Apache Spark

2017-08-10, Academy

김상우 @ VCNC
kevin@between.us

Index

- 빅데이터 개요
- Apache Spark 개요
- 데이터 분석을 위한 Scala
- Spark 동작원리 & 실습

대용량 데이터 처리 기술 개요

빅데이터?

빅 데이터 란 기존 데이터베이스 관리도구의 능력을 넘어서는 대량의 정형 또는 심지어 데이터베이스 형태가 아닌 비정형의 데이터 집합조차 포함한 데이터로부터 가치를 추출하고 결과를 분석하는 기술이다.

- 위키피디아

빅데이터?

빅 데이터 란 **기존 데이터베이스 관리도구**의 능력을 넘어서는 **대량**의 **정형** 또는 심지어 데이터베이스 형태가 아닌 **비정형**의 데이터 집합조차 포함한 데이터로부터 **가치를 추출하고 결과를 분석**하는 **기술**이다.

- 위키피디아

빅데이터?

빅 데이터 란 **기존 데이터베이스 관리도구**의 능력을 넘어서는 **대량**의 **정형** 또는 심지어 데이터베이스 형태가 아닌 **비정형**의 데이터 집합조차 포함한 데이터로부터 **가치를 추출하고 결과를 분석**하는 **기술**이다.

- 위키피디아

- SQL기반의 데이터베이스
- 주로 컴퓨터 1대에서 돌아감, 고성능이 필요하면 좋은(비싼) 컴퓨터 사용
- 컴퓨터 1대로 처리할 수 있는 용량과 성능의 한계

빅데이터?

빅 데이터 란 **기존 데이터베이스 관리도구**의 능력을 넘어서는 **대량**의 **정형** 또는 심지어 데이터베이스 형태가 아닌 **비정형**의 데이터 집합조차 포함한 데이터로부터 **가치를 추출하고 결과를 분석**하는 **기술**이다.

- 위키피디아

- 대량 - 컴퓨터 1대로 처리할 수 없는 양 (수십TB 이상)

- 3V (by IBM) - Volume, Velocity, Variety

빅데이터?

빅 데이터 란 **기존 데이터베이스 관리도구**의 능력을 넘어서는 **대량**의 **정형** 또는 심지어 데이터베이스 형태가 아닌 **비정형**의 데이터 집합조차 포함한 데이터로부터 **가치를 추출하고 결과를 분석**하는 **기술**이다.

- 위키피디아

- 기술 - 컴퓨터 1대로 처리하지 못하므로, 여러대를 연결해서 데이터를 저장하고 처리하자!
- 주로 구글 등 검색엔진 회사들이 웹 전체를 저장하고 처리하려다보니 기술 개발이 필요하게됨
- 구글이 이끌고, 야후 등이 오픈소스를 통해 (하둡) 적극 지원, 접근하기 쉬워지고 널리 쓰이기 시작
- 빅데이터 기술 = 대부분 하둡이라고 생각해도 무방

빅데이터?

빅 데이터 란 **기존 데이터베이스 관리도구**의 능력을 넘어서는 **대량**의 **정형** 또는 심지어 데이터베이스 형태가 아닌 **비정형**의 데이터 집합조차 포함한 데이터로부터 **가치를 추출하고 결과를 분석**하는 **기술**이다.

- 위키피디아

- SQL기반의 데이터는 거의 행렬 형태로 정형화된 데이터였으나 일반 문서 (웹 문서) 등과 같이 비정형화된 데이터도 초점

빅데이터?

빅 데이터 란 **기존 데이터베이스 관리도구**의 능력을 넘어서는 **대량**의 **정형** 또는 심지어 데이터베이스 형태가 아닌 **비정형**의 데이터 집합조차 포함한 데이터로부터 **가치를 추출하고 결과를 분석**하는 **기술**이다.

- 위키피디아

- 데이터를 저장만 해서는 쓸모가 없음
- 데이터를 읽어드리고, 변환하고, 핵심을 추출하는 것도 마찬가지로 컴퓨터 1대로 할 수 있는 것보다 훨씬 빨라져야 함
- 맵리듀스 (MapReduce) - 분산 데이터 처리
- 현재는 스파크 (Apache Spark) 가 널리 쓰임

빅데이터?

빅 데이터 란 기존 데이터베이스 관리도구의 능력을 넘어서는 대량의 정형 또는 심지어 데이터베이스 형태가 아닌 비정형의 데이터 집합조차 포함한 데이터로부터 가치를 추출하고 결과를 분석하는 기술이다.

- 위키피디아

왜 데이터를 분석하는가?

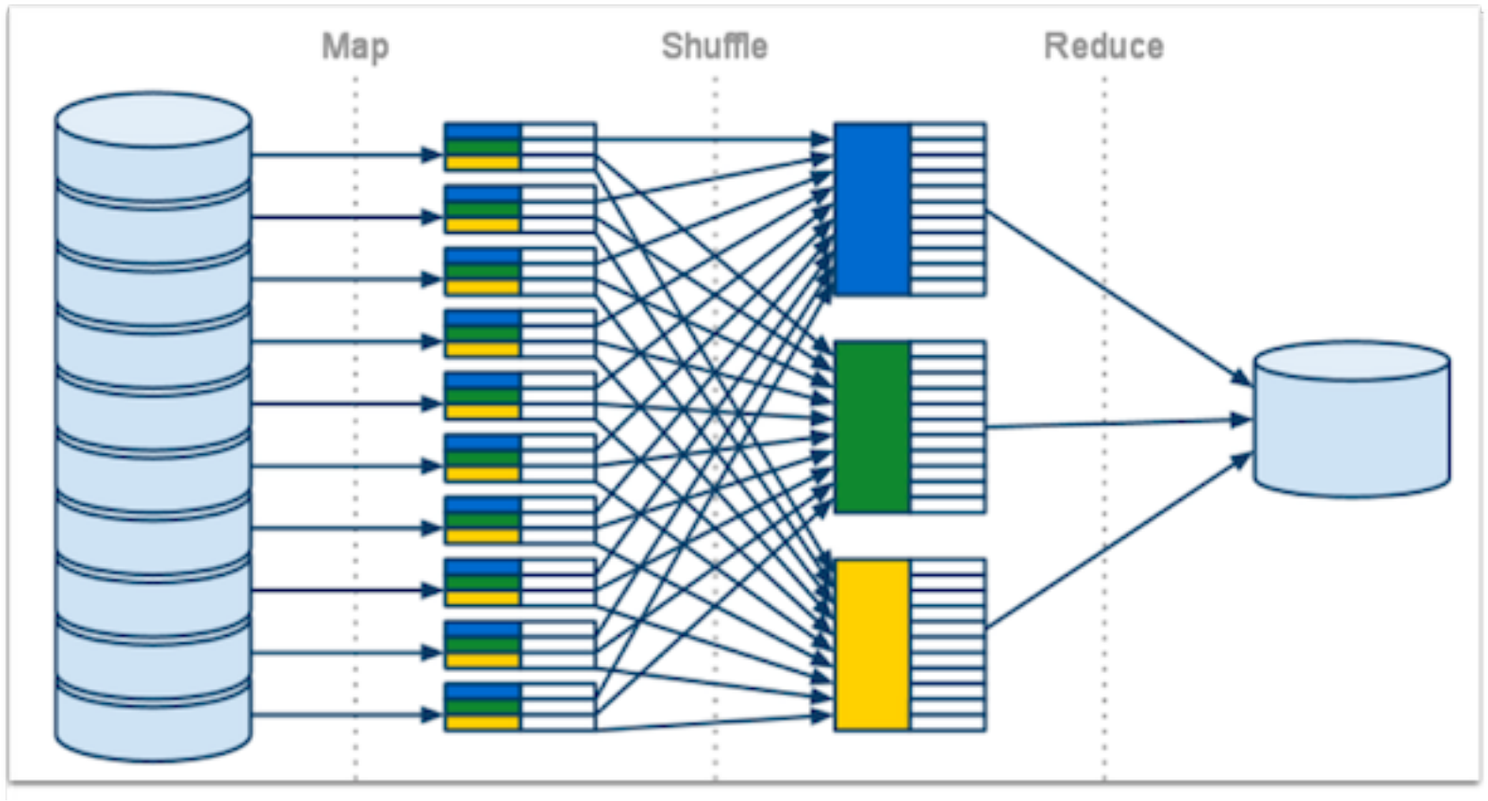
- 비싼 비용에도 불구하고 큰 양의 데이터를 분석하는 이유
- 실제로 사업에 도움이 되기 때문!
- 광고비즈니스, 커머스, 금융 분야에서는 데이터가 매출과 직결되기에 비싼 비용에도 데이터 분석에 공을 들임
- 데이터 분석이 쉽고 저렴해지면서, 일반 서비스 회사들에서도 독자적인 데이터 분석을 하는 경우 많음
  - BI (Business Intelligence) 데이터 분석을 통해 인사이트를 얻고 제품이나 경영 전략을 세우는 일

빅데이터의 시초: GFS

- Google File System 논문 (2003년)
- 막대한 양의 웹 문서를 저장 조회해야 하는데, 컴퓨터 1대로는 당연히 처리가 불가능
- 저렴한 하드웨어를 사용하면서, 대신 중복저장을 통해 파일 유실을 방지
- 파일을 새로 추가하는데 집중, 삭제나 파일 덮어쓰기는 어려움
- Latency보다 Throughput 을 중시
- 클러스터 댓수를 늘릴수록 저장용량과 throughput이 점점 올라감

빅데이터의 시초: MapReduce

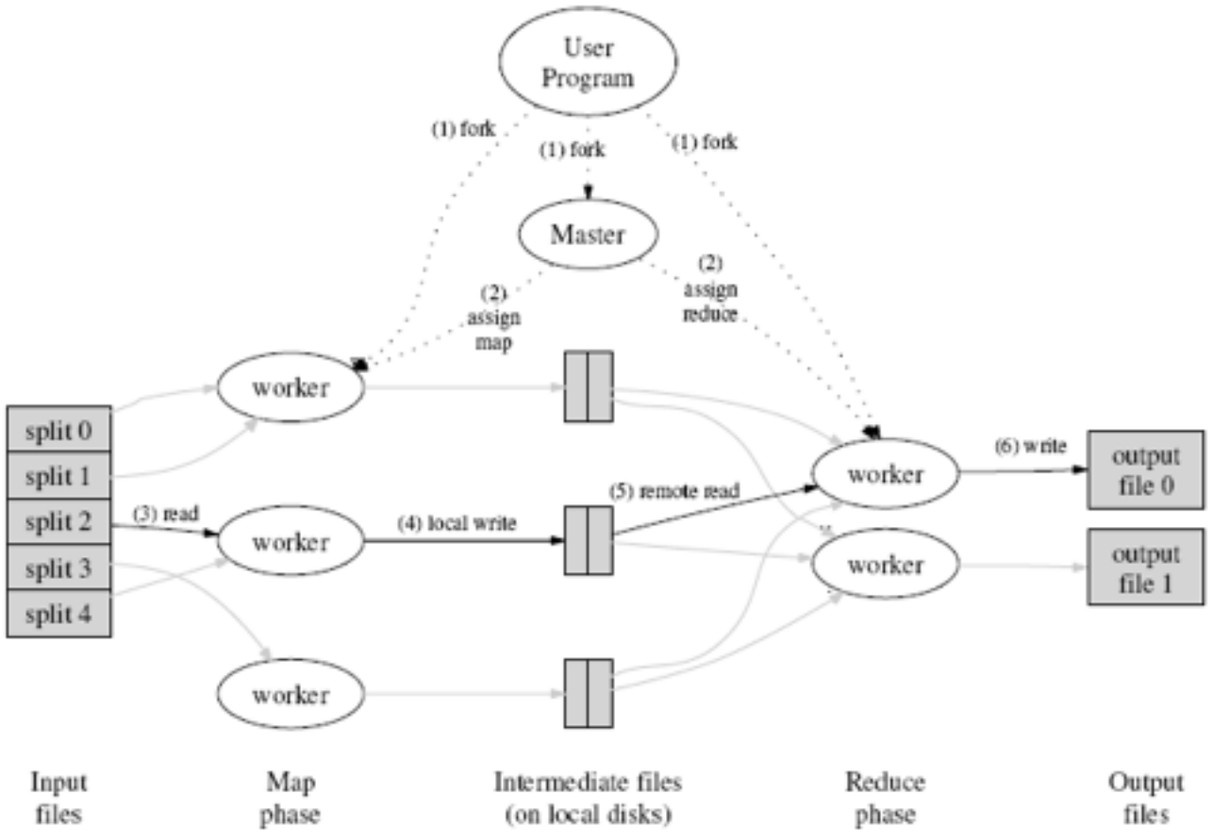
- Google MapReduce (2004년 논문)
- 여러대의 분산 저장소에 존재하는 데이터를 변환하거나 계산하기 위한 프레임워크
- Functional Programming의 Map() 함수와 Reduce() 함수를 조합하여 효율적으로 분산 환경에서 다양한 계산을 함



빅데이터의 시초: MapReduce

- Map() : A인 데이터를 B로 변환시키는 계산을 리스트에 대해 수행
- List(1,2,3).map(x => x * 2) // result: List(2,4,6)
- Reduce() : 리스트에 들어있는 A, B, C 를 특정 룰에 의해 합치는 작업
- List(1,2,3).reduce((a,b) => a + b) // result
- Map() 과 Reduce() 를 조합하면 다양한 작업을 수행할 수 있음

빅데이터의 시초: MapReduce



- Input 파일을 잘게 나눠서 여러 worker에서 나눠서 Map() 작업 수행
- 중간 파일을 저장한후, 이를 수합해서 Reduce() 작업 수행
- Reduce()를 수행한 워커에서 각각의 결과물을 저장

빅데이터의 시초 2: Hadoop

- GFS(2003년)와 MapReduce(2004년) 논문을 보고, Doug Cutting과 Mike Cafarella가 이를 오픈소스로 구현
- 야후에서 프로젝트를 하던 중 그 한 부분으로 만듬, 이후 오픈소스로 공개 (2006년)
- Hadoop: 아들의 노란 코끼리 장난감의 이름을 따서 지음

MapReduce 개념

- Map() : A인 데이터를 B로 변환시키는 계산을 리스트에 대해 수행
- List(1,2,3).map(x => x * 2) // result: List(2,4,6)
- Reduce() : 리스트에 들어있는 A, B, C 를 특정 룰에 의해 합치는 작업
- List(1,2,3).reduce((a,b) => a + b) // result
- Map() 과 Reduce() 를 조합하면 다양한 작업을 수행할 수 있음

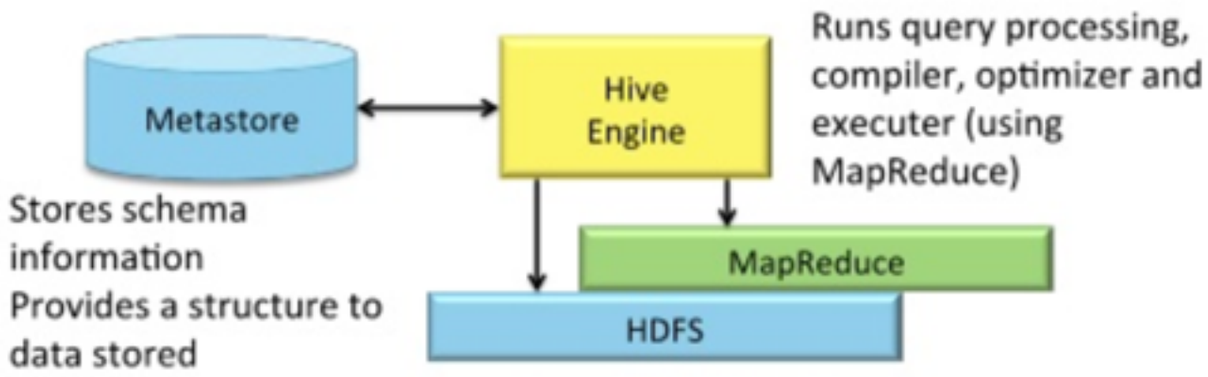
MapReduce 코드

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}

  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Hadoop Hive

- Hive
  - SQL로 분석 쿼리를 실행하면, 이를 MapReduce코드로 변환하여주는 도구
  - MapReduce 코드는 작성하기 아주 불편하므로 큰 인기를 끔
  - 현재까지도 많이 사용됨



하둡 생태계의 많은 프로젝트들

- Pig
  - Pig Latin이라는 하이레벨 언어로 MapReduce를 실행할 수 있는 도구
  - Netflix 등에서 사용하면서 주목을 받았고, 현재는 거의 사용되지 않음
- Tez
  - MapReduce의 성능적, 표현적 한계를 극복하고자 하는 실행엔진
  - Spark 의 급부상으로 거의 주목을 받지 못함
- Impala
  - MapReduce 기반의 Hive 의 느린 응답성을 개선한 도구
  - Spark의 급부상으로 (Spark SQL) 초기에 약간 주목을 받다가 현재는 거의 사용되지 않음

Apache Spark

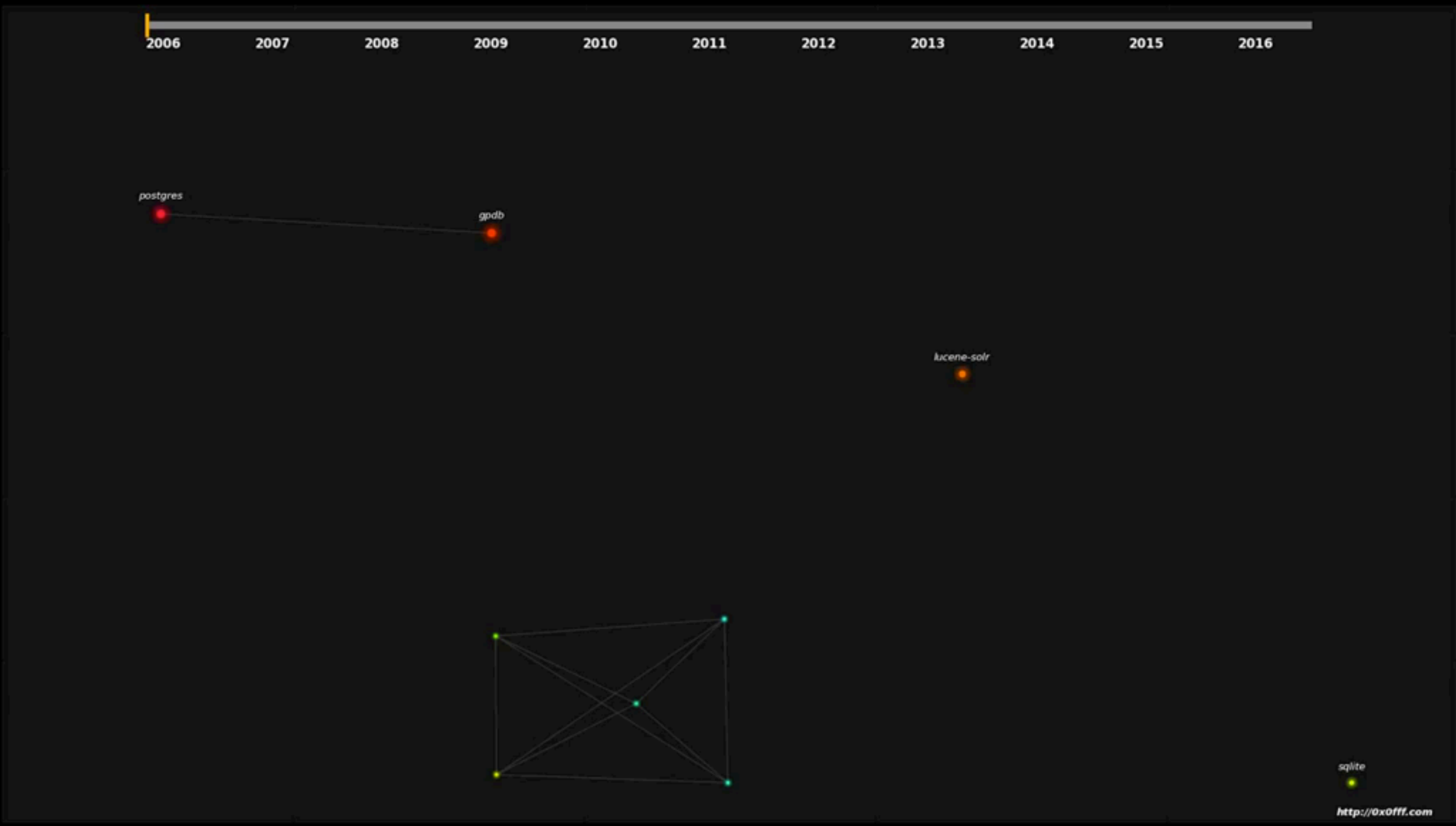
- 비교적 최근에 (2012년) 등장하여 선풍적인 인기를 얻고 있는 분산처리 프레임워크
- 메모리 기반의 처리를 통한 고성능과 Functional Programming 인터페이스를 활용한 편리한 인터페이스가 특징

Apache Spark

단순한 연산인 경우 나쁘지 않지만
반복연산일수록 효율이 매우 떨어짐

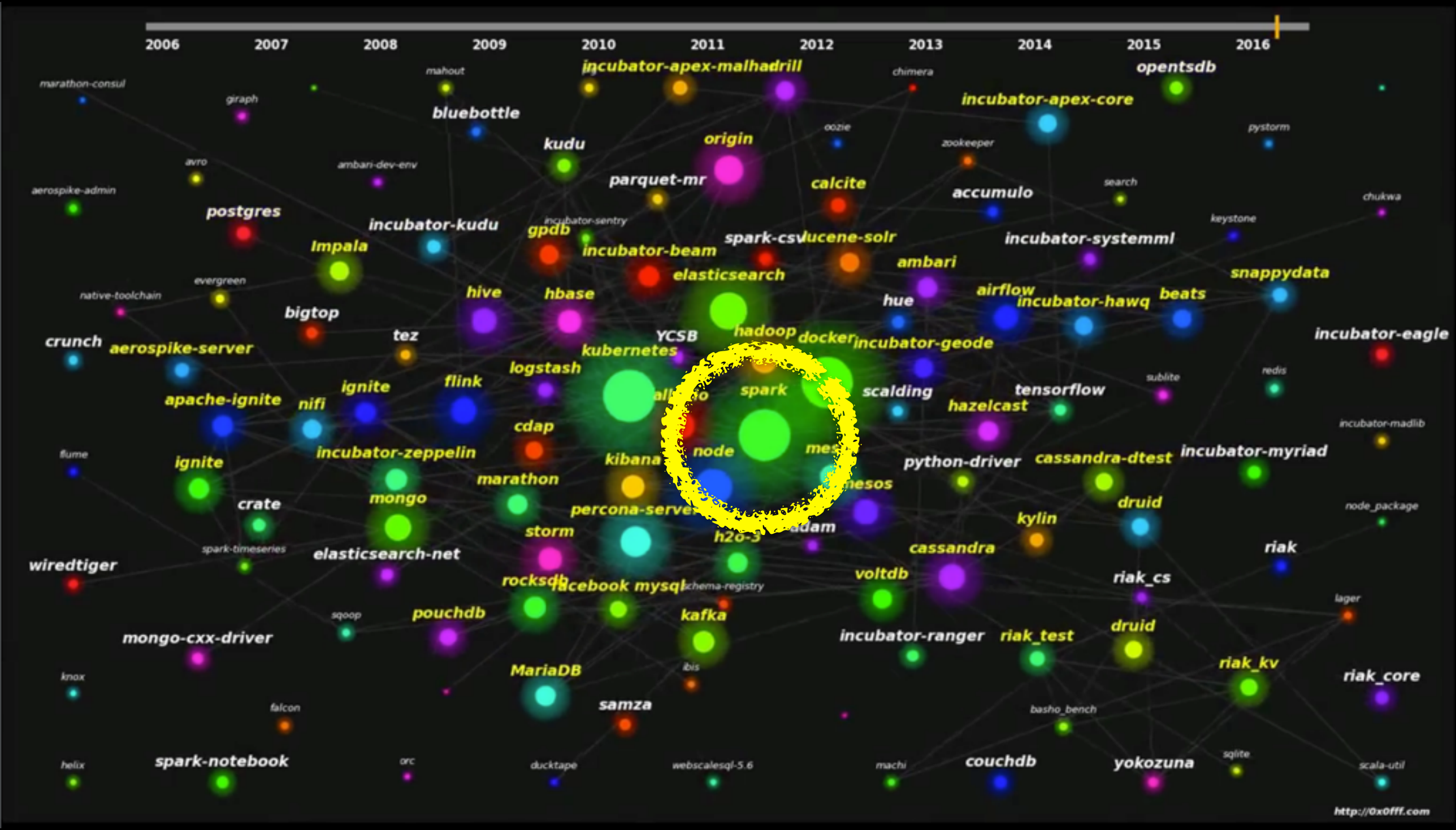
다단계의 연산이나 반복 연산의 경우
중간 결과를 메모리에 저장하면
매번 디스크에 쓰는것 보다 훨씬 빠르다

최근 10년 간 빅데이터 오픈소스 프로젝트들의 세력도 (영상)



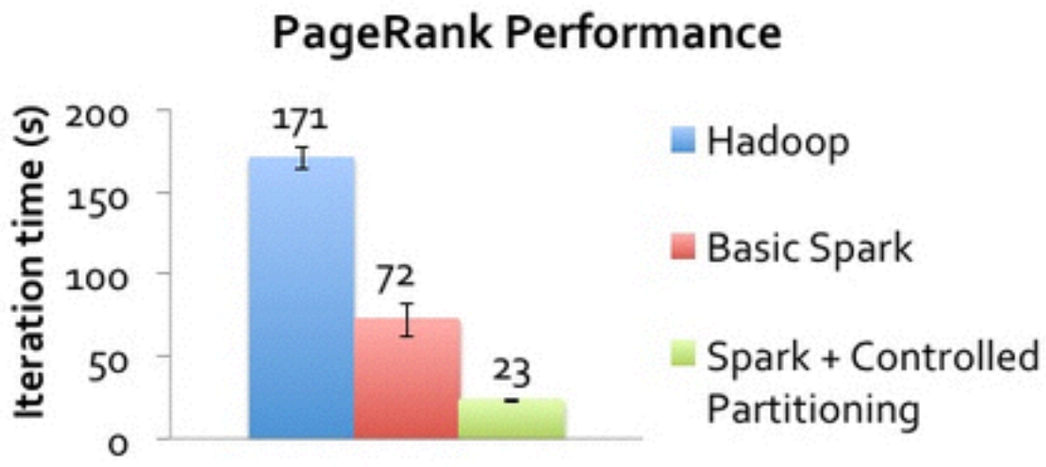
별의 크기: Contributor 수 (*오픈소스 프로젝트를 평가하는 데 객관적인 하나의 지표가 됨)

최근 10년 간 빅데이터 오픈소스 프로젝트들의 세력도 (영상)



별의 크기: Contributor 수 (*오픈소스 프로젝트를 평가하는 데 객관적인 하나의 지표가 됨)

Apache Spark



- Hadoop (MapReduce)는 매번 중간 결과를 디스크에 저장하지만, Spark은 이를 메모리에서 처리하므로 효율이 좋음
- PageRank나 머신러닝 알고리즘같이 반복계산이 많은 경우 특히 성능이 좋음

Apache Spark 핵심개념

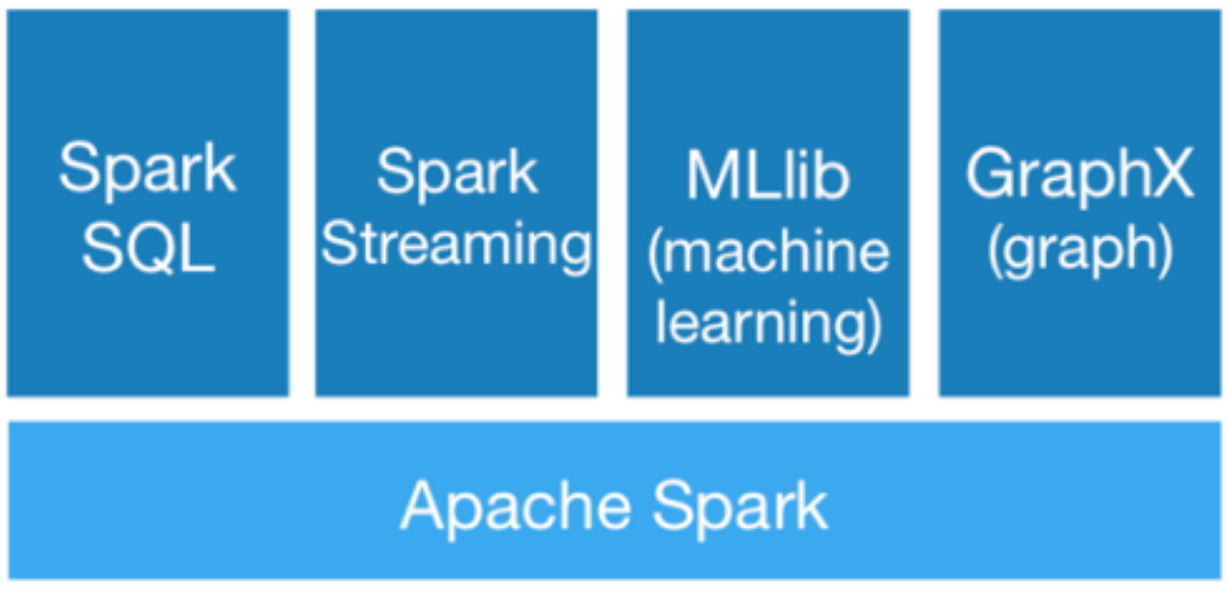
- RDD (Resilient Distributed Dataset - 탄력적으로 분산된 데이터셋)
  - 오류 자동복구 기능이 포함된 가상의 리스트
  - 다양한 계산을 수행 가능, 메모리를 활용하여 높은 성능을 가짐
- Scala Interface
  - 매우 간결한 표현이 가능한 모던 프로그래밍 언어
  - Functional Programming이 가능해 데이터의 변환을 효과적으로 표현할 수 있음

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Word count in Spark

Apache Spark 확장 프로젝트

- Spark을 엔진으로 하는 확장 프로젝트들이 같이 제공됨
  - Spark SQL: Hive와 비슷하게 SQL로 데이터 분석
  - Spark Streaming: 실시간 분석
  - MLlib: 머신러닝 라이브러리
  - GraphX: 페이지랭크같은 그래프 분석



스트리밍 기술들

- Apache Storm
  - 이벤트 기반 실시간 스트리밍 기술

- Spark Streaming
  - 아주 작은 타임윈도우로 배치프로세스를 계속 실행하여 실시간처럼 보이는 기술

Spark Streaming

- DStream: RDD의 연속. 스트림 데이터를 쪼개거나 윈도우를 만들어 짧은 단위로 RDD연산을 수행
- Spark RDD와 사용방법이 거의 유사하여 lambda 아키텍쳐를 만들기 좋음

빅데이터 분석 워크플로우 (Hadoop MR)

서버 로그
센서
SNS등

비정형으로 저장
(파일 그대로)

ETL 및 분석

시각화

데이터 수집

데이터 저장

데이터 준비

데이터 분석

표현

Flume

HDFS

Hive
MapReduce

다양한 BI툴

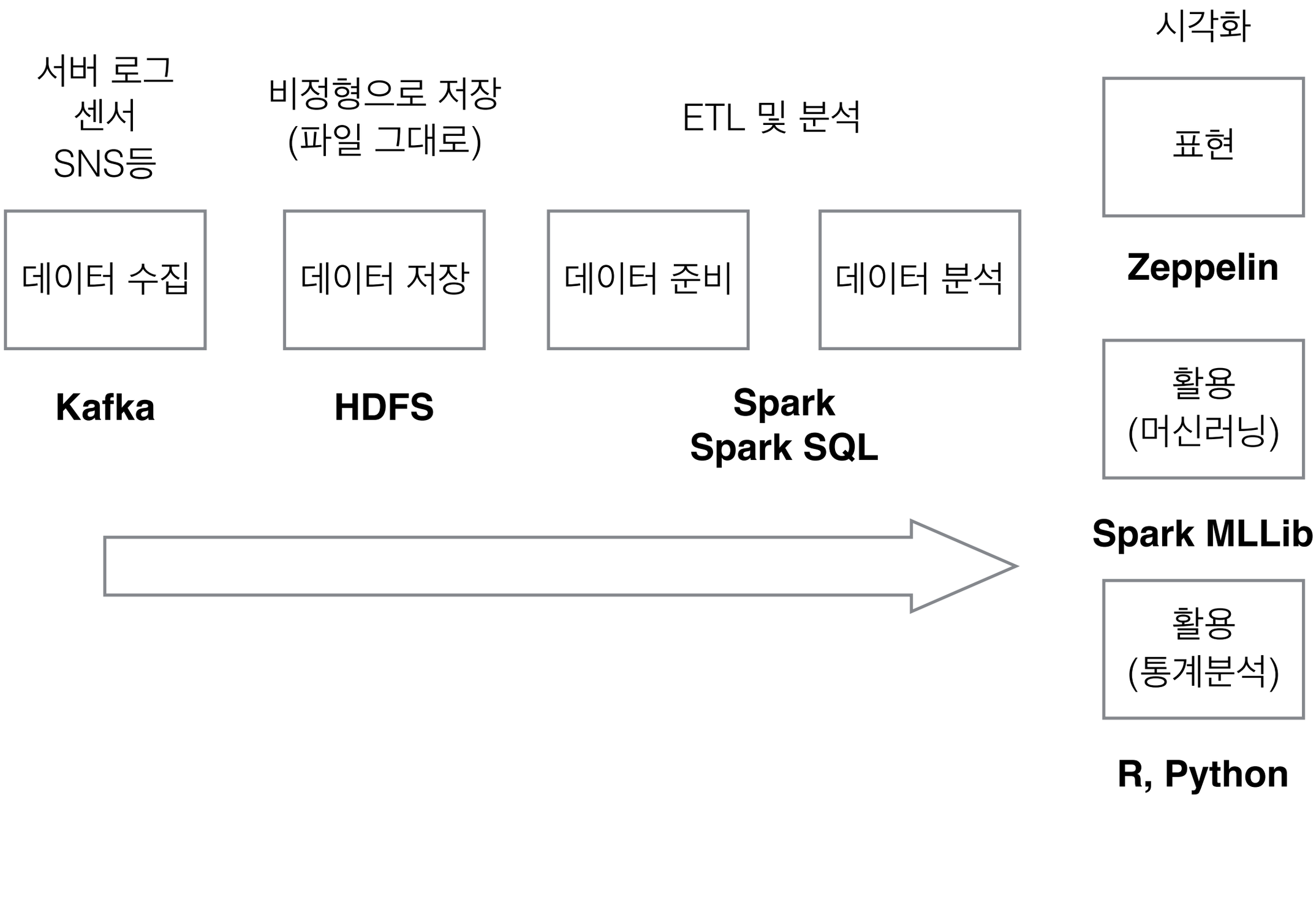
활용
(머신러닝)

Mahout

활용
(통계분석)

SAS, SPSS

빅데이터 분석 워크플로우 (Spark)



향후 방향성

- Apache Spark이 계속 힘을 얻을것
  - 강력한 성능과 좋은 인터페이스, 확장성
  - 수많은 사용자와 개발자
- 오픈소스 커뮤니티는 사람들이 원하는대로 발전해 나아감 (민주적)
  - 최근 머신러닝에 대한 큰 관심
  - 오픈소스 기반 데이터 제품들도 머신러닝 관련 지원이 대폭 강화되는중
  - 예: 알고리즘이 대폭 보강된 Spark ML, Tensorflow on Spark 프로젝트 등

정리

- 빅데이터 분석의 시초는 GFS (2003), MapReduce (2004) 논문에서 시작
- Hadoop (2006) 이 탄생하며 계속해서 발전
- HBase 등의 분산 데이터베이스들이 널리 사용중
- 중간에 다양한 프로젝트들이 생겼으며 현재는 Spark으로 수렴 중

Apache Spark

들어가기전에,

Scala부터 배워보자

Spark 핵심개념

- RDD (Resilient Distributed Datasets)
  - 클러스터 전체에서 공유되는 리스트, 메모리상에 올라가있음. (메모리 부족한 경우, 디스크에 spill)
  - map, reduce, count, filter, join 등 다양한 작업 가능
  - 여러 작업을 설정해두고, 결과를 얻을 때 lazy하게 계산

- Scala
  - 데이터 분석 하기에 아주 좋은 언어
  - 강력한 expression, Java와의 호환성
  - Interactive Shell (REPL)

데이터 분석을 위한 Scala

Scala는 특정 분야에 국한되지 않은 범용 프로그래밍 언어입니다. 본 자료에서는 데이터 분석 분야에 초점을 맞추어 Scala가 생소한 사람들을 위해 Scala의 일부를 소개하고 있습니다.

```java
public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```

Word count in MapReduce (Java)

```
public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce
IOException {
      int sum = 0;
      while (values.ha
        sum += values
      }
      output.collect(
    }
  }

  public static void r
    JobConf conf = new
    conf.setJobName("w

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Word count in Spark(Scala)

Index

- Scala 개요
- 왜 Scala인가?
- Scala 기초 맛보기
- 좀만 더 파보기

Scala 개요

Scalable **La**nguage!

- 간결한 표현과 강력한 기능을 통해 더 큰 프로그램을 만들기 위한 언어
- Scala가 가진 여러가지 특징들이 데이터 분석하기에 좋은 것들이 많다

Scala

- 아주 간결한 문법 (like, Python)
- OOP, Functional Programming 스타일 가능
- JVM에서 실행, Java와 호환
- 좋은 성능 (== Java)
- 정적 타입 (!= Python, == Java)
- REPL (Shell), Scripting

* 그 밖에도 좋은 특징이 많지만, 데이터 분석 분야와 관련된 특징 위주로 언급하였습니다

간결한 문법 (Java와 비교)

Job.java

```java
public class Job {
    public void main(String[] args) {

        Person kevin = new Person();
        kevin.setName("Kevin");
        kevin.setWork("Between");
    }
}
```

Person.java

```java
public class Person {

    private String name;
    private String work;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setWork(String work) {
        this.work = work;
    }

    public String getWork() {
        return work;
    }
}
```

칸이 모자라..

---

job.scala

```scala
class Person(val name: String, val work: String)

val kevin = new Person("Kevin", "Between")
```

GOOD

OOP & Functional Programming

- 오해: OOP와 Functional Programming은 반대말이다? (X)
- Scala는 Pure OOP

```
class Person(val name: String, val work: String)

val kevin = new Person("Kevin", "Between")
```

- Scala는 Functional Programming이 가능

```
val list = List(1, 2, 3)

def aMultiplyFunction(x: Int) = {
  x * 2
}

val result = list.map(aMultiplyFunction)
```

함수가 1st-class citizen!
함수를 데이터로 간주하고,
인자로 넘기는 등의 행위가 가능

JVM에서 실행, Java와 호환

- Scala 코드를 컴파일하면 Java와 마찬가지로 .class 파일이 나옴
- JVM에서 실행, Java와 거의 동일한 실행 성능을 가짐
- Java Class Import하여 사용 가능
- Java file과 Scala file을 혼용하여 컴파일도 가능

정적 타입 언어

- 정적 타입 vs 동적 타입?
  - 서로 장단점이 뚜렷함
  - 정적 타입 언어의 장점: 컴파일시 타입 체킹, 좋은 성능
  - 동적 타입 언어의 장점: 간편한 코드작성, 깔끔한 코드
- Scala는 정적 타입 언어
  - 컴파일시 타입체크, type safety, 좋은 성능
  - 비교적 깔끔한 type interface - 타입을 추론(type inference)하여 넣어 줌
  - 코드를 단순하게 유지하기 위한 implicit conversion등의 장치

왜 Scala인가?

왜 Scala인가?

- 간결한 문법과 강력한 expression
- Functional Programming
- Java와 호환 (= Hadoop 호환!)
- REPL, Scripting
- Apache Spark
- Collection library, Pattern matching, 그 외 멋진 도구들

간결한 문법, 강력한 표현력

- (당연하게도) 문법이 간결하면 좋다.

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
  counts.saveAsTextFile("hdfs://...")
```

- if-else분기 혹은 try-catch 등이 모두 expression임

```
// if statement is an expression!
println(if (a == "A") "It's A!" else "It's not A")

// try catch is an expression!
val value = try {
  doSomeDangerousOperation
} catch {
  case _ => "some value"
}
```

간결한 문법, 강력한 표현력

- 일관성 있는 operator들

```
// Java
"A".equals("B")

// Scala
"A" == "B"
```

- 합리적인 class equality

```
case class Person(name: String, work: String)

val kevin = Person("Kevin", "Between")
val anotherKevin = Person("Kevin", "Between")

kevin == anotherKevin // true
```

case class의 생성에는
new 가 필요 없다

Functional Programming

- 기존의 프로그램에서의 함수가 아닌, 수학적인 의미에서의 함수를 생각해 봅시다!
- y = sin(x) : Side effect가 없음. 어떤 상황에서도 x 를 넣으면 그에 맞는 y가 나옴
- tan(x) = sin(x) / cos(x) : 함수를 데이터처럼 생각하여, 파라메터로 넘기거나 조합하는 등의 작업이 가능
- y = sin(x) : y는 x가 한번 정해지면 변하지 않음. ’변수’ 가 없게!
  - 변수들을 immutable하게 만들자!

* 예제 및 일부 설명을 Programming Scala 책에서 차용했습니다.

FP의 이러한 특성들이 왜 좋은가?

- 버그를 줄여준다 (변수에 의해 예기치못한 동작에 빠지는것을)
- 한번 만들어놓은 함수를 믿을 수 있다 (no side effect!)
- immutable 변수는 문제를 단순화해준다 (data share, parallelism 에 강함)

Java와의 호환성

- JVM에서 구동 -> 많은 양의 데이터 처리할 때 성능 좋음!
- Java library들을 그대로 활용 가능
  - Hadoop eco-system의 Java 코드들을 그대로 사용할 수 있다!
- 예전에 존재하던 코드를 적은 노력으로 convert해서 사용 가능
- Java 코드와 혼용해서 컴파일 가능
  - src/java/…, src/scala/…

REPL

- **R**ead–**E**val–**P**rint **L**oop (aka Shell)
- 새로운 언어를 빠르게 배우고, 시험할 수 있다!
- 데이터를 들여다 볼 경우, step-by-step으로 작업이 가능해서 좋다

```
scala> val learnScala = "learnScala
<console>:1: error: unclosed string literal
       val learnScala = "learnScala
                        ^
```

에러가 나도 즉각 알수 있다

```
scala> logs.head
res1: String = 2014-12-03 10:00:00,SIGN_UP,001
```

데이터를 다루는 과정이 interactive해짐!

Apache Spark

- 메모리 기반 고성능 분산 데이터 처리 시스템 (기존의 10~100배)
- Scala로 쓰여짐. Scala의 collection library와 유사한 인터페이스
- Scala shell에 기능을 추가한 Spark shell 제공
- 범용적으로 사용하기 위한 다양한 연관 프로젝트 존재
  - SQL, Machine Learning, Graph Analysis.. 등등
- 지금도 빠르게 개발되고 있고 많은 사람들의 관심을 받고 있음

그 밖에..

- Collection library
- Pattern matching
- implicit같은 우아한 도구들
- 뒷부분에서 더 자세히 다룰 예정

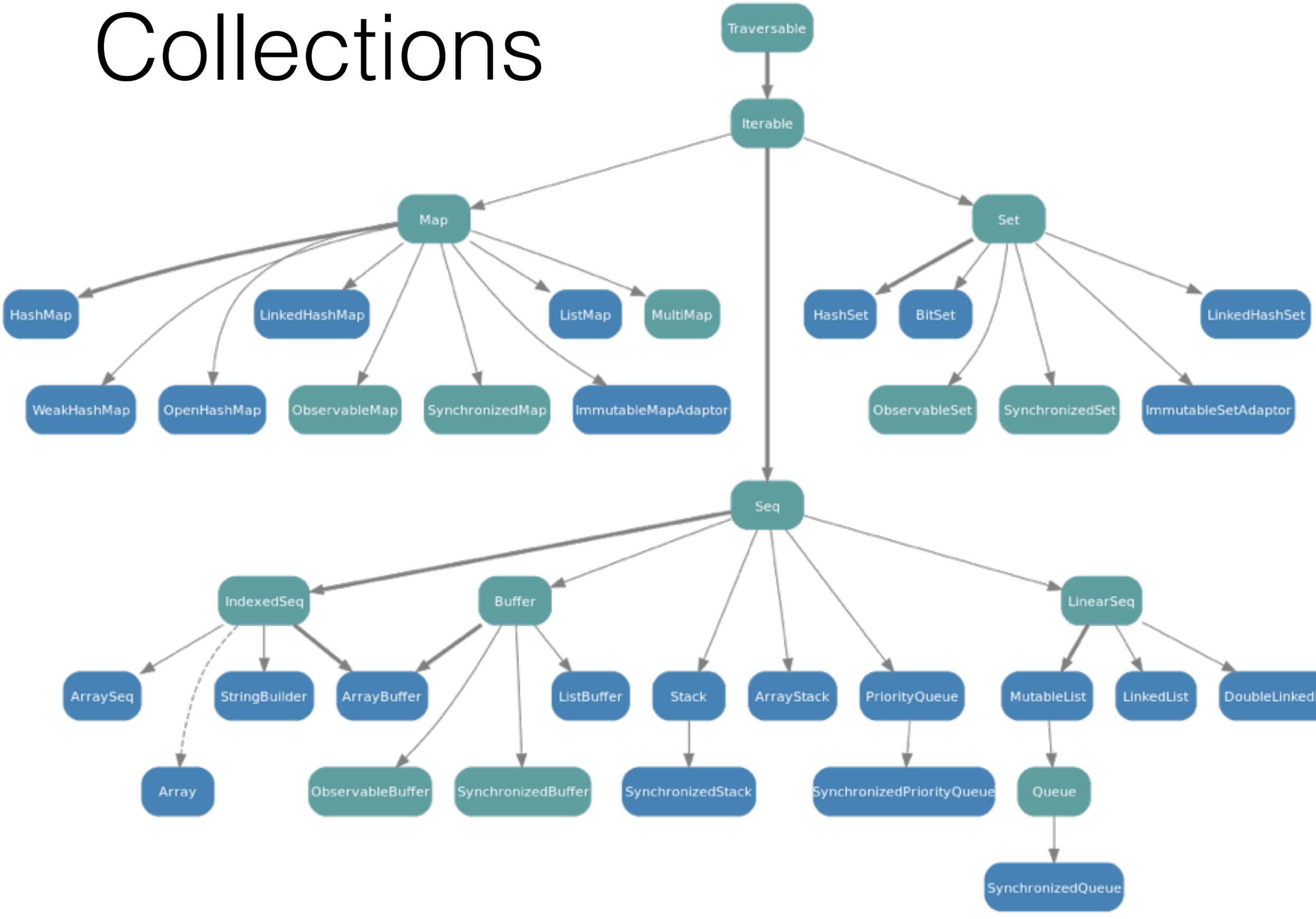
Scala 기초 맛보기

데이터와 관련된 부분만

데이터 구조

- List, Map, Set 등의 collection 들
  - List(1, 2, 3), Map(1 -> “a”, 2 -> “b”), Set(1, 2)
- Tuple
  - val sparkTechTalk = (“2017-07-29”, 50)
  - sparkTechTalk._1
  - case (key, value) => println(key)
- Option
  - 값이 없을 때, null 대신! (더 편하고, 안전한 프로그래밍)
  - a = 1, a = null (기존) a = Some(1) a = None (Option활용)
  - a.nonEmpty, a.getOrElse(0)
- Range
  - for (i <- 0 to 10) println(i)
  - (0 to 10).foreach(println)
  - (0 until 10) (0 to 10) (0 to -10 by -1)

Collections



Collection 다루기

- (n), head, tail, last, contains, distinct, drop, …
- Functional Combinators
  - map: element에 함수를 적용하여 다른 형태로 변환
  - filter: element를 true/false 판별 함수 적용 후 true인 항목만 남김
  - foreach: map과 비슷, 다른형태로 변환하지 않고 iteration만 수행
  - foldLeft (foldRight, reduce): 왼쪽의 element부터 시작하여 하나로 합침
- 뒷부분에 사용 예를 봅시다

Function Literal

수열에서 3 미만인 값 구하기
  많은 부분을 축약 가능

```
val list = List(1, 2, 3, 4)

list.filter((x: Int) => x < 3)

val testNumber1 = (x: Int) => x < 3 // function as a 1st-class object!
list.filter(testNumber1)

list.filter((x) => x < 3) // target typing
list.filter(x => x < 3)
list.filter(_ < 3) // placeholder


def testNumber2(x: Int) = x < 3 // function

list.filter(x => testNumber2(x))
list.filter(testNumber2(_))
list.filter(testNumber2 _)
list.filter(testNumber2)
```

많은 부분을 축약 가능!

Pattern Matching & Case Class

- Java의 switch ~ case 와 비슷하지만, 훨씬 강력한 도구

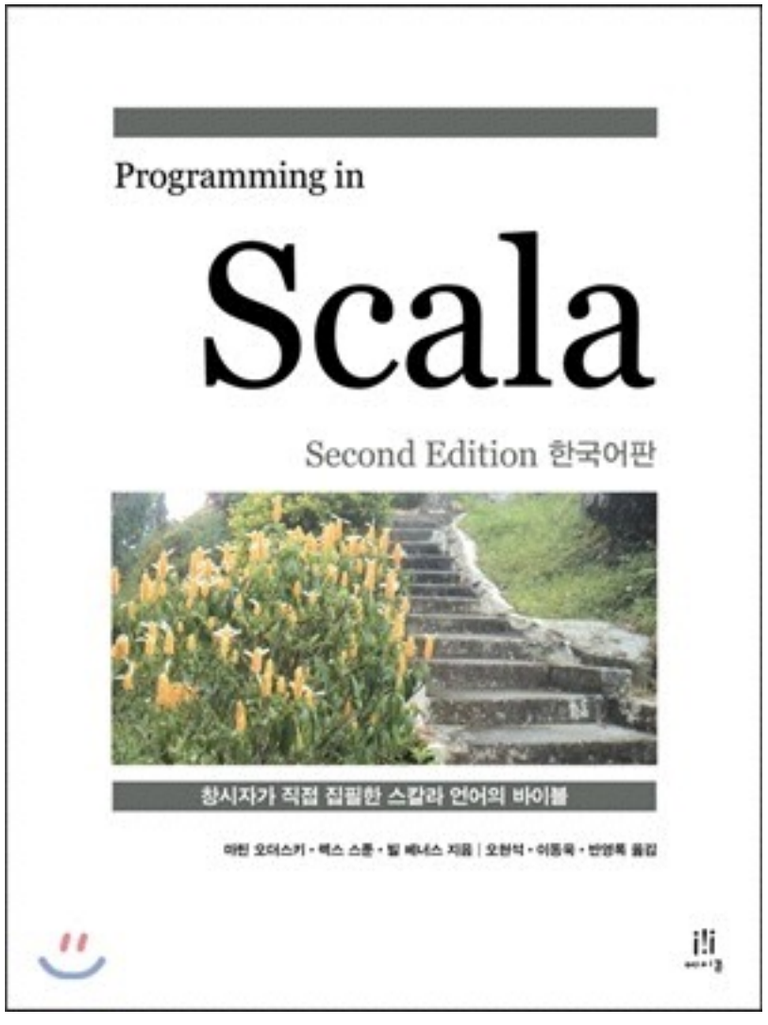
```
val input1 = "three"

case class Chart(date: String, count: Int)
val input2 = Chart("2017-07-29", 50)

val input3 = ("spark-techtalk", 100)

def matchTest(x: Any): Any = {
  x match {
    case 1 => "one"
    case "two" => 2
    case (key, value) => s"key: $key, value: $value"
    case Chart(date, count) => s"date: $date, count: $count"
    case _ => "others"
  }
}

matchTest(input1)
res0: Any = others

matchTest(input2)
res1: Any = date: 2014-12-02, count: 50

matchTest(input3)
res2: Any = key: spark-techtalk, value: 100
```

case class: 데이터 구조화에 편리

다른 종류의 타입이라도 매치 가능
case 혹은 case class 활용하면 더욱 편리

기초 문법들이나, 더 자세한 이론적 내용은 책을 참고합시다.

추천도서: Programming in Scala (한국어판 있음)



예제: 로그에서 간단한 지표 구하기

```
// load log file
val logFile = new java.io.File(path + "example_log.txt")
val log = scala.io.Source.fromFile(logFile).getLines().toList

// parse log and get sign up numbers
case class LogEntry(dateTime: String, action: String, id: String)
val logEntries = log.map(csv => csv.split(",")).map(arr => LogEntry(arr(0), arr(1),
arr(2))).toList
```

```
// get sign up
val logEntriesToday = logEntries.filter(_.dateTime.contains("2017-07-29"))
val signUp = logEntriesToday.filter(_.action == "SIGN_UP").size

// active user
val userIds = logEntriesToday.map(_ id)
val activeUser = userIds.distinct.size
```

Bonus: Spark Version

```
// load log file
val log = sc.textFile("file:///example_log.txt")

// parse log and get sign up numbers
case class LogEntry(dateTime: String, action: String, id: String)
val logEntries = log.map(csv => csv.split(",")).map(arr => LogEntry(arr(0), arr(1),
arr(2)))

// get sign up
val logEntriesToday = logEntries.filter(_.dateTime.contains("2017-07-29"))
val signUp = logEntriesToday.filter(_.action == "SIGN_UP").count

// active user
val userIds = logEntriesToday.map(_ id)
val activeUser = userIds.distinct.count
```

Scala collection API와 거의 완전히 동일!

좀만 더 파보기

Implicits

Implicit Conversion

- 기능의 확장을 편하게 하고싶을때
- 예상되는 타입으로 변환하는 함수를 정의해놓고, 자동으로 적용

```
implicit def stringToInt(number: String): Int = {
  number match {
    case "one" => 1
    case "two" => 2
  }
}

def printNumber(n: Int) = println(n)

printNumber("one")
```

원래대로라면, compile error.
implicit conversion이 선언되어 있으므로,
String => Int 로 자동 변환이 일어남

Implicit Conversion 활용

```
DateParser.parse("2017-07-29") // java style

"2017-07-29".toDateTime // better solution using implicit conversion
```

```
object DateParser {
  def parse(dateString :String) = new java.util.Date
}

DateParser.parse("2017-07-29")
```

```
class DateConverter(val s: String) {
  def toDateTime = DateParser.parse(s)
}
implicit def string2DateConverter(s: String) = new DateConverter(s)

"2017-07-29".toDateTime
```

더 직관적이고 일관성 있는 코드를 만들 수 있다!

Implicit Parameter

- 반복 적용되는 파라메터를 간단하게 만들고 싶을때

```
val date = "2017-07-29"

calculateSignUp(date)
calculateActiveUser(date)
calculateActionCount(date)
```

```
def calculateSignUp(implicit date: String) = ...

implicit val date = "2017-07-29"

calculateSignUp
calculateActiveUser
calculateActionCount(date)
```

- 단, implicit을 남발하면 힘들어진다!

정리

- Scala는 데이터 분석하기에 좋은 언어 (다른 용도로도 좋아요)
- 간결한 표현, 좋은 성능, Functional Programming
- REPL, Scripting가능
- 우아한 방식으로 원하는 개념을 구현할 수 있음

참고할만한 자료

- Scala 5분만에 배우기

  http://learnxinyminutes.com/docs/scala/
- Coursera Scala 강의

  https://www.coursera.org/course/progfun
- Scala 배우기 (블로그)

  http://joelabrahamsson.com/learning-scala/
- Scala School (트위터)

  http://twitter.github.io/scala_school/ko/
- Programming in Scala (한국어판)

  Scala의 창시자인 마틴 오더스키가 직접 저술, 전국 서점에서 구매 가능

Spark
동작원리 & 실습

Spark 핵심개념

- RDD (Resilient Distributed Datasets)
  - 클러스터 전체에서 공유되는 리스트, 메모리상에 올라가있음. (메모리 부족한 경우, 디스크에 spill)
  - map, reduce, count, filter, join 등 다양한 작업 가능
  - 여러 작업을 설정해두고, 결과를 얻을 때 lazy하게 계산

- Scala
  - 데이터 분석 하기에 아주 좋은 언어
  - 강력한 expression, Java와의 호환성
  - Interactive Shell (REPL)

RDD

- Transformations
  - 데이터를 어떻게 구해낼지를 표현
- Actions
  - 표현된 데이터를 가져옴
- Lineage
  - 클러스터 중 일부의 고장 등으로 작업이 중간에 실패하더라도, Lineage를 통해 데이터를 복구
- Lazy Execution
  - Transformation시에는 계산을 수행하지 않고, Action이 수행되는 시점부터 데이터를 읽어들여서 계산을 시작

RDD Transformations

- RDD의 데이터를 다른 형태로 변환
  - 실제로 데이터가 변환되는 것이 아니라, 데이터를 읽어들여서 어떻게 바꾸는지 방식만을 기록
  - 실제 변환은 Action이 수행되는 시점에서 이루어짐
- map, filter, flatMap, mapPartitions, sample, union, intersection, distinct, groupByKey, reduceByKey, join, repartition 등
- 최신의 자료는 Spark Programming Guide 참고
  - https://spark.apache.org/docs/latest/rdd-programming-guide.html

RDD Actions

- 여러가지 변환 (Transformation)이 담긴 RDD의 정보를 통한 계산을 수행함
- reduce, collect, count, first, take, saveAsTextFile, countByKey, foreach 등
- 최신의 자료는 Spark Programming Guide 참고
  - https://spark.apache.org/docs/latest/rdd-programming-guide.html

Word Count Example

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

- sc.textFile() - 파일을 로드
- flatMap(line => line.split(“ “)) - 로드된 텍스트 라인을 space 로 나누고, 2단 array 형식의 데이터를 1단으로 flatten 함
- map(word => (word,1)) - 단어단위로 나눠진 것을, 1 이라는 레이블을 붙임
- reduceByKey(_+_) - 데이터를 같은 key (word) 끼리 묶고, 레이블을 서로 더함 (_ : placeholder, _+_ : 그것과 저것을 더하라)

Data Loading

- Spark의 데이터 입력 부분은 Hadoop의 코드를 그대로 사용하기 때문에, Hadoop에서 지원하는 모든 소스를 사용할 수 있다
  - 로컬 파일 : sc.textFile(“file:///…)
  - HDFS : sc.textFile(“hdfs://…)
  - Amazon S3 : sc.textFile(“s3://…)
  - HBase, Cassandra : Spark HBase Connector 등 이용
  - 등등
- 여러 기능 포함
  - 압축파일도 읽어들이기 가능
  - 와일드카드 (*) 사용 가능

Transformation - Map, Filter

- map, filter와 같은 Transformation은 즉시 계산이 수행되는것이 아니라, count() 와 같은 Action이 수행될 때 실제 계산이 수행됨
- transformation이 기록된 새 RDD를 리턴해줌
- map(*func*) : *func* 로 기술되는 동작을 RDD에 모든 element에 수행.
- filter(*func*): true/false 를 판별하는 *func* 이 true인 element만 남겨둠

Transformation - Reduce, GroupBy

- reduce(*func*), 기술한 *func* 대로 RDD의 element를 합치는 작업을 수행함
- map() 은 각 클러스터 간 데이터 교환 없이 element-wise 데이터 변환만을 수행하므로 아주 효율적으로 병렬 처리가 가능
- reduce()도 최종적으로 클러스터간의 데이터가 모이기 전에 클러스터 내부의 데이터부터 reduce 계산이 가능하기에 효율적인 operation임
- groupBy(func) : reduce와 비슷하지만 데이터를 줄이는것이 아니라, 전부 보존해서 수집해야 함
  - 대량의 네트워크 트래픽이 발생
  - 메모리 문제가 발생할 가능성

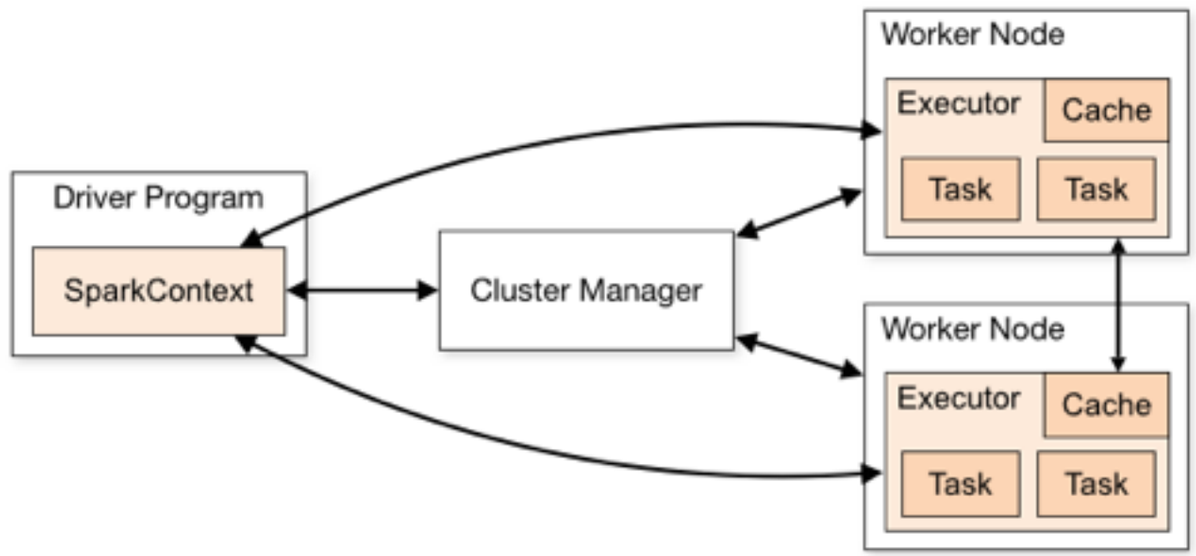
Action - Count, Collect, Take 등

- Spark은 Action 이 수행되면 그때서야 파일을 로드하고, 기록된 Transformation을 수행하고, 최종 Action을 수행한다.
- count() : RDD의 element 갯수를 세는 동작
- collect() : RDD의 내용을 전부 드라이버 프로그램으로 가져옴
  - RDD의 내용이 큰 경우, collect 할때 메모리가 꽉차서 프로그램이 죽을 수 있음
  - RDD의 내용이 충분히 작지 않으면, 안전한 take를 사용
- take(n) : 처음 n 개의 element를 가져옴

RDD 캐싱

- Spark은 메모리 캐시를 활용하여 성능을 극대화할 수 있다
  - MapReduce 와의 큰 차이점
  - persist() or cache() 를 이용
  - 데이터가 커서 메모리에 올라가지 않는 경우는, 캐시하지 못한 데이터는 다시 계산한다
  - 메모리 부족 시 디스크 저장 등 다양한 옵션을 선택할 수 있음

Spark Cluster Mode



- Hadoop의 전형적인 Master-Slave 구조
- Cluster Manager
- Driver (Master)
  - SparkContext (App)이 구동됨
- Worker (Slave)
  - Executor 가 구동됨, 여러개의 Task들이 수행

실습

- Spark을 다운로드, 설치
- Spark Shell을 구동하여
- 여러가지 Spark Operation 들을 시험해보고 익힘
- Word Count 실습
- Zeppelin 연동 및 Spark SQL을 이용한 데이터 분석 및 시각화 실습

Cluster Mode 실습

- conf/slaves 세팅
- sbin/start-all.sh
- Master monitoring
  - http://192.168.0.14:8080/
- Worker monitoring
  - http://192.168.0.14:8081/
- Application monitoring
  - http://192.168.0.14:4040/
- bin/spark-shell --master spark://ubuntu:7077

정리

정리

- 빅데이터의 역사와 발전에 대해 알아봄
- 최근 가장 각광받는 Apache Spark에 핵심 개념에 대해 알아봄
- Spark을 이해하는데 필수적인 Scala 언어를 학습
- Spark Core의 동작 원리를 익히고 실습해봄