1-1. Introduction, Logistics, What you'll learn.

- 이전에 배운 2강의는 함수형 프로그래밍과 병렬처리에 대해 배웠다. 이 수업 들으면 커버 됨.
- 스파크에서 어떻게 데이터를 병렬처리하는지 깊게

* Why scala?

a. 스파크 vs. 파이썬, R, Matlab

- 파이썬, R, Matlab은 소규모 데이터에 한정되어 분산처리 해야함. (메모리 베이스 프로그래밍 언어들이다.)
- 대규모 데이터 처리는 단일 컴퓨팅 노드에서 해결될 수 없으며 스파크를 활용해 1개의 노드에서 수십개에서 수천개의 노드로 신속하게 확장이 가능하다.

b. 하둡 vs. 스파크

- 스파크의 표현력(more expressive)이 더 좋음. ex) map, flatMap, filter, reduce
- 성능. 대화식으로 처리가능
- 데이터 과학에 더 좋음(iteration에 좋음). 하둡은 boilerplate를 너무 많이 선언해야함

c. 스파크와 스칼라의 기술이 스택오버플로우의 미국에서 가장 돈 많이 버는 기술 이라고 한다. 그리고 스파크가 스칼라 API로 만들어져있다. 스파크 스칼라를 둘다 배우는 것이 좋다.

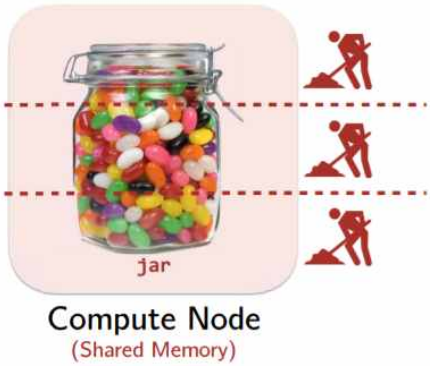
* What we Learn in this Course.

- 병렬 프로그래밍에서 배웠던 스파크를 이용한 분산처리 수업을 확장한 데이터 병렬 패러다임
- 스파크 프로그래밍 모델을 심도있게 본다.
- 분산 계산과 클러스터가 어떻게 배치되는지.
- 스파크의 성능을 향상시키는 방법(오래 배울 것이다.) 재계산 방지 방법과 data locality(모아주는지) 그리고 특히 **data shuffle**
- relational operations in SQL 모듈, 데이터프레임과 데이터 셋. 의 이점과 제약 (RDD보다 성능이 좋다. RDD 알면 쉽다.)

1-2. Data-Parallel to Distributed Data-Parallel (데이터 병렬처리에서 분산 데이터 병렬처리로)

* Shared memory data parallelism:

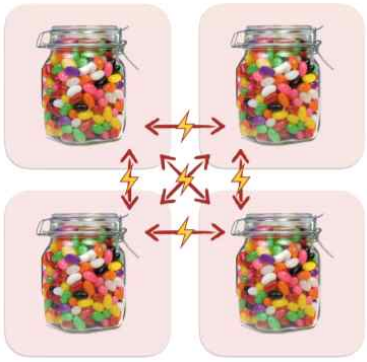
- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).



val res = jar.map(jellyBean => doSomething(jellyBean))

그림으로 다음과 같다. worker(쓰레드, CPU)가 작업을 수행할 것이다.
스칼라의 병렬 컬렉션(list, set, map(딕셔너리))은 Shared memory data parallelism로 실행이 되도록 추상화 되어 있다.

* Distributed data parallelism:
- Split the data over several nodes. (사실 jar를 잘라야함)
- Nodes independently operate on the data shards in parallel.
- Combine when done (if necessary).



val res = jar.map(jellyBean => doSomething(jellyBean)) 같다 코드는
스칼라 데이터 컬렉션에서 분산 데이터 병렬처리를 할 줄 알면 스파크에서도 할 줄 안다고 할 수 잇다.

레이턴시가 중요해졌다.(네트워크 비용)
CPU에서 나눈 것을 노드들에게 나누는 것이 달라졌고, 하나의 노드에 여러 노드가 있을 수 도 있다.

스파크는 RDDs(Resilient Distributed Datasets) 라는 분산 데이터 병렬 모델로 수행하며, 이것은 low-level이며 처음 나온 데이터 컬렉션(structure)이다?

cf) 동시성과 병렬성은 다르다.

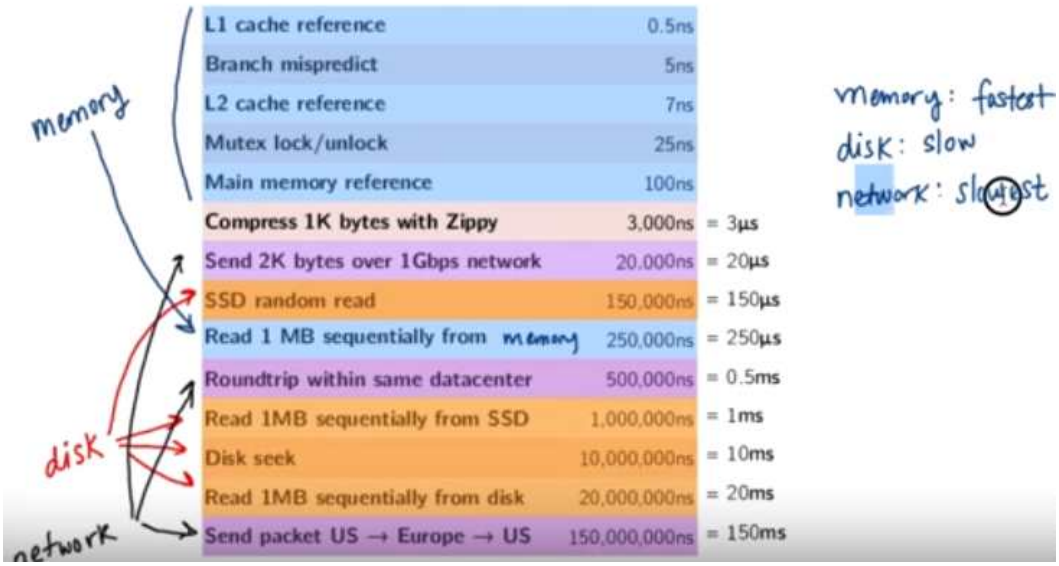
1-3. Latency

이전의 병렬 프로그래밍 강의에서 멀티 코어(프로세서)에서의 데이터 병렬처리와 병렬 페러다임을 배웠다면
이제 분산의 설정에서 데이터 병렬처리를 배우고, 아파치 스파크의 추상화된 분산 컬렉션에서의 패러다임을 배우자.

* Distribution: 분산 병렬처리는 공유 메모리에서의 병렬처리에서 2가지를 더 고려해야한다

- Partial failure: crash failures of a subset of the machines involved in a distributed computation.
(노드가 뻗는 것)
- Latency: certain operations have a much higher latency than other operations due to network communication. 특히 그리고 latency를 최소화 하기위해 생각해봐야한다.

* 메모리, 디스크, 네트워크에서의 latency 비교.



하둡/맵리듀스는 간결한 API(map과 reduce step으로 수행하는)와 FAULT TOLERANCE를 보장하여 ground-breaking 했다.

FAULT TOLERANCE: 하나의 노드가 죽어도, 다른 노드가 그 일을 수행할 수 있도록 하는 것. -> 여러 노드를 수행할 수 있도록 함.

* 하둡/맵리듀스 vs. 스파크
하둡/맵리듀스는 데이터를 FAULT TOLERANCE를 위해, 디스크에 write한다.
스파크는 FAULT TOLERANCE를 유지하고 latency를 조절하기 위해 다른 전략을 사용한다.
아이디어: 모든 data를 메모리에서 불변하게(immutable) 하도록 하여.(디스크까지 안가고) 이 것이 함수형 프로그래밍의 아이디어?
이를 활용해 하둡보다 100배(iter 30일 때 로지스틱 회귀에서) 빨라지고 준 실시간으로 사용할 수 있도록 해줌
특히 iteration이 많아질수록 더 차이남.

1-4 RDDs, Spark's Distributed Collection

* RDDs는 immutable한 seq와 parallel한 스칼라 컬렉션과 닮았다.

```
abstract class RDD[TJ {
  def map[U](f: T => U): RDD[UJ = ...
  def flatMap[UJ(f: T => TraversableOnce[UJ): RDD[UJ = ...
  def filter(f: T => Boolean): RDD[TJ = ...
  def reduce(f: (T, T) => T): T = ...
}
```

map, flatMap, filter, reduce는 스칼라 컬렉션

여기서, map 이 jar, f(element?)가 jelly bean 느낌, (f: T => U)는 메소드 표현 한 것? U는 return 타입이 새로 만들어 지는 것.

fiter는 True가 되는 것만
reduce는 RDD가 포함한 element 2개를 합치는 느낌

* Combinators on Scala parallel/ sequential collections과 Combinators on RDDs:
 map, flatMap, filter, reduce, fold, aggregate로 이름이 같다!

컬레션들의 시그니처(함수의 정의)를 보면 약간 다르지만 그래도 의미적으로는 같다.

```
map[BJ(f: A=> B): List[BJ // Scala List
map[B](f: A=> B): RDD[BJ // Spark RDD
flatMap[BJ(f: A=> TraversableOnce[B]): List[BJ // Scala List
flatMap[BJ(f: A=> TraversableOnce[B]): RDD[BJ // Spark RDD
filter(pred: A=> Boolean): List[AJ // Scala List
filter(pred: A=> Boolean): RDD[AJ // Spark RDD
reduce(op: (A, A)=> A): A // Scala List
reduce(op: (A, A)=> A): A // Spark RDD
fold(z: A)(op: (A, A)=> A): A // Scala List
fold(z: A)(op: (A, A)=> A): A // Spark RDD
aggregate[BJ(z: => B)(seqop: (B, A)=> B, combop: (B, B) => B): B // Scala
aggregate[BJ(z: B)(seqop: (B, A)=> B, combop: (B, B) => B): B // Spark RDD
```

map[BJ(f: A=> B): List[BJ
B는 func element의 타입 , f의 input 이름, A => B이거는 A가 인풋이가 B가 아웃풋인 함수
B를 아웃풋으로 하는 function으로 보면 된다.

TraversableOnce는 arr나 list같은 iterator
aggregate은 어떻게 다른지 생각해보라 라고 강의에서 했다. 스칼라에서는 (z: => B)가 evaluate(?)되야 값으로 저장되는 것이지만 RDDs에서는 func로 돌리는 것보다 값을 그냥 처음부터 주는 것이 데이터적으로 편하기 때문에 저렇게 했다.

* val encyclopedia: RDD[String]가 주어졌을 때, encyclopedia에서 "EPFL"이 들어간 페이지를 찾는 코드

```
val result= encyclopedia.filter(page => page.contains("EPFL")).count()
```

encyclopedia라는 RDD에서, page.contains("EPFL")이 참이 되는 페이지들만 count해 넘겨준다는 의미.

* Spark에서 Hello, World는 Word Count이다.

```
// Create an RDD
val rdd = spark.textFile("hdfs://...")

val count = rdd.flatMap(line => line.split(" ")) // separate lines into words
              .map(word => (word, 1))           // include something to count
              .reduceByKey(_ + _)               // sum up the 1s in the pairs
```

val rdd = ~ 는 각 줄을 RDD[string]에 넣어준다. rdd element는 각 줄 이 된다.

flatMap에서 각 줄을 공백으로 쪼갠 element를 빼주는 rdd.flatMap(line => line.split(" "))는 한 단어, 단어가 element가 된다.
.map(word => (word, 1)):는 일종의 dictionary로 만들어 주는 느낌인 것 같고
.reduceByKey(_ + _): key들을 이용해 값들을 합치는 것?

* How to Create an RDD?
- Transforming an existing RDD.
- From a SparkContext (or SparkSession) object.: (항상 sparkContext를 이용해 spark랑 대화한다.
  - parallelize: 로컬스칼라 컬렉션을 RDD로 변환
  - textFile: HDFS또는 local File system을 읽어 각 줄을 element로 하는 RDD로 변환

- SparkContext(renamed: 스파크 섹션)는 스파크 클러스터에 대한 너의 handle로 생각할 수 있다. spark 클러스터와 실행중인 애플리케이션 사이의 연결을 나타내며, 이는 새 RDD를 생성하고 채우고 사용할 수 있는 방법을 정의한다.

1-5. RDDs: Transformation and Actions

* 스칼라에서의 Transformers 와 accessors
- Transformers: (element를 변경해줌) Return new collections as results. (Not single values.)
  ex) map(리스트에서 리스트), filter(리스트에서 리스트), flatMap(리스트에서 리스트인데 값이 다른 리스트), groupBy(flatMap의 반대되는 연산)

  map(f: A=> B): Traversable[BJ

- accessors: (element에 접근해 변경해줌) Return single values as results. (Not collections.)
 ex) reduce, fold, aggregate.

  reduce(op: (A, A)=> A): A
 같은 타입 A 두 개를 가져와 새로운 하나로 만들어 주는거 string과 과 같은.

* 위와 유사하게 스파크에서는 transformations과 actions을 RDDs에서 한다.
스칼라와 유사하지만 중요하게 다른 점들이 있다.

- Transformations: Return new RDDs as results. Not Collections
다른점은 Transformations은 lazy하다. 바로 계산하지 않는다. ex) map을 쓰면 일종의 예약만한다.

- Actions. Compute a result based on an RDD, and either returned orsaved to an external storage system (e.g., HDFS). RDD에 기반하여
액션은 트랜스포메이션과 다르게 eager하다.

lazy, eager 특성이 스칼라와 가장 다른 특징이며
=> 이렇게 하는 이유는 다음 네트워크 통신을 제한할 수 있게 하기 때문이며 자세한 방법은 다음 강의에 나온다.

* Example

```
val largelist: List[String] = ...
val wordsRdd = sc.parallelize(largelist)          // RDD[String] 이 되도록 예약
val lengthsRdd = wordsRdd.map(_.length)           // RDD[Int] 이 되도록 예약
```

sc가 sparkcontext이며 2번째 줄이 리스트를 RDD로 만드는 것이다.
3번째 줄은 각 줄(element)의 길이의 데이터로 변형하는 코드지만, 예약만하고 실행은 되지 않는다.

```
val totalChars = lengthsRdd.reduce(_ + _)
```
이거 까지 해야 실행이 된다.

* Transformation

map	**map[B](f: A => B): RDD[B]** ← Apply function to each element in the RDD and retrun an RDD of the result.
flatMap	**flatMap[B](f: A => TraversableOnce[B]): RDD[B]** ← Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned.
filter	**filter(pred: A => Boolean): RDD[A]** ← Apply predicate function to each element in the RDD and return an RDD of elements that have passed the predicate condition, pred.
distinct	**distinct(): RDD[B]** ← set과 비슷함 Return RDD with duplicates removed.

* Actions: 결과들이 바로 메모리에 들어간다.

collect	**collect(): Array[T]** ← Return all elements from RDD.
count	**count(): Long** ← Return the number of elements in the RDD.
take	**take(num: Int): Array[T]** ← Return the first num elements of the RDD.
reduce	**reduce(op: (A, A) => A): A** ← Combine the elements in the RDD together using op function and return result.
foreach	**foreach(f: T => Unit): Unit** ← Apply function to each element in the RDD.

* Another Example

Let's assume that we have an RDD[String] which contains gigabytes of logs collected over the previous year. Each element of this ROD represents one line of logging.

Assuming that dates come in the form, YYYY-MM-DD:HH:MM:SS, and errors are logged with a prefix that includes the word "error" ...

How would you determine the number of errors that were logged in December 2016?

```
val lastYearslogs: RDD[String] = ...
val numDecErrorlogs = lastYearslogs.filter(lg => lg.contains("2016-12") && lg.contains("error")).count()
```

근데 10개만 필요하다면 다음과 같이 코딩하면 된다.

```
val lastYearslogs: RDD[String] = ...
val firstlogsWithErrors = lastYearslogs.filter(_.contains("ERROR")) .take(10)
```

이렇게 함으로써 언제 멈추어 시간을 아낄 수 있는지 결정할 수 있다.

* Transformations on Two RDDs

union
`union(other: RDD[T]): RDD[T]` ←
Return an RDD containing elements from both RDDs. ex) val rdd3 = rdd1.union(rdd2)

intersection
`intersection(other: RDD[T]): RDD[T]` ←
Return an RDD containing elements only found in both RDDs.

subtract
`subtract(other: RDD[T]): RDD[T]` ←
Return an RDD with the contents of the other RDD removed.

cartesian
`cartesian[U](other: RDD[U]): RDD[(T, U)]` ←
Cartesian product with the other RDD.

* Other Useful ROD Actions (이건 스칼라와 좀 다르다)

takeSample
`takeSample(withRepl: Boolean, num: Int): Array[T]` ←
Return an array with a random sample of num elements of the dataset, with or without replacement.

takeOrdered
`takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]` ←
Return the first n elements of the RDD using either their natural order or a custom comparator.

saveAsTextFile
`saveAsTextFile(path: String): Unit` ←
Write the elements of the dataset as a text file in the local filesystem or HDFS.

saveAsSequenceFile
`saveAsSequenceFile(path: String): Unit` ←
Write the elements of the dataset as a Hadoop SequenceFile in the local filesystem or HDFS.

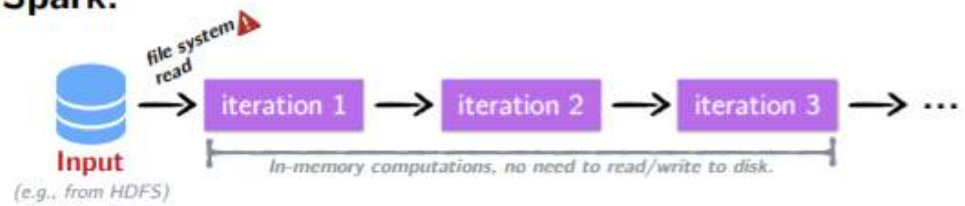
1-6. Evaluation in Spark: Unlike Scala Collections!

* Iteration and Big Data Processing: 데이터 사이언스에서 iteration을 포함하는 문제가 많아 스파크가 적합하다.

Iteration in Hadoop:

하둡은 90%의 시간이 IO에 시간을 낭비한다.

Iteration in Spark:



로지스틱회귀는 open-form 이기 때문에 iter를 해야 한다.(경사하강법으로) 로지스틱 회귀 코드는 다음과 같다.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

여기서, 그레디언트를 구하는
val gradient = points.map { p =>(1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y}.reduce(_ + _)
를 할 때 매번 **val points = sc.textFile(...).map(parsePoint)** 를 해야하니까, 시간이 오래 걸린다.
그래서 아래처럼 캐싱을 어떻게 하는지 알려준다.

* Caching and Persistence
- 스파크는 무엇이 메모리에 캐시가 되야되는지의 권한을 사용자에게 준다.

To tell Spark to cache an RDD in memory, simply call **persist()** or **cache()** on it!
왠만하면, cache()를 쓰는데, cache()를 까면 persist()를 인메모리에 저장해라! 그리고 cache()도 트랜스포메이션이여서 lazy execution이며 액션이 들어와야한다.

ex)
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
val numErrors = logsWithErrors.count() // 그리고 이렇게 실행했을 때 실제로 더 빠르더라.

* Logistic Regression Example

```
val points = sc.textFile(...).map(parsePoint).persist()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

* cache() and persist()
- cache(): 바로 메모리에
- persist(): 어디에 저장할지 정해줄 수 있다. 다음과 같은 공간에
 cf) serialize: arrays of bytes와 같은 형태로 저장. 계산시간이 걸리지만 용량은 더 적음.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
MEMORY_ONLY	High	Low	Y	N
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK*	High	Medium	Some	Some
MEMORY_AND_DISK_SER†	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

그리고 Default는 Memory_only 이다. 강의에서는 모두 메모리에 올릴 것이다.

이렇게 Spark의 RDDs는 deferred하는 것이 스칼라 컬렉션과 다르다.

그리고 아래와 같이 transformation을 한번에 걸어주는 것이 데이터를 한 번만 읽기 때문에 더 좋다.

```
val lastYearsLogs: RDD[String] = ...
val numErrors = lastYearsLogs.map(_.lowercase)
                             .filter(_.contains("error"))
                             .count()
```

그리고 take(10)과 같은 것은 10개만 찾고 끝내므로 좋다.

가장 많이 하는 실수가 캐시에 있을 때 several transformation에 대해 re-evaluate를 하는 것이다.

1-7. Cluster Topology Matters!

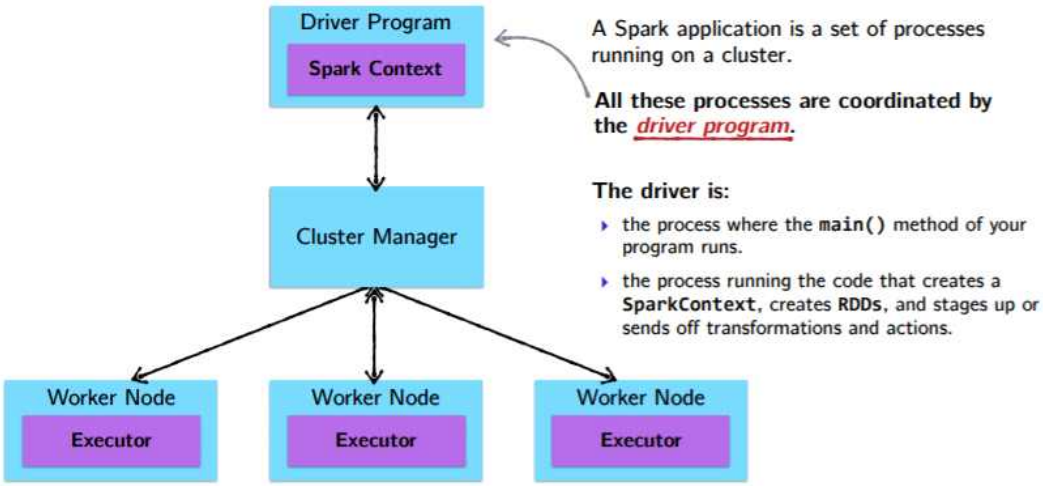
- 클러스터의 구조의 중요성. 스파크에서 클러스터가 어떤 구조로 있고 구조의 어떤 부분에서 내 코드가 실행되고 있는지 숙지하고 있어야 디버깅이 편하다.

* How Spark Jobs are Executed

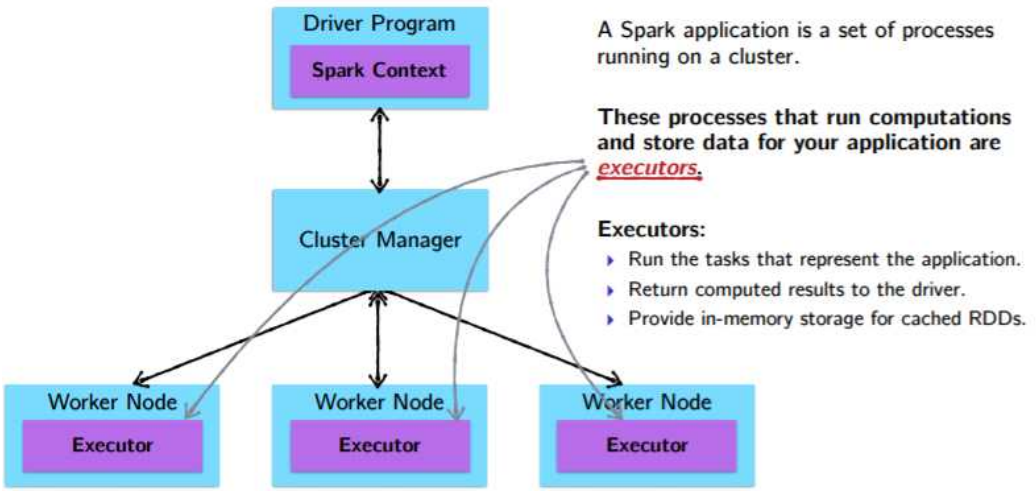
- Master는 주로 사용자가 실제로 다루는 컴퓨터. IDE가 있다면 IDE가 돌아가고 있는 컴퓨터. 그리고 Spark Context를 갖고 있다.

- Worker는 job들을 실제로 실행하는 노드들이다.

- Cluster Manager를 통해 자원(resource)을 할당하고 스케줄을 관리하여 통신한다.
  ex) Mesos/YARN



- 스파크 애플리케이션은 클러스터에서 실행되는 프로세스의 집합이며 모든 프로세스들은 driver program에 의해 조정된다.
- Driver: 프로그램의 main() 메소드가 실행되는 프로세스, spark Context와 RDD를 생성하고 Transformation을 예약하고 Actions을 수행.



- Executors: 실제로 일을 수행.

- 일은 오른쪽과 같은 순서로 수행이 된다.
- 4번에서 app code를 보내야하는데 어떻게 보내야하냐:
User가 쓰는 모든 것들이 Serialize된다? 어떤 파일에 저장해
User가 만든 함수와 같은 것들이 노드에서도 사용될 수 있도록

Serialize되는 과정에서 에러가 많이 날 수 있으니 유념해야한다.

Execution of a Spark program:

1. The driver program runs the Spark application, which creates a **SparkContext** upon start-up.
2. The **SparkContext** connects to a cluster manager (e.g., Mesos/YARN) which allocates resources.
3. Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application.
4. Next, driver program sends your application code to the executors.
5. Finally, **SparkContext** sends tasks for the executors to run.

* Example1: A simple prinln

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

val people: RDD[Person] = ...
people.foreach(println)
```

다음을 실행하면 Driver에서 아무것도 하지 않는다.
executor에서 바로 실행되며, 드라이버 노드에 output을 가져오지 않는다. 일종의 부작용?

* Example2: A simple take

```
case class Person(name: String, age: Int)
val people: RDD[Person] = ...
val first10 = people.take(10)
```

그리고 위를 실행하면 take(10)은 driver에 저장된다.
보통 액션을 수행하면 driver program에 올라가게 된다.

* 위와 같은 구조와 어떤 코드가 lazy고 eager인지 잘 알아야 한다.

<SUMMARY>
* Shared Memory Data Parallelism(SDP)와 Distributed Data Parallelism(DDP)의 공통점과 차이점
- 공통점: 데이터를 나누어 병렬로 처리하고 처리가 완료되어 필요하면 처리한 결과를 합친다.
- 차이점: SDP는 하나의 노드에서 여러 프로세서가 병렬로 처리하는 방식이고, DDP는 여러 노드가 메모리에서 병렬로 처리하고 네트워크를 활용하여 Latency와 Partial Failure를 고려해야한다.

데이터를 병렬적으로 처리한다는 점은 동일하나, SDP의 경우 단일 머신의 메모리 안에서(멀티코어/멀티프로세서) 병렬 처리가 이루어지고, DDP의 경우 여러 노드/머신에 나눠서 처리함. DDP의 경우 네트워크 비용이 발생하게 되며, latency를 해결해야 한다.

* 분산처리 프레임워크 Haddop의 Fault Tolerance는 DDP의 어떤 문제를 해결하는가.
- Partial Failure의 문제점을 보완해 처리하던 노드가 죽어도 다른 노드가 그 일을 수행할 수 있도록 해 DDP를 가능하게 했다.
- 여러 노드 중 한 노드에 이상이 생겨도, 다른 노드로 같은 데이터를 전달시킨 후 작업시켜서 성공할때까지 시도함으로써, 작업의 안정성을 보장한다.

* Spark가 하둡과 달리 데이터를 메모리에 저장하면서 개선한 것 무엇이고, 왜 메모리에 저장하면 그것이 개선이 되는가.
- 하둡은 디스크에 데이터를 write하면서 수행하는 반면, 스파크는 데이터를 immutable 하도록 하는 함수형 프로그래밍의 아이디어를 통해 하둡보다 iter 30 일 때의 로지스틱회귀에서 속도를 약 100배 빠르게 처리할 수 있도록 했으며 빅데이터를 활용한 준 실시간 서비스를 수행할 수 있도록 했다.
- Hadoop에서는 발생 가능한 failure로부터 복구할 목적으로 모든 map, reduce 과정에서 데이터를 disk에 쓴다. 반면 Spark에서는 disk 대신 in-memory를 이용, disk 쓰기 속도보다 in-memory가 100x 정도 빠르다

* Eager execution와 Lazy execution의 차이점은 무엇인가.
- Transformations들은 Lazy execution으로 RDD를 처리하여 새로은 RDD를 결과로 나타낼 것을 예약하는 실행이며, Actions들은 Eager execution으로 바로 메모리에 결과 값을 저장한다.

* Transformation과 Action의 결과물 (Return Type)은 어떻게 다른가.
- Transformations은 RDD를 결과로 Action은 Array, Long, Unit과 같은 값 또는 HDFS와 같은 파일로 리턴한다.

* RDD.cache()는 어떤 작동을 하고, 언제 쓰는 것이 좋은가.
- 데이터를 메모리에 올려놓는다. 기계학습을 수행할 때 데이터를 메모리에 저장해 iteration을 돌 때 한 번만 데이터를 읽는 작업을 수행할 때 좋다.
- evaluate된 RDD를 메모리에 저장하여, 이후에 다시 사용할 수 있음. 한 RDD를 여러 action의 input으로 쓸 때 이렇게 하면 중복 계산 작업을 막을 수 있다. 특히 iteration 에서 반복적으로 사용되는 경우 메모리에 올려놓고 사용 하는게 유리함.

* Lazy Execution이 Eager Execution 보다 언제 더 빠른가.
- Eager execution은 여러 단계의 연산을 수행 할 때, 한 단계씩 차례차례 진행하는 반면, lazy execution은 한번에 진행한다. Lazy execution 과정에서는 모든 단계를 고려한 최적화 작업이 자동으로 이루어지는데, 그 과정에서 필요없는 연산은 생략 되는 등의 효과가 있기 때문에 각 단계를 무조건 수행하는 eager execution 보다 빠를 수 있다.