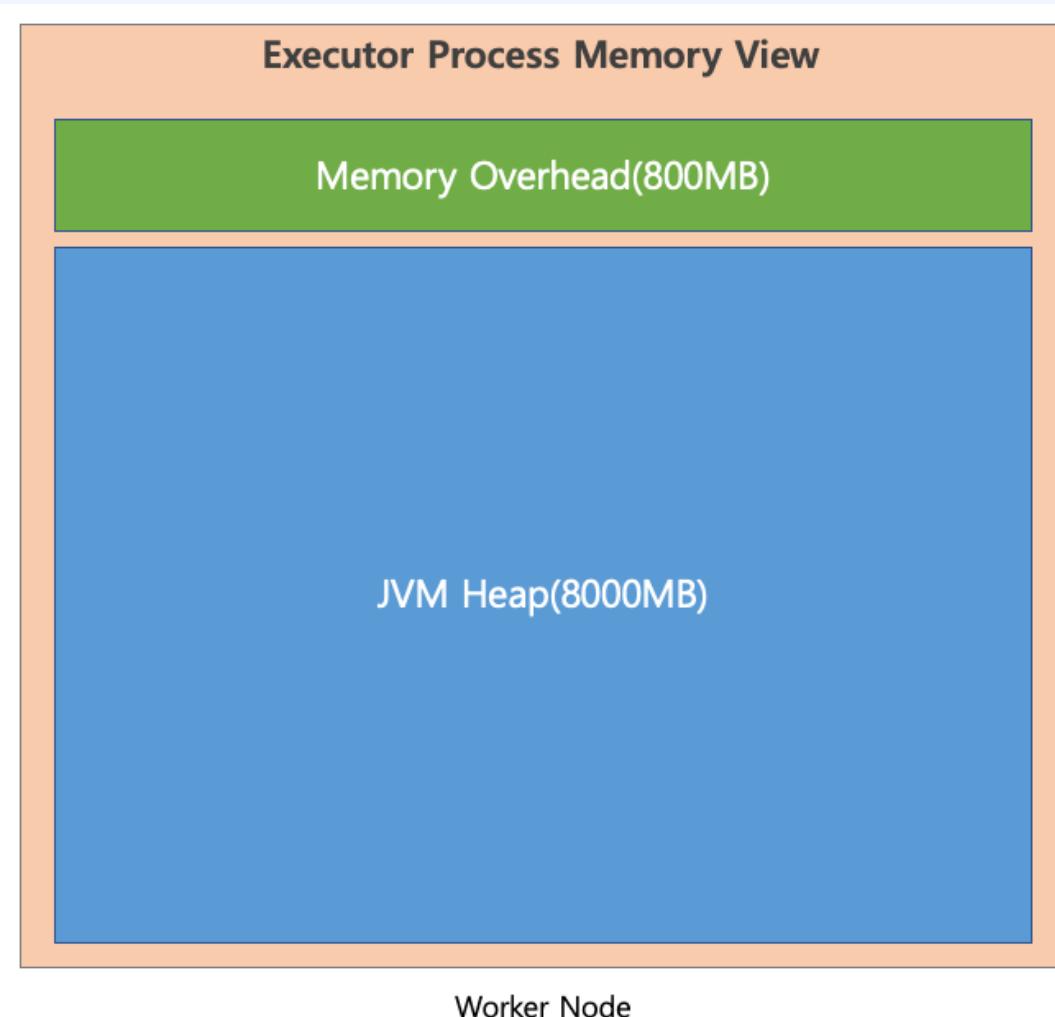
1. Executor 메모리 구조

Ex)

spark.executor.memory = 8GB

spark.executor.cores = 4

→ default Memory overhead : 800MB

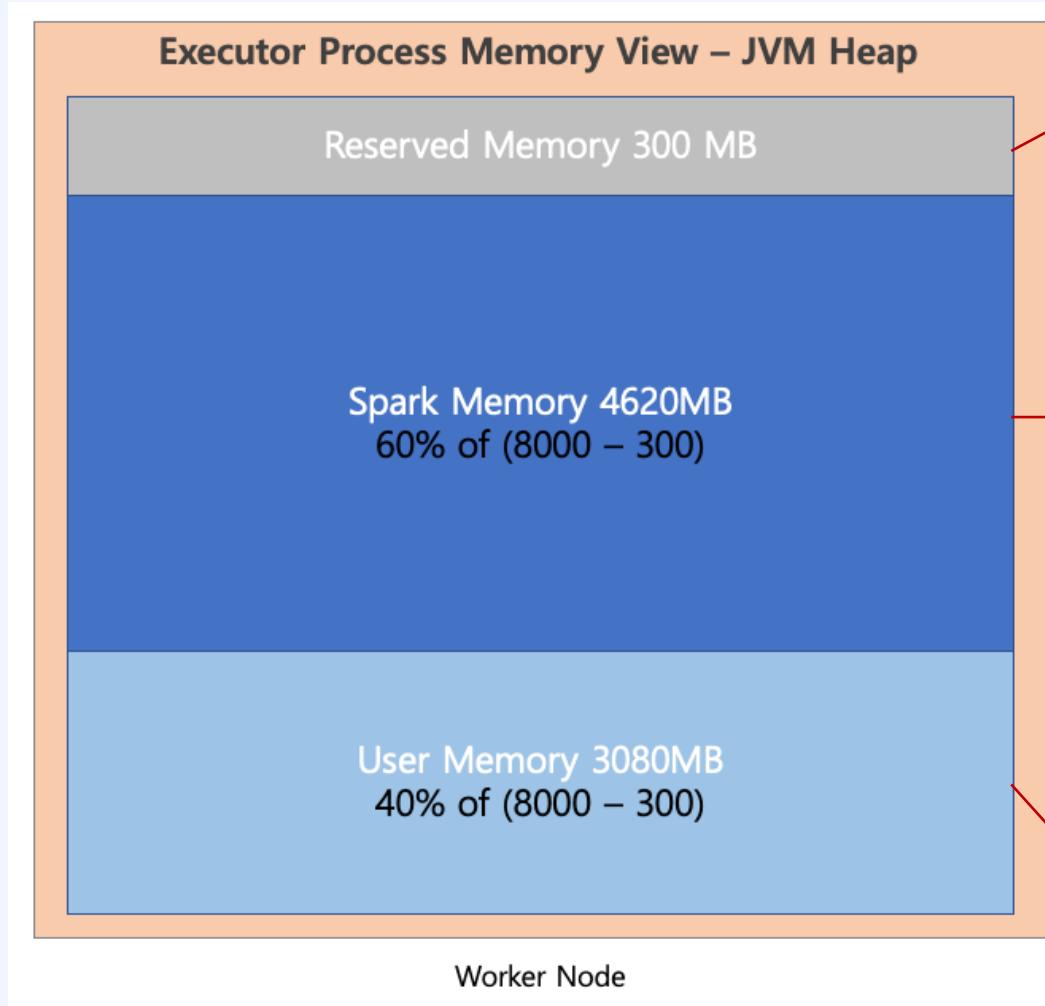


1.1 Executor 메모리 구조 – JVM Heap

Ex)

```
spark.executor.memory=8GB  
spark.executor.cores=4
```

- Overhead를 제외하고,
JVM Heap은 세가지 영역으로
구분
- 1. Reserved Memory
- 2. Spark Memory
- 3. User Memory



스파크 엔진이 고정적으로 점유하는 영역
(변경 불가)

`spark.memory.fraction = 0.6`

- Dataframe 연산에 사용.
- 1. Dataframe operations
- 2. Caching
 - (Dataframe이 RDD로 컴파일 되더라도 User Memory를 사용하지는 않음)

Spark Memory 할당 후 남는 영역

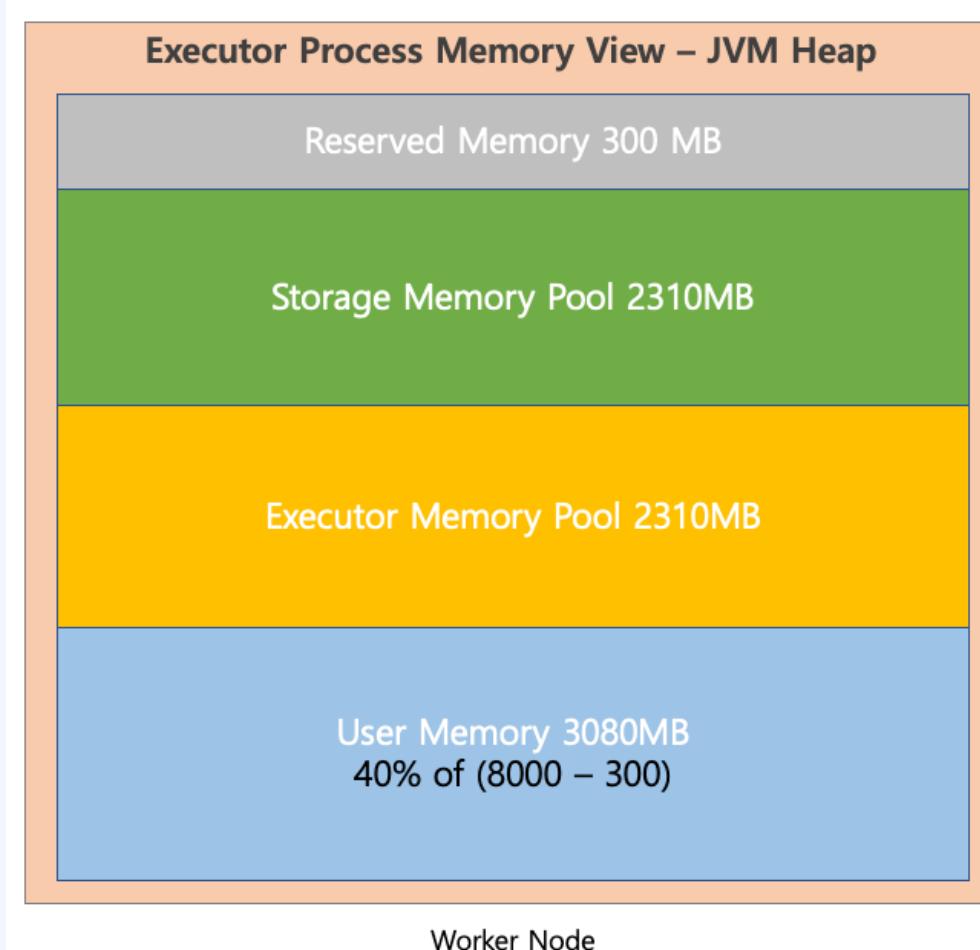
- User operations에 사용
- 1. User-defined 자료 구조
(HashMap etc)
- 2. Spark 내부 메타데이터
- 3. UDF 함수
- 4. RDD Conversion 연산
- 5. RDD 계보, 의존 관계

1.2 Executor 메모리 구조 – Storage/Executor Memory pool

Ex)

`spark.executor.memory=8GB`
`spark.executor.cores=4`

- Spark Memory는 다시
Storage Memory Pool과
Executor Memory Pool로 구분



`spark.memory.storageFraction = 0.5`
Cache for DataFrame

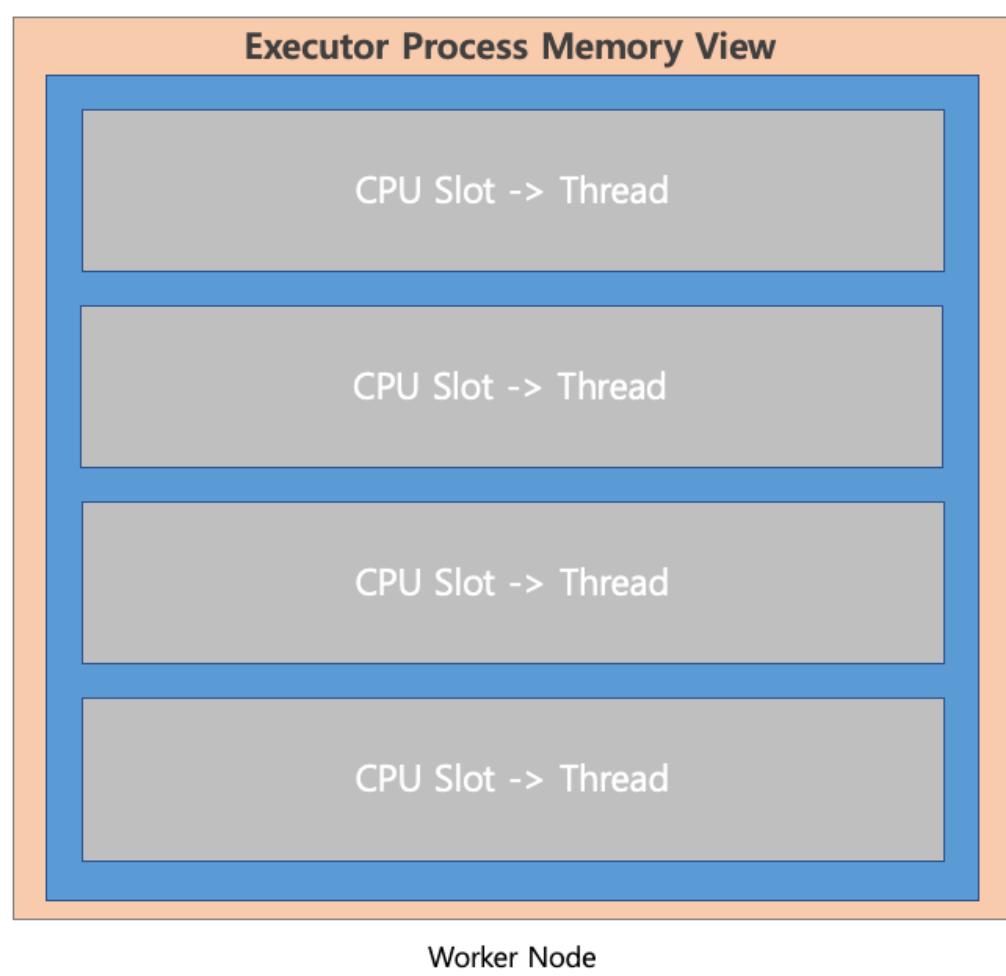
DataFrame 연산을 위한 Buffer

1.3 Executor 메모리 구조 – Slot

Ex)

```
spark.executor.memory=8GB  
spark.executor.cores=4
```

- 각각의 CPU Slot은 Process가 아닌 Thread.
- Executor 당 1개의 JVM 존재.

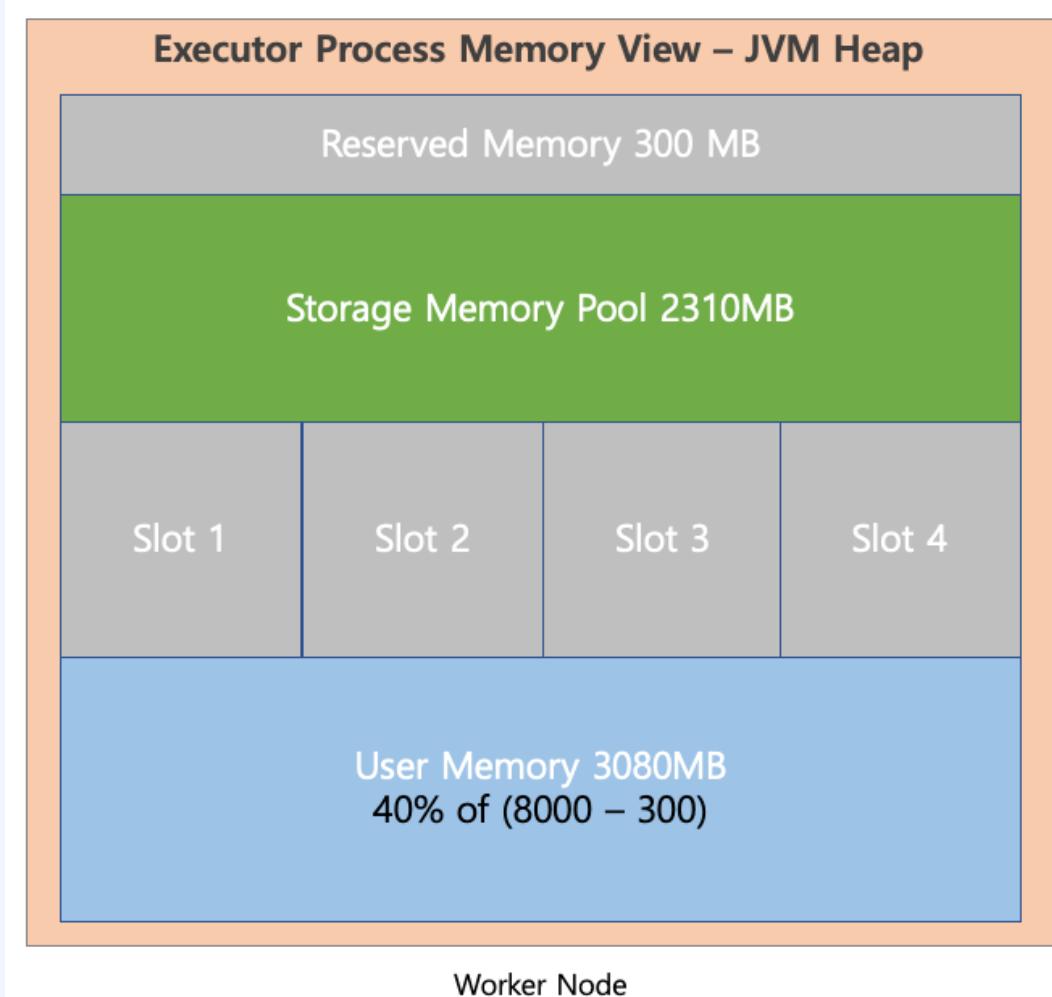


1.4 Executor 메모리 구조 – static memory management

Ex)

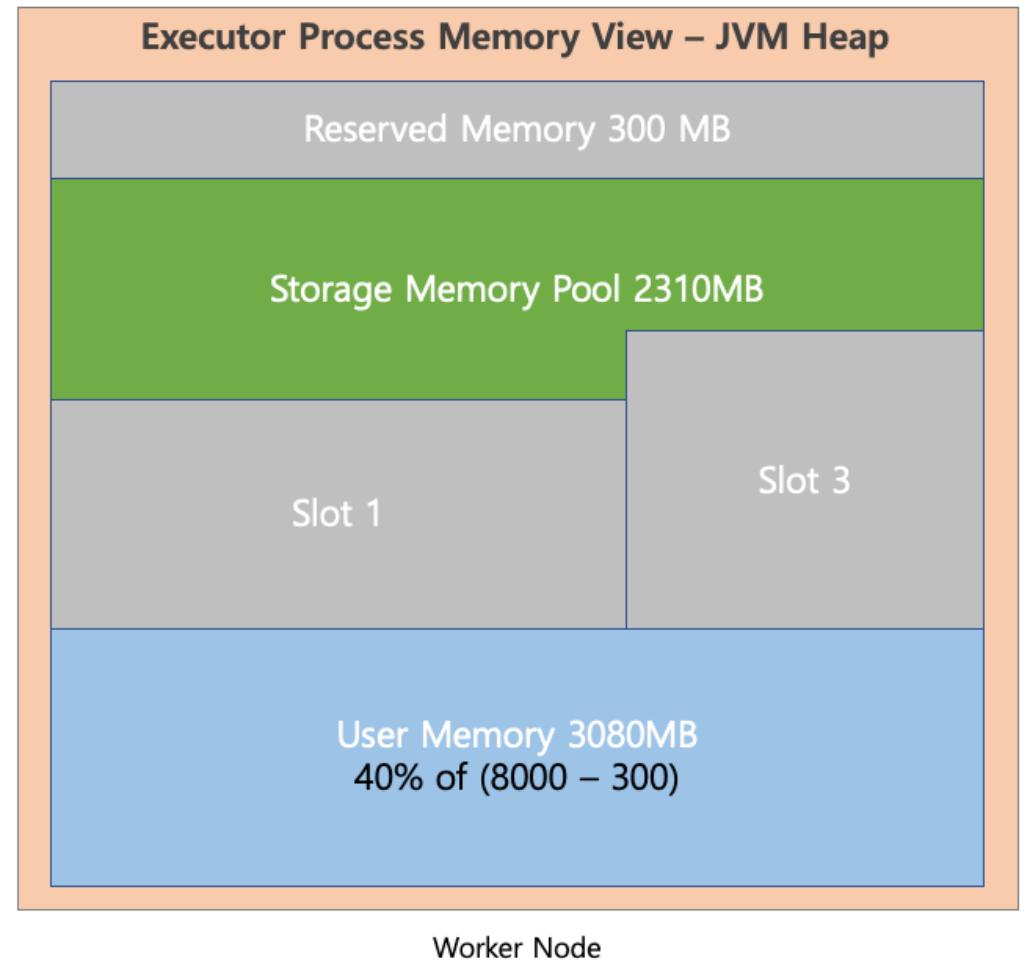
```
spark.executor.memory=8GB  
spark.executor.cores=4
```

- 각 Slot이 Executor Memory Pool을 공평하게 차지.
- Spark 1.6 까지는 기본 설정.



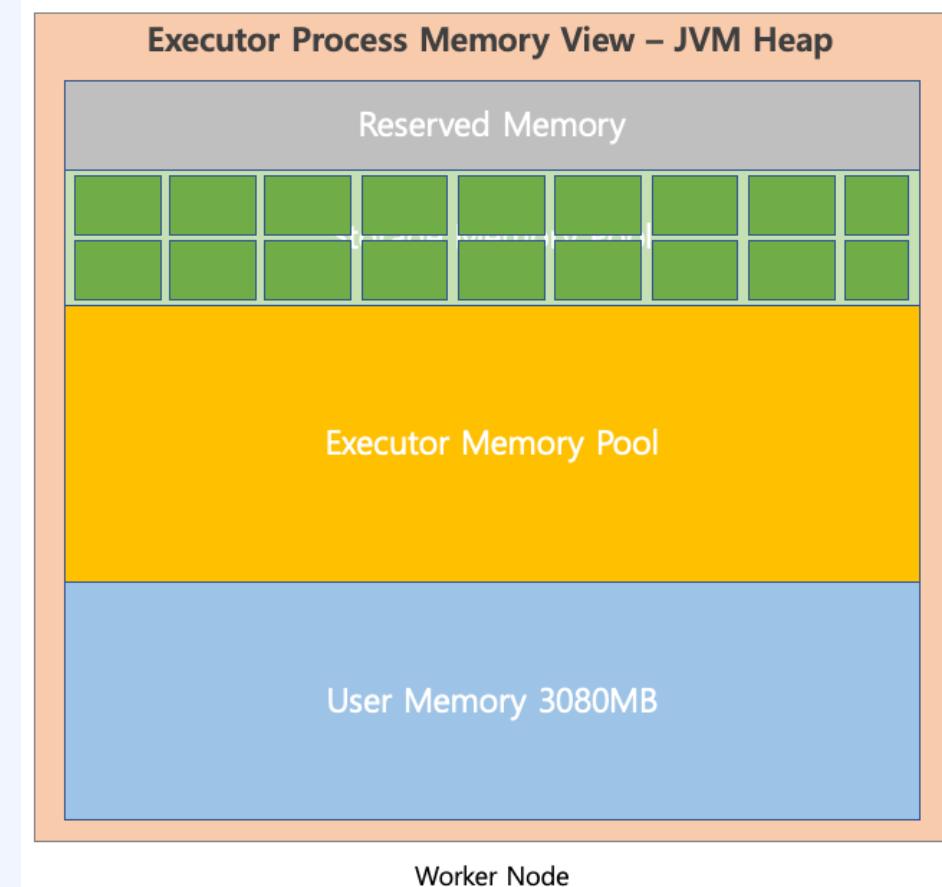
1.5 Executor 메모리 구조 – unified memory manager

- 활성화된 task를 기준으로,
공평하게 자원을 분배.
- Ex) 4개 Slot 중 2개 Slot(Slot1, Slot 3)만
활성화된 상태라면?
 - Slot 1, Slot 3만 Executor memory pool에 할당.
 - 만약 Executor memory pool이
꽉 찼다면, Storage memory pool 영역도
사용이 가능.
- 반대로, 캐싱해야할 데이터가 많아서 Storage Memory
pool이 꽉 찼다면,
Executor memory pool 영역도 사용 가능



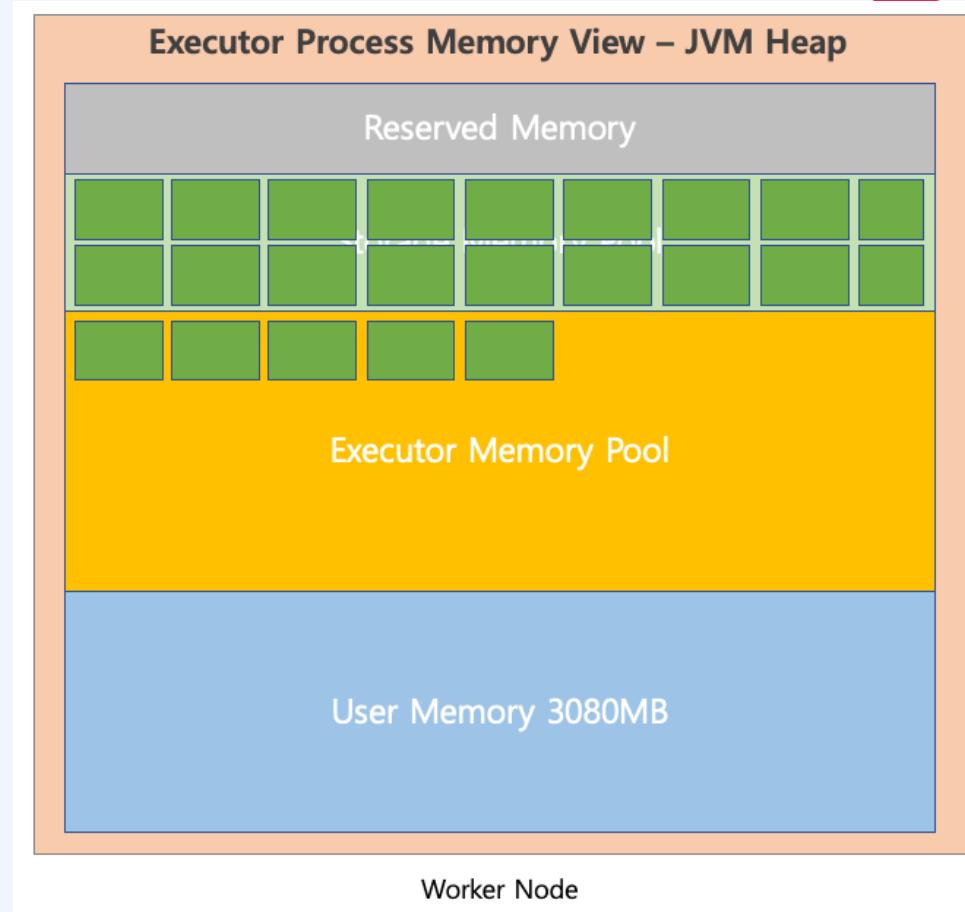
1.6 Executor 메모리 구조 – 메모리 사용 시나리오

1. DataFrame을 캐싱하여 storage memory pool이 꽉 찬 상태.
-> 만약 더 캐싱해야 된다면?



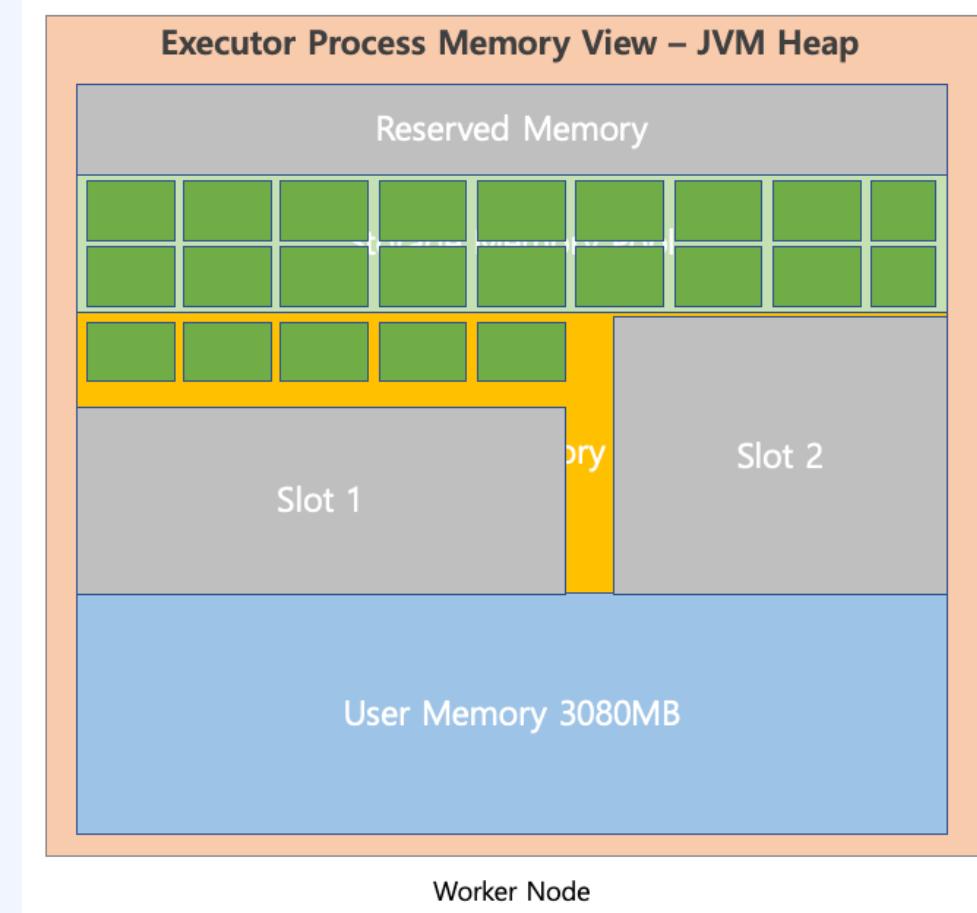
1.6 Executor 메모리 구조 – 메모리 사용 시나리오

2. Executor Memory Pool 의 남는 영역을 사용.
-> Task가 실행되야 한다면?



1.6 Executor 메모리 구조 – 메모리 사용 시나리오

3. Executor Memory Pool에 Slot 할당.
-> Executor Memory Pool이 꽉 차게 된다면?



1.6 Executor 메모리 구조 – 메모리 사용 시나리오

4. Executor Memory Pool에 할당 되어 있던 캐시 데이터는 Disk로 Spill이 되고, 남는 영역은 Slot이 차지함.

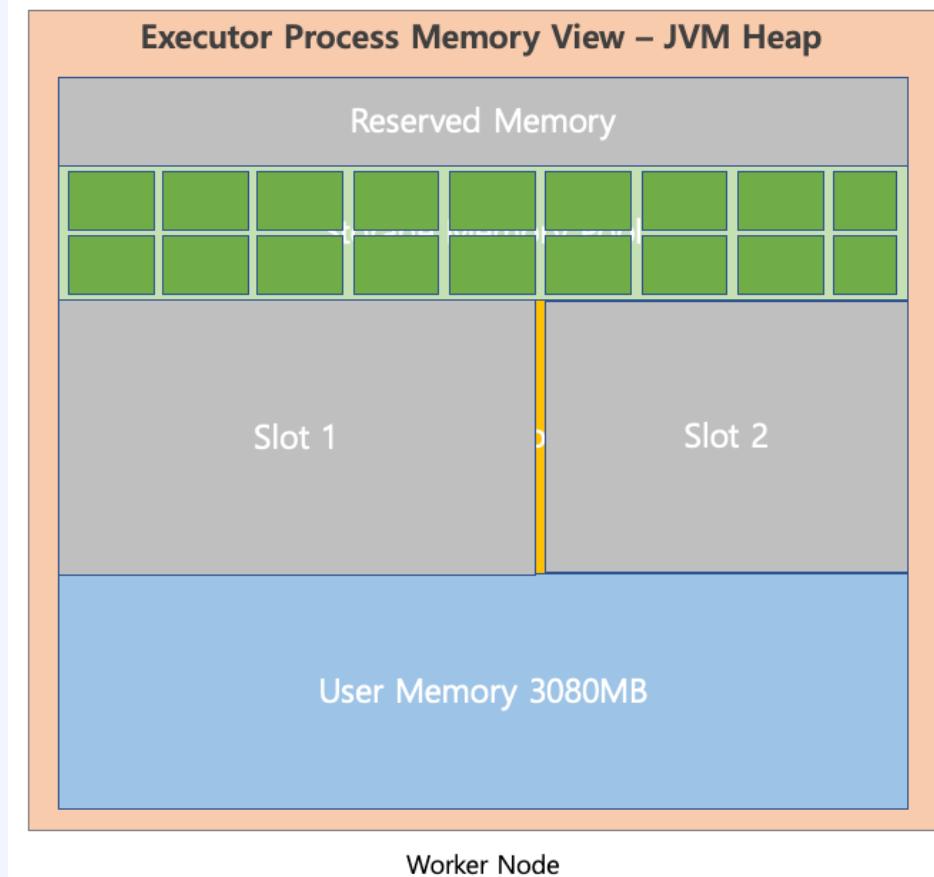
- Executor Memory Pool은 원래 Slot이 차지해야 하는 영역이기 때문.

-> Executor memory가 더 필요하다면?

- Memory manager는 필요 없는 데이터들을 disk로 추가로 spill 시도 함.

-> 만약 Spill이 안된다면?

- OOM (Out of Memory) Exception 발생!



1.7 Executor 메모리 구조 – 메모리 관련 스파크 파라미터 정리

JVM Heap 관련 설정

1. spark.executor.memoryOverhead
2. spark.executor.memory (= total JVM Heap size)
3. spark.memory.fraction(= JVM Heap 중, reserved 영역을 제외하고, DataFrame 연산에 사용하게 될 영역 비율)
4. spark.memory.storageFraction(= DataFrame 영역 중 캐시 영역의 비율)
5. spark.executor.cores(= 한 executor 내에서 최대로 사용 가능한 스레드의 개수)
 - 너무 작게 세팅하면, 리소스를 온전히 사용하지 못함.
 - 너무 크게 세팅하면, 스레드간 경합 발생.
 - 권장 : 2~4

1.8 Executor 메모리 구조 – Off heap

- 대부분의 스파크 연산, 데이터 캐싱은 JVM heap 내에서 발생.
- heap 메모리를 사용할 때 효율적인 경우가 많으나,
JVM heap은 garbage collection이 발생함
-> Executor의 heap 메모리에 많은 양의 데이터를 할당할 때, GC delay가 크게 발생할 수 있음.
- Spark 3.0+ 에서는 off-heap 메모리에서 몇몇 연산을 수행하는 것이 최적화
- 큰 메모리 사이즈를 요구할 때 on-heap, off-heap을 섞는 것은 GC delay를 줄이는데 도움이 됨.
- 기본 설정은 off-heap을 사용하지 않음

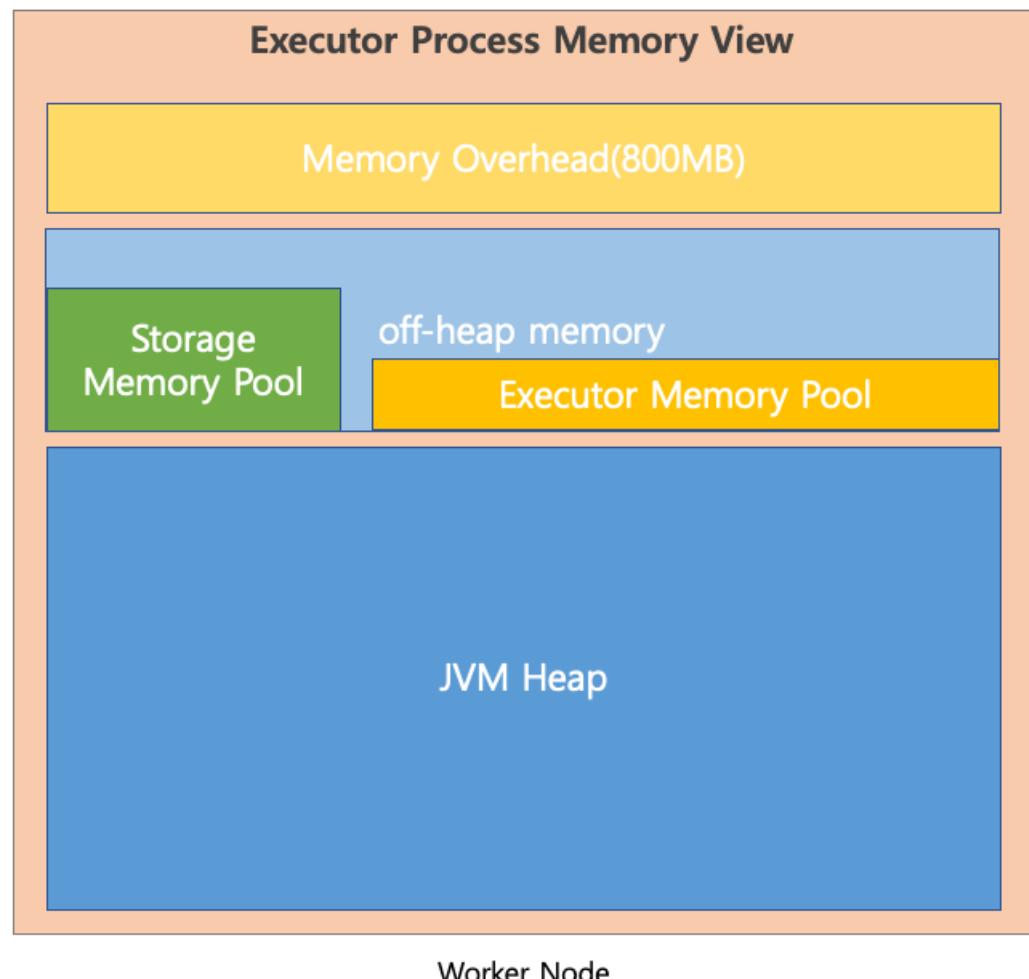
JVM off heap 관련 설정

1. spark.memory.offHeap.enabled(default : false)
2. spark.memory.offHeap.size

1.8 Executor 메모리 구조 – Off heap

Off heap size 설정 시.

- 기존 Jvm heap 내의 Storage Memory Pool, Executor Memory pool 가, Off-heap 영역에도 일부 할당될 수 있음.
(executor, storage memory pool의 사이즈를 확장하는 간접적인 방법)



1.9 Executor 메모리 구조 – pyspark memory.

- Scala, Java가 아닌 PySpark 사용시, Python worker가 추가로 필요할 수도 있음.
- Python worker는 JVM heap memory 를 직접 사용하지 못함.
-> off-heap, overhead memory를 사용해야 함.
- 두 memory외에 추가로 memory가 필요할 때,
`spark.executor.pyspark.memory` 파라미터를 세팅
- 기본 세팅은 `spark.executor.pyspark.memory` 를 사용하지 않는 것.
 - 대부분의 Pyspark application의 경우, 외부 Python 라이브러리를 필요로 하지 않음.
-> Python worker를 필요로 하지 않음.

Repartition, Coalesce에 대한 이해

1. 실습 코드 링크

- link)
 - https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/repartition_coalesce_ex.py

2. Repartition

- Wide dependency transformation -> (Shuffle 동반)
- RDD
 - 1. repartition(numPartitions)
- DataFrame
 - 1. repartition(numPartitions, *cols)
 - Hash 기반의 partitioning
 - 2. repartitionByRange(numPartitions, *cols)
 - 값들의 범위를 기반으로 하는 partitioning.
 - partitioning 의 범위를 정하기 위해, 데이터 샘플링 작업을 수행

2. Repartition

Ouput partition 개수

- Output partition 개수는, 기본적으로 파라미터로 넘겨준 numPartitions.
- 만약 numPartitions를 넘기지 않았다면, spark.sql.shuffle.partitions 값을 사용(default = 200)

Code example)

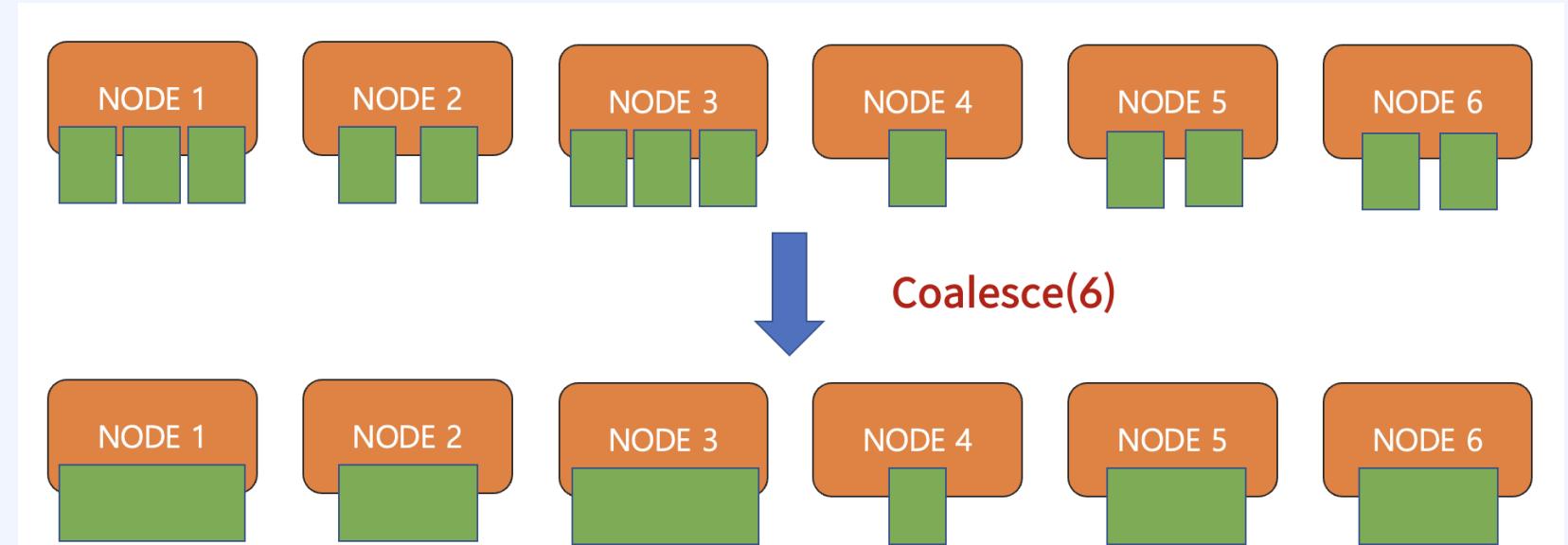
- repartition(10)
 - > 각 partition의 크기가 거의 균일함.
- repartition(10, "age")
 - > 특정 column을 지정하는 경우, 그 column의 값의 분포에 따라, 각 partition의 크기가 균일하지 않게 될 가능성이 높음.
- repartition(10, "age", "gender")
- repartition("age")

3. Repartition 특징 정리

- Repartition은 Shuffle을 동반하기 때문에, 함부로 사용하면 안됨.
 - Repartition 사용 전, 후 성능이 어떻게 변하는지 모니터링.
- 사용하면 좋은 경우.
 1. DataFrame reuse and repeated column filters.
 - 이 때는 filter로 사용되는 column에 대해 DF를 repartition 해놓으면 좋음.
 2. DataFrame/RDD 의 partition 이 skewed 되어 있을 때
 - size가 큰 일부 partition 때문에 전체 stage를 처리하는 시간이 증가.
 3. DataFrame/RDD의 partition 개수가 너무 작을 때(= 각 partition의 크기가 너무 클 때)
- Partition 의 개수를 줄이는 목적으로 사용하는 것은 일반적으로 좋지 않음.
(줄일 때는 Repartition이 아닌 Coalesce를 사용하는 것이 좋음)

4. Coalesce

- **Narrow dependency transformation -> (Shuffle X)**
- **RDD**
 1. coalesce(numPartitions)
- **DataFrame**
 1. coalesce(numPartitions)



5. Coalesce 특징

- Coalesce는 shuffle/sort가 발생하지 않기 때문에, partition size를 줄이는 데에는, 일반적으로 repartition 보다 좋음. 그래도 사용에 주의가 필요!
- local (같은 Executor) 내의 partition들을 결합하기만 함.
- Coalesce는 partition의 개수를 증가시키지 않음.
- Coalesce는 skewed partition을 생성할 수 있음.
- Narrow transformation이기 때문에, coalesce() 가 속한 stage 내에서, coalsece() 와 같이 수행되는 narrow transformation들의 병렬성을 안좋게 할 수 있음. -> ?

5.1 Coalesce 특징

- Narrow transformation이기 때문에, coalesce() 가 속한 stage 내에서, coalesce() 이전에 수행되는 narrow transformation들의 병렬성을 안좋게 할 수 있음. -> ?
- ex) coalesce를 하지 않았다면, where, select transformation이 20개의 task에서 수행됐을 것.

```
df = df.repartition(20) \task : 5개 (20개 X)
    .where("Pclass = 1") \
    .select("PassengerId", "Survived") \
    .coalesce(5) \
    .where("Age >= 20")\
```

Completed Stages (5)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
7	count at NativeMethodAccessorImpl.java:0	+details 2023/03/26 17:18:18	24 ms	1/1			295.0 B	
4	count at NativeMethodAccessorImpl.java:0	+details 2023/03/26 17:18:18	0.2 s	5/5			1060.0 B	295.0 B
2	count at NativeMethodAccessorImpl.java:0	+details 2023/03/26 17:18:18	0.3 s	1/1	59.8 KiB			1060.0 B
1	csv at NativeMethodAccessorImpl.java:0	+details 2023/03/26 17:18:17	0.2 s	1/1	59.8 KiB			

9. Caching, Persistence에 대한 이해

1. 실습 코드 링크

- RDD example)
 - https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/cache_persistence_rdd_ex.py
- SparkSQL example)
 - https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/cache_persistence_sql_ex.py

2. Caching, Persistence의 목적

- 동일한 RDD나, DataFrame의 transformation 연산을 중복으로 하지 않게 하기 위함.
 - transformation 연산이 lazy evaluation, 즉 action이 실행될 때 실제로 transformation 연산이 수행된다는 것을 인지해야 함.

```
df = ss.range(1000000) \
    .toDF("id") \
    .withColumn("square", col("id") * col("id"))

df.count()
df.count()
```

- 왼쪽 코드에서,
df를 생성하는 transformation은
1번만 호출 ? -> X (2번 호출됨)

3. Caching, Persistence의 차이

- cache()는 내부적으로 Persistence를 호출.
 - cache() = persist(StorageLevel.MEMORY_ONLY)
 - 메모리에만 데이터를 캐싱
- persist()는 다양한 StorageLevel에 데이터를 캐싱할 수 있음.
<https://sparkbyexamples.com/spark/spark-persistence-storage-levels/>
 - ex)
 - MEMORY_ONLY
 - DISK_ONLY (MEMORY_ONLY 대비 더 큰 사이즈의 데이터 캐싱 가능, 대신 접근 속도 느림)
 - MEMORY_ONLY_2 (2: 캐싱을 두 번 함. -> 캐싱 시간이 더 오래 걸리나, 더 안전함.)
 - OFF_HEAP
 - etc..

```
def cache(self: "RDD[T]") -> "RDD[T]":  
    """  
    Persist this RDD with the default storage level ('MEMORY_ONLY').  
    """  
  
    self.is_cached = True  
    self.persist(StorageLevel.MEMORY_ONLY)  
    return self
```

4. Caching 유무 비교

cache_persistence_ex application UI

Spark Jobs (3)

User: eomhyeonho
Total Uptime: 53 s
Scheduling Mode: FIFO
Completed Jobs: 3

Event Timeline

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_pers... count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_ex.py:20	2023/03/26 14:24:58	7 s	1/1	1/1
1	count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_pers... count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_ex.py:19	2023/03/26 14:24:51	7 s	1/1	1/1
0	count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_pers... count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_ex.py:18	2023/03/26 14:24:42	8 s	1/1	1/1

Page: 1 1 Pages. Jump to 1 Show 100 items in a page. Go

캐싱 X

cache_persistence_ex application UI

Spark Jobs (3)

User: eomhyeonho
Total Uptime: 25 s
Scheduling Mode: FIFO
Completed Jobs: 3

Event Timeline

Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_pers... count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_rdd_ex.py:23	2023/03/26 14:30:35	3 s	1/1	1/1
1	count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_pers... count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_rdd_ex.py:22	2023/03/26 14:30:32	3 s	1/1	1/1
0	count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_pers... count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_rdd_ex.py:21	2023/03/26 14:30:18	14 s	1/1	1/1

Page: 1 1 Pages. Jump to 1 Show 100 items in a page. Go

캐싱 O

- 캐싱을 하는 데에도 리소스가 필요하기 때문에, Job0의 경우 더 오래 걸림. (8s vs 14s)
- 그러나 Job1, Job2를 비교해보면, 캐싱 적용 시 더 빨라지는 것을 확인 가능. (7s vs 3s)

5. Caching 적용시 Spark UI (Stages)

APACHE  3.3.1 Jobs Stages Storage Environment Executors cache_persistence_ex application

Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 14 s
Locality Level Summary: Process local: 1
Associated Job Ids: 0

▼ DAG Visualization

Stage 0

```
graph TD; A[ParallelCollectionRDD [0] readRDDFromFile at PythonRDD.scala:274] -- parallelize --> B[PythonRDD [1] [Cached] RDD at PythonRDD.scala:53]; B --> C[PythonRDD [2] count at /Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/cache_persistence_rdd_ex.py:21]
```

Show Additional Metrics
Event Timeline

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	14 s	14 s	14 s	14 s	14 s
GC Time	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s

- 캐싱된 RDD, Dataframe의 경우 UI에서 초록색으로 표시됨

5.1 Caching 적용시 Spark UI (Storage)

cache(), persist(StorageLevel.MEMORY_ONLY)

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
1	PythonRDD	Memory Serialized 1x Replicated	1	100%	190.1 MiB	0.0 B

cache(), persist(StorageLevel.DISK_ONLY)

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
1	PythonRDD	Disk Serialized 1x Replicated	1	100%	0.0 B	190.1 MiB

cache(), persist(StorageLevel.MEMORY_ONLY_2)

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
1	PythonRDD	Memory Serialized 2x Replicated	1	100%	190.1 MiB	0.0 B

6. Caching, Persistence를 사용하면 좋은 경우 / 안되는 경우

- 사용하면 좋은 경우
 - 큰 RDD, DataFrame에 반복적으로 접근해야 되는 경우.
 - ex) 반복적인 머신러닝 학습, ETL 파이프라인에서 빈도 높은 transformation 연산으로 자주 접근해야 하는 데이터.
- 사용하면 안되는 경우
 - DataFrame이 메모리에 들어가기는 너무 큰 경우.
 - 크기에 상관 없이, 자주 쓰지 않는 DataFrame에 대해 비용이 크지 않은 transformation 수행.
- cache() (= persist(StorageLevel.MEMORY_ONLY))의 경우,
직렬화(Serialization), 비직렬화(Deserialization) 과정이 동반되기 때문에 주의.

10. Shared Variable (Accumulator, Broadcast variable)

1. 실습 코드 링크

- Accumulator example)
 - https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/accumulator_ex.py
- Broadcast variable example)
 - https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/broadcast_variable_ex.py

2. Accumulator

- 더하기 (add) 연산만 가능한 특별한 변수.
 - 여러 task에서 병렬로 접근할 수 있어 효율적.
- counter, sum 등을 구하는데 사용.
- (Scala 기준) LongAccumulator, DoubleAccumulator를 기본으로 제공하고, **AccumulatorV2** 클래스를 상속해서 custom한 Accumulator를 정의하는 것도 가능.
- Low level api.
- details : <https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#accumulators>

2.1. Accumulator - use case

- Ex) 유효하지 않은 포맷인 row count 계산
 - accumulator를 사용하지 않고, filter(), count()를 이용해 쉽게 구할 수 있지만, count()는 action 연산이기 때문에, Job 하나가 추가되고, 불필요한 Shuffle이 발생.
 - 이 때 accumulator를 사용한다면, action을 추가하지 않고도, count를 구할 수 있음.

2.2. Accumulator - 주의할 점.

- Accumulator를 Transformation 안에 둘 수도 있고, action 안에 둘 수도 있음.
- In transformation.
 - ex) select()에서 사용.
- In action.
 - ex) accumulator를 forEach()에서 사용.
- **transformation에서 Accumulator를 사용해서 계산한 값은 정확하다는 보장이 없음.**
 - ex) 특정 task가 실패해서 retry될 수도 있는데, 이런 경우
Accumulator에 더해진 값은 보정되지 않음. 즉 두 번 이상 더해질 수도 있음.
-> 그러므로 정확한 값을 얻으려면 Accumulator를 Action에 넣어야 함.
(100% 정확한 값이 필요 없다, 예컨대 값의 트렌드 정도만 보면 된다 하면 transformation에
추가해도 괜찮음)

3. Broadcast variable

- Low level API.
- Shared, immutable 변수.
- Executor 당 1번만 직렬화(Serialized).
- 여러 Stage에서 task들에서 동일한 데이터가 필요하거나,
역직렬화된 형태로 데이터를 캐싱하는 것이 중요한 경우 활용 가능.
 - Broadcast variable을 잘 활용하면, 불필요한 Shuffle을 줄일 수 있음.
- 모든 Executor에 복사되는 값이기 때문에, 데이터 사이즈가 크지 않아야 함!

(Spark의 broadcast join을 사용한다면, 대부분의 경우에서
broadcast variable을 굳이 사용할 필요는 없음..)

- details : <https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#broadcast-variables>

3.1 Broadcast variable - Use case

- Ex) Product DataFrame에 카테고리 이름(category_name) 컬럼 추가.
 - category 정보를 담고 있는 dictionary의 데이터 사이즈가 크지 않기 때문에, Broadcast variable로 사용할 수 있음.

11. 스파크 Query Plan

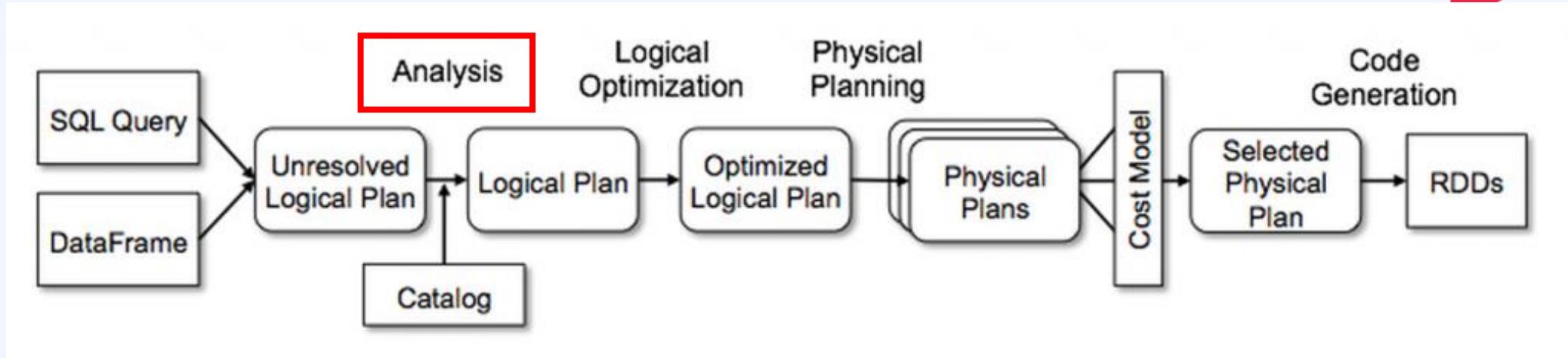
Query Plan

- SparkSQL
- Dataframe API
- Dataset API
 - Scala, Java에서만 사용 가능.

로 작성된 코드는 Query Plan이 생성되면서 결국 RDD로 컴파일 됨.

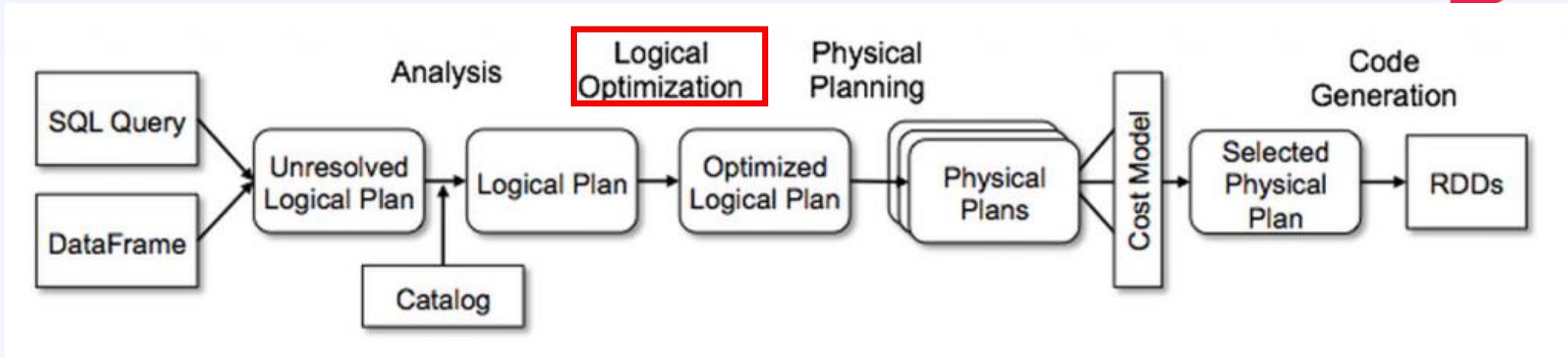
- 각각의 Spark Job (Action에 의해 나뉘는) 별로 logical query plan이 생성.

1. Query Plan - 1) Analysis



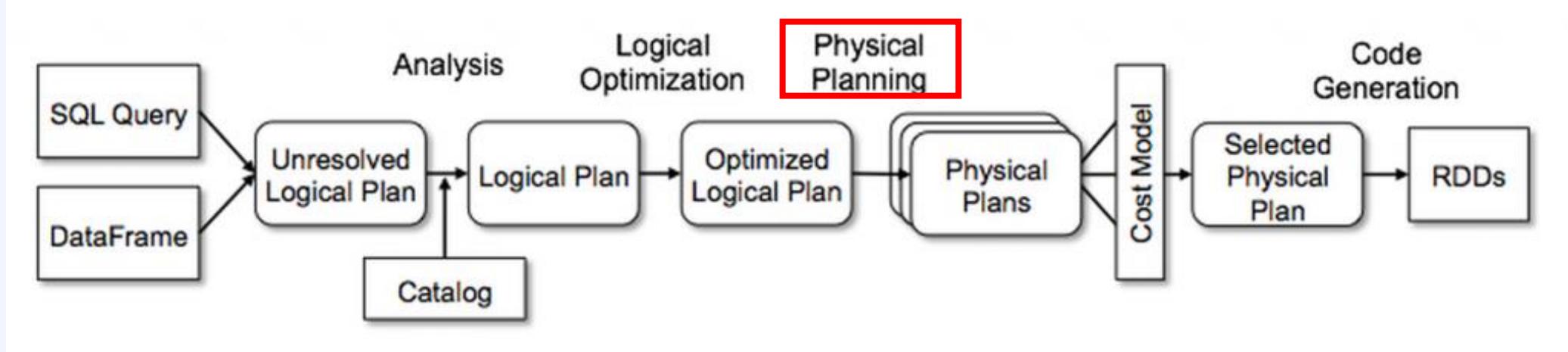
- 코드를 파싱해서 에러가 존재하는지, 잘못된 컬럼 명 등이 있는지 검증 후 Logical Plan 생성
- ex) SparkSQL (`SELECT product_id, item_id FROM items_table...`)
→ `product_id, item_id` 컬럼이 존재하는지 어떤 데이터 타입인지, application이 시작되기 전에는 알 수 없음.
- SparkSQL 엔진은 Analysis 단계에서 Catalog를 보고 컬럼 이름과 데이터 타입 등을 확인.
- 잘못된 부분이 존재할 시 에러 반환.

1. Query Plan - 2) Logical Optimization



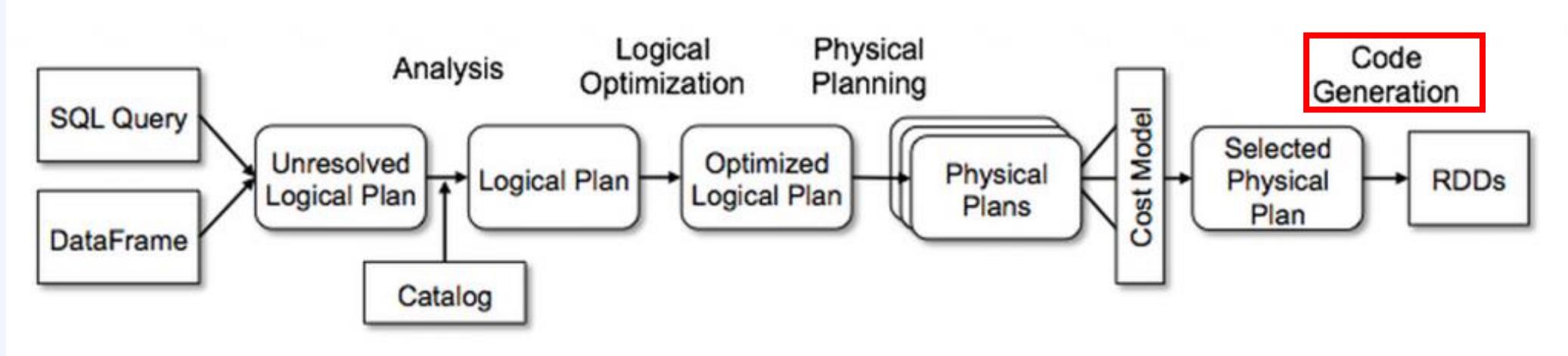
- Logical Plan에 몇 가지 rule-based 최적화 기법들을 적용 후 Optimized Logical Plan을 생성
ex)
 - Constant folding (상수 표현식을 runtime이 아닌 compile time에 미리 계산)
 - Predicate Pushdown (Filter 연산을 최대한 앞 단계에 적용)
 - Partition pruning (partitioning된 데이터를 읽을 때, 전체 데이터가 아닌 필요한 데이터만 읽음)
 - Null propagation
 - Boolean expression simplification etc..

1. Query Plan - 3) Physical Planning



- Optimized Logical Plan에 cost-based 최적화 기법을 적용
- spark engine은 여러 개의 Plan을 생성하고 각각의 cost를 계산 후, cost가 가장 적게 들었던 Plan을 선택.
- ex) broadcast hash join, sort merge join, shuffle hash join을 적용해본 후 가장 cost가 적게 든 join 계획을 선택.

1. Query Plan - 4) Code Generation



- 선택된 Physical Plan을 바탕으로 RDD를 새로 생성.
- Spark engine은 마치 compiler 처럼 동작

Dynamic Resource Allocation

1. Spark Resource Allocation

- Spark는 크게 두 가지 Resource Allocation 방식을 가지고 있음.

1. Static Allocation

- Driver가 RM으로부터 최대한 많은 가용 리소스를 요청.
- 그 후 Spark Application은 Spark 프로그램이 실행되는 동안 받은 리소스를 고정으로 계속 가지고 있음.
- Application이 그 자원을 다 이용하고 있지 않은 상태라도, 계속 가지고 있다가, Spark 프로그램이 완료된 후에야 cluster에 리소스 반환.
- FIFO와 유사. resource 낭비가 발생할 수 있음.

2. Dynamic Allocation

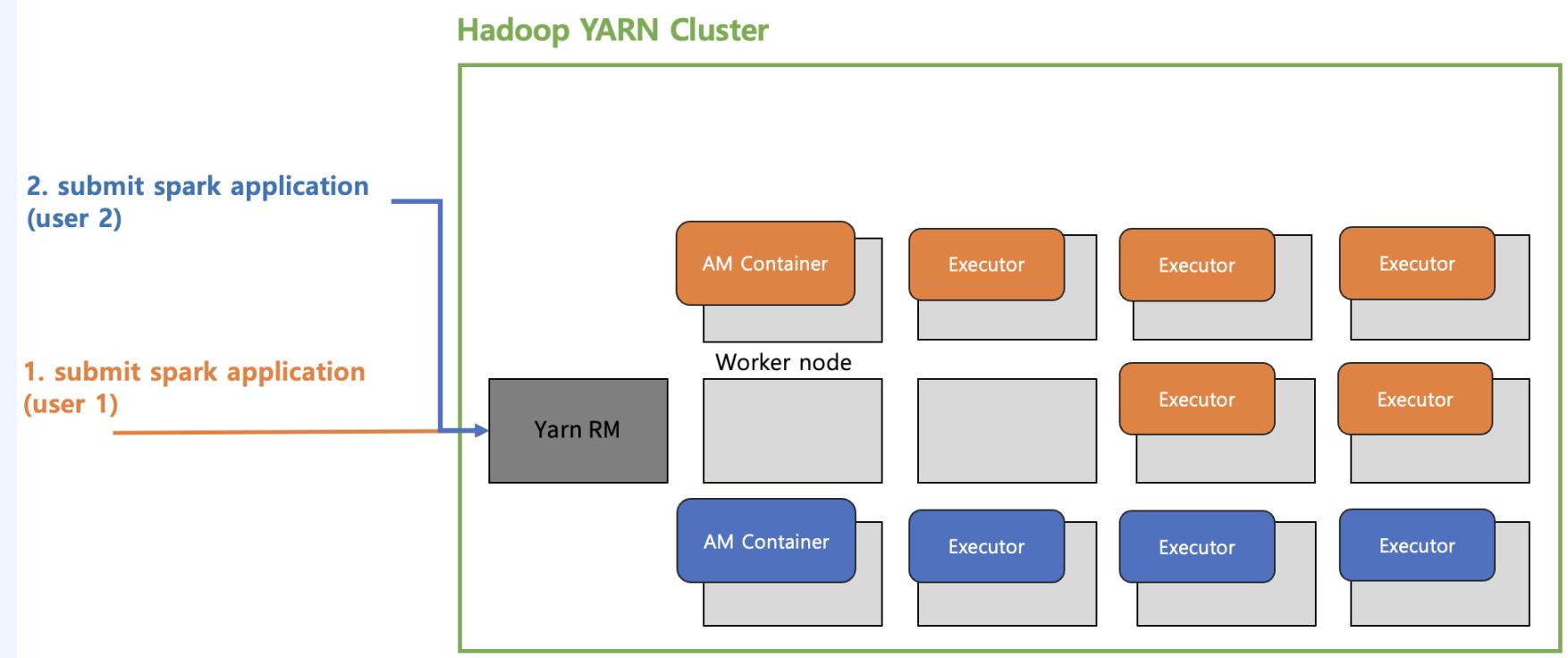
- Static Allocation과 달리, Application에서 Spark를 실행하는 도중 남는 자원이 생기면, 자원 할당을 해제할 수 있음.
- resource 절약 가능!

1. Spark Resource Allocation

- 두 방식은, cluster의 Resource Allocation이 아닌, Spark의 Resource Allocation. (cluster와는 아무런 관련이 없음)
- application이 어떻게 RM에 리소스를 요청하고, 작업 후에 RM에 리소스를 반환하는 방법의 차이

2. Scenario - Application 간 Scheduling.

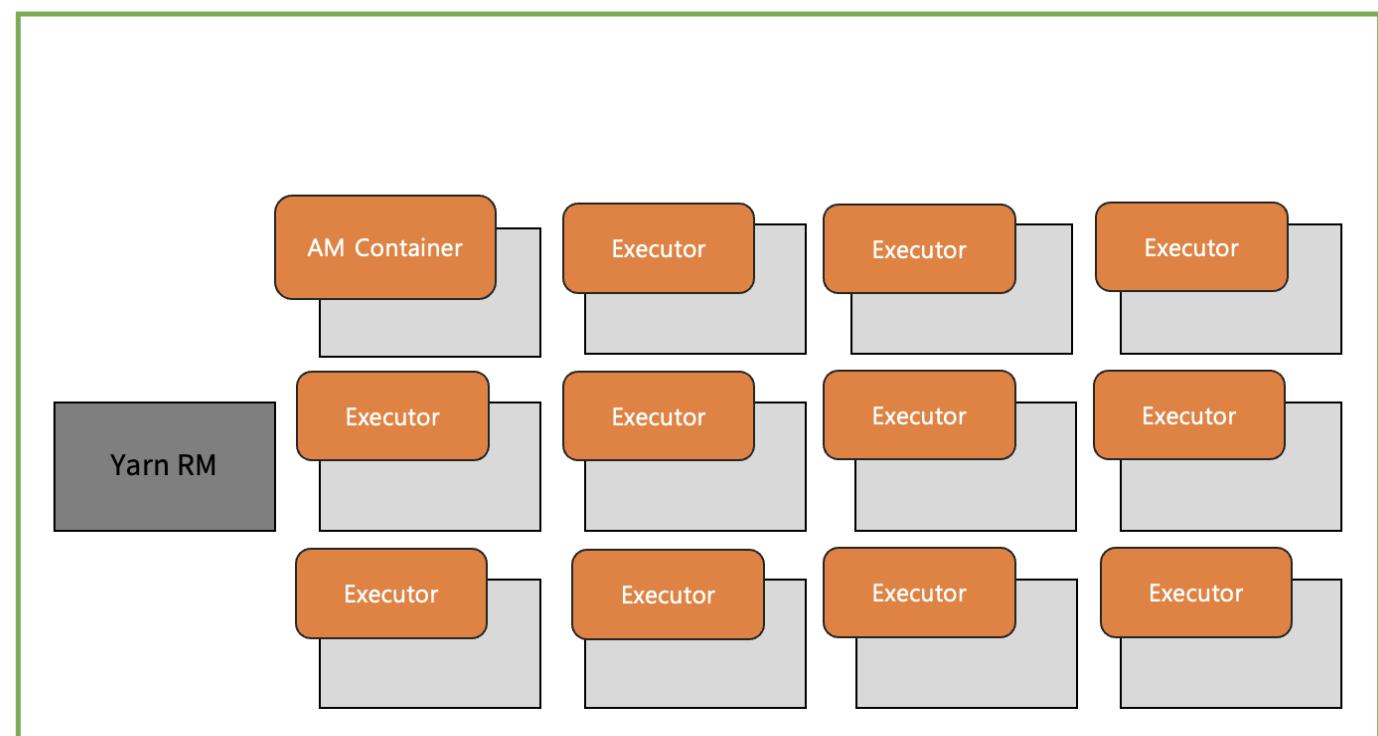
- YARN cluster에서, 1개 이상의 spark app이 실행될 수 있음



2. Scenario - Application 간 Scheduling.

- 첫 번째 Spark application이 cluster의 모든 Executor를 사용하고 있는 상황에서,
- 두 번째 Spark application을 Yarn RM에 제출하면 어떻게 될지?
(두 번째 Spark application은 3개의 Worker node를 사용해야 한다고 가정)

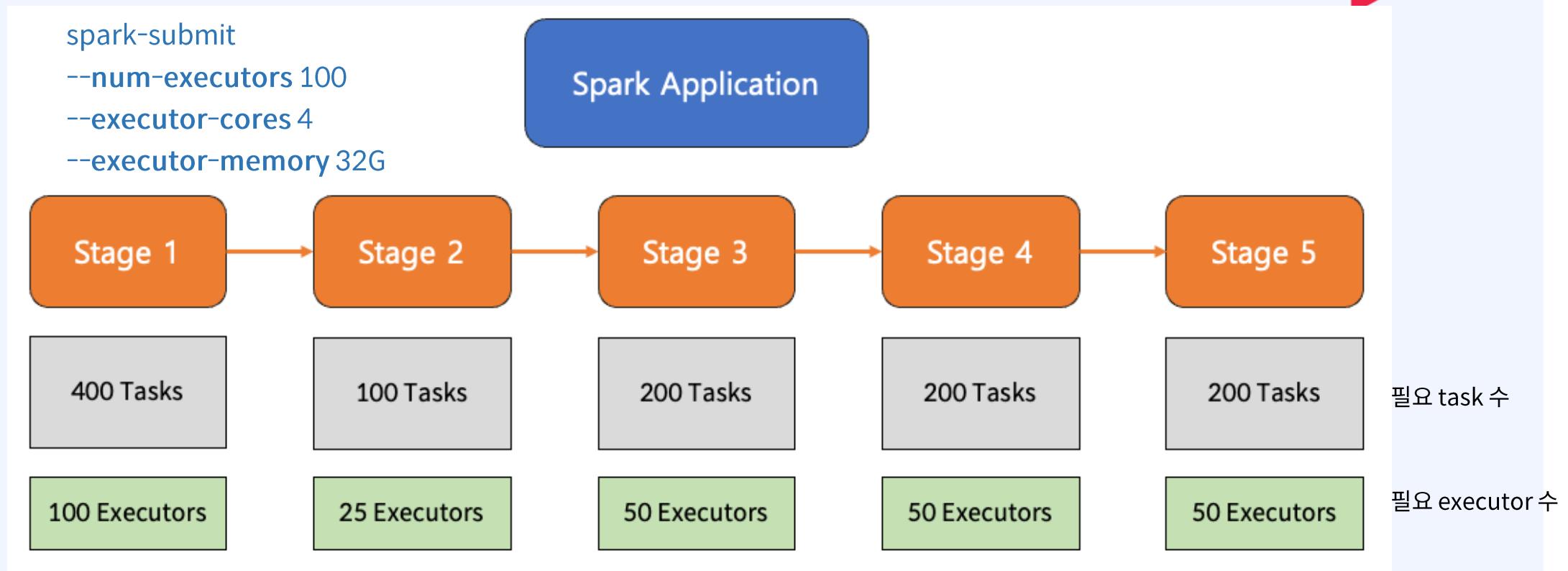
Hadoop YARN Cluster



Total cluster spec : (48 CPU, 384GB RAM) (12 executors)
1 executor spec: 4 CPU, 32 GB RAM

2. Scenario - Application 간 Scheduling (Static Allocation)

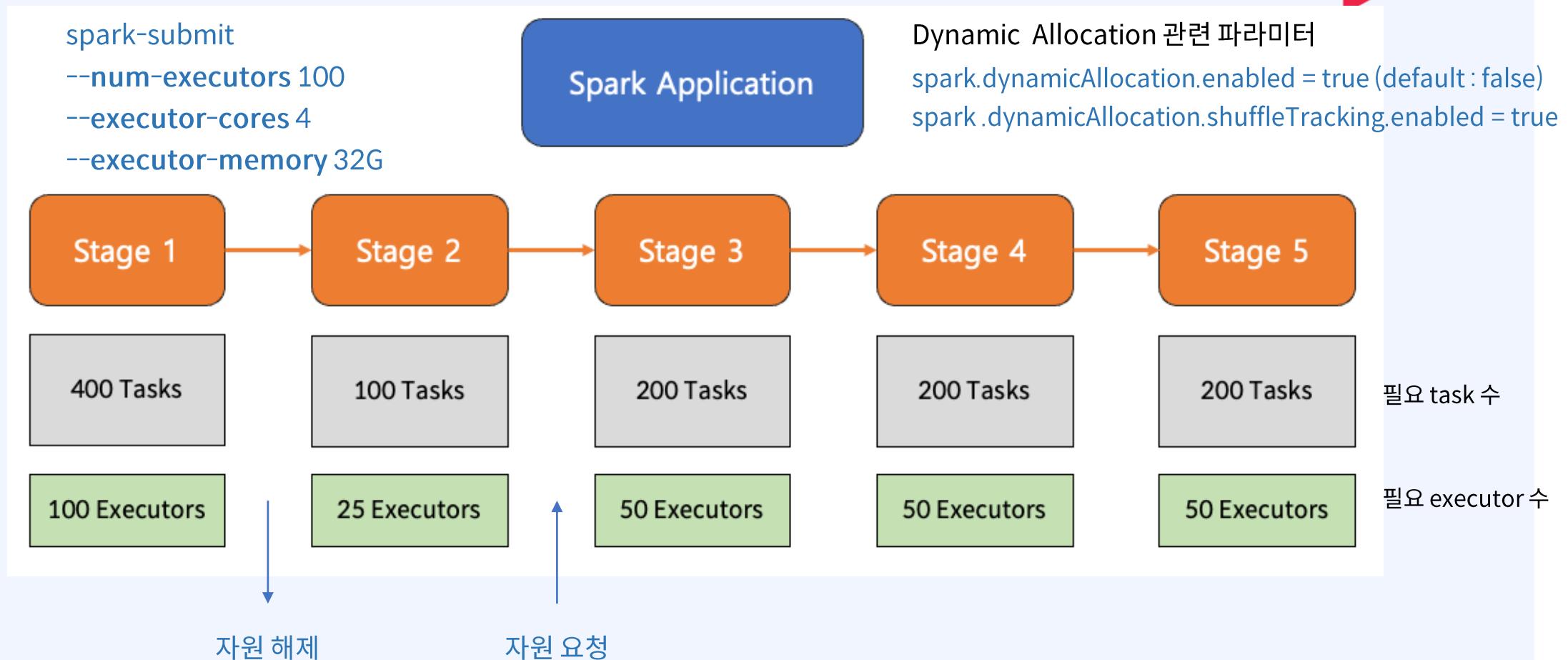
- cluster에서 1개의 application만 구동될 때는 Static Allocation 사용



- Stage 1에서만 cluster의 전체 Executor를 사용하고, 나머지 Stage들에서는 다 사용하지 못하지만, 한 번 할당된 자원을 Cluster에 다시 반납할 수 없음.

2. Scenario - Application 간 Scheduling (Dynamic Allocation)

- cluster 내에 여러 spark application이 구동될 때는 Dynamic Allocation 사용 권장.



3. Dynamic Allocation 관련 파라미터 정리

1. `spark.dynamicAllocation.enabled = true`
2. `spark.dynamicAllocation.shuffleTracking.enabled = true`
 - executor의 shuffle file의 상태를 추적해, dynamic allocation 시 external shuffle service 를 사용할 필요가 없도록 함.
3. `spark.dynamicAllocation.executorIdleTimeout = 60s (release time)`
 - 특정 executor가 idle 상태가 된지 60초가 지나면,
spark application은 executor 자원을 cluster manager에 돌려줌.
 - 60초 정도로 세팅하는 것을 권장.
4. `spark.dynamicAllocation.schedulerBacklogTimeout = 1s (request time)`
 - 1초 이상 pending되는 task가 있고, pending된 task를 할당할 수 있는 free executor가 없을 때, spark application은 더 많은 executor를 요청.

Spark Job scheduler에 대한 이해

1. 실습 코드 링크

- https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/spark_scheduler_ex.py
- https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/spark_scheduler_multi_thread_ex.py

2. Spark Job Scheduler

- Dynamic Resource Allocation 강의에서는, Application 간의 Scheduling에 관해서 논의.
- 이번에는 Application 내에서 Job들이 Scheduling되는 방법에 대해 논의.

2. Spark Job Scheduler

- Spark application에서, 기본적으로 Job들은 순차적으로 수행됨.
- Job1, Job2가 있다고 했을 때, 만약 Job2의 input 데이터가 Job1의 output 아웃풋 데이터에 의존하지 않는다면 꼭 순차적으로 수행되지 않아도 됨.

3. Scheduler의 종류

- FIFO
 - 항상 첫 번째로 들어온 job부터 처리.
- FAIR
 - spark.scheduler.mode=FAIR
 - Round robin 기반의 할당 방식.

(Spark 3.0 +) AQE - Adaptive Query Execution에 대한 이해.

1. Spark AQE 란?

AQE(Adaptive Query Execution)

- Spark 3.0 + 에 도입.
- Spark 2.0 대에서는 런타임 시점에는 쿼리 플랜이 고정됨.
- **런타임 시점에, Shuffle 데이터에 대한 통계치를 저장, 테이블 사이즈를 추적하여, 쿼리 플랜을 동적으로 변경**
- 세 가지 Key Features
 1. Dynamically coalescing shuffle partitions
 2. Dynamically switching join strategies
 3. Dynamically optimizing skew joins

2. Dynamically coalescing shuffle partitions

적용 전)

- 개발자가 spark.shuffle.partitions 파라미터를 세팅하거나 repartition() 메서드 등을 통해, 파티션 수를 조정할 수는 있으나, shuffle 연산 수행 시 최적의 파티션 수가 몇일지 파악이 어려움.
 - if 파티션 수를 적게 세팅
 - 각 파티션의 사이즈는 증가.
 - 각 테스크는 많은 메모리 용량을 필요로 하며, OOM이 발생할 수 있음.
 - If 파티션 수를 크게 세팅
 - 각 파티션의 사이즈는 감소.
 - 불필요한 I/O, 네트워크 비용 증가.
 - Spark task scheduler 과부화

적용 후)

- **Shuffle 시 테이블의 통계 정보를 바탕으로** 최적의 파티션 수를 결정.
- 작은 파티션들을 결합하거나, 큰 파티션을 쪼갠다.

3. Dynamically switching join strategies

적용 전)

- 런타임에 Query Plan이 고정되기 때문에, Spark 2.x에서는 join 방법을 변경할 수 없음.
- Ex)
 - 실제 테이블 조회 후 join하는 시점에서, 테이블의 사이즈가 broadcast hash join을 사용할 수 있을 정도로 작다고 확인이 되어도,
Query Plan에서는 Sort Merge Join을 사용하도록 되어 있으므로,
Sort Merge Join을 사용한다.

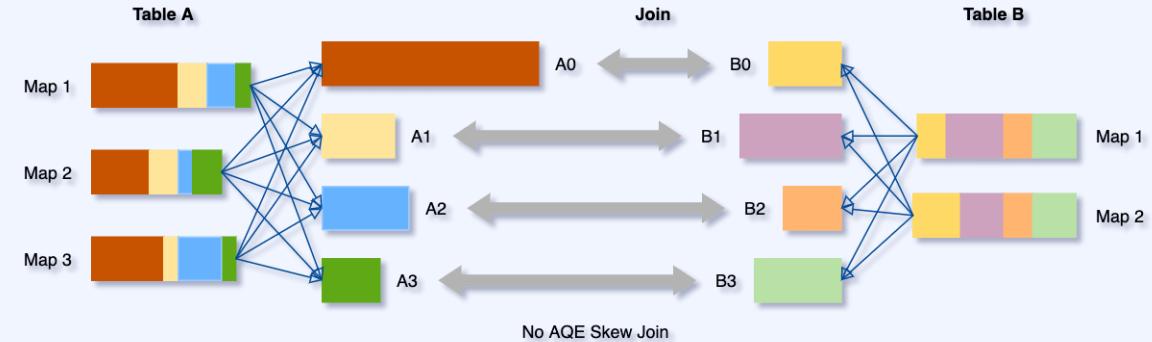
적용 후)

- **Shuffle 시 테이블의 통계 정보를 바탕으로**, 조인하려는 테이블의 사이즈가 작다면
Query Plan에서는 Sort Merge Join을 사용하도록 계획되어도, 동적으로 broadcast hash join을
사용할 수 있음.
- 다만, 처음 Query Plan에서부터 broadcast hash join을 사용하도록 계획된 것 보다는 성능이 좋지 않음.

4. Dynamically optimizing skew joins

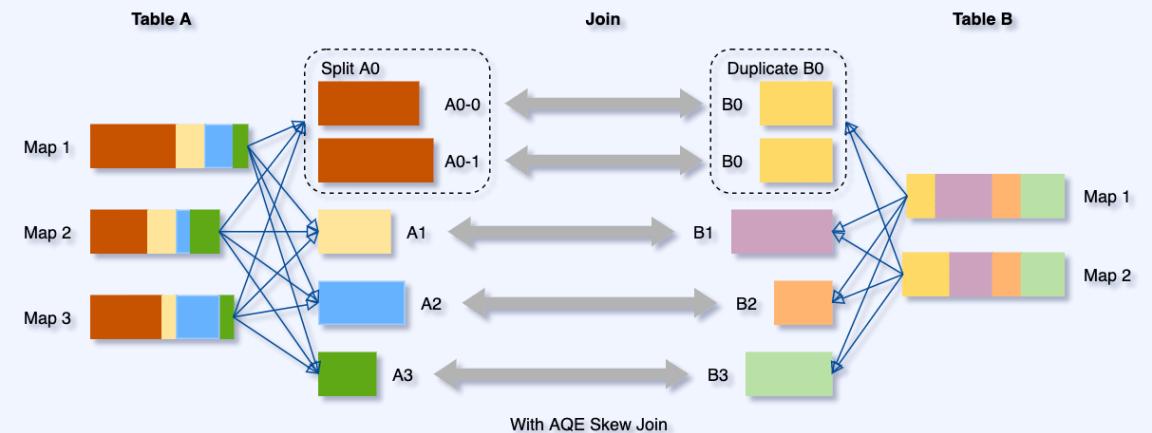
적용 전)

- Skewed join이 발생해 OOM 등의 에러가 발생하는 경우, 메모리 사이즈를 크게 세팅할 수 있음.
 - 특정 task에 대해서만 메모리 사이즈를 크게 할 수 없기 때문에, 낭비 발생.
 - 영구적인 해결책 X
(인풋 데이터가 달라지는 등, 외부 변화에 대응이 어려움)



적용 후)

- **Join 시점에서, 테이블의 통계 정보를 바탕으로** Skewed된 파티션의 경우, 2개 이상의 파티션으로 분리



(Spark 3.0 +) DPP - Dynamic Partition Pruning에 대한 이해.

15

1. 실습 코드 링크

- https://github.com/startFromBottom/fc-spark-practices/blob/main/5_dive_deep_spark/dpp_ex.py

2. Predicate Pushdown, Partition Pruning

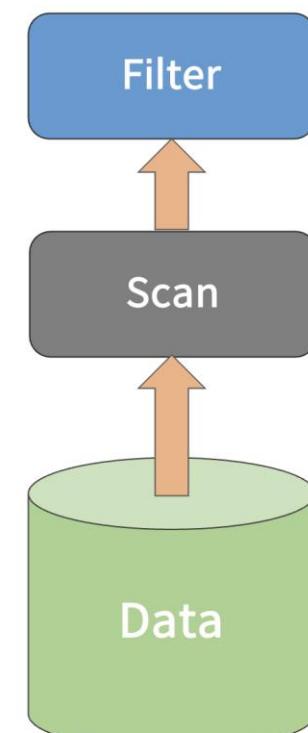
- Dynamic Partition Pruning을 배우기 전에,
Predicate Pushdown, Partition Pruning의 개념에 대해 확인.

1) Predicate Pushdown.

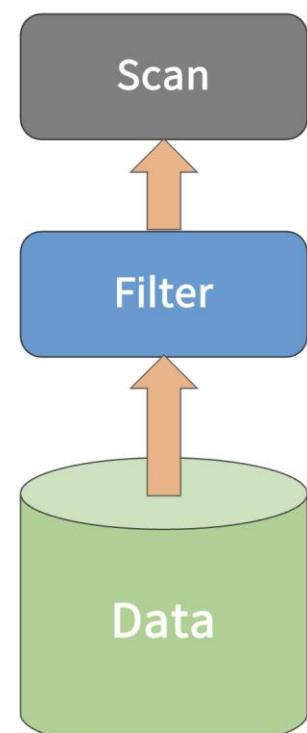
- Filter 연산을 최대한 데이터 소스에 가깝게 이동.
- 데이터를 읽어온 후 Filter를 하는 것이 아닌,
읽는 시점에부터 필요한 데이터만 읽기!
(전제 조건 : Filter 조건에 포함된 컬럼으로 파일이
Partitioning 되어 있어야 함.)

2) Partition Pruning.

- Partitioning되어 있는 데이터에서,
특정 Partition의 데이터만 읽어오는 방법.



Predicate Pushdown 전



Predicate Pushdown 후

2.1 Predicate Pushdown, Partition Pruning - ex)

테이블 생성 쿼리 -> 100개의 partitioning된 파일 생성

```
orders_ddl = """
CREATE TABLE Orders
USING parquet
PARTITIONED BY (uid)
AS
SELECT
    CAST((rand()*100) AS INT) AS uid,
    CAST((rand()*100) AS INT) AS quantity
FROM RANGE (1000000);
"""

ss.sql(orders_ddl)
```

데이터 조회 쿼리

```
pp_dml = """
SELECT uid, quantity * 2 AS double_quantity
FROM Orders
WHERE uid >= 5 AND uid <= 7
"""

ss.sql(pp_dml).count()
```

-> spark UI에서, 3개의 파일만 읽음.

```
Scan parquet default.orders
number of files read: 12
scan time total (min, med, max )
779 ms (194 ms, 195 ms, 195 ms )
metadata time: 32 ms
size of files read: 36.4 KiB
number of output rows: 29,889
number of partitions read: 3
```

2.1 Predicate Pushdown, Partition Pruning - ex)

- 실행 계획에서 PartitionFilters 확인 가능!

(1) Scan parquet default.orders

Output [1]: [uid#12]

Batched: true

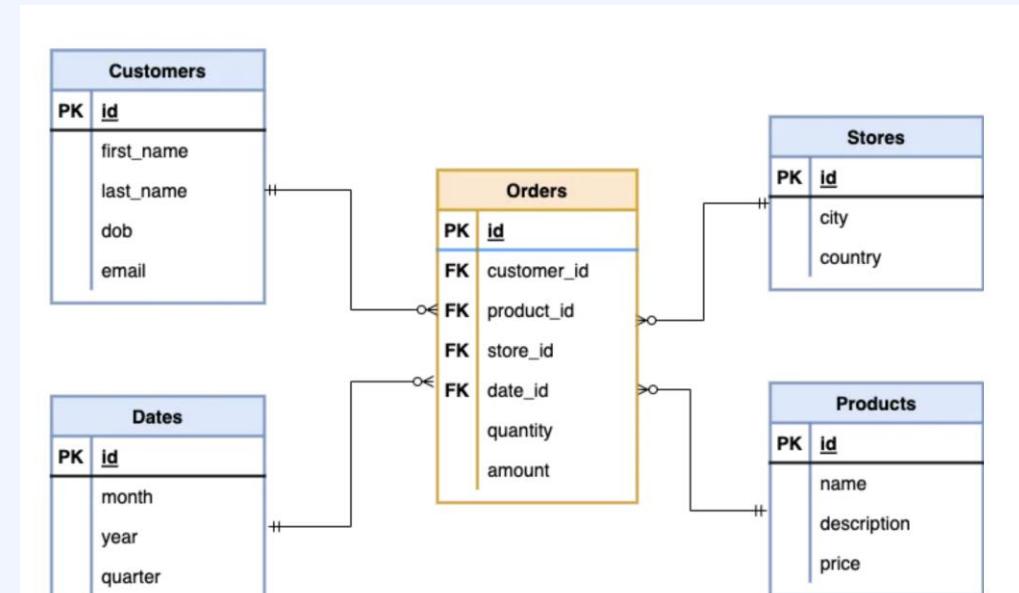
Location: InMemoryFileIndex [file:/Users/eomhyeonho/Workspace/spark-practices/]

PartitionFilters: [isnotnull(uid#12), (uid#12 >= 5), (uid#12 <= 7)]

ReadSchema: struct<>

3. Dynamic Partition Pruning

- A(큰 테이블), B(작은 테이블) 두 테이블을 조인할 때, B를 broadcasting 하고, B에 걸려 있는 Filter 조건을 이용해, A에서 Predicate Pushdown을 적용할 수 있게 하는 기법
- Dimension table, Fact table use-case에서 많이 사용될 수 있음.
 - Dimension table
 - Fact table들의 row에 포함된 모든 관련 필드에 대한 설명 정보가 포함.
 - ex 주문 정보(Orders)
 - Fact table
 - Dimension table의 pk 하나를 가지고 있고, 그에 대한 실제 정보를 담고 있는 테이블.
ex. 상품, 고객 등)



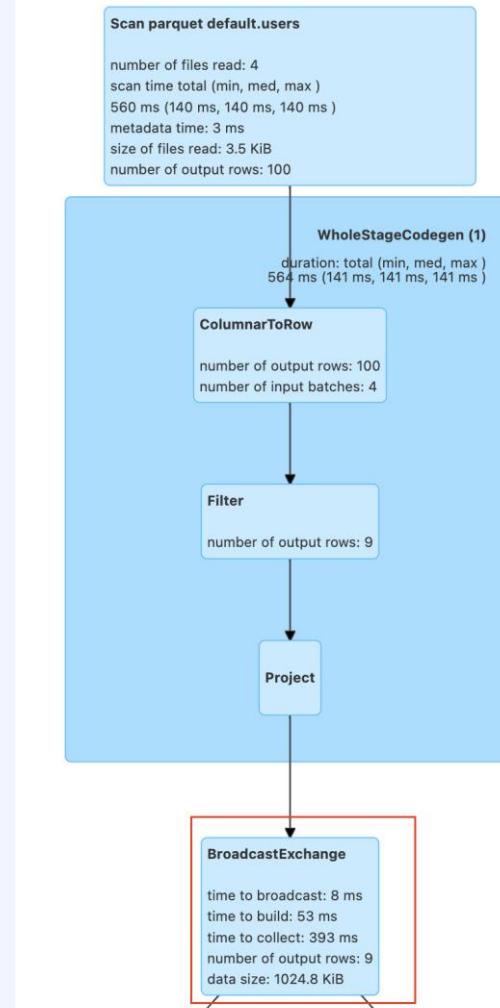
Example Star Schema with a Fact table in orange and Dimension tables in blue — Source: Author

3.1 Dynamic Partition Pruning - ex)

- 데이터 조회 쿼리
(Users(Fact), Orders(Dimension)) 테이블 생성 쿼리-> 코드에서 확인 가능)

```
dpp_dml = """  
SELECT u.uid, o.quantity  
FROM Users AS u  
JOIN Orders AS o  
ON u.uid = o.uid  
WHERE u.grade = 7  
"""
```

- Users는 자동으로 broadcasting 됨.
(default spark.sql.autoBroadcastJoinThreshold=10MB)



3.1 Dynamic Partition Pruning - ex)

Scan parquet default.orders

```
number of files read: 36
scan time total (min, med, max )
210 ms (50 ms, 54 ms, 54 ms )
dynamic partition pruning time: 22 ms
metadata time: 3 ms
size of files read: 109.5 KiB
number of output rows: 89,829
static number of files read: 400
static size of files read: 1218.1 KiB
number of partitions read: 9
```

- Users의 컬럼으로 Filter 조건을 걸었지만, Orders 테이블을 불러올 때도, 일부 파티션(100개 중 9개)만 읽는 것을 확인 가능.
- Orders 테이블에도 PartitionFilter가 적용된 것을 확인 가능.

```
(7) Scan parquet default.orders
Output [1]: [uid#12]
Batched: true
Location: InMemoryFileIndex [file:/Users/eomhyeonho/Workspace/spark-practices/5_dive_deep_spark/spark-warehouse/orders,
PartitionFilters: [isnotnull(uid#12), dynamicpruningexpression(cast(uid#12 as bigint) IN dynamicpruning#33)]
ReadSchema: struct<
```

3.2 Dynamic Partition Pruning. - 주의할 점

1. 큰 사이즈의 테이블(broadcasting 되지 않은)은 반드시 Partitioning이 되어 있어야 함.
2. 작은 사이즈의 테이블은 broadcasting 될 수 있어야 함.

Partitioning

1. Partitioning 개요, 중요성

1. Partition ?

- 스파크에서 Partition은, 데이터를 논리적인 연산 단위인 chunk로 나눈 것을 의미.
- chunk로 나눌 때, 데이터의 분포는 완전히 랜덤일 수도 있고, 특정한 규칙이 있을 수 있음.
- 1개의 Partition에 대한 연산은, 기본적으로 1개의 task에서 수행함.
- 즉 Partition과 task는 1:1의 관계이기 때문에, Partition을 스파크 프레임워크의 병렬 처리의 단위라고 볼 수 있음.

1.1 Partition 개수 확인 실습.

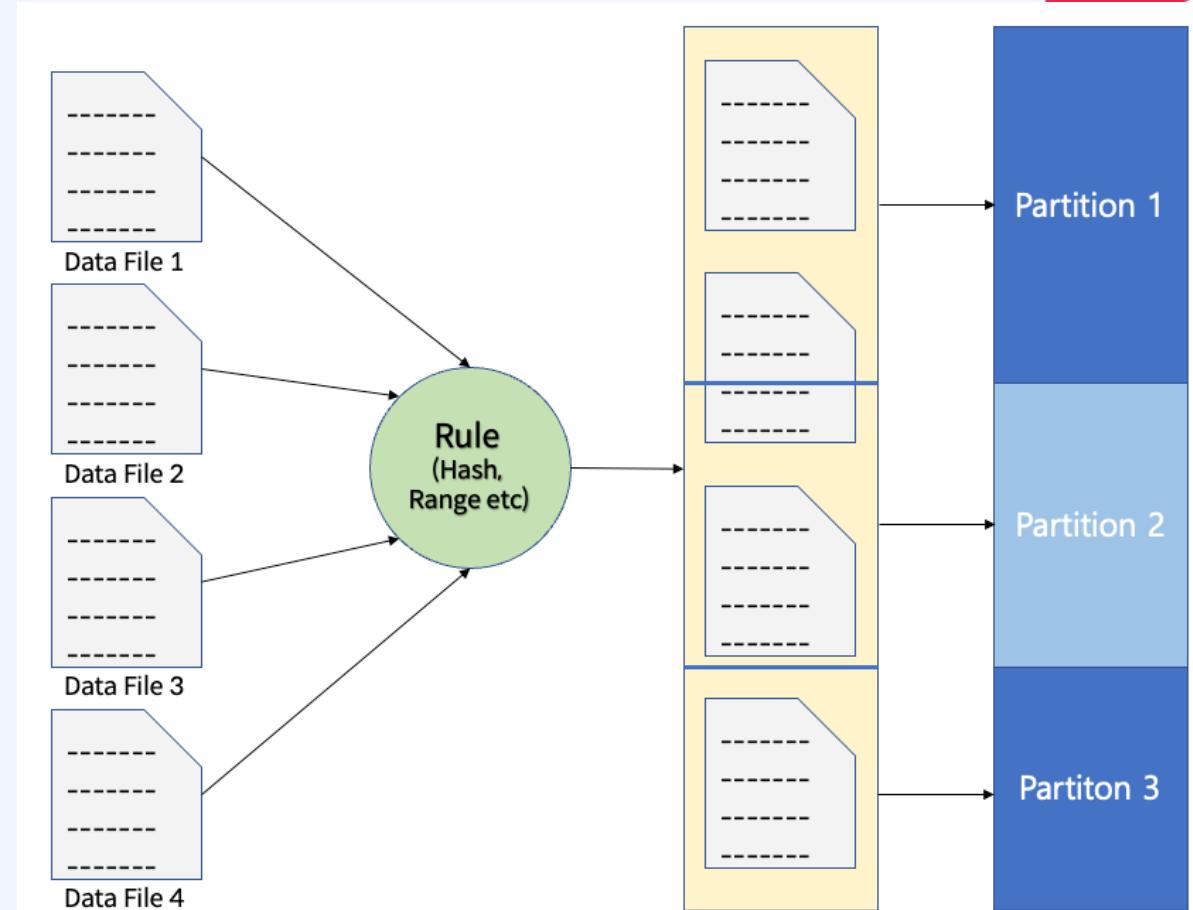
- https://github.com/startFromBottom/fc-spark-practices/blob/main/6_partitioning/check_partition_count_ex.py

2. Partitioning.

- Partition을 만드는 방법.
- 이 방법에는 아래의 두 가지가 고려되어야 함.
 1. Partition의 개수
 2. Partition의 개수가 주어졌을 때 데이터가 어떻게 분배되는지.
- Spark에서 기본으로 제공하는 Partitioning 방법으로는 크게 Hash Partitioning과 Range Partitioning이 존재.
 1. Hash Partitioning
 - 데이터의 각 row를 대표하는 key (ex: Scala, Java의 PairRDD에서 key, DF의 특정 컬럼 등) 를 hash function에 넣어 나온 hash code를 기반으로 Partitioning.
 2. Range Partitioning.
 - 각 partition마다 특정 범위를 가지고 있고, 그 범위는 정렬가능한 key값들을 데이터에서 일부 샘플링하여 결정. (항상 같은 값들로 샘플링되지 않으므로, 항상 같은 범위 임을 보장 x)
 - 범위가 정해지면, 나머지 값들을 그 범위에 맞게 Partitioning.

2.1 Partitioning in RDD

- 다양한 input data source가 존재할 수 있음.
- ex)
 - parquet file
 - HDFS
 - RDBMS, Hive table 등.
- 미리 정의된 rule에 따라 이 input data가 RDD의 Partition 으로 변환됨.
- 오른쪽 예시에서 input data : Data File
-> (Data File과 Partition이 꼭 1:1로 대응 하는 것은 아님.)
- Rule : Partitioner



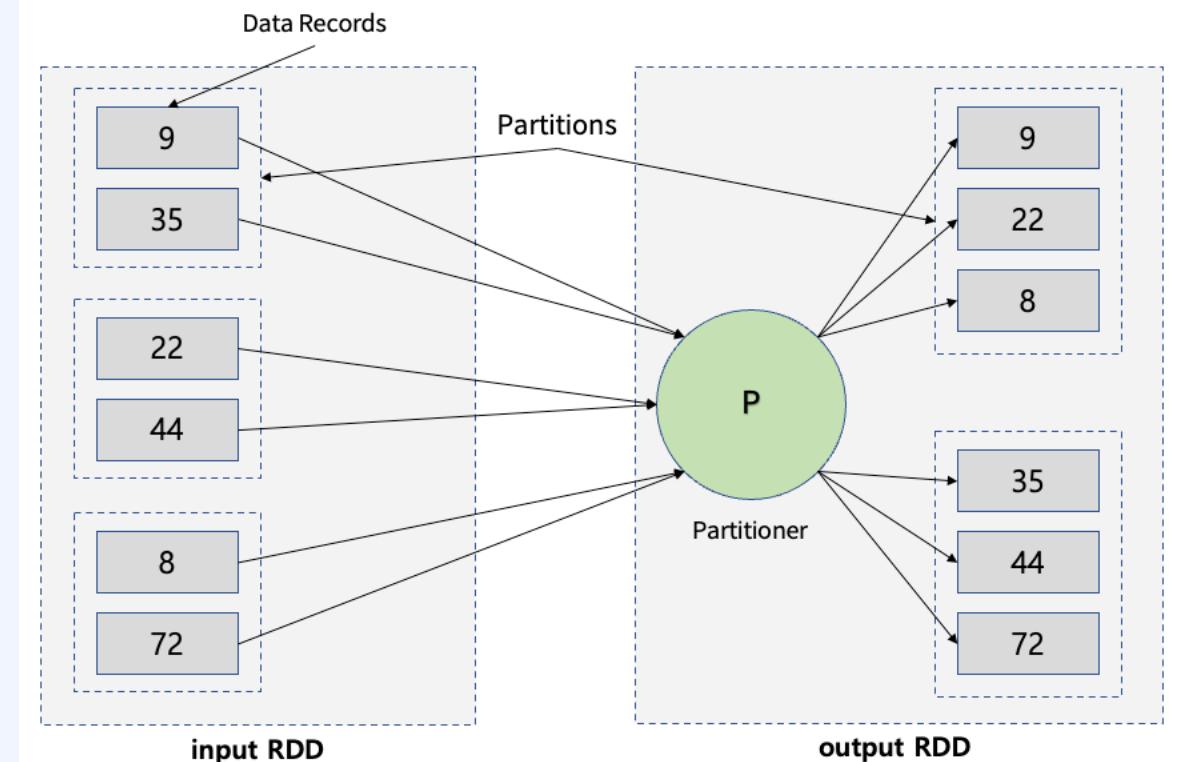
input data file -> partition

2.1 Partitioning in RDD

- RDD의 경우,
Spark에서 기본적으로 제공하는 HashPartitioner, RangePartitioner 외에,
데이터의 특성에 맞는 더 Custom한 Partitioner를 직접 구현 가능.
 - Scala, Java의 경우 abstract class인 Partitioner를 상속 받아 구현하면 되지만,
Pyspark에서는 지원 X
- RDD 탑입의 변수 A가 있다고 하면,
A.partition() (Scala), A.partition (Pyspark) 이런 식으로, A의 Partitioner 정보를
가져올 수 있음.
- RDD A와 RDD B가 같은 partitioner를 사용해 partitioning 되어있다면,
join, groupby 등의 연산에서 wide transformation이 아닌 narrow transformation이
될 수 있음(= Shuffle 이 발생하지 않음.)
 - output RDD도 같은 partitioner 정보를 갖게 됨.

2.1 Partitioning in RDD - 주의!

- Q) partitioner 정보가 None이 아닌 input RDD에 어떤 transformation 수행 시, partitioner 정보는 항상 보존되는가?
 - 보존되지 않는 한 가지 예시는 PairRDD의 mapToPair 연산.
 - mapToPair 연산 시 key 값 자체가 바뀔 수 있는 가능성이 존재.
 - 1개의 key라도 변경되게 되면, Partitioner가 보장했던 데이터 분포의 일관성을 깨뜨림.
 - Pyspark의 map 함수도 비슷한 예
-> 이런 transformation 연산들에서는 output RDD의 partitioner = None
(실습 코드 확인)



RDD 간 partitioning 수행

2.2 Partitioning in DataFrame.

- RDD와 대부분 비슷하지만, 차이점이 존재.
1. DataFrame, Dataset(scala, java)에서는 custom partitioner를 생성할 수 없음.
대신 Partitioning에 사용할 column들을 지정해줄 수는 있음.

3. Partitioning의 중요성 - 1) 병렬 처리 성능

- Partitioning을 어떻게 했는지에 따라, 동일한 input 데이터에 대해서도 stage별 lead time이 크게 달라짐.
(성능)
 - Partition: Spark의 병렬성의 단위
 - Partitioning을 어떻게 했는지에 따라 병렬 처리의 능률이 달라짐!

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
13	groupBy c	2023/04/01 14:00:06	56 min	200/200		2.5 TB	1697.4 GB	
Completed Stages (14)								
13	groupBy c	+details 2023/04/01 17:21:59	35 min	5000/5000		2.6 TB	1760.2 GB	

ex-1) partition 수를 200 -> 5000으로 늘렸을 때, 같은 input data에 대해,
56min -> 35min으로 감소하는 것을 확인. (partition 당 데이터 사이즈가 너무 커진 것이 문제)

3. Partitioning의 중요성 - 1) 병렬 처리 성능

- Partitioning을 어떻게 했는지에 따라, 동일한 input 데이터에 대해서도 stage별 lead time이 크게 달라짐. (성능)
 - Partition: Spark의 병렬성의 단위
 - Partitioning을 어떻게 했는지에 따라 병렬 처리의 능률이 달라짐!

AS-IS		TO-BE								
Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read
1.9 min	6/6			4.1 GB		2.1 min	6/6			4.3 GB
6.8 min	2300/2300			23.9 GB	4.1 GB	3.7 min	500/500			24.4 GB
4.3 min	2300/2300	45.9 GB			23.9 GB	2.7 min	500/500	47.3 GB		
10 s	2300/2300	45.9 GB				10 s	500/500	47.3 GB		
52 s	2300/2300	Similar input size		24.6 GB		54 s	500/500			25.1 GB
5.9 min	24999/24999	1403.1 GB			24.6 GB	5.9 min	24999/24999	1403.3 GB		

ex-2) partition을 2300개에서 500개로 줄였을 때, 오히려 두 stage에서 lead time이 총 약 40% 정도 감소하는 것을 확인. (partition당 데이터 사이즈가 너무 작았던 것이 문제, small file problem)

3. Partitioning의 중요성 - 2) Resilience (장애 회복)

- 스파크에서 잡 실패도, partition level로 관리됨.
- 어떤 연산 중간 단계의 데이터에 대한 partition에서, 1개의 row에서 연산이 실패하면, 그 row가 포함된 전체 partition 을 재연산해야 함.
-> 즉 partition 사이즈가 너무 크다면, 재연산의 오버헤드가 커질 수 있음.

3. Partitioning의 중요성 - 3) Data Shuffling

- input 데이터와 output 데이터의 Partitioning 상태는, Partitioning 과정 자체에 큰 영향을 미침.
 - output partition 수 \gg input partition 수
 - Shuffle 연산을 시작하는 시점에 과부하가 발생 (input partition 1개당 사이즈가 더 크기 때문)
 - output partition 수 \ll input partition 수
 - Shuffle 완료 후에, 완료된 데이터를 가져오는 과정에서 부하 발생.
(output partition 1개당 사이즈가 더 크기 때문)
- > 두 케이스 모두 shuffle 연산에서 지연이나 에러를 발생시킬 수 있음.

3. Partitioning의 중요성 - 4) 연산의 효율성 증가 (성능)

- join, aggregation 연산 수행 시, 대상이 되는 데이터들의 partitioner가 동일하다면, wide transformation이 아닌 narrow transformation 수행 -> shuffle 연산 방지.
- ex) 약 2TB 정도의 데이터들을 join하는 연산이 포함된 spark application에서, input data 의 partitioner를 같게 세팅했을 때, 총 lead time이 1h 20m -> 1h 정도로 약 20% 정도 감소하는 것을 확인!

3. Partitioning의 중요성 - 5) Data skew 문제.

- partition 간의 데이터 사이즈 차이가 큰 경우.
- 병렬 처리 애플리케이션에서 data skew는 큰 성능 저하의 원인이 될 수 있음.
 - 크기가 작은 partition들의 연산이 아무리 빨리 수행되었더라도, skew된 partition에서의 연산이 끝나지 않으면, 다음 stage로 넘어갈 수 없기 때문.
- File 형태로 데이터를 쓸 때도, File 간의 skew 문제가 발생할 수 있음.

CustomPartitioner

1. 실습 코드 링크

- https://github.com/startFromBottom/fc-spark-practices/blob/main/6_partitioning/custom_partitioner_ex.py

2. 개요

- Spark에서는 기본적으로 HashPartitioner, RangePartitioner를 제공.
- 데이터의 특성에 맞는 Custom Partitioner를 사용하면, 특정 요구 조건에 맞게 데이터를 더 잘 Partitioning 할 수 있음.
 - ex) id가 어떤 특정한 형태를 나타낼 때, (Axxxx_Bxxxx , A2234_B3333, A2234_B3664)
기본 hash partitioning아닌 이 id의 형태에 맞게 hash function을 재정의.)
- Scala, Java에서는 아래의 abstract class를 상속 받아,
abstract method인 numPartitions, getPartition를 구현하는 식으로
Custom Partitioner 만들 수 있음.
<https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/Partitioner.html>
- 그러나 Pyspark에서는 위 방법을 사용할 수 없음. 대안이 필요.

3. 예제

- ‘subject’ 컬럼을 기준으로 3개의 partition이 나오도록 repartition.

```
subjects = ["Math", "English", "Science"]

data = [
    [i, subjects[i % 3], random.randint(1, 100)] for i in range(100)
]

df = ss.createDataFrame(data).toDF("id", "subject", "score")
score_df = df.repartition(3, "subject")
```

- 실제로 각 row가 어느 partition에 들어갔는지를 print 해보면, Partition 0에는 아무 데이터가 들어가지 않은 것을 확인 가능.

```
Partition 0:
```

```
Partition 1:
```

```
Row 0:Row(id=0, subject='Math', score=17)
Row 1:Row(id=1, subject='English', score=74)
Row 2:Row(id=3, subject='Math', score=87)
```

3. 예제

- 실제 각 row 별 hash 값(Hash #)과, hash 값을 partition 개수(3)으로 나눈 나머지 (Partition #) 값을 보면, subject가 Math인 경우와 English인 경우가 같은 값을 가지게 됨.
-> skewed partition의 원인!

id	subject	score	Hash#	Partition#
0	Math	33	1445171467	1
1	English	43	1666143475	1
2	Science	54	-636009586	-1
3	Math	03	1445171467	1

3. 예제

- 해결 방법)
 - subject list의 인덱스 값을 hash function의 input으로 사용하도록 하는 subject_partitioning이라는 udf function을 정의

```
# custom partitioning.
# coooooool
def subject_partitioning(k):
    return subjects.index(k)

udf_subject_hash = F.udf(lambda s: subject_partitioning(s))

num_partitions = 3
# if scala or java
# custom_df = df.partitionBy(num_partitions, subjectPartitioner)

custom_df = df.withColumn("Hash#", udf_subject_hash(df['subject']))
custom_df = custom_df.withColumn("Partition#", custom_df["Hash#"]
                                % num_partitions)
```

3. 예제

- 결과)
 - subject 별로 모두 다른 Partition에 데이터가 들어가는 것을 확인 가능

id	subject	score	Hash#	Partition#
0	Math	17	0	0.0
1	English	74	1	1.0
2	Science	99	2	2.0
3	Math	87	0	0.0
4	English	100	1	1.0

partition by vs bucket by

1. 실습 코드 링크

- https://github.com/startFromBottom/fc-spark-practices/blob/main/6_partitioning/bucketby_vs_partitionby_ex.py
- https://github.com/startFromBottom/fc-spark-practices/blob/main/6_partitioning/bucketby_join_ex.py

1. bucket by?

- output 데이터를 파일로 저장할 때, 지정된 컬럼의 값을 hash function에 넣은 후 나온 값을, bucketBy에 파라미터로 넣은 숫자로 나눈 나머지인 bucket_number에 해당하는 파일에 저장.

```
# 2. bucket by
# 만들어지는 파일의 개수 : dataframe의 현재 partition 개수 X bucketBy에 파라미터로 넣은 정수값.
# hash function 기반에 따라 파일을 생성하기 때문에,
# 데이터가 고르게 분배되기 위해서는 cardinality가 큰 컬럼으로 bucketBy하는 것이 좋음.
# ex)
# bucketBy(5, "age")
#     => bucket_number = hash(col("age")) % 5.
# 각 row는 위의 식으로부터 배정 받은 bucket_number에 해당하는 파일에 저장됨.

df.repartition(5).write.bucketBy(5, "age") \
    .mode("overwrite").saveAsTable("df_bucket_by_age")
```

part-00000-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00000.c000.snappy.parquet
part-00000-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00001.c000.snappy.parquet
part-00000-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00002.c000.snappy.parquet
part-00000-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00003.c000.snappy.parquet
part-00000-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00004.c000.snappy.parquet
part-00001-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00000.c000.snappy.parquet
part-00001-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00001.c000.snappy.parquet
part-00001-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00002.c000.snappy.parquet
part-00001-03b6eb8a-8d7f-4f41-b91b-6058e7255cda_00003.c000.snappy.parquet

bucket number

2. partition by, bucket by 비교

- **partition by**

- 파일로 쓸 때, 지정한 컬럼의 고유한 값들 만큼 partition 폴더가 생성됨.
- 파일의 총 개수 =

(dataframe의 현재 partition 개수 X

partitionBy에 지정한 컬럼의 고유한 값의 수.)

- partitionBy에 지정한 column의 cardinality가 크다면(=고유한 값의 종류가 많은), 너무 많은 파일이 만들어지는 문제가 발생할 수 있음.

- 일반적으로 date('2023-03-25')와 같은 컬럼을 사용.

- 데이터를 쓸 때 partition by를 해놓으면, Predicate Pushdown, Dynamic Partition Pruning 기법 등을 적용 가능!



2. partition by, bucket by 비교

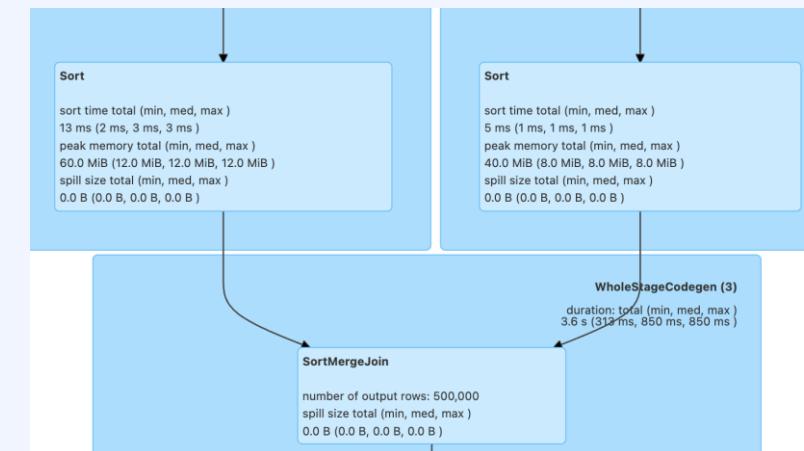
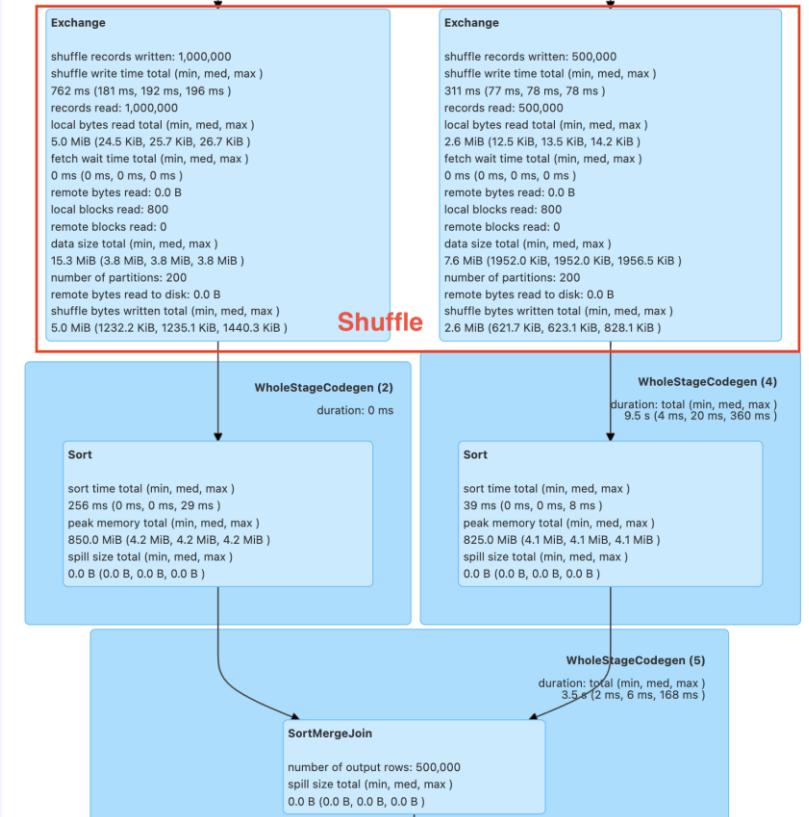
- **bucket by**

- 파일의 총 개수 =
(dataframe의 현재 partition 개수 X bucketBy 메서드에 지정한 정수)
- hash function 기반에 따라 파일을 생성하기 때문에,
데이터가 고르게 분배되기 위해서는 cardinality가 큰 컬럼으로 bucketBy하는 것이 좋음.(ex : id)
- 데이터를 쓸 때 bucketBy를 해놓으면,
join 연산 수행 시 wide transformation이 아닌 narrow transformation을 사용할 수 있게 됨.
(불필요한 Shuffle 발생 X)
 - 주의) bucketBy된 두 데이터를 읽어서 join할 때, 같은 숫자로 bucketing이 되어 있어야,
위의 효과를 얻을 수 있음!

```
df1.write.bucketBy(5, "id") \
    .mode("overwrite").saveAsTable("df1_bucket")

df2.write.bucketBy(5, "id") \
    .mode("overwrite").saveAsTable("df2_bucket")
```

3. bucket by 적용 전, 후 비교



4. 기타

- 파일로 쓸 때, bucketBy와 partitionBy를 모두 적용하는 것도 가능.
- ex)

```
# 3. partition by + bucket by
# 만들어지는 파일의 개수 :
#   (df partition 개수) X (partitionBy에 지정한 컬럼의 고유한 값의 수) X (bucketBy에 파라미터로 넣은 정수 값)
df.repartition(5).write.bucketBy(5, "salary") \
    .partitionBy("age").mode("overwrite").saveAsTable("df_bucket_by_partition_by")
```

Input partitions from Data files

개요

- 대부분의 Spark Application은, 다양한 포맷의 data file을 읽음.
(text, csv, parquet, orc, sequence File, binary, json etc)
- 각 data file의 특성을 이해하고, input RDD, DataFrame의 partition을 몇 개로 세팅할 것인지는 application 전체 성능에서 중요한 요소.
- input partition의 특성은 크게 아래 네 가지에 의해 결정됨.
 1. bucket 지정 여부.
 2. splittable file?
 3. 관련 spark config 파라미터.
 4. 전체 데이터의 크기.

1. input partition - bucket 지정 여부

- bucketing은 input 데이터의 partitioning과 직접적인 관련이 있음.
- bucketing에는 아래의 세 가지 파라미터가 필요함.
 - Bucket의 수
 - Bucketing할 컬럼들의 이름.
 - 정렬할 컬럼들의 이름.
- ex) https://github.com/startFromBottom/fc-spark-practices/blob/main/6_partitioning/bucketby_vs_partitionby_ex.py#L35

```
df.repartition(5).write.bucketBy(5, "age") \
    .mode("overwrite").saveAsTable("df_bucket_by_age")
```

2. input partition - splittable file?

- splittable?
 - 한 파일 내의 부분들이 병렬로 처리 될 수 있는가?
 - splittable한 File은 1개의 큰 파일을 작은 부분들로 쪼개서, 각각의 부분에서 병렬 처리를 가능하게 해줌.
 - 특정 File이 splittable인지는, File format과 Compression(압축) 여부 등에 따라 다름.
1. 압축과 상관 없이 Splittable한 File format -> 사용 강력 추천!
 - Parquet, ORC
 2. 나머지 file format 들은 대부분 특정 방법으로 압축(ex : bZip)되고, 모든 각각의 row가 1줄을 넘지 않을 때만 Splittable함.
 - ex) CSV, JSON, TEXT.
 3. 항상 Splittable이 아닌 format.
 - ex) Binary

3. input partition - 관련 spark config 파라미터

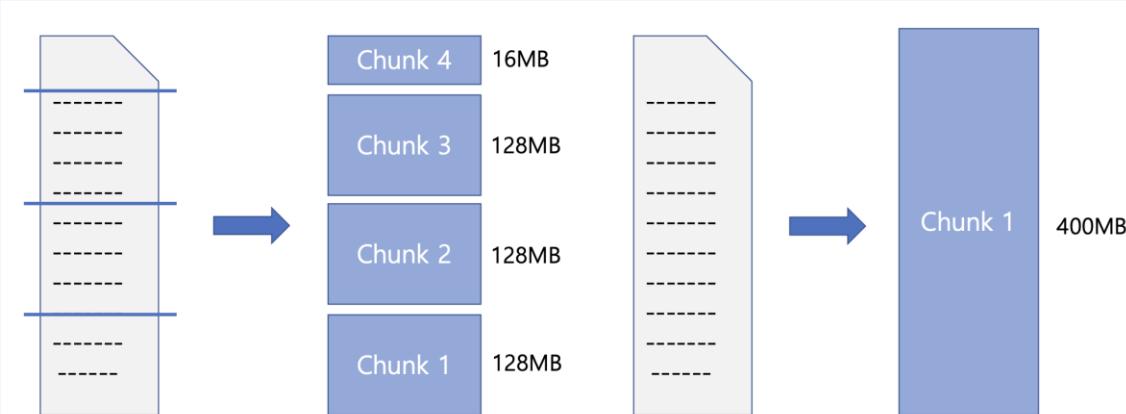
- **spark.default.parallelism.** -> for RDD
 - Spark application에 할당하는 total CPU cores의 기본 값
- **spark.sql.files.maxPartitionBytes.** -> for DataFrame
 - 각 partition의 최대 사이즈를 정의(default : 128MB)
- **spark.sql.files.openCostInBytes.** -> for DataFrame
 - input data file을 열 때의 비용(default : 4MB)
- **minSize (mapred.min.split.size).** -> for RDD
 - hadoop conf(default : 1MB)
- **blockSize (dfs.blocksize).** -> for RDD
 - hadoop conf - 각 Hadoop block의 사이즈 (default : 128MB)

4. input partition 결정 - Bucketing이 사용된 경우

- input 데이터의 partition 크기는,
다른 설정과 상관 없이 항상 bucketing을 하면서 세팅했던 partition의 개수와 동일함.
- partition의 개수 ?
 - bucketing 된 데이터를 dataframe 형태로 불러온 후, `df.getNumPartitions()` 메서드로 확인 가능.
- 뒤의 케이스들은 모두 bucketing이 안되어 있는 경우를 가정.

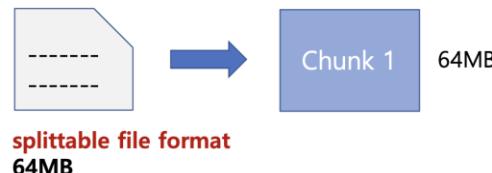
4. input partition 결정 - Dataframe) step 1. chunk 생성

- input file format : csv, json, parquet, orc.
- bytesPerCore = (데이터의 총 크기 + 데이터 내 파일의 개수 * openCostInBytes) / default.parallelism
- **maxSplitBytes = Min(maxPartitionBytes, bytesPerCore)**
- file 이 splittable 하고, file의 크기가 maxSplitBytes보다 크다면, file은 maxSplitBytes 크기의 partition로 쪼개진다. (마지막 chunk의 크기는 maxSplitBytes보다 작거나 같을 수 있음)



splittable file format
400MB maxSplitBytes = 128MB

unsplittable file format
400MB maxSplitBytes = 128MB

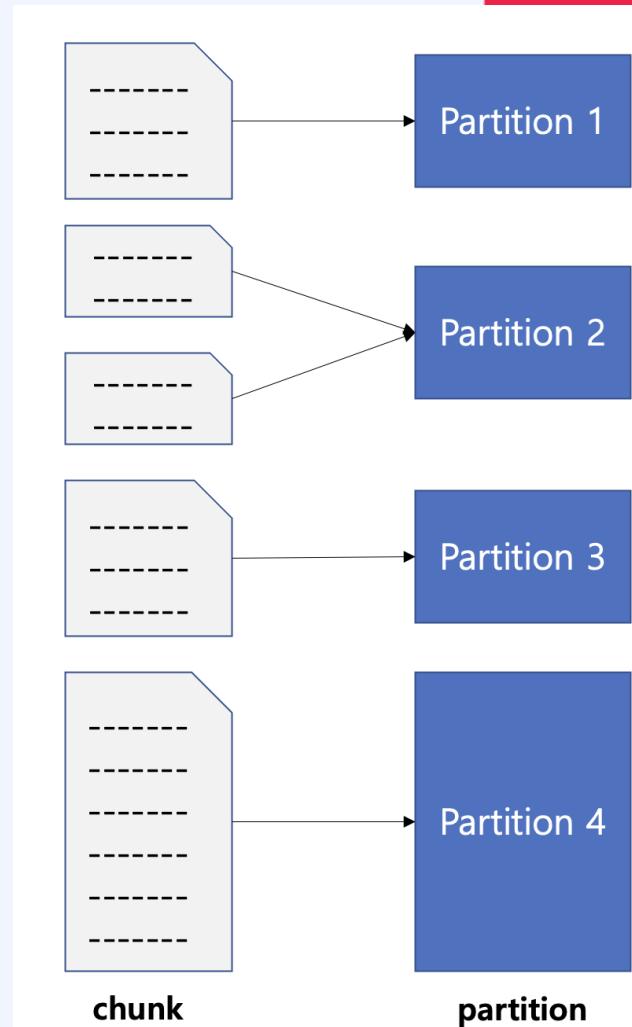


splittable file format
64MB

- file이 unsplittable하거나, splittable한 file의 크기가 maxSplitBytes보다 작다면, 그 file size 만큼의 1개의 Chunk가 생성됨.

4. input partition 결정 - Dataframe) step 2. chunk -> partition 구성

- 모든 data file들에서 chunk가 생성된 후에, 1개 이상의 chunk들로부터 1개의 partition이 구성됨.
- 아직 1개의 partition도 안 만들어졌으면, 새로운 partition을 초기화하고 반복문을 돌면서 chunk들을 partition에 할당.
- partition 1개의 크기 =
(partition을 구성하는 chunk들의 합) + (openCostInByte)
- partition 1개의 크기를 넘지 않을 때까지, chunk들을 partition에 넣고 packing.
- chunk 하나가 partition 1개의 크기보다 큰 경우(from Unsplittable)
-> 1개의 partition으로!

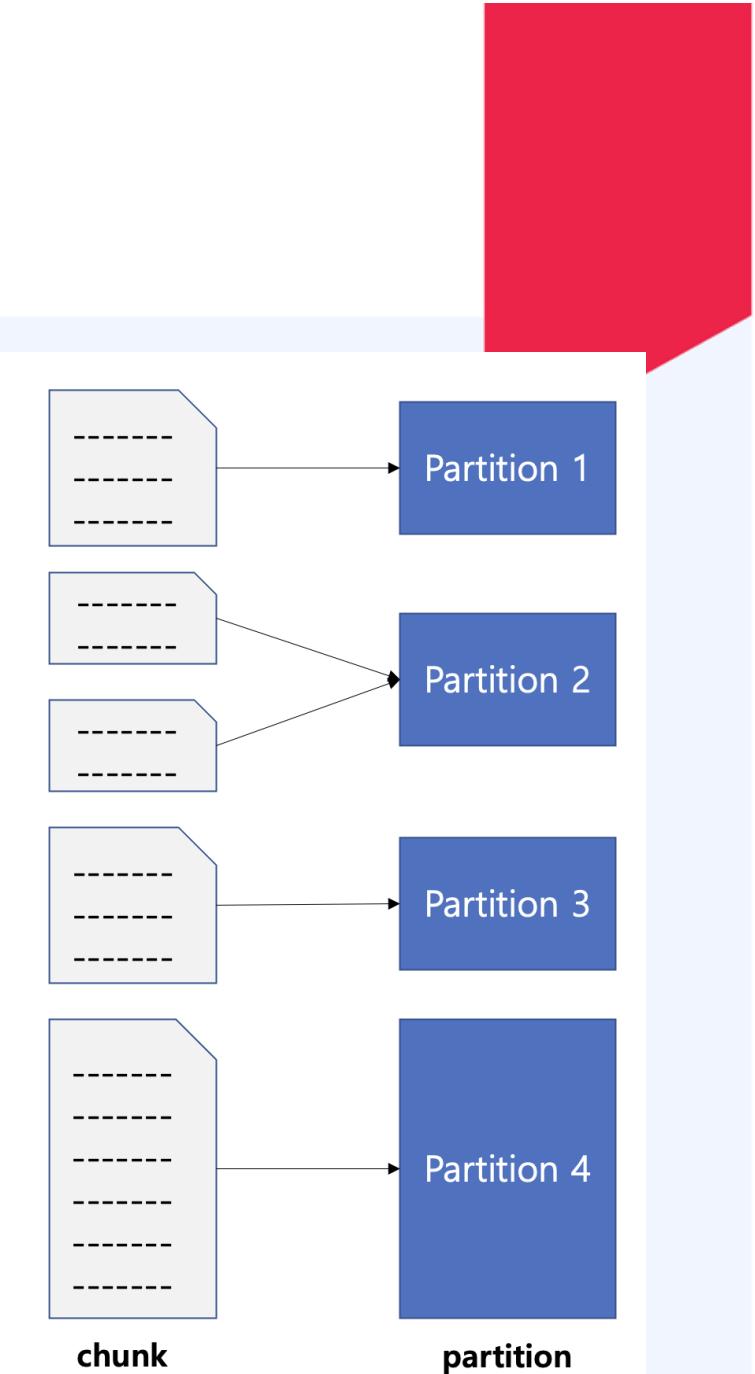
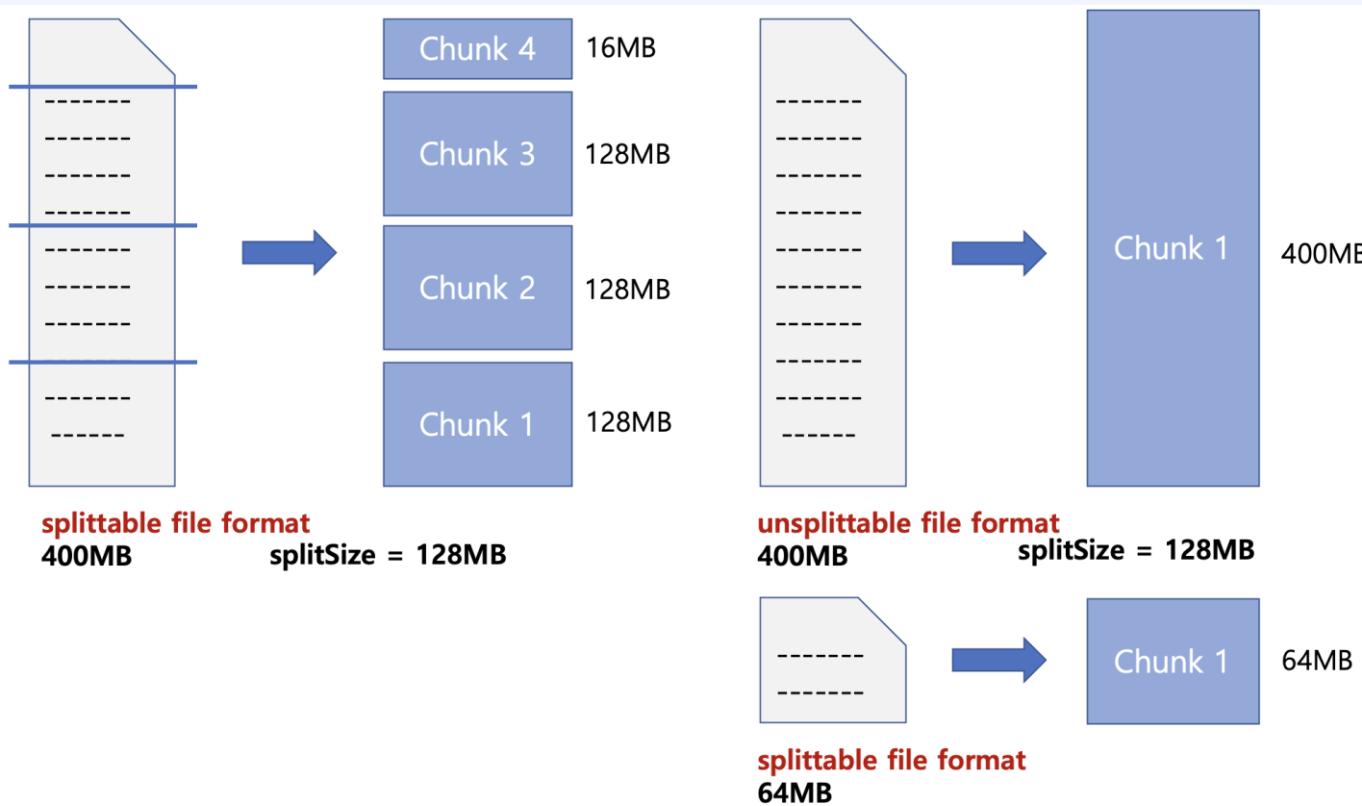


5. input partition 결정 - RDD

- input file : HadoopFile, textFile, sequenceFile
- DataFrame의 경우와 유사하게 chunk를 만들고, 그 chunk를 partition에 packing하는 방식이지만, 사용되는 파라미터가 다름.
- minSize = (mapred.min.split.size)
- blockSize = (dfs.blocksize)
- minPartitions (default value : 1)
- goalSize = (읽어들이는 모든 파일들의 크기 합 / minPartitions)
- **splitSize = Math.max(minSize, Math.min(goalSize, blockSize))**, (DataFrame에서의 maxSplitBytes 와 같음)
- file이 splittable이고 크기가 splitSize보다 크다면, file은 splitSize 단위의 chunk로 쪼개짐.
(DataFrame과 마찬가지로, 마지막 chunk의 크기는 splitSize보다 작거나 같음)
- file이 unsplittable하거나, splittable한 file의 크기가 splitSize보다 작다면,
그 file size 만큼의 1개의 Chunk가 생성됨.
- 이후 Chunk로부터 partition을 packing하는 과정은 DataFrame과 거의 유사.

5. input partition 결정 - RDD

- DataFrame의 경우와 유사하게 chunk를 만들고, 그 chunk를 partition에 packing하는 방식이지만, 사용되는 파라미터만 다름.



Partitioning during spark transformations

1. 개요

아래의 transformation 연산들이 수행된 후, output partition이 어떻게 세팅되는지 확인

- Narrow transformations.
 - map, filter, flatmap, union, hash-partitioned join, Coalesce.
- Wide transformations.
 - Join, Aggregation(groupby, groubyKey, reduceByKey etc..)
 - Repartition

2. Narrow transformation

- Narrow transformation에서는, coalesce(), union() 연산을 제외하고 input과 output의 partition 개수가 항상 동일.
 - (애초에 각 partition끼리만 연산되고, partition 간의 연산이 존재하지 않기 때문!)
- coalesce(N)의 경우, output partition의 개수는 파라미터로 넘겨준 N
- union() -> 뒷 장에서 확인.

2.1 Narrow transformation - Union (RDD)

- RDD와 DataFrame 두 케이스가 약간 다름.
 1. 만약 모든 input RDD들이 같은 Partitioner를 가지고 있고, partition의 개수도 같음.
-> output RDD의 partition 개수 = 각각의 input RDD의 partition 개수와 동일.
 2. 만약 input RDD의 partition 개수가 하나라도 서로 다르거나 다른 Partitioner를 가지고 있음.
-> output RDD의 partition 개수 = 모든 input RDD의 partition 개수의 합
 - $\text{union}(A, B, C)$, partition A = 2000, partition B = 3000, partition C = 1000
-> output partition 개수 = $2000 + 3000 + 1000 = 6000$
 3. inputRDD들 중 1개라도 Partitioner = None. (Partition 개수가 모두 같아도 적용)
-> output RDD의 partition 개수 = 모든 input RDD의 partition 개수의 합

2.2 Narrow transformation - Union (Dataframe)

- output Df의 partition 개수 = 모든 input Df의 partition 개수의 합 (항상!)

3. Wide transformation

- Join, Aggregation(Group by) 연산에서 로직이 조금씩 다름
- repartition(N)의 경우, output partition의 개수는 파라미터로 넘겨준 N

3.1 Wide transformation - Join (RDD)

- inner join, left outer join, right outer join, full outer join의 경우에서 모두 동일하게 적용.
 - input RDD 들이 모두 같은 Partitioner를 같은 경우는 narrow transformation이기 때문에, 여기서는 고려 대상 X
1. join의 대상이 되는 PairRDD (PySpark : RDD)의 경우에서, 모든 Partitioner = None 인 경우
 1. spark.default.parallelism이 세팅
 - output partition 개수 = spark.default.parallelism.
 2. spark.default.parallelism이 세팅 X.
 - output partition 개수 = max (input RDD들의 partition 개수)

3.1 Wide transformation - Join (RDD)

2. 1개 이상의 input RDD가 key에 대한 Partitioner 정보를 가지고 있는 경우.

- Partitioner 를 갖고 있는 input RDD(=A)들의 partition 개수와 `spark.default.parallelism(config)` 파라미터, 두 가지 숫자에 의해 결정됨.
 - config가 세팅된 경우.
 - output partition 개수 = max (config, A들의 partition 개수)
 - config가 세팅되지 않은 경우,
 - output partition 개수 = max (A들의 partition 개수)

3.2 Wide transformation - Join (Dataframe)

- Broadcast hash join
- Sort Merge join

두 가지의 경우에 대해 확인!

3.3 Wide transformation - Broadcast hash join (Dataframe)

- output partition의 개수 = 항상 broadcast 되지 않은 datafram의 partition 개수

3.4 Wide transformation - Sort Merge Join (Dataframe)

1. 두 input dataframe이 모두 특정 key(column)으로 hash partitioning 되어 있지 않은 경우.
 - output partition 수 = “spark.sql.shuffle.partitions”
2. 1개 이상의 input dataframe가 key(column)에 대해 hash partitioning 되어 있는 경우.
 - partitioning 되어 있는 input dataframe(=A)들의 partition 개수와 spark.sql.shuffle.partitions(config) 파라미터, 두 가지 숫자에 의해 결정됨.
 1. config가 세팅된 경우.
 - output partition 개수 = max (config A들의 partition 개수)
 2. config가 세팅되지 않은 경우,
 - output partition 개수 = max (A들의 partition 개수)

3.5 Aggregation (RDD)

1. input RDD가 이미 특정 Partitioner로 partitioning 되어 있는 경우
 - output RDD의 partition 수 = input dataframe의 partition의 수
2. inputRDD의 Partitioner = None
 - output RDD의 partition 수 = “spark.default.parallelism”
 - spark.default.parallelism이 세팅이 안된 경우,
output RDD의 partition 수 = input dataframe의 partition의 수

3.6 Aggregation (Dataframe)

1. input Dataframe이 이미 partitioning 되어 있는 경우
 - output dataframe의 partition 수 = input dataframe의 partition의 수
2. input Dataframe이 partitioning이 되어 있지 않은 경우.
 - output partition 수 = “spark.sql.shuffle.partitions (default = 200)”

Partitioning to output files

1. 개요

- Dataframe의 경우, 데이터를 파일로 쓰는 방법은 크게 네 가지 경우 존재.
 1. Basic (None)
 - 각각의 partition을 1개의 file로 쓰는 방식.
 - 단순하고, 높은 throughput (처리량)
 2. PartitionBy
 3. BucketBy
 4. PartitionBy + BucketBy
 - PartitionBy, BucketBy의 장점을 모두 취할 수 있음.(단점만 취할 수도 있음)
 - PartitionBy, BucketBy에 사용되는 컬럼은 반드시 달라야 함(같은 경우 에러 발생)
- PartitionBy와 BucketBy의 경우, “3. partition by vs bucket by” 강의에서 이미 다루었으니,
주의할 점만 언급
- 주의할 점)
 - partitionBy 시 cardinality가 높은 컬럼을 기준으로 하게 되면, 파일이 너무 많이 생성됨.
 - bucketBy 시 cardinality가 낮은 컬럼을 기준으로 하게 되면, bucketing의 이점을 누리기 어려움.

스파크 실무 팁

1. 데이터 파이프라인 운영시 발생했던 스파크 에러
들에 관한 case study.

1. 개요 - error의 종류

- Cluster, Spark Driver, executor 레벨에서 각각 여러 에러 케이스들이 존재.
- SparkUI, Spark log를 꼭 잘 확인하기!
- ex)
 1. upstream 데이터의 문제
 2. Executor OOM (Out of Memory), Executor Node lost failure.
 3. Driver OOM
 4. Disk 공간 부족.
 5. 느린 Join, GroupBy 속도.
 6. Serialization error
- etc..

1.1 upstream 데이터의 문제

- upstream으로부터 들어오는 데이터에 null 값이 존재.
- spark application에서 미리 정의한 schema와, upstream 데이터의 schema가 다른 경우.
etc..
- 다른 팀에서 upstream 데이터를 생성했다면, 그 쪽 담당자와 커뮤니케이션 필요.

1.2 Executor OOM (Out of Memory), Executor Node lost failure.

- 가능한 원인들 / 해결 방법
 - Executor의 메모리가 너무 작게 세팅됨.
-> Executor의 메모리 사이즈 증가
 - input data의 사이즈가 커짐.
-> Executor의 메모리 사이즈 증가, cluster의 사이즈를 증가시켜야 할 수도.
 - Skew된 Partition.
-> repartition 사용 고려.
 - etc..

1.3 Driver OOM (Out of Memory)

- 가능한 원인들 / 해결 방법
 - 사용자 코드에서 collect() 연산을 실행해 너무 큰 데이터셋을 driver에 전송하려 했을 때.
-> Driver에는 가급적 큰 데이터셋을 전송하면 안됨. 아니면 Driver의 메모리 크기 크게 세팅.
 - broadcast 하기 너무 큰 데이터를 broadcast join에 사용하려 할 때.
-> spark config 조정
 - 장시간 실행되는 Application에서, Driver에서 많은 객체를 생성 후에 제 때 해제하지 못하는 경우.
-> Java의 jmap 도구 등을 사용해 Heap 메모리 사용량 확인.

1.4 Disk 공간 부족

- 가능한 원인들 / 해결 방법
 - input data 크기 증가.
 - > 더 많은 Disk 공간 확보 필요.
 - skew된 파티션으로 인해, output file도 skew되는 경우.
 - > 데이터의 Partition 재분배 고려.
 - 오래된 로그 파일들이 지워지지 않고 쌓이는 경우.
 - > 파일 제거.

1.5 느린 Join, GroupBy 속도

- 가능한 원인들 / 해결 방법
 - 해당 stage의 task 수가 너무 작게 세팅됨. -> Partition 수 증가시키기
 - Executor의 메모리가 너무 작게 세팅. -> Executor 메모리 크게 세팅
 - skew 된 데이터.
 - 불필요한 Shuffle 발생.

1.6 Serialization error

- SparkSQL, Dataframe에서는 거의 발생하지 않음.
- UDF나 RDD 사용시 발생할 수 있음.
- Java나 Scala 클래스에서 UDF 생성 시, UDF 내에서 enclosing 객체를 참조하는 경우 발생.
(enclosing 객체 -> 내부 클래스를 감싸고 있는 외부 클래스의 객체 인스턴스)
-> 관련 필드를 클로저와 동일한 범주의 local 변수로 복사해 사용.

스파크 튜닝팁

1. 개요 - Spark Application의 성능 개선 방법

방법)

1. 도메인 지식을 바탕으로 코드의 근본 로직을 변경.
2. 근본 로직은 그대로 두고 스파크 application의 코드를 최적화.
3. spark의 config 튜닝.
4. input, output 데이터의 저장 방식.
5. 클러스터/애플리케이션 설정 최적화.
6. 최신 스파크 버전 사용. 등등

주의할 점)

1. 최적화 기법 적용 전, 후의 성능 변화를 항상 모니터링!
 - 모니터링 도구, spark application의 이력 추적이 가능한 환경이 반드시 구성되어야 함.
2. 최적화 기법 적용 전, 후 반드시 테스트를 거쳐 output 데이터에 이상이 없는지를 확인!
 - 데이터에 문제가 없는지 테스트해 볼 수 있는 환경이 구축되어야 함.

1.1 도메인 지식을 바탕으로 코드의 근본 로직을 변경.

- 도메인의 특성에 따라, 로직 변경이 쉬울 수도, 어려울 수도 있음.
- 개인적인 경험 상, 도메인에 대한 깊은 이해가 없이는 적용이 어렵고 관련 아이디어를 생각해내기도 어려움.

1.2 근본 로직은 그대로 두고 스파크 application의 코드를 최적화.

- 코드 최적화에는 여러 가지 방법이 존재.

1. Scala vs Java vs Python 언어 선택

- SparkSQL, DataFrame 사용 시에는 크게 중요하지 않음. 상황에 따라 가장 적합한 언어 선택.
- RDD나 UDF 사용 시 Python 사용 지양. (성능 이슈)
- ML 관련 -> Python에 풍부한 관련 라이브러리가 존재하기 때문에 Python 사용 권장.

2. DataFrame, SparkSQL vs RDD

- 특별히 RDD를 사용해야 할 목적이 있는게 아니라면 DataFrame, SQL 사용 권장.
- RDD는 spark engine에서 최적화가 안되고, 코드의 표현력 측면에서도 좋지 않음.

3. 그래도 RDD를 사용하는 경우, 기본 Java Serialization 대신 Kryo Serialization 사용!

- Java Serializer보다 훨씬 간결하고 효율적.

1.2 근본 로직은 그대로 두고 스파크 application의 코드를 최적화.

4. spark 연산들에서 발생할 수 있는 shuffle은 최소화!

- (RDD) -> groupByKey 사용 자양, reduceByKey, aggregateByKey 사용.
- Join, aggregate 연산 시 input RDD/Dataframe의 Partitioning 방법을 같게 하여
Narrow transformation이 되게 하기.
- Join하려는 테이블들 중, 작은 테이블이 있다면 Broadcast Hash join이 되도록 하기. etc

5. Partition 개수 조절

- Partition 개수 = Spark 의 병렬 처리 능력!
- 너무 작으면 병렬성을 제대로 활용하지 못하고, 너무 크면 scheduling의 오버헤드가 커짐.
- 한 Partition 당 사이즈가 대략 128MB 정도가 되도록 Partition 개수를 세팅.
- Data skew -> repartition() 사용 고려.
- Partition의 개수를 줄일 때는 repartition() 보다는 coalesce() 사용. etc=

6. 자주 접근하는 DataFrame, RDD에 대해서는 캐싱 적용.

7. UDF 함수 사용 자양.

8. 사용자 정의 Partitioning 사용.

1.3 spark 의 config 튜닝

1. Partition 개수 관련 설정

- spark.default.parallelism, spark.sql.shuffle.partitions 조정

2. Driver, executor의 core, executor 관련 설정

- --executor-memory (머신 스펙에 맞게 적용) , --executor-core (2 ~ 4 정도의 값 추천)

3. 공식 문서나, spark 관련 컨퍼런스에서 공유된 추천 config들 적용

- spark doc : <https://spark.apache.org/docs/latest/>
- 컨퍼런스 링크 예시 : <https://www.databricks.com/session/tuning-apache-spark-for-large-scale-workloads>

4. GC tuning. (공식 문서 확인)

- <https://spark.apache.org/docs/latest/tuning.html>

1.4 input, output data의 저장 방식

1. output 데이터를 file에 쓸 때 partitionBy, bucketBy 사용 고려
 - partitionBy : predicate pushdown, partition pruning.
 - bucketBy : bucketing된 데이터들끼리 join할 때 shuffle 발생 X
2. 파일의 개수 고려
 - 파일 개수가 너무 많으면 small file 문제.
 - 파일 개수가 너무 적으면 Spark의 병렬 연산을 제대로 활용하지 못함.
3. Splittable한 파일 포맷 사용.
 - 압축 방법에 관계 없이 항상 Splittable한 파일 포맷인 parquet, orc 사용

1.5 클러스터/애플리케이션의 설정 최적화

1. Dynamic Allocation 사용 고려

- 더 이상 사용하지 않는 자원을 클러스터에 반환하고 필요할 때 다시 요청할 수 있음

2. (AWS) spot instance 사용

- ondemand instance 대비 비용 50% 이상 감소.
- 모든 node를 spot으로 사용하기는 위험. ondemand, spot을 혼합해서 사용을 권장.
- spot instance ? https://docs.aws.amazon.com/ko_kr/AWSEC2/latest/UserGuide/using-spot-instances.html

3. 클러스터의 스펙 최적화

1.6 최신 스파크 버전 사용

1. 최신 스파크에 적용된 최적화 기법들을 통해 성능 개선 가능
 - ex) Spark 3.0 + 에 도입된 AQE, DPP 기법

Spark Unit Test 작성.

1. 실습 코드 링크

https://github.com/startFromBottom/fc-spark-practices/tree/main/7_spark_tips