Spark

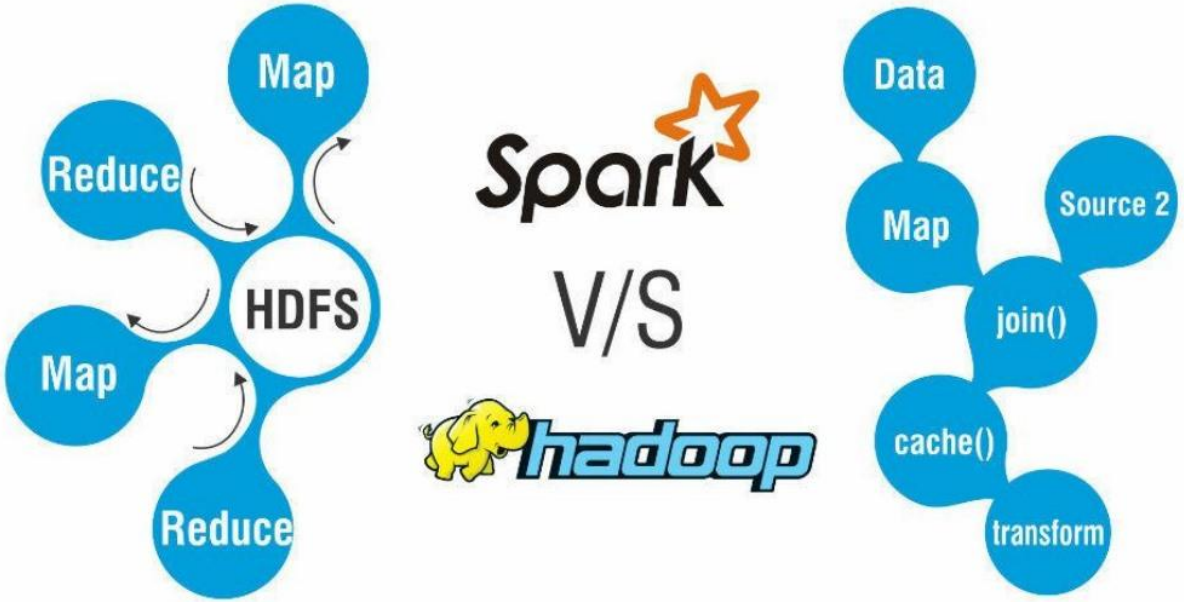
스파크 완벽 가이드

하둡 vs 스파크

■ **분산/병렬 처리**

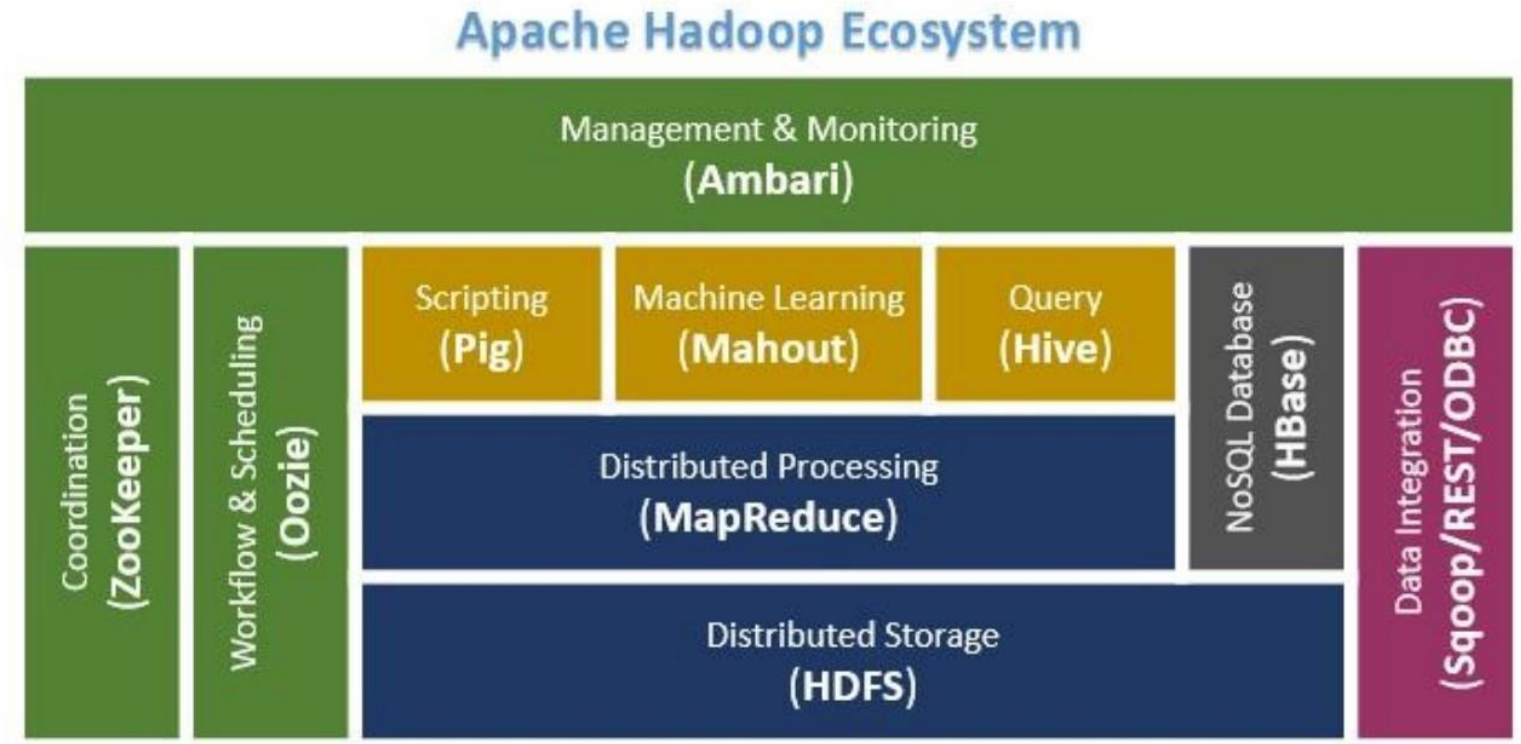
○ 하둡(Hadoop)

○ 스파크(Spark)



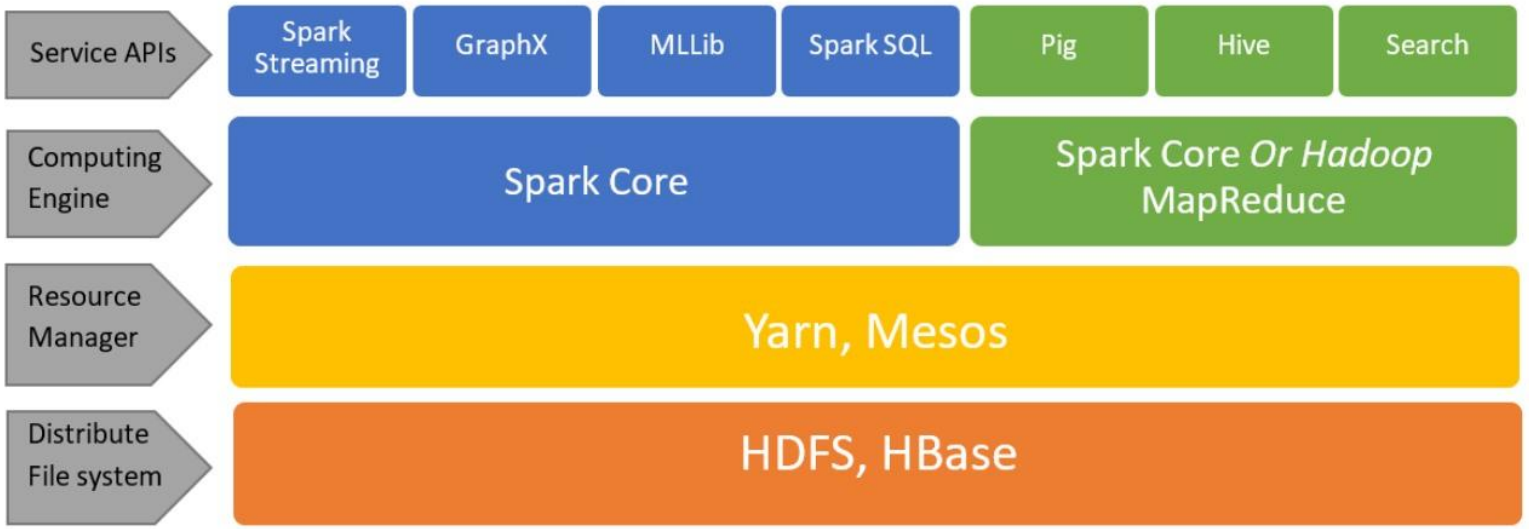
하둡 에코시스템

■ 하둡


Apache Hadoop Ecosystem
Management & Monitoring
(Ambari)
Coordination
(ZooKeeper)
Workflow & Scheduling
(Oozie)
Scripting
(Pig)
Machine Learning
(Mahout)
Query
(Hive)
Distributed Processing
(MapReduce)
Distributed Storage
(HDFS)
NoSQL Database
(HBase)
Data Integration
(Sqoop/REST/ODBC)

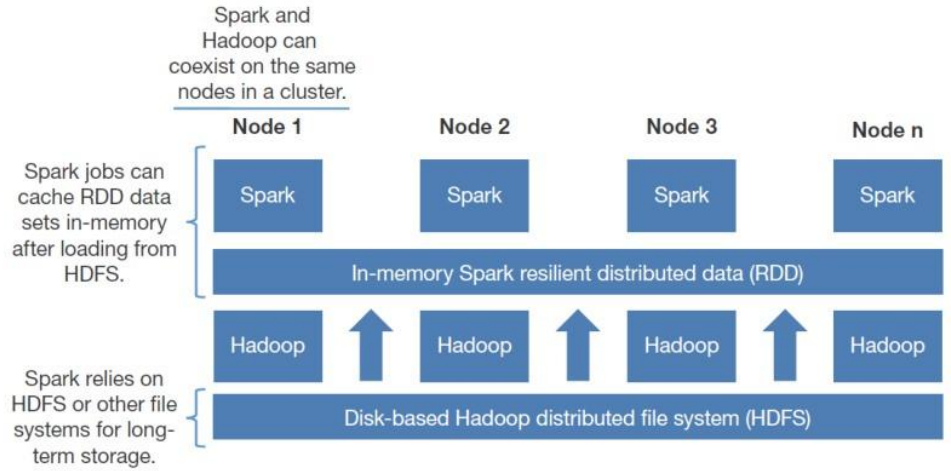
스파크 구성

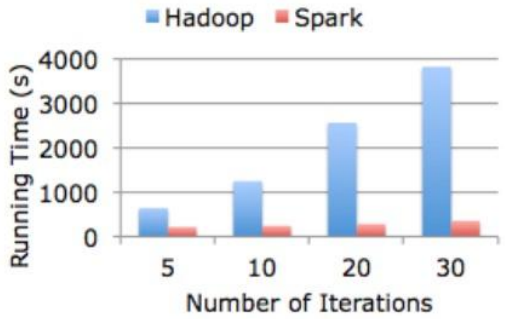
■ 스파크


Service APIs
Spark Streaming
GraphX
MLLib
Spark SQL
Pig
Hive
Search
Computing Engine
Spark Core
Spark Core *Or Hadoop* MapReduce
Resource Manager
Yarn, Mesos
Distribute File system
HDFS, HBase

하둡 vs 스파크

■ 하둡 vs 스파크



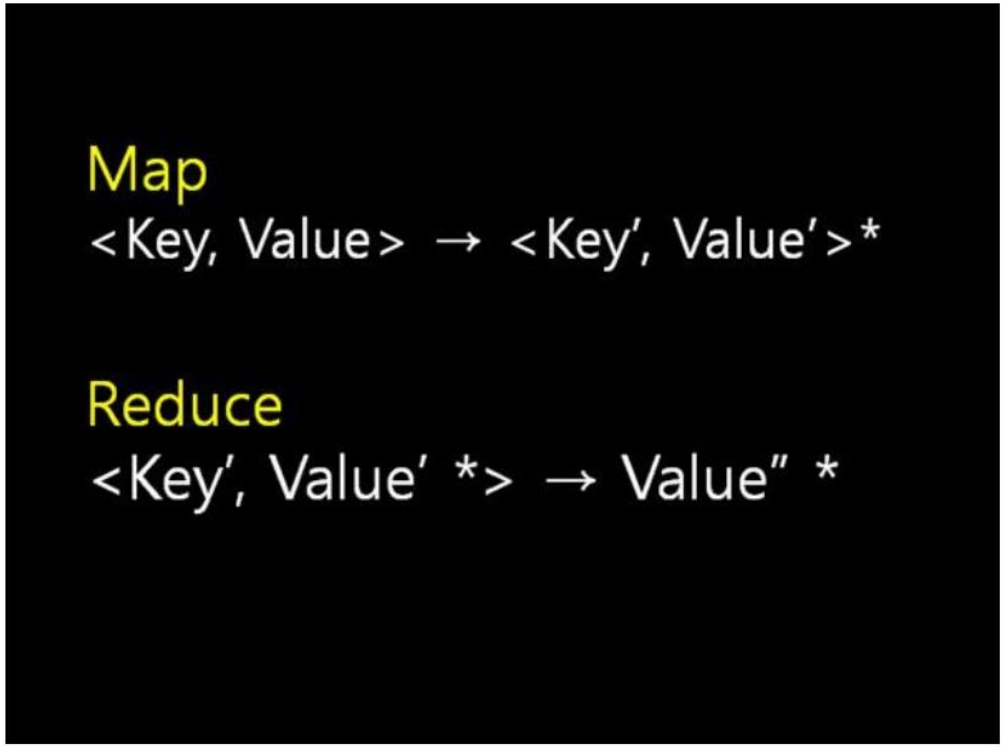


구분	하둡	스파크
1. 하둡과 스파크는 다르다	분산 데이터 인프라	분산된 데이터 처리 도구
2. 서로 필요는 없다	HDFS와 MapReduce라는 자체 데이터 처리 도구가 포함	하둡과 가장 잘 동작하지만, 굳이 하둡이 필요하지는 않음
3. 속도	맵리듀스를 배치방식이라 늦음	인메모리 처리 방식이라 배치는 10배, 인메모리 분석에는 100배 빠름
4. 모든 작업에 속도가 중요한가	정적이며 배치형태 작업에 적합	실시간 IoT 센서, 기계학습, 실시간 마케팅, 온라인 추천 등에 적합
5. 장애 복구 능력	태생적으로 시스템 장애에 내성이 강함	RDD로 장애 복구능력을 기본 제공

출처: InfoWorld, Five things you need to know about Hadoop v. Apache Spark

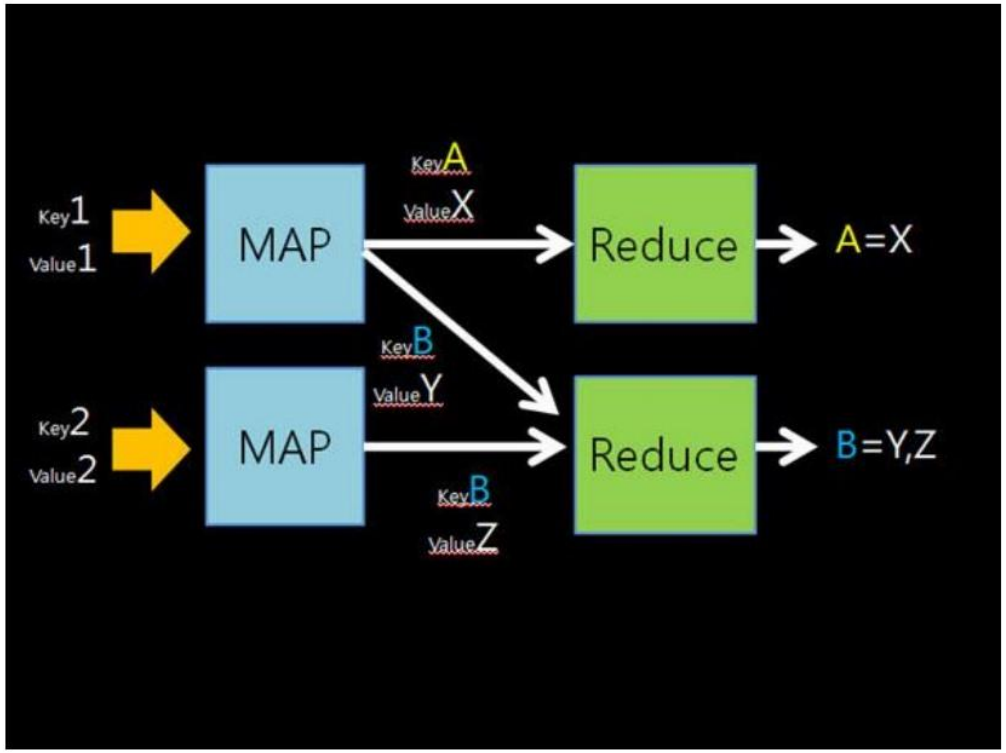
맵리듀스

■ 맵리듀스



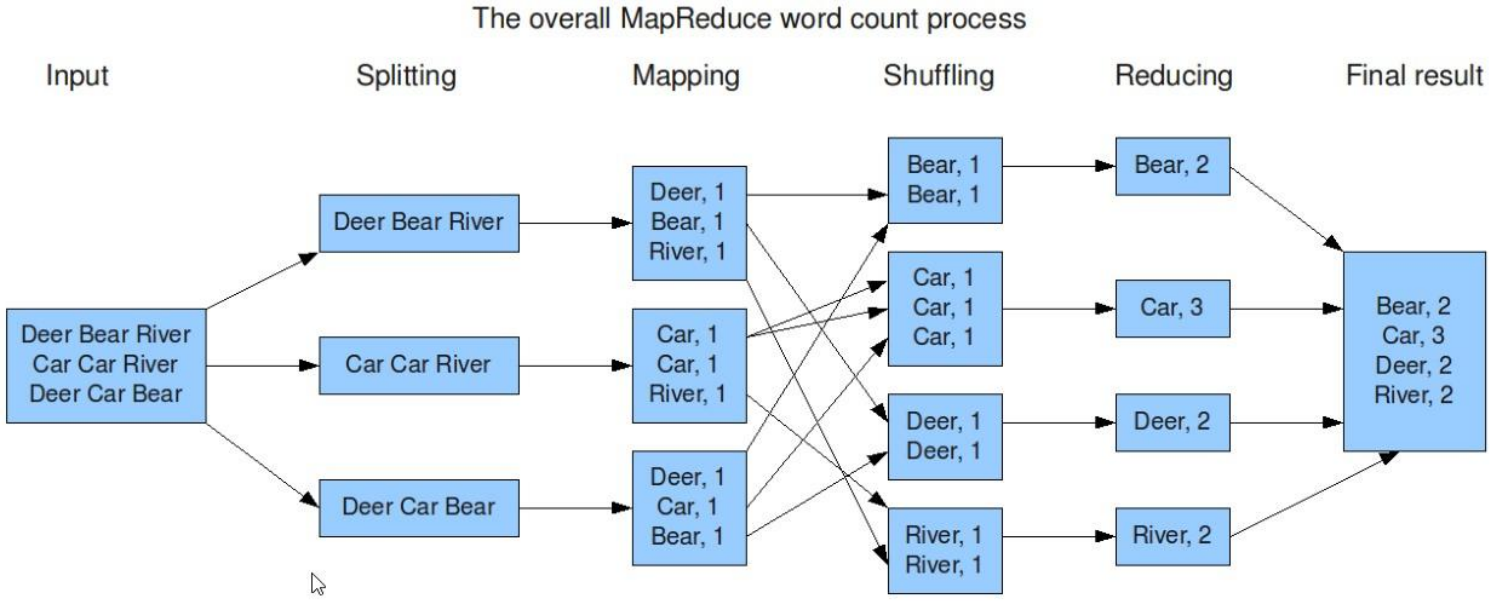
맵리듀스 과정

■ 맵리듀스 과정



맵리듀스 예제

■ 맵리듀스 예제



하둡

■ 하둡(Hadoop)

○ 대용량 데이터를 분산 처리할 수 있는 Java 기반의 오픈소스 프레임워크

- 2005년 더그 커팅(Doug Cutting)이 구글이 발표한 GFS(Google File System)와 맵리듀스(MapReduce)를 구현
- 루씬(Lucene) 기반 오픈소스 검색 엔진인 너치(Nutch)에 적용하기 위해 시작
- 이후 독립적인 프로젝트로 만들어졌고, 2008년에는 아파치 최상위 프로젝트로 승격

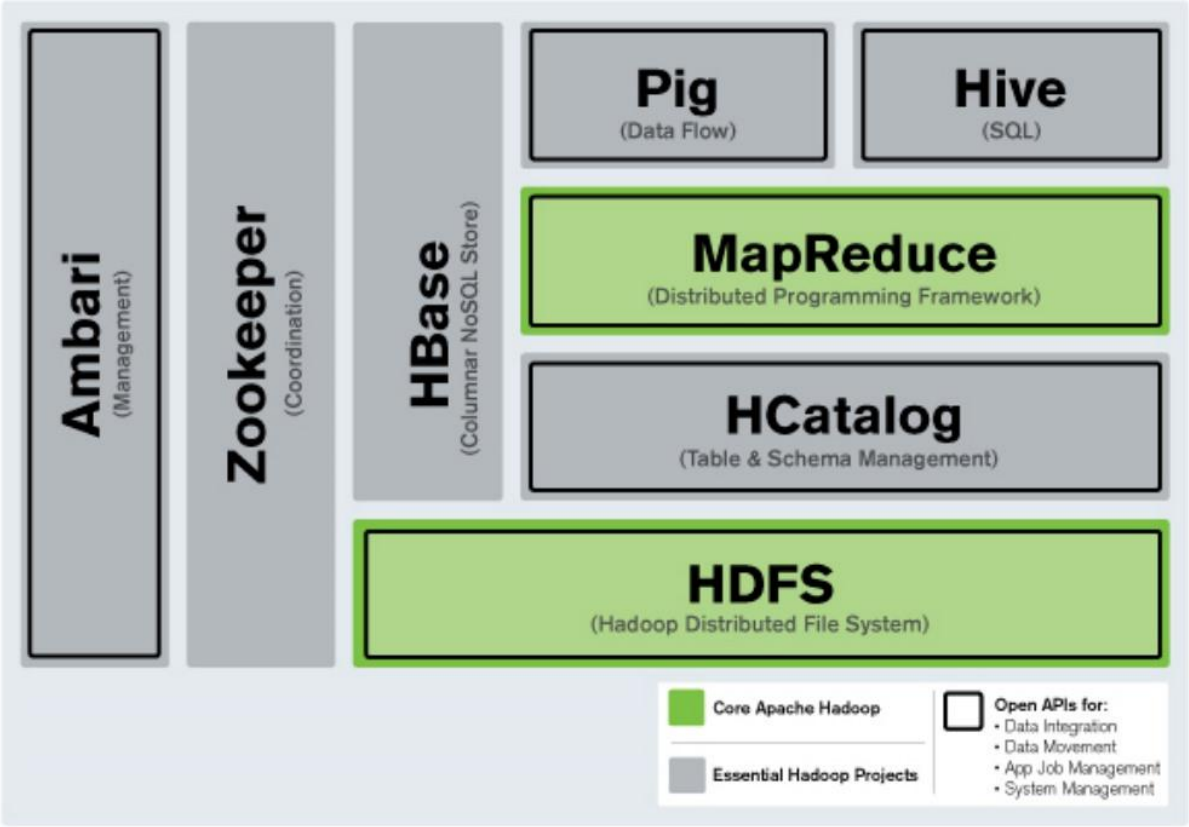
○ 특징

- Distributed
  - 정형 데이터의 경우 기존 RDBMS에 저장할 수 있지만, 비정형 데이터를 RDBMS에 저장하기에는 크기가 너무 큼
- Open Source
  - 하둡은 오픈소스 프로젝트이기에 소프트웨어 라이선스 비용에 대한 부담이 없음
- Scalable
  - 값비싼 유닉스 장비를 사용하지 않고, x86 CPU에 리눅스 서버 면 얼마든지 설치하고 운영할 수 있음
  - 데이터 저장 용량이 부족할 경우, 필요한 만큼 리눅스 서버만 추가하면 됨
- Fault-Tolerant
  - 노드마다 데이터의 복제본을 저장하기 때문에 데이터의 유실이나 장애가 발생했을 때도 데이터 복구 가능

하둡 구성

○ 구성

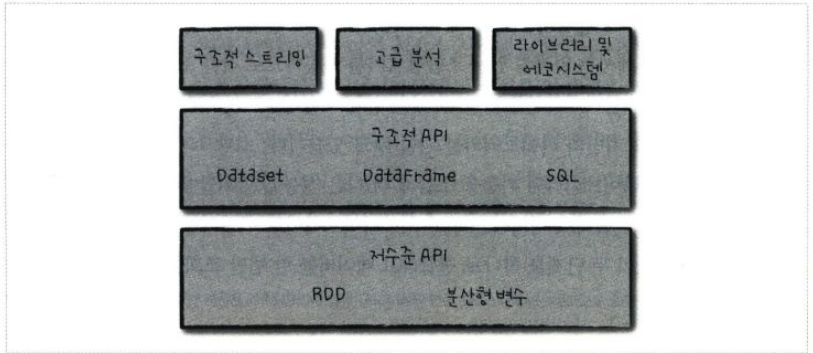
- HDFS(Hadoop Distributed File System)
- MapReduce



아파치 스파크란

- 스파크는 통합 컴퓨팅 엔진이며 클러스터 환경에서 데이터를 병렬로 처리하는 라이브러리 집합
  - 분산 컴퓨팅용 자바 기반 오픈소스 프레임워크로 하둡 분산 파일 시스템(HDFS)과 맵리듀스 처리 엔진으로 구성
  - 하둡과 유사하지만, 대량의 데이터를 메모리에 유지하는 독창적인 설계로 약 10 ~ 100배 더 빠른 속도로 같은 작업을 처리
- 현재 활발하게 개발되고 있는 병렬 처리 오픈소스 엔진
- 네가지 언어 지원하며 SQL뿐만 아니라 스트리밍, 머신러닝에 이르기까지 넓은 범위의 라이브러리 제공
  - 파이썬, 자바, 스칼라, R
- 단일 노트북 환경에서부터 수천 대의 서버로 구성된 클러스터까지 다양한 환경에서 실행할 수 있음
- 빅데이터 처리를 쉽게 시작할 수 있고 엄청난 규모의 클러스터로 확장해나갈수 있음

그림 1-1 스파크 기능 구성



아파치 스파크란

- 일부 애플리케이션은 스파크를 사용하기에 적합하지 않은 경우도 있음
  - 분산 아키텍처 때문에 처리 시간에 약간의 오버헤드가 필연적으로 발생하기 때문에 대량의 데이터를 다룰 때는 오버헤드가 무시할 수 있는 수준이지만, 단일 머신에서도 충분히 처리할 수 있는 데이터 셋을 다룰 때는 작은 데이터셋의 연산에 최적화된 다른 프레임워크를 사용하는 것이 효율적임.
- 온라인 트랜잭션 처리(Online Transaction Processing, OLTP) 애플리케이션을 염두에 두고 설계되지 않음
  - 대량의 원자성 트랜잭션을 빠르게 처리해야 하는 작업에는 스파크가 적합하지 않음
- 일괄 처리 작업이나 데이터 마이닝 같은 온라인 분석처리(Online Analytical Processing, OLAP) 작업에는 적합

- 참고
  - OLTP
    - 빈번한 데이터의 입력, 수정, 삭제 과정에서의 효율성, 즉 효과적인 갱신이 주요 목표
  - OLAP
    - 데이터를 기초로 하여 효과적으로 분석하고 조회하는 것이 주목적

아파치 스파크의 철학

- 빅데이터를 위한 통합 컴퓨팅 엔진과 라이브러리 집합
- 통합
  - 데이터 읽기, SQL 처리, 머신러닝 그리고 스트림 처리에 이르기까지 다양한 데이터 분석 작업을 같은 연산 엔진과 일관성 있는 API로 수행할 수 있도록 설계되어 있음
  - 스파크에서는 일관성 있는 조합형 API를 제공하므로 작은 코드 조각이나 기존 라이브러리를 사용해 애플리케이션을 만들수 있음
  - 다른 라이브러리의 기능을 조합해 더 나은 성능을 발휘할 수 있도록 설계
    - 예) SQL 쿼리로 데이터를 읽고 ML 라이브러리로 머신러닝 모델을 평가할 경우 스파크 엔진은 이 두 단계를 하나로 병합하고 데이터를 한 번만 조회할 수 있게 해줌
  - 스파크가 발표되기 전에는 통합 엔진을 제공하는 병렬 데이터 처리용 오픈소스가 없었음
- 컴퓨팅 엔진
  - 스파크는 통합이라는 관점을 중시하면서 기능의 범위를 컴퓨팅 엔진으로 제한
  - 저장소 시스템의 데이터를 연산하는 역할만 할 뿐 영구 저장소 역할은 수행하지 않음
  - 하지만 애저 스토리지, 아마존 S3, 분산 파일 시스템인 아파치 하둡, 아파치 카산드라, 아파치 카프카등의 저장소를 지원

아파치 스파크의 철학

- 라이브러리
  - 데이터 분석 작업에 필요한 통합 API를 제공하는 통합 엔진 기반의 자체 라이브러리
  - 스파크 엔진에서 제공하는 표준 라이브러리와 오픈소스 커뮤니티에서 서드파티 패키지 형태로 제공하는 다양한 외부 라이브러리를 지원합니다.
  - SQL과 구조화된 데이터를 제공하는 스파크 SQL, 머신러닝을 지원하는 MLlib, 스트림 처리 기능을 제공하는 스파크 스트리밍, 그래프 분석 엔진인 GraphX 라이브러리

등장 배경

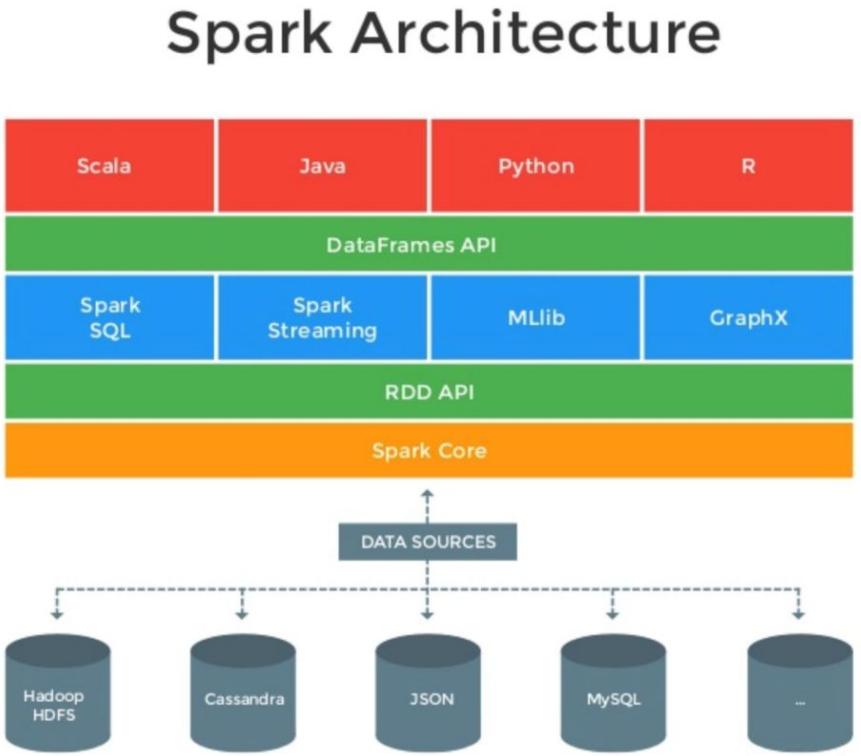
- 컴퓨터의 성능은 2005년경에서 멈췄습니다.(온도 때문에 CPU의 클럭을 더 높일 수 없는 문제) 그 이후에 CPU 제조사들은 코어를 추가하기 시작했습니다.
- 이러한 현상은 애플리케이션의 성능 향상을 위해 병렬 처리가 필요
- 데이터를 생성하는 디바이스 및 저장 장치는 시간이 흐를수록 낮은 가격에 형성되어 데이터의 홍수가 일어나기 시작함
- 데이터 수집 비용은 극히 저렴해졌지만, 데이터는 클러스트에서 처리해야 할 만큼 거대해짐
- 지난 50년간 지배한 전통적인 프로그래밍 모델도 더는 힘을 발휘하지 못함
- 새로운 프로그래밍 모델이 필요해졌으며 이런 문제를 해결하기 위해 아파치 스파크가 탄생

스파크의 역사

- UC버클리 대학교에서 2009년 스파크 연구 프로젝트로 시작
- Spark : Cluster Computing with Working Sets를 통해 처음 세상에 알려짐
- 당시에는 하둡 맵리듀스가 병렬 프로그래밍 엔진의 대표주자였음
- 맵리듀스로 처리하면 단계별로 잡을 개발하고 클러스터에서 각각 실행해야 하므로 매번 데이터를 처음부터 읽어야 하는 불편함이 있음
- 이러한 문제를 해결하기 위해서 애플리케이션을 간결하게 개발할 수 있는 함수형 프로그래밍 기반의 API를 설계
- 분석가나 데이터 과학자들이 스파크에서 대화형으로 SQL을 실행할 수 있는 엔진인 Shark를 개발
- 머신러닝 라이브러리 MLlib, 스파크 스트리밍 그리고 GraphX를 만들기 시작
- 2013년 아파치 재단에 기부

스파크의 기본 아키텍처

- 컴퓨터 클러스터는 여러 컴퓨터의 자원을 모아 하나의 컴퓨터처럼 사용할 수 있게 만듭니다.
- 이 클러스터에서 작업을 조율할 수 있는 프레임워크가 필요합니다. 그 역할을 스파크가 할 수 있습니다.



스파크 코어

- 스파크 코어
  - 스파크 잡과 다른 스파크 컴포넌트에 필요한 기본 기능을 제공
  - 가장 중요한 개념은 스파크 API의 핵심 요소인 RDD(Resilient Distributed Dataset)이다.
  - 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 가지고 있음
- 파일 시스템에 접근
  - HDFS, GLUSTERfs, 아마존 S3등 다양한 파일 시스템에 접근
  - 공유 변수(broadcast variable)와 누적 변수(accumulator)를 사용해 컴퓨팅 노드 간에 정보를 공유할 수 있음
- 네트워크, 보안, 스케줄링 및 데이터 셔플링등 기본 기능이 구현

스파크 SQL

- 스파크 SQL
  - 스파크와 하이브 SQL이 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 개선하며, 스파크 SQL을 스파크에서 가장 중요한 컴포넌트로 부상시킴
- 다양한 정형 데이터를 읽고 쓰는 데 사용할 수 있음
  - JSON 파일: (JavaScript Object Notation)
  - Parquet 파일 : 데이터와 스키마를 함께 저장할 수 있는 파일 포맷으로 최근 널리 사용
  - 관계형 데이터 베이스 테이블
  - 하이브 테이블
- DataFrame과 Dataset에 적용된 연산을 일정 시점에 RDD 연산으로 변환해 일반 스파크 잡으로 실행
- 카탈리스트라는 쿼리 최적화 프레임워크를 제공하며, 사용자가 직접 정의한 최적화 규칙을 적용해 프레임워크를 확장할 수 있음

스파크 스트리밍

- 스파크 스트리밍
  - 다양한 데이터 소스에서 유입되는 실시간 스트리밍 데이터를 처리하는 프레임워크
- 지원하는 스트리밍 소스
  - HDFS, 아파치 카프카, 아파치 플럼, 트위터, ZeroMQ가 있고, 커스텀 데이터 소스도 정의할 수 있음
- 장애가 발생하면 연산 결과를 자동으로 복구함
- 스파크 스트리밍과 다른 스파크 컴포넌트를 단일 프로그램에서 사용해 실시간 처리 연산과 머신러닝 작업, SQL 연산, 그래프 연산 등을 통합할 수도 있음

스파크 MLlib

- 스파크 MLlib
  - UC 버클리의 MLbase 프로젝트에서 개발한 머신 러닝 알고리즘 라이브러리
  - 로지스틱 회귀, 나이브 베이즈 분류, 서포트 벡터 머신, 의사 결정 트리, 랜덤 포레스트, 선형회귀, k-평균 군집화 지원

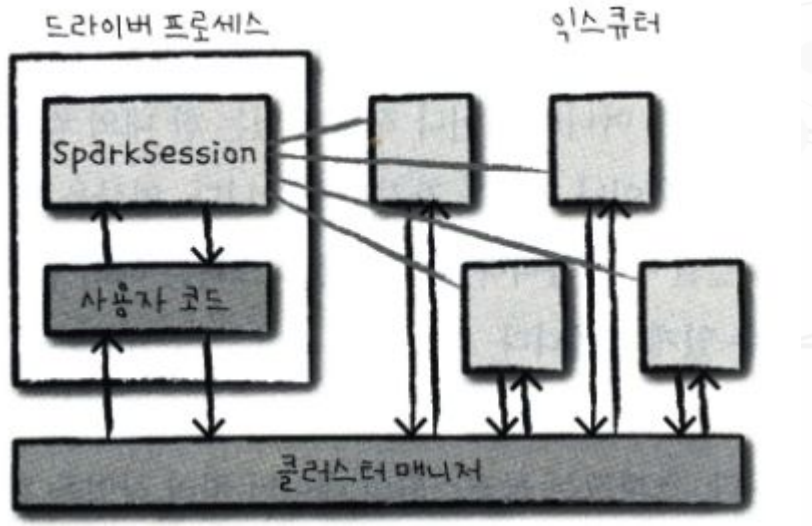
- 아파치 머하웃
  - 하둡에서 분산 머신 러닝 알고리즘을 구현한 오픈소스 프로젝트
  - 스파크 MLlib보다 더 오래되고 성숙했지만, 스파크 MLlib는 머하웃이 지원하는 머신러닝 알고리즘 대부분을 지원함
  - 머하웃도 기존 맵리듀스 처리 엔진을 스파크로 전환하고 있음

스파크 GraphX

- 스파크 GraphX
  - 그래프는 정점과 두 정점을 잇는 간선으로 구성된 데이터 구조
  - 그래프 RDD(EdgeRDD 및 VertexRDD) 형태의 그래프 구조를 만들 수 있는 다양한 기능을 제공
- 지원 알고리즘
  - 페이지랭크
  - 연결요소
  - 최단 경로 탐색
  - SVD++

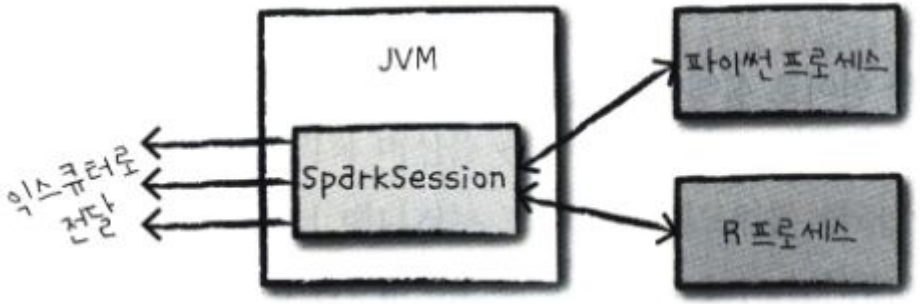
스파크의 기본 아키텍처

- 스파크 애플리케이션
  - 스파크 애플리케이션은 드라이버 프로세스와 다수의 익스큐터 프로세스로 구성
  - 드라이버
    - 드라이버 프로세스는 클러스터 노드 중 하나에서 실행되며 main() 함수를 실행되는데 이는 스파크 애플리케이션 정보의 유지 관리, 사용자 프로그램이나 입력에 대한 응답, 전반적인 익스큐터 프로세스의 작업과 관련된 분석, 배포 그리고 스케줄링 역할을 수행하기 때문에 필수적입니다.
    - 드라이버 프로세스는 스파크 애플리케이션의 심장과 같은 존재로서 애플리케이션의 수명 주기 동안 관련 정보를 모두 유지
  - 익스큐터
    - 드라이버 프로세스가 할당한 작업을 수행
    - 드라이버가 할당한 코드를 실행하고 진행 상황을 다시 드라이버 노드 보고하는 두 가지 역할을 수행
- 스파크 애플리케이션을 이해하기 위한 핵심사항
  - 스파크는 사용 가능한 자원을 파악하기 위해 클러스터 매니저를 사용
  - 드라이버 프로세스는 주어진 작업을 완료하기 위해 드라이버 프로그램 명령을 익스큐터에서 실행할 책임이 있음



스파크의 다양한 언어 API

- 스파크는 모든 언어에 맞는 몇몇 '핵심 개념'을 제공합니다.
- 이러한 핵심 개념은 클러스터 머신에서 실행되는 스파크 코드로 변환됨.
- 구조적 API만으로 작성된 코드는 언어에 상관없이 유사한 성능을 발휘
- 지원 언어
  - 스칼라
    - 스파크는 스칼라로 개발되어 있으므로 스칼라가 기본 언어임
  - 자바
    - 스파크 창시자들은 자바를 이용해 스파크 코드를 작성할 수 있도록 심혈을 기울임
  - 파이썬
    - 스칼라가 지원하는 거의 모든 구조를 지원
  - R
    - 일반적으로 사용하는 두 개의 R 라이브러리가 존재
    - 하나는 스파크 코어에 포함된 SparkR, 다른 하나는 커뮤니티 기반 패키지인 sparklyr



Spark 시작하기

- SparkSession
  - SparkSession이라 불리는 드라이버 프로세스로 제어
  - SparkSession 인스턴스는 사용자가 정의한 처리 명령을 클러스터에서 실행
  - 하나의 SparkSession은 하나의 스파크 애플리케이션에 대응

```
spark
```

```
res4: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@5fd8fcf8
```

```
%spark.pyspark
myRange = spark.range(1000).toDF("number")
```

  - 일정 범위의 숫자를 생성하는 코드
  - DataFrame은 한 개의 컬럼과 1,000개의 로우로 구성되며 각 로우에는 0부터 999까지의 값이 할당
  - 클러스터 모드에서 코드 예제를 실행하면 숫자 범위의 각 부분이 서로 다른 익스큐터에 할당되는데 이것이 스파크의 DataFrame

DataFrame

- DataFrame은 대표적인 API임.
- 테이블의 데이터를 로우와 컬럼으로 단순하게 표현
- 컬럼과 컬럼의 타입을 정의한 목록을 스키마라고 부름
- 스파크의 DataFrame은 수천 대의 컴퓨터에 분산되어 있음
  - 분산하는 이유는 단순하다... 왜? 하나의 컴퓨터에 실행하기엔 너무 오랜 시간이 걸리니까
- 스파크는 파이썬과 R 언어를 모두 지원하기 때문에 Pandas 라이브러리 DataFrame과 R의 DataFrame을 스파크 DataFrame으로 쉽게 변환가능 함

파티션

- 스파크는 모든 익스큐터가 병렬로 작업을 수행할 수 있도록 파티션이라 불리는 청크 단위로 데이터를 분할.
- 파티션은 클러스터의 물리적 머신에 존재하는 로우의 집합을 의미
- DataFrame의 파티션은 실행 중에 데이터가 컴퓨터 클러스터에서 물리적으로 분산되는 방식을 나타냄
- 만약 파티션이 하나라면 스파크에 수천개의 익스큐터가 있더라도 병렬성은 1이 됨
- 수백 개의 파티션이 있더라도 익스큐터가 하나밖에 없다면 병렬성은 1이 됨
- DataFrame을 사용하면 파티션을 수동 혹은 개별적으로 처리할 필요는 없음. 스파크가 실제 처리 방법을 결정

트랜스포메이션

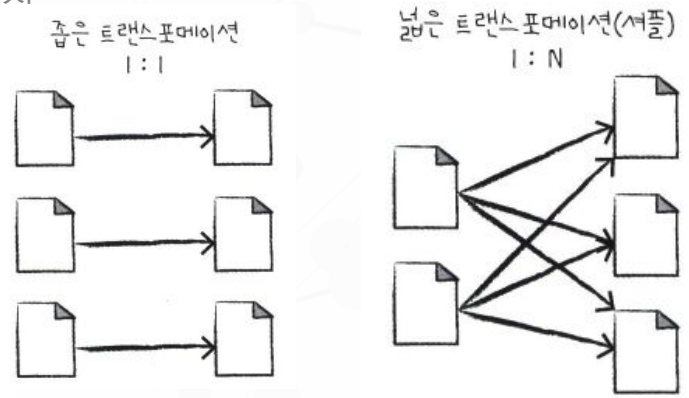
- 스파크의 핵심 데이터 구조는 불변성을 가짐. 즉 한번 생성하면 변경할 수 없음
- DataFrame을 변경하려면 원하는 변경 방법을 스파크에게 알려줘야 함. 이 때 사용하는 명령을 트랜스포메이션이라고 부름

```
%spark.pyspark

divisBy2 = myRange.where("number % 2 = 0")
```

- 위의 코드는 아무런 실행 화면을 볼수 없음. 추상적인 트랜스포메이션만 지정한 상태이기 때문에 액션을 호출하지 않으면 스파크는 실제 트랜스포메이션을 수행하지 않음

트랜스포메이션

- 트랜스포메이션의 두 가지 유형
  - 좁은 의존성
    - 좁은 의존성을 가진 트랜스포메이션은 각 입력 파티션이 하나의 출력 파티션에만 영향을 미침
    - 위 예제에서 where 구문은 좁은 의존성을 가짐
    - 옆의 그림에서 볼 수 있듯이 하나의 파티션이 하나의 출력 파티션에만 영향을 미치



  - 넓은 의존성
    - 하나의 입력 파티션이 여러 출력 파티션에 영향을 미침
    - 셔플 (Shuffle) 스파크가 클러스터에서 파티션을 교환

  - 좁은 트랜스포메이션을 사용하면 스파크에서 파이라이닝을 자동으로 수행. 즉 DataFrame에 여러 필터를 지정하는 경우 모든 작업이 메모리에서 일어남
  - 하지만 셔플은 다른 방식으로 동작. 스파크는 셔플의 결과를 디스크에 저장

지연연산(Lazy Evaluation), 액션

- 지연연산
  - 스파크가 연산 그래프를 처리하기 직전까지 기다리는 동작 방식을 의미합니다.
  - 스파크는 특정 연산 명령이 내려진 즉시 데이터를 수정하지 않고 원시 데이터에 적용할 트랜스포메이션의 실행 계획을 생성
  - 코드를 실행하는 마지막 순간까지 대기하다가 원형 DataFrame 트랜스포메이션을 간결한 물리적 실행 계획으로 컴파일
- 액션
  - 사용자는 트랜스포메이션을 사용해 논리적 실행 계획을 세울 수 있음. 하지만 실제 연산을 수행하려면 액션 명령을 내려야 함
  - 액션은 일련의 트랜스포메이션으로부터 결과를 계산하도록 지시하는 명령
  - 아래 예제는 가장 단순한 액션인 count 메서드를 Datal 환

```
%spark.pyspark
divisBy2.count()
```

```
500
```

  - 세 가지 유형의 액션이 존재
    - 콘솔에서 데이터를 보는 액션
    - 각 언어로 된 네이티브 객체에 데이터를 모으는 액션
    - 출력 데이터소스에 저장하는 액션
  - 액션을 지정하면 스파크 잡이 시작되고 필터(좁은 트랜스포메이션)를 수행한 후 파티션별로 레코드 수를 카운트(넓은 트랜스포메이션)을 함. 그리고 각 언어에 적합한 네이티브 객체에 결과를 모음

RDD

- RDD 개념
  - Resilient Distributed Dataset
  - 메모리 내 데이터 손실 발생 시, 다시 생성할 수 있음. 즉 손상된 파티션을 재연산해 복구
  - 클러스터를 통해 메모리에 분산되어 저장
  - 파일을 통해 가져올 수 있음
  - 스파크에서 기본적인 데이터 단위이며, 모든 작업을 새로운 RDD를 만들거나, 존재하는 RDD를 변경하거나, 결과 계산을 위해 RDD에서 연산을 호출
  - RDD는 지연연산(lazy evaluation)하며 실행 계획만 가지고 있음
  - Action 함수의 실행 전까지는 실제로 이벤트가 일어나지 않음
- Hadoop
  - Hadoop을 이용한 프로그램은 HDFS 읽기-쓰기 작업을 하는데 90% 이상 소비
  - 스파크는 RDD를 이용하여 메모리 내 처리 연산을 지원하며 이로 인해 실행 속도가 HDFS보다 10 ~ 100배 빠름

RDD

- RDD 구조

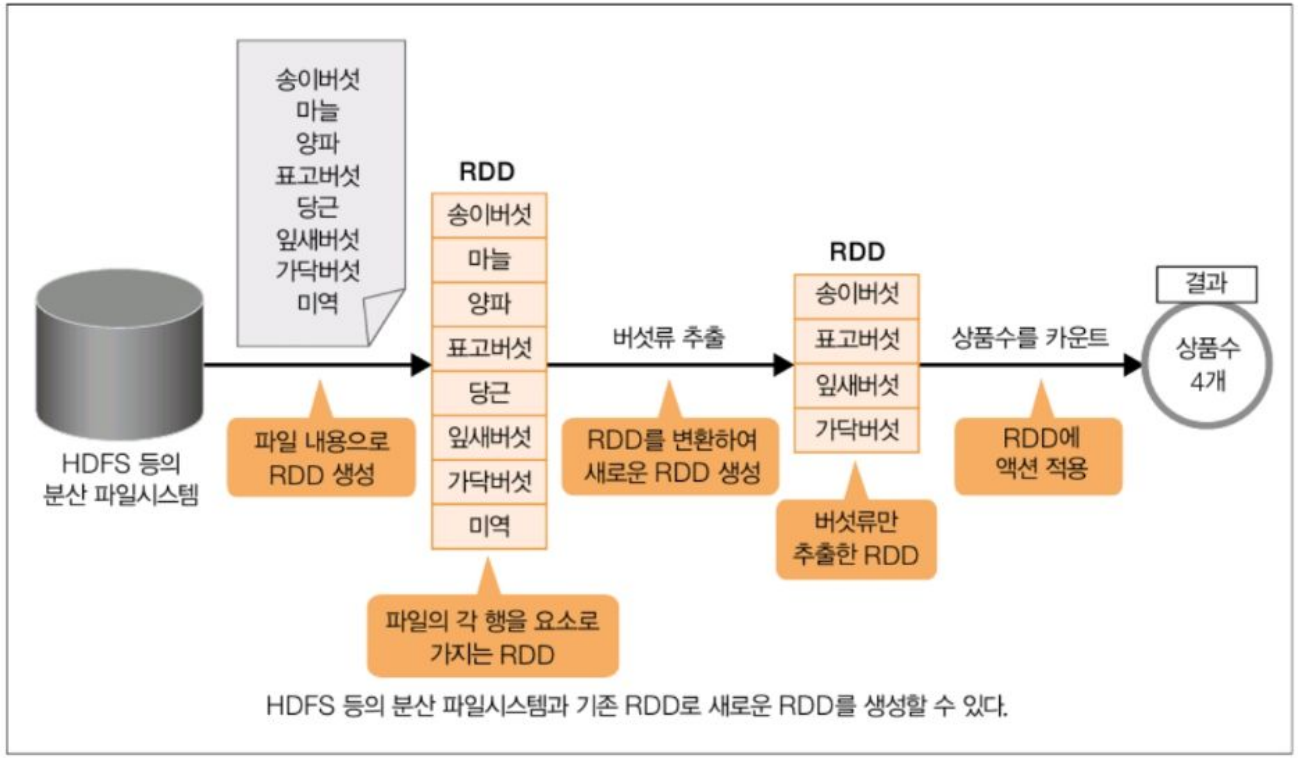
그림 2-1 RDD 구조

- HDFS 등의 분산 파일시스템에 있는 파일 내용을 RDD로 로드하고, RDD를 가공하는 형식으로 대량의 데이터를 분산처리
- 스파크에서는 이런 가공을 변환(transformation)이라고 함
- RDD의 내용에 따라 액션(action)이라는 처리를 적용하여 원하는 결과를 도출

RDD

- RDD의 생성 및 가공과 결과 취득

그림 2-2 RDD의 생성·가공과 결과 취득



HDFS 등의 분산 파일시스템과 기존 RDD로 새로운 RDD를 생성할 수 있다.

- RDD에는 불변(immutable), 즉 내부 요소 값이 변경 불가한 성질과 생성이나 변환이 지연 평가되는 성질을 가지고 있음

RDD – 변환

- RDD에는 변환과 액션 두 종류의 처리를 적용할 수 있음
- 변환(transformation)
  - RDD를 가공하고 그 결과 새로운 RDD를 얻는 처리
  - 변환처리 후의 RDD가 가지는 요소는 변환처리 전의 RDD에 들어있는 요소를 가공하거나 필터링해 생성
  - 변환은 다시 두 종류로 구분할 수 있음
  - filter
    - 요소를 필터링
  - map
    - 각 요소에 동일한 처리를 적용
  - flatmap
    - 각 요소에 동일한 처리를 적용하고 여러 개의 요소를 생성
  - zip
    - 파티션 수가 같고, 파티션에 있는 요소의 수도 같은 두 개의 RDD를 조합해 한쪽의 요소 값을 키로, 다른 한쪽의 요소 값을 밸류로 가지는 쌍(key-value pair)을 만든다.
  - reduceByKey
    - 같은 키를 가지는 요소를 집약처리(aggregation)이라고 한다.
  - join
    - 두 개의 RDD에서 같은 키를 가지는 요소끼리 조인한다.

RDD – 변환

그림 2-3 filter 개요

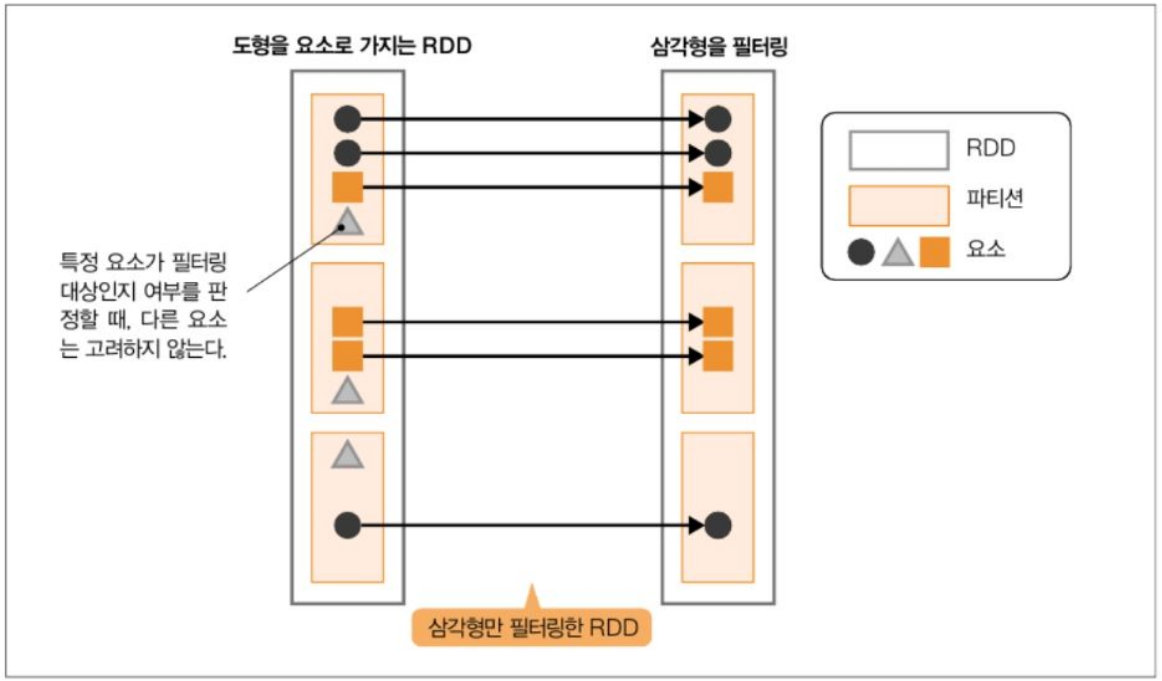
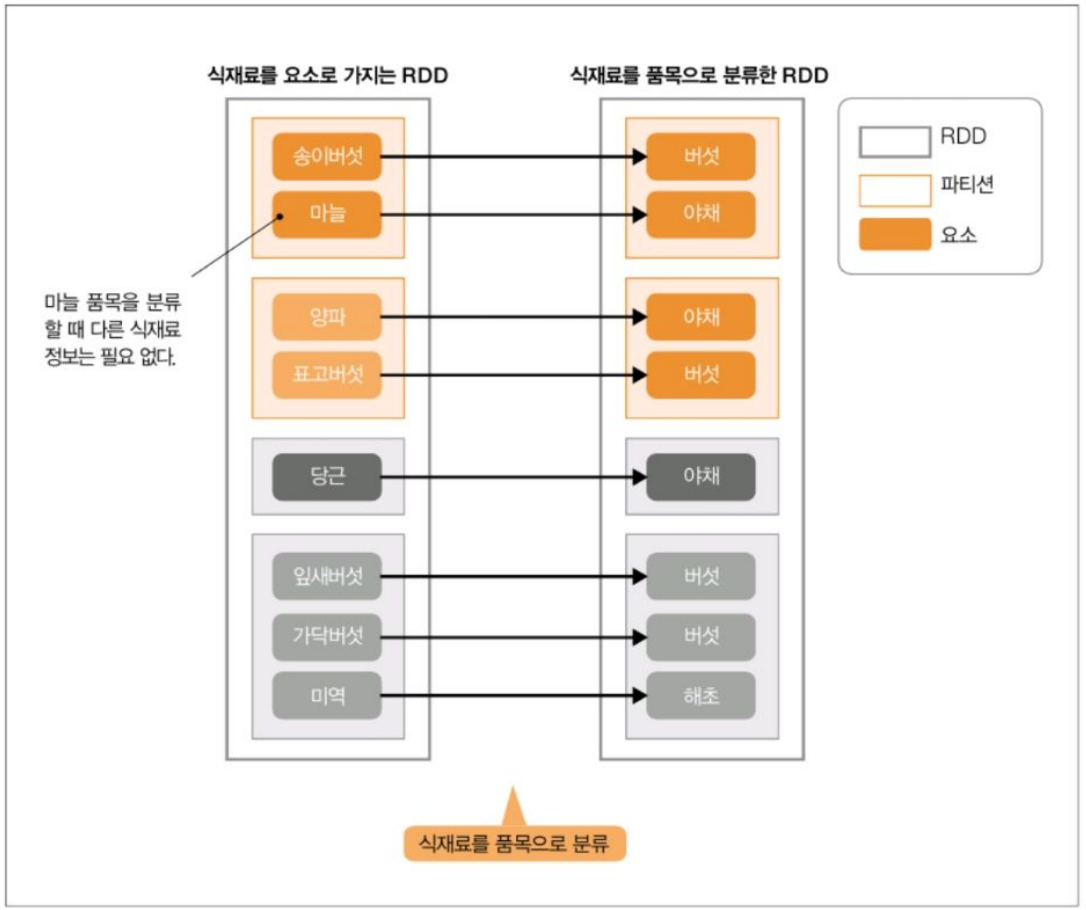


그림 2-4 map 개요



RDD – 변환

그림 2-5 flatMap 개요

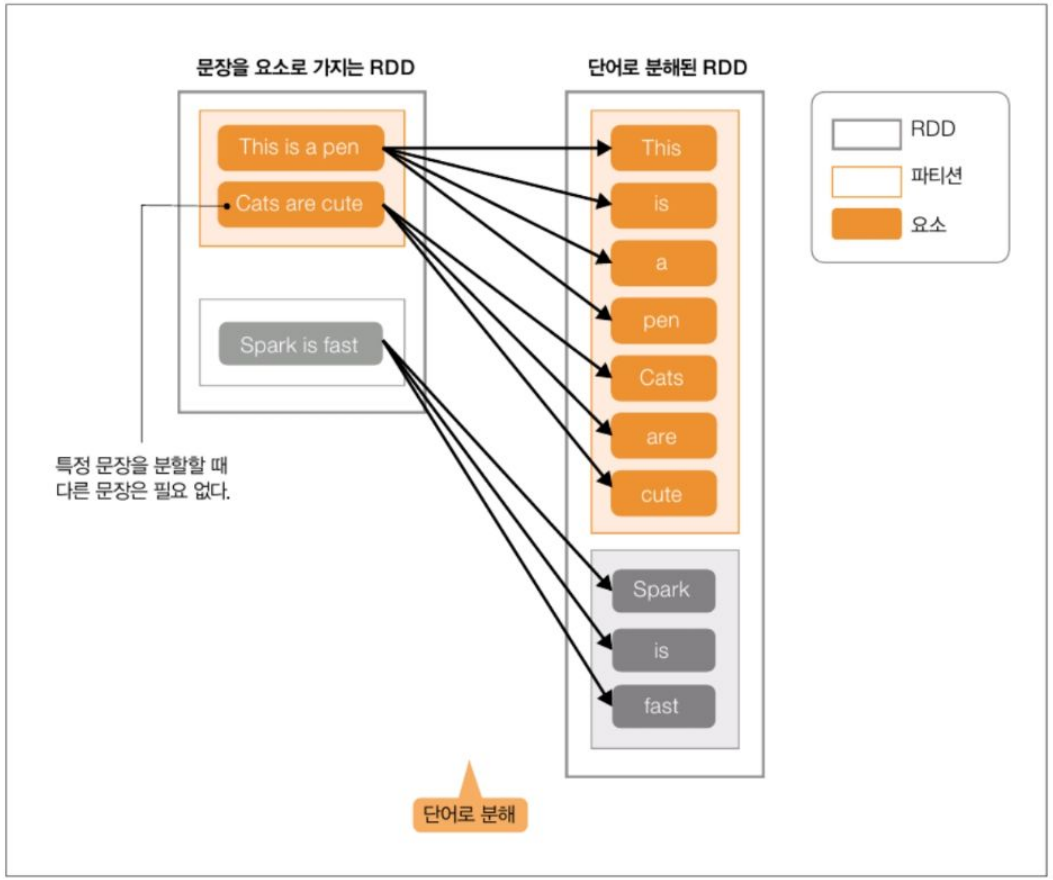
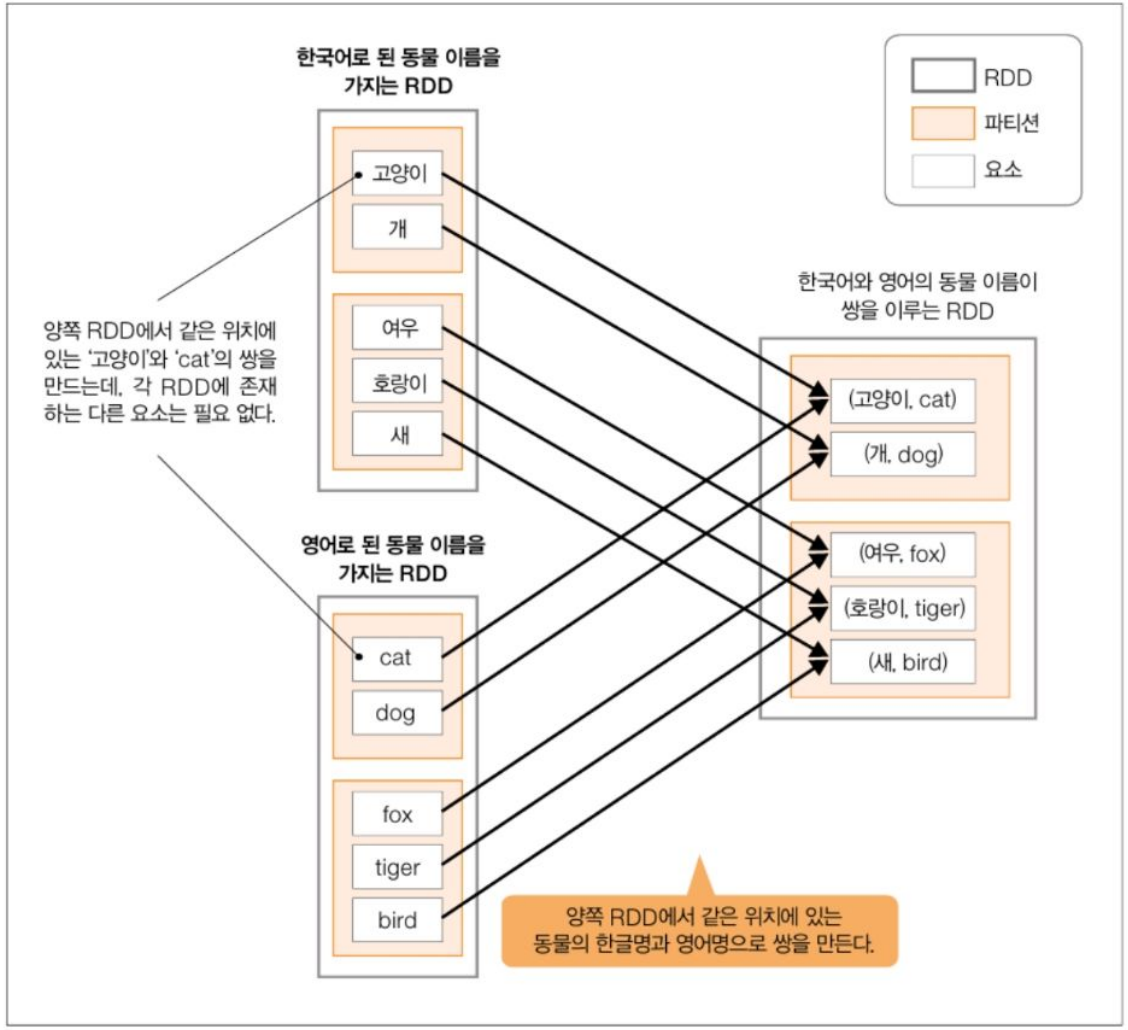


그림 2-6 zip 개요



RDD – 변환

- zip
  - 같은 키를 가지는 요소를 한데 모아 처리하는데, 이때 같은 키를 가지는 요소가 전부 같은 파티션에 있어야 한다.
  - 스파크는 파티션 단위로 독립해 분산처리하기 때문이다.
  - 이때 서로 다른 파티션에 있는 같은 키를 가지는 요소의 자리를 바꾸는 것이 셔플(shuffle)이라고 한다.
  - 셔플은 변환하기 전에 RDD 요소를, 변환 후에 키를 기준으로 각 파티션에 배분한다. 따라서 셔플에 의해 같은 키를 가지는 요소가 같은 파티션에 있도록 보증할 수 있다.

RDD – 변환

그림 2-7 reduceByKey 개요

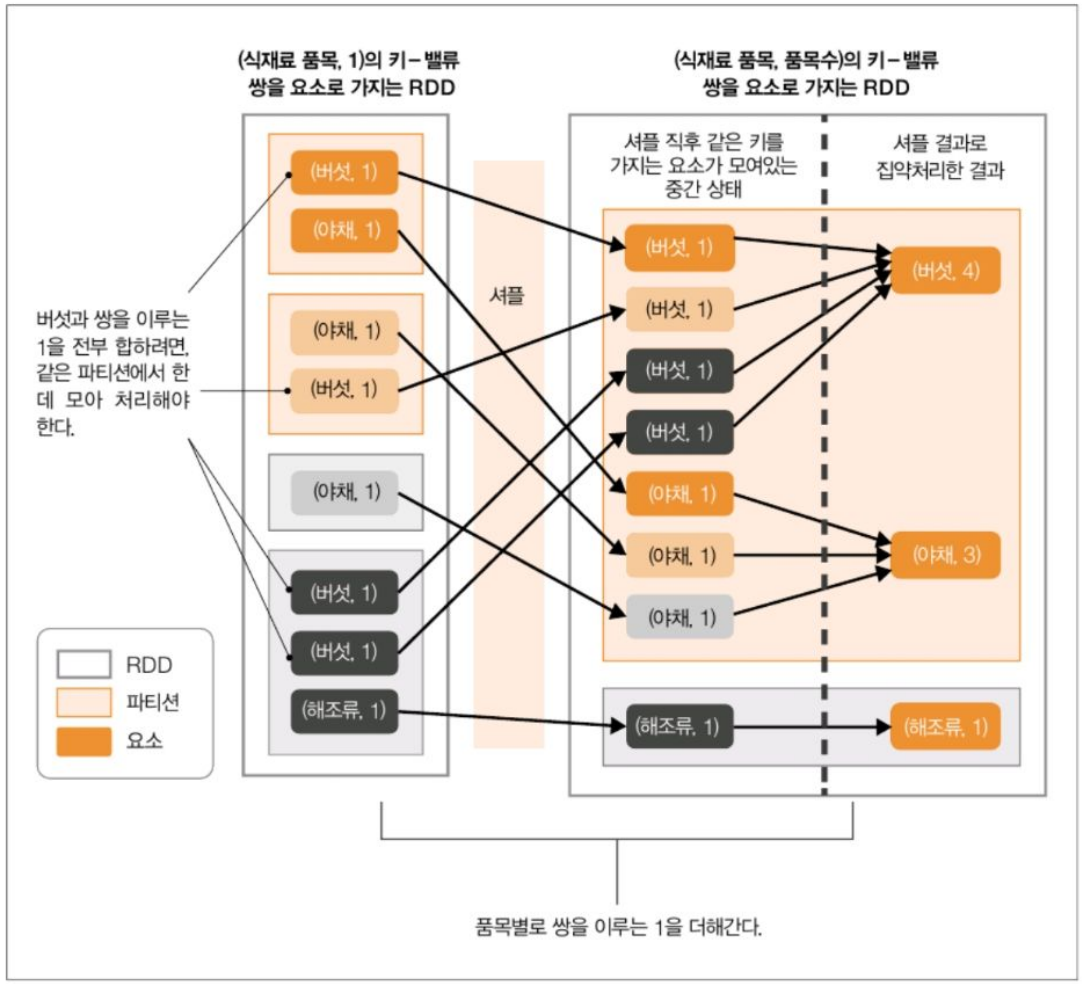
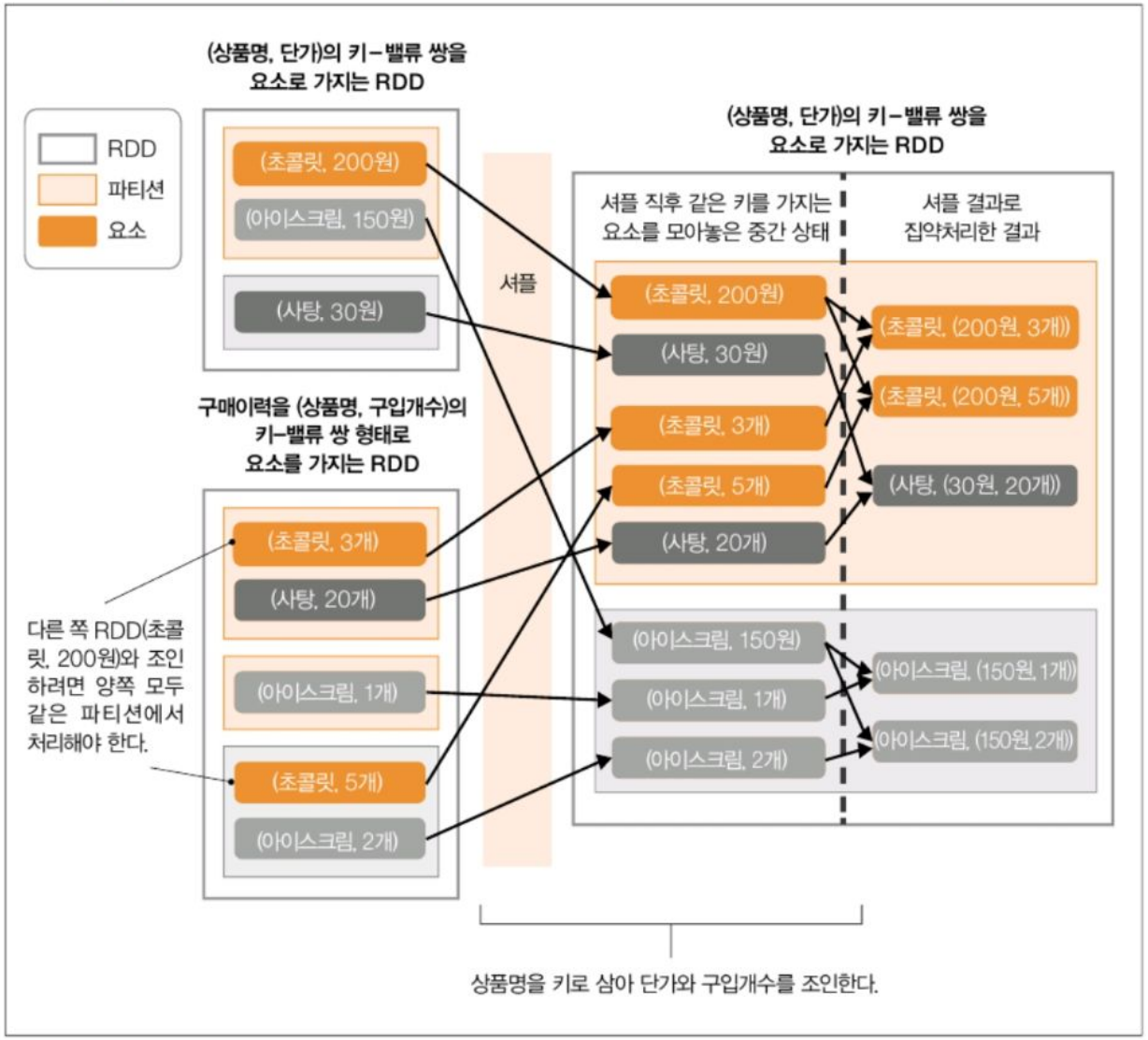


그림 2-8 join 개요



RDD – 변환

- 파티셔너의 역할
  - 셔플할 때 같은 키를 가지는 요소를 같은 파티션에 모이는 것
  - 파티셔너는 변환 후 RDD의 파티션 수와 파티션 이동의 대상이 되는 요소의 키를 기준으로, 요소를 모을 곳(파티션)을 결정
- 스파크는 키의 해싯값을 변환 이후 RDD의 파티션 수로 나눈 나머지를 디폴트 기준으로 삼아 모을 곳(파티션)을 결정

RDD – 액션

- 액션
  - 변환이 RDD로부터 다른 RDD를 얻는 '데이터 가공'에 해당하는 조작이라면, 액션은 RDD 내용을 바탕으로 데이터를 가공하지 않고 원하는 결과를 얻는 조작
  - 액션에는 예를 들면 다음과 같은 것이 있다
    - saveAsTextFile
      - RDD의 내용을 파일로 출력
    - count
      - RDD의 요소 수를 센다.

그림 2-9 saveAsTextFile 개요

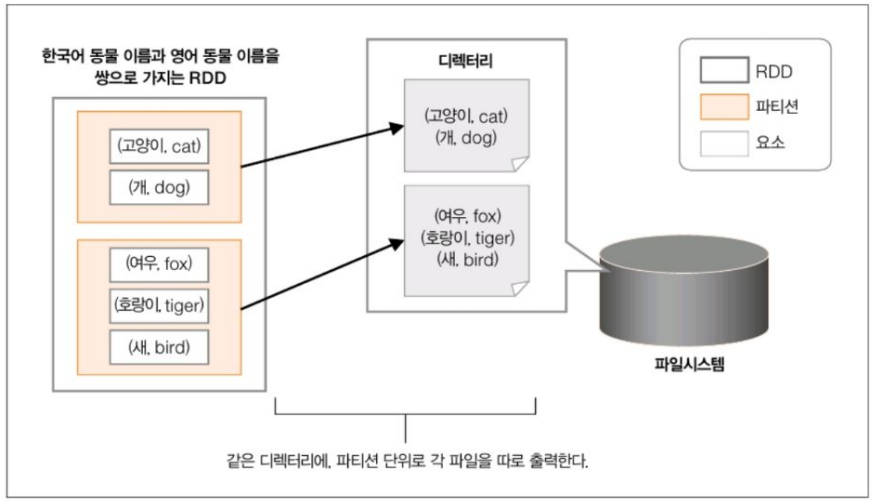
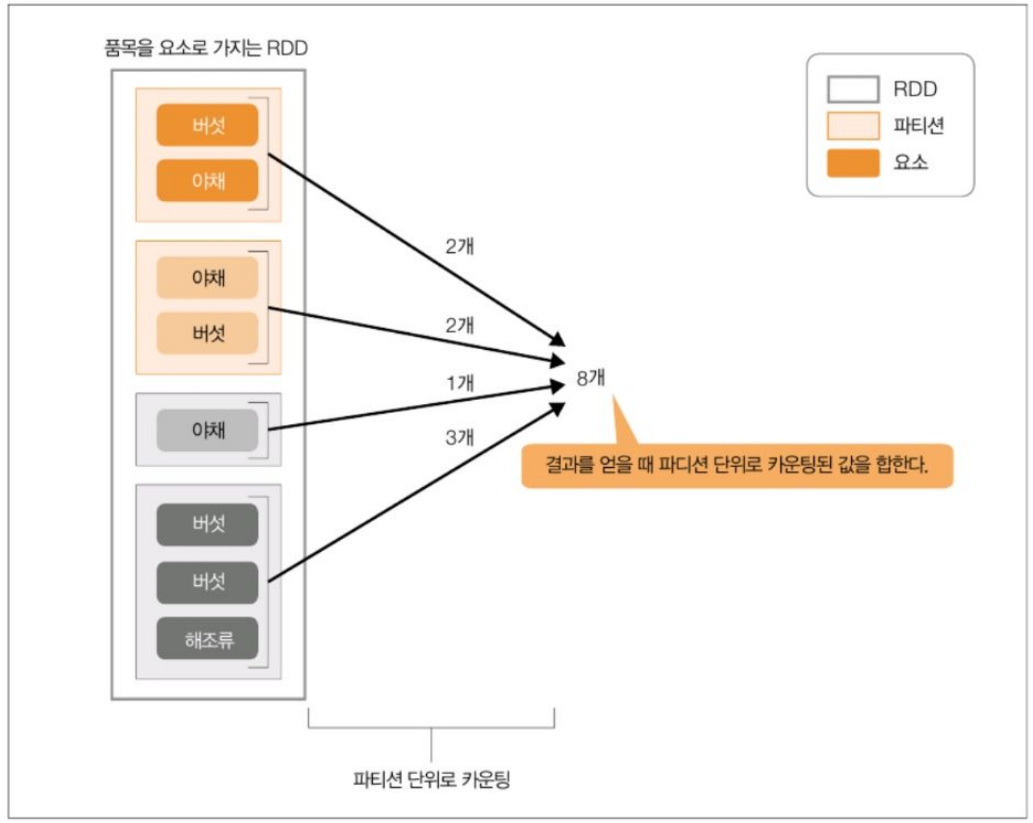


그림 2-10 count 개요



종합 예제

- 이번 예제에서는 미국 교통통계국의 항공운항 데이터 중 일부(csv형태)를 스파크로 분석합니다.
- DataFrame의 스키마 정보를 알아내는 스키마 추론 기능을 사용. 그리고 파일의 첫 로우를 헤더로 지정하는 옵션도 함께 설정

```
# 파이썬 코드
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/data/flight-data/csv/2015-summary.csv")
```

  - DataFrame은 불특정 다수의 로우와 컬럼을 가집니다. 로우의 수를 알 수 없는 이유는 데이터를 읽는 과정이 지연 연산 형태의 트랜스포메이션이기 때문. 스파크는 각 컬럼의 데이터 타입을 추론하기 위해 적은 양의 데이터를 읽음

종합 예제

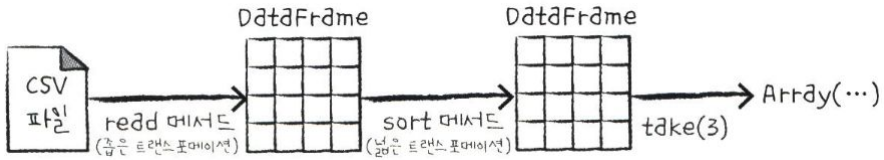
- take 액션을 호출하면 아래의 결과 출력

```
%spark.pyspark
flightData2015.take(3)
```

```
[Row(DEST_COUNTRY_NAME=u'United States', ORIGIN_COUNTRY_NAME=u'Romania', count=15), Row(
NTRY_NAME=u'Ireland', count=344)]
```

' DataFrame에서 CSV 파일을 읽어 로컬 배열이나 리스트 형태로 변환하는 과정

- sort 메서드는 단지 트랜스포메이션이기 때문에 호출 시 데이터에 아무런 변화도 일어나지 않습니다.
- 하지만 스파크는 실행 계획을 만들고 검토하여 클러스터에서 처리할 방법을 알아냅니다.



종합 예제

- 실행 계획

```
%spark.pyspark
flightData2015.sort('count').explain()
```

```
== Physical Plan ==
*(2) Sort [count#12 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#12 ASC NULLS FIRST, 200)
   +- *(1) FileScan csv [DEST_COUNTRY_NAME#10,ORIGIN_COUNTRY_NAME#11,count#12] Batched: fa
s: [], ReadSchema: struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

  - 논리적 실행 계획은 DataFrame의 계보를 정의함. 스파크는 계보를 통해 입력 데이터의 수행한 연산을 전체 파티션에서 어떻게 재연산하는지 알 수 있음. 이 기능은 스파크의 프로그래밍 모델인 함수형 프로그래밍의 핵심

DataFrame과 SQL

- 스파크는 언어에 상관없이 같은 방식으로 트랜스포메이션을 실행할 수 있음
- 사용자가 SQL이나 DataFrame으로 비즈니스 로직을 표현하면 스파크에서 실제 코드를 실행하기 전에 그 로직을 기본 실행 계획으로 컴파일 함
- 스파크 SQL을 사용하면 모든 DataFrame을 테이블이나 뷰로 등록한 후 SQL 쿼리를 사용할 수 있음
- 스파크는 SQL 쿼리를 DataFrame 코드와 같은 실행 계획으로 컴파일하므로 둘 사이의 성능 차이는 없음
- createOrReplaceTempView 메서드를 호출하면 모든 DataFrame을 테이블이나 뷰로 만들 수 있습니다.
- 예제

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

  - 이제 SQL로 데이터를 조회할 수 있음. 새로운 DataFrame을 반환하는 spark.sql 메서드로 SQL 쿼리를 실행

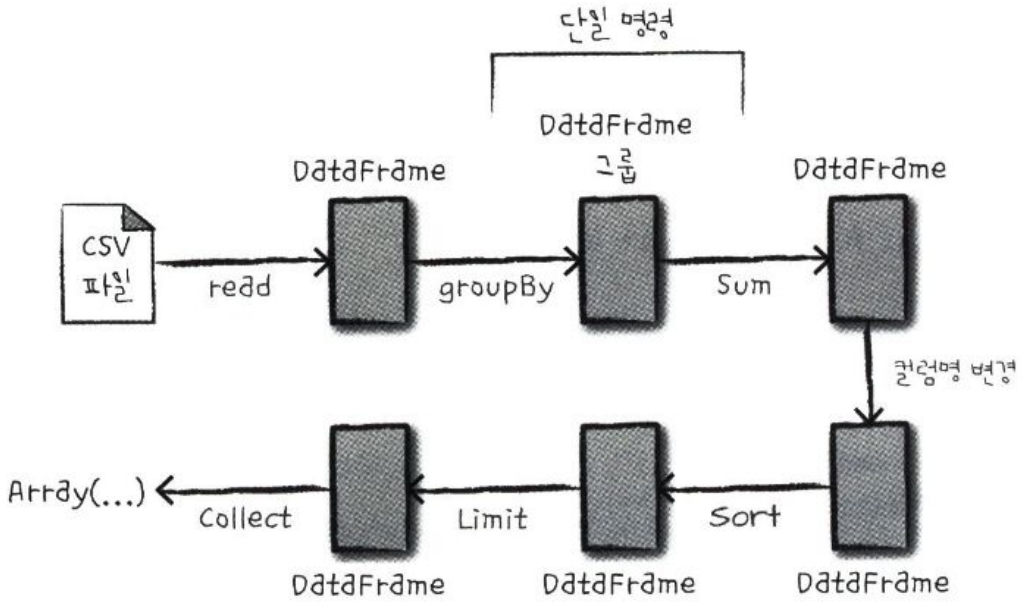
```
sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

dataFrameWay = flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .count()

sqlWay.explain()
dataFrameWay.explain()
```

데이터 전리 단계

- 예제의 결과 각 단계

```
%spark.pyspark


from pyspark.sql.functions import desc

flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
  .sort(desc("destination_total"))\
  .limit(5)\
  .show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```



1. 데이터를 읽기. 스파크는 해당 DataFrame이나 자신의 원본 DataFrame에 액션이 호출되기 전까지 데이터를 읽지 않습니다.
2. 데이터를 그룹화
3. 집계 유형을 지정하기 위해 컬럼 표현식이나 컬럼명을 인수로 사용하는 sum 메서드 사용. 하지만 아무 연산도 일어나지 않습니다.
4. 컬럼명 변경. withColumnRenamed 메서드를 사용하지만 아직 아무일도 일어나지 않는다.
5. 데이터 정렬
6. limt 메서드로 반환 결과의 수 제한
7. 마지막 액션을 수행 이제 DataFrame의 결과를 모으는 프로세스 시작

스파크 기능 둘러보기

- 스파크는 기본 요소인 저수준 API와 구조적 API 그리고 추가 기능을 제공하는 일련의 표준 라이브러리로 구성되어 있음

구조적 API 개요

- 구조적 API는 비정형 로그 파일부터 반정형 CSV 파일, 매우 정형적인 파케이(Parquet) 파일까지 다양한 유형의 데이터를 처리할 수 있음
- 구조적 API에는 다음과 같은 세 가지 분산 컬렉션 API가 존재
  - Dataset
  - DataFrame
  - SQL 테이블과 뷰
- 배치와 스트리밍처리에서 구조적 API를 사용할 수 있음.
- 구조적 API는 데이터 흐름을 정의하는 기본 추상화 개념
- 구조적 API에서 반드시 이해해야 하는 세 가지 기본 개념
  - 타입형(typed)/비타입형(untyped) API의 개념과 차이점
  - 핵심 용어
  - 스파크가 구조적 API의 데이터 흐름을 해석하고 클러스터에서 실행하는 방식

DataFrame과 Dataset

- 스파크는 DataFrame과 Dataset이라는 두 가지 구조화된 컬렉션 개념을 가지고 있음
- DataFrame과 Dataset은 잘 정의된 로우와 컬럼을 가지는 분산 테이블 형태의 컬렉션임.
- 각 컬럼은 다른 컬럼과 동일한 수의 로우를 가져야 함
- 컬렉션의 모든 로우는 같은 데이터 타입 정보를 가지고 있어야 함
- DataFrame과 Dataset은 결과를 생성하기 위해 어떤 데이터에 어떤 연산을 적용해야 하는지 정의하는 지연연산의 실행계획이며, 불변성을 가짐.
- 액션을 호출하면 스파크는 트랜스포메이션을 실제로 실행하고 결과를 반환

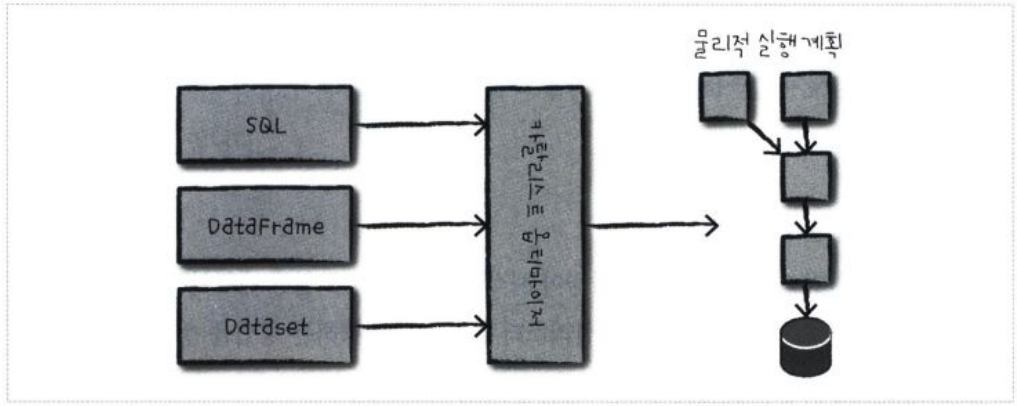
스키마

- 스키마는 DataFrame의 컬럼명과 데이터 타입을 정의
- 스키마는 데이터소스에서 얻거나 직접 정의할 수 있음
- 스파크는 실행 계획 수립과 처리에 사용하는 자체 데이터 타입 정보를 가지고 있는 카탈리스트엔진을 사용

구조적 API의 실행 과정

- 아래 내용은 스파크 코드가 클러스터에 실제 처리되는 과정임
  - DataFrame/Dataset/SQL을 이용해 코드를 작성합니다.
  - 정상적인 코드라면 스파크가 논리적 실행 계획으로 변환합니다.
  - 스파크는 논리적 실행 계획을 물리적 실행 계획으로 변환하여 그 과정에서 추가적인 최적화를 할 수 있는지 확인
  - 스파크는 클러스터에서 물리적 실행 계획(RDD 처리)을 실행합니다.

그림 4-1 카탈리스트 옵티마이저

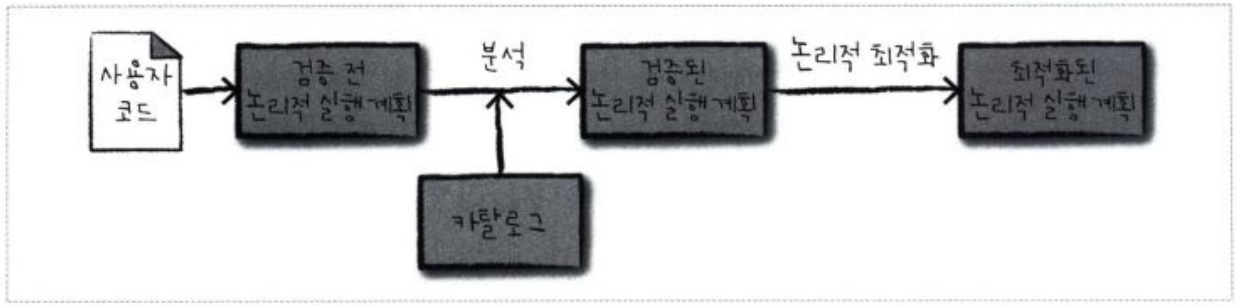


- 카탈리스트 옵티마이저는 코드를 넘겨받고 실제 실행 계획을 생성합니다.

논리적 실행 계획

- 첫 번째 실행 단계에서 사용자 코드를 논리적 실행 계획으로 변환함.

그림 4-2 구조적 API의 논리적 실행 계획 수립 과정

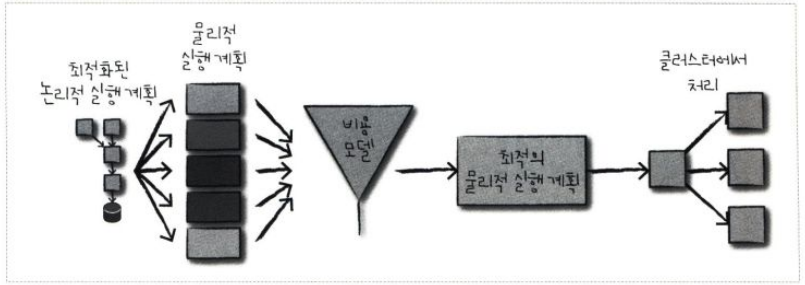


  - 논리적 실행 계획 단계에서는 추상적 트랜스포메이션만 표현하고 익스큐터의 정보를 고려하지 않음
  - 사용자의 다양한 표현식을 최적화된 버전으로 변환
  - 이 과정으로 사용자 코드는 검증 전 논리적 실행 계획으로 변환하지만, 아직은 실행 계획은 검증하지 않은 상태
  - 스파크 분석기는 컬럼과 테이블을 검증하기 위해 카탈로그, 모든 테이블의 저장소 그리고 DataFrame 정보를 활용
    - 필요한 테이블이나 컬럼이 카탈로그에 없다면 검증 전 논리적 실행 계획이 만들어지지 않음
  - 테이블과 컬럼에 대한 검증 결과는 카탈리스트 옵티마이저로 전달
  - 카탈리스트 옵티마이저는 조건절 푸시 다운이나 선택절 구문을 이용해 논리적 실행 계획을 최적화하는 규칙의 모음임

물리적 실행 계획

- 스파크는 실행 계획이라고도 불리는 물리적 실행 계획은 논리적 실행 계획을 클러스터 환경에서 실행하는 방법을 정의함.
- 다양한 물리적 실행 전략을 생성하고 비용 모델을 이용해서 비교한 후 최적의 전략을 선택
  - 비용을 비교하는 한 가지 예는 사용하려는 테이블의 크기나 파티션 수등의 물리적 속성을 고려해 지정된 조인 연산 수행에 필요한 비용을 계산하고 비교

그림 4-3 물리적 실행 계획 수립 과정



- 물리적 실행 계획은 일련의 RDD와 트렌스포메이션으로 변환
- DataFrame, Dataset, SQL로 정의된 쿼리를 RDD 트랜스포메이션으로 컴파일
- 물리적 실행 계획을 선정한 다음 저수준 프로그래밍 인터페이스인 RDD를 대상으로 모든 코드를 실행
  - 스파크는 런타임에 전체 태스크나 스테이지를 제거할 수 있는 자바 바이트 코드를 생성해 추가적인 최적화를 수행
  - 마지막으로 스파크는 처리 결과를 사용자에게 반환

구조적 API 기본 연산

- DataFrame은 Row 타입의 레코드와 각 레코드에 수행할 연산 표현식을 나타내는 여러 컬럼으로 구성되어 있음
- 스키마는 각 컬럼명과 데이터 타입을 정의함
- DataFrame의 파티셔닝은 DataFrame이나 Dataset이 클러스터에서 물리적으로 배치되는 형태를 정의
- 파티셔닝 스키마는 파티션을 배치하는 방법을 정의함

```
# 파이썬 코드
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
```

```
df.printSchema()
```

  - 스키마는 관련된 모든 것을 하나로 묶는 역할을 함

스키마

- 스키마는 DataFrame의 컬럼명과 데이터 타입을 정의함
- 데이터소스에서 스키마를 얻거나 직접 정의할 수 있음
- 아래 예제는 미국 교통통계국이 제공하는 항공운항 데이터로 줄로 구분된 JSON 데이터

```
# 파이썬 코드
spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema
```

```
StructType(List(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true)))
```

  - 스키마는 여러 개의 StructField 타입 필드로 구성된 StructType 객체임
  - StructField는 이름, 데이터 타입, 컬럼이 값이 없거나 null일 수 있는지 지정하는 Boolean 값을 가짐

스키마

- 스키마는 복잡한 데이터 타입인 StructType을 가질 수 있음

```
# 파이썬 코드
from pyspark.sql.types import StructField, StructType, StringType, LongType

myManualSchema = StructType([
  StructField("DEST_COUNTRY_NAME", StringType(), True),
  StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
  StructField("count", LongType(), False, metadata={"hello":"world"})
])
df = spark.read.format("json").schema(myManualSchema)\
  .load("/data/flight-data/json/2015-summary.json")
```

컬럼과 표현식

- 스파크의 컬럼은 R, Python의 DataFrame 컬럼과 유사함
- 사용자는 표현식으로 DataFrame의 컬럼을 선택, 조작, 제거할 수 있음
- 스파크의 컬럼은 표현식을 사용해 레코드 단위로 계산한 값을 단순하게 나타내는 논리적 구조
- 컬럼의 실젯값을 얻으려면 로우가 필요하고, 로우를 얻으려면 DataFrame이 필요
- DataFrame을 통하지 않으면 외부에서 컬럼에 접근할 수 없음
- 컬럼 내용을 수정할려면 반드시 DataFrame의 스파크 트랜스포메이션을 사용해야 함

컬럼과 표현식

- 컬럼을 생성하고 참조할 수 있는 col 함수나 column 함수를 사용하는 것이 가장 간단함

```
# 파이썬 코드
from pyspark.sql.functions import col, column

col("someColumnName")
column("someColumnName")
```

- 컬럼이 DataFrame에 있는지 없을지는 알 수 없음. 컬럼은 컬럼명을 카탈로그에 저장된 정보와 비교하기 전까지 미확인 상태로 남아 있음
- 명시적 컬럼 참조
  - .DataFrame의 컬럼은 col 메서드로 참조함. col 메서드는 조인 시 유용하게 사용할 수 있음.

컬럼과 표현식

- 표현식
  - DataFrame을 정의할 때 컬럼은 표현식
  - 표현식은 DataFrame 레코드의 여러 값에 대한 트렌스포메이션 집합을 의미함
  - 표현식은 expr 함수로 가장 간단히 사용할 수 있음
    - expr("someCol")은 col("someCol") 구문과 동일하게 동작
- 표현식으로 컬럼 표현
  - 컬럼은 표현식의 일부 기능을 제공
  - col() 함수를 호출해 컬럼에 트랜스포메이션을 수행하려면 반드시 컬럼 참조를 사용함
  - expr 함수의 인수로 표현식을 사용하면 표현식을 분석해 트랜스포메이션과 컬럼 참조를 알아낼 수 있으며, 다음 트랜스포메이션에 컬럼 참조를 전달할 수 있음
    - expr("someCol - 5"), col("someCol") - 5, expr("someCol") - 5는 스파크가 연산 순서를 지정하는 논리적 트리로 컴파일하기 때문에

      모두 같은 트랜스포메이션 과정을 거침

      컬럼은 단지 표현식일 뿐

      컬럼과 컬럼의 트랜스포메이션은 파싱된 표현식과 동일한 논리적 실행 계획으로 컴파일 됨

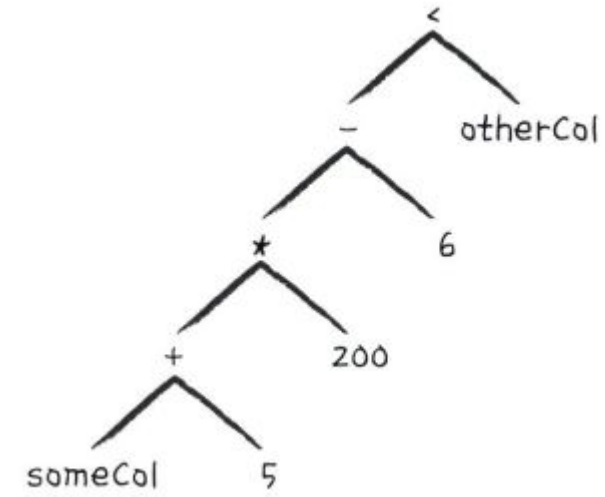
컬럼과 표현식

- 표현식으로 컬럼 표현
  - col 메서드 사용

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```

  - 표현식 사용

```
# 파이썬 코드
from pyspark.sql.functions import expr

expr("(((someCol + 5) * 200) - 6) < otherCol")
```



지향성 비순환 그래프(DAG)

컬럼과 표현식

- DataFrame 컬럼에 접근하기
  - printSchema 메서드로 DataFrame의 전체 컬럼 정보를 확인할 수 있음
  - 프로그래밍 방식으로 컬럼에 접근할 때는 아래와 같이 columns 속성을 사용함

```
spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
  .columns
```

레코드와 로우

- 스파크에서 DataFrame의 각 로우는 하나의 레코드임
- 스파크는 레코드를 Row 객체로 표현함
- 스파크는 값을 생성하기 위해 컬럼 표현식으로 Row 객체를 다룸
- Row 객체는 내부에 바이트 배열을 가지고, 이 바이트 배열 인터페이스는 오직 컬럼 표현식으로만 다룰 수 있으므로 사용자에게 절대 노출되지 않음

```
df.first()
```

  - DataFrame의 first 메서드로 로우를 확인

레코드와 로우

- 로우 생성하기
  - 각 컬럼에 해당하는 값을 사용해 Row 객체를 직접 생성할 수 있음
  - Row 객체는 스키마 정보를 가지고 있지 않음. DataFrame만 유일하게 스키마를 갖고 있음
  - Row 객체를 직접 생성하려면 DataFrame의 스키마와 같은 순서로 값을 명시해야 함

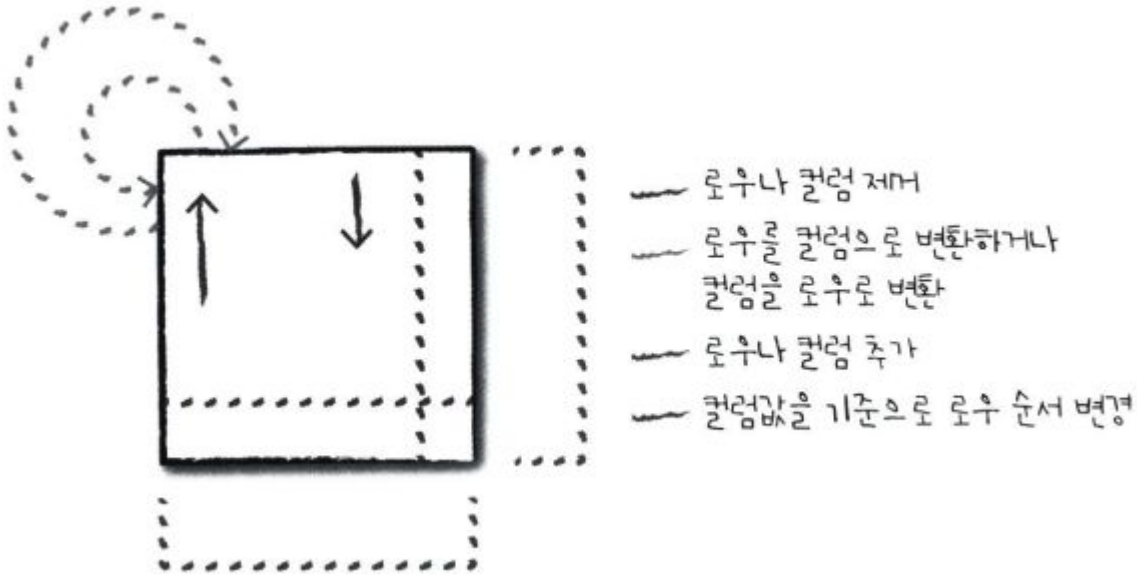
```
# 파이썬 코드
from pyspark.sql import Row

myRow = Row("Hello", None, 1, False)
```

  - 로우의 데이터에 접근하는 방법은 아주 쉽게 되어 있음. 원하는 위치를 지정하면 됨

```
# 파이썬 코드
myRow[0]
myRow[2]
```

DataFrame의 트랜스포메이션

- DataFrame을 다루는 방법
  - 로우나 컬럼 추가
  - 로우나 컬럼 제거
  - 로우를 컬럼으로 변환하거나, 그 반대로 변환
  - 컬럼값을 기준으로 로우 순서 변경



- DataFrame 생성

```
# 파이썬 코드
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
df.createOrReplaceTempView("dfTable")
```

```
+-----+----+-----+
| some| col|names|
+-----+----+-----+
|Hello|null|    1|
+-----+----+-----+
```

```
# 파이썬 코드
from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType, LongType

myManualSchema = StructType([
  StructField("some", StringType(), True),
  StructField("col", StringType(), True),
  StructField("names", LongType(), False)
])

myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame([myRow], myManualSchema)
myDf.show()
```

selec와 selectExpr

- select와 selectExpr 메서드를 사용하면 데이터 테이블에 SQL을 실행하는 것처럼 DataFrame에서도 SQL을 사용할 수 있습니다.

```
# 파이썬 코드
df.select("DEST_COUNTRY_NAME").show(2)
```

```
-- SQL
SELECT DEST_COUNTRY_NAME FROM dfTable LIMIT 2
```

- 같은 형태의 쿼리로 여러 컬럼을 선택할 수 있음. 여러 컬럼을 선택하려면 select 메서드에서 원하는 컬러명추가

```
# 파이썬 코드
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)

-- SQL
SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM dfTable LIMIT 2
```

- expr 함수는 단순 컬

```
# 파이썬 코드
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

-- SQL
SELECT DEST_COUNTRY_NAME as destination FROM dfTable LIMIT 2
```

select와 selectExpr

- selectExpr
  - select 메서드에서 expr 함수를 사용하는 패턴을 효율적으로 할 수 있게 해줌

```
# 파이썬 코드
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)
```

  - selectExpr 메서드는 새로운 DataFRame을 생성하는 복잡한 표현식을 간단하게 만드는 도구 임.
  - 컬럼을 식별할 수 있다면 모든 유효한 비집계형 SQL 구문을 지정할 수 있음

```
# 파이썬 코드
df.selectExpr(
    "*", # 모든 원본 컬럼 포함
    "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")\
  .show(2)
```

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
```

  - DataFrame의 컬럼에 대한 집계 함수를 지정할 수 있음

```
# 파이썬 코드
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)

-- SQL
SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FROM dfTable LIMIT 2
```

스파크 데이터 타입으로 변환하기

- 명시적인 값은 상숫값일 수 있고, 추후 비교에 사용할 무언가가 될 수도 있음. 이때 리터럴을 사용
- 리터럴은 프로그래밍 언어의 리터럴값을 스파크가 이해할 수 있는 값으로 변환
  - 파이썬 코드
    - from pyspark.sql.functions import lit
    - df.select(expr("*"), lit(1).alias("one")).show
  - SQL
    - SELECT *, 1 as One FROM dfTable LIMIT 2

```
+-----------------+-------------------+-----+---+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----------------+-------------------+-----+---+
|    United States|            Romania|   15|  1|
|    United States|            Croatia|    1|  1|
+-----------------+-------------------+-----+---+
```

- 어떤 상수나 프로그래밍으로 생성된 변숫값이 특정 컬럼의 값보다 큰지 확인할 때 리터럴을 사용
- 리터럴은 프로그래밍 언어의 리터럴값을 스파크가 이해할 수 있는 값으로 변환
- SQL에서 리터럴은 상수값을 의미

컬럼 추가하기

- DataFrame에 신규 컬럼을 추가하는 공식적인 방법은 DataFrame의 withColumn 메서드를 사용하는 것임
- 숫자 1을 값으로 가지는 컬럼을 추가하는 예제
  - 파이썬 코드
    - df.withColumn("numberOne", lit(1)).shov

```
+-----------------+-------------------+-----+---------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|numberOne|
+-----------------+-------------------+-----+---------+
|    United States|            Romania|   15|        1|
|    United States|            Croatia|    1|        1|
+-----------------+-------------------+-----+---------+
```

- 출발지와 도착지가 같은지 여부를 불리언 타입으로 현하는 표현하는 예제
  - 파이썬 코드
    - df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))
  - withColumn 메서드는 두 개의 인수를 사용
    - 하나는 컬럼명, 다른 하나는 값을 생성할 표현식
    - withColumn 메서드로 컬럼명을 변경할 수도 있다는 것

컬러명 변경하기

- withColumn 메서드 대신 withColumnRenamed 메서드로 컬럼명을 변경할 수도 있음
- withColumnRenamed 메서드는 첫 번째 인수로 전달된 컬럼명을 두 번째 인수의 문자열로 변경

```
# 파이썬 코드
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns

... dest, ORIGIN_COUNTRY_NAME, count.
```

예약 문자와 키워드

- 공백이나 하이픈(-) 같은 예약 문자는 컬럼명에 사용할 수 없습니다.
- 예약 문자를 컬럼명에 사용하려면 벡터(`) 문자를 이용해 이스케이핑해야 합니다.
- withColumn 메서드를 사용해 예약 문자가 포함된 컬럼을 생성하는 예제

```
# 파이썬 코드
dfWithLongColName = df.withColumn(
    "This Long Column-Name",
    expr("ORIGIN_COUNTRY_NAME"))
```

- withColumn 메서드의 첫 번째 인수로 새로운 컬럼명을 나타내는 문자열을 지정했기 때문에 이스케이프 문자가 필요 없습니다. 하지만 다음 예제에서는 표현식으로 컬럼을 참조하므로 백틱(`) 문자를 사용

```
# 파이썬 코드
dfWithLongColName.selectExpr(
    "`This Long Column-Name`",
    "`This Long Column-Name` as `new col`")\
  .show(2)

dfWithLongColName.createOrReplaceTempView("dfTableLong")
```

컬럼 제거 및 변경

- DataFrame에서 컬럼을 제거하는 방법
- select 메서드로 컬럼을 제거할 수 있지만 컬럼을 제거하는 메서드인 drop을 사용할 수도 있음
  - df.drop("ORIGIN_COUNTRY_NAME").columns
- 다수의 컬럼명을 drop 메서드의 인수로 사용해 컬럼을 한꺼번에 제거할 수 있음
  - dfWithLongColName.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")