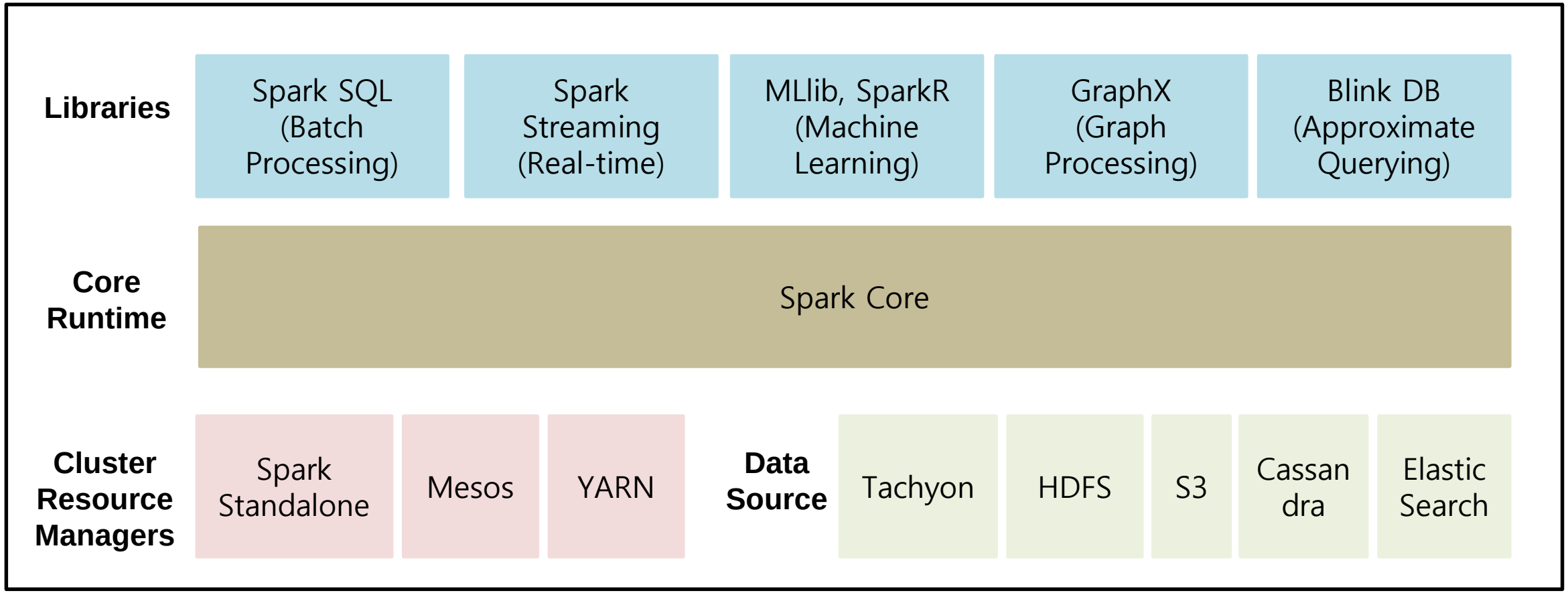
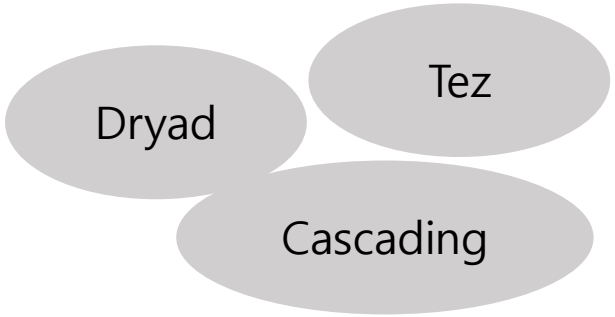
Apache Spark 개요

개요

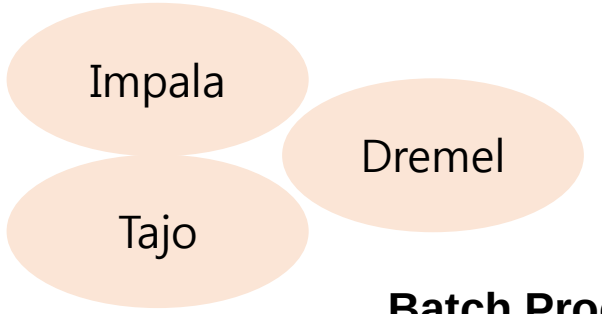
- ❑ 빠르고 범용적으로 사용할 수 있는 cluster computing platform
- ❑ Berkeley AMPLab(2009~)의 BDAS(Berkeley Data Analytics Stack)의 일부분



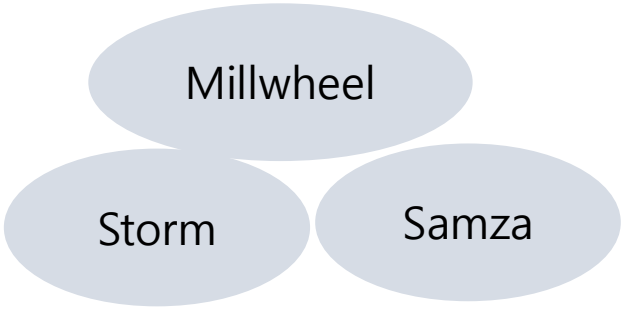
Big Data Processing Techniques



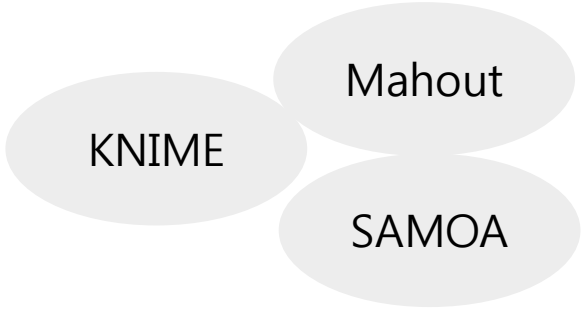
Execution Model



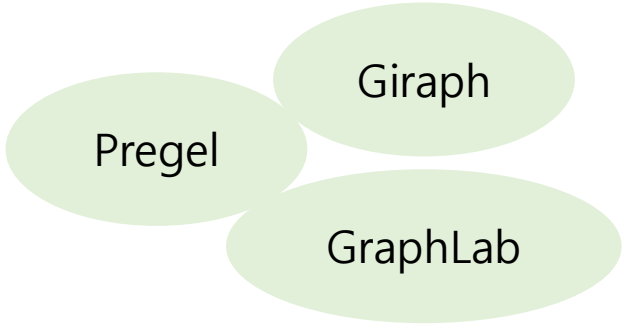
Batch Processing



Streaming Processing



Machine Learning



Graph Prorcessing

Overview

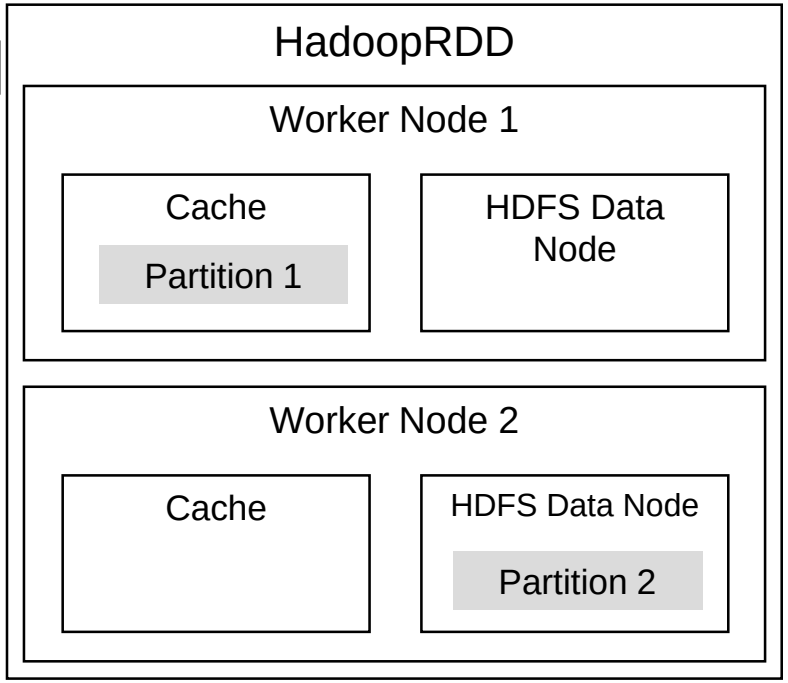
- Microsoft Dryad Paper를 기반으로 개발
- Scala로 작성(소스 코드 양 대폭 감소)
- Scala, Java, Python API 제공
- In-memory Architecture
- MapReduce에 배해 10배(disk) ~ 100배(memory) 의 성능
- Hadoop과 완벽 호완
  - HDFS, SerDes, UDF

➢ **Daytona Gray Sort 100TB Benchmark**

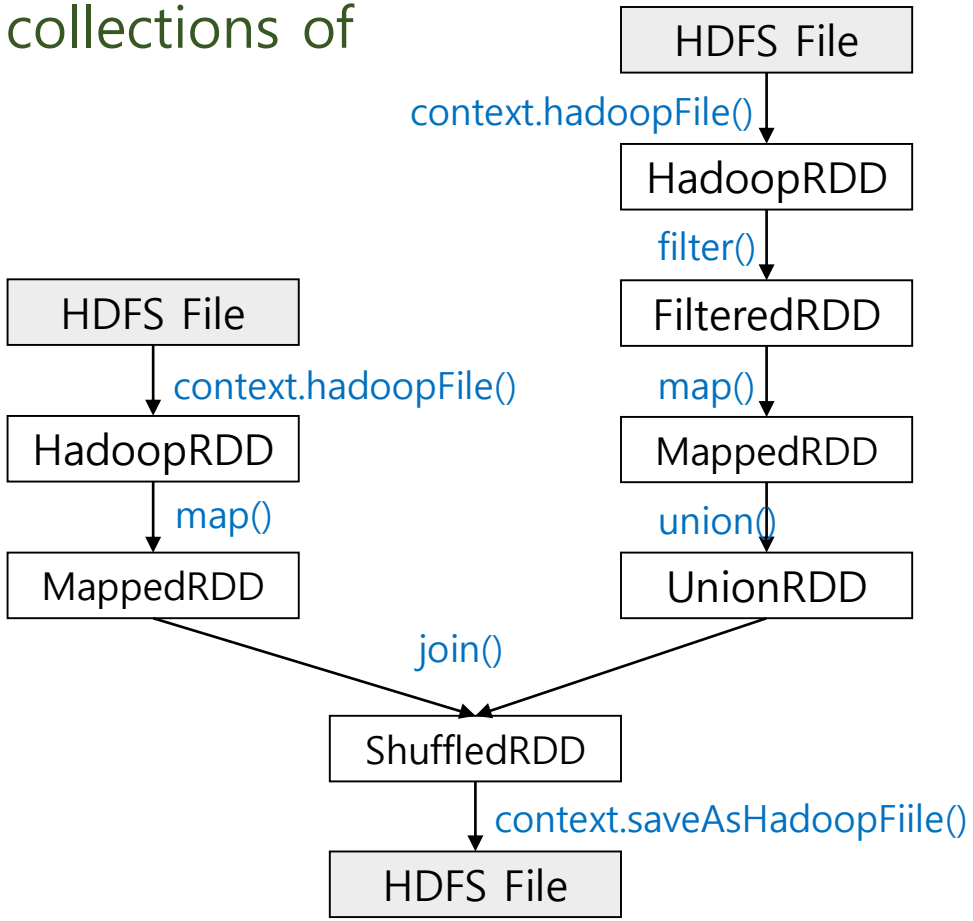
	Hadoop	Spark
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400	6592
# Reducers	10,000	29,000
Rate	1.42 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min
Sort Benchmark Daytona Rules	Yes	Yes
Environment	dedicated data ce nter	EC2 (i2.8xlarg

RDD(Resilient Distributed Dataset)

- Spark의 핵심 추상화 기법
- Immutable, recomputable, fault-tolerant partitioned collections of records
- 클러스터 노드들 사이에 파티션을 표현
- Data Set의 병렬 처리
- 파티션은 메모리나 디스크에 존재



< RDD Partition >



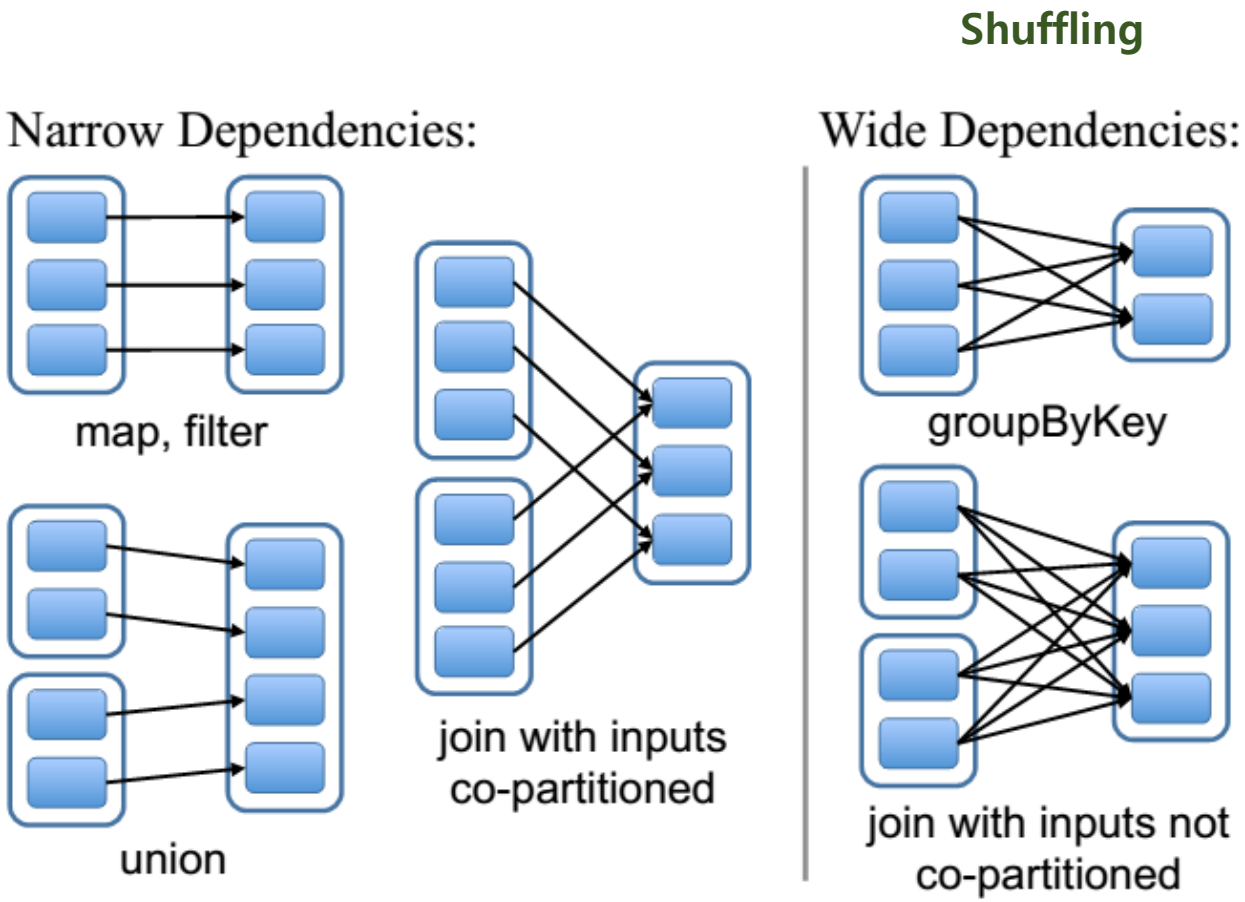
< RDD Lineage >

RDD Operation

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

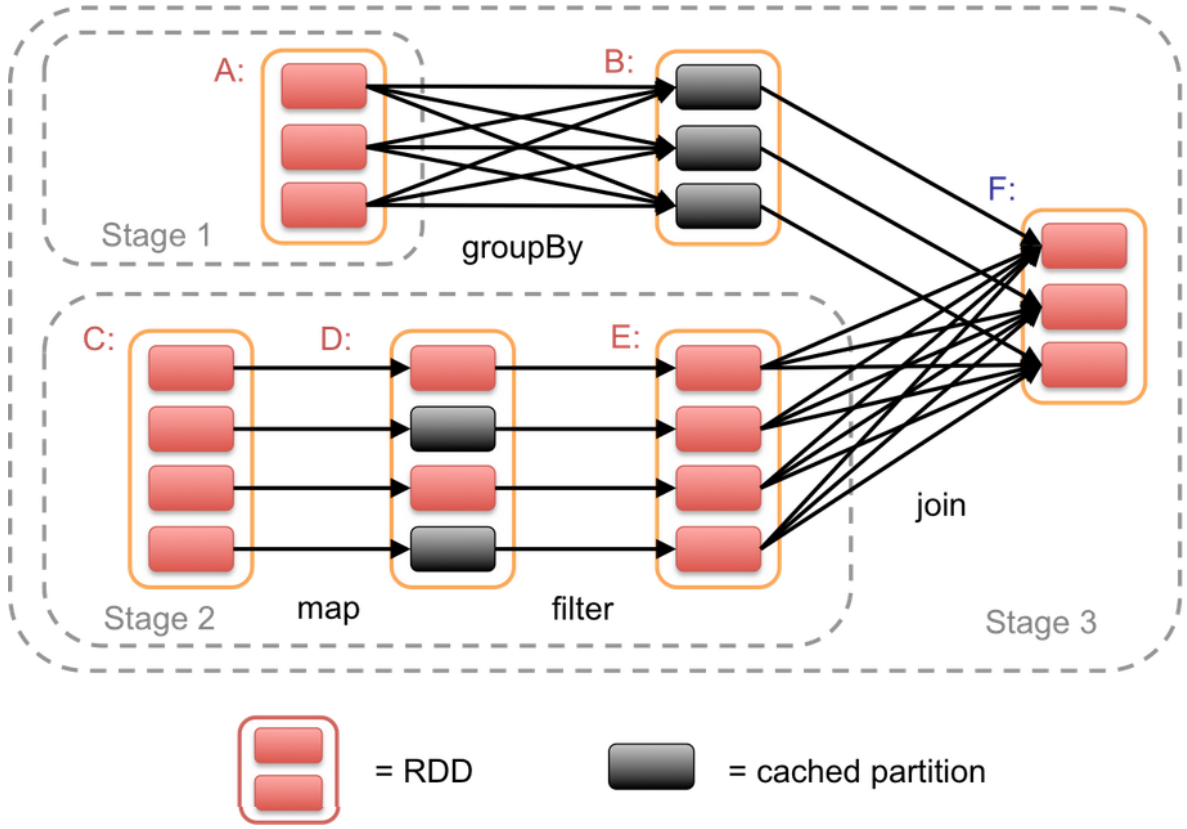
- lazy-evaluation : transformation 연산은 실제 데이터를 가져와서 RDD를 만드는 대신 RDD를 생성할 수 있는 lineage 정보만 생성, action 연산이 실행되면 그 때 실제 데이터를 가져와서 RDD를 생성하고 연산 수행
- 자원을 효율적으로 분배하여 사용 가능

RDD Dependency



Spark Execution Model

- Parallel, Distributed
- DAG-based
- Lazy evaluation
- Optimizatioins
  - Reduce disk I/O
  - Reduce shuffle I/O
  - Parallel execution
  - Task pipelining
- 데이터의 위치 파악
- fault tolerance 는 각 파티션의 RDD lineage graph를 이용

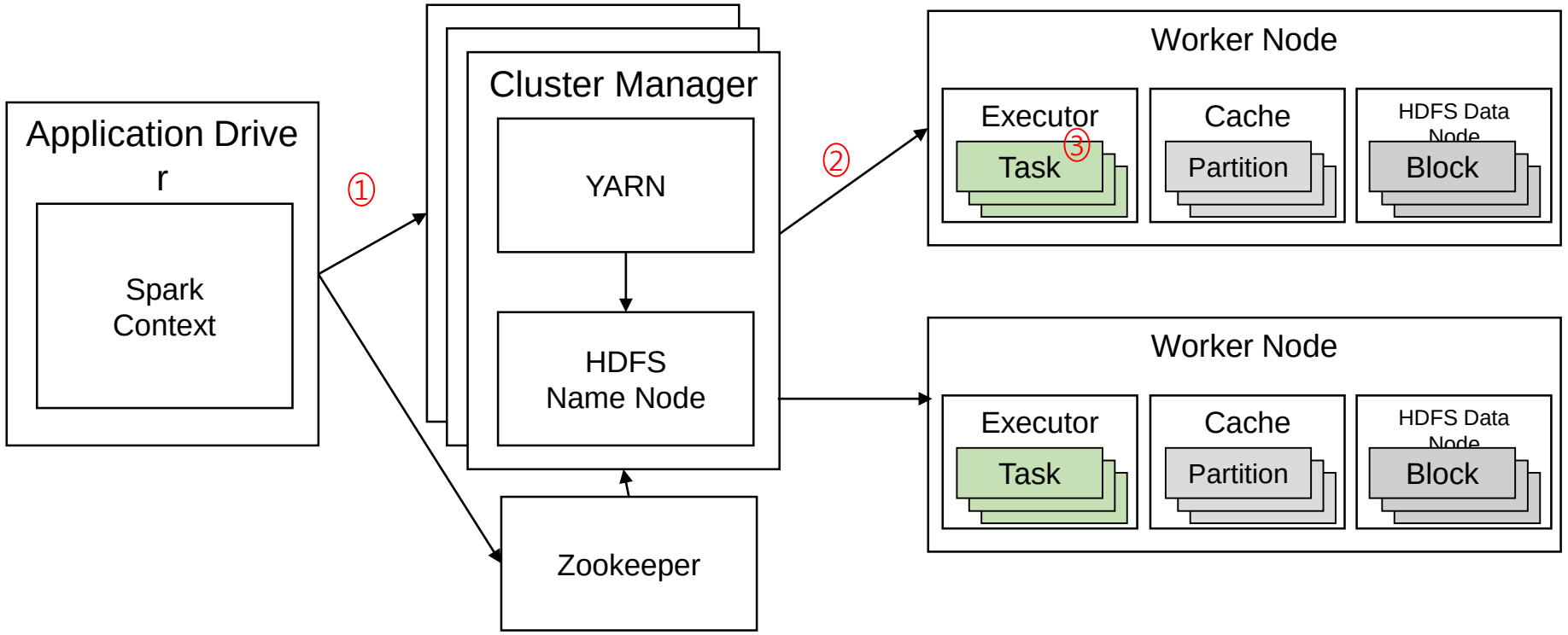


Spark Cluster

- 지원 가능 Cluster : Standalone, Mesos, YARN
- 용어 정의

용어	의미
Application	**Spark**으로 작성된 사용자 프로그램. 드라이버(**Driver**) 프로그램과 실행자들(**Executors**)로 구성
Application jar	사용자의 **Spark** 어플리케이션이 들어있는 **Jar.** 대부분 어플리케이션과 **dependency jar**들을 모두 포함 함하고 있는 **"uber jar"**형태로 사용. **Hadoop**이나 **Spark** 라이브러리는 절대 포함되서는 안된다. 그러나 나 이것들은 실행 시에 추가될 것이다.
Driver program	어플리케이션의 **main()** 함수를 실행하고 **SparkContext**를 생성하는 프로세스
Cluster manager	클러스터의 리소스를 획득하기 위한 외부 서비스(**standalone manager, Mesos, YARN**)
Deploy mode	드라이버 프로세스가 실행되는 곳 구분. 클러스터 모드에서 프레임워크는 클러스터내에 드라이버를 런치한다. 클라이언트 모드에서는 클러스터 외부에 드라이버를 런치한다.
Worker node	클러스터에서 어플리케이션 코드를 샐행 할 수 있는 노드
Executor	워커 노드에서 어플리케이션을 위해 런치된 프로세스, 태스크를 실행하고 메모리나 디스크에 데이터를 보관한다. 각 어플리케이션은 자신의 실행자를 가지고 있다.
Task	하나의 실행자에 보내지는 작업의 단위
Job	**Spark Action**(예, **save, collect**)에 반응하여 만들어진 여러 개의 작업들로 구성된 병렬 계산. 이 용어를 를 드라이버 로그에서 볼 수 있다.
Stage	각 **job**은 서로 의존성을 가지고 있고 **stage**라고 불리는 더 작은 작업의 집합으로 나뉜다. (**MapReduce** 에서 **map** 과 **reduce stages** 와 비슷) 이 용어를 드라이버 로그에서 볼 수 있다.

Spark Cluster



1. master는 어플리케이션들 사이에 리소스 배분하기 위해 cluster mange에 접근, 클러스터 상에서 task를 수행하고 데이터를 cache 할 수 있는 executor 획득
2. app code를 executor에 전송
3. task를 executor에 전송

Spark Cluster

- 각 application 은 여러 개의 thread 에서 task를 수행하는 executor process 를 획득
  - 서로 다른 application 들은 서로 독립적으로 수행(driver process , executor process 모두)
  - 따라서, 서로 다른 application 들 사이에 데이터는 공유 될 수 없음
- Spark은 특정 cluster manager에 종속적이지 않음
  - executor process 들을 얻을 수 있고 그것들이 서로 통신할 수 있다면 어떠한 cluster manager도 사용 가능
- Driver가 cluster 상의 모든 작업에 대한 Scheduling을 담당
  - 가능한 driver와 worker node는 로컬 네트워크 내에서 실행
  - 원격으로 cluster에 요청을 보내고 싶다면, driver를 worker node와 먼 곳에서 실행하지 말고 RPC를 통해 worker node와 가까운 곳에서 드라이버에게 요청을 보내야 한다.

개발 환경 구축

- ❑ JDK : http://www.oracle.com/technetwork/java/javase/downloads/index.html (Java 7+)
- ❑ maven : http://maven.apache.org/download.cgi (Maven 3.0.4 +)
- ❑ git : http://www.git-scm.com/
- ❑ spark : http://spark.apache.org/downloads.html
- ❑ spark source : git clone https://github.com/apache/spark.git -b branch-2.0.0
- ❑ IDE : https://www.jetbrains.com/idea/download/ (Scala plugin)

Hadoop Free Build Version

```
### in conf/spark-env.sh ###

# If 'hadoop' binary is on your PATH
export SPARK_DIST_CLASSPATH=$(hadoop classpath)

# With explicit path to 'hadoop' binary

export SPARK_DIST_CLASSPATH=$(/path/to/hadoop/bin/hadoop classpath)

# Passing a Hadoop configuration directory

export SPARK_DIST_CLASSPATH=$(hadoop --config /path/to/configs classpath)
```

- Spark Runtime Environment
  - Java 7+
  - Python2.6+
  - R 3.1+
  - Scala 2.10+

Build Apache Spark

- Maven (Maven 3.3.3+, Java 7+)

```
export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M -XX:ReservedCodeCacheSize=512m"
build/mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -DskipTests clean package
./dev/make-distribution.sh --name custom-spark --tgz -Psparkr -Phadoop-2.4 -Phive -Pyarn
```

- Build Profile

feature		Profile
Hadoop Version	1.x ~ 2.1.x	Hadoop-1
	2.2.x	hadoop-2.2
	2.3.x	hadoop-2.3
	2.4.x	hadoop-2.4
	2.6.x and later 2.x	Hadoop-2.6
YARN Version	2.2.x ~	yarn
Hive	0.13.1	hive
	0.12.0	hive-0.12.0
	JDBC	hive-thriftserver

Build Apache Spark

```
# Apache Hadoop 2.2.X

mvn -Pyarn -Phadoop-2.2 -Dhadoop.version=2.2.0 -DskipTests clean package

# Cloudera CDH 4.2.0

mvn -Pyarn -Dhadoop.version=2.0.0-cdh4.2.0 -DskipTests clean package

# Apache Hadoop 2.4.X with Hive 13 support

mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -Phive -Phive-thriftserver -DskipTests clean package

# for Scala 2.11

mvn -Pyarn -Phadoop-2.4 -Dscala-2.11 -DskipTests clean package

# Debian Packages

mvn -Pdeb -DskipTests clean package

# sbt build

build/sbt -Pyarn -Phadoop-2.3 assembly
```

Build Apache Spark

- Scala 2.10

```
./dev/change-scala-version.sh 2.10
./build/mvn -Pyarn -Phadoop-2.4 -Dscala-2.10 -DskipTests clean package
```

- submodule

```
./build/mvn -pl :spark-streaming_2.11 clean install
```

Spark Shell

❑ Spark Shell : Interactive Analysis, Spark Context가 내장되어 있는 Scala shell

%SPARK_HOME%\bin\spark-shell

scala> val data = 1 to 10000; data.filter(_ < 10).collect

scala> val distData = sc.parallelize(data); distData. filter(_ < 10).collect

scala> val textFile = sc. textFile("README.md") // 파일로 부터 RDD 생성, sc.textFile("hdfs://… …")

textFile: org.apache.spark.rdd.RDD[String] = README.md MappedRDD[17] at textFile at <console>:12

scala> textFile.count

scala> textFile.first

scala> val lineWithSpark = textFile.filter(line=>line.contains("Spark"))

scala> textFile.filter(_.contains("Spark")).count

scala> val words = lineWithSpark.map(_.split(" ")).map(r => r(1))

scala> words.cache

scala> words.toDebugString

Spark Shell

❑ pyspark shell

```
# bin/pyspark
>>>  textFile = sc.textFile(“README.md”)
>>> textFile.count()
```

❑ SparkR shell

```
# bin/sparkR
>  df <- read.json("people.json")
> head(df)
```

❑ SparkSQL shell

```
# bin/spark-sql
spark-sql>  CREATE TEMPORARY TABLE people
          > USING org.apache.spark.sql.json
          > OPTIONS (
          > path "examples/src/main/resources/people.json"
          > );
spark-sql> select * from people;
```

Spark Shell

- beeline shell

  # bin/beeline –u <url> -n <username> -p <password>

  # bin/beeline –e “query”

  # bin/beeline –f <query file>

- run-example shell

  # bin/run-example SparkPi

  # bin/run-example sql.RDDRelation

- spark-submit shell

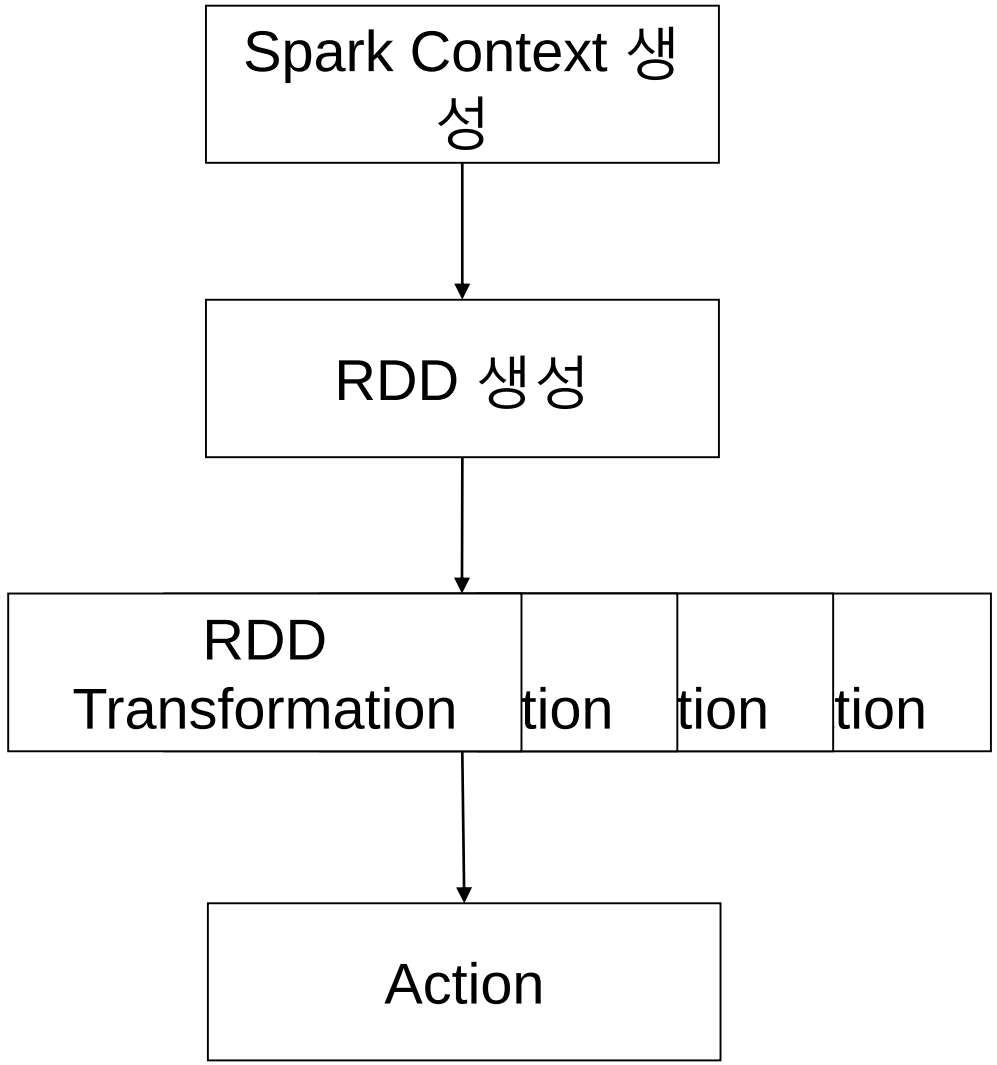
  # bin/spark-submit --master local[*] --class org.apache.spark.examples.SparkPi lib/spark-examples-1.5.1-hadoop2.6.0.jar

  # bin/spark-submit examples/src/main/python/pi.py

  # bin/spark-submit examples/src/main/r/dataframe.R

Apache Spark Core

Spark Application Structure



Spark Context

```
val conf = new SparkConf().setAppName(appName).setMaster(masterURL)
val sc = new SparkContext(conf)
```

- appName : Application 이름, masterURL : Spark Master URL

- Spark Master URL

Master URL	의미
local	하나의 worker thread로 로컬 실행(parallelism 아님)
local[n]	N개의 worker thread로 로컬 실행(core 개수로 설정)
local[*]	가능한 최대의 worker thread로 로컬 실행(최대 core 수)
spark://HOST:PORT	Spark standalone cluster URL
mesos://HOST:PORT	Mesos cluster url
mesos://zk://HOST:PORT	Zookeeper를 사용하는 mesos cluster URL
yarn-client	Yarn cluster(client mode)
yarn-cluster	Yarn cluster(cluster mode)

RDD 생성

- Parallelized Collection : Collection을 이용하여 병렬화 된 RDD 생성
  - paralleize function의 두번째 인자는 파티션 개수, 지정하지 않으면 클러스터의 CPU에 기반하여 자동 할당

```
val data= Array (1,2,3,4,5)
val distData = sc.parallelize(data)
```

```
val data= 1 to 100000
val distData = sc.parallelize(data, 5)
```

- External Datasets : 외부 저장소로부터 RDD 생성
  - local file system, HDFS, Casandra, Hbase, Amazon S3, ElasticSearch 등
  - textFile function의 두번째 인자는 파티션 개수(기본값: 각 block 당 1개, 64MB)
  - 두번째 인자는 block 보다 작게 지정할 수 없음

```
val file = sc.textFile("/tmp/data.txt")
val file = sc.textFile("/directory")
val file = sc.textFile("/directory/*.txt")
val file = sc.textFile("/directory/*.gz", 5)
```

```
val file: RDD[String]  = sc.textFile("/tmp")
val file: RDD[String, String]  = sc.wholeTextFile("/tmp")
```

Transformation Operation

❑ Base RDD : val rdd = sc.parallelize(1 to 5)

Transformation	의미
map	**def map [U : ClassTag](f : T = > U) : RDD [U]**
	함수를 통하여 소스의 각 요소를 전달하여 만든 새로운 RDD 생성

	rdd.map(_*2).collect // Array[Int] = Array(2, 4, 6, 8, 10)
	rdd.map(x => (x.toString, (x*x*x).toDouble)).collect
	// Array[(String, Double)] = Array((1,1.0), (2,8.0), (3,27.0), (4,64.0), (5,125.0))

filter	**def filter (f : T = > Boolean) : RDD [T]**
	함수가 참을 반환하는 소스의 각 요소를 선택하여 만든 RDD 생성
	rdd.filter(_ %2!=0).collect // Array[Int] = Array(1, 3, 5)
flatMap	**def flatMap [U : ClassTag](f : T = > TraversableOnce [U]) : RDD [U]**
	map과 비슷하지만, 함수의 결과가 하나 이상의 나올 수 있음. Function은 단일 item이 아닌 Sequence를 반환해야 함

	rdd.flatMap(1 to _).collect // Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5)
	rdd.map(x => x) vs. rdd.flatMap(x => x)
	rdd.map(x => List(x)) vs. rdd.flatMap(x => List(x))

Transformation Operation

Transformation	의미

mapPartitions	**def mapPartitions [U : ClassTag](f : Iterator [T] = > Iterator [U] , preservesPartitioning : Boolean = false) : RDD [U]**
	
	각 파티션(블록)별로 각자 수행되는 map function. 각 파티션의 모든 content 는 입력 함수(f: Iterator[T])를 통해 값들의 sequence로 변환될 수 있어야 하며 그 함수는 Iterator[U]를 반환해야 한다.
	
	def myfunc[T](iter: Iterator[T]) : Iterator[(T, T)] = {
	val tc = TaskContext.get() // mapPartitionWithContext
	println("Partition ID : %s, Attempt ID : %s".format(tc.partitionId(), tc.attemptId()))
	  var res = List[(T, T)](); var pre = iter.next
	  while (iter.hasNext) { val cur = iter.next; res .::= (pre, cur); pre = cur; }
	  res.iterator
	}
	rdd.mapPartitions(myfunc _).collect
	val rdd = sc.parallelized(1 to 9, 3); rdd.mapPartitions(myfunc).collect

Transformation Operation

Transformation	의미
mapPartitionsWithIndex	**def mapPartitionsWithIndex[U: ClassTag](f: (Int, Iterator[T]) => Iterator[U], preservesPartitioning: Boolean = false): RDD[U]**
	f 함수의 첫번째 인자 : 파티션 번호

	def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
	  iter.toList.map(x => index + "," + x).iterator
	}
	rdd.mapPartitionsWithIndex(myfunc).collect
	// Array(0,1, 0,2, 1,3, 1,4, 1,5, 2,6, 2,7, 3,8, 3,9, 3,10)

sample	**def sample(withReplacement: Boolean, fraction: Double, seed: Int): RDD[T]**
	Fraction과 seed를 이용하여 RDD의 item들을 무작위로 선택하여 새로운 RDD 생성, withReplacement가 true이면 값이 여러 번 나올수 있음

	rdd.sample(false, 0.5, 1).collect // Array(2, 4, 5, 6, 7, 8)
	rdd.sample(true, 0.5, 1).collect // Array(1, 6, 6, 7, 7, 9, 10)

Transformation Operation

Transformation	의미
union	**def union(other: RDD[T]): RDD[T]** 합집합 a.union(b).collect; (a ++ b).collect

intersection	**def intersection(other: RDD[T], numPartitions: Int): RDD[T]**
	교집합
	rdd.interaction(other,3) // HashPatitioner를 이용하여 3의 파티션에 저장
	rdd.interaction(other,myPart) // myPart를 이용하여 파티션에 저장

distinct	**def distinct(): RDD[T]** 중복 제거
groupBy	**def groupBy[K: ClassTag](f: T => K): RDD[(K, Iterable[T])]** 함수에 따른 그룹화 rdd.groupBy(x => x % 2 match { case 0 => "even"; case _ => "odd" }).collect // Array[(String, Iterable[Int])] = Array((even,CompactBuffer(2, 4, 6, 8, 10)), (odd,CompactBuffer(1, 3, 5, 7, 9)))

Transformation Operation

Transformation	의미
groupByKey	**def groupByKey(): RDD[(K, Iterable[V])]** **def groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]** **def groupByKey(partitioner: Partitioner): RDD[(K, Iterable[V])]**
	그룹화를 위한 함수를 제공하지 않고 patitioner에 의한 자동 분류 주의: 각 키마다의 집계(합 또는 평균 같은)를 수행하기 위해 군집화를 수행하려 한다면, reduceByKey 나 combineByKey 를 사용하는 것이 성능이 좋을 것이다. 주의: 기본적으로, 출력의 병렬화 수준은 부모 RDD의 파티션 숫자에 의존한다.
	val myPartitioner = rdd.keyBy(_%2) myPartitioner.groupByKey.collect // Array[(Int, Iterable[Int])] = Array((0,CompactBuffer(2, 4, 6, 8, 10)), (1,CompactBuffer(1, 3, 5, 7, 9)))
reduce (Action Operation)	**def reduce(f: (T, T) => T): T**
	Reduce 함수. 항상 동일한 결과를 얻으려면 함수 f는 반드시 commutative 해야 함
	rdd.reduce(_ + _); rdd.reduce(_ - _)

Transformation Operation

Transformation	의미
reduceByKey	**def reduceByKey(func: (V, V) => V): RDD[(K, V)]** **def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]** **def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)]** **def reduceByKeyLocally(func: (V, V) => V): Map[K, V]**
	RDD[(K, V)]에 대한 reduce 함수.
	val mappedRdd = rdd.map(x => (x%2,x)) mappedRdd.reduceByKey(_ + _).collect //Array[(Int, Int)] = Array((0,30), (1,25))
aggregate (Action Operation)	**def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U): U**
	RDD 각 파티션의 데이터를 seqOp 함수로 reduce 하여 combOp 함수로 combine. 이 때, zeroValue는 각 파티션에서 reduce의 처음에서 적용되고 combine 시 다시한번 적용

	val z = sc.parallelize(List(1,2,3,4,5,6), 2); z.aggregate(0)(math.max(_, _), _ + _) // 9
	val z = sc.parallelize(List("a","b","c","d","e","f"),2); z.aggregate("")(_ + _, _+_) //abcdef
	z.aggregate("x")(_ + _, _+_) //xxdefxabc

Transformation Operation

Transformation	의미

aggregate	val z = sc.parallelize(List("12","23","345","4567"),2)
	z.aggregate("")((x,y) => math.max(x.length, y.length).toString, (x,y) => x + y) //42
	z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y) //11
	val z = sc.parallelize(List("12","23","345",""),2)
	z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y) // 10
	val z = sc.parallelize(List("12","23","","345"),2)
	z.aggregate("")((x,y) => math.min(x.length, y.length).toString, (x,y) => x + y) // 11

aggregateByKey	**def aggregateByKey[U: ClassTag](zeroValue: U)(seqOp: (U, V) => U,** **combOp: (U, U) => U): RDD[(K, U)]** **def aggregateByKey[U: ClassTag](zeroValue: U, numPartitions: Int)(seqOp: (U, V) => U,** **combOp: (U, U) => U): RDD[(K, U)]** **def aggregateByKey[U: ClassTag](zeroValue: U, partitioner: Partitioner)(seqOp: (U, V) => U,** **combOp: (U, U) => U): RDD[(K, U)]**
	RDD[(K, V)]에 대한 aggregate함수.
	val mappedRdd = rdd.map(x => (x%2,x)) mappedRdd.aggregateByKey(0)(math.max(_,_), _+_).collect // Array((0,18), (1,17))

Transformation Operation

Transformation	의미
sortBy	**def sortBy[K](f: (T) ⇒ K, ascending: Boolean = true, numPartitions: Int = this.partitions.size)(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]**
	정렬

	val z = sc.parallelize(Array(("H", 10), ("A", 26), ("Z", 1), ("L", 5)))
	z.sortBy(c => c._1, true).collect // Array((A,26), (H,10), (L,5), (Z,1))
	z.sortBy(c => c._2, true).collect // Array((Z,1), (L,5), (H,10), (A,26))

sortByKey	**def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.size): RDD[P]**
	PairRDD(RDD[(K, V)])에 대한 sortBy 함수.

	val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
	val b = sc.parallelize(1 to a.count.toInt, 2)
	val c = a.zip(b)
	c.sortByKey(true).collect //Array((ant,5), (cat,2), (dog,1), (gnu,4), (owl,3))
	c.sortByKey(false).collect // Array((owl,3), (gnu,4), (dog,1), (cat,2), (ant,5))

Transformation Operation

Transformation	의미
join leftOuterJoin rightOuterJoin fullOuterJoin	**def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]** **def join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]** **def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))]**
	PairRDD(RDD[(K, V)])에 대한 inner Join 함수
	val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3) val b = a.keyBy(_.length) val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"), 3) val d = c.keyBy(_.length) b.join(d).collect; b.leftOuterJoin(d); b.rightOuterJoin(d); b.fullOuterJoin(d)
cogroup (groupWith)	**def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]** **def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)]): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]**
	3 key-value RDD 생성

Transformation Operation

Transformation	의미

cogroup
(groupWith)
	val a = sc.parallelize(List(1, 2, 1, 3), 1)
	val b = a.map((_, "b")) // Array((1,b), (2,b), (1,b), (3,b))
	val c = a.map((_, "c")) // Array((1,c), (2,c), (1,c), (3,c))
	b.cogroup(c).collect // Array((1,(CompactBuffer(b, b),CompactBuffer(c, c))),
	(2,(CompactBuffer(b),CompactBuffer(c))), (3,(CompactBuffer(b),CompactBuffer(c))))
	val d = a.map((_, "d"))
	b.groupWith(c,d).collect // Array((1,(CompactBuffer(b, b),CompactBuffer(c, c),CompactBuffer(d, d))),
	(2,(CompactBuffer(b),CompactBuffer(c),CompactBuffer(d))), 3,(CompactBuffer(b), CompactBuffer(c),
	CompactBuffer(d))))

cartesian	**def cartesian[U: ClassTag](other: RDD[U]): RDD[(T, U)]** 두 개의 RDD 간의 Cartesian product(데카르트 곱) val x = sc.parallelize(List(1,2,3)) val y = sc.parallelize(List(5,6)) x.cartesian(y).collect // Array((1,5), (1,6), (2,5), (2,6), (3,5), (3,6))

Action Operation

Action	의미
collect (toArray)	**def collect(): Array[T]** **def collect[U: ClassTag](f: PartialFunction[T, U]): RDD[U]**
	드라이버 프로그램에서 RDD의 모든 데이터를 List로 반환
count	**def count(): Long**
	RDD 데이터 개수 반환
first	**def first(): T**
	RDD의 첫 번째 데이터 반환 (= take(1))
take	**def take(num: Int): Array[T]**
	RDD의 *n* 번째 데이터 반환
takeOrdered	**def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]**
	RDD의 데이터를 정렬하여 처음 n 개의 데이터 반환
	val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2) b.takeOrdered(2) // Array(ape, cat)

Action Operation

Action	의미
takeSample	**def takeSample(withReplacement: Boolean, num: Int, seed: Int): Array[T]**
	Sample과 유사하지만 takeSample은 정확히 num 개의 데이터를 무작위 순서로 Array로 반환
	rdd.takeSample(true, 3, 1) // Array(9, 4, 7)
saveAsTextFile	**def saveAsTextFile(path: String)** **def saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec])**
	RDD를 로컬 파일 시스템, HDFS 또는 다른 하둡 지원 파일 시스템에 저장. 파티션 개수 만큼 파일 생성
	rdd.saveAsTextFile("c:/temp/rdd") import org.apache.hadoop.io.compress.GzipCodec rdd.saveAsTextFile("mydata_b", classOf[GzipCodec]) // 압축
saveAsOjbectFile	**def saveAsObjectFile(path: String)**
	RDD를 직렬화하여 바이너리 형태로 저장
	rdd.saveAsObjectFile(“objFile”) val loadRdd = sc.objectFile(“objFile”)

Action Operation

Action	의미
saveAsSequnceFile	**def saveAsSequenceFile(path: String, codec: Option[Class[_ <: CompressionCodec]] = None)**
	RDD를 Hadoop sequnce 파일로 저장
countByKey	**def countByKey(): Map[K, Long]**
	RDD[(K, V)]에 대한 count 함수
countByValue	**def countByValue(): Map[T, Long]**
	Value 값과 그 값이 나타나는 회수에 대한 Map 반환, 단일 reducer에서 aggregation이 일어남
	val b = sc.parallelize(List(1,2,3,4,5,6,7,8,2,4,2,1,1,1,1,1)) b.countByValue //Map(5 -> 1, 8 -> 1, 3 -> 1, 6 -> 1, 1 -> 6, 2 -> 3, 4 -> 2, 7 -> 1)
foreach	**def foreach(f: T => Unit)**
	RDD의 각 데이터에 대하여 f 함수 실행
	rdd.foreach(x => println(x)); rdd.foreach(println(_)); rdd.foreach(println)
foreachPartition	**def foreachPartition(f: Iterator[T] => Unit)**
	RDD의 각 파티션에서 f 수행

Passing Function

- Anonymous function

```
rdd.map(x => x * 2)
```

- Static function in Singleton

```
object MyFunctions {
    def double(x: int):int = x * 2
}
rdd.map(MyFunctions.double)
```

- Static function in class

```
class MyClass {
    def double(x: int):int = x * 2
    def doSomething(rdd) =
                rdd.map(double)
}
```

- Variable scope

```
class MyClass {
    val base = 3
    def doSomething(rdd) =  rdd.map(x => base + 3)
}
```

- Variable scope

```
class MyClass {
    val base = 3
    def doSomething(rdd) =  {
        val base_ = this.base
        rdd.map(x => base_ + 3)
    }
}
```

Exection Location

- rdd.foreach(println)
- rdd.map(println)
- rdd.collect().foreach(println)
- rdd.take(100).foreach(println)

Spark Application

❑ Spark Application 작성 준비

1. idea 실행
2. Create New Project > maven
3. Scala library 확인
4. pom.xml 작성 - Spark Dependency 추가
5. Add Framework Support > Scala 추가
6. Maven Project로 변경
7. New > Scala class > 이름 입력> Object 선택
8. Coding

Spark Application

- maven – pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.raonbit.nuclear</groupId>
    <artifactId>nuclear</artifactId>
    <version>1.0</version>

    <properties>
        <spark.version>2.0.0</spark.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.11</artifactId>
            <version>${spark.version}</version>
            <!--scope>provided</scope-->
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
    </dependencies>
```

Spark Application

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object First extends App{
  val conf = new SparkConf().setAppName("First").setMaster("local[*]")
  val sc = new SparkContext(conf)
  val readme = sc.textFile("C:\\edu\\spark\\README.md")
  val lineWithSpark = readme.filter(line=>line.contains("Spark"))
  val words = lineWithSpark.map(_.split(" ")).map(r => r(1))
  words.cache()
  val aCnt = words.filter(_.contains("a")).count()
  val bCnt = words.filter(_.contains("b")).count()
  println("Lines with a : %s, Lines with b : %s".format(aCnt, bCnt))
}
```

반드시 모두 선언
(implicit conversion)

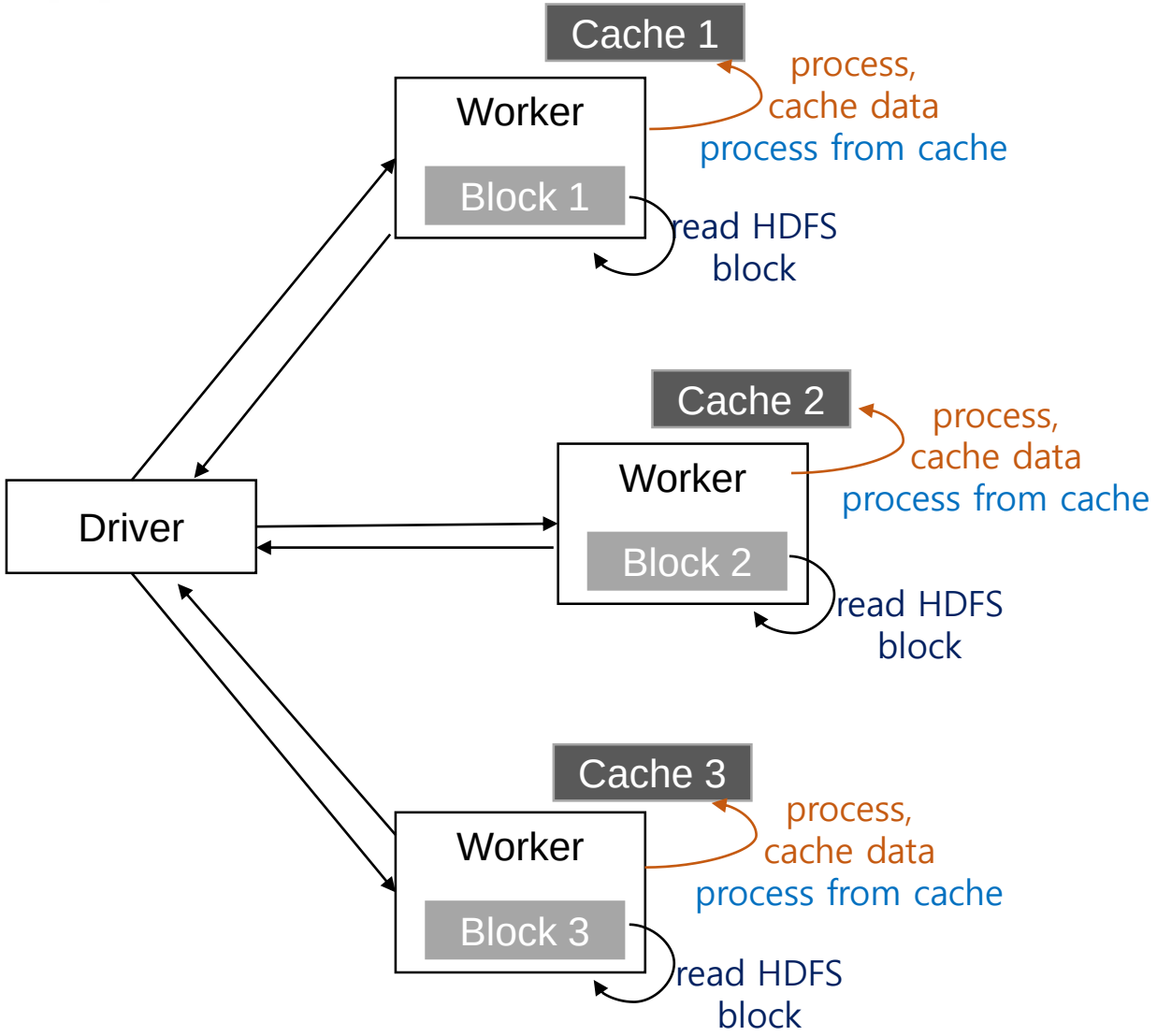
Spark Application(Java)

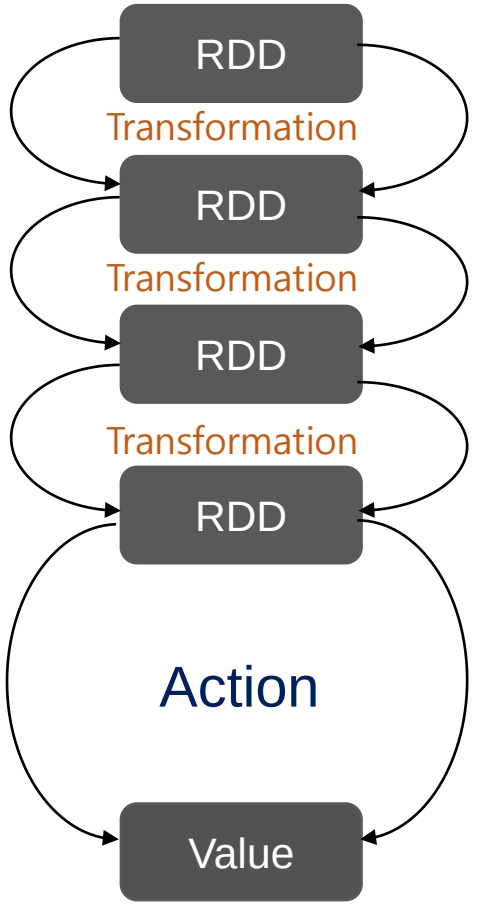
```
import java.util.Arrays;
import java.util.regex.Pattern;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;

public class First {

  private static final Pattern SPACE = Pattern.compile(" ");

  public static void main(String[] args) {

    SparkConf conf = new SparkConf().setAppName("First").setMaster("local[*]");
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<String> readme = sc.textFile("D:\\spark\\README.md");
    JavaRDD<String> lineWithSpark = readme.filter(new Function<String, Boolean>() {
```

Spark Application(Java)

```
    @Override
    public Boolean call(String s) throws Exception { return s.contains("Spark"); } });
    JavaRDD<String> words = lineWithSpark.flatMap(new FlatMapFunction<String, String>() {

    @Override
    public Iterable<String> call(String s) { return Arrays.asList(SPACE.split(s)); } });

    words.cache();

    JavaRDD<String> containa = words.filter(new Function<String, Boolean>() {
    @Override
    public Boolean call(String s) throws Exception { return s.contains("a"); } });
    JavaRDD<String> containb = words.filter(new Function<String, Boolean>() {

    @Override
    public Boolean call(String s) throws Exception { return s.contains("b"); } });
      System.out.println(String.format("Lines with a : %s, Lines with b : %s",
                                       containa.count(), containb.count()));
  }
}
```

Spark Application





MapReduce vs. Spark Application

```
package com.raonbit.edu;

import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class WordCount {
   public static class Map extends MapReduceBase implements Mapper<LongW
ritable, 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, IntWrita
ble> 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 Redu
cer<Text, IntWritable, Text, IntWritable> {
       public void reduce(Text key, Iterator<IntWritable> values, OutputColl
ector<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);
   }
}
```

MapReduce vs. Spark Application

```scala
package com.raonbit.edu

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object WordCount {

  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("WordCount").
               setMaster("local[*]")
    val sc = new SparkContext(conf)
    val file = sc.textFile(args(0))
    val word = file.flatMap(_.split(" ")).map(w => (w, 1)).cache()
    word.reduceByKey(_ + _).saveAsTextFile(args(1))
  }

}
```

RDD Persistence(Caching)

- rdd.persist() (== rdd.cache()) 를 통해 RDD를 저장, 인자로 저장 수준을 전달하지 않으면 메모리에만 저장
  - cache() == cache(StorageLevel.MEMORY_ONLY)

저장 수준	의미
MEMORY_ONLY	기본값, RDD를 직렬화 하지 않은 객체로 메모리에 저장. RDD가 메모리 보다 크면 어떤 파티션은 저장되지 않고 필요할 때 마다 재 산출 된다.
MEMORY_AND_DISK	RDD를 직렬화 하지 않은 객체로 메모리에 저장. RDD가 메모리 보다 크면 나머지 파티션들은 디스크에 저장하고 필요할 때 마다 읽어온다.
MEMORY_AND_SER	RDD를 직렬화 한 객체로 메모리에 저장. 공간이 적게 필요하지만 읽는데 CPU 소모가 많다
MEMORY_AND_DISK_SER	RDD를 직렬화 한 객체로 메모리에 저장. RDD가 메모리 보다 크면 나머지 파티션들은 디스크에 저장하고 필요할 때 마다 읽어온다.
DISK_ONLY	RDD를 디스크에만 저장
MEMORY_ONLY_2, MEMORY_AND_DISK_2, …	위와 동일, 차이점은 각 파티션을 두 개의 클러스터 노드에 복제

- 명시적으로 저장하지 않아도도 reduceByKey와 같은 shuffle 연산 중에 중간 데이터 자동 저장
- 중간 데이터(RDD)를 재사용할 계획이라면 명시적으로 persist(cache) 호출 권장

RDD Persistence(Caching)

- 저장 수준 선택 가이드
  - RDD의 크기가 Memory 용량 보다 작으면 MEMORY_ONLY 사용, 가장 빠르고 효율적
  - 그렇지 않으면, MEMORY_ONLY_SER 사용. 이때 Kryo serialization을 사용하면 빠른 접근 가능
  - RDD를 생성하는데 아주 많은 자원을 사용지 않거나 많은 양의 데이터를 필터링 하지 않는다면 디스크에 저장하지 말 것. 디스크에서 읽어 오는 것보다 재 산출이 보다 효율적
  - 빠른 장애 복구를 원한다면 클러스터에 복제하는 저장 수준 사용
- 오래된 데이터 파티션은 LRU(latest-recently-used)에 의해 제거
- 명시적으로 제거하려면 unpersist() 호출

Broadcast Variables, Accumulator

- Broadcast Variable
  - 읽기 전용 변수의 복사본을 클러스터 상의 각 task에 복사하여 보내는 대신 각 노드의 cache에 보관하는 것
  - 큰 입력 데이터의 복사본을 각 노드에 효율적으로 보내는데 사용

```
val broadcastVar = sc.broadcast(Array(1,2,3))
boradcastVar.value() // return [1, 2, 3]
```

- Accumulator
  - associative 연산을 통하여 누적만 가능한 변수, counter나 sum을 병렬로 효율적으로 구현하는데 사용
  - 숫자 형과 standard mutable collection에 대해 accumulator를 기본 제공하고, 확장 가능
  - driver 프로그램만이 accumulator 값을 읽을 수 있음

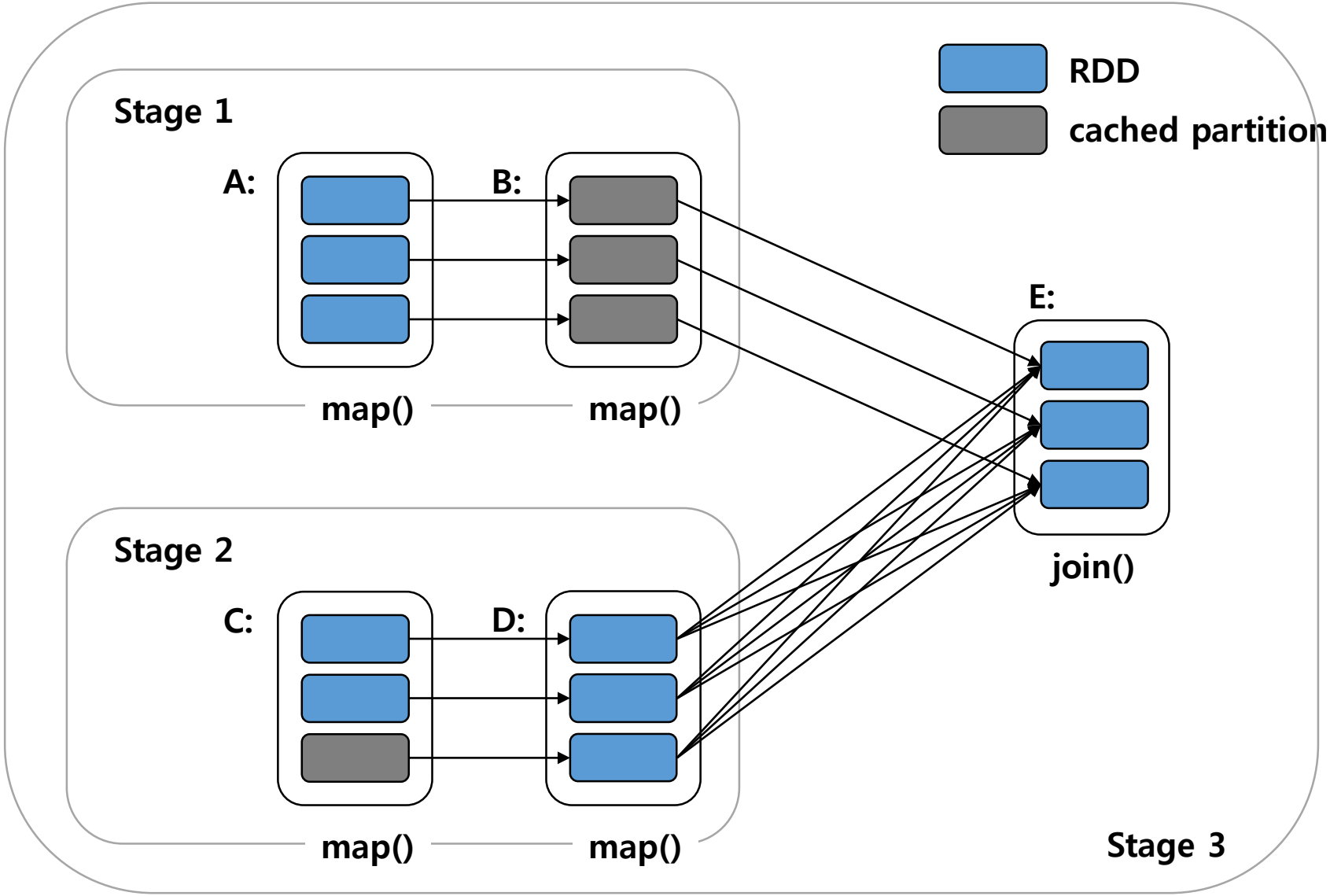
```
val accm = sc.accumulator(0)
sc.parallelize(List(1,2,3,4)
    .foreach(x => accum.add(x))
accum.value()
```

Spark Application Operator Graph

```
val format = new java.text.SimpleDateFormat("yyyy-MM-dd")
case class Register (d: java.util.Date, uuid: String, cust_id: String, lat: Float,
                     lng: Float)
case class Click (d: java.util.Date, uuid: String, landing_page: Int)
val reg = sc.textFile("reg.tsv").map(_.split("\t")).map(
  r => (r(1), Register(format.parse(r(0)), r(1), r(2), r(3).toFloat, r(4).toFloat))
)
val clk = sc.textFile("clk.tsv").map(_.split("\t")).map(
  c => (c(1), Click(format.parse(c(0)), c(1), c(2).trim.toInt))
)
reg.join(clk).toDebugString
res5: String =
FlatMappedValuesRDD[46] at join at :23 (1 partitions)
    MappedValuesRDD[45] at join at :23 (1 partitions)
        CoGroupedRDD[44] at join at :23 (1 partitions)
            MappedRDD[36] at map at :16 (1 partitions)
                MappedRDD[35] at map at :16 (1 partitions)
                    MappedRDD[34] at textFile at :16 (1 partitions)
                        HadoopRDD[33] at textFile at :16 (1 partitions)
            MappedRDD[40] at map at :16 (1 partitions)
                MappedRDD[39] at map at :16 (1 partitions)
                    MappedRDD[38] at textFile at :16 (1 partitions)
                        HadoopRDD[37] at textFile at :16 (1 partitions)
```

Parallel Processing

Spark Application Operator Graph



Apache Spark Streaming

Overview

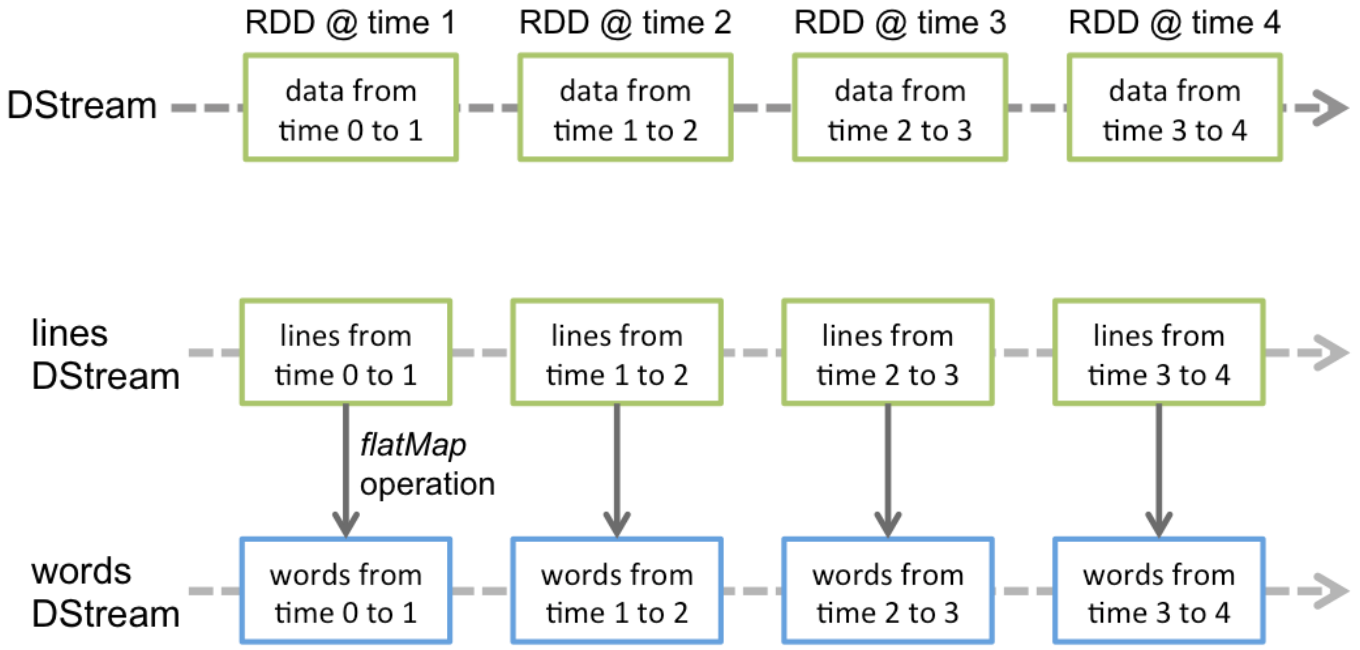
❑ 대규모의 실시간 데이터 처리를 위한 고성능의 장애 허용 framework, Spark Core 확장 API

❑ streaming 연산을 아주 작은 batch 작업의 연속으로 처리

- 실시간 stream을 X 초의 batch 들로 나눔, 0.5초 보다 작은 batch의 지연은 대략 1초
- 각 각의 batch 는 RDD이고 RDD 연산을 사용하여 처리 가능
- RDD 연산의 처리 결과가 batch에 반환

DStream

- Spark Streaming의 Programming Model
  - Stream Data를 표현하는 연속된 RDD
  - RDD와 마찬가지로 input source에서 생성하거나 Dstream을 transform 하여 생성
  - RDD 연산을 그대로 사용 – Batch(Historical) Data와 Stream Data를 동일한 방식으로 처리



Spark Streaming Application

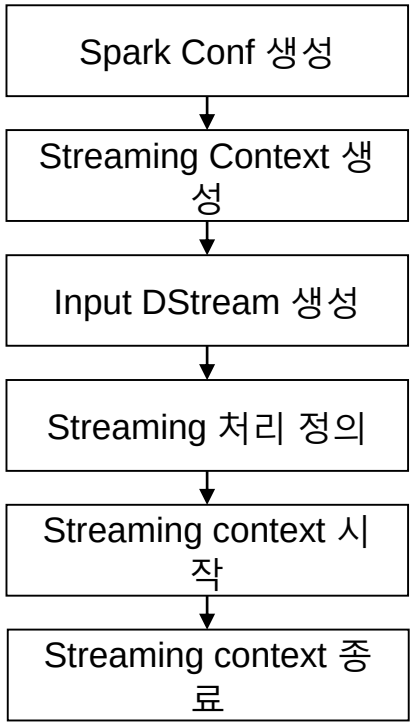
```
package org.apache.spark.examples.streaming

import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}

object HdfsWordCount {
  def main(args: Array[String]) {
    if (args.length < 1) {
      System.err.println("Usage: HdfsWordCount <directory>")
      System.exit(1)
    }

    StreamingExamples.setStreamingLogLevels()
    val sparkConf = new SparkConf().setAppName("HdfsWordCount")
    val ssc = new StreamingContext(sparkConf, Seconds(2))

    val lines = ssc.textFileStream(args(0))
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
    wordCounts.print()
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Start Streaming programming

❑ Libraries Dependency

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.10</artifactId>
    <version>1.5.2</version>
</dependency>
```

❑ Streaming programming Structure




```
val sparkConf = new SparkConf().setAppName("TwitterPopularTags").setMaster("local[*]")

val ssc = new StreamingContext(sparkConf, Seconds(2))

val stream = TwitterUtils.createStream(ssc)

// business logic

ssc.start()
ssc.awaitTermination()

ssc.stop()
```

Input DStreams

- Streaming 소스에서 들어오는 데이터 스트림을 표현하는 것으로 StreamingContext API에서 직접 제공하는 Basic source와 외부 library에 존재하는 Advanced source로 구분
  - File Stream을 제외한 모든 input dstream은 하나의 receiver 객체와 연결, 따라서 모든 input dstream은 데이터의 단일 스트림을 받음
  - receiver는 streaming application에 할당된 하나의 core 점유
  - application에 할당된 core의 개수가 receiver의 개수 보다 작으면 데이터 수집 불가
  - application을 로컬에서 하나의 core로 실행하면 처리 불가

Input DStreams – Basic Source

```
def actorStream[T: ClassTag](props: Props, name: String, slevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2,
    supervisorStrategy: SupervisorStrategy = ActorSupervisorStrategy.defaultStrategy
  ): ReceiverInputDStream[T]

def socketTextStream(hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
  ): ReceiverInputDStream[String]

def socketStream[T: ClassTag](hostname: String, port: Int, converter: (InputStream) => Iterator[T],
    storageLevel: StorageLevel
  ): ReceiverInputDStream[T]

def rawSocketStream[T: ClassTag](hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
  ): ReceiverInputDStream[T]

def fileStream[K: ClassTag, V: ClassTag,F <: NewInputFormat[K, V]: ClassTag]
    (directory: String): InputDStream[(K, V)]

def fileStream[K: ClassTag,V: ClassTag,F <: NewInputFormat[K, V]: ClassTag]
  (directory: String, filter: Path => Boolean, newFilesOnly: Boolean): InputDStream[(K, V)]

def textFileStream(directory: String): DStream[String]

def queueStream[T: ClassTag](queue: Queue[RDD[T]], oneAtATime: Boolean = true): InputDStream[T]

def queueStream[T: ClassTag](queue: Queue[RDD[T]], oneAtATime: Boolean, defaultRDD: RDD[T]): InputDStream[T]
```

Input DStreams – Advanced Source

- Twitter (TwitterUtils)

```
def createStream(ssc: StreamingContext,   twitterAuth: Option[Authorization],  filters: Seq[String] = Nil,
   storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[Status]
```

- Kafka (KafkaUtils)

```
def createStream(ssc: StreamingContext, zkQuorum: String, groupId: String, topics: Map[String, Int],
storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[(String,String)]
```

- Flume (FlumeUtils)

```
def createStream(ssc: StreamingContext, hostname: String, port: Int,
storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[SparkFlumeEvent]
```

- Kinesis (KinesisUtils)

```
def createStream(ssc: StreamingContext, streamName: String,endpointUrl: String, checkpointInterval: Duration,
initialPositionInStream:InitialPositionInStream, storageLevel: StorageLevel):ReceiverInputDStream[Array[Byte]]
```

- MQTT (MQTTUtils)

```
def createStream(ssc: StreamingContext, brokerUrl: String, topic: String,
storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2):ReceiverInputDStream[String]
```

- ZeroMQ(ZeroMQUtils)

```
def createStream[T](ssc: StreamingContext, publisherUrl: String, subscribe:Subscribe, bytesToObjects: (Seq[ByteString])
⇒ Iterator[T], storageLevel:StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2, supervisorStrategy:SupervisorStr
ategy = ...)(implicit arg0: ClassTag[T]):ReceiverInputDStream[T]
```

Transformation Operation

Transformation	의미
transform	**def transform[U: ClassTag](transformFunc: RDD[T] => RDD[U]): DStream[U]** **def transform[U: ClassTag](transformFunc: (RDD[T], Time) => RDD[U]): DStream[U]**
	Dstream의 모든 RDD마다 함수를 적용하여 새로운 DStream을 생성, Dstream 에 직접 사용할 수 없는 RDD 연산 수행 가능, Mllib, GraphX 를 Dstream에 직접 적용 가능

	val spamRDD .. // spam 정보를 담고 있는 RDD
	val cleanedDStream = wordCounts.transform(rdd => {
	rdd.join(spamInfoRDD).filter(...) // Dstream의 각 RDD와 spamRDD join
	})

updateStateByKey	**def updateStateByKey[S: ClassTag](updateFunc: (Seq[V], Option[S]) => Option[S]): DStream[(K, S)]** **def updateStateByKey[S: ClassTag](updateFunc: (Seq[V], Option[S]) => Option[S], numPartitions: Int): DStream[(K, S)]** **def updateStateByKey[S: ClassTag](updateFunc: (Seq[V], Option[S]) => Option[S], partitioner: Partitioner): DStream[(K, S)]** **def updateStateByKey[S: ClassTag](updateFunc: (Iterator[(K, Seq[V], Option[S])]) => Iterator[(K, S)], partitioner: Partitioner, rememberPartitioner: Boolean): DStream[(K, S)]**

Transformation Operation

Transformation	의미
updateStateByKey	주어진 함수가 적용되어 각 키의 상태가 갱신되었을 때, 키의 이전 상태와 새로운 상태를 반환. 이것은 각 키의 상태 데이터를 관리하는데 사용

```
val updateFunc = (values: Seq[Int], state: Option[Int]) => {
    val currentCount = values.sum
    val previousCount = state.getOrElse(0)
    Some(currentCount + previousCount)    }
val stateDstream = wordDstream.updateStateByKey[Int](updateFunc)
```

Checkpointing

❑ 장애 복구를 위한 정보 저장

- Metadata checkpointing : Streaming 구성 정보를 HDFS 같은 장애 허용 저장소에 저장. driver 노드에서 발생한 장애를 복구 할 때 사용.
  - ✓ Configuration – Streaming application을 생성하는데 사용했던 구성 정보
  - ✓ Dstream operation - Streaming application을 정의하는데 사용했던 Dstream 연산 집합
  - ✓ Incomplete batches – queue에 쌓여 있으나 처리되지 않은 batch 들
- Data checkpointing : 생성된 RDD 들을 저장. 여러 개의 batch들 사이에서 데이터를 합하는 stateful transformation에서는 필수(예, updateStateByKey, window operations)

❑ 설정 방법

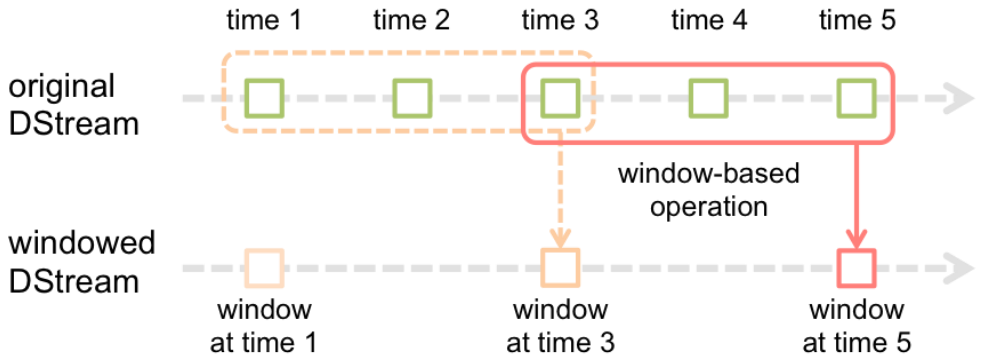
```
// StreamContext or Spark Context
ssc.checkpoint(directory)
// DStream
stream.checkpoint(Duration)
```

❑ Driver 실패 복구

```
def functionToCreateContext(): StreamingContext = {
  val ssc = new StreamingContext(...) // new context
  val lines = ssc.socketTextStream(...) // create DStreams
  ...
  ssc.checkpoint(checkpointDirectory) // set checkpoint directory
  ssc
}
// Get StreamingContext from checkpoint data or create a new one
val context = StreamingContext.getOrCreate
              (checkpointDirectory, functionToCreateContext _)
```

Operation

Transformation	Window	Output
▪ map(func), flatMap(func), filter(func), count() ▪ repartition(numPartitions) ▪ union(otherStream) ▪ reduce(func), countByValue(), reduceBykey(func, [numTasks]) ▪ join(otherStream, [numTasks]), cogroup(otherStream, [numTasks]) ▪ transform(func) ▪ updateStateByKey(func)	▪ window(length, interval) ▪ countByWindow(length, interval) ▪ reduceByWindow(func, length, interval) ▪ reduceByKeyAndWindow(func, length, interval, [numTasks]) ▪ countByValueAndWindow(length, interval, [numTasks])	▪ Print() ▪ foreachRDD(func) ▪ saveAsObjectFiles(prefix, [suffix]) ▪ saveAsTextFiles(prefix, [suffix]) ▪ saveAsHadoopFiles(prefix, [suffix])



- window length : window의 기간
- sliding interval : window 연산이 수행되는 간격
  (batch 크기의 배수로 지정해야 함)
- checkpointing 필수 : checkpointing duration은 sliding interval 의 5~10배가 적당

Window Operation(Transformation)

Transformation	의미
window	**def window(windowDuration:Duration)** **def window(windowDuration:Duration , slideDuration: Duration)**
	새로운 window DStream 생성
countByWindow	**def countByValueAndWindow(windowDuration: Duration, slideDuration: Duration,** **numPartitions: Int = ssc.sc.defaultParallelism)(implicit ord: Ordering[T] = null) : DStream[(T, Long)]**
	Window item count 함수
countByValueAndWind ow	**def countByValueAndWindow(windowDuration: Duration, slideDuration: Duration,** **numPartitions: Int = ssc.sc.defaultParallelism)(implicit ord: Ordering[T] = null) : DStream[(T, Long)]**
	windowDuration 동안의 데이터를 Key, 해당 데이터의 개수를 Value로 담고 있는 Dstream 생성

Window Operation(Transformation)

Transformation	의미
reduceByWindow	**def reduceByWindow(reduceFunc: (T, T) => T, windowDuration: Duration, slideDuration: Duration): DStream[T]** **def reduceByWindow(reduceFunc: (T, T) => T, invReduceFunc: (T, T) => T, windowDuration: Duration, slideDuration: Duration): DStream[T]**
	Window reduce 함수
	val retweetCnt = stream.map(status => status.getRetweetCount) val retSum = retweetCnt.reduceByWindow(_+_, Seconds(4), Seconds(4)) retSum.print val retSum = retweetCnt.reduceByWindow(_+_, _-_, Seconds(4), Seconds(4)) retSum.print

Window Operation(Transformation)

Transformation	의미
reduceByKeyAndWind ow	**def reduceByKeyAndWindow(reduceFunc: (V, V) => V, windowDuration: Duration): DStream[(K, V)]** **def reduceByKeyAndWindow(reduceFunc: (V, V) => V, windowDuration: Duration,** **slideDuration: Duration): DStream[(K, V)]** **def reduceByKeyAndWindow(reduceFunc: (V, V) => V, windowDuration: Duration,** **slideDuration: Duration, numPartitions: Int): DStream[(K, V)]**
	DStream[(K,V)]의 reduceByWindow 함수,

Join Operation

- Join

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

- Window Join

```
val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)
```

- Stream dataset Join

```
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```

Output Operation

Output	의미
print	**def print()** 드라이버 노드에서 실행되며 DStream의 각 배치에서 처음 10개의 데이터 출력, 개발시에 디버깅 용도로 유용
foreachRDD	**def foreachRDD(foreachFunc: RDD[T] => Unit)** **def foreachRDD(foreachFunc: (RDD[T], Time) => Unit)** Dstream 안의 각 RDD에 foreachFunc 적용. RDD를 파일로 저장하거나 네트워크를 통해 데이터베이스에 저장하는 것 처럼 각 RDD를 외부 시스템으로 보냄. foreachFunc 함수는 stream application이 실행되고 있는 드라이버 프로세스에서 실행되고 보통 RDD action을 가지고 있어서 스트림의 RDD연산 수행.

Output Operation

- foreachRDD Design Pattern

```
dstream.foreachRDD(rdd => {
  val connection = createNewConnection()  // executed at the driv
er
  rdd.foreach(record => {
    connection.send(record) // executed at the worker
  })
})
```

connection 객체가 각 worker로 보내져야 함
connection 객체는 직렬화 할 수 없음

```
dstream.foreachRDD(rdd => {
  rdd.foreach(record => {
    val connection = createNewConnection()
    connection.send(record)
    connection.close()
  })
})
```

모든 record 마다 connection을 만드는 대신
파티션마다 connection 생성

```
dstream.foreachRDD(rdd => {
  rdd.foreachPartition(record => {
    val connection = createNewConnection()
    record.foreach(r => connection.send(r))
    connection.close()
  })
})
```

ConnectionPool 사용

```
dstream.foreachRDD(rdd => {
  rdd.foreachPartition(record => {
    val connection = ConnectionPool.getConnection
()
    record.foreach(r => connection.send(r))
    ConnectionPool. returnConnectioin(connection)
  })
})
```

Apache Spark SQL

Overview

- SQL, HiveQL, Scala로 표현된 관계 형 질의가 Spark을 이용하여 실행될 수 있도록 하는 확장 API
- DataFrames (Dataset[Row])
  - 이름있는 컬럼으로 구성된 분산 데이터 컬렉션
  - 관계형 데이터 베이스의 테이블이나 R/Python의 data frame과 개념적으로 동일
  - RDD, Parquet 파일, JSON dataset, Hive에 저장된 데이터로 부터 생성 가능

SQLContext

- Libraries Dependency

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.0.0</version>
</dependency>
```

- SparkSession

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder()
            .appName("Spark SQL Example")
            .config("spark.some.config.option", "some-value")
            .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

Data Sources

```
val usersDF = spark.read.load("examples/src/main/resources/users.parquet")
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")

val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

val sqlDF = spark.sql("SELECT * FROM
parquet.`examples/src/main/resources/users.parquet`")

val csvDF = spark.read.option("header","true").csv("src/main/resources/sales.csv")
```

DataFrame Operations

```
df.show();

df.printSchema();

df.select("name").show();

df.select(df.col("name"), df.col("age").plus(1)).show();

df.filter(df.col("age").gt(21)).show();

df.groupBy("age").count().show();
```

SparkSQLExample.scala

Schema Mapping

- Reflection

```
import spark.implicits._

case class Person(name: String, age: Int)

val people = spark.sparkContext.textFile("examples/src/main/resources/people.txt")
                .map(_.split(","))
                .map(p => Person(p(0), p(1).trim.toInt)).toDF()

people.createOrReplaceTempView("people")

val teenagers = spark.sql("SELECT name, age FROM people WHERE age >= 13 AND age <= 19")

teenagers.map(t => "Name: " + t(0)).show()
teenagers.map(t => "Name: " + t.getAs[String]("name")).show()

implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
implicit val stringIntMapEncoder: Encoder[Map[String, Int]] = ExpressionEncoder()
teenagers.map(_.getValuesMap[Any](List("name", "age"))).collect().foreach(println)
```

Schema Mapping

- Programmatically

```
val people = spark.sparkContext.textFile("examples/src/main/resources/people.txt")
val schemaString = "name age"
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._
val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
val peopleDataFrame = spark.createDataFrame(rowRDD, schema)
peopleDataFrame.createOrReplaceTempView("people")
val results = spark.sql("SELECT name FROM people")
results.map(t => "Name: " + t(0)).show()
```

Spark Streaming 과의 연동

```
val words: DStream[String] = ...

words.foreachRDD { rdd =>

    val sqlContext = SQLContext.getOrCreate(rdd.sparkContext)

    import sqlContext.implicits._

    val wordsDataFrame = rdd.toDF("word")

    wordsDataFrame.registerTempTable("words")

    val wordCountsDataFrame = sqlContext.sql("select word, count(*) as total from words group by word")

    wordCountsDataFrame.show()

}
```

DataFrame 저장

```
df.select("name", "age").write().format("parquet").save("namesAndAges.parquet");

df.select("name", "age").write().format("parquet").mode(SaveMode.ErrorIfExists).save("namesAndAges.parquet");
```

Save Mode	의미
SaveMode.ErrorIfExists(Default)	데이터가가 존재하면 Exception 발생
SaveMode.Append	존재하는 데이터 뒤에 추가
SaveMode.Overwirte	기존데이터가 지워지고 새로운 데이터로 쓰여짐
SaveMode.Ignore	기존 데이터를 그대로 두고 새로운 데이터는 쓰여지지 않음

Performance Tuning

- Cache
  - spark.cacheTable("records") / dataFrame.cache()
  - spark.uncacheTable("records")
- 성능 관련 설정

속성	기본 값	의미
spark.sql.inMemoryColumnarStorage.compressed	true	데이터 통계에 근거하여 각 행의 압축 코덱을 선택할지 여부
spark.sql.inMemoryColumnarStorage.batchSize	10000	Columnar caching 배치 크기 조정. 큰 배치는 메모리 사용성과 압축을 높이지만 데이터를 가져올 때 메모리 부족을 일으킬 수 있다.
spark.sql.files.maxPartitionBytes	128 MB	파일을 읽을 때 하나의 파티션에 들어가는 최대 파일 크기
spark.sql.files.openCostInBytes	4 MB	여러 개의 파일이 하나의 파티션에 저장된 경우, 이 파일을 읽어 들일때 한번에 얼마나 많이 스캔해야 하는지에 대한 추정 값
spark.sql.autoBrodcastJoinThreshold	10MB	Join을 실행 할 때 모든 워커 노드에 전달될 테이블의 최대 크기 설정. -1로 설정하면 전달되지 않음. 현재 통계 정보는 "ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan" 명령이 실행되는 Hive metastore 테이블에서만 지원
spark.sql.shuffle.partitions	200	Join이나 aggregation 시 shuffling에 사용할 파티션 개수

Data Type

Spark SQL	Scala	API
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	scala.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(elementType, [containsNull])
MapType	scala.collection.Map	MapType(keyType, valueType, [containsNull])
StructType	org.apache.spark.sql.Row	StructType(fields)
StructField	해당 Filed 값의 타입	StructField(name, datatype, nullable)

Apache Spark Application 실행

Build, Package – Maven

```
<project>
  <groupId>com.raonbit</groupId>
  <artifactId>edu-project</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Edu Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <properties>
    <spark.version>1.2.1</spark.version>
  </properties>
  <repositories>
    <repository>
      <id>Akka repository</id>
      <url>http://repo.akka.io/releases</url
>
    </repository>
  </repositories>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>${spark.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Spark dependency -->
      ……
    </dependency>
  </dependencies>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <filters>
            <filter>
              <artifact>*:*</artifact>
              <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
              </excludes>
            </filter>
          </filters>
          <finalName>${project.artifactId}-uber-${project.version}</final
Name>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Build, Package – SBT

- assembly plugin 설정
  - $USER_HOME/.sbt/0.13/plugins/build.sbt 없으면 생성, 있으면 아래 줄 추가
  - addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")

```
name := "edu-project"

version := "1.0.0"

scalaVersion := "2.10.4"

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.2.1" % "provided",
  "org.apache.spark" % "spark-streaming_2.10" % "1.2.1"  % "provided",
  "org.apache.spark" % "spark-sql_2.10" % "1.2.1"  % "provided",
  "org.apache.spark" % "spark-hive_2.10" % "1.2.1"  % "provided",
  "org.apache.spark" % "spark-streaming-twitter_2.10" % "1.2.1"
)
```

Deploy

Usage: spark-submit [options] <app jar | python file> [app options]

Options:

--master MASTER_URL	spark://host:port, mesos://host:port, yarn, local, local[*]
--deploy-mode DEPLOY_MODE	local(기본 값), cluster
--class CLASS_NAME	application's main class (Java / Scala apps).
--name NAME	application 이름
--jars JARS	driver에 포함될 “,”로 구분된 jar 파일 목록과 executor classpath
--py-files PY_FILES	“,”로 구분된.zip, .egg, .py 파일 목록(PYTHONPATH 상의)
--files FILES	각 executor 의 실행 디렉토리에 놓여질 파일 목록(“,”로 구분)
--conf PROP=VALUE	Spark 속성 설정
--properties-file FILE	속성 파일 지정, 기본값 : conf/spark-defaults.conf.
--driver-memory MEM	driver 메모리 (예. 1000M, 2G) (기본 값: 512M).
--driver-java-options --driver-library-path --driver-class-path	
--executor-memory MEM	각 executor 메모리(예. 1000M, 2G) (기본 값: 1G).
### Spark standalone with cluster deploy mode only:	
--driver-cores NUM	Cores for driver (Default: 1).
--supervise	지정하면, 드라이버가 실패했을 때 다시 시작
### Spark standalone and Mesos only:	
--total-executor-cores NUM	모든 executor의 총 core 개수
### YARN-only:	
--executor-cores NUM	각 executor의 core 개수 (기본값: 1).
--queue QUEUE_NAME	사용할 queue 이름 (기본값: ″ default ″).
--num-executors NUM	실행할 executors 개수 (기본 값: 2).
--archives ARCHIVES	C각 executor의 실행 디렉토리에 풀려질 압출 파일 리스트(“,”로 구분)

Deploy(launcher)

```
package com.raonbit.edu;

import org.apache.spark.launcher.SparkLauncher;

public class MyLauncher {
  public static void main(String[] args) throws Exception {
    Process spark = new SparkLauncher()
      .setAppResource("/my/app.jar")
      .setMainClass("my.spark.app.Main")
      .setMaster("local")
      .setConf(SparkLauncher.DRIVER_MEMORY, "2g")
      .launch();
    spark.waitFor();
  }
}
```

Deploy Strategy

❑ Client Mode : Application을 배포하려는 곳이(spark-submit 을 실행하려는 곳)이 work node와 물리적으로 같은 곳에 있다면(예를 들면, Master node in a standalone EC2 cluster) “client” 모드 사용이 적절

- client mode : driver 가 클러스터의 클라이언트처럼 동작하는 spark-submit process 안에서 직접 탑재 (launch)

❑ Cluster Mode : worker node로 부터 멀리 떨어진 곳에서 application을 배포 한다면(spark-submit 실행하는 곳이 로컬 네트워크 밖이라면) “cluster” 모드 사용이 적절

- cluster mode : driver 가 클러스터 내부에 탑재(launch)
- driver와 executor 사이의 네트워크 지연 최소화
- cluster 모드는 현재 standalone cluster, mesos cluster, python application에서는 사용 불가

Standalone Cluster

- Cluster 시작 / 중지
  - sbin/start-master.sh – 머신의 마스터 인스턴스를 시작
  - sbin/start-slaves.sh – conf/slaves 파일에 정의된 각 머신에 슬레이브 인스턴스 시작.
  - sbin/start-all.sh – 마스터와 슬레이브들 시작
  - sbin/stop-master.sh – bin/start-master.sh 스크립트를 통해 시작된 마스터 중지
  - sbin/stop-slaves.sh – conf/slaves 파일에 정의된 각 머신의 모든 슬레이브 인스턴스 중지
  - sbin/stop-all.sh – 마스터와 슬레이브들 중지
- 환경 변수 설정
  - conf/spark-env.sh : conf/spark-env.sh.template을 복사해서 생성
- standalone cluster에서 spark-shell 실행
  - bin/spark-shell –master spark://HOST:PORT
- application 종료 시키기
  - bin/spark-class org.apache.spark.deploy.Client kill <master url> <driver ID>
  - driver id는 standalone master web UI에서 확인 가능(http://master_url:web_ui_port)

Standalone Cluster – High Availability

❑ ZooKeeper를 이용항 Master 노드 이중화 : spark-env.sh 의 SPARK_DEAMON_JAVA_OPTS 설정

속성	기본값	의미
spark.deploy.recoverMode	NONE	ZOOKEEPER로 설정하면 Master노드 이중화(대기 Master노드)
spark.deploy.zookeeper.url		ZooKeeper cluster url
spark.deploy.zookeeper.dir	/spark	복구 상태를 저장하는 zookeeper내의 디렉토리

- spark master url : spark://host1.port1,host2:port2

❑ Local File System을 이용한 단순 Master 노드 재시작 : spark-env.sh 의 SPARK_DEAMON_JAVA_OPTS 설정

속성	의미
spark.deploy.recoverMode	FILESYSTEM 으로 설정하면 재시작 모드
spark.deploy.recoveryDirectory	복구 상태를 저장하는 디렉토리(Master 노드가 접근 가능한 디렉토리)

- stop-master.sh 로 master 노드를 중지 시키면 복구 상태가 제거 되지 않음.
- 복구 디렉토리로 NFS 사용 가능

추천 Hardware 구성

- Storage Systems
  - 가능하면 HDFS와 같은 노드에서 실행
  - HDFS와 같은 로컬 네트워크 영역에 있는 다른 노드에서 실행
  - Hbase와 같은 low-latency 저장소에서는 간섭을 피하기 위해 다른 노드에서 실행
- Local Disks
  - RAID 구성 없이 4~8개 디스크, noatime 으로 mount 권장
  - HDFS를 사용한다면 HDFS와 동일한 디스크 사용
- Memory
  - 8~100GB의 memory에서 잘 동작(전체 메모리의 75%만 Spark의 위해 사용 권장)
- Network
  - 10G 이상 사용 권장
- CPU Core
  - 8 ~ 16개

Spark Configuration

❑ 설정 방법

- SparkConf Object

```
val conf = new SparkConf()
  .setMaster("local[2]")
  .setAppName("CountingSheep")
  .set("spark.executor.memory", "1g")
val sc = new SparkContext(conf)
```

- Java System Properties

```
-D"spark.executor.memory“="1g"
```

- Command Line Argument

```
./bin/spark-submit --name “My app” --master local[4] --conf spark.shuffle.spill=false
--conf “spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
```

- $SPARK_HOME/conf
  - ✓ fairscheduler.xml.template
  - ✓ log4j.properties.template
  - ✓ metrics.properties.template
  - ✓ spark-defaults.conf.template
  - ✓ spark-env.sh.template