

Apache Spark

아파치 스파크(Apache Spark)는 오픈 소스 클러스터 컴퓨팅 프레임워크이다. 캘리포니아 대학교 버클리의 AMPLab에서 개발된 스파크의 코드베이스는 나중에 아파치 소프트웨어 재단에 기부되었으며 그 이후로도 계속 유지보수를 해오고 있다. 기존엔 batch processing을 하기 위해 MapReduce를 사용하고, sql을 사용하기 위해서는 hive를 사용하는 등 다양한 플랫폼을 도입해야 했었다. 이제는 Spark 하나만을 설치해도 batch, streaming, graph processing, sql 등의 처리가 가능하다. 또한 Spark은 Java, Scala, Python, SQL, 그리고 R언어의 API를 제공하기 때문에 언어적인 선택의 폭이 넓다.



[Spark의 특징]

- High-Level Tools
- Spark SQL for SQL
Hadoop의 Hive가 아닌 Spark SQL을 통해 SQL을 MapReduce없이 빠르게 처리가 가능하다.
- 구조적 데이터 프로세싱
Json, Parquet(파켓 : Apache Parquet은 Apache Hadoop 에코 시스템의 무료 오픈 소스 열 기반 데이터 저장 형식), DataFrame, DataSet
- MLlib for machine learning
Classification, Regression, Abnormal Detection, Clustering 등의 다양한 machine learning algorithm을 제공한다.
- GraphX for graph processing
graph processing을 지원하는 GraphX를 제공한다.
- Spark Streaming.
batch processing 외에도 streaming처리가 가능하다.
- Launching on a Cluster
Sparks를 클러스터에서 동작하게 하기 위해서는 cluster manager가 필요하다.

[Cluster Manager 종류]

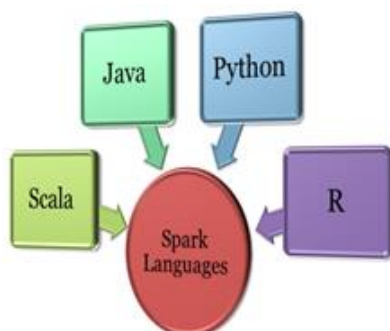
Standalone Deploy Mode

Apache Mesos

Hadoop Yarn

Kubernetes

Cloud



spark.apache.org/downloads.html



Download Libraries Documentation Examples Community Developers

Download Apache Spark™

1. Choose a Spark release: 2.4.5 (Feb 05 2020)
2. Choose a package type: Pre-built for Apache Hadoop 2.7
3. Download Spark: spark-3.0.0-preview2-bin-hadoop2.7.tgz
4. Verify this release using the 3.0.0-preview2 signatures, checksums and project release KEYS.

Note that, Spark is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12.

Latest Preview Release

Preview releases, as the name suggests, are releases for previewing upcoming features. Unlike nightly packages, preview releases have been audited by the project's management committee to satisfy the legal requirements of Apache Software Foundation's release policy. Preview releases are not meant to be functional, i.e. they can and highly likely will contain critical bugs or documentation errors. The latest preview release is Spark 3.0.0-preview2, published on Dec 23, 2019. You can select and download it above.

(1) 스파크 사이트에서 spark-2.4.5-bin-hadoop2.7.tgz 을 다운로드하고 C:\W 에 압축 해제

(2) 다음 환경 변수들을 시스템 변수에 추가한다.

```
PATH %SPARK_HOME%\bin;%SPARK_HOME%\sbin
```

```
scala> counts.collect()
res: Array[(String, Int)] = Array((대통령으로서의,1), (차별없는,1), (심심한,1), (반칙이,1), (주요,1), (안팎으로,1), (전황위부의,1), (물었습니다,1), (정치,1), (대한민국,3), (지역과,1), (올바른,1), (정례화하고,1), (토대를,1), (약속을,2), (소외된,1), (마침내,1), (나누겠습니다,2), (여러분,4), (제가,2), (질문에서,1), (선거,1), (시대,1), (열어가는,1), (한미동맹을,1), (동반사추하셨습니다,1), (존경하는,3), (서러운,1), (통합과,1), (일을,3), (국민은,2), (도쿄에도,1), (중식돼야,1), (않고,1), (충반자입니다,2), (막강한,1), (
```

```
counts.saveAsTextFile("hdfs://192.168.111.120:9000/result/sparkresult1")
```

```
[root@master hadoop]# hdfs dfs -cat /result/sparkresult1/part-00000
(대통령으로서의, 1)
(차별없는, 1)
(심심한, 1)
(반칙이, 1)
(주요, 1)
(인편을, 1)
```

[테스트 코드]

```
val rdd1 = sc.textFile("hdfs://192.168.111.120:9000/edudata/fruits.txt")
rdd1.collect()
val rdd2 = rdd1.flatMap(line => line.split(" "))
rdd2.collect()
val rdd3 = rdd2.map(word => (word, 1))
rdd3.collect()
val rdd4 = rdd3.reduceByKey(_ + _)
rdd4.collect()
```

```
C:\Windows\system32\cmd.exe - spark-shell

scala> val rdd1 = sc.textFile("hdfs://192.168.111.120:9000/edudata/fruits.txt")
rdd1: org.apache.spark.rdd.RDD[String] = hdfs://192.168.111.120:9000/edudata/fruits.txt MapPartitionsRDD[10] at textFile
at <console>:24

scala> rdd1.collect()
res7: Array[String] = Array(apple banana cherry peach, banana cherry peach cherry cherry, apple cherry peach grape, ap
ple grape pear cherry peach, orange banana cherry orange, apple banana pear banana peach banana, orange banana orange pe
ach orange orange, apple banana pear pear pear peach, apple tomato cherry peach, tomato tomato tomato tomato)

scala> val rdd2 = rdd1.flatMap(line => line.split(" "))
rdd2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at flatMap at <console>:25

scala> rdd2.collect()
res8: Array[String] = Array(apple, banana, cherry, peach, banana, cherry, peach, "", cherry, "", cherry, apple, cherry,
peach, grape, apple, grape, pear, cherry, peach, orange, banana, cherry, orange, apple, banana, pear, banana, peach, ban
ana, orange, banana, orange, peach, orange, orange, apple, banana, pear, "", pear, "", pear, peach, apple, tomato, cherr
y, peach, tomato, tomato, tomato, tomato)

scala> val rdd3 = rdd2.map(word => (word, 1))
rdd3: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[12] at map at <console>:25

scala> rdd3.collect()
res9: Array[(String, Int)] = Array((apple,1), (banana,1), (cherry,1), (peach,1), (banana,1), (cherry,1), (peach,1), ("",
1), (cherry,1), ("",1), (cherry,1), (apple,1), (cherry,1), (peach,1), (grape,1), (apple,1), (grape,1), (pear,1), (cherry
,1), (peach,1), (orange,1), (banana,1), (cherry,1), (orange,1), (apple,1), (banana,1), (pear,1), (banana,1), (peach,1),
(banana,1), (orange,1), (banana,1), (orange,1), (peach,1), (orange,1), (orange,1), (apple,1), (banana,1), (pear,1), ("",
1), (pear,1), ("",1), (pear,1), (peach,1), (apple,1), (tomato,1), (cherry,1), (peach,1), (tomato,1), (tomato,1), (tomato
,1), (tomato,1))

scala> val rdd4 = rdd3.reduceByKey(_ + _)
rdd4: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[13] at reduceByKey at <console>:25

scala> rdd4.collect()
res10: Array[(String, Int)] = Array((banana,8), (pear,5), (orange,6), (grape,2), (cherry,8), (tomato,5), ("",4), (peach,
```

[워드 카운트 수행 : Java (프로그램 방식)]

```
// Java 7
```

```
JavaRDD<String> rdd1 = sc.textFile("hdfs://192.168.111.120:9000/edudata/fruits.txt");
```

```
JavaRDD<String> rdd2 = rdd1.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterator<String> call(String v) throws Exception {
        return Arrays.asList(v.split("[\\w\\s]+")).iterator();
    }
})
```

```

});
JavaPairRDD<String, Long> rdd3 = rdd2.mapToPair(new PairFunction<String, String,
Long>() {
    @Override
    public Tuple2<String, Long> call(String s) throws Exception {
        return new Tuple2<String, Long>(s, 1L);
    }
});
JavaPairRDD<String, Long> rdd4 = rdd3.reduceByKey(new Function2<Long, Long, Long>() {
    @Override
    public Long call(Long v1, Long v2) throws Exception {
        return v1 + v2;
    }
});
System.out.println(rdd4.collect());
rdd4.saveAsTextFile("hdfs://192.168.111.120:9000/result/sparkresult2");
sc.stop();

```

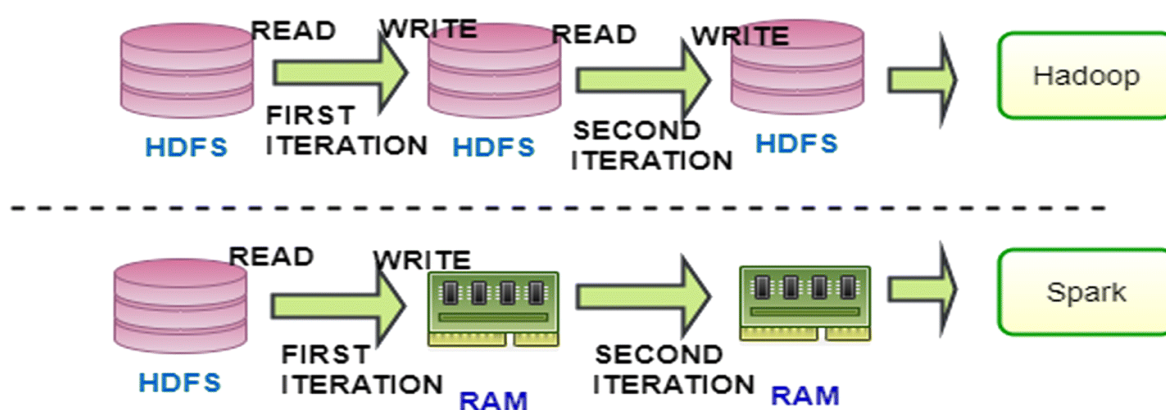
// Java 8 – 람다식 사용

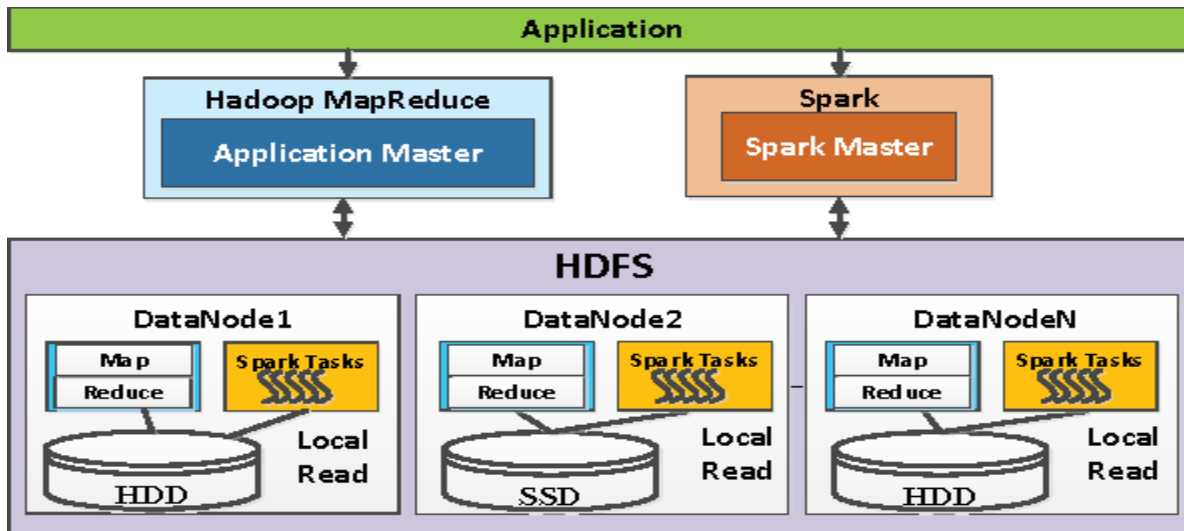
```

JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> rdd1 = sc.textFile("hdfs://192.168.111.120:9000/edudata/fruits.txt");
JavaRDD<String> rdd2 = rdd1.flatMap(line ->
    Arrays.asList(line.split("[\w\s]+")).iterator());
JavaPairRDD<String, Long> rdd3 = rdd2.mapToPair(w -> new Tuple2<String, Long>(w, 1L));
JavaPairRDD<String, Long> rdd4 = rdd3.reduceByKey((x, y) -> x + y);
System.out.println(rdd4.collect());
rdd4.saveAsTextFile("hdfs://192.168.111.120:9000/result/sparkresult3");
sc.close()

```

Apache Hadoop MapReduce와 Apache Spark



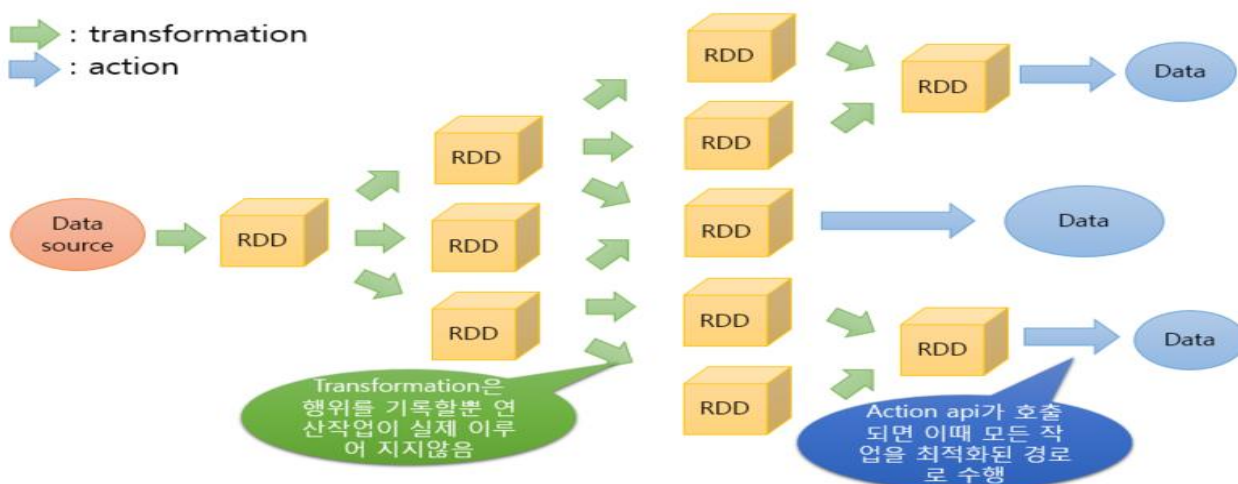


SparkContext : Spark 애플리케이션과 클러스터의 연결을 관리하는 객체
 RDD 등 Spark 에서 사용되는 주요 객체는 SparkContext 객체를 통해서 생성된다.

```
SparkConf conf = new SparkConf().setMaster("local").setAppName("xxx");
JavaSparkContext sc = new JavaSparkContext(conf);
```

RDD(Resilient Distributed DataSet)) :
 Spark 에서 처리되는 데이터는 RDD 객체로 생성하여 처리한다.
 RDD 객체는 두 가지 방법으로 생성 가능하다.
 (1) Collection 객체를 만들어서
 (2) HDFS 의 파일을 읽어서

RDD 객체의 특징 : Read Only(immutable)
 1~n 개의 partition 으로 구성 가능
 병렬적(분산) 처리가 가능하다.
 RDD의 연산은 Transformation 연산과 Action 연산으로 나뉜다.
 Transformation은 Lazy-execution을 지원한다.
 lineage를 통해서 fault tolerant(결함 내성)을 확보한다.



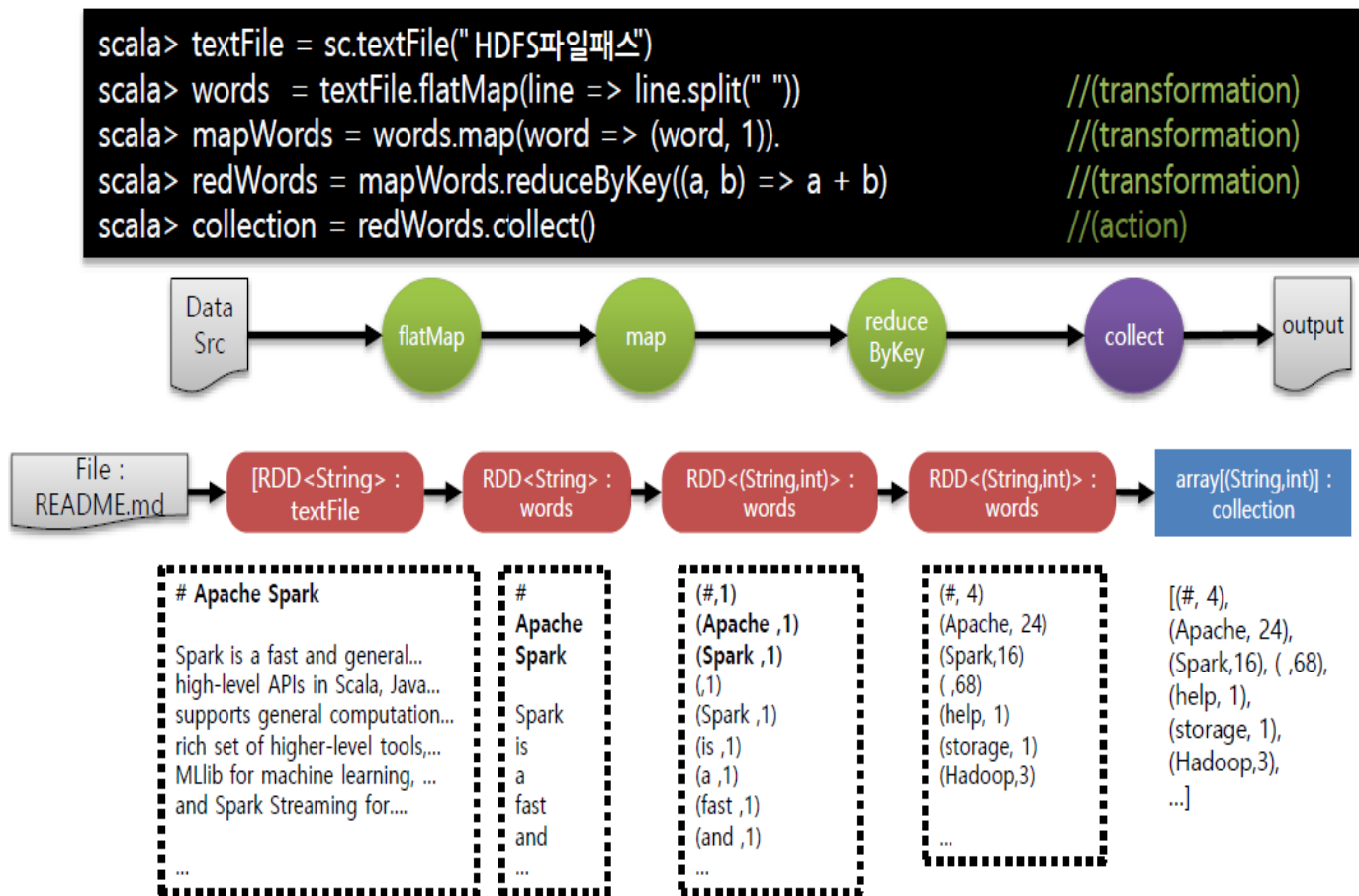
Transformations
map(func) flatMap(func) filter(func) groupByKey() reduceByKey(func) mapValues(func) sample(...) union(other) distinct() sortByKey() ...

Actions
reduce(func) collect() count() first() take(n) saveAsTextFile(path) countByKey() foreach(func) ...

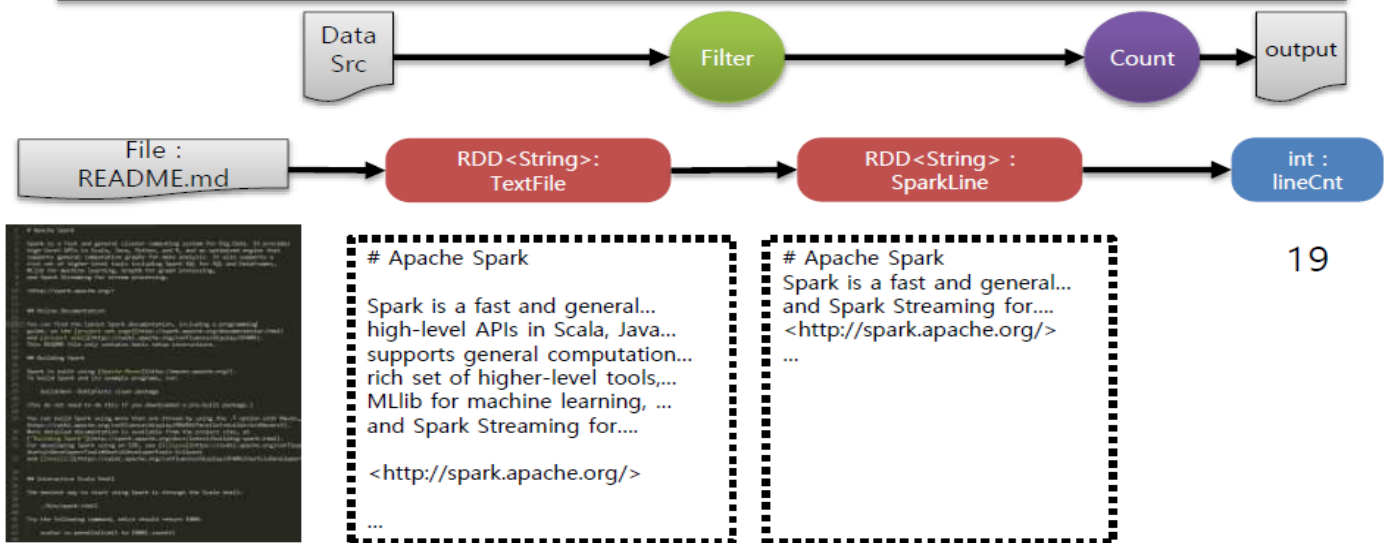
- [Transformation과 Action]
- 연산의 수행 결과가 RDD 이면 Transformation
 - Transformation 은 기존 RDD를 이용해 새로운 RDD를 생성하는 연산이다.
(변환, 필터링 등의 작업들 : 맵, 그룹화, 필터, 정렬...) Lineage를 만들어 가는 과정이다.
 - 연산의 수행 결과가 정수, 리스트, 맵 등 RDD 가 아닌 다른 타입이면 Action
(first(), count(), collect(), reduce()...) Lineage를 실행하고 결과를 만든다.

Lazy-execution 이란 Action 연산이 실행되기 전에는 Tranformation 연산이 처리되지 않는 것을 의미한다. Transformation 연산은 관련 메서드를 호출하여 연산을 요청해도 실제 수행은 되지 않고 연산 정보만 보관한다. 이렇게 Transformation 연산 정보를 보관한 것을 Lineage(리니지)라고 한다. 보관만 하다가 첫 번째 Action 연산이 수행될 때 모든 Lineage 에 보관된 Transformation 연산을 한 번에 처리한다.

Lazy-execution과 Lineage를 활용함으로써 처리 효율을 높이고 클러스터 중 일부 고장으로 작업이 실패해도 Lineage를 통해 데이터를 복구한다.



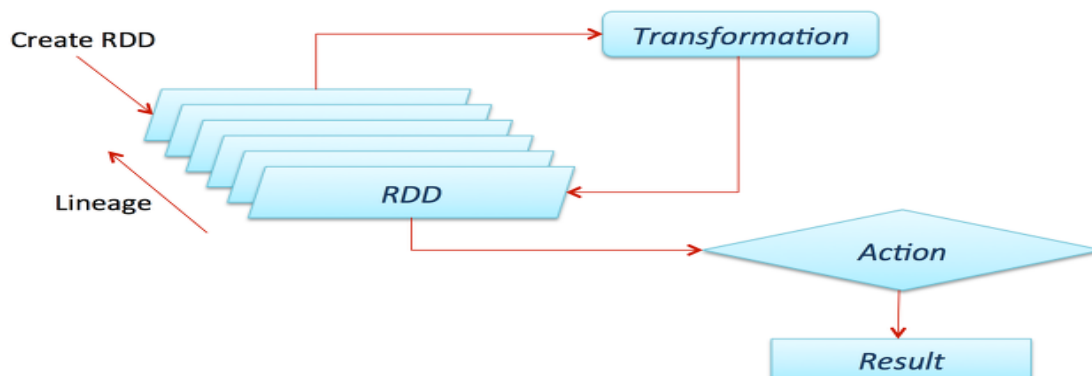
```
scala> textFile = sc.textFile(" HDFS파일패스")
scala> SparkLine = textFile.filter(line => line.contains("Spark")) //(transformation)
scala> lineCnt = SparkLine.count() //(action)
```



19

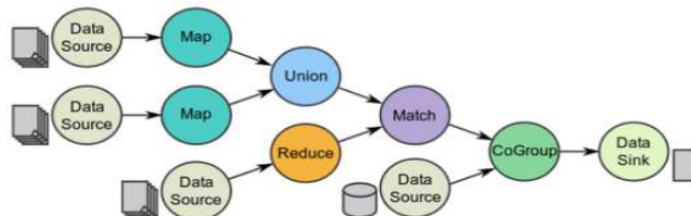
RDD를 제어하는 API operation은 크게 2개의 타입

- **Transformation** : RDD에서 데이터를 조작하여 새로운 RDD를 생성하는 함수
- **Action** : RDD에서 RDD가 아닌 타입의 data로 변환하는 하는 함수들



operation의 순서를 기록해 DAG로 표현한 것을 **Lineage**라 부름

Lineage의 예시



1. fault-tolerant 확보

- 계보(**lineage**)만 기록해두면 동일한 RDD를 생성할 수 있음
- RDD의 copy를 보관하기 보다, Lineage만 보관해도 복구가 가능
- 일부 계산 코스트가 큰 RDD는 디스크에 Check pointing

reliability 확보

2. Lazy-execution이 가능해짐

- 인터프리터에서 Transformation 명령어를 읽어 들일 때는 단순히 lineage만 생성
- Action 명령어가 읽히면, 쌓여있던 lineage를 실행

Transformations	$map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$ $filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$ $flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]$ $sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling) $groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$ $reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]$ $join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$ $cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$ $crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$ $mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning) $sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$ $partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	$count() : RDD[T] \Rightarrow Long$ $collect() : RDD[T] \Rightarrow Seq[T]$ $reduce(f : (T, T) \Rightarrow T) : RDD[T] \Rightarrow T$ $lookup(k : K) : RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs) $save(path : String) : \text{Outputs RDD to a storage system, e.g., HDFS}$

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

parallelize() : 스칼라컬렉션 객체를 이용해서 RDD 객체를 생성한다.

count() : RDD의 요소 개수를 리턴한다.

first() : RDD 객체의 첫 번째 요소를 리턴한다.

collect() : RDD 객체의 모든 요소를 배열로 리턴한다

map() : 입력 RDD의 모든 요소에 f를 적용한 결과를 저장한 RDD를 반환한다.

flatMap() : 입력 RDD의 모든 요소에 f를 적용하고 모든 요소들을 하나로 묶어서 반환한다.

filter() : 입력 RDD의 모든 요소에 f를 수행하고 참을 리턴하는 결과만 저장한 RDD를 반환한다.

reduce() : 지정된 f를 수행시켜 입력 RDD의 개수를 축소시켜서 생성된 RDD를 반환한다.

reduceByKey() : 키값을 기준으로 f를 수행하고 RDD를 반환한다.

groupByKey() : 입력 RDD의 모든 요소에 f를 적용한 결과로 그룹핑된 RDD를 반환한다.