스파크 다운로드

<http://spark.apache.org/downloads.html> 접속

2번은 Pre-built Hadoop 2.6

3번은 Direct Download로 선택

4번 옆의 파일링크 클릭.

7-Zip으로 tar압축 해제.

스칼라 셸 열기

bin\spark-shell

드라이버 프로그램 : main 함수를 갖고 있고 클러스터의 분산 데이터세트를 정의하고 그 데이터 세트에 연산 작업을 수행한다.

드라이버 프로그램은 연산 클러스터에 대한 연결을 나타내는 SparkContext 객체를 통해 스파크에 접속한다.

SparkContext 객체를 하나 만들었다면 그것으로 RDD를 만들어 낼 수 있다.

그러고 count()같은 다양한 연산을 수행해 볼 수 있다.

연산을 수행하기 위해 드라이버 프로그램들은 익스큐터라 불리느 다수의 노드(클러스터의 머신 하나)를 관리한다,

단독 애플리케이션에서는 직접 SparkContext 객체를 초기화 한다. (코드 첫줄) -> 코딩한다.. -> sbt 빌드 파일 만들기 -> .빌드하고 실행

RDD 는 외부 데이터 세트를 로드하거나 드라이버 프로그램에서 객체 컬렉션을 분산시키는 두가지 방법으로 만들 수 있다.(textFile() 을 써서 텍스트 파일을 문자열 RDD로 로딩 등)

ex) val input = sc.parallelize(List(1,2,3,4))

RDD는 두가지 타입의 연산 지원.

**트랜스포메이션 –** 존재하는 RDD에서 새로운 RDD를 만들어 냄.( ex. filter() )

**액션 –** RDD를 기초로 결과 값을 계산하며 그 값을 드라이버 프로그램에 되돌려 주거나 외부 스토리지에 저장. ( ex. first() )

* 여유로운 방식 : 새로운 RDD는 처음 액션을 사용하는 시점에 처리.
* RDD들은 액션이 실행 될 때마다 매번 새로 연산
* 여러 액션에서 RDD 하나를 재상용 하고 싶으면 스파크에게 RDD. persist() 를 사용하여 계속 결과를 유지하도록 요청.
* 스파크에 함수 전달하기(p40)

전체 객체 전달 X. 지역 변수에 필요한 필드만 빼내어서 사용

* 많이 쓰이는 트랜스포메이션 과 액션
* 데이터 요소 위주 트랜스포메이션 (p43)

map(), filter() : map은 함수를 받아 RDD의 각 데이터에 적용하고 결과 RDD에 각 데이터의 새 결과 값을 담는다.

flatMap()에서 함수는 단일값을 리턴하는 대신 반복자를 리턴해야한다. 그러면 반복자가 포함된 RDD를 포함된 RDD를 리턴받는 것이 아니라 반복자가 생성하는 데이터들이 담긴 RDD를 받게 된다. flatMap()은 반환받은 반복자들을 “펼처놓는다”고 생각하면 쉽다.

(실제 코드 아님)

tokenize(“coffee panda”) = List(“coffee”, “panda”)

RDD1는 {“coffee panda”, “happy panda”, “happiest panda party”}

rdd1.map(tokenize)

=> mappedRDD는 {[“coffee”,”panda”],[“happy”,”panda”],[“happies”,”panda”,”party”]}

rdd1.flatMap(tokenize)

* flatMappedRDD는 {“coffee”,”panda”,”happy”,”panda”,”happiest”,”panda”,”party”}
* 가상집합연산

RDD1.union(RDD2): 중복 유지 합집합

RDD1.distinct(RDD2) : 중복 제거 합집합

RDD1.intersection(RDD2) : 교집합

RDD1.subtract(RDD2) : 차집합

위에 4개는 RDD가 서로 같은 타입

RDD1.cartesian(RDD2) : 카테시안 곱 {a1,a2,a3} 와 {b1,b2} => (a1,b1),(a1,b2)…6개의 쌍이 나옴.

* 액션(p.52)

reduce() 함수 : 인자로 두 개의 데이터를 합쳐 같은 타입 데이터 하나를 반환.

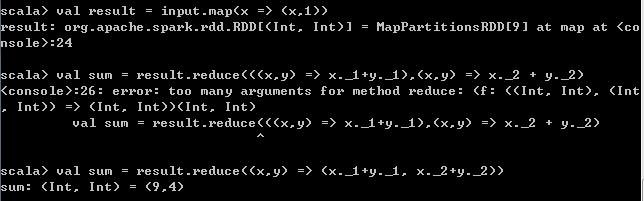
aggregate(zeroValue)(seqOp,combOp) :reduce와 유사하나 다른 타입 리턴

seqOp – RDD의 값들을 누적값에 연계해 주는 함수

combOp – 자체적으로 값들을 합칠 수 있도록 두개의 누적값을 합쳐주는 두번째 함수

collect() : 드라이버 프로그램에 값을 되돌려주는 가장 간단하고 일반적인 연산.

데이터가 모두 드라이버 프로그램으로 카피.



* RDD 타입간 변환하기
* 특정 함수를 갖고 있는 RDD로 변환 하는 것은 묵시적 변환에 의해 자동으로 동작.
* mean()은 수치형 RDD에서만 쓸 수 있다. .
* RDD 에서 mean() 같은 함수를 호출하려면 RDD 클래스의 스칼라 문서를 보고 mean()이 없는지 확인.
* 영속화(캐싱)
* 동일한 RDD를 여러 번 사용하고 싶을 때
* 스파크는 RDD와 RDD에서 호출하는 액션들에 대한 모든 의존성을 재연산.

ex ) val result = input.map(x => x\*x)

println(result.count())

println(result.collect().mkString(“,”))

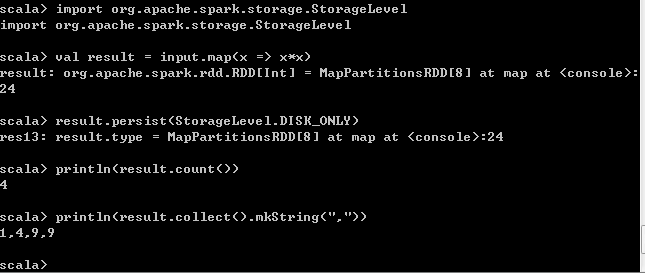
RDD 영속화(persist) 요청을 하면 RDD를 계산한 노드들은 그 파티션들을 저장하고 있게 된다. 영속화 된 데이터를 갖고 있는 노드에 장애가 생기면 스파크는 필요시 유실된 데이터 파티션을 재연산한다.

스파크는 목적에 맞는 여러 수준의 영속화(persist) 제공한다.

스칼라에서는 기본적으로 persist()가 데이터를 JVM heap에 직렬화되지 않은 객체 형태로 저장한다. 데이터를 디스크나 off-heap 저장 공간에 쓸 때는 데이터가 늘 직렬화 된다.

org.apache.spark.stroage.StorageLevel

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 레벨 | 공간 사용 | CPU 사용 시간 | 메모리에 저장 | 디스크에 저장 | 비고 |
| MEMORY\_ONLY | 높음 | 낮음 | O | X |  |
| MEMORY\_ONLY\_SER | 낮음 | 높음 | O | X |  |
| MEMORY\_AND\_DISK | 높음 | 중간 | 일부 | 일부 | 메모리에 넣기에 데이터가 너무 많으면 디스크에 나눠 저장 |
| MEMORY\_AND\_DISK\_SER | 낮음 | 높음 | 일부 | 일부 | 메모리에 넣기에 데이터가 너무 많으면 디스크에 나눠 저장 .메모리에 직렬화된 형태로 저장. |
| DISK\_ONLY | 낮음 | 높음 | X | O |  |

ex) 

메모리에 데이터가 차면 LRU(Least Recently Used) 캐시정책에 따라 오래된 파티션들을 자동으로 버린다. 디스크와 메모리 같이 쓰는 수준은 버린 것들을 디스크에 쓰지만, 메모리만 쓰는 정책에서는 다음에 파티션에 접근할 때 다시 계산한다.

**4. 키/값 페어로 작업하기**

* 키/값 페어 RDD로 어떻게 작업하는지 설명.
* **파티셔닝** : 전체 노드 위에서 페어 RDD의 구조를 사용자가 제어할 수 있는 새로운 고급 기능
* 제어 가능한 파티셔닝을 활용하면 데이터가 함께 액세스 되는지 동일 노드에 있는지 판단이 가능하므로 노드 간 통신 비용을 획기적으로 줄일 수 있다. 즉, 처리 속도가 매우 빨라질 수 있다. 파티셔닝 방법은 여러가지가 있고 분산 데이터 세트에서 올바른 파티셔닝을 사용하는 것은 마치 로컬 애플리케이션에서 올바른 자료구조를 선택하는 것과 비슷하다.
* 스파크는 페어 RDD에 대해 특수한 연산들을 제공한다.
* 페어 RDD 생성
* 키를 가지는 데이터를 위한 함수들을 위해 튜플을 리턴.

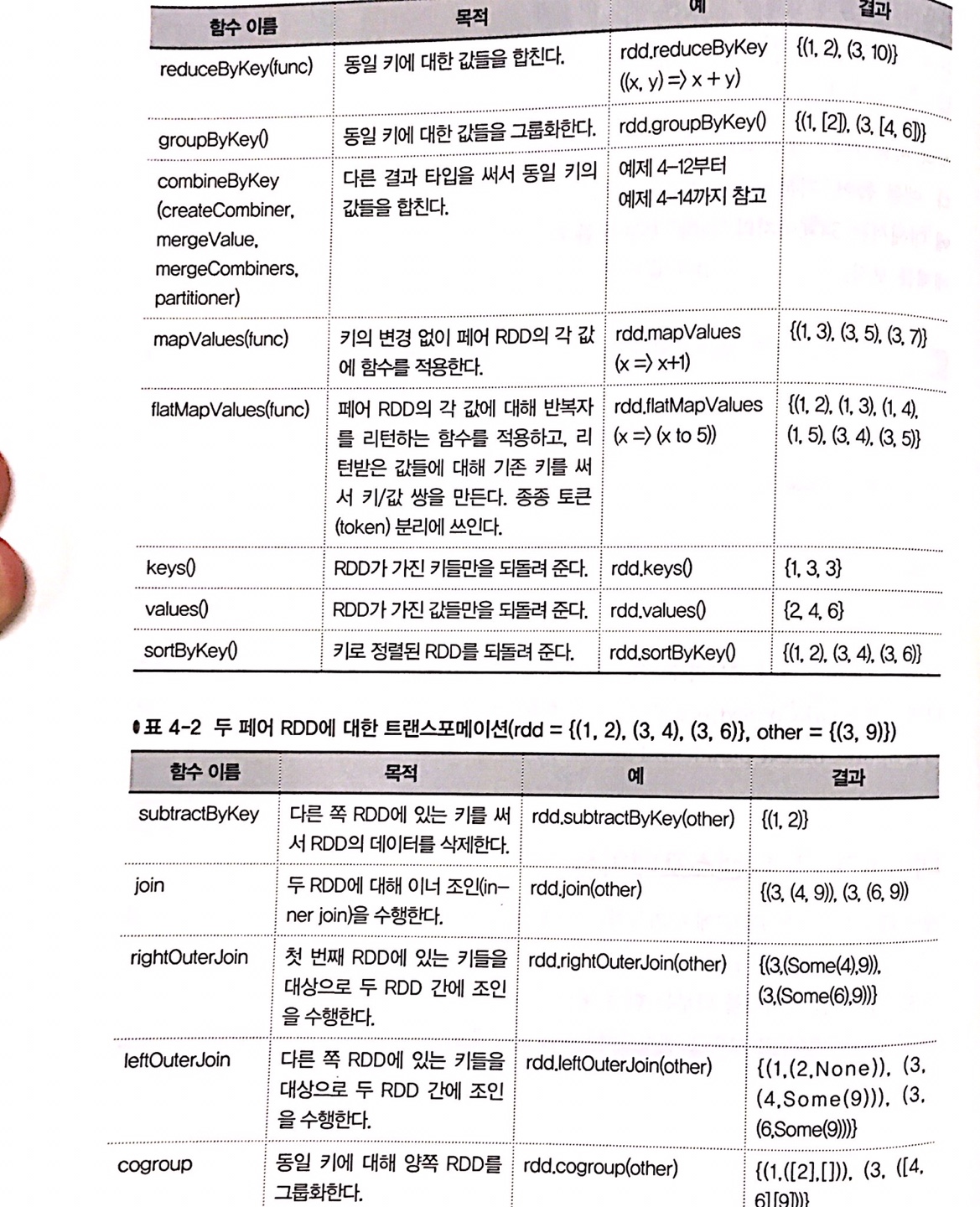
val pairs = lines.map( x => (x.split(“ “)(0) , x))

: 첫 번째 단어를 키로 사용한 페어 RDD 생성.

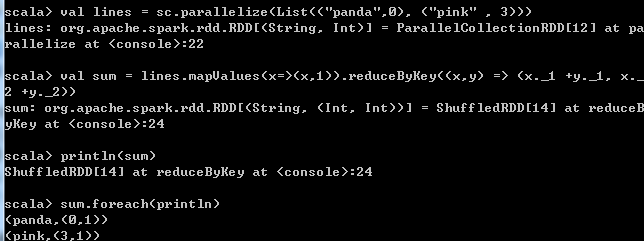
* 메모리에 있는 데이터세트에 parallelize() 호출
* 페어 RDD 트랜스포메이션
* 페어 RDD는 기본 RDD에서 가능한 모든 트포 사용 가능.
* p.62의 표

ex) pairs.filter{case (key, value) => value. length<20}

* 스파크는 map{case ( x , y ) : ( x , func( y ) ) }과 동일한 표현으로 쓸수 있도록 mapValues(func) 함수를 제공한다.



* 집합 연산
* fold(), aggregate(), reduce() 같은 함수들과 유사한 각 키별 트랜스포메이션 존재
* reduceByKey()는 reduce와 유사



* 단어 세기(p66)

textFile함수로 txt를 불러와서 flatMap후에 map으로 (value,1) 형태로 바꾼 뒤 reduceByKey으로 단어의 합을 구한다.

input.flatMap(x => x.split(“ “)).countByValue() 하면 더 빠르게 단어를 셀 수 있다.

* combineByKey()

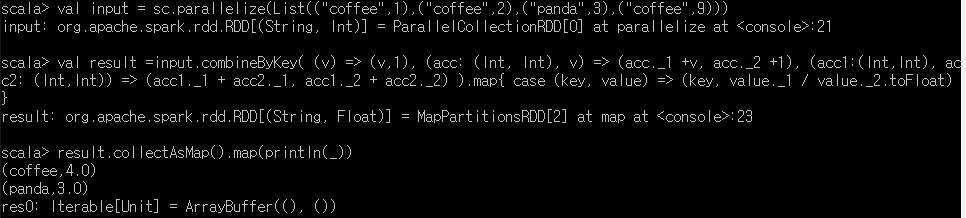
키별 집합 연산 함수 중 가장 일반적으로 쓰인다.

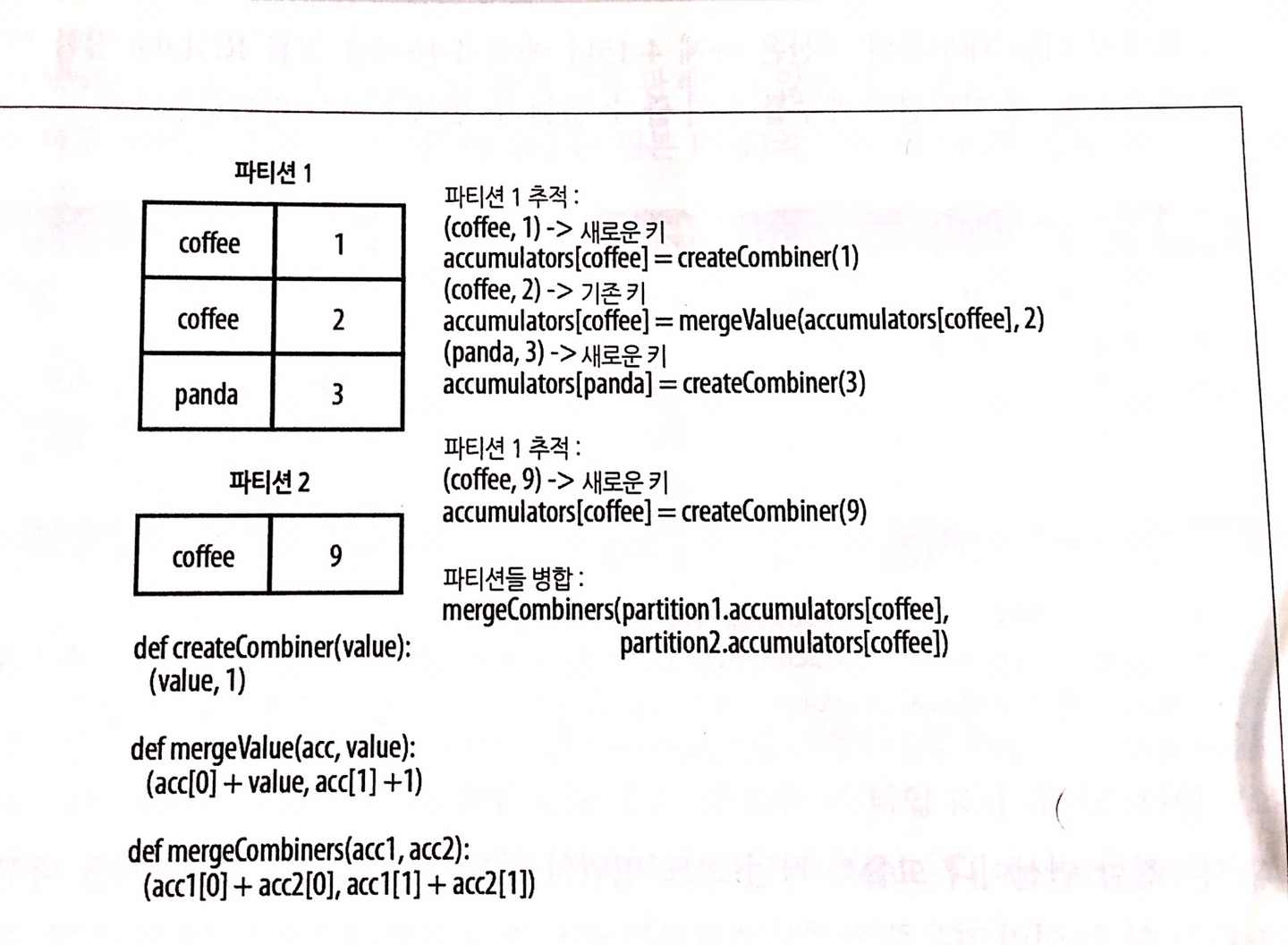
combineByKey는 한 파티션 내의 데이터들을 하나씩 처리하게 되며, 각 데이터는 이전에 나온 적이 없는 키를 갖고 있을 수도 있고 이전 데이터와 같은 키를 가질 수도 있다.

만약 새로운 데이터라면 combineByKey()는 우리가 넘겨준 createCombiner()함수를 써서 해당 키에 대한 어큐뮬레이터의 초깃값을 만든다. 중요한 점은 이 작업을 RDD 전체 기준으로 첫 키가 나올 때가 아니라 각 파티션에서 처음 나오는 키마다 실행한다는 것이다.

그리고 파티션을 처리하는 도중 출현한 적이 있는 값이라면 대신 mergeValue() 함수를 해당 키에 대한 어큐뮬레이터의 현재 값과 새로운 값에 적용해서 합친다.

각 파티션에서 독립적으로 작업이 이루어지므로 동일 키에 대해 여러 개의 어큐뮬레이터를 가질 수도 있게 된다. 각 파티션으로부터 결과를 최종적으로 합칠 때 둘 이상의 파티션이 동일 키에 대한 어큐뮬레이터를 갖고 있다면 이 어큐뮬레이터들은 마찬가지로 사용자가 제공한 함수인 mergeCombiners()를 써서 합쳐지게 된다.





* 스파크에서 제공하는 특화된 집합 연산 함수들을 사용하는 것이 데이터를 그룹화하고 병합 처리하는 원초적인 방법보다는 훨씬 빠르다.
* 병렬화 수준 최적화
* 대개의 경우, 스파크는 클러스터의 사이즈에 맞는 적절한 파티션 개수를 찾는 방식으로 동작.
* 더 나은 퍼포먼스를 내기 위해 병렬화의 수준을 직접 정해 주어야 할 경우도 있다.
* 두번째 인자로 병렬화 수준 지정한다.

Ex) data.reduceByKey( (x,y) => x+y , 10)

* 데이터 그룹화
* groupByKey(), groupBy(), cogroup()
* 조인
* 내부 조인. 양쪽 RDD에 모두 존재하는 키만이 결과가 된다.
* 키가 양쪽 RDD에 모두 존재하지 않는 경우. leftOuterJoin(), reghtOuterJoin()
* 데이터 정렬

sortByKey()

* 페어 RDD 액션
* countByKey() : 각 키에 대한 값의 개수를 센다
* collectAsMap() : 쉬운 검색을 위해 결과를 맵 형태로 모은다.
* lookup(key) : 들어온 키에 대한 모든 값을 되돌려 준다.
* 데이터 파티셔닝
* RDD 하나가 100개의 파티션으로 나뉘도록 해시 파티셔닝을 선택했다면 100으로 나눈 나머지에 대해 동일 해시 값을 갖는 키들은 동일한 노드에 오게 된다.
* 같은 범위의 키들이 같은 노드에 모이도록 RDD룰 범위별 파티셔닝을 수행할 수도 있다.

ex) val userData = sc.sequenceFile[UserID, UserInfo](“hdfs://..)

.partitionBy(new HashPartioner(100)).persist()

* 100은 파티션 개수를 나타내는데 일반적으로 이 숫자는 최소한 클러스터의 전체 CPU 코어 개수 이상이 되도록 한다.