스파크로
구축하는 분산처리
빅데이터 플랫폼

1.Spark 기초와 빅데이터 플랫폼의 개념 및 설계

1.2 스파크 API

1. Pair RDD 다루기

Pair RDD

- 키-값 쌍은 간단하고 범용적이고 확장성이 뛰어난 데이터 모델
- 기존 키-값 쌍에 새로운 타입의 키와 값을 손쉽게 추가할 수 있고, 키-값 쌍을 독립적으로 저장할 수 있음
- 확장성과 간 결성 덕분에 키-값 쌍 모델은 여러 프레임워크와 애플리케이션의 기본 요소
- 키-값 쌍의 키와 값에는 정수형이나 문자열 등 기본 타입도 사용할 수 있고, 복잡한 데이터 구조 체도 사용

1. Pair RDD 다루기

Pair RDD

- 키-값 쌍은 전통적으로 연관 배열(associative array)이라는 자료 구조를 사용해 표현 (파이썬에서는 딕션너리(dictionary), 스칼라와 자바에서는 맵(map))
- 스파크에서는 키-값 쌍(또는 키-값 튜플)으로 구성된 RDD를 Pair RDD
- Pair RDD를 사용하면 데이터를 편리하게 집계, 정렬 , 조인 할 수 있음

1. Pair RDD 다루기

Pair RDD 생성

- 스파크에서는 다양한: 방법으로 Pair RDD를 생성 할 수 있음
- SparkContext의 일부 메서드는 Pair RDD를 기본으로 반환
- RDD 의 keyBy 변환 연산자는 RDD 요소로 키를 생성하는 f 함수를 받고，각 요소를 (f (요소), 요소) 쌍의 튜플로 매핑
- Pair RDD 함수는 PairRDD Functions 클래스에 정의

1. Pair RDD 다루기

기본 Pair RDD 함수

- "한 쇼핑 사이트의 마케팅 부서에서 고객에게 선별적으로 사은품을 보내는 행사를 기획"
- "어제 날짜의 구매 기록을 읽어 들여 특정 규칙에 따라 사은품을 추가하는 프로그램을 개발"
  - 구매 횟수가 가장 많은 고객에게는 곰 인형을 보낸다.
  - 바비 쇼핑몰 놀이 세트를 두 개 이상 구매하면 청구 금액을 5% 할인해 준다.
  - 사전을 다섯 권 이상 구매한 고객에게는 칫솔을 보낸다.
  - 가장 많은 금액을 지출한 고객 에게는 커플 잠옷 세트를 보낸다.
- "사은품은 구매 금액이 0.00달러인 추가 거래로 기입해야 하며, 사은품을 받는 고객이 어떤 상품을 구매했는지 알려 달라고 요청"

1. Pair RDD 다루기

기본 Pair RDD 함수 - 생성

- 파일의 각 줄에는 구매 날짜, 시간，고객 id, 상품 id, 구매 수량, 구매 금액이 순서대로 기록되어 있으며 각 항목은 기호(#)로 구분
- Pair RDD의 키는 고객 ID이며，값에는 각 구매 로그(예제 파 일의 각 줄)의 모든 정보가 저장
- scala> val tranFile = sc. textFile(“data_transactions.txt")
- scala>val tranData = tranFile.map(_. split (“#”))
- scala> var transByCust = tranData.map(tran => (tran(2).toInt, tran))

1. Pair RDD 다루기

기본 Pair RDD 함수 – 키 및 값 가져오기

- Pair RDD의 키 또는 값으로 구성된 새로운 RDD를 가져오려면 keys 또는 values 변환 연산자를 사용
- scala> transByCust.keys.distinct().count()
- keys 변환 연산자가 반환한 RDD는 일부 고객 ID가 중복된 구매 기록 1000개로 구성
- distinct() 변환 연산자로 중복을 제거한 후 구매 고객의 인원수를 계산해 100명이라는 결과

1. Pair RDD 다루기

기본 Pair RDD 함수 – 키별 개수 세기

- 데이터 파일에서 한 줄은 구매 한 건을 의미하며 각 고객별로 줄 개수를 세면 그것이 바로 각 고객의 구매 횟수
- RDD의 countByKey 행동 연산자를 사용
- RDD의 행동 연산자는 변환 연산자와 달리 연산 결과를 즉시 자바(또는 스칼라나 파이썬) 객체로 반환
- countByKey는 다음과 같이 각 키의 출현 횟수를 스칼라 Map 형태로 반환
- scala> transByCust.countByKey()
- scala> transByCust.countByKey().values.sum

1. Pair RDD 다루기

기본 Pair RDD 함수 – 키별 개수 세기

- 구매 횟수가 가장 많았던 고객을 찾기
- scala> val (cid, purch) = transByCust.countByKeyO.toSeq.sortBy(_._2).last
- complTrans 변수를 만들고, 사은품(곰 인형(상품 ID 4번))의 추가 구매 기록을 String 배열 형태로 저장
- scala> var complTrans = Array(Array("2015-03-30", "11:59 PM", "53", "4", "1", "0.00"))

1. Pair RDD 다루기

기본 Pair RDD 함수 – 단일 키로 값 찾기

- 구매 횟수가 가장 많았던 고객을 찾기
- “사은품을 받을 고객이 정확하게 어떤 상품을 구매했는지도 알려 달라고 요청”
- Lookup() 행동 연산자를 사용해 고객(id 53번)의 모든 구매 기록을 가져옴
- scala> transByCust.lookup(53)
- WrappedArray 클래스는 암시적 변환을 이용해 Array를 Seq(가변 컬렉션)로 표현
- scala> transByCust.lookup(53).foreach(tran => println(tran.mkString(", ")))

1. Pair RDD 다루기

기본 Pair RDD 함수 – mapValues 변환연산자

- "바비 쇼핑몰 놀이 세트를 두 개 이상 구매하면 청구 금액을 5% 할인해 주는 것"
- Pair RDD의 mapValues 변환 연산자를 사용하면 키를 변경하지 않고 Pair RDD에 포함된 값 만 변경할 수 있음
- scala> transByCust = tansByCust.mapValues(tran => { if(tran(3).tolnt == 25 && tran(4).toDouble > 1)tran(5) = (tran(5).toDouble * 0.95).toString tran })
- mapValues가 반환하는 새로운 RDD 를 동일 변수(transByCust)에 다시 할당

1. Pair RDD 다루기

기본 Pair RDD 함수 – flatMapValues 변환연산자

- “사전(상품 ID 81 번)을 다섯 권 이상 구매한 고객에게 사은품으로 칫솔(상품 ID 70번)을 보내야함”
- transByCust RDD에 배열 형태의 구매 기록을 추가
- flatMapValues 변환 연산자는 각 키 값을 0개 또는 한 개 이상 값으로 매핑해 RDD에 포함된 요소 개수를 변경할 수 있음
- flatMapValues는 변환 함수가 반환한 컬렉션 값들을 원래 키와 합쳐 새로운 키-값 쌍으로 생성

1. Pair RDD 다루기

기본 Pair RDD 함수 – flatMapValues 변환연산자

- 변환 함수가 인수로 받은 값의 결과로 빈 컬렉션을 반환하면 해당 키-값 쌍을 결과 Pair RDD 에서 제외
- 컬렉션에 두 개 이상의 값을 넣어 반환하면 결과 Pair RDD에 이 값들을 추가

```
scala> transByCust = transByCust.flatMapValues(tran => {
    if(tran(3).toInt == 81 && tran(4).toDouble >= 5) { ····· 구매 제품과 수량으로 필터링한다.
        val cloned = tran.clone() ····· 구매 기록 배열을 복제한다.
        cloned(5) = "0.00"; cloned(3) = "70"; cloned(4) = "1"; ·····
        List(tran, cloned) ····· 원래 요소와 추가한 요소를 반환한다.
    }
    else
        List(tran) ····· 원래 요소만 반환한다.
    })
```

복제한 배열에서 가격은 0.00달러로 수정하고,
제품 ID는 70으로 수정하며, 수량은 1로 수정한다.

1. Pair RDD 다루기

기본 Pair RDD 함수 – reduceByKey 변환연산자

- reduceByKey는 각 키의 모든 값을 동일한 타입의 단일 값으로 병합
- 두 값을 하나로 병합하는 merge 함수를 전달해야 하며, reduceByKey는 각 키 별로 값 하나만 남을 때까지 merge 함수를 계속 호출
- merge 함수는 결합 법칙을 만족(associative)해야 함.
- 만족하지 않으면 같은 RDD라도 reduceByKey를 호출할 때마다 결과 값이 달라짐

1. Pair RDD 다루기

기본 Pair RDD 함수 – folderByKey 변환연산자

- “가장 많은 금액을 지출한 고객을 찾는 것”
- foldByKey는 reduceByKey와 기능은 같지만, merge 함수의 인자 목록 바로 앞에 zeroValue 인자를 담은 또 다른 인자 목록을 추가로 전달해야 한다는 점 다름
- zeroValue는 반드시 항등원(neutral value 또는 identity element)이어야 함
- zeroValue는 가장 먼저 func 함 수로 전달해 키의 첫 번째 값과 병합하며, 이 결과를 다시 키의 두 번째 값과 병합

1. Pair RDD 다루기

기본 Pair RDD 함수 – folderByKey 변환연산자

- 객별로 구매 금액을 합산해야 하지만, transByCust에 저장된 구매 기록은 문자열 배열 이기 때문에 그대로 사용할 수 없음
- 튜플에 금액 정보만 포함하도록 데이터를 매핑한 후 foldByKey를 사용
- 합계 금액을 기준으로 결과 Array를 오름차순으로 정렬하고 가장 큰 마지막 요소를 추출
- scala> val amounts = transByCust.mapValues(t => t(5).toDouble)
- seala〉 val totals = amounts.foldByKey(0)((p1, p2) => p1 + p2).collect()
- scala> totals.toSeq.sortBy(_._2). last

1. Pair RDD 다루기

기본 Pair RDD 함수 – aggregateByKey 변환연산자

- aggregateByKey는 zeroValue를 받아 RDD 값을 병합한다는 점에서 foldByKey나 reduceByKey와 유사하지만 값 타입을 바꿀 수 있다는 점은 다름
- aggregateByKey를 호출하려면 zeroValue와 다음 두 가지 함수를 인수로 전달
- 첫번째는 임의의 V 타입을 가진 값을 또 다른 U 타 입으로 변환{(U, V) => U]하는 변환 함수
- 두번째는 첫 번째 함수가 변환한 값을 두 개씩 하나 로 병합{(U, U) => U]하는 병합 함수
- aggregateByKey 연산자에 zeroValue 인수만 넣어 호출하면(즉, 첫 번째 괄호의 인수만 전달하면) 변환 함수와 병합 함수를 인수로 받는 새로운 함수가 반환 됨

1. Pair RDD 다루기

기본 Pair RDD 함수 – aggregateByKey 변환연산자

- scala> val prods = transByCust.aggregateByKey(List[String]())(

  (prods) tran) => prods ::: List(tran(3)),

  (prods1, prods2) => prods1::: prods2)

- scata> prods.collect()

2. 데이터 파티셔닝과 데이터 셔플링

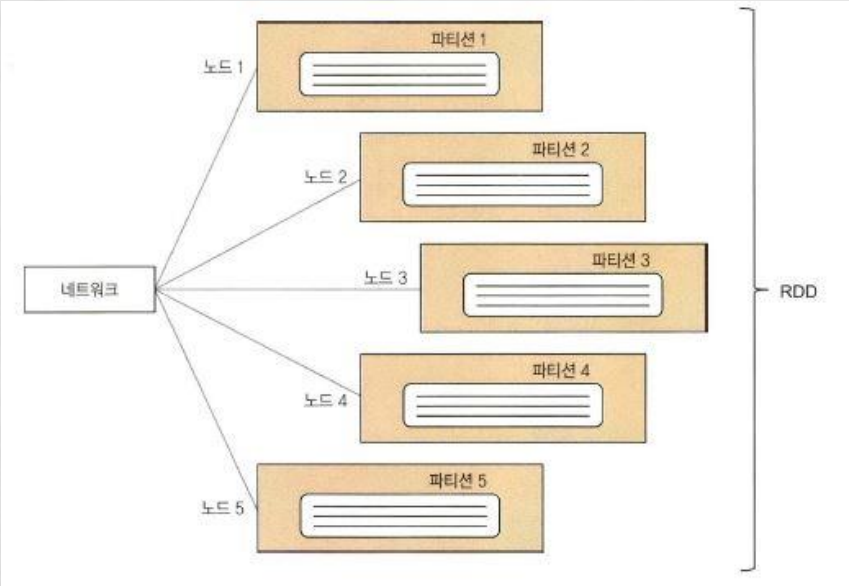
데이터 파티셔닝

- 스파크의 데이터 파티셔닝은 데이터를 여러 클러스터 노드로 분할하는 메커니즘을 의미
- 데이터 파티셔닝은 스파크의 성능과 리소스 점유량을 크게 좌우할 수 있는 RDD의 가장 기본적인 개념
- RDD의 파티션은 RDD 데이터의 일부(조각 또는 슬라이스)를 의미
- 로컬 파일 시스템에 저장된 텍스트 파일을 스파크에 로드하면, 스파크는 파일 내용을 여러 파티션으로 분할해 클러스터 노드에 고르게 분산 저장

2. 데이터 파티셔닝과 데이터 셔플링

데이터 파티셔닝

- 여러 파티션을 노드 하나에 저장할 수도 있음
- 분산된 파티션이 모여서 RDD 하나를 형성
- 스파크는 RDD별로 RDD의 파티션 목록을 보관하며, 각 파티션의 데이터를 처리할 최적 위치를 추 가로 저장할 수 있음



2. 데이터 파티셔닝과 데이터 셔플링

데이터 파티셔닝

- 스파크에서는 RDD의 파티션 개수가 매우 중요
- 파티션 개수가 데이터를 클러스터에 분배하는 과정에서 영향을 미치기도 하지만, 해당 RDD에 변환 연산을 실행할 태스크 개수와 직결되기 때문
- 태스크 개수가 필요 이하로 적으면 클러스터를 충분히 활용할 수 없음
- 각 태스크가 처리할 데이터 분량이 실행자의 메모리 리소스를 초과해 메모리 문제가 발생할 수 있음
- 클러스터의 코어 개수보다 서너 배 더 많은 파티션을 사용하는 것이 좋음
- 태스크가 너 무 많으면 태스크 관리 작업에 병목 현상이 발생

2. 데이터 파티셔닝과 데이터 셔플링

RRD의 데이터 파티셔닝

- RDD의 각 요소에 파티션 번호를 할당하는 Partitioner 객체가 수행
- 스파크는 Partitioner의 구현체(impleinentation)로 HashPartitioner와 RangePartitioner를 제공
- 사용자 정의 Partitioner를 Pair RDD에 사용할 수 있음
- HashPartitioner는 스파크의 기본 Partitioner이며 각 요소의 자바 해시 (hash) 코드(Pair RDD는 키의 해시 코드)를 단순한 mod 공식(partitionIndex = hashCode % numberOfPartitions)에 대입해 파티션 번호를 계산함

2. 데이터 파티셔닝과 데이터 셔플링

RRD의 데이터 파티셔닝

- 요소의 파티션 번호를 거의 무작위로 결정하기 때문에 모든 파티션을 정확하게 같은 크기로 분할할 가능성이 낮음
- RangePartitioner는 정렬된 RDD의 데이터를 거의 같은 범위 간격으로 분할할 수 있음
- RangePartitioner는 대상 RDD에서 샘플링한 데이터를 기반으로 범위 경계를 결정함
- 파티션(또는 파티션을 처리하는 태스크)의 데이터를 특정 기준에 따라 정확하게 배치해야 할 경우 사용자 정의 Partitioner로 Pair RDD를 분할할 수 있음

2. 데이터 파티셔닝과 데이터 셔플링

RRD의 데이터 파티셔닝

- 대부분의 Pair RDD 변환 연산자는 두 가지 추가 버전을 제공
- 첫 번째 버전은 Int 인수(변경할 파티션 개수)를 추가로 받으며, 두 번째 버전은 사용할 Partitioner(스파크 지원 Partitioner 또는 사용자 정의 Partitioner)를 추가 인수로 받음
- 파티션 개수를 받는 메서드를 호출하면 기본 Partitioner인 HashPartitioner를 사용
- rdd.foldByKey(afunction, 100)
- rdd.foldByKey(afunction, new HashPartitioner(100))

2. 데이터 파티셔닝과 데이터 셔플링

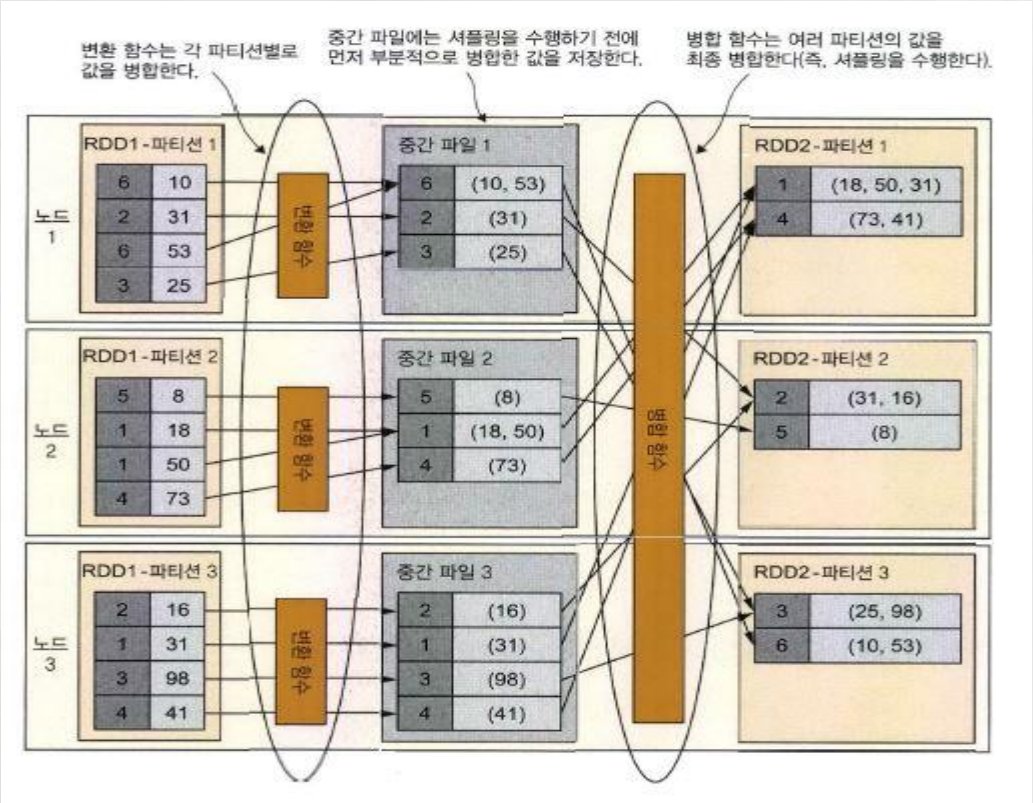
RRD의 데이터 파티셔닝

- Pair RDD 변환 연산자를 호출할 때 Partitioner를 따로 지정하지 않으면 스파크는 부모 RDD (현재 RDD를 만드는 데 사용한 RDD들)에 지정된 파티션 개수 중 가장 큰 수를 사용함
- Partitioner를 정의한 부모 RDD가 없다면 spark.default.paraUeUsm 매개변수에 지정된 파티션 개수로 HashPartitioner를 사용함
- 기본 HashPartitioner를 그대로 사용하면서 임의의 알고리즘으로 키의 해시 코드만 바꾸어도 Pair RDD 데이터의 파티션 배치를 변경할 수 있음

2. 데이터 파티셔닝과 데이터 셔플링

불필요한 셔플링 줄이기

- 스파크의 셔플링은 파티션 간의 물리적인 데이터 이동을 의미
- 셔플링은 새로운 RDD의 파티션을 만들려고 여러 파티션의 데이터를 합칠 때 발생



2. 데이터 파티셔닝과 데이터 셔플링

불필요한 셔플링 줄이기

- 사용자 정의 Partitioner를 쓰면 반드시 셔플링이 발생
- 이전 HashPartitioner와 다른 HashPartitioner를 사용해도 셔플링이 발생
- 이전에 사용한 HashPartitioner와 파티션 개수가 다른 HashPartitioner를 변환 연산자에 사용 하면 셔플링이 발생
- 셔플링을 수행하면 실행자는 다른 실행자의 파일을 읽어 들여야 함(스파크의 셔플링은 풀링 (pulling) 방식을 사용)
- 셔플링 도중 일부 실행자에 장애가 발생하면 해당 실행자가 처리한 데이터를 더 이상 가져올 수 없어서 데이터 흐름이 중단됨

2. 데이터 파티셔닝과 데이터 셔플링

RDD 파티션 변경

- 작업 부하를 효율적으로 분산시키거나 메모리 문제를 방지하려고 RDD의 파티셔닝을 명시적으로 변경해야 할 때가 있음
- 일부 스파크 연산자에는 파티션 개수의 기 본 값이 너무 작게 설정되어 있어 이 값을 그대로 사용하면 파티션에 매우 많은 요소를 할당하고 메모리를 과다하게 점유해 결과적으로 병렬 처리 성능이 저하될 수 있음
- RDD의 파티션을 변경 할 수 있는 변환 연산자에는 partitionBy, coalesce, repartition, repartitionAndSortWithinPartition이 있음

2. 데이터 파티셔닝과 데이터 셔플링

파티션의 데이터를 수집하는 glom변환 연산자

- glom 연산자는 각 파티션의 모든 요소를 배열 하나로 모으고, 이 배열들을 요소로 포함하는 새로운 RDD를 반환함
- 새로운 RDD에 포함된 요소 개수는 이 RDD의 파 티션 개수와 동일하다. glom 연산자는 기존의 Partitioner를 제거함
- scala> val list = List.fill(500)(scala.util.Random.nextInt(100))
- scala> val rdd = sc.parallelize(list, 30).glom()
- scala> rdd.collect()
- scala> rdd.count()

3. 데이터 조인, 정렬, 그루핑

데이터 조인,정렬, 그루핑

- “어제 판매한 상품 이름과 각 상품별 매출액 합계(알파벳 오름차순으로 정렬할 것)”
- “어제 판매하지 않은 상품 목록”
- “전일 판매 실적 통계: 각 고객이 구입한 상품의 평균 가격, 최저 가격 및 최고 가격, 구매 금액 합계”

3. 데이터 조인, 정렬, 그루핑

데이터 조인

- 어제 판매한 상품 이름과 각 상품별 매출액 합계
- RDD를 다음과 같이 매핑해 상품 ID를 키로 설정한 Pair RDD를 생성
- scala> val transByProd = tranData.map(tran => (tran(3).toInt, tran))
- reduceByKey 변환 연산자를 사용해 각 상품의 매출액 합계를 계산
- val totalsByProd = transByProd.mapValues(t => t(5).toDouble).

  reduceByKey{case(tot1, tot2) => tot1 + tot2}
- 상품 이름 정보는 별도 파일에 저장(ch04_data_products.txt 파일)

3. 데이터 조인, 정렬, 그루핑

데이터 조인

- 어제 판매한 상품 이름을 알아내려면 이 파일과 전일 구매 기록을 조인해야 함
- 상품 이름 파일을 로드하고 Pair RDD로 변환
- val products = sc.textFile("first-edition/ch04/ch04_data_products.txt").

  map(line => line.split("#")). map(p => (p(0).toInt, p))
- zip, cartesian, intersection 등 다양한 변환 연산자를 사용해 여러 RDD 내용을 합칠 수 있음

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - 조인연산자

- 스파크의 조인 연산은 PairRDD에서만 사용할 수 있음
- 조인 연산자를 호출하려면 (K, V) 타입의 요소를 가진 RDD에 (K, W) 타입의 요소를 가진 다른 RDD를 전달해야 함
  - join : RDBMS의 내부 조인(inner join)과 동일하며, 첫 번째 Pair RDD[(K, V)]와 두 번째 Pair RDD[(K, W)]에서 키가 동일한 모든 값의 조합이 포함된 Pair RDD[此 (V, W))]를 생성하며 두 RDD 중 어느 한쪽에만 있는 키의 요소는 결과 RDD에서 제외
  - leftOuterJoin : (K, (V, W)) 대신 , (K, (V, Option(W))) 타입의 Pair RDD를 반환한다. 첫 번째 RDD에만 있는 키의 요소는 결과 RDD에 (key, (V, None)) 타입으로 저장되며, 두 번째 RDD에만 있는 키의 요소는 결과 RDD에서 제외

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - 조인연산자

- 조인 연산자
  - rightOuterJoin : leftOuterJoin과는 반대로 (K, (Option(V), W)) 타입의 Pair RDD를 반 환한다. 두 번째 RDD에만 있는 키의 요소는 결과 RDD에 (key, (None, W)) 타입으로 저장되며, 첫 번째 RDD에만 있는 키의 요소는 결과 RDD에서 제외
  - fullOuterJoin : (K, (Option(V), Option(W)) 타입의 Pair RDD를 반환하며 두 RDD 중 어느 한 쪽에만 있는 키의 요소는 결과 RDD에 (key, (v, None)) 또는 (key, (None) w)) 타입으로 저장됨

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - 조인연산자

- 조인할 두 RDD의 요소 중에서 키가 중복된 요소는 여러 번 조인함
- 다른 Pair RDD 변환 연산자와 마찬가지로 조인 연산자에 Partitioner 객체나 파티션 개수를 전달할 수 있음
- 연산자에 파티션 개수만 지정하면 HashPartitioner를 사용함
- scala> val totalsAndProds = totalsByProd.join(products)
- scala> totalsAndProds.first()

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - 조인연산자

- 조인할 두 RDD의 요소 중에서 키가 중복된 요소는 여러 번 조인함
- 다른 Pair RDD 변환 연산자와 마찬가지로 조인 연산자에 Partitioner 객체나 파티션 개수를 전달할 수 있음
- 연산자에 파티션 개수만 지정하면 HashPartitioner를 사용함
- scala> val totalsAndProds = totalsByProd.join(products)
- scala> totalsAndProds.first()

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - 조인연산자

- 각 상품 ID별로 두 (해당 상품의 매출액, 상품 데이터의 문자열 배열) 쌍으로 구성된 2-요소 튜플을 계산
- val totalsWithMissingProds = totalsByProd.rightOuterJoin(products)
- rightOuterJoin의 경우 totalsWithMissingProds는 (Int, (Option [Double], Array[String])) 타입의 요소로 구성
- 매출 데이터에 포함하지 않은 상품은 Option 값으로 None 객체를 전달

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - 조인연산자

- 어제 판매 하지 않은 상품 목록을 얻으려면 None 값을 가진 요소만 남도록 결과 RDD를 필터링한 후 키와 None 객체를 제거하고, 상품 데이터만 가져와야 한다. rightOuterJoin을 사용할 때는 다음과 같이 실행
- val missingProds = totalsWithMissingProds.filter(x => x._2._1 == None).map(x => x._2._2)
- missingProds.foreach(p => println(p.mkString(", ")))

3. 데이터 조인, 정렬, 그루핑

데이터 조인 - subtract나 subtractByKey 변환 연산자로 공통 값 제거

- subtract는 첫 번째 RDD에서 두 번째 RDD의 요소를 제거한 여집합을 반환
- subtractByKey는 PairRDD에서만 사용할 수 있는 메서드
- val missingProds = products.subtractByKey(totalsByProd).values
- missingProds.foreach(p => println(p.mkString(", ")))
- subtract와 subtractByKey 변환 연산자는 파티션 개수를 받는 버전과 Partitioner 객체를 받는 버전을 추가로 제공

3. 데이터 조인, 정렬, 그루핑

데이터 정렬

- 어제 판매한 상품 이름과 각 상품별 매출액 목록을 totalsAndProds RDD로 추출했음. 결과를 알파벳 순으로 정렬해야 함
- RDD의 데이터를 정렬하는데 주로 사용하는 변환 연산자는 repartitionAndSortWithinPartition, sortByKey, sortByKey. repartitionAndSortWithinPartition 임

3. 데이터 조인, 정렬, 그루핑

데이터 정렬

- val sortedProds = totalsAndProds.sortBy(_._2._2(1))
- sortedProds.collect()

3. 데이터 조인, 정렬, 그루핑

데이터 그룹핑

- 데이터 그루핑(grouping)은 데이터를 특정 기준에 따라 단일 컬렉션으로 집계하는 연산을 의미
- aggregateByKey, groupByKey(또는 groupBy), combineByKey 등 다양한 PairRDD 변환 연산자로 데이터를 그루핑할 수 있음

3. 데이터 조인, 정렬, 그루핑

데이터 그룹핑

- rdd.map(x => (f(x), x)).groupByKey()
- rdd.groupBy(f)
- groupByKey는 각 키의 모든 값을 메모리로 가져오기 때문에 이 메서드를 사용할 때는 메모리 리소스를 과다하게 사용하지 않도록 주의해야 함

Any Questions?