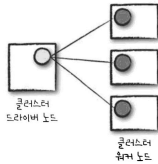


Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

	<ul style="list-style-type: none">- 스파크에서 코드를 실행할 때 어떤 일이 발생하는지 알아보자이 장에서 알아볼 주제<ul style="list-style-type: none">- 스파크 애플리케이션의 아키텍처와 컴포넌트- 스파크 내/외부에서 실행되는 스파크 애플리케이션의 생애주기- 파이프라이닝과 같은 중요한 저수준 실행 속성- 스파크 애플리케이션을 실행하는데 필요한 사항
	<p>15.1 스파크 애플리케이션의 아키텍처</p> <ul style="list-style-type: none">- 스파크 애플리케이션 관련된 고수준 컴포넌트를 다시 알아보자- 스파크 드라이버<ul style="list-style-type: none">물리적 머신의 프로세스, 클러스터 실행 중인 애플리케이션 상태 유지- 스파크 익스큐터<ul style="list-style-type: none">스파크 드라이버가 할당한 태스크를 수행하는 프로세스태스크 상태와 결과(성공/실패)를 드라이버에 보고모든 스파크 애플리케이션은 개별 익스큐터 프로세스를 사용- 클러스터 매니저<ul style="list-style-type: none">스파크 애플리케이션을 실행할 클러스터 머신을 유지드라이버(마스터)와 워커 개념을 가지고 있음. 물리적 머신에 연결되는 개념
	<div><p>그림 15-1 실행 중인 스파크 애플리케이션이 없는 클러스터 드라이버와 워커</p><p>클러스터 드라이버 노드</p><p>클러스터 워커 노드</p><ul style="list-style-type: none">○ 클러스터 드라이버● 클러스터 워커 프로세스</div> <ul style="list-style-type: none">- 원은 개별 워커 노드를 실행/관리하는 데몬 프로세스- 그림에서 스파크 애플리케이션 실행 전임

Part 4 - 운영용 애플리케이션

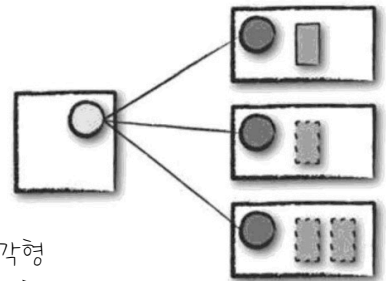
chapter 15 - 클러스터에서 스파크 실행하기

- 스파크가 지원하는 클러스터 매니저
 - 스탠드얼론 클러스터 매니저
 - 아파치 메소스
 - 하둡 YARN

15.1 실행 모드

- 실행 모드: 애플리케이션 실행 시 자원¹의 물리적 위치 결정, 3가지 모드 선택 가능
 - 클러스터 모드, 클라이언트 모드, 로컬 모드

그림 15-2 스파크의 클러스터 모드



- 드라이버 프로세스: 실선 직사각형
- 익스큐터 프로세스: 점선 직사각형

클러스터 모드

- 가장 흔하게 사용되는 스파크 애플리케이션 실행 방식
- 모든 스파크 애플리케이션과 관련된 프로세스를 유지
- 컴파일된 JAR파일, 파이썬/R 스크립트를 클러스터 매니저에 전달
 - > 워커노드에서 드라이버와 익스큐터 프로세스 실행
- [그림 15-2]는 하나의 워커 노드에 드라이버 할당하고 다른 노드에 익스큐터 할당

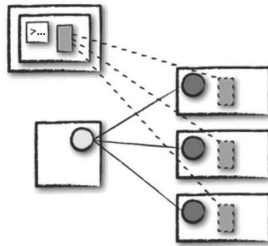
Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

클라이언트 모드

- 클러스터 모드와 비슷, 차이점은 클라이언트 머신에 스파크 드라이버가 위치함
- 드라이버 프로세스를 유지하며 클러스터 매니저는 엑스큐터 프로세스를 유지

그림 15-3 스파크의 클라이언트 모드



- 스파크 애플리케이션이 클러스터와 무관한 머신에서 동작
- 이러한 머신을 게이트웨이 머신 또는 에지 노드라고 부름
- 드라이버는 클러스터 외부 머신에서 실행되고 나머지 워커는 클러스터에 위치함

로컬 모드

- 모든 스파크 애플리케이션이 단일 머신에서 실행
- 애플리케이션 병렬 처리를 위해 단일 머신 스레드 활용
- 스파크 학습 or 애플리케이션 테스트 or 개발 중인 애플리케이션 실험하는 용도

15.2 스파크 애플리케이션 생애주기(스파크 외부)

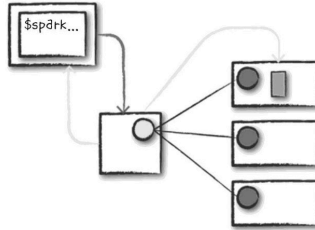
spark-submit 명령을 사용해 애플리케이션 실행 예제 설명할게!

15.2.1 클라이언트 요청

- 첫 단계는 스파크 애플리케이션(컴파일된 jar/라이브러리 파일)을 제출하는 것
- 이 과정에서 스파크 드라이버 프로세스의 자원을 함께 요청
- 클러스터 매니저: 클러스터 노드 중 하나에 드라이버 프로세스 실행
- 클라이언트 프로세스 종료-> 클러스터에서 애플리케이션 실행

- 스파크 애플리케이션을 제출하기 위한 명령 실행(터미널)

그림 15-4 드라이버 실행을 위한 자원 요청

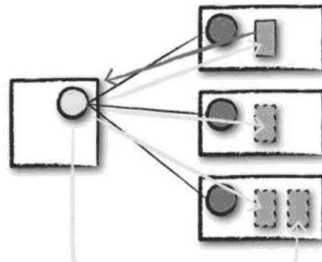


```
./bin/spark-submit \  
--class <main-class> \  
--master <master-url> \  
--deploy-mode cluster \  
--conf <key>=<value> \  
... # 다른 옵션  
<application-jar> \  
[application-arguments]
```

15.2.2 시작

- 드라이버 프로세스가 클러스터에 배치되었으므로 사용자 코드를 실행 (스파크 클러스터(ex 드라이버와 익스큐터)를 초기화하는 SparkSession 포함)**
- SparkSession은 클러스터와 매니저와 통신해 스파크 익스큐터 프로세스의 실행을 요청
- 사용자는 spark-submit 실행때 사용하는 명령행 인수로 익스큐터 수/설정 수 지정 가능

그림 15-5 스파크 애플리케이션 시작하기

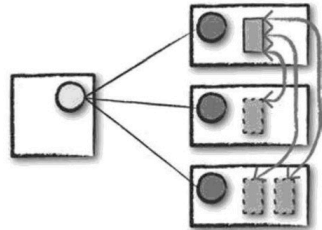


- 클러스터 매니저 역할 익스큐터 프로세스 시작 및 결과 응답 받아 익스큐터의 위치 관련 정보를 드라이버 프로세스로 전송
- 모든 작업이 완료되면 스파크 클러스터 완성

15.2.3 실행

- 코드 실행, 드라이버와 워커는 코드 실행 후 데이터 이동 과정에서 서로 통신
- 드라이버>워커에 태스크 할당, 워커>태스크 상태/성공/실패 여부 드라이버 전송

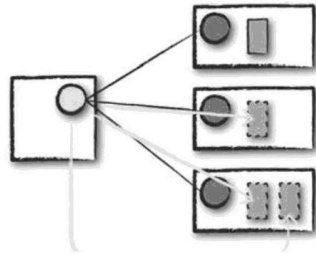
그림 15-6 애플리케이션 실행



15.2.4 완료

- 스파크 애플리케이션의 실행이 완료되면 드라이버 프로세스가 성공/실패로 종료
- 이 시점에 스파크 애플리케이션 성공/실패 여부를 클러스터 매니저에 요청 확인 가능

그림 15-7 애플리케이션 종료



Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

15.3 스파크 애플리케이션의 생애주기(스파크 내부)

- 애플리케이션을 실행하면 스파크 내부에서 어떤 일이 발생하는지 알아야함
- 이 장은 '사용자 코드'와 관련된 이야기가 진행됨
- 스파크 애플리케이션은 하나 이상의 스파크 잡으로 구성
- 스레드를 사용해 병렬 사용하는 경우가 아니라면 스파크 잡은 차례대로 실행됨

15.3.1 SparkSession

- 모든 스파크 애플리케이션은 SparkSession 생성(대화형 모드 자동 생성)
- SparkSession 빌더 메서드 사용해 생성할 것을 추천
- SparkSession 생성하면 스파크 코드 실행 가능
- SparkSession 클래스는 2.x 버전(이상)에서 사용 가능
- 과거 버전은 SparkContext와 SQLContext를 직접 생성

// 스칼라에서 SparkSession 생성하기

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

```
val spark = SparkSession.builder().appName("Databricks Spark Example")  
    .config("spark.sql.warehouse.dir", "/user/hive/warehouse")  
    .getOrCreate()
```

파이썬에서 SparkSession 생성하기

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.master("local").appName("Word Count")\  
    .config("spark.some.config.option", "some-value")\  
    .getOrCreate()
```

Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

SparkContext

- SparkSession의 SparkContext는 스파크 클러스터에 대한 연결을 나타냄
- SparkContext를 이용해 RDD, 어큐뮬레이터, 브로드캐스트 변수 생성 및 사용 가능
- 일반적으로 sc 변수를 사용함
- 명시적으로 SparkContext를 초기화 할 필요 없음
- 초기화하는 일반적인 방법은 getOrCreate 메서드 활용
- SparkSession이 초기화되면 코드를 실행. 모든 스파크 코드는 RDD명령으로 컴파일 함
SparkSession, SQLContext, HiveContext
- 과거에는 SQLContext와 HiveContext 사용해 DataFrame과 스파크SQL 사용
- 스파크 2.0에서 두 컨텍스트를 SparkSession으로 단일화

15.3.2 논리적 명령

- 스파크 코드는 트랜스포메이션과 액션으로 구성됨
- 선언적 명령(DataFrame 등)을 사용하는 방법과 논리적 명령이 물리적 실행 계획으로 변환되는 과정을 이해하는 것이 중요함
- 이 절에서는 잡, 스테이지, 태스크를 차례로 따라가며 코드를 실행

논리적 명령을 물리적 실행계획으로 변환하기

파이썬 코드

```
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")

step4.collect() # 결과는 2,500,000,000,000
```

- 코드를 실행하면 액션으로 하나의 스파크 잡이 완료됨
- 이제 실행계획을 살펴보자
- 실행계획 정보는 스파크 UI의 SQL 탭에서도 확인가능

Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

```
step4.explain()

== Physical Plan ==
*HashAggregate(keys=[], functions=[sum(id#15L)])
+- Exchange SinglePartition
   +- *HashAggregate(keys=[], functions=[partial_sum(id#15L)])
      +- *Project [id#15L]
         +- *SortMergeJoin [id#15L], [id#10L], Inner
            :- *Sort [id#15L ASC NULLS FIRST], false, 0
               :- +- Exchange hashpartitioning(id#15L, 200)
                  :- +- *Project [(id#7L * 5) AS id#15L]
                     :- +- Exchange RoundRobinPartitioning(5)
                        :- +- *Range (2, 100000000, step=2, splits=8)
               +- *Sort [id#10L ASC NULLS FIRST], false, 0
                  +- Exchange hashpartitioning(id#10L, 200)
                     +- Exchange RoundRobinPartitioning(6)
                        +- *Range (2, 100000000, step=4, splits=8)
```

- collect 같은 액션을 호출하면

개별 스테이지와 태스크로

이루어진 스파크 잡이 실행됨

- 로컬 머신에서 스파크 잡을 실행

하면 localhost:4040에 접속해

스파크 UI를 Jobs 탭에서 확인

가능

15.3.3 스파크 잡

- 보통 액션 하나당 하나의 스파크 잡이 생성되며 액션은 항상 결과를 반환

- 스파크 잡은 스테이지로 나뉘고 스테이지의 수는 셔플 작업이 얼마나 발생하는지에 따라 달라짐

- 스테이지 1: 태스크 8개
- 스테이지 2: 태스크 8개
- 스테이지 3: 태스크 5개
- 스테이지 4: 태스크 6개
- 스테이지 5: 태스크 200개
- 스테이지 6: 태스크 1개

<- 이전 예제의 잡은 6개의 스테이지로

나뉘짐

이게 어떻게 이렇게 나온거지?라는

의문이 들었다면 다음 절에서 확인해

보자

Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

15.3.4 스테이지

- 다수의 머신에서 동일한 연산을 수행하는 태스크의 그룹을 나타냄
- 스파크는 가능한 많은 태스크(job의 transformation)를 동일 스테이지로 묶으려 노력함
- 셔플 작업 후에는 반드시 새로운 스테이지 시작(셔플 : 데이터의 물리적 재분배 과정)
- 셔플 작업의 예: DataFrame 정렬, key별로 적재된 파일 데이터를 그룹화 등
 - 파티션 재분배 과정은 데이터를 이동시키는 작업->익스큐터 간의 조정 필요
 - 셔플 끝난 다음 새로운 스테이지 시작하며 최종 결과 계산(스테이지 실행 순서 추적)
- 스테이지, 2는 range명령을 수행하는 단계임
- 단계 1: range 명령으로 DataFrame을 생성하면 8개의 파티션 생성
- 단계 2: 파티션 재분배 단계(셔플로 파티션 수를 변경, 두 개의 DataFrame은 스테이지 3과 4의 태스크 수에 해당하는 5개, 5개의 파티션으로 재분배 됨)
- 단계 3: 스테이지 3과 4는 개별 DataFrame에서 수행됨. 마지막 두 스테이지는 조인(셔플) 수행. sparkSQL 기법 설정(spark.sql.shuffle.partition=200)으로 200개의 셔플 파티션을 생성됨. 이 설정 값은 변경 가능함

Tip - 파티션 수는 매우 중요한 파라미터임.

이 값은 효율적 실행을 위해 클러스터 코어 수에 맞춰 설정함

`spark_conf.set("spark.sql.shuffle.partitions", 50)` <- 설정법

- 클러스터의 익스큐터 수보다 파티션 수를 더 크게 지정하는 것이 좋음
- 로컬 머신 실행의 경우 병렬로 처리하는 태스크가 제한적 -> 값을 작게 설정
- 최종 스테이지는 개별적으로 수행된 결과를 단일 파티션으로 모으는 작업 수행

15.3.5 태스크

- 스파크 스테이지는 태스크로 구성됨.
- 익스큐터에서 실행할 데이터의 블록과 다수의 트랜스포메이션 조합임.
- 데이터 단위(파티션)에 적용되는 연산 단위임 (1 파티션 : 1태스크, 100파티션 : 100태스크)

Part 4 - 운영용 애플리케이션

chapter 15 - 클러스터에서 스파크 실행하기

15.4 세부 실행 과정

- 스테이지와 태스크의 중요한 특성 2가지
 - map 연산 후 다른 map 연산이 이어지면 함께 실행가능하도록 스테이지와 태스크를 자동으로 연결
 - 셔플 작업시 데이터를 디스크에 저장하므로 여러 잡에서 재사용 가능
 - 스파크 UI로 확인 가능

15.4.1 파이프라이닝

- 스파크는 메모리나 디스크에 데이터를 쓰기전에 최대한 많은 단계를 수행
- 파이프라이닝(RDD나 RDD 더 아래에서 발생)은 스파크 주요 최적화 기법 중 하나
- 파이프라이닝: 노드 간 데이터 이동 없이 각 노드가 데이터를 직접 공급할 수 있는 연산만 모아 태스크의 단일 스테이지로 만들어버림
- 파이프라인으로 구성된 연산 작업은 단계별로 메모리/디스크에 중간 결과를 기록하는 방식보다 처리 속도가 빠름(select, filter, DataFrame, SQL 연산에도 적용됨)
- 스파크 런타임에서 파이프라이닝을 자동 수행(스파크 UI로 확인 가능)

15.4.2 셔플 결과 저장(shuffle persistence)

- 노드 간 복제(reduce-by-key 등)를 유발하는 연산 실행하면 네트워크 셔플이 발생
- 이 연산은 각 키에 대한 입력 데이터를 여러 노드로부터 복사 (소스 태스크를 먼저 수행)
- 소스 태스크 스테이지 실행되는 동안 셔플 파일을 로컬 디스크에 기록
- 이후 그룹화 / 리듀스 수행 스테이지가 시작됨 (셔플 파일에서 레코드를 읽고 연산 수행)
- 부작용: 이미 셔플된 데이터로 새로운 잡 실행시 소스와 관련된 셔플이 실행되지 않음
- 더 나은 성능을 얻기 위해 DataFrame이나 RDD의 cache 메서드를 사용 가능
- 사용자가 직접 캐싱 가능하며 어떤 데이터가 어디에 저장되는지 제어 가능

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

- 스파크 애플리케이션 개발하고 클러스터에 배포하는 과정에 대해서 알아보자
- 빌드 도구 설정, 단위 테스트, 애플리케이션 구성 방법을 담은 템플릿 사용해보자

16.1 스파크 애플리케이션 작성하기

- 스파크 애플리케이션은 두 가지 조합으로 구성됨 [스파크 클러스터, 사용자 코드]

16.1.1 간단한 스칼라 기반 앱

- 스파크 애플리케이션 빌드는 sbt, 아파치 메이븐으로 가능(sbt가 더 쉽다)
- 환경 구성을 위해 build.sbt 파일을 정의해야 함. 아래는 build.sbt 파일 핵심 항목
 - 프로젝트 메타데이터(패키지명, 패키지 버전 정보 등)
 - 라이브러리 의존성 관리 장소
 - 라이브러리 포함된 의존성 정보
- 더 깊게 공부하고 싶다면 [스칼라를 위한 SBT 시작하기] 책 참고
- 아래 코드는 사클라우드 build.sbt 파일 내용 중 일부(템플릿 전체 <https://bit.ly/2NcAN72>)

```
src/  
  main/  
    resources/  
      <JAR 파일에 포함할 파일들>  
    scala/  
      <메인 스칼라 소스 파일>  
    java/  
      <메인 자바 소스 파일>  
  test/  
    resources  
      <테스트 JAR에 포함할 파일들>  
    scala/  
      <테스트용 스칼라 소스 파일>  
    java/  
      <테스트용 자바 소스 파일>
```

```
name := "example"  
organization := "com.databricks"  
version := "0.1-SNAPSHOT"  
scalaVersion := "2.11.8"  
  
// 스파크 관련 정보  
val sparkVersion = "2.2.0"  
  
// 스파크 패키지 포함  
resolvers += "bintray-spark-packages" at  
  "https://dl.bintray.com/spark-packages/maven/"  
  
resolvers += "Typesafe Simple Repository" at  
  "http://repo.typesafe.com/typesafe/simple/maven-releases/"  
  
resolvers += "MavenRepository" at  
  "https://mvnrepository.com/"  
  
libraryDependencies += Seq(  
  // 스파크 코어  
  "org.apache.spark" %% "spark-core" % sparkVersion,  
  "org.apache.spark" %% "spark-sql" % sparkVersion,  
  // 나머지 부분은 생략  
)
```

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

- 스크라와 자바 디렉터리에 소스코드 작성
- 아래는 SparkSession 초기화하고 애플리케이션 실행 후 종료하는 예제

```
object DataFrameExample extends Serializable {  
  def main(args: Array[String]) = {  
  
    val pathToDataFolder = args(0)  
  
    // 명시적으로 몇 가지 설정값을 지정한 다음 SparkSession을 시작합니다.  
    val spark = SparkSession.builder().appName("Spark Example")  
      .config("spark.sql.warehouse.dir", "/user/hive/warehouse")  
      .getOrCreate()  
  
    // UDF를 등록합니다.  
    spark.udf.register("myUDF", someUDF(_ :String):String)  
  
    val df = spark.read.json(pathToDataFolder + "data.json")  
    val manipulated = df.groupBy(expr("myUDF(group)")).sum().collect()  
      .foreach(x => println(x))  
  }  
}
```

- spark-submit 명령으로 코드 제출(to 클러스터). main 클래스 지정 부분을 눈여겨보자
- 다음은 빌드: sbt assemble 명령 사용(uber-jar or fat-jar로 빌드하기 위해)
- 위의 명령은 라이브러리 의존성 충돌 문제로 복잡한 상황이 발생할 수 있음
- 쉬운 빌드 방법은 sbt package 명령 실행하는 것
- 이 명령은 관련 라이브러리를 target 폴더로 모을 수 있지만 JAR파일을 만들지 않음

애플리케이션 실행하기

- Target 폴더에 spark-submit에서 인수로 사용할 jar 파일이 들어있음
- 빌드(<https://bit.ly/2C0gg3T>)하면 아래와 같이 로컬 머신에서 실행 가능

```
$SPARK_HOME/bin/spark-submit \  
--class com.databricks.example.DataFrameExample \  
--master local \  
target/scala-2.11/example_2.11-0.1-SNAPSHOT.jar "hello"
```

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

16.1.2 파이썬 애플리케이션 작성하기

- PySpark 애플리케이션 작성법은 일반 파이썬 패키지 작성법과 거의 비슷
- 스파크는 빌드 개념이 없고 PySpark 애플리케이션은 파이썬 스크립트임
- 코드 재사용을 위해 파이썬 파일을 egg나 ZIP 파일 형태로 압축
- spark-submit의 --py-file 인수로 .py, .zip, .egg 파일 지정하면 애플리케이션과 함께 배포 가능
- 스칼라나 자바의 메인 클래스 역할을 하는 파이썬 파일 작성 필요
- 아래는 sparkSession 실행 가능한 스크립트 파일 예제(spark-submit의 main 인수로 지정할 클래스 코드)

```
# 파이썬 코드
from __future__ import print_function

if __name__ == '__main__':
    from pyspark.sql import SparkSession
    spark = SparkSession.builder \
        .master("local") \
        .appName("Word Count") \
        .config("spark.some.config.option", "some-value") \
        .getOrCreate()

    print(spark.range(5000).where("id > 500").selectExpr("sum(id)").collect())
```

- 위 코드가 실행되면 sparkSession 객체 생성됨. 모든 파이썬 클래스에서 sparkSession 객체를 생성하는 것보다 런타임 환경에서 변수를 생성해 클래스에 전달 방식을 추천
- PySpark 의존성 정의 위해 pip 사용 가능(pip install pyspark 명령으로 설치)

애플리케이션 실행하기

```
$SPARK_HOME/bin/spark-submit --master local pyspark_template/main.py
```

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

16.1.3 자바 애플리케이션 작성하기

- 자바와 스칼라를 이요한 스파크 애플리케이션 작성은 매우 유사
- 가장 큰 차이점은 라이브러리 의존성 지정하는 방법
- 아래의 자바 애플리케이션 예제는 메이븐 활용해 의존성 지정, 메이븐 사용하려면 스파크 패키지 저장소를 반드시 추가 해야함

```
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.1.0</version>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.1.0</version>
  </dependency>
  <dependency>
    <groupId>graphframes</groupId>
    <artifactId>graphframes</artifactId>
    <version>0.4.0-spark2.1-s_2.11</version>
  </dependency>
</dependencies>
<repositories>
  <!-- 저장소 목록을 나열합니다. -->
  <repository>
    <id>SparkPackagesRepo</id>
    <url>http://dl.bintray.com/spark-packages/maven</url>
  </repository>
</repositories>
```

- 디렉터리 구조는 스칼라 프로젝트와 동일 (메이븐 명세를 따름)



```
import org.apache.spark.sql.SparkSession;

public class SimpleExample {
    public static void main(String[] args) {
        SparkSession spark = SparkSession
            .builder()
            .getOrCreate();
        spark.range(1, 2000).count();
    }
}
```

자바 코드 빌드 및 실행



```
$SPARK_HOME/bin/spark-submit \
--class com.databricks.example.SimpleExample \
--master local \
target/spark-example-0.1-SNAPSHOT.jar "hello"
```

애플리케이션 실행하기

16.2 스파크 애플리케이션 테스트

- 약간 지루하면서 아주 중요한 테스트 방법을 알아보자
- 테스트를 위해선 몇 가지 핵심 원칙과 구성 전략을 고려해야 함

16.2.1 전략적 원칙

- 테스트 코드 개발(파이프라인, 애플리케이션)은 실제 애플리케이션 개발만큼이나 중요
- 미래에 발생할 수 있는 데이터, 로직, 결과 변화에 유연하게 대처할 수 있음

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

입력 데이터에 대한 유연성

- 비즈니스 요구사항이 변하면 데이터도 변함
- 오류상황을 적절하게 대처 / 제어 가능해야 함
- 입력 데이터로 발생 가능한 다양한 예외 상황을 테스트하는 코드 작성 필요

비즈니스 로직 변경에 대한 유연성

- 파이프라인 내부 비즈니스 로직이 바뀔 수 있음
- 이 유형의 테스트는 스파크 기능을 테스트(스파크 단위 테스트)하는게 아님
- 비즈니스 로직을 테스트해 복잡한 비즈니스 파이프라인이 의도대로 동작하는지 확인

결과의 유연성과 원자성

- 입력 데이터 / 비즈니스 로직 테스트가 완료되면 의도한대로 데이터가 반환되는지 확인
- 대부분의 스파크 파이프라인은 다른 스파크 파이프라인의 입력으로 사용됨
- 다음 파이프라인에서 데이터의 상태(갱신 주기, 데이터가 완벽한지(누락 확인 등), 마지막 순간에 데이터가 변경되지 않았는지)를 확인하자

16.2.2 테스트 코드 작성 시 고려사항

- 테스트를 쉽게 만들어주는 테스트 구성 전략을 알아보자
- 적절한 단위 테스트를 작성해 입력 데이터나 구조가 변경되어도 비즈니스 로직이 정상 작동하는지 확인해야 함
- 단위 테스트는 스키마 변경 상황에 따라 쉽게 대응 가능
- 단위 테스트 구성 방법은 비즈니스 도메인, 도메인 경험에 따라 다양함(개발자 역량)

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

SparkSession 관리하기

- 스파크 로컬 모드로 단위 테스트용 프레임워크(unit, scalaTest 등)로 테스트 가능
- 테스트 하네스(test harness: 테스트를 지원하기 위해 생성된 코드와 데이터)의 일부로 로컬 모드의 sparkSession을 만들어 사용하면 됨
- 테스트 방식이 잘 동작하기 위해선 스파크 코드에서 의존성 주입 방식으로 sparkSession을 관리하도록 만들어야 함
- sparkSession을 한번만 초기화하고 런타임 환경에서 함수와 클래스에 전달

테스트 코드용 스파크 API 선정하기

- API는 사용자 애플리케이션의 유지보수성 / 테스트 용이성 측면에서 다른 영향을 미침
- 개발 속도를 올리기 위해 SQL, DataFrame을 사용할 수 있고 타입 안정성을 위해 Dataset과 RDD API를 사용할 수 있음
- 타입 안정성 API: 규약(함수 시그니처)는 다른 코드에서 재사용하기 용이함
- DataFrame, SQL: 각 함수의 입출력 타입을 문서로 만들고 테스트하는 노력이 필요
- 저수준RDD API: 파티셔닝같은 저수준 API가 필요한 경우에만 사용(최적화)
- 언어 선택시도 비슷한 고려사항이 적용(정답 없음)
- 스칼라/자바: 대규모 애플리케이션 / 저수준 API 성능 제어에 이점
- 파이썬/R: 강력한 라이브러리 활용하는 경우에 이점

16.2.3 단위 테스트 프레임워크에 연결하기

- 각 언어 표준 프레임워크(Junit, scalaTest) 사용. 테스트 하네스마다 sparkSession을 생성하고 제거하도록 설정
- 각 프레임워크는 sparkSession 생성과 제거를 수행할 수 있는 메커니즘(before, after 메서드 등) 제공

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

16.2.4 데이터소스 연결하기

- 테스트 코드에서 운영 환경의 데이터소스에 접속하지 말아야 함
- 스파크 구조적 API를 사용하는 경우 지정된 테이블을 이용해 환경을 구성 가능
- 간단히 몇 개의 데미 데이터셋에 이름 붙여 테이블로 등록하고 사용 가능

16.3 개발 프로세스

- 스파크 애플리케이션 개발 프로세스는 기존에 사용하던 개발 흐름과 유사
- 대화형 노트북 및 유사 환경에 초기화된 작업 공간 마련
- 핵심 컴포넌트, 알고리즘을 개발 후 영구적인 영역(라이브러리/패키지 등)으로 옮김
- 노트북을 운영용 애플리케이션(데이터브릭스 등)처럼 실행할 수 있는 도구도 있음
- 로컬 머신에서 실행하는 경우
 - spark-shell과 스파크가 지원하는 다른 언어용 셸을 사용해 애플리케이션 개발에 활용하는 것이 가장 적합
 - spark-shell은 스파크 클러스터에 운영용 애플리케이션 실행 위해 사용

16.4 애플리케이션 시작하기

- 대부분의 스파크 애플리케이션은 spark-submit 명령으로 실행

```
./bin/spark-submit \  
  --class <메인 클래스> \  
  --master <스파크 마스터 URL> \  
  --deploy-mode <배포 모드> \  
  --conf <키>=<값> \  
  ... # 다른 옵션  
  <애플리케이션 JAR 또는 스크립트> \  
  [애플리케이션의 인수]
```

- cluster / client 모드 선택 해야함
- 드라이버와 익스큐터 지연 시간을 줄이기 위해 클러스터 모드로 실행 추천
- 클러스터 장비 중 하나에서 클라이언트 모드로 실행하든가
- spark-submit --help 명령 실행하면 전체 옵션 활용 가능 (or 4이페이지 표 16-1로 확인)

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

- 16.4.1 애플리케이션 시작 예제
- 스파크의 examples 디렉터리에서 다양한 예제와 데모 애플리케이션 확인 가능
- 로컬 머신에서 sparkPi 클래스를 메인 클래스로 사용해 테스트 가능

```
./bin/spark-submit \  
  --class org.apache.spark.examples.SparkPi \  
  --master spark://207.184.161.138:7077 \  
  --executor-memory 20G \  
  --total-executor-cores 100 \  
  replace/with/path/to/examples.jar \  
  1000
```

- 파이썬에서는 아래의 코드로 위 예제와 같은 작업 수행 가능

```
./bin/spark-submit \  
  --master spark://207.184.161.138:7077 \  
  examples/src/main/python/pi.py \  
  1000
```

- master 옵션의 값을 local이나 local[*] (머신의 모든 코드 이용)으로 변경하면 애플리케이션 로컬 모드로 실행 가능
- replace/with/path/to/examples.jar 파일을 로컬에서 사용 중인 스파크 버전에게 맞게 컴파일한 파일로 바꿔야 할 수도 있음

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

16.5 애플리케이션 환경 설정하기

- 이 절은 참고용이니 가볍게 살펴보자. 대부분 설정은 다음과 같이 분류
 - 애플리케이션 속성, 런타임 환경, 셔플 동작 방식, 스파크 UI, 압축과 직렬화, 메모리 관리, 처리 방식, 네트워크 설정, 스케줄링, 동적 할당, 보안, 암호화, 스파크 SQL, 스파크 스트리밍, SparkR
- 스파크에서는 아래와 같이 시스템 설정 가능
 - 스파크 속성은 대부분 애플리케이션 파라미터를 제어(SparkConf)
 - 자바 시스템 속성
 - 하드코딩된 환경 설정 파일
 - /conf 디렉터리에서 템플릿 파일 확인 가능
 - IP 주소 같은 환경변수는 conf/spark-env.sh 스크립트로 머신별로 설정
 - log4j.properties 파일로 로그 관련 설정

16.5.1 SparkConf

- 애플리케이션 모든 설정 관리, SparkConf 객체는 불변성

```
// 스칼라 코드
import org.apache.spark.SparkConf

val conf = new SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")
    .set("some.conf", "to.some.value")

# 파이썬 코드
from pyspark import SparkConf

conf = SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")\
    .set("some.conf", "to.some.value")
```

- SparkConf 객체는 개별 애플리케이션에 대한 속성값을 구성하는 용도로 사용
- 스파크 속성값 -> 스파크 애플리케이션 동작 방식과 클러스터 구성 방식을 제어
- 위의 예제는 로컬 클러스터에 2개 스레드를 생성하도록 설정과 애플리케이션 이름 지정

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

- 이런 설정값은 명령행 인수를 통해 런타임에 구성할 수 있음
- 아래의 예제와 같이 스파크 애플리케이션 포함하는 스파크 셸을 시작할때 도움됨

```
./bin/spark-submit --name "DefinitiveGuide" --master local[4] ...
```

- 시간 주기 형태의 속성값 정의: 25ms, 5s, 10m or 10min, 3h, 5d, 1y로 사용

16.5.2 애플리케이션 속성

- spark-submit 명령이나 스파크 애플리케이션 개발 시 설정 가능

표 16-3 애플리케이션 속성		
속성명	기본값	설명
spark.app.name	(none)	사용자 애플리케이션의 이름을 지정합니다. 이 이름은 스파크 UI와 로그 데이터에서 확인할 수 있습니다.
spark.driver.cores	1	드라이버 프로세스에서 사용할 코어 수를 지정합니다. 단, 클러스터 모드로서만 사용 가능합니다.

- 4040포트 접속 후 스파크 UI에서 확인 가능 (environment 탭)
- spark-defaults.conf, sparkconf, 명령행에서 확인 가능

속성명	기본값	설명
spark.driver.maxResultSize	1g	스파크 액션(예: collect)에 대한 최적화된 값이며 최대 크기. 최종값은 JVM에 의뢰로 설정하는 경우와 동일합니다. 총 값이 크기가 이 제한을 넘어서는 경우에는 값을 초과합니다. 너무 큰 값으로 지정하면 드라이버에서 OutOfMemoryError(spark.driver.memory 설정값과 JVM 메모리 메모리 오버헤드 크기)에 따라 발생할 수 있습니다.)가 발생할 수 있습니다. 그렇기 때문에 적절한 값을 설정해 드라이버에서 OutOfMemoryError가 발생하지 않도록 보장해야 합니다.
spark.driver.memory	1g	SparkContext가 초기화되는 드라이버 프로세스에서 사용할 총 메모리 크기를 지정합니다(예: 1g, 2g). client 모드에서 이 설정을 적용하려면 해당 시스템에 이미 드라이버 JVM이 실행되고 있기 때문에 애플리케이션 구한 시 SparkConf에 직접 설정해야 합니다. 또한 명령행의 --driver-memory 옵션이나 기본 속성 파일에서 지정할 수도 있습니다.
spark.executor.memory	1g	각 엑스큐터 프로세스에서 사용할 메모리의 크기를 지정합니다(예: 2g, 8g).
spark.extraListeners	(none)	SparkListener를 상속받아 구현한 클래스를 클러스터 구동된 목적으로 지정합니다. SparkContext 초기화 시점에 이 클래스의 인스턴스가 생성되어 스코프의 리스너 엔스리Listener bus)에 등록됩니다. 만약 구현한 클래스는 SparkConf를 단일 인자로 사용하는 생성자가 존재하면 그 생성자가 자동으로 호출됩니다. 그렇지 않으면 인수를 사용하지 않는 기본 생성자가 호출됩니다. 만약 유효한 생성자가 없으면 SparkContext 생성에 실패하고 오류가 발생합니다.
spark.logConf	FALSE	SparkContext가 시작될 때 SparkConf에 포함된 정보를 INFO 로 로그 출력합니다.
spark.master	(none)	연결할 클러스터의 매스터를 지정합니다. [표 16-1]의 --master 속성으로 사용 가능한 master URL을 확인할 수 있습니다.
spark.submit.deployMode	(none)	스파크 드라이버 프로그램의 배포 모드를 지정합니다. 'client' 또는 'cluster'를 사용할 수 있습니다. 즉, 드라이버 프로그램을 로컬에서 실행(client)할지, 클러스터의 노드 중 하나에서 원격으로 실행(cluster)할지 지정합니다.
spark.log.callerContext	(none)	Yarn과HDFS 환경에서 실행할 때 Yarn RM 로그나 HDFS 간접 로그에 대해 애플리케이션 정보를 지정합니다. 이 설정값의 값은 하위 설정인 hadoop.caller.context.max.size의 설정값에 따라 달라질 수 있으며 설정값은 간결해야 하며 최대 50자까지 사용할 수 있습니다.
spark.driver.supervise	FALSE	값이 true인 경우 종료 신호가 0이 아닌 다른 값으로 드라이버를 종료합니다. 스파크 스트림 애플리케이션 모드는 예외적으로 모드에서만 사용할 수 있습니다.

- 16.5.3 런타임 속성
- 드물게 런타임 환경 설정하는 경우 있음
- 스파크 공식 문서의 런타임 환경 설정표 참고
- 드파이버/익스큐터를 위한 클래스패스, 파이썬패스, 파이썬 워커 설정
- 로그 관련 속성 정의의 가능

Part 4 - 운영용 애플리케이션

Chapter 16 - 스파크 애플리케이션 개발하기

16.5.4 실행 속성

- 실제 처리를 세밀하게 제어 가능해서 자주 사용됨
- 여기도 공식 문서를 참고하자
- 자주 사용되는 속성은 `spark.executor.cores`, `spark.files.maxPartitionBytes`

16.5.5 메모리 관리 설정

- 최적화를 위해 메모리 옵션을 수동으로 관리 필요(스파크 2.x 버전 이후로 많이 자동화)
- 여기도 공식문서 메모리 관리 표를 참고하자

16.5.6 셔플 동작방식 설정

- 셔플은 과도한 네트워크 부하를 발생시켜 큰 병목 구간이 될 수 있음
- 스파크 공식 문서에서 셔플 동작 방식 표를 참고하자

16.5.7 환경변수

- `conf/spark-env.sh` 에서 스파크 설정 구성 가능
- `standalone`, `mesos` 모드는 파일로 머신에 특화된 정보 제공 가능
- `spark-env.sh` 설정 가능 변수

- **JAVA_HOME**
자바가 설치된 경로를 지정합니다(기본 PATH에 자바 경로가 포함되지 않은 경우).
- **PYSPARK_PYTHON**
PySpark의 드라이버와 워커 모두에서 사용할 파이썬 바이너리 실행 명령을 지정합니다(기본값은 `python2.7`입니다. 만약 `python2.7` 명령을 사용할 수 없다면 `python` 명령을 사용합니다). `spark.pySpark.python` 속성은 `PYSPARK_PYTHON`보다 우선권을 가집니다.
- **PYSPARK_DRIVER_PYTHON**
드라이버에서 PySpark를 사용하기 위해 실행 가능한 파이썬 바이너리를 지정합니다. 기본값은 `PYSPARK_PYTHON`입니다. `spark.pySpark.driver.python` 속성은 `PYSPARK_PYTHON`보다 우선권을 가집니다.
- **SPARK_DRIVER_R**
SparkR 셸에서 사용할 R 바이너리 실행 명령을 지정합니다. 기본값은 `R`입니다. `spark.r.shell.command` 속성은 `SPARK_DRIVER_R`보다 우선권을 가집니다.
- **SPARK_LOCAL_IP**
마스터 IP 주소를 지정합니다.
- **SPARK_PUBLIC_DNS**
스파크 프로그램이 다른 머신에 일련을 호스트명입니다.

- 이외에 코어 수, 최대 메모리 크기 같은 스파크 스탠드얼론 클러스터 설정 관련 옵션도 있음

Part 4 - 운영용 애플리케이션

chapter 16 - 스파크 애플리케이션 개발하기

- | | |
|--|--|
| | <p>16.5.8 애플리케이션에서 잡 스케줄링</p> <ul style="list-style-type: none">- 스파크 스케줄러는 스레드 안정성을 충분히 보장- 기본적으로는 FIFO 방식- 라운드 로빈 방식으로도 구성 가능 (여러 잡이 자원을 공평하게 나눠쓰, 사용자가 많은 환경에 가장 적합)- 페어 스케줄러 지원(여러 개의 잡을 pool로 그룹화하는 방식)<ul style="list-style-type: none">- 개별 풀에 다른 스케줄링 옵션/가중치 설정 가능- 하둡의 페어 스케줄러 모델을 본떠 개발됨 |
|--|--|

Part 4 - 운영용 애플리케이션

chapter 17 - 스파크 배포 환경

- 이 장에서는 인프라 구조를 알아보자

- 클러스터 배포 시 선택사항

- 스파크 지원하는 클러스터 매니저

- 배포 시 고려사항과 배포 환경 설정

- 클러스터를 직접 설정하려면 각 클러스터 매니저의 설정을 잘 알아야 함

- 어떤 클러스터 매니저를 사용하는 것도 중요한 결정 사항

- 클러스터 매니저 근본적 차이점 설명 및 공식사이트 제공 자료를 살펴보자

- 스파크 공식문서는 실행 가능한 예제를 활용해 스파크 애플리케이션 배포 방식을 제공

- 지원 가능한 3가지 클러스터 매니저(standalone, YARN, mesos)

- 클러스터 매니저는 클러스터 머신 유지 관리함. 각가가의 장단점을 이해/사용해야 함

17.1 스파크 애플리케이션 실행을 위한 클러스터 환경

- 크게 2가지로 나뉨 (on-premise cluster, public cloud)

17.1.1 설치형 클러스터 배포 환경

- 자체 데이터센터 운영시 적합, 트레이드 오프를 알아보자

- 장점: 하드웨어 완전 제어 가능. 특정 워크로드 성능 최적화 가능

- 단점:

1. 클러스터 크기 제한적

2. HDFS, 분산 key-value store 등 자체 저장소 시스템 선택/운영

3. 상황에 따라 지리적 복제 및 재해 복구 체계 구축 필요

- 자원 활용 문제 해결을 위한 방법은 클러스터 매니저 사용하는 것

Part 4 - 운영용 애플리케이션

chapter 17 - 스파크 배포 환경

- 클러스터 매니저 활용: 다수의 스파크 애플리케이션 실행/애플리케이션 자원 동적 할당/하나의 클러스터에서 다른 프로그램 실행 가능
- 설치형 클러스터 활용은 여러 종류 저장소 선택 가능

17.1.2 클라우드 배포 환경

- 시간이 지나며 클라우드 환경이 일반적인 스파크 운영 플랫폼으로 자리 잡는 중
- 장점:
 1. 자원을 탄력적으로 늘리고 줄일 수 있음
 2. S3(AWS), Blob(Azure), 구글 클라우드 저장소(GCP)를 저장소로 사용
 3. 연산이 필요한 경우에만 클러스터 비용 지불
- 데이터브릭스는 클라우드 환경을 제공하고 무료 커뮤니티 에디션도 지원

17.2 클러스터 매니저

- 스탠드얼론, 얇은, 메소스를 알아보자

17.2.1 스탠드얼론 모드

- 경량화 플랫폼, 하나의 클러스터에서 다수의 애플리케이션 실행 가능
- 스탠드얼론 클러스터 시작하기
 - 실행할 버전의 스파크를 내려받아 전체 노드에 설치 후 수동 실행
 - 이를 자동화 해주는 스크립트가 있음
- 스크립트를 이용한 스탠드얼론 클러스터 시작하기
 - conf/slaves 파일 생성. slaves 파일에 머신의 호스트명 기록(ssh 활용)

Part 4 - 운영용 애플리케이션

chapter 17 - 스파크 배포 환경

- `$$SPARK_HOME/sbin/start-master.sh`
스크립트를 실행한 마신에서 마스터 인스턴스를 시작합니다.
- `$$SPARK_HOME/sbin/start-slaves.sh`
`conf/slaves` 파일에 명시된 각 마신에서 슬레이브 인스턴스를 시작합니다.

- `$$SPARK_HOME/sbin/start-slave.sh`
스크립트를 실행한 마신에서 슬레이브 인스턴스를 시작합니다.
- `$$SPARK_HOME/sbin/start-all.sh`
마스터 인스턴스를 시작하고 `conf/slaves` 파일에 명시한 각 마신에서 슬레이브 인스턴스를 시작합니다.
- `$$SPARK_HOME/sbin/stop-master.sh`
`bin/start-master.sh` 스크립트로 시작한 마스터 인스턴스를 중지시킵니다.
- `$$SPARK_HOME/sbin/stop-slaves.sh`
`conf/slaves` 파일에 명시한 각 마신에서 슬레이브 인스턴스를 중지시킵니다.
- `$$SPARK_HOME/sbin/stop-all.sh`
마스터 인스턴스를 중지시키고 `conf/slaves` 파일에 명시한 각 마신에서 슬레이브 인스턴스를 중지시킵니다.

애플리케이션 제출하기

- 클러스터 생성하면 마스터 프로세스 UR드를 이용해 마스터 노드나 `spark-submit` 명령으로 마신에서 애플리케이션을 제출

17.2.2 YARN에서 스파크 실행하기

- 스파크는 하둡과 거의 관련이 없습니다. YARN을 지원하지만 하둡이 필요한건 아님
- `spark-submit` 명령의 `--master` 인수를 `yarn`으로 지정해 실행 스파크 잡 실행 가능
- 하둡 YARN은 다양한 실행 프레임워크를 지원하는 통합 스케줄러

애플리케이션 제출하기

- 스파크는 `HADOOP_CONF_DIR / YARN_CONF_DIR` 환경변수로 YARN 파일을 찾아냄
- `spark-submit`에서 YARN 고유 설정 사용 가능(우선순위 큐, 보안관련 `key tab` 파일 제어)

Part 4 - 운영용 애플리케이션

chapter 17 - 스파크 배포 환경

17.2.3 YARN 환경의 스파크 애플리케이션 설정

- 기본 설정 시 참고할 몇 가지 예제와 스파크 애플리케이션 주요 설정을 알아보자

하둡 설정

- 스파크 이용해 HDFS 파일 읽기 위해선 두 개의 하둡 설정 파일을 포함시켜야 함
- hfs-site.xml, core-site.xml.

YARN 애플리케이션 속성

- 하둡 설정과 기능 중 YARN 실행 및 보안 설정은 스파크에 영향을 미침
- 관련 설정은 스파크 공식 문서 YARN 설정 표 참고

17.2.4 메소스에서 스파크 실행하기

- 메소스는 CPU, 메모리, 저장소, 다른 연산 자원을 머신에서 추상화
- 이를 통해 내고장성(fault-tolerant), 탄력적 분산 시스템(elastic distributed system)을 쉽게 구성하고 효과적으로 실행 가능
- 짧게 실행되는 애플리케이션 관리 가능. 또한 데이터센터 규모의 클러스터 매니저 지향
- 스파크에서 지원하는 클러스터 매니저 중 가장 무거움(대규모 메소스 배포 환경일때 사용 추천)

애플리케이션 제출하기

- 메소스를 사용할때는 cluster 모드 방식이 가장 좋음
- client 모드는 분산 자원 관리와 관련된 추가 설정이 필요

메소스 설정하기

- 스파크 공식 문서 메소스 설정 표 참고

Part 4 - 운영용 애플리케이션

chapter 17 - 스파크 배포 환경

17.2.5 보안 관련 설정

- 주로 통신 방식과 관련됨 (인증, 네트워크 구조간 암호화, TLS, SSL 설정 가능)
- 자세한건 스파크 공식 문서에서 보안 설정 표 참고

17.2.6 클러스터 네트워크 설정

- 클러스터 노드 사이에서 proxy를 사용하기 위해 클러스터에 사용자 정의 배포 설정을 적용하는 경우에도 도움이 됨
- 성능을 높이고 싶다면 사용자가 정의한 배포 시나리오에 맞게 적용

17.2.7 애플리케이션 스케줄링

- 스케줄링 기능

1. 각 스파크 애플리케이션은 독립적인 익스큐터 프로세스를 실행
2. 여러 개의 잡(스파크 액션)을 다른 스레드가 제출한 경우 동시 실행 가능
네트워크를 통한 요청에 응답하는 애플리케이션에 적합(페어 스케줄러 기능 제공)

- 단일 클러스터를 공유해 다수의 사용자가 스파크 애플리케이션 실행하면 클러스터 매니저에 자원 할당을 관리할 수 있는 여러 옵션이 있음
- 가장 간단한 방법은 자원을 고정된 크기로 나누는 것
- spark-submit은 특정 애플리케이션 자원 할당을 제어하기 위한 여러 설정값 있음

동적할당

- 하나의 클러스터에서 여러 스파크 애플리케이션 실행하려면 워크로드에 따라 애플리케이션이 점유하는 자원을 동적으로 조정해야 함
- 동적할당: 필요할 때 요청해서 자원 사용하고 반환하는 기능
- 자세한 내용은 역시 스파크 공식 문서 동적 할당 설정표에서 확인

Part 4 - 운영용 애플리케이션

chapter 17 - 스파크 배포 환경

17.3 기타 고려사항

1. 애플리케이션의 개수와 유형

- 클러스터를 확장할 때 연산용 클러스터와 저장소 클러스터를 동시에 확장
- 메소스는 YARN이 가진 기능을 조금 더 개선 + 다양한 애플리케이션 유형 지원
- 메소스는 큰 규모의 클러스터에 적합

2. 다양한 스파크 버전 관리

- 다양한 스파크 버전으로 여러 애플리케이션 실행은 버전별 설정 스크립트 관리에 많은 시간을 들여야 함

3. 클러스터 매니저에 상관없이 애플리케이션 디버깅에 필요한 로고를 기록하는 방식 결정

4. 데일캐트의 메타데이터 관리를 위한 메타스토어 사용을 고려해야함

5. 외부 서플 서비스를 사용해야 할 수도 있음

6. 모니터링 솔루션이 필요

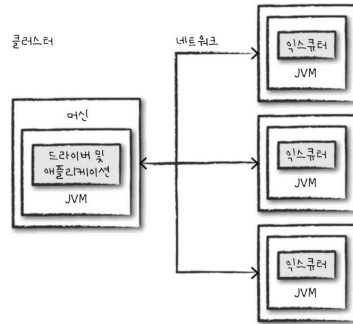
Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

18.1 모니터링 범위

- 모니터링 대상과 필요한 옵션을 알아보자
- 스파크 애플리케이션과 잡
 - 스파크 UI와 스파크 로그 확인 필요(RDD와 쿼리 실행 계획 등 개념적 수준 정보 제공)
- JVM
 - JVM 도구: 스택 트레이스, stack, heap dump, jmap, jstat, jconsole 등이 있음
 - JVM 내부 동작 방식 이해하는데 도움이 됨
 - jvisualvm으로 스파크 잡 동작 특성 확인 가능
- OS와 머신
 - 머신 상태, CPU, 네트워크, I/O 등의 자원 모니터링
- 클러스터
 - YARN, 메소스, 스탠드얼론 클러스터 매니저가 모니터링 대상
 - 유명한 도구로는 강글리아와 프로메테우스가 있음

그림 18-1 모니터링 대상 스파크 애플리케이션 컴포넌트



Part 4 - 운영용 애플리케이션
chapter 18 - 모니터링과 디버깅

- 실행중인 사용자 애플리케이션 프로세스(CPU, 메모리 사용률 등)
- 프로세스 태부의 쿼리 실행 과정(예: 잡과 태스크)

18.2.1 드라이버와 익스큐터 프로세스

- 스파크 지원 모니터링 시스템: 드롭위자드 메트릭 라이브러리 기반 메트릭 시스템
- \$SPARK_HOME/conf/metrics.properties 파일로 구성 가능
- 이러한 메트릭은 다양한 시스템으로 내보낼 수 있음(강글리아 등)

18.2.2 쿼리, 잡, 스테이지, 태스크

- 특정 쿼리에서 무슨일이 일어나는지 알아야 할 때도 있음

18.3. 스파크 로그

- 파이썬은 자바 기반 로깅 라이브러리 사용 못함(logging, print 구문으로 확인)
- 로그 수준 변경 명령: spark.sparkContext.setLogLevel("INFO")

18.4 스파크 UI

- 4040 포트로 실행. 다중-클러스터는 순차적으로 증가(4041, 4042..)

그림 18-2 스파크 UI 전체 탭

Jobs	Stages	Storage	Environment	Executors	SQL
------	--------	---------	-------------	-----------	-----

- Jobs: 스파크 잡에 대한 정보를 제공합니다.
- Stages: 개별 스테이지(스테이지의 태스크를 포함합니다)와 관련된 정보를 제공합니다.
- Storage: 스파크 애플리케이션에 캐싱된 정보와 데이터 정보를 제공합니다.
- Environment: 스파크 애플리케이션의 구성과 설정 관련 정보를 제공합니다.
- Executors: 애플리케이션에서 사용 중인 익스큐터의 상세 정보를 제공합니다.
- SQL: SQL과 DataFrame을 포함한 구조적 API 쿼리 정보를 제공합니다.

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

- 아래 예제는 아래의 코드 실행 후 UI로 실행 과정을 추적 가능

```
# 파이썬 코드
spark.read\
  .option("header", "true")\
  .csv("/data/Spark-The-Definitive-Guide/data/retail-data/all/\
online-retail-dataset.csv")\
  .repartition(2)\
  .selectExpr("instr(Description, 'GLASS') >= 1 as is_glass")\
  .groupBy("is_glass")\

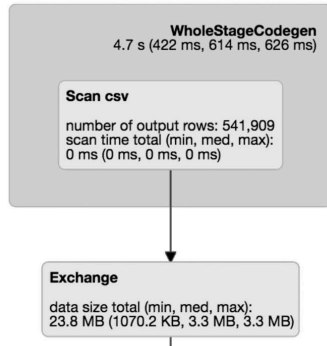
  .count()\
  .collect()
```

- 가장 먼저 볼 태웁은 쿼리에 대한 요약 통계

Submitted Time: 2017/04/08 16:24:41
Duration: 2s
Succeeded Jobs: 2

- 스테이지 별로 보자

그림 18-4 스테이지 1



- wholestagecodegen으로 표시된

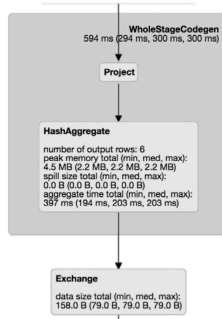
상자는 csv 파일을 모두 스캔

하는 스테이지

- 하단 상자는 파티션 재분배로

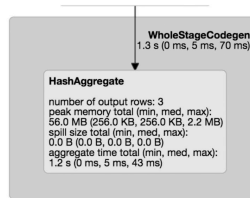
인해 발생하는 셔플 스테이지

그림 18-5 스테이지 2



- 프로젝션 (컬럼 선택/추가/필터링)과 집계 수행
- 6개 로우 반환 확인
- 최종 단계를 위한 준비 작업으로 데이터 샘플하기 전 파티션별로 집계 수행
- 이번 예제에는 해시 기반 집계 수행이므로 파티션별 집계 결과 수와 파티션 수를 곱해 결과로 반환되는 로우 수 계산 가능

그림 18-6 스테이지 3



- 이전 과정에서의 집계 수행
- 두 개의 파티션을 결합해 세 개 로우 반환

18.4.1 스파크 REST API

- REST API로 스파크 상태와 메트릭 확인 가능
- REST API 주소는 localhost:4040/api/v1
- UI와 동일한 정보 제공하지만 SQL 관련 정보는 제공하지 않음

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

18.4.2 스파크 UI 히스토리 서버

- 스파크 UI는 sparkcontext가 실행되는 동안 사용가능
- 종료 후에는 스파크 히스토리 서버 이용
- 이벤트 로그를 저장하도록 스파크 애플리케이션을 설정하면 히스토리 서버를 이용해 스파크 UI와 REST API를 재구성 가능

18.5 디버깅 및 스파크 응급 처치

- 스파크 문제(outofmemoryError 등)와 사용자가 경험할 수 있는 증상(느린 테스트 등)을 포함해 스파크 잡에서 발생할 수 있는 문제를 알아보자

18.5.1 스파크 애플리케이션이 시작되지 않는 경우

징후와 증상

- 스파크 잡이 시작되지 않음
- 스파크 UI가 클러스터 노드 정보를 표시하지 않음
- 스파크 UI가 잘못된 정보를 표시

잠재적 대응법

- 실행에 필요한 자원을 적절하게 설정하지 않았을 때 발생
- 드라이버와 익스큐터 간 통신 오류(IP, 포트 등)
- 클러스터 머신 간 통신할 수 있는지 확인해야 함
- 자원 설정 체크 후 실행하여 정상 동작하는지 확인

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

18.5.2 스파크 애플리케이션 실행 전에 오류가 발생한 경우

징후와 증상

- 명령이 실행되지 않고 오류 메시지 출력
- 스파크 UI에서 잡, 스테이지, 태스크 정보 확인 불가

잠재적 대응법

- 스파크 UI의 Environment 탭에서 애플리케이션 정보가 올바른지 확인
- 코드 상의 문제 확인
- 드라이버, 워커, 저장소 시스템 간 네트워크 연결상태 확인
- 라이브러리 및 클래스패스 확인

18.5.3 스파크 애플리케이션 실행 중에 오류가 발생한 경우

- 다양한 상황에서 발생

징후와 증상

- 스파크 잡이 전체 클러스터에서 성공적으로 실행되지만 이어진 잡에서는 실패
- 여러 단계로 처리되는 쿼리의 특정 단계가 실패
- 어제 정상 작동한 예약 작업이 오늘 실패
- 오류 메시지 해석에 어려움

잠재적 대응법

- 데이터가 존재하는지, 올바른 포맷인지 확인
- 실행 즉시 오류 발생이라면 쿼리 실행 계획을 확인(컬럼명, 뷰, 테이블 등 확인)
- 연관된 컴포넌트 확인하기 위해 스택 트레이스(stack trace) 분석해 단서 찾기

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

- 잡의 태스크가 비정상 종료되면 입력 데이터 자체의 문제가 있을 수 있음
- 데이터 처리하는 코드의 오류

18.5.4 느리게나 뒤회진 태스크

- 최적화 시 혼하게 발생
- 작업이 균등 분배되지 않거나(skew, 데이터 치우침) 특정 머신이 다른 머신에 비해 처리 속도가 느린 경우(예: 하드웨어 문제)

징후와 증상

- 대부분의 태스크가 정상 실행되고 소수의 태스크가 오랫동안 실행
- 스파크 UI에서 위와 같은 문제를 확인할 수 있고 동일한 데이터셋을 다루는 경우
- 여러 스테이지에서 번갈아가며 2번째 증상과 같은 현상이 발생
- 머신 수를 늘려도 상황 개선이 되지 않고 특정 태스크가 훨씬 오래 실행
- 스파크 메트릭에서 특정 익스큐터가 다른 익스큐터에 비해 훨씬 많은 데이터 I/O 발생

잠재적 대응법

- 파티션 별 데이터 양을 줄이기 위해 파티션 수를 증가
- 다른 컬럼을 조합해 파티션 재분배
- 익스큐터 메모리 증가시킴
- 익스큐터에 문제 있는지 모니터링하고 해당 문제가 다른 잡에서 발생하는지 확인
- 조인/집계시문제라면 18.5.5 느린 집계 속도나 18.5.6 느린 조인 속도 내용 참조
- 비즈니스 로직에 쓸모없는 부분 확인하고 가능하면 DataFrame 코드로 변환
- UDF, DAF가 적당한 크기의 데이터를 사용해 실행하는지 확인

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

- 투기적 실행(speculative execution)을 사용하면 느린 태스크의 복제본 실행으로 데이터 중복 (17장 참조)
- dataset은 레코드를 사용자 정의 함수의 자바 객체로 변환하기 위해 수많은 객체 생성하므로 가비지 컬렉션이 빈번하게 발생

18.5.5 느린 집계 속도

징후와 증상

- group by 호출 시 느린 태스크 발생
- 집계 처리 이후 잡도 느림

잠재적 대응법

- 집계 연산 전 파티션 수를 증가시켜 태스크별로 처리
- 익스큐터 메모리를 증가시킨 경우 데이터가 많은 키를 처리하는 익스큐터는 느림
- 집계 처리 후 이어 실행되는 태스크가 느린 경우 데이터셋에 불균형 현상(repartition 명령을 추가해보자)
- 모든 필터와 SELECT 구문이 집계 연산보다 먼저 처리된다면 필요한 데이터 이용해서 집계 연산 수행
- null 값을 나타내기 위해 empty 같은 값을 사용하는지 확인
- 일부 집계 함수는 다른 함수에 비해 태생적으로 느림

18.5.6 느린 조인 속도

- 조인 스테이지의 처리 시간이 오래 걸림
- 조인 전후의 스테이지는 정상 동작

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

잠재적 대응법

- 조인 타입을 변경해 최적화 가능
- 조인 순서 변경하며 잡의 처리 속도가 올라가는지 테스트
- 조인 수행 전 데이터셋 분할하여 클러스터 노드 간 데이터 이동을 줄임
- 데이터 치우침 현상은 느린 조인을 유발
- 모든 필터와 SELECT 구문이 조인 연산보다 우선 처리되면 필요한 데이터만 조인
- null 값을 제어하기 위해 empty와 같은 값으로 대체하는지 확인
- 입력 DataFrame이나 테이블 통계가 없는 경우 브로드캐스트 조인을 사용하는 실행 계획을 생성하지 못함 조인 대상 테이블 중 하나가 작은 경우 강제 브로드캐스팅 or 스파크 통계 수집 명령을 사용해 테이블을 분석

18.5.7 느린 읽기와 쓰기 속도

- 느린 I/O는 진단이 어려울 수 있음

징후와 증상

- 분산파일 시스템이나 외부 시스템의 데이터를 읽는 속도가 느림
- 네트워크 파일 시스템이나 blob 저장소에 데이터를 쓰는 속도가 느림

잠재적 대응법

- 스파크 투기적 실행(spark.speculation 속성을 True로 설정)을 사용하면 느린 읽기와 쓰기 속도를 개선하는데 도움
- 스파크 클러스터와 저장소 시스템 간 네트워크 대역폭이 충분하지 않을 수 있음
- 단일 클러스터의 노드마다 스파크와 분산 파일 시스템 모두 동일한 호스트 명을 인식하는지 확인

18.5.8 드라이버 outOfMemoryError 또는 응답 없음

- 드라이버 outOfMemoryError는 스파크 애플리케이션이 비정상 종료되므로 매우 심각한 문제

징후와 증상

- 스파크 애플리케이션이 응답하지 않거나 비정상종료
- 드라이버 로그에 outOfMemoryError 또는 가비지 컬렉션 관련 메시지 출력
- 명령이 장시간 실행되거나 실행되지 않음
- 반응이 거의 없음
- 드라이버 JVM의 메모리 사용량이 많음

잠재적 대응법

- 원인이 다양하기 때문에 진단하기 쉽지 않음
- collect 메서드 같은 연산을 실행해 큰 데이터셋을 드라이버에 전송하려고 시도했을 수 있음
- 브로드캐스트하기 큰 데이터를 브로드캐스트 조인에 사용했을 수 있음
- 장시간 실행되는 애플리케이션은 드라이버에 많은 양의 객체를 생성해 해제하지 못할 수 있음
- 드라이버 가용 메모리를 증가
- JVM 메모리 부족 현상은 파이썬과 같은 다른 언어를 함께 사용하는 경우 발생
- SQL JDBC 서버와 노트북 환경으로 다른 사용자와 sparkContext를 공유하는 상황이라면 여러 사용자가 동시에 대량의 데이터를 드라이버 메모리로 전송할 수 있는 명령을 실행하지 못하게 막아버려야함

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

18.5.9 익스큐터 outofMemoryError 또는 응답 없음

- 스파크 애플리케이션에서 자동으로 복구 가능

징후와 증상

- 익스큐터 로그에 outofMemoryError 또는 가비지 컬렉션과 관련 메시지가 출력
- 익스큐터 비정상 종료 및 응답 없음
- 특정 노드의 느린 태스크가 복구되지 않음

잠재적 대응법

- 익스큐터 가용 메모리와 익스큐터 수 증가
- 관련 파이썬 설정을 변경해 PySpark의 워커 크기 증가
- 익스큐터 로그에 가비지 컬렉션 오류 메시지가 발생했는지 확인
- null 값을 제거하기 위해 empty 값이 있는지 확인
- RDD와 Dataset은 객체를 생성하기 때문에 문제가 발생할 가능성이 큼
- 자바의 jmap 도구를 사용해 익스큐터 힙 메모리 히스토그램을 확인
- 캐בל류 스토어 같이 다른 워크로드를 처리하는 노드에 익스큐터가 위치한다면 스파크 잡을 다른 작업과 분리해야 함

18.5.10 의도하지 않은 null 값이 있는 결과 데이터

징후와 증상

- 트랜스포메이션이 실행된 결과에 의도치 않은 null 값이 발생
- 잘 동작하던 운영 환경의 예약 작업이 동작하지 않거나 정확한 결과를 생성 못함

Part 4 - 운영용 애플리케이션

chapter 18 - 모니터링과 디버깅

잠재적 대응법

- 데이터 포맷 변경되었는지 확인
- 어큐뮬레이터 활용해 레코드나 특정 데이터 타입의 수를 확인
- 트랜스포메이션이 실제 유효한 쿼리 실행 계획을 생성하는지 확인. 암시적 형변환을 수행하는 경우 혼란스러운 결과가 반환될 수 있음

18.5.11 디스크 공간 없음 오류

징후와 증상

- no space left on disk 오류 메시지 발생

잠재적 대응법

- 더 많은 디스크 공간 확보
- 데이터 파티션 재분배
- 몇가지 저장소 설정 실험 (로그 유지 기간 지정 등)
- 오래된 로그 파일과 셔플 파일을 수동으로 제거

18.5.12 직렬화 오류

징후와 증상

- 직렬화 오류와 함께 잡 실패

잠재적 대응법

- UDF나 RDD를 이용해 개발된 로직을 수행하는 익스큐터 확인
- 자바나 스칼라 클래스에서 UDF를 생성시 인클로징 객체 필드를 참조하지 말 것