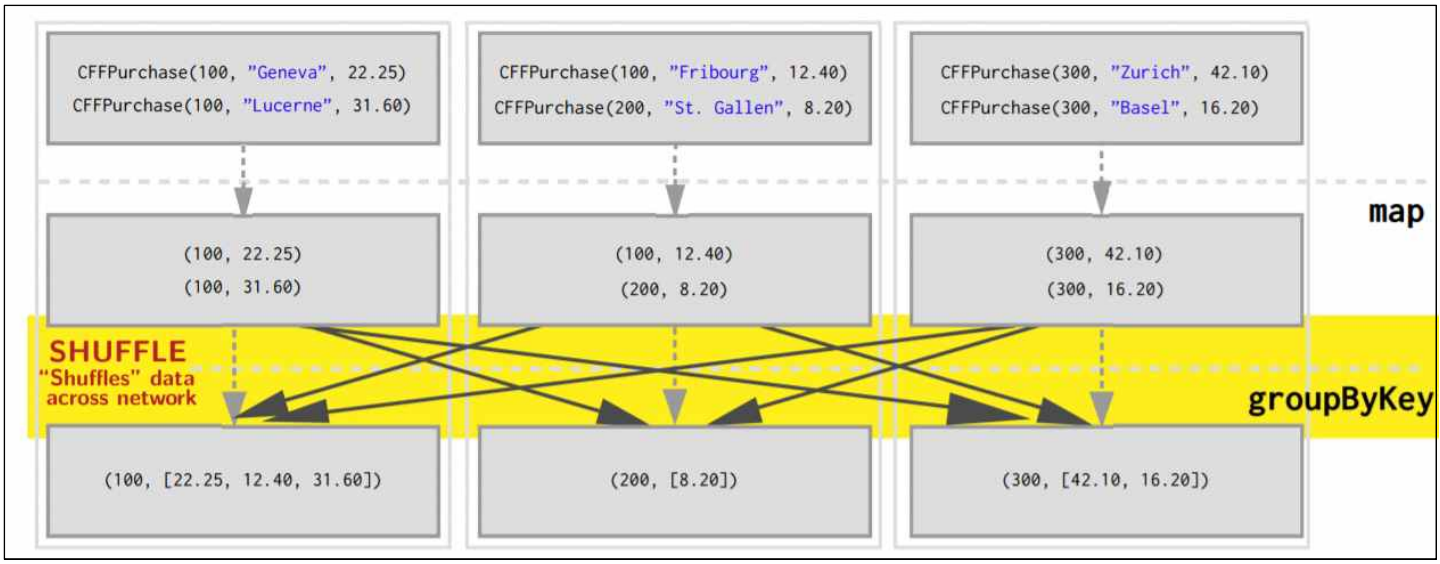
W3

12. Shuffling: What it is and why it's important

- groupByKey와 같은 함수는 데이터를 나누고 합쳐야하는데(shuffle) 어떻게 해야하는지.
- Shuffles can be an enormous hit to because it means that Spark must send data from one node to another. Why? **Latency!**
- map과 같은 함수에 비해 latency가 크다.
- 다음과 같이 같은 key는 노드에서 모아줘야하니까 GBK는 latency가 일어난다.

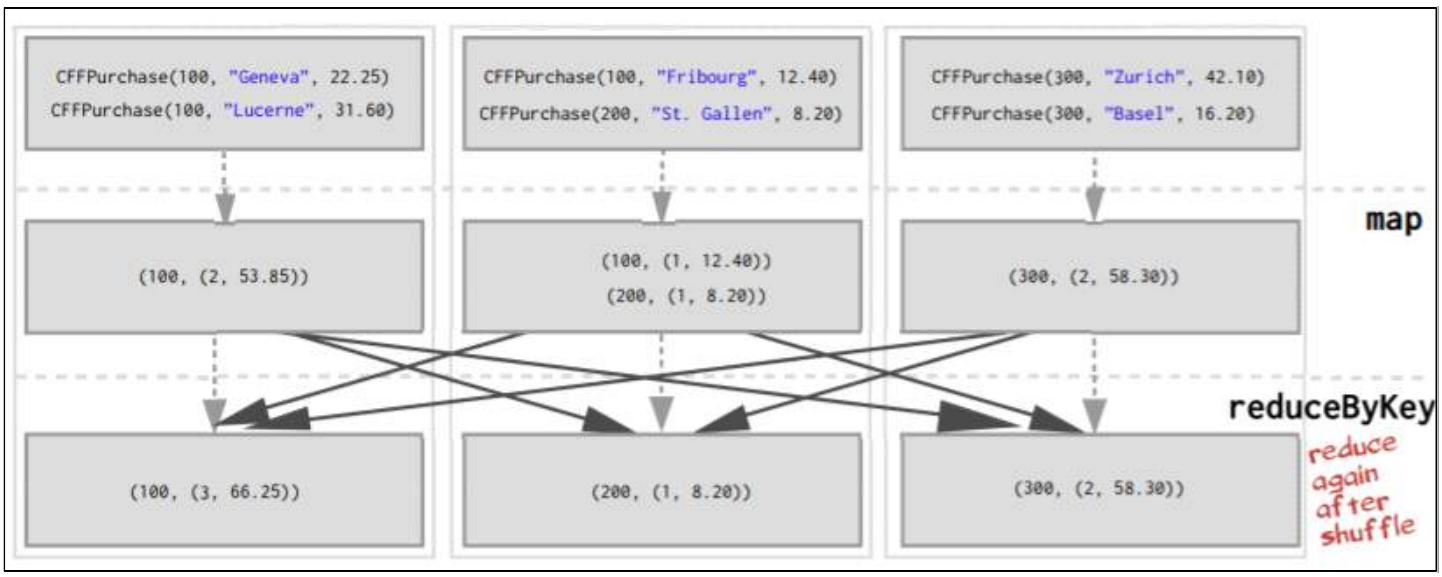


- 이를 어떻게 하면 줄일 수 있는지와 어떤 것이 latency를 악화시키는지 알아보자.

```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey() // groupByKey returns RDD[K, Iterable[V]]
```

- 위 코드를 RBK로 바꾸어주면

```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
              .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
              .collect()
```



- map 아래를 RBK로 보자. 그러면 network에 태워야하는 비용이 줄어든다.

== By reducing the dataset first, the amount of data sent over the network during the shuffle is greatly reduced.

- Let's benchmark on a real cluster. This can result in non-trival gains in performance!

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
                                                    .groupByKey()
                                                    .map(p => (p._1, (p._2.size, p._2.sum)))
                                                    .count()

purchasesPerMonthSlowLarge: Long = 100000
Command took 15.48s

> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
                                                    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
                                                    .count()

purchasesPerMonthFastLarge: Long = 100000
Command took 4.65s
```

* How does Spark know which key to put on which machine? GBK나 RBK 둘다
- By default, Spark uses hash partitioning to determine which key-value pair should be sent to which machine. key의 해시값을 통해 같은 노드에 놓는다.

13. Partitioning

- 파티션의 성질:
a. Partitions never span multiple machines.
   i.e., tuples(key-val pair) in the same partition are guaranteed to be on the same machine.
   같은 파티션은 하나의 노드에 있다.

b. Each machine in the cluster contains one or more partitions .
  노드는 하나 이상의 파티션을 갖고있어야 한다.

c. The number of partitions to use is configurable.
   By default, it equals the total number of cores on all executor nodes.
   파티션의 개수는 config 가능하며, 디폴트로 파티션의 개수는 executor nodes의 수와 같다.

- 종류: Hash Partitioning, Range Partitioning
  Custominzing은 Pair RDDs에서만 가능하다. (Paritioning은 Key를 갖고 나눠줘야 해서)

* Hash Partitioning
- pair RDD에 groupByKey를 때리면 (k, v)인 튜플에 대해 파티션 번호는 k.hashcod()의 파티션 개수로 나눈 나머지로 설정된다
  > p = k.hashCode() % numPartitions
- 특징: Hash partitioning attempts to spread data evenly across partitions based on the key.
- 단점: The result coud be(바뀜) very unbalanced distribution which hurts performance.
      해시값을 잘 신경 안 쓰니까, 몰릴 수가 있겠다. -> 길어짐

* Range Partitioning
- key 값이 정렬이 되어있다면 range partition의 결과도 key에 대해 정렬되어 나올 수 있다.
- 보다 균등하게 나눠줄 가능성이 높다.
ex) 좌 해시, 우 레인지(Set of ranges: [1 , 200], [201 , 400], [401 , 600], [601 , 800]) 일 때

- partition 0: [8, 96, 240, 400, 800]
- partition 1: [401]
- partition 2: []
- partition 3: []

- partition 0: [8, 96]
- partition 1: [240, 400]
- partition 2: [401]
- partition 3: [800]

* How do we set a partitioning for our data?
방법 2개
- 1. Call 'partitionBy' on an RDD, providing an explicit Partitioner.
- 2. Using transformations that return RDDs with specific partitioners.

* partitionBy

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))

val tunedPartitioner = new RangePartitioner(8, pairs)
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

데이터를 여러 번 쓸 때는 persist() , 셔플링을 할 것이니까 셔플링을 여러번 할 것 아니면 persist를 한다.! (중요)
원하는 파티션 개수 명세 가능하고, key에 대해 정렬된 형태로 반환할 것이며 range는 알맞게 정해준다.

* Partitioning Data Using Transformations

- Partitioner from parent RDD:
  PairRDD는 partitioned PairRDD 트랜스포메이션의 결과다. 이 때 만들어진 해시 파티셔너로 만들 수 있다.

- Automatically-set partitioners:
  몇 개의 RDD operation은 알려지고 말이되는 파티셔너를 사용한다.
  예를 들어 sortByKey는 Range, gbk는 해시.

Operations on Pair RDDs that hold to (and propagate) a partitioner:

- cogroup
- groupWith
- join
- leftOuterJoin
- rightOuterJoin
- groupByKey
- reduceByKey
- foldByKey
- combineByKey
- partitionBy
- sort
- mapValues (if parent has a partitioner)
- flatMapValues (if parent has a partitioner)
- filter (if parent has a partitioner)

All other operations will produce a result without a partitioner.

- 여기 없는 것들은 파티셔너가 없는 결과를 낸다. 다른 operation은 key를 바꿀 수 있기 때문이다.

Hence mapValues . It enables us to still do map transformations
without changing the keys, thereby preserving the partitioner. 이므로 맵밸류스가 좋다!

14. Optimizing with Partitioners

- Lec13 지난 강의에서 out-of-the-box(바로 사용가능한) 파티셔너 hash와 range 파티셔너들을 보았다. 셔플링을 할 때 파티션을 하는 것이 성능을 많이 높여준다.

* Optimization using range partitioning
- range partitioners를 reduceByKey를 하기 전에 한다면, 네트워크의 shuffling을 아예 없앨 수도 있다.

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
val tunedPartitioner = new RangePartitioner(8, pairs)

val partitioned = pairs.partitionBy(tunedPartitioner)
                       .persist()

val purchasesPerCust =
  partitioned.map(p => (p._1, (1, p._2)))

val purchasesPerMonth = purchasesPerCust
      .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
      .collect()
```

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
                                                    .groupByKey()
                                                    .map(p => (p._1, (p._2.size, p._2.sum)))
                                                    .count()

  purchasesPerMonthSlowLarge: Long = 100000
  Command took 15.48s

> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
                                                    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
                                                    .count()

  purchasesPerMonthFastLarge: Long = 100000
  Command took 4.65s
```

- 지난번에 GBK에서 3배 빨라진 것을 9배 빠르게 수행할 수 있다.

cf)근데 위에서 파티셔닝을 한 것 까지 합쳐 계산하면 이정도의 차이는 아닐 수도 있다.

On the range partitioned data:

```
> val purchasesPerMonthFasterLarge = partitioned.map(x => x)
                                                .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
                                                .count()

  purchasesPerMonthFasterLarge: Long = 100000
  Command took 1.79s
```

* 책 Learning Spark 의 다른 예시

- userData - BIG, containing (User ID, User Info) pairs, 사용자들이 구독한 주제들의 리스트
- events - small, containing (UserID, LinkInfo) pairs, 사용자들이 웹사이트에 5분 안에 누른 링크 정보
  -> 여기서, 구독된 주제가 아닌데 방문한 링크들을 알아내고 싶어 join을 할 것이다.

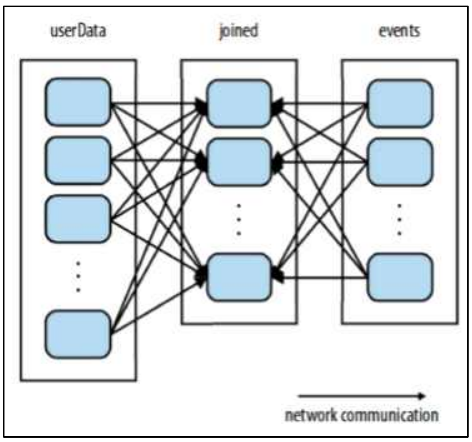
```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events) //RDD of (UserID, (UserInfo, LinkInfo))
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

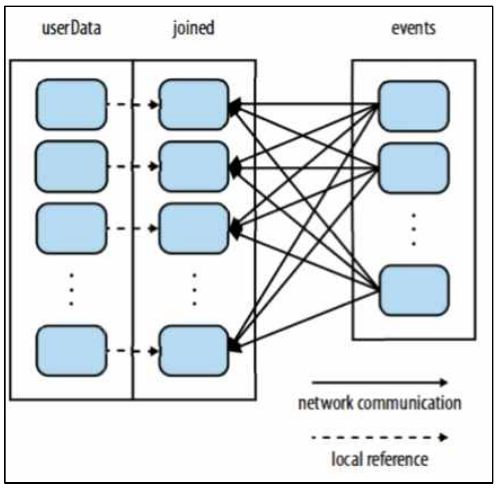
코드가 나쁘진 않지만 효율적으로 짜지 않았다.
왜냐하면 processNewLogs함수를 부를 때마다, join을 할 때 key들이 데이터셋에서 어떻게 파티션이 되어있는지 전혀 알지 못하기 때문이다.
그리고 UserdData는 바뀌지도 않는데, 위의 join에서 UserdData와
event RDDpair의 key를 모두 부를때마다 해시해 연결한다.

해결방법: UserdData RDD를 시작할 때 partitionBy를 사용!

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100)) // Create 100 partitions
                 .persist()
```

한줄만 추가해 user를 pre-partitioning하면 된다.
이러면, join을 할 때 파티션이 user에서는 유지가되고,
event만 해싱하고 셔플하면 된다. cf)오른쪽 아래 그림





* Back to Shuffling
- groupByKey에서의 파티셔닝은 어떻게 되는지 이해해보자.

```
val purchasesPerCust =
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
                .groupByKey()
```

key-value pair를 같은 키로 모든 값들을 그룹핑할 때, 같은 machine에서
key-value pair의 같은 key를 collect해야 한다.
그리고 gbk는 해시를 default로 수행하며, purchasesPerCust의 key는
만들어 질 때 받은 같은 해시값을 파티셔너로 사용할 것이다.

- 우리는 언제 셔플이 일어날지 알 것인가?
-> Rule of thumb은 a shuffle can occur when the resulting RDD depends on other elements from the same RDD or another RDD.
-> 셔플은 resulting RDD가 같거나 다른 RDD의 element(row)에 depend(참조)할 때 일어난다고 생각하면 된다.
ex) 같은 RDD에서는 reduce나 groupby, 다른 것은 join 같은 연산

이럴 때, 염두하고 프로그래밍을 하면 셔플링을 막거나 줄일 수 있을 것이다.

- 셔플링이 일어났는지 알아보는 또 다른 방법 2가지

1. The return type of certain transformations, *e.g.*,

```
org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366]
```

2. Using function toDebugString to see its execution plan: toDebugString을 통해 언제 파티션되는지 알 수 있다

```
partitioned.reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
           .toDebugString
res9: String =
(8) MapPartitionsRDD[622] at reduceByKey at <console>:49 []
 |  ShuffledRDD[615] at partitionBy at <console>:48 []
 |      CachedPartitions: 8; MemorySize: 1754.8 MB; DiskSize: 0.0 B
```

- 그리고 어떤 operation이 셔플을 발생할 가능성이 높은지 아는 것이 베스트이긴하다.
ex) cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey,
combineByKey, distinct, intersection, repartition, coalesce

* 보통 셔플링을 어떻게 피하는지(복습)
1. reduceByKey running on a pre-partitioned RDD will cause the values to be computed locally, requiring only the final reduced value has to be sent from the worker to the driver.
(groupByKey와 같은 연산을 pre-partitioned RDD에서 하면 연산은 locally 이루어지면서 셔플을 피할 수 있다.

2. join called on two RDDs that are pre-partitioned with the same partitioner and cached on the same machine will cause the join to be computed locally, with no shuffling across the network
join과 같은 연산을 하기 전에 pre-partitioned RDD에서 수행되게 하기.

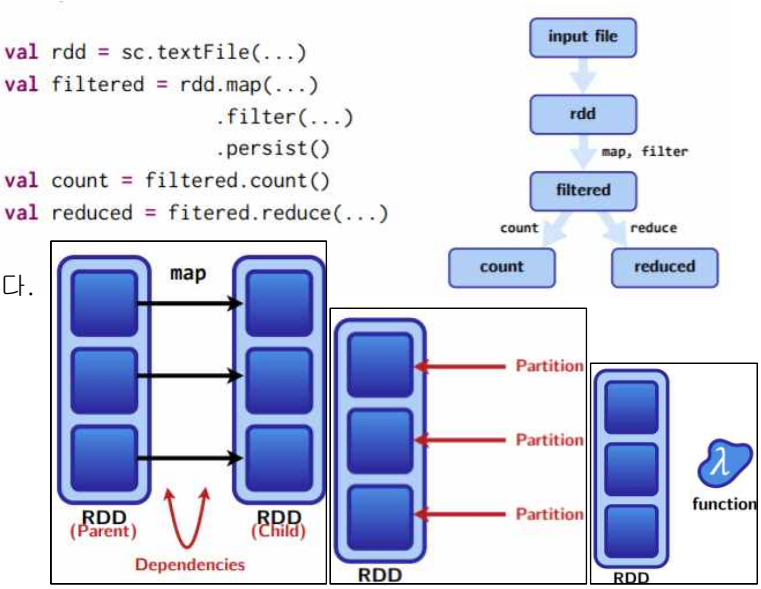
-> 이를 고려하면, 약 10배의 연산 성능을 올릴 수 있다.

15. Wide vs Narrow Dependencies

- 계산그래프에서 RDD간의 관계를 규정(dictate)하는 wide, narrow dependency에 대해 알아보자.
- Rdd가 어떻게 표현되고, 스파크가 어떻게 언제 데이터를 셔플할지, 그리고 dependency가 어떻게 fault tolerance를 가능하게 하는지, 또 얼마나 쉽게하는지 알아보자.

* Lineages (혈통, 계보)

- Computation을 DAG로 표현로 표현 가능하다.
  그리고 이러한 표현이 스파크가 최적화하는 것을 분석하는 것을 가능하게 한다.



* How are RDDs represented?

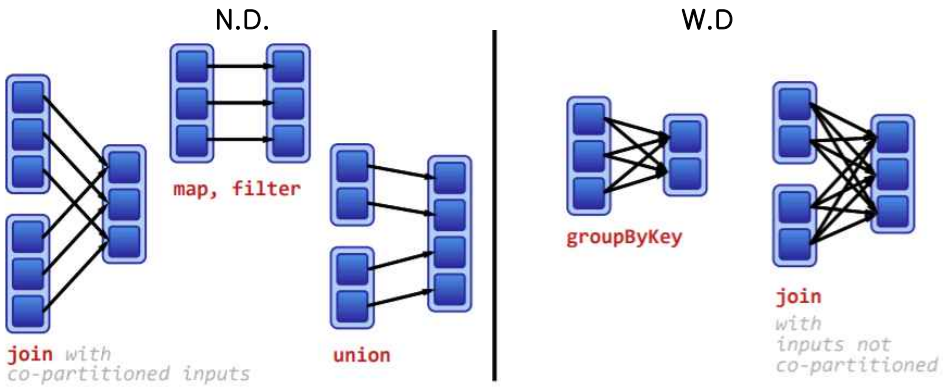
- RDD는 총 4개와 2개의 중요한 부분으로 나눌 수 있다.
  a. Partitions: 노드에 하나 이상에 있는 데이터셋의 atomic pieces
  b. dependencies: RDD와 그 파티션들과 그 RDD가 유도된 RDD간의 관계
  c. function: 부모 RDD에서 어떻게 연산할지
  d. meta data: 파티셔닝 스키마와 data placement

* RDD Dependencies and Shuffles

- 14강에서 셔플은 Resulting RDD가 같거나 다른 elements에 depend할 때 발생한다고 배웠다.
- RDD Dependencies는 데이터가 언제 네트워크를 탈지 encode한다.
- 이 말은 Transformation이 suffle을 일어나게 한다는 것이며,
  Transformation는 2가지 Dependencies 종류이며 이를 통해 언제 셔플이 일어나는지 알 수 있다.
  a. Narrow b. Wide

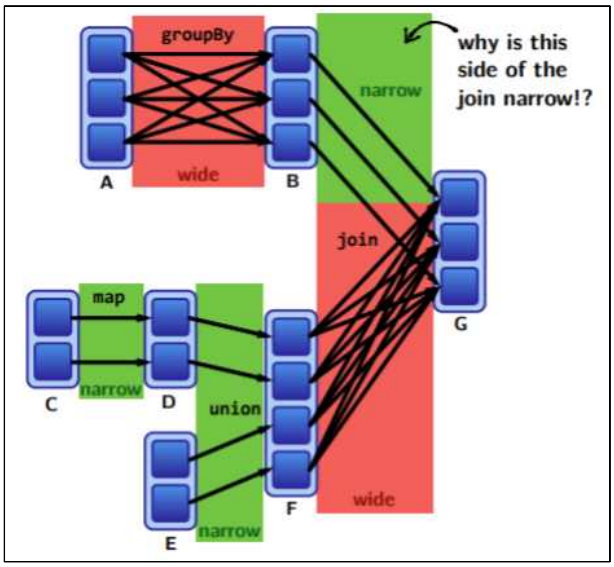
* Narrow Dependencies vs. Wide Dependencies

a. Narrow Dependencies: 부모 RDD의 파티션과 자식의 각 파티션이 1대 1 관계일 때 셔플이 필요 없고 빠르다.
b. Wide Dependencies: 여러 자식 파티션들이 하나의 부모 파티션을 참조(depend)할 때 셔플이 필요해 느리다.



- join이 같은 파티션된 인풋이면 N.D이다.

* vis ex on program and its dependencies
- 오른쪽의 왜 왼쪽 join은 narrow dep.인가?
G는 B에서 유도되었고, 이는 A에서 groupby되었기 때문에
B는 G의 입장에서 이미 co-partitioned되며 메모리에 캐시
됐다고 보는 것이다.



* 어떤 transformation이 어떤 dep.를 갖는지 잘 생각해보자.
- Narrow: map, mapValues, flatMap, filter, mapPartitions
mapPartitionsWithIndex,
- Wide: cogroup, groupWith, join, leftOuterJoin, rightOuterJoin
groupByKey, reduceByKey, combineByKey, distinct
intersection, repartition, coalesce
(these might cause shuffle)
join은 물론 Narrow일 때도 있지만.

* 고민하는 것도 좋지만 dep.를 알기위해 'dependencies' RDD 함수를 사용할 수 있다.
- Narrow: OneToOneDependency, PruneDependency, RangeDependency (거의 1:1dep가 자주 나온다.)
- Wide: ShuffleDependency:

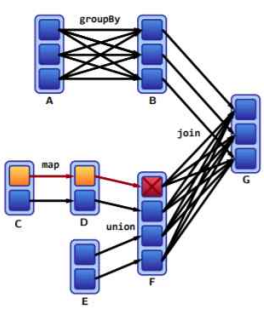
ex)
```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
                    .groupByKey()
                    .dependencies
// pairs: Seq[org.apache.spark.Dependency[_]] =
// List(org.apache.spark.ShuffleDependency@4294a23d)
```

ex) 또 이전에 배운 toDebugString을 사용하는 방법, lineage를 볼 수 있다.
```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
                    .groupByKey()
                    .toDebugString
//pairs: String =
//(8) ShuffledRDD[219] at groupByKey at <console>:38 []
// +-(8) MapPartitionsRDD[218] at map at <console>:37 []
//    |  ParallelCollectionRDD[217] at parallelize at <console>:36 []
```

* Lineages and Fault Tolerance
- Lineages가 Fault Tolerance을 가능케 핵심이다.
a. 함수형 프로그래밍인 스파크 -> **디스크에서 체크포인트를 안쓰고 F.T를 가능하게 하는 부분**
b. RDDs 는 immuatble하고,
c. 함수형 transformatino인 high-order function(map, flatmap, filter, ,..)을 immuatable한 데이터에 사용하며
d. 부모 RDD에 의존한 함수 연산이 RDD의 표현의 부분인 것 **-> 인메모리하고 F.T를 가능하게 하는 부분들 (b~d)**

이러한 dep. partition에 대해 잘 기록해둘 수 있도록 하는 성질들이
lineage graph에서 잃은 파티션들을 다시 계산해 failures를 복구를 통해 Fault Tolerance을 가능하게한다.

오른쪽 그림에서 빨강 부분 파티션에 failure가 일어났다고 하면 lineage한 dependency에
따라 부모 RDD의 파티션으로 부터 다시 유도해 재계산하면 된다.
이 때 narrow dep면 빠르지만 Wide이면 재계산이 훨씬 느리다.



<summary>

* 셔플링은 무엇이고 언제 발생하는가.
- 셔플링은 RDD가 어떠한 연산을 통해 재구성될 때, 네트워크를 통해 다른 서버로 이동하는 것이다.
셔플은 resulting RDD가 같거나 다른 RDD의 element(row)에 depend(참조)할 때 일어나며, 다음과 같은 연산을 수행할 때 발생할 가능성이 높다.
cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, distinct, intersection, repartition, coalesce
- 한 노드에서 다른 노드로 데이터가 옮겨지는 것을 셔플링이라고 하고, 네트워크 레턴시는 스파크 퍼포먼스에 영향을 많이 주기 때문에 항상 유의해야하며, 같은 key를 가진 데이터를 옮길때도 발생하지만 데이터를 partitioning 할때도 셔플링이 발생한다.

* 파티션은 무엇이며 어떤 특징이 있는가.
- 파티셔너에 의해 노드들로 나뉘어진 pairRDD 들로
a. 같은 파티션은 하나의 노드에 있다.
b. 노드는 하나 이상의 파티션을 갖고있어야 한다.
c. 파티션의 개수는 config 가능하며, 디폴트로 파티션의 개수는 executor nodes의 수와 같다.
는 성질을 갖고 있다.

* 스파크에서 제공하는 partitioning 의 종류는 어떤 것인가.
- Hash Partitioning: key의 해시코드를 파티션의 개수로 나눈 나머지로 노드에 설정이 되며, 키에 의해 고르게 나누려고 해서 고르지 않게 나뉠 수 있는 단점이 있따.
- Range Partitioning: Key 값을 범위를 활용해 노드에 파티션을 나누는 방법으로 균등하게 나눌 가능성이 더 높고, key값이 정렬이 되어있다면 결과도 정렬이 되게 받을 수 있다.

* 파티셔닝은 어떻게 퍼포먼스를 높여주는가.
- reduceByKey와 같은 연산을 pre-partitioned RDD에서 수행하면 셔플링을 피해 수행할 수 있다.
- Join과 같은 연산을 하기 전의 데이터를 pre-partitioned되게 하면 셔플링을 줄이거나 막을 수 있다.
셔플링을 줄이거나 없애서 latency를 줄여 퍼포먼스를 높인다.
- data locality를 최적화하여 많은 shuffling이 발생할 수 있는 작업을 대폭 줄일 수 있다.
예를 들면 pre-partitioned RDD에서 reduceByKey 를 수행하는 경우,
같은 partitioner로 pre-partitioned된 두개의 RDD를 join하는 경우 등은 shuffling이 발생하지 않게 된다.

* rdd 의 toDebugString 의 결과값은 무엇을 보여주는가.
- toDebugString을 통해 어디서 파티션되는지 알 수 있다.
어떤 함수는 어떤 RDD를 결과로 보여주는지 lineage를 볼 수 있다.

* 파티션 된 rdd에 map 을 실행하면 결과물의 파티션은 어떻게 되고 mapValues의 경우는 어떤가.
- 결과도 같은 노드에서 연산이 locally 이루어지므로 latency를 줄일 수 있다. 파티션된 rdd가 해싱 파티션이라면 같은 해시값을 사용할 것이며, mapValues 연산까지 이어질 것이다.
- partitioned 된 rdd에 map, flatMap을 사용하면 partitioner가 유지되지 않음 (map은 key가 바뀔 수 있는 transformation이기 때문에 map 할 경우 partitioner가 유지되지 않음)
mapValues에서는 parent가 partitioner를 가졌을 경우 그대로 유지됨(mapValues는 key를 바꾸지 않고 map transformation을 하게 해주기 때문에 partitioner가 유지됨)

* Narrow Dependency 와 Wide Dependency 그리고 각 Dependency를 만드는 operation은 무엇인가.
- Narrow Dependencies: 부모 RDD의 파티션과 자식의 각 파티션이 1대 1 관계일 때
- Wide Dependencies: 여러 자식 파티션들이 하나의 부모 파티션을 참조(depend)할 때

- Narrow: map, mapValues, flatMap, filter, mapPartitions
  mapPartitionsWithIndex,
- Wide: cogroup, groupWith, join, leftOuterJoin, rightOuterJoin
  groupByKey, reduceByKey, combineByKey, distinct
  intersection, repartition, coalesce
  (these might cause shuffle)
  join은 물론 Narrow일 때도 있다.

* Lineage 는 어떻게 Fault Tolerance를 가능하게 하는가.
- lineage graph에서 잃은 파티션들을 다시 계산해 failures를 복구를 통해 Fault Tolerance을 가능하게 한다.
ineage한 dependency에 따라 부모 RDD의 파티션으로 부터 다시 유도해 재계산하면 된다. 이 때 narrow dep면 빠르지만 Wide이면 재계산이 훨씬 느리다.
- RDD는 immutable하고, RDD를 transformation하기 위해 map, flatMap, filter 등 deterministic한 higher-order function을 사용한다. RDD 자체가 어떤 데이터셋, 어떤 function을 거쳐서 만들어졌는지 lineage 정보를 기억하고 있기 때문에 특정 partition이 문제가 생길 경우 dependency 를 추적하여 다시 계산할 수 있다.