스파크로
구축하는 분산처리
빅데이터 플랫폼

Agenda

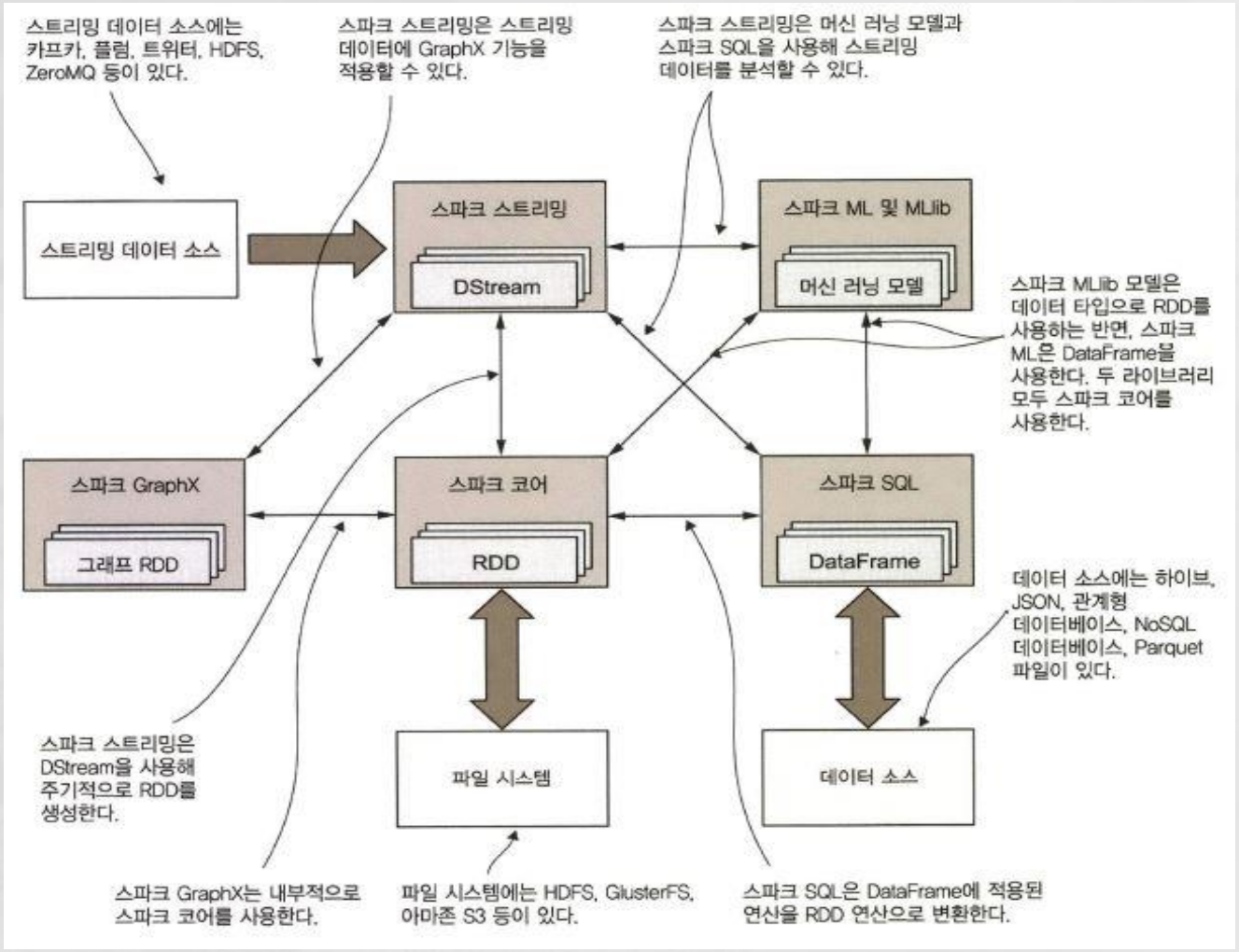
1. Spark기초와 BigData Platform 개념
2. Spark를 이용한 빅데이터 플랫폼 구축
3. 효율적인 데이터 분석을 위한 Hadoop Ecosystem
4. 빠른 데이터 분석을 위한 Spark
5. 실전 빅데이터 분석 프로젝트

1.Spark 기초와 빅데이터 플랫폼의 개념 및 설계

1.1 스파크란

1. 스파크 플랫폼

스파크 구성



1.스파크란

스파크란

- 아파치 하둡은 분산 컴퓨팅용 자바 기반 오픈소스 프레임워크로 하둡 분산 파일 시스템 (Hadoop Distributed File System, HDFS)과 맵리듀스 처리 엔진으로 구성
- 스파크는 범용 분산 컴퓨팅 플랫폼이라는 점에서 하둡과 유사하지만 , 대량의 데이터를 메모리에 유지하는 독창적인 설계로 계산 성능 개선
- 스파크 프로그램은 맵리듀스보다 약 100배 더 빠른 속도로 같은 작업을 수행

1.스파크란

스파크란

- 아파치 하둡은 분산 컴퓨팅용 자바 기반 오픈소스 프레임워크로 하둡 분산 파일 시스템 (Hadoop Distributed File System, HDFS)과 맵리듀스 처리 엔진으로 구성
- 스파크는 범용 분산 컴퓨팅 플랫폼이라는 점에서 하둡과 유사하지만, 대량의 데이터를 메모리에 유지하는 독창적인 설계로 계산 성능 개선
- 스파크 프로그램은 맵리듀스 보다 약 100배 더 빠른 속도로 같은 작업을 수행
- 스파크는 오픈 소스로 공개했지만, 데이터브릭스는 아파치 스파크 개발을 주도하며 스파크 코드의 75% 이상을 기여 또는 데이터브릭스 클라우드(Databricks Cloud)라는 스파크 기반의 빅데이터 분석 솔루션을 상용

1.스파크란

스파크란

- 스파크는 파이썬과 자바,스칼라, 최근에는 R 언어까지 지원해 사용자를 광범위하게 포용
- 전통적으로 파이썬과 R을 선호하는 학계, 여전히 많은 사용자를 보유한 자바 커뮤니티, 자바 가상 머신(Java Virtual Machine, JVM)에서 함수형 프로그래밍 방식을 지원하며 점차 대중화되는 스칼라 사용자도 스파크를 활용
- 스파크는 맵리듀스와 유사한 일괄 처리 기능,실시간 데이터 처리 기능, SQL과 유사한 정형 데이터 처리 기능, 그래프 알고리즘, 머신 러닝 알고리즘을 모두 단일 프레임워크와 통합

1.스파크란

스파크란

- HDFS와 맵리듀스 처리 엔진으로 구성된 하둡 프레임워크는 분산 컴퓨팅을 최초로 대중화하는데 성공
- 병렬 처리(parallelization): 전체 연산을 잘게 나누어 동시에 처리하는 방법
- 데이터 분산(distribution): 데이터를 여러 노드로 분산하는 방법
- 장애 내성(fault tolerance) ; 분산 컴포넌트의 장애에 대응하는 방법

1.스파크란

스파크란

- 스파크에서는 분산 아키텍처 때문에 처리 시간에 약간의 오버해드(overhead)가 필연적으로 발생
- 대량의 데이터를 다 룰 때는 오버헤드가 무시할 수 있는 수준이지만, 단일머신에서도 충분히 처리할 수 있는 데이터 셋을 다룰 때는 작은 데이터셋의 연산에 최적화된 다른 프레임워크를 사용
- 스파크는 온라인 트랜잭션 처리(OnLine Transaction Processing, OLTP) 애플리케이션을 염두에 두고 설계되지 않음
- 대량의 원자성(atomicity) 트 랜잭션을 빠르게 처리해야 하는 작업에는 스파크가 적합하지 않음

1.스파크란

맵리듀스의 한계

- 맵리듀스 잡의 결과를 다른 잡 에서 사용하려면 먼저 이 결과를 HDFS에 저장해야 함
- 맵리듀스는 이전 잡의 결과가 다음 작업의 입력이 되는 반복 알고리즘에는 본질적으로 맞지 않음
- 많은 유형의 문제가 맵리듀스 2단계 패러다임에 쉽게 들어맞지 않으며 , 모든 문제를 일련의 맵과 리듀스 연산으로 분해할 수는 없음
- 일부 문제에서 맵리듀스 API를 사용하기 다소 어려울 때도 많음
- 스파크는 하둡의 이러한 문제들을 상당 부분 해결했음

1.스파크란

스파크의 핵심

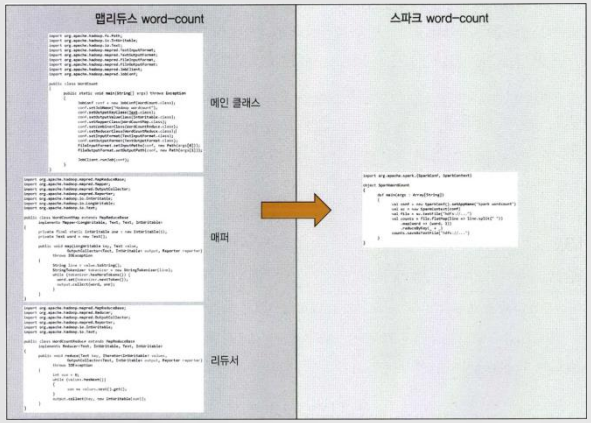
- 스파크의 핵심은 맵리듀스처럼 잡에 필요한 데이터를 디스크에서 매번 가져오는 대신, 데이터를 메모리에 캐시로 저장하는 인-메모리 실행 모델
- 스파크는 동일한 작업을 맵리듀스보다 최대 100배더 빠르게 실행할 수 있음
- 맵리듀스로 구현하려면 각 단계의 계산 결과를 디스크(즉,HDFS)에 저장
- 2단계에서는 1단계 결과를, 3단계에서는 2단계 결과를 각각 디스크에서 읽어 들임
- 스파크에서는 모든 정점 간의 최단 경로를 찾은 후 이 결과를 메모리에 캐시할 수 있음

1.스파크란

스파크의 핵심

- 3단계에서는 최종 캐시된 데이터에서 가장 먼 정점과의 거리가 가장 짧은 정점을 찾을 수 있음
- 따라서 데이터를 매번 디스크에 읽고 쓰는 맵리듀스보다 스파크의 성능이 더 좋을 가능성이 높음
- 스파크 API는 맵리듀스 API보다 훨씬 사용하기가 쉬움

```
val spark = SparkSession.builder().appName("Spark wordcount")
val file = spark.sparkContext.textFile("hdfs:// ... ")
val counts = file.flatMap(line => line.split(" "))
     .map(word => (word, 1)).countByKey()
counts.saveAsTextFile("hdfs:// ... ")
```



1.스파크란

스파크의 핵심

- 스파크는 스칼라, 자바, 파이썬, R을 아우르는 다양한 프로그래밍 언어를 지원해 더 많은 사용자를 포용
- 스파크가 제공하는 대화형 콘솔인 스파크 셸(또는 스파크 REPL(Read-Eval-Print Loop))을 이용
- 스파크 자체(standalorie) 클러스터, 하둡의 YARN(Yet Another Resource Negotiator) 클러스터, 아파치 메소스(Mesos) 클러스터 등 다양한 유형의 클러스터 매니저를 사용해 스파크를 실행

1.스파크란

스파크의 통합 플랫폼

- 스파크는 하둡 생태계의 여러 도구가 제공한 다양한 기능을 플랫폼 하나로 통합
- 스파크의 범용적인 실행 모델을 사용해 실시간 스트림 데이터 처리, 머신 러닝, SQL 연산, 그래프 알고리즘,일괄 처리 등 여러 종류의 프로그램을 단일 프레임워크에서 구현할

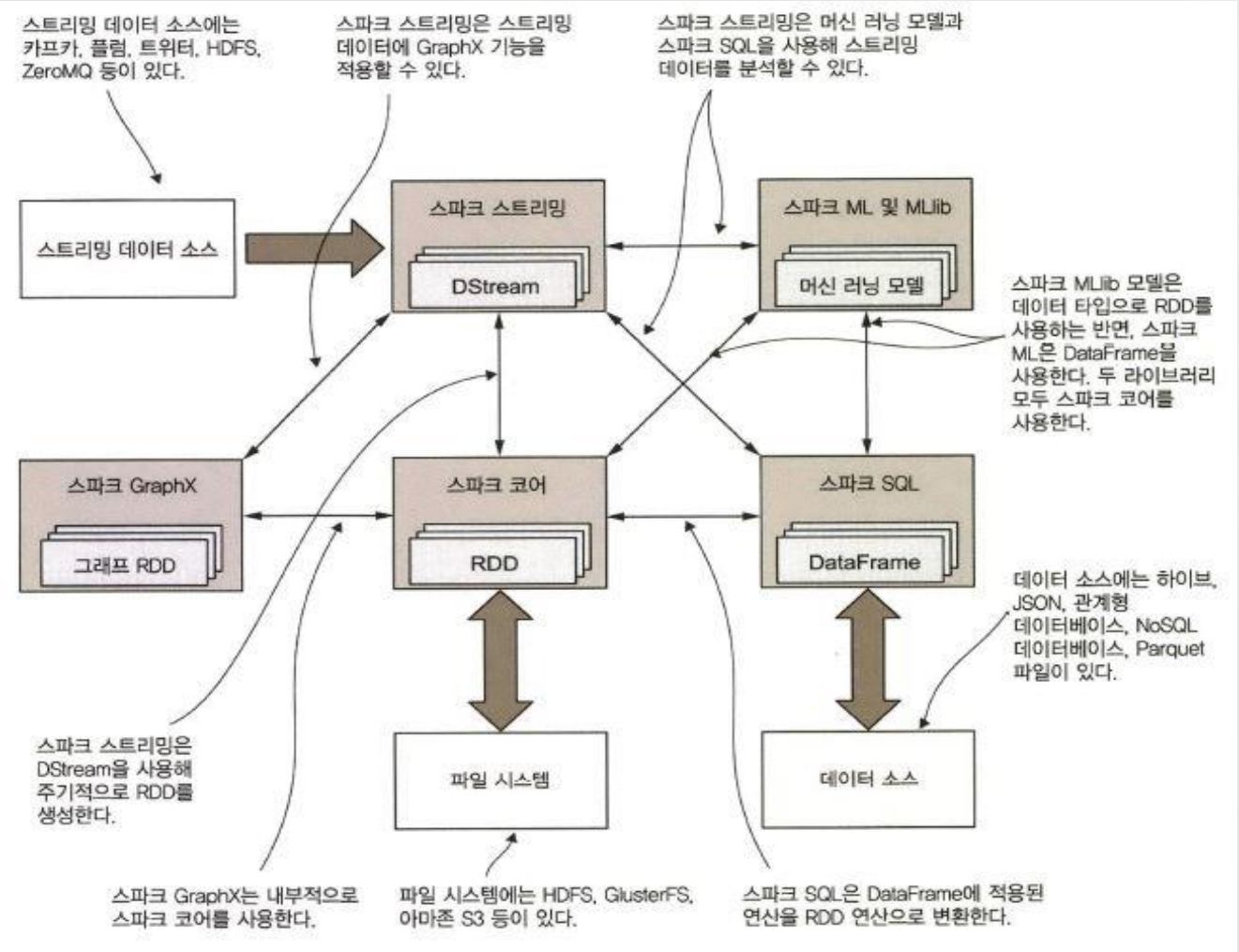
1.스파크란

스파크의 과제

- 스파크는 일괄 분석을 염두에 두고 설계했기 때문에 공유된 데이터를 비동기적으로 갱신하는 연산(예: 온라인 트랜잭션 처리 등)에는 적합하지 않음
- 실시간 데이터를 처리하는 스파크 스트리밍은 단순히 타임 윈도로 분할한 스트림 데이터에 일괄 처리를 적용한 것
- 스파크는 잡(job)과 태스크를 시작하는 데 상당한 시간을 소모하기 때문에 대량의 데이터를 처리하는 작업이 아니라면 굳이 스파크를 사용할 필요가 없음
- 소량의 데이터를 처리할 때는 스파크 같은 분산 시스템보다 간단한 관계형 데이터베이스나 잘 짜인 스크립트가 훨씬 더 빠름

2.스파크를 구성하는 컴포넌트

스파크 컴포넌트 구성



2.스파크를 구성하는 컴포넌트

스파크 코어

- 스파크 코어는 스파크 잡과 다른 스파크 컴포넌트에 필요한 기본 기능을 제공
- 스파크 API의 핵심 요소인 RDD(Resilient Distributed Dataset)
- RDD는 분산 데이터 컬렉션(즉, 데이터셋)을 추상화한 객체로 데이터셋에 적용할 수 있는 연산 및 변환 메서드를 함께 제공
- RDD는 노드에 장애가 발생해도 데이터셋을 재구성할 수 있는 복원성을 갖춤

2.스파크를 구성하는 컴포넌트

스파크 코어

- 스파크 코어는 HDFS, GlusterFS, 아마존 S3 등 다양한 파일 시스템에 접근할 수 있음
- 공유 변수(broadcast variable)와 누적 변수(accumulator)를 사용해 컴퓨팅 노드 간에 정보를 공유할 수 있음
- 스파크 코어에는 네트워킹, 보안, 스케줄링 및 데이터 셔플링(shuffling) 등 기본 기능 이 구현

2.스파크를 구성하는 컴포넌트

스파크 SQL

- 스파크 SQL은 스파크와 하이브 SQL(HiveQL) 이 지원하는 SQL을 사용해 대규모 분산 정형 데이터를 다룰 수 있는 기능을 제공
- 스파크 버전 1.3에 도입된 DataFrame과 스파크 버전 1.6에 도입된 Dataset은 정형 데이터의 처리를 단순화하고 성능을 크게 개선
- 파일, Parquet 파일(데이터와 스키마를 함께 저장할 수 있는 파일 포맷으로 최근 널리 사용), 관계형 데이터 베이스 테이블, 하이브 테이블 등 다양한 정형 데이터를 읽고 쓰는 데도 스파크 SQL을 사용할 수 있음

2.스파크를 구성하는 컴포넌트

스파크 SQL

- 스파크 SQL은 DataFrame과 Dataset에 적용된 연산을 일정 시점에 RDD 연산으로 변환해 일반 스파크 잡으로 실행함
- 스파크 SQL은 카탈리스트(Catalyst)라는 쿼리 최적화 프레임워크 를 제공하며, 사용자가 직접 정의한 최적화 규칙을 적용해 프레임워크를 확장할 수 있음
- BI(Businccss Intelligence) 도구 등 외부 시스템과 스파크를 연동할 수 있는 아파치 쓰리프트(Thrift) 서버도 제공
- 외부 시스템은 기존JDBC 및 ODBC 프로토콜을 이용해 스파크 SQL 쿼리를 실행

2.스파크를 구성하는 컴포넌트

스파크 스트리밍

- 스파크 스트리밍은 다양한 데이터 소스에서 유입되는 실시간 스트리밍 데이터를 처리하는 프레임워크
- 스파크가 지원하는 스트리밍 소스에는 HDFS, 아파치 카프카(Kafka, 아파치 플럼 (Flume), 트위터(.Twitter), ZeroMQ
- 스파크 스트리밍은 장애가 발생하면 연산 결과를 자동으로 복구(스트리밍 데이터를 처리할 때는 이러한 장애 복원성이 매우 중요)
- 스파크 스트리밍은 이산 스트림(Discretized Stream, DStream) 방식으로 스트리밍 데이터를 표현하는데 , 가장 마지막 타임 윈도 안에 유입된 데이터를 RDD로 구성해 주기적으로 생성

2.스파크를 구성하는 컴포넌트

스파크 스트리밍

- 스파크 스트리밍과 다른 스파크 컴포넌트를 단일 프로그램에서 사용해 실시간 처리 연산과 머신 러닝 작업, SQL 연산, 그래프 연산 등을 통합
- 정형 스트리밍 API를 새롭게 도입해 마치 일괄 처리 프로그램을 구현하는 것처럼 스트리밍 프로그램을 구현

2.스파크를 구성하는 컴포넌트

스파크 MLlib

- 스파크 MLUb는 UC 버클리의 MLbase 프로젝트에서 개발한 머신 러닝 알고리즘 라이브러리
- 스파크 MLlib는 로지스틱 회귀(logistic regression), 나이브 베이즈 분류(naive Bayes classification), 서포트 벡터 머신(Support Vector Machine, SVM), 의사 결정 트리(decision tree), 랜덤 포레스트 (random forests), 선형 회귀(linear regression), k_평균 군집화(k-means clustering) 등 다양한 머신 러닝 알고리즘을 지원
- 스파크 MLlib를 사용해 RDD 또는 DataFrame의 데이터셋을 변환하는 머신 러닝 모델을 구현 할 수 있음

2.스파크를 구성하는 컴포넌트

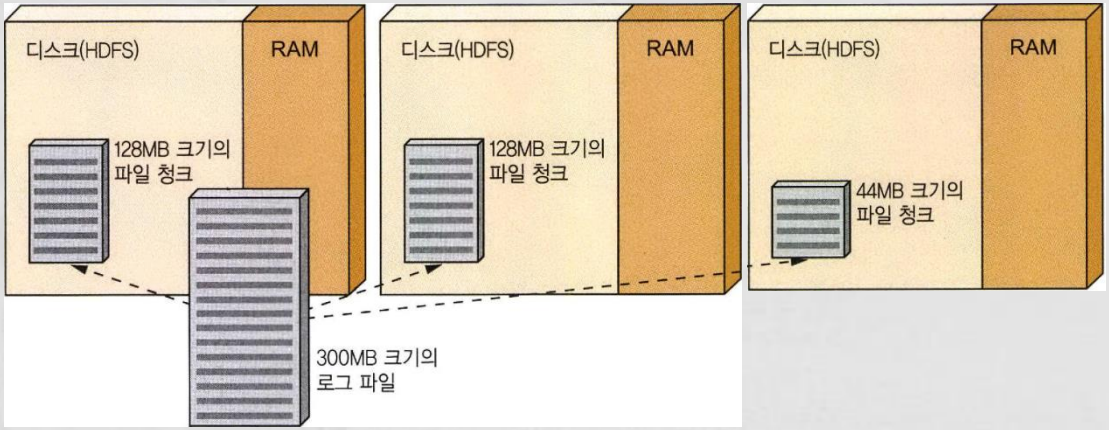
스파크 GraphX

- 그래프(graph)는 정점과 두 정점을 잇는 간선으로 구성된 데이터 구조
- 스파크 GraphX는 그래프 RDD(EdgeRDD 및 VertexRDD) 형태의 그래프 구조를 만들 수 있는 다양한 기능을 제공
- GraphX에는 페이지랭크(page rank), 연결요소(connected components), 최단 경로(shortest: path) 탐색, SVD++(Singular Value Decomposition++) 등 그래프 이론에서 가장 중요한 알고리즘이 구현
- 지라프(Giraph)(하둡에서 그래프 알고리즘을 실행할 수 있도록 지원하는 아파치 프로젝트)에 구현된 대규모 그래프 처리 및 메시지 전달 API인 프리겔(Pregd)도 동일하게 제공

3.스파크 프로그램의 실행

스파크 프로그램 실행 과정

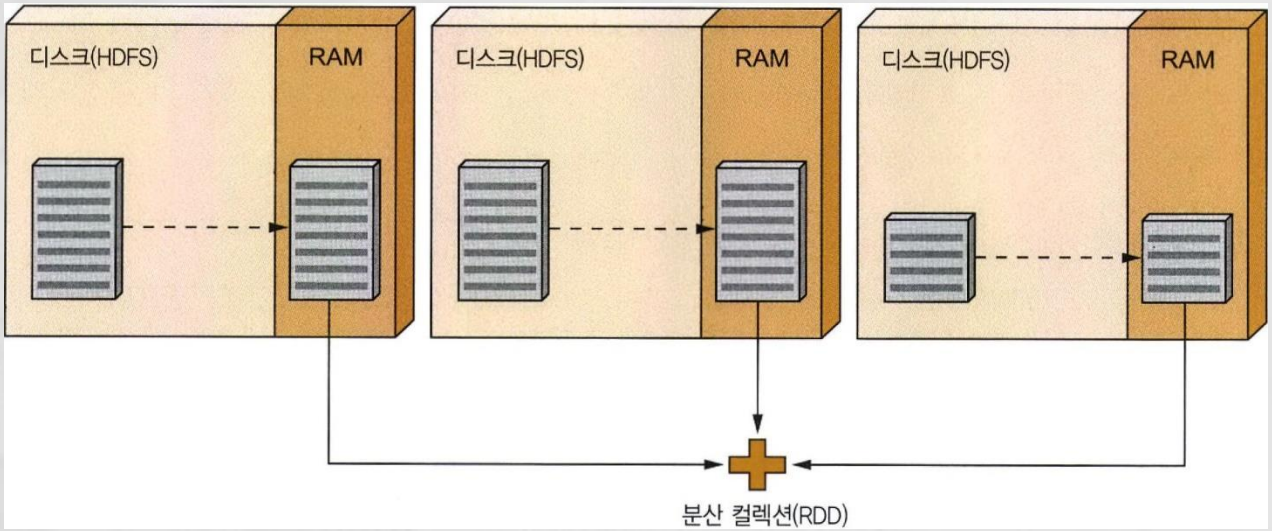
- 300MB 크기의 로그 파일이 노 드 세 개로 구성된 HDFS 클러스터에 분산 저장되어 있다고 가정
- HDFS는 이 파일을 자동으로 128MB 크기의 청크(chunk)로 분할하고(하둡에서는 블록(block) 이라는 용어를 사용)
- 각 블록 을 클러스터의 여러 노드에 나누어 저장
- YARN에서 스파크를 실행하며, YARN 또한 스파크와 동일한 하둡 클러스터 에서 실행한다고 가정



3.스파크 프로그램의 실행

스파크 프로그램 실행 과정

- val lines = sc.textFile("hdfs://path/to/the/file")
- 스파크는 데이터 지역성(data locality)을 최대한 달성하려고 로그 파일의 각 블록이 저장된 위치를 하둡에게 요청한 후, 모든 블록을 클러스터 노드의 RAM 메모리로 전송



3.스파크 프로그램의 실행

스파크 프로그램 실행 과정

- 데이터 전송이 완료 되면 스파크 셸에서 RAM에 저장된 각 블록(이를 스파크 용어로 파티션(partition))을 참조할 수 있음
- 이 블록, 즉 파티션의 집합이 바로 RDD가 참조하는 분산 컬렉션이며, 이 컬렉션에는 분석해야 할 로그 파일 줄(line)이 저장됨
- RDD를 사용하면 비-분산(non-distributed) 로컬 컬렉션을 처리하는 것과 같은 방식으로 대규모 분산 컬렉션을 다룰 수 있음

3.스파크 프로그램의 실행

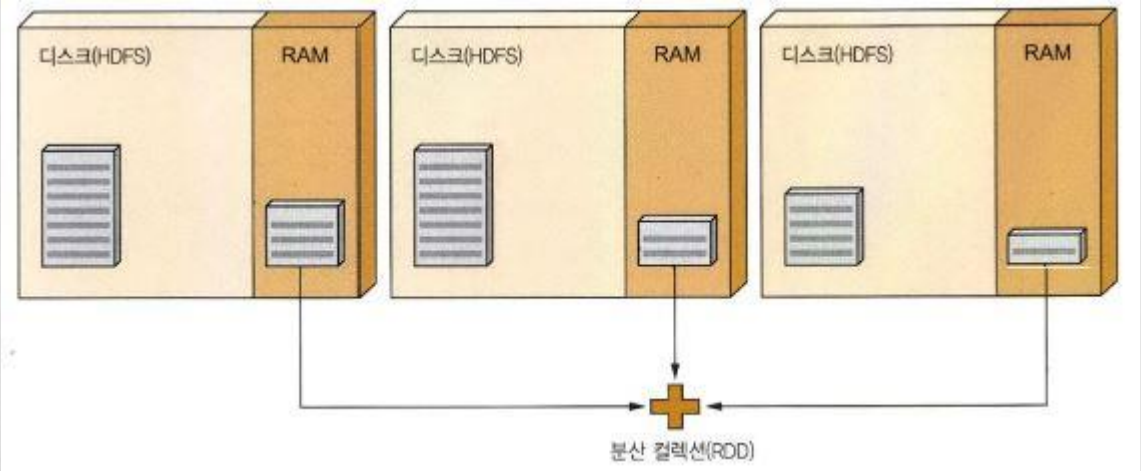
스파크 프로그램 실행 과정

- 스파크는 자동화된 장애 내성과 데이터 분산 기능 외에도 RDD의 컬렉션에 함수형 프로그래밍을 사용할 수 있는 정교한 API를 제공
- RDD API를 사용해 RDD의 컬렉션을 필터링하고, 사용자 정의 함수로 컬렉션을 매핑하고, 누적 값 하나로 리듀스하고, 두 RDD를 서로 빼거나 교차하거나 결합하는 등 다양한 작업을 실행할 수 있음

3.스파크 프로그램의 실행

스파크 프로그램 실행 과정

- val oomLines = lines.filter(l => l.contains("OutOfMemoryError")).cache()
- 컬렉션의 필터링이 완료되면 oomLines RDD에는 분석에 필요한 데이터만 포함됨
- RDD에 cache 함수를 호출해서 추후 다른 잡을 수행할 때도 RDD가 메모리에 계속 유지되도록 지정됨



3.스파크 프로그램의 실행

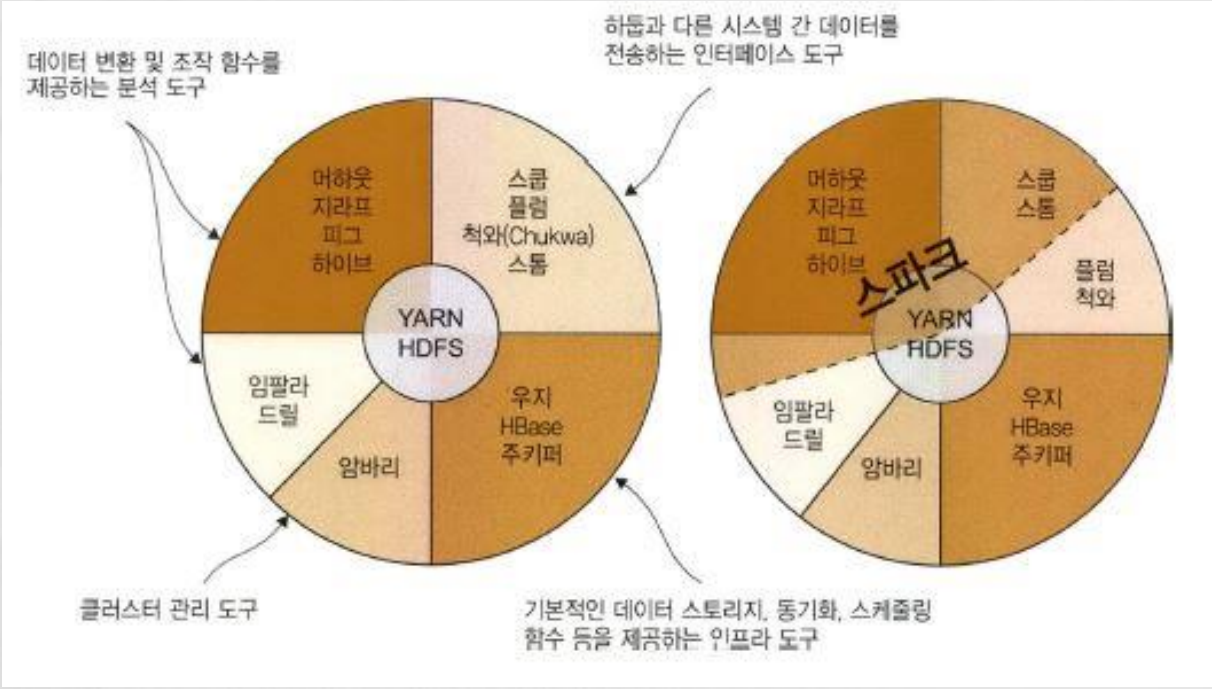
스파크 프로그램 실행 과정

- val result = oomLines.count()
- 필터링된 로그에 남은 줄 개수를 센다.
- oomLines 객체를 캐시에 저장했으므로 스파크는 HDFS의 로그 파일을 다시 로드하는 대신 캐시의 데이터를 재사용

4.스파크 생태계

스파크 생태계

- 하둡 생태계는 인터페이스 도구,분석 도구,클러스터 관리 도구,인프라 도구로 구성
- 아파치 지라프는 스파크 GraphX로 대체할 수 있고, 아파치 머하웃은 스파크 MLlib로 대체



4.스파크 생태계

스파크 생태계

- 아파치 스톰(Storm)은 스파크 스트리밍과 기능이 상당 부분 겹치기 때문에 대부분은 스톰 대신 스파크 스트리밍을 사용
- 아파치 피그(Pig)와 아파치 스쿱(Sqoop)은 스파크 코어와 스파크 SQL이 같은 기능을 지원
- 아파치 우지(Oozie), HBase, 주키퍼(Zookeeper) 등 하둡 생태계의 인프라 및 관리 도구들은 스파크로 대체할 수 없음
- 우지는 다른 여러 유형의 하둡 잡을 스케줄링하는 데 사용하며 스파크 잡도 스케줄링할 수 있음

4.스파크 생태계

스파크 생태계

- HBase는 스파크와는 완전히 다른 성격의 대규모 분산 데이터베이스
- 주키퍼는 분산 애플리케이션에 필요한 분산 동기화, 네이밍(nammg), 그룹 서비스 프로비저닝 (provisioning) 등 공통 기능을 견고하게 구현한 고성능 코디네이션 서비스(하둡 외 다른 여러 분산 시스템에서도 널리 활용)
- 아파치 임팔라(Impala)와 드릴(Drill)은 스파크와 공존할 수 있음
- 스파크를 반드시 YARN에서 실행할 필요가 없다는 점
- YARN 대신 사용할 수 있는 스파크의 클러스터 매니저로 아파치 메소스와 스파크 자체 클러스터가 있음

4.스파크 생태계

스파크 생태계

- 아파치 메소스는 분산 리소스를 추상화한 고급 분산 시스템 커널
- 메소스 클러스터는 장애 내 성을 유지하면서도 클러스터 노드를 수만 개로 늘릴 수 있는 확장성을 갖추었음
- 스파크 자체 클러스터는 스파크의 전용 클러스터 매니저로 오늘날 여러 애플리케이션의 운영 환경에 활용 함

1.Spark 기초와 빅데이터 플랫폼의 개념 및 설계

1.2 스파크 기초

1.스파크 프로그램 작성

LICENSE파일에서 BSD 문자열 찾기

- scala> val licLines = sc.textFile("/opt/spark/LICENSE")
  - 스파크 API를 사용해 LICENSE 파일을 읽어 들이고, 이 파일에 포함된 줄 개수를 세기
- scala> val lineCnt = licLines.count
- scala> val bsdLines = licLines.filter(line => line.contains("BSD"))
  - licLines 컬렉션에 필터(filter)를 적용해 BSD를 포함하지 않는 줄은 컬렉션에서 제외하는 것
  - 사용한 filter 함수는 licLines 컬렉션의 각 요소(파일의 각 줄)를 굵은 화살표 로 정의한 익명 함수에 전달하고, 익명 함수가 true로 판별한 요소만으로 구성된 새로운 컬렉션 (bsdLines)을 반환함
- Scala> bsdLines.foreach(println)

1.스파크 프로그램 작성

LICENSE파일에서 BSD 문자열 찾기

- 기명 함수function)를 정의해 파일의 줄을 필터링
- scala> def isBSD(line: String) = { line.contains("BSD") }
- scala> val isBSD = (line: String) => line.contains("BSD")
- scala> val bsdLines1 = licLines.filter(isBSD)
- scala> bsdLines1.count
- scala>bsdLines1.foreach(bLine => println(bLine))
- scala> bsdLines1.foreach(println)

1.스파크 프로그램 작성

RDD의 개념

- LicLines와 bsdLines는 RDD라는 스파크 전용 분산 컬렉션
- RDD는 스파크의 기본 추상화 객체로 다음 성질이 있음
  - 불변성(immutable): 읽기 전용(read-only)
  - 복원성(resilient): 장애 내성
  - 분산(distributed): 노드 한 개 이상에 저장된 데이터셋
- RDD는 데이터를 조작할 수 있는 다양한 변환 연산자를 제공하지만, 변환 연산자는 항상 새로운 RDD 객체를 생성함
- 생성된 RDD는 절대 바뀌지 않는 불변의 성질이 있음

1.스파크 프로그램 작성

RDD의 개념

- 스파크는 불변 컬렉션을 사용해 분산 시스템에서 가장 중요한 장애 내성을 직관적인 방법으로 보장
- 컬렉션이 여러 머신(여러 실행 컨텍스트, 즉 여러 JVM)에 분산되어 있다는 사실은 사용자에게 투명(transparent)
- RDD를 사용하는 방법은 리스트(List), 맵(Map),셋(Set) 등 기존의 평범한 로컬 컬렉션을 사용하는 방법과 같음
- 즉, RDD의 목적은 분산 컬렉션의 성질과 장애 내성을 추상화하고 직관적인 방식으로 대규모 데이터셋에 병렬 연산을 수행할 수 있도록 지원

1.스파크 프로그램 작성

RDD의 개념

- 스파크에 내장된 장애 복구 메커니즘은 RDD에 복원성을 부여
- 스파크는 노드에 장애가 발생해도 유실된 RDD를 원래대로 복구할 수 있음
- RDD는 데이터셋 자체를 중복 저장하지 않는 대신, 데이터셋을 만드는 데 사용된 변환 연산자의 로그(즉, 데이터셋을 어떻게 만들었는지)를 남기는 방식으로 장애 내성을 제공
- 일부 노드에 장애가 발생하면 스파크는 해당 노드가 가진 데이터셋만 다시 계산해 RDD를 복원
- RDD에 적용된 변환 연산자와 그 적용 순서를 RDD 계보(lineage)라고 함

2.RDD의 기본 행동 연산자 및 변환연산자

RDD의 기본 행동 연산자 및 변환 연산자

- RDD 연산자는 크게 변환(transformation)과 행동(action)이라는 두 유형
- 변환 연산자는 RDD의 데이터를 조작해 새로운 RDD를 생성(예: filter나 map 함수)
- 행동 연산자는 연산자를 호출한 프로그램으로 계산 결과를 반환하거나 RDD 요소에 특정 작업을 수행하려고 실제 계산을 시작하는 역할(예: count나 foreach 함수)

2.RDD의 기본 행동 연산자 및 변환연산자

map 변환 연산자

- map은 RDD의 모든 요소에 임의의 함수를 적용할 수 있는 변환 연산자
- map 함수는 또 다른 함수를 인자로 받아 RDD 하나를 반환
- map 함수가 반환하는 RDD는 map 함수가 호출된 RDD와는 다른 타입의 요소로 구성될 수 있음
- filter 함수와는 달리 map 함수가 호출된 RDD(즉, this 객체)의 타입은 map 함수가 반환하는 RDD의 타입과 같을 수도 있고 다를 수도 있음

map 메서드 시그니처

```
class RDD[T] { ------- RDD는 타입 매개변수 T를 가진 클래스로 정의되었다.
    // ... other methods ...
    def map[U](f: (T) => U): RDD[U] ------- map 함수는 또 다른 함수(f: (T) => U)를 인자로 받아
                                            이 RDD와는 다른 타입(U)의 RDD를 반환한다.
    // ... other methods ...
}
```

2.RDD의 기본 행동 연산자 및 변환연산자

map 함수를 사용해 RDD 요소의 제곱 값을 계산하는 프로그램

- scala> val numbers = sc.parallelize(10 to 50 by 10)
- 스파크 컨텍스트의 parallelize 메서드는 Seq 객체(Seq는 스파크의 컬렉션 인터페이스로 이 인터페이스를 구현한 클래스에는 Array나 List) 를 받아 이 Seq객체의 요소로 구성된 새로운 RDD를 만듬
- Seq 객체의 요소는 여러 스파크 실행자(executor)로 분산
- parallelize 메서드는 makeRDD라는 별칭으로도 호 출할 수 있음
- 메서드의 인수로 전달된 내 to 50 by 10은 스칼라 특유의 표현식으로 Seq 인터페이스를 구현한 Range 클래스 객체를 생성

2.RDD의 기본 행동 연산자 및 변환연산자

Distinct와 flatMap 변환 연산자

- “지난 한 주 동안 고객이 온라인에서 상품을 구매한 이력을 기록 한 가상의 로그 파일이 하나 있다. 웹 사이트 서버는 고객이 상품을 구매할 때마다 이 고객의 고유 ID를 로그 파일의 맨 끝에 추가한다. 또 하루가 끝날 때마다 파일 마지막에 개행 문자를 추가한다”
- 즉, 이 로그 파일에는 하루 동안 구매한 모든 고객의 ID가 한 줄에 나열되며, 각 줄의 고객 ID는 쉼표(,)로 구분
- 한 주 동안 최소 한 번 이상 구매한 고객이 몇 명인지 분석

```
/********************************** terminal:
echo "15,16,20,20
77,80,94
94,98,16,31
31,15,20" > ~/client-ids.log
end terminal ***********************************/
```

2.RDD의 기본 행동 연산자 및 변환연산자

Distinct와 flatMap 변환 연산자

- scala> val lines = sc.textFile("/home/spark/client-ids.log")
- 파일의 각 줄을 쉼표로 분리해 문자열 배열을 생성
- scala> val idsStr = lines.map(line => line.split(","))
- scala> idsStr.foreach(println(_)
- scala> idsStr.first
- scala> idsStr.collect

2.RDD의 기본 행동 연산자 및 변환연산자

Distinct와 flatMap 변환 연산자

- 배열의 배열을 단일 배열로 분해
- 변환 연산자가 반환한 여러 배열의 모 든 요소를 단일 배열로 통합하려는 상황에는 flatMap
- flatMap은 기본적으로 주어진 함 수를 RDD의 모든 요소에 적용한다는 점에서 map과 동일
- 다른 점은 익명 함수가 반환한 배 열의 중첩 구조를 한 단계 제거하고 모든 배열의 요소를 단일 컬렉션으로 병합

2.RDD의 기본 행동 연산자 및 변환연산자

Distinct와 flatMap 변환 연산자

- scala> val ids = lines.flatMap(_.split(","))
- scala> ids.collect
- map 함수를 적용한 후 collect를 호출했을 때는 배열의 배열이 반환
- 반면 flatMap 은 이 중첩 구조를 한 단계 벗겨 낸 1차원 배열을 반환
- scala> ids.first
- scala> ids.collect.mkString("; ")
- scala> val intIds = ids.map(_.toInt)
- scala> intIds.collect

2.RDD의 기본 행동 연산자 및 변환연산자

Distinct와 flatMap 변환 연산자

- RDD의 distinct 메서드를 호출하면 스파크는 해당 RDD의 고유 요소로 새로운 RDD를 생성
- scala> val uniqueIds = intIds.distinct
- scala> uniqueIds.collect
- scala> val finalCount = uniqueIds.count

2.RDD의 기본 행동 연산자 및 변환연산자

RDD의 일부 요소 가져오기

- sample은 호출된 RDD(즉, this 객체)에서 무작위로 요소를 뽑아 새로운 RDD를 만드는 변환 연산자
- def sample(withReplacement: Boolean, fraction: Double, seed: Long = Utils.random-nextLong): RDD[T]
- 첫번째 인자인 vdthReplacement는 같은 요소가 여러 번 샘플링 될 수 있는지 지정(인자를 false로 지정하면 한 번 샘플링된 요소는 메서드 호출이 끝날 때까지 다음 샘플링 대상에서 제외)

2.RDD의 기본 행동 연산자 및 변환연산자

RDD의 일부 요소 가져오기

- 두 번째 인자인 fraction은 복원 샘플링에서는 각 요소가 샘플링될 횟수의 기댓값(0 이상의 값)을 의미
- 비복원 샘플링에서는 각 요소가 샘플링될 기대 확률(0~1 사이의 부동소수점 숫자)을 의미한다
- 복원 샘플링(sampling with replacement)과 비복원 샘플링 (sampling without replacement)
- 세 번째 인자는 난수 생성에 사용하는 시드(seed). 같은 시드는 항상 같은 유사 난수(quasirandom ruimhcr)를 생성하기 때문에 프로그램을 테스트하는 데 유용

2.RDD의 기본 행동 연산자 및 변환연산자

RDD의 일부 요소 가져오기

- scala> val s = uniqueIds.sample(false, 0.3)
- scala> s.count
- scala> s.collect
- sample 메서드가 반환한 요소는 총 두 개
- 이는 고유한 고객 ID 개수(여덟 개)의 30% (2.4개)에 가깝다
- scala> val swr = uniqueIds.sample(true, 0.5)
- scala> swr.count
- scala> swr.collect

2.RDD의 기본 행동 연산자 및 변환연산자

RDD의 일부 요소 가져오기

- 확률 값 대신 정확한 개수로 RDD의 요소를 샘플링하려면 takeSample 행동 연산자를 사용
- 첫 번째는 takeSample 메서드의 두 번째 인자가 샘플링 결과로 반환될 요소의 개수를 지정하는 정수형 변수라는 점[즉, 요소 개수의 기댓값이 아닌 항상 정확한 개수(num 인자)로 샘플링]
- 두 번째는 sample이 변환 연산자인 반면 takeSample 은 (collect와 마찬가지로) 다음과 같이 배열을 반환하는 행동 연산자라는 점
- scala> val taken = uniqueIds.takeSample(false, 5)

2.RDD의 기본 행동 연산자 및 변환연산자

RDD의 일부 요소 가져오기

- 데이터의 하위 집합을 가져올 수 있는 또 다른 행동 연산자로 take를 사용할 수 있음
- 지정된 개수의 요소를 모을 때까지 RDD의 파티션(클러스터의 여러 노드에 저장된 데이터의 일부 분)을 하나씩 처리해 결과를 반환
- scala> uniqueIds.take(3)
- take 메서드의 결과는 단일 머신에 전송되므로 인자에 너무 큰 수를 지정해서는 안 됨

3. Double RDD 전용 함수

Double RDD 함수로 기초 통계량 계산

- scala> intlds.mean
- scala> intlds.sum
- scala> intlds.variance
- scala> intlds.stdev

3. Double RDD 전용 함수

히스토그램으로 데이터 분포 시각화

- 히스토그램은 데이터를 시각화하는 데 주로 사용
- 히스토그램의 x축에는 데이터 값의 구간 (interval)을 그리고, Y축에는 각 구간에 해당하는 데이터 밀도나 요소 개수를 그림
- double RDD의 histogram 행동 연산자에는 버전이 두 가지
- 첫 번째 버전은 구간 경계를 표현하는 Double 값의 배열을 받고, 각 구간에 속한 요소 개수를 담은 Array 객체를 반환, 구간 경계를 표현하는 Double 배열은 반드시 오름차순으로 정렬되어 있어야 하고, (구간을 한 개 이상 표현할 수 있도록) 두 개 이상의 요소를 포함해야 하며, 중복된 요소가 있으면 안됨
- scala> intIds.histogram(Array(1.0, 50.0 , 100.0))

3. Double RDD 전용 함수

히스토그램으로 데이터 분포 시각화

- histogram의 두 번째 버전은 구간 개수를 받아 이것으로 입력 데이터의 전체 범위를 균등하게 나눈 후 요소 두 개로 구성된 튜플(tuple) 하나를 결과로 반환
- 반환된 튜플의 첫 번째 요소는 구 간 개수로 계산된 구간 경계의 배열이며, 두 번째 요소는 (histogram의 첫 번째 버전과 마찬가지 로) 각 구간에 속한 요소 개수가 저장된 배열
- scala> intlds.histogram(3)

3. 스파크 애플리케이션

JSON(JavaScript Object Notation)

- SparkContext와 SQLContext를 SparkSession 클래스
- SQLContext의 read 메서드는 다양한 데이터를 입수하는 데 사용할 수 있는 DataFrameReader 객체를 반환

json 메서드 시그니처

```
def json(paths: String*): DataFrame
Loads a JSON file (one object per line) and returns the result as a [[DataFrame]].
```

- json 메서드는 한 줄당 JSON 객체 하나가 저장된 파일을 로드

Any Questions?