Spark

EE412: Foundation of Big Data Analytics

Announcements

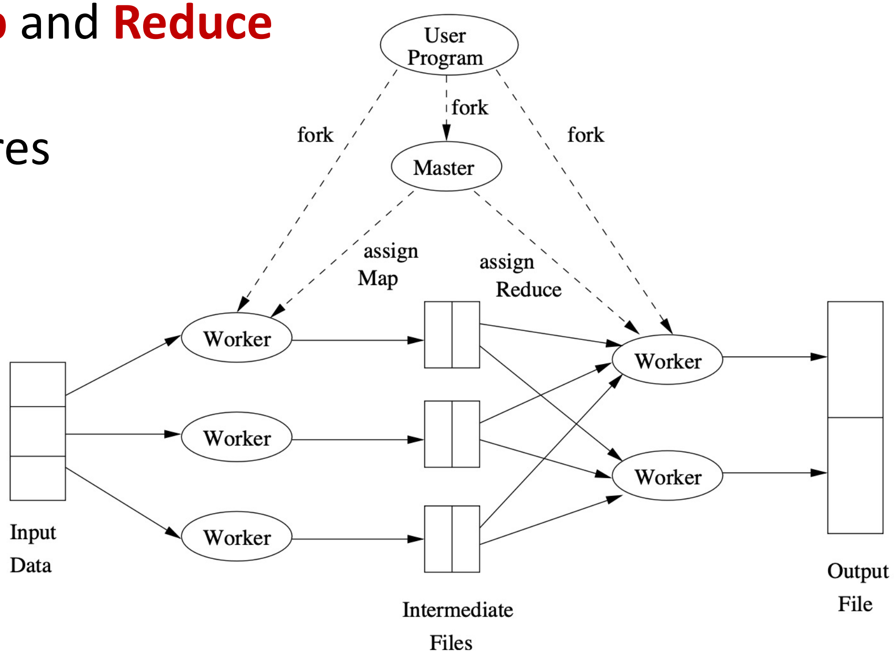
- **To do reminder**
  - Register to Classum
  - Login to Haedong machine and change your password
    - Please check an announcement at Classum
- **Homeworks**
  - Will post HW0 this Thursday (deadline: 09/21)
  - Will post HW1 next Thursday (deadline: 10/05; updated)
- **Classes**
  - No classes at 09/19 and 09/21 (videos will be uploaded)

Class Schedule (tentative)

Date	Out	In
09/07 (Thu)	HW0	
09/14 (Thu)	HW1	
09/21 (Thu)		HW0
10/05 (Thu)	HW2	HW1
10/19 (Thu)	**Midterm**	
10/26 (Thu)	HW3	HW2
11/23 (Thu)	HW4	HW3
12/14 (Thu)	**Final**	HW4

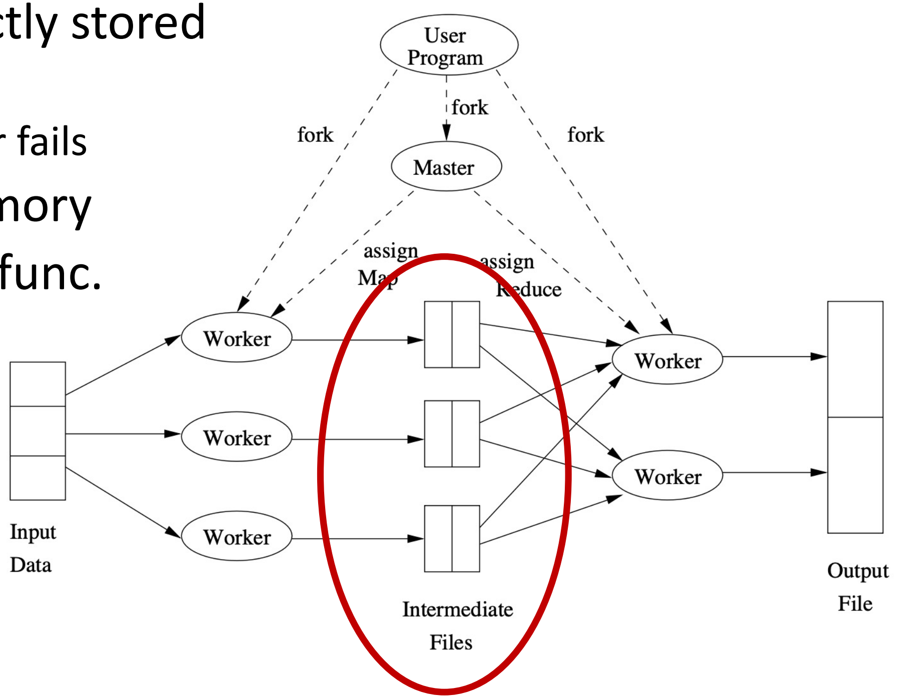
Recap: MapReduce

- **MapReduce** is suitable for large files that are rarely updated in place
  - User implements two functions: **Map** and **Reduce**
  - System handles program execution automatically coping with node failures



Recap: Intermediate Files

- MapReduce stores intermediate files on disk
  - The output of every Map task is directly stored on the local file system of the worker
    - The files are not accessible if the worker fails
  - **Pros:** Can be run with little main memory
  - **Cons:** Disk overhead after each Map func.



Recap: Coping with Node Failures

- **Master failure**
  - Entire MapReduce job must be restarted (whole restart)
- **Map worker failure** 완료되어도 map task는 새로 배정되어.
  - All Map tasks assigned this Worker have to be redone, even if completed
    - Output for Reduce tasks reside in compute node and are unavailable
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle and the Reduce task is restarted

Outline

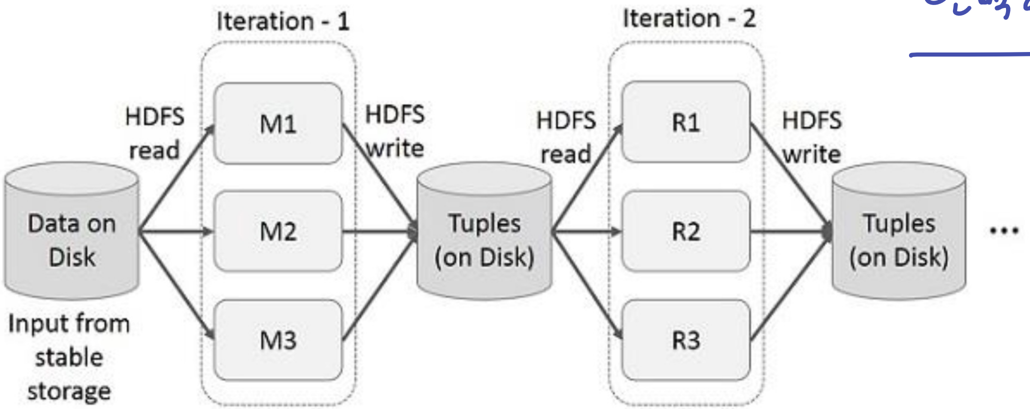
1. **Spark: Overview**
2. Spark: Operations
3. Spark: Implementation

Problems with MapReduce

increase linearly
of number of MapReduce

- For a complex job, MapReduce incurs substantial overheads due to data replication, disk I/O, and serialization
- Moreover, many problems aren't easily described as MapReduce

완벽히 묘사되지 않는다.



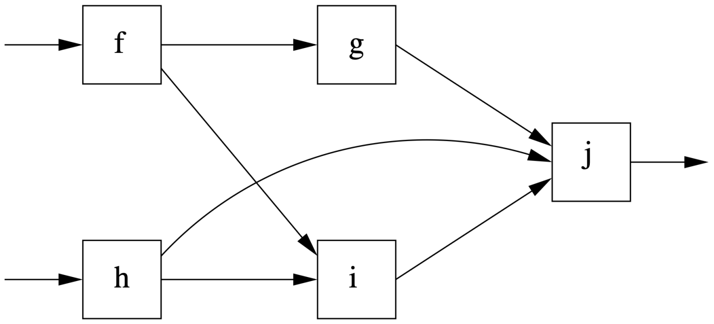
Source: Stanford CS246 (2022)

Workflow System

2.7.

- Extends MapReduce into a **directed acyclic network (DAG)** of tasks
  - Allow functions other than Map and Reduce
  - Allow any number of tasks (more than 2)
- Master controller is responsible for dividing work among tasks

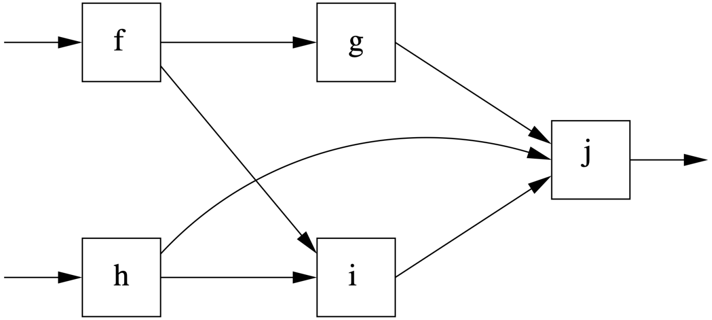
larger → more room to optimize



Source: Textbook

Blocking Property

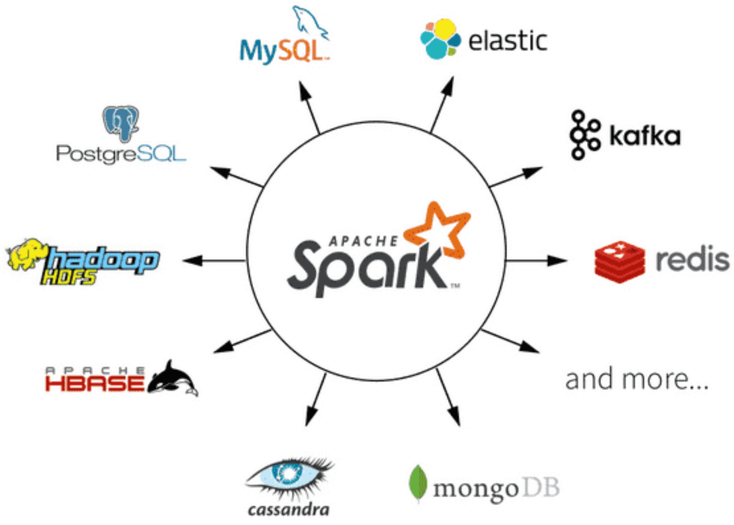
- Workflow functions only deliver output after completion
- If a task fails, no output is delivered to any successors in flow graph
- Master controller can restart failed tasks at another compute node



Source: Textbook

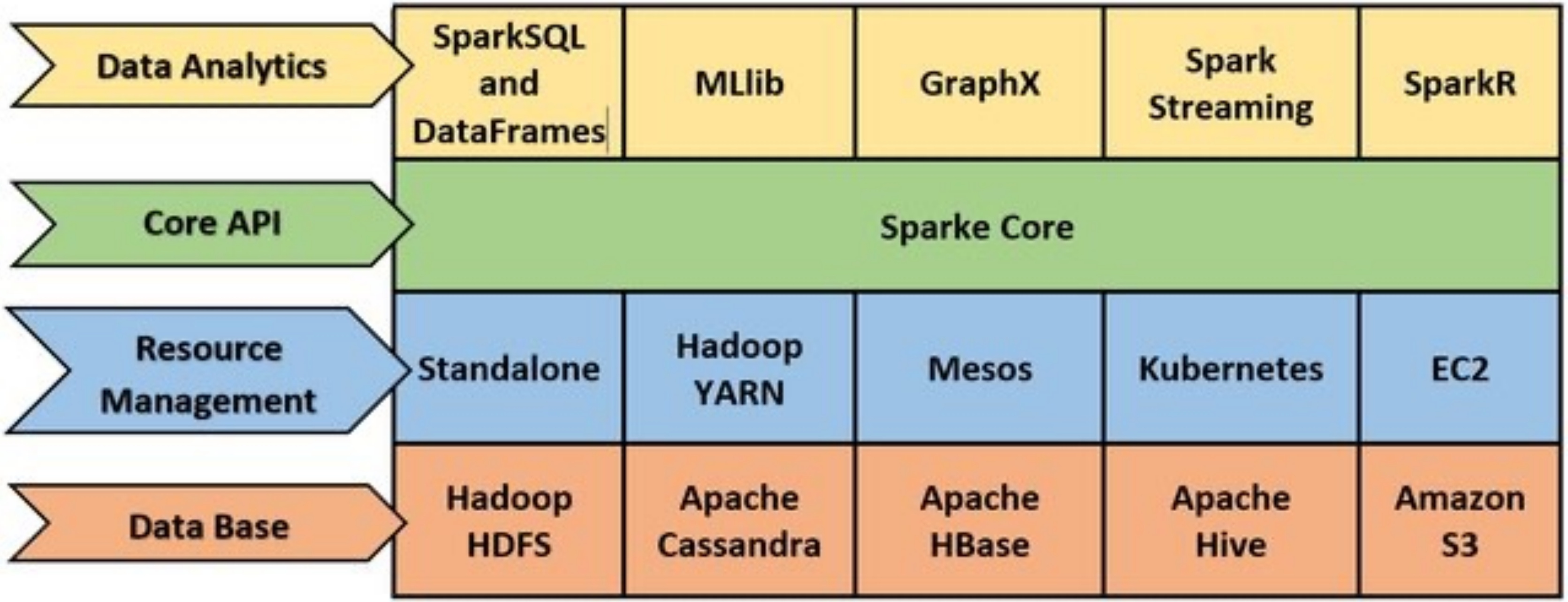
Spark: Most Popular Workflow System

- Expressive computing system, not limited to the map-reduce model
  - Developed by UCB and Databricks, now maintained by Apache
- Additions to MapReduce:
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for ML)
  - General execution graphs (DAGs)
  - Richer functions than just Map and Reduce
- Compatible with Hadoop



Source: DataBricks

Data Analytics Software Stack

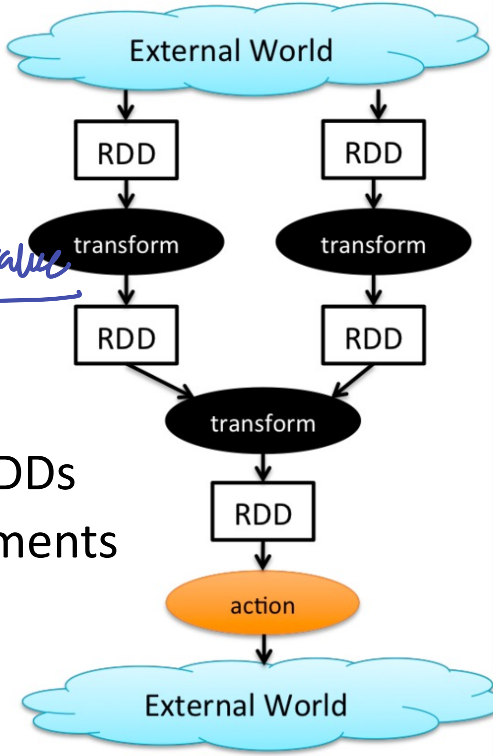


Source: Maleki et al. (2019)

Resilient Distributed Dataset (RDD)

way of representing data

- **RDDs:** Key idea / central data abstraction of Spark
  - Partitioned collection of records of one type
    - Generalizes key-value pairs in MapReduce generalization of key-value
  - Spread across the cluster (i.e., chunks) and read-only
  - Cached in main memory
  - RDDs can be created from Hadoop, or by transforming other RDDs
  - RDDs are best suited if the same operation is applied to all elements



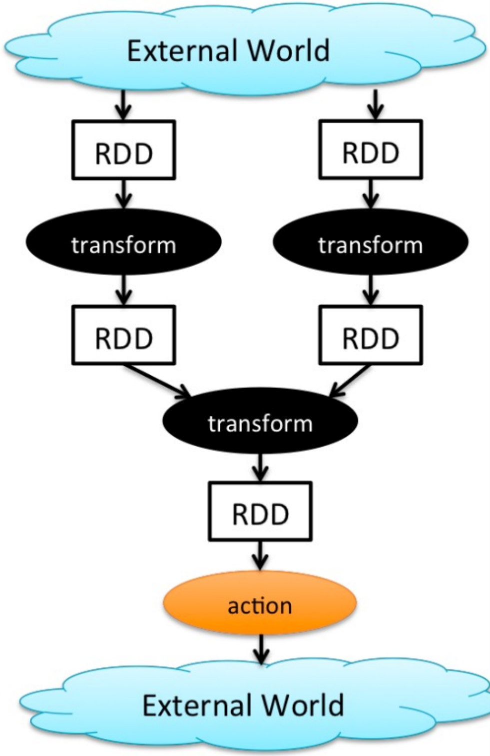
Source: Stanford CS246 (2022)

Spark Program

- A Spark program contains two types of steps:
  - **Transformations** build RDDs through deterministic operations on other RDDs
    - Include map, filter, join, union, intersection, distinct
    - **Lazy evaluation:** Nothing computed until an action requires it
  - **Actions** are used to return value or export data
    - Include count, collect, reduce, save
    - Can be applied to RDDs; force calculations and return values

user는 중간과정을 볼 수도
볼 필요도 없다.

action이 실행될 때, 필요할 때, transform이
일어난다.



Source: Stanford CS246 (2022)

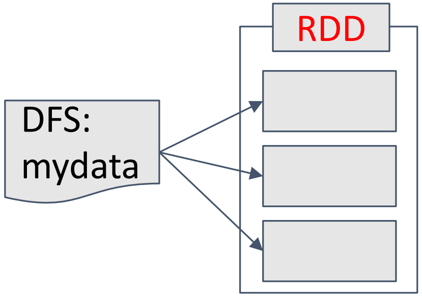
Outline

1. Spark: Overview
2. **Spark: Operations**
3. Spark: Implementation

Example: Average Word Length by Letter

spark context

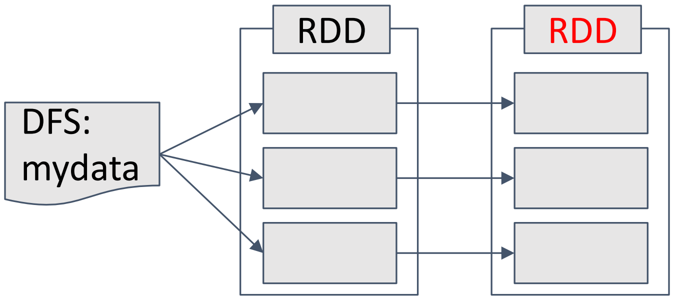
```
> avglens = sc.textFile(file)
```



Example: Average Word Length by Letter

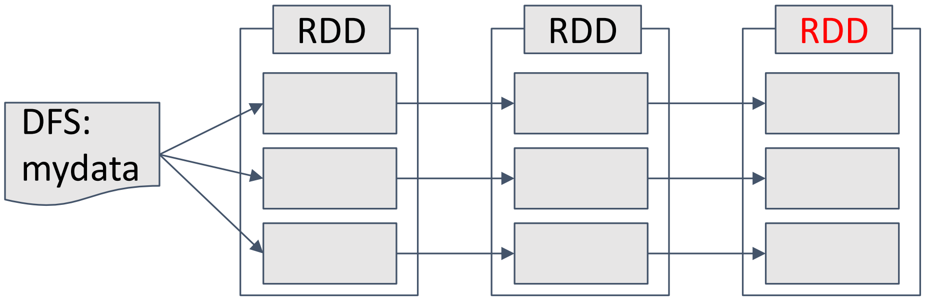
```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```

flatten



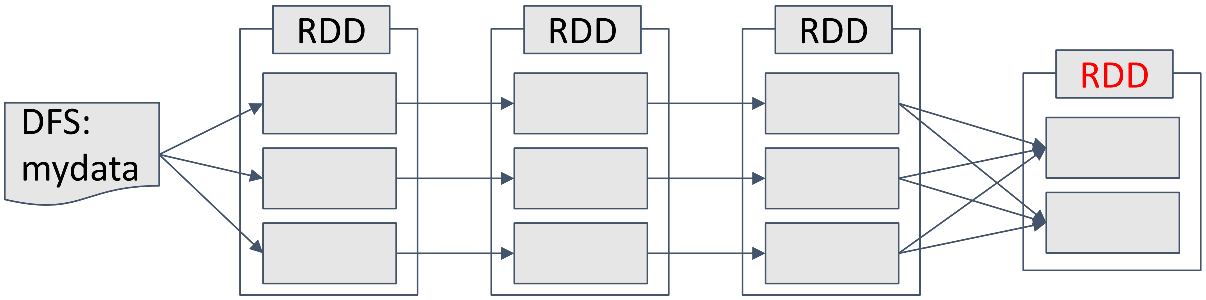
Example: Average Word Length by Letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word)))
```



Example: Average Word Length by Letter

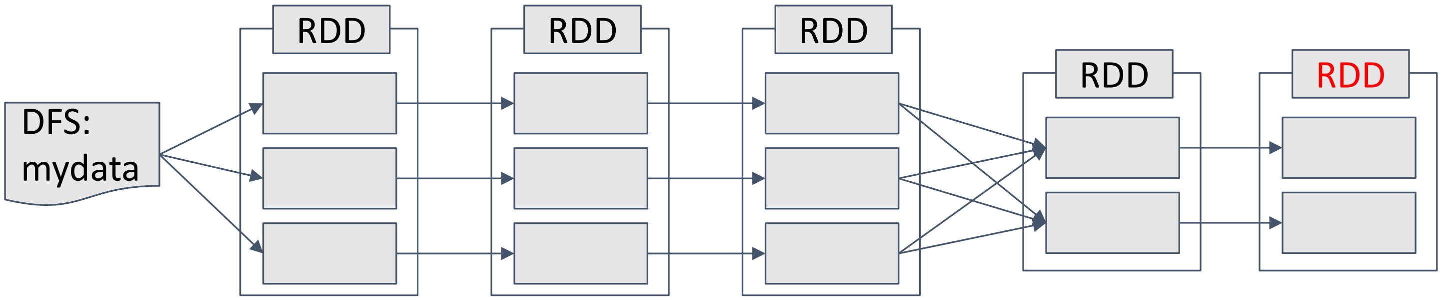
```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey()
```



Example: Average Word Length by Letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values) / len(values)))
```

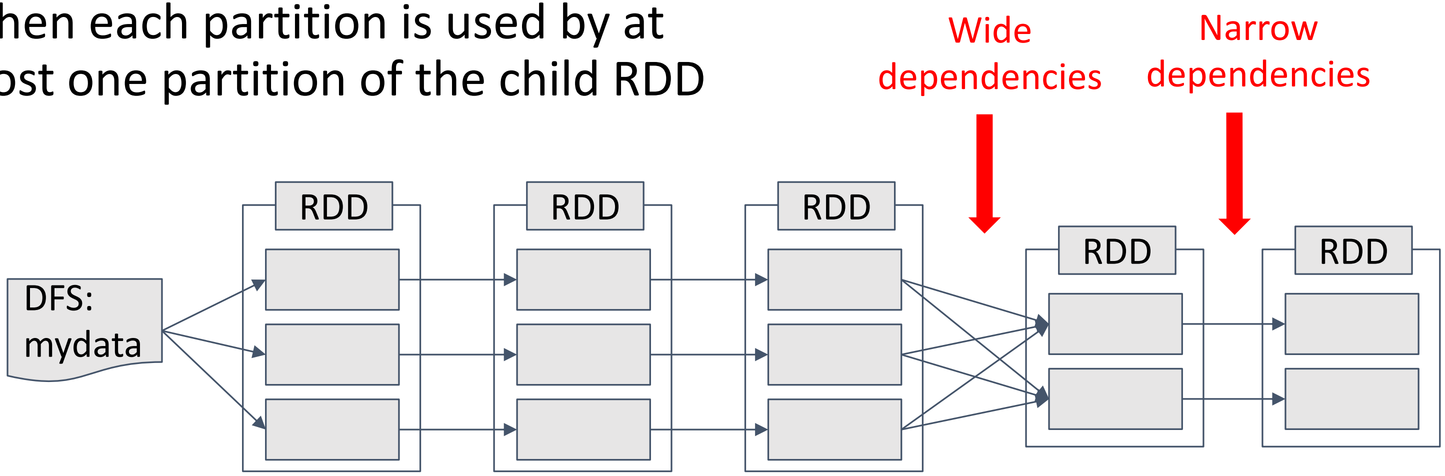
↳ average



Example: Average Word Length by Letter

- **Wide dependency** (slow; shuffling)
  - When each partition may be depended on by multiple child partitions
- **Narrow dependency** (fast; pipelining)
  - When each partition is used by at most one partition of the child RDD

Interchange



Map

- Transformation that applies a function to every element of an RDD
- Not exactly the same as Map of MapReduce
  - In MapReduce, Map is a key-value pair → a set of key-value pairs
  - In Spark, Map is any object type → exactly one object

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values) / len(values)))
```

FlatMap

- Transformation like MapReduce's Map, but no restriction on the type
- Each object maps to a list of 0 or more objects
- All the lists are then "flattened" into a single RDD of objects

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   .map(lambda word: (word[0], len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
     (k, sum(values) / len(values)))
```

Filter

- Transformation that takes a predicate that applies to the RDD object type and returns elements that satisfy the predicate

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .filter(lambda word: word not in stoplist) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values) / len(values)))
```

Reduce

- An action (not transformation) that returns a value instead of an RDD
- Takes parameter that is a function of type $(V, V) \Rightarrow V$
  - The function is repeatedly applied on pairs of RDD elements in any order
  - Each pair reduces to one element, and eventually we are left with only one
  - The function *can be* associative and commutative (e.g., addition)

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   .reduce(lambda a, b: a + b)
```

Join

- Takes two RDDs of key-value pairs having the same key types
- For each pair (k, x) and (k, y), it produces $\big(k, (x, y)\big)$
- Outputs an RDD consisting of all such objects

→ query: inner join
같은 건가?

```
> x = sc.parallelize([("a", 1), ("b", 4)])
> y = sc.parallelize([("a", 2), ("a", 3)])
> x.join(y).collect()
[('a', (1, 2)), ('a', (1, 3))]
```

GroupByKey

- Takes an RDD of key-value pairs, produces a set of key-value pairs
  - The value type for the output is a list of values of the input type
- Sorts the input RDD by key
- For each key, it produces the pair $(k, [v_1\ v_2, \dots, v_n])$

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   .map(lambda word: (word[0], len(word))) \
   .groupByKey() \
   .map(lambda (k, values): \
     (k, sum(values) / len(values)))
```

Pop Quiz

- There are many other transformations and actions on Spark
- E.g., `reduceByKey(func)` is like `groupByKey()`, but also applies a reduce function `func` of the form $(V, V) \Rightarrow V$ on the values 왜 저렇죠?
- **Problem:** Implement word count using `reduceByKey()`

```
wc = sc.textFile (경로).
        .flatMap ( lamda line: line.split()) .map ( lamda word: (word, 1) )
        .reduceByKey ( lamda. (a, b) => a+b )

→       .sort ( lamda x : x[1] )
```

아마
안될듯

Outline

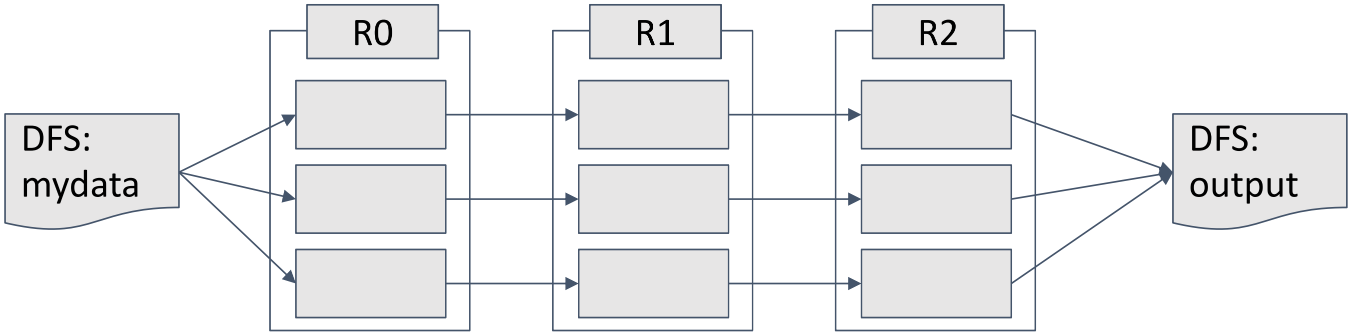
1. Spark: Overview
2. Spark: Operations
3. **<u>Spark: Implementation</u>**

Spark Implementation

- Similar to MapReduce,
  - RDD is divided into chunks, which are given to different compute nodes
  - Transformation on RDD can be performed in parallel on each of the chunks
- but two key improvements
  - Lazy evaluation of RDDs
  - Lineage for RDDs

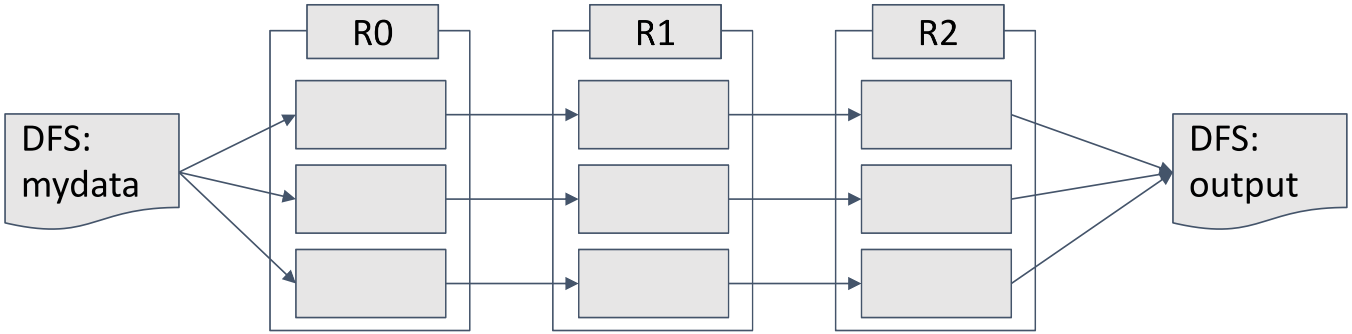
Lazy Evaluation

- Spark does not apply transformations until it is required to do so
  - E.g., storing an RDD to file system or returning a result to application
- As a result, many RDD's are not constructed all at once
  - An RDD chunk can be used again to apply another transformation
  - **Benefit:** This RDD is never stored, never transmitted to other nodes



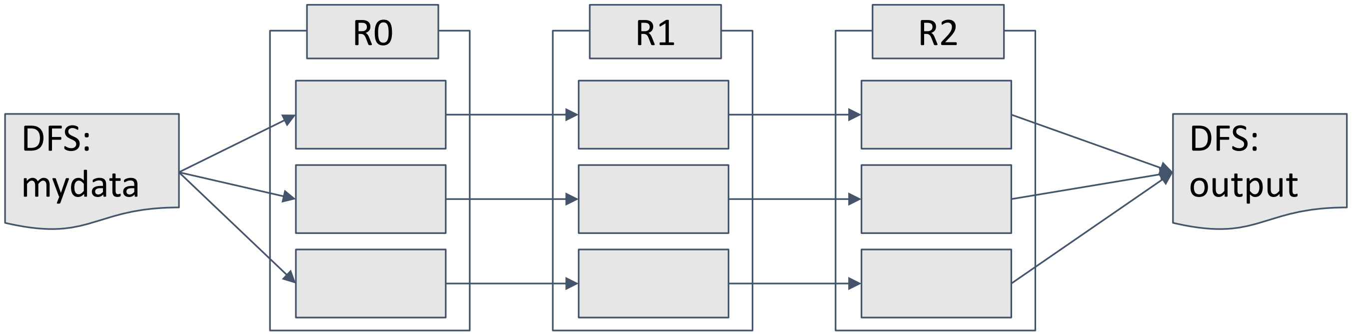
Lazy Evaluation Example

- Count words in a document that are not stop words
  - Apply `Flatmap` to input RDD R0 to create $(w, 1)$ pairs in R1
  - Apply `Filter` to each chunk in R1 to produce R2
  - Only if R2 is stored in DFS (action) then R1 and R2 are actually produced on same compute node and subsequently dropped after being used



Resilience of RDDs

- Spark records the **lineage** of every RDD, which is used to re-create it in case of the node failure
  - If R2 is lost, reconstruct from R1
  - If R1 is lost, reconstruct from R0
  - If R0 is lost, reconstruct from file system



Spark Programming Guide and Paper

- To learn more about writing Spark applications, please read the Spark programming guide: https://spark.apache.org/docs/latest/rdd-programming-guide.html
- (Optional) More technical details of Spark in this paper: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Spark vs. MapReduce / Hadoop

	MapReduce / Hadoop	Spark
Speed	Faster than traditional systems	Much faster (100x) than MapReduce
Written in	Java	Scala
Data Processing	Batch processing	Batch / real-time / iterative / interactive / graph
Ease of Use	Complex and lengthy	Compact and easier than MapReduce
Caching	No caching	Caches data in memory, which enhances system performance

When to Use What

- MapReduce / Hadoop
  - Linear processing of large datasets
  - No intermediate results required
- Spark
  - Fast and interactive data processing
  - Joining datasets
  - Graph processing
  - Iterative jobs
  - Real-time processing
  - Machine learning

Summary

1. Spark: Overview
   - Resilient Distributed Dataset (RDD)
2. Spark: Operations
   - Map, FlatMap, Filter, Reduce, Join, etc.
3. Spark: Implementation
   - Lazy evaluation
   - Lineage of RDDs