

PySpark를 이용한 데이터 분석

: 스파크 자료형의 이해

2022년 9월 22일

데이터분석에서 활용 가능한 PySpark에 대한 개념 및 강력한 기능들을 습득하고
데이터 전처리 및 분석 알고리즘을 실습하는 과정

시작하기 앞서, 기본적으로 이해하면 좋은 지식



엔지니어



데이터 분석가

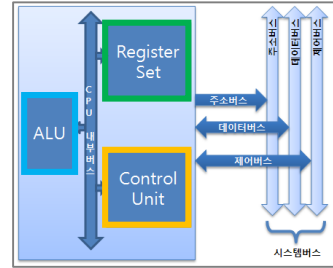


CPU - 메모리 (SRAM/DRAM) - 디스크 (HDD/SDD)

** RAM : Random Access Memory

** ROM : Read Only Memory

* CPU : 중앙 처리 장치



* 산술/논리 연산 장치(ALU, Arithmetic and logical unit)

➢ 제어 장치의 명령에 따라 실제 연산을 수행하는 장치

* 제어 장치

➢ 컴퓨터에 있는 모든 장치들의 동작을 지시하고 제어

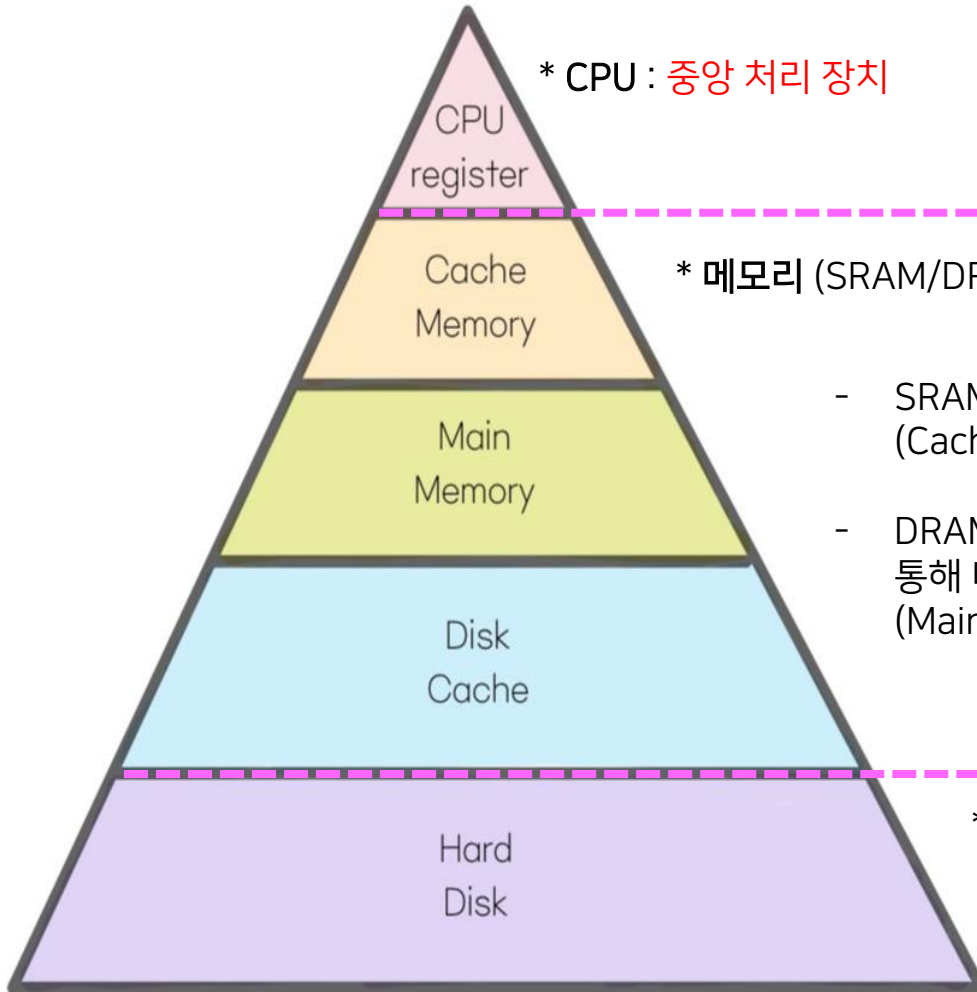
* 레지스터

➢ CPU 내부에서 처리할 명령이나 연산의 결과, 주소 등을 일시적으로 저장

* 메모리 (SRAM/DRAM) : 주기억장치 & 휘발성 메모리 & 저장량

- SRAM은 플립플롭(Flip-Flop) 이라는 기억능력을 가진 논리 회로 (Cache Memory)
- DRAM은 축전기(Capacitor)를 이용하여 만들어지며 축전기의 충전상태를 통해 데이터를 기록하고 저장 (Main Memory)

* 디스크 (HDD/SDD) : 보조기억장치 & 비휘발성 메모리 & 대용량

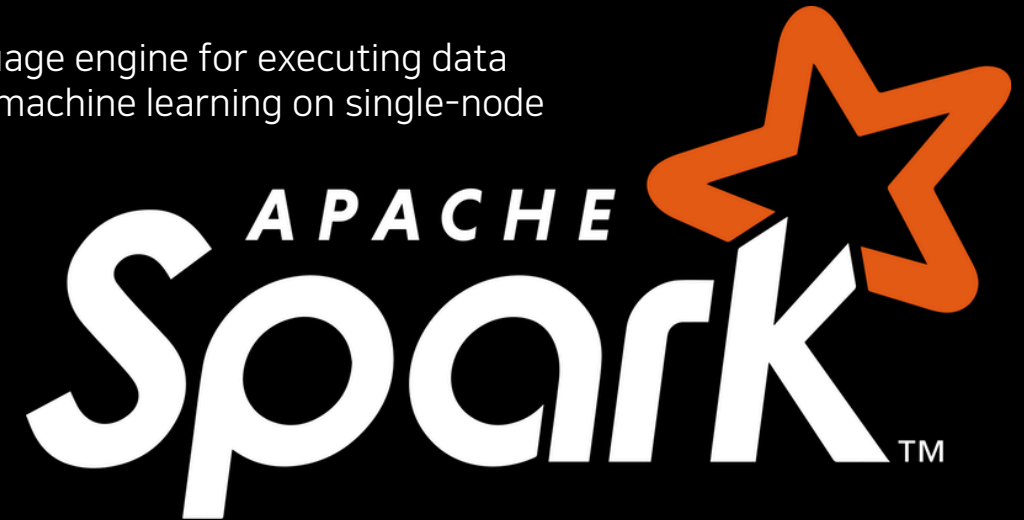


△ 메모리의 계층 구조

관련 링크 : <https://information-factory.tistory.com/270>

오픈 소스 클러스터 컴퓨팅 프레임워크

Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.



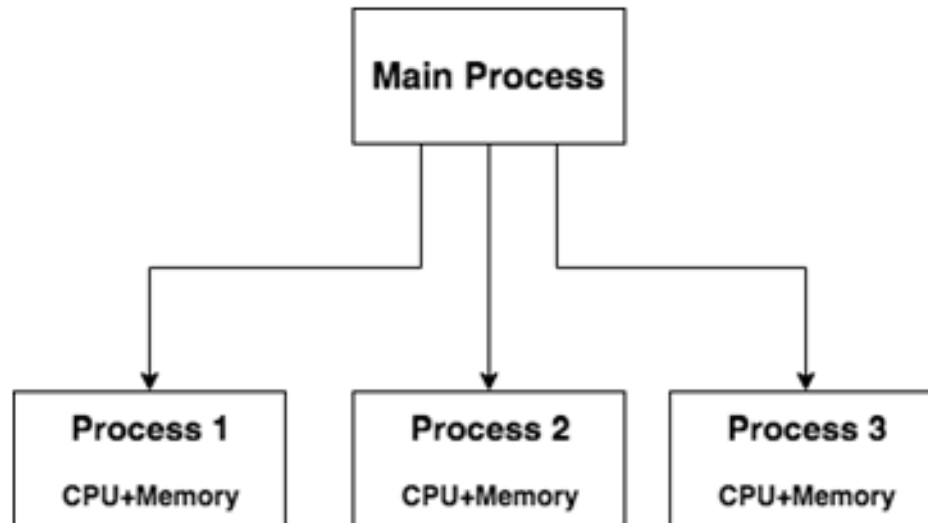
일시적으로 반짝거렸다 사라져버리는 불꽃

시작하기 앞서, 기본적으로 이해하면 좋은 지식

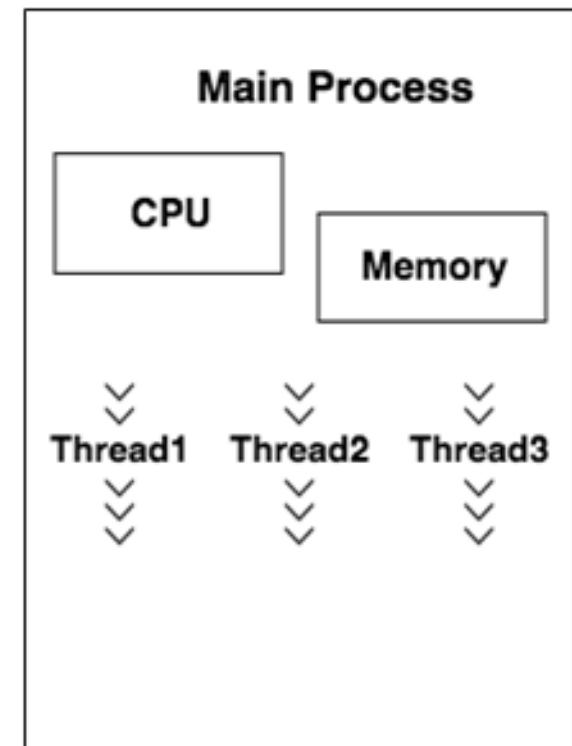


Q: 기존에도 병렬처리 라이브러리가 있는데, 분산처리(클러스터 컴퓨팅) 스파크를 꼭 도입해야 하나?

Multiprocessing



Multithreading





Q: 스파크는 왜 나왔을까?

A: 아파치 스파크는 **하둡의 맵리듀스 구현에 대한 대안**으로 만들어져 매우 효율적이다.

jeffreyAven, 『Data Analytics with SPARK Using PYTHON』(서울: 에이콘출판주식회사, 2019), 31



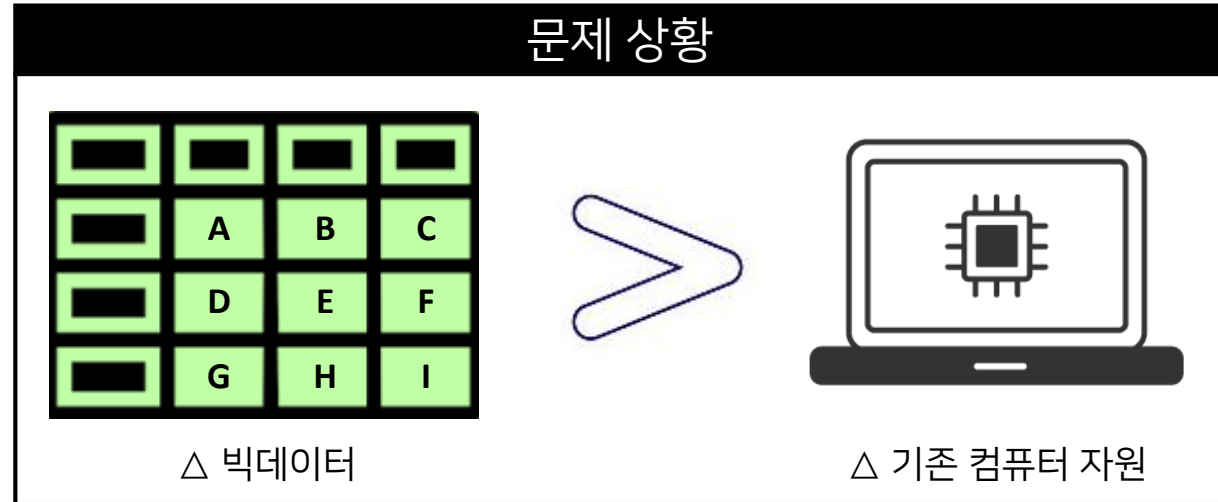
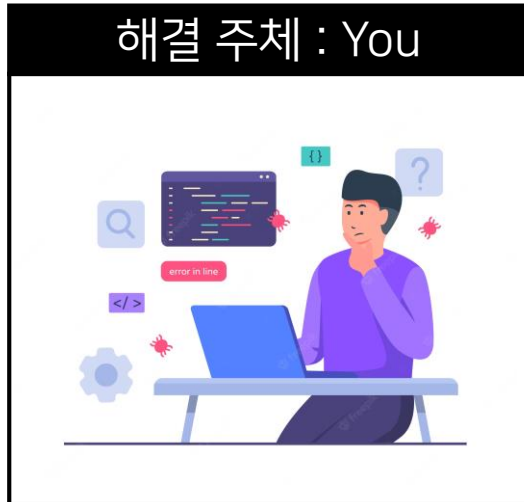
[More]

Q1: **하둡**이 무엇일까? 어떤 역할을 할까? 왜 나왔을까?

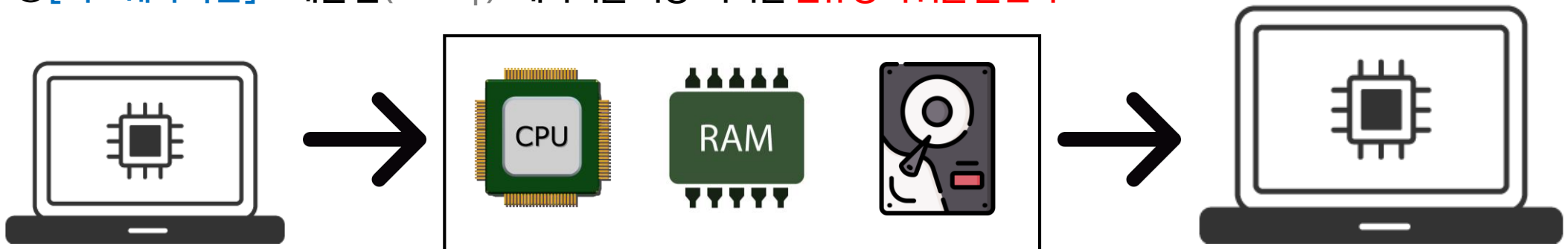
Q2: 스파크는 하둡의 **맵리듀스의 어떤 점을 보완**해줄 수 있을까?



[More] Q1: 하둡이 무엇일까? 어떤 역할을 할까? 왜 나왔을까?



① [하드웨어 측면] 스케일 업 (Scale up): 데이터를 저장·처리할 **컴퓨팅 파워를 늘린다**



한계: 하드웨어 허용 범위 내에서만 확장이 가능하기 때문에 그 이상 업그레이드 불가

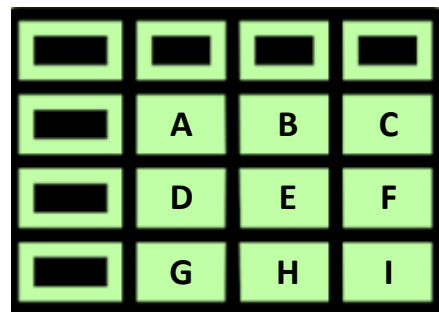
시작하기 앞서, 기본적으로 이해하면 좋은 지식



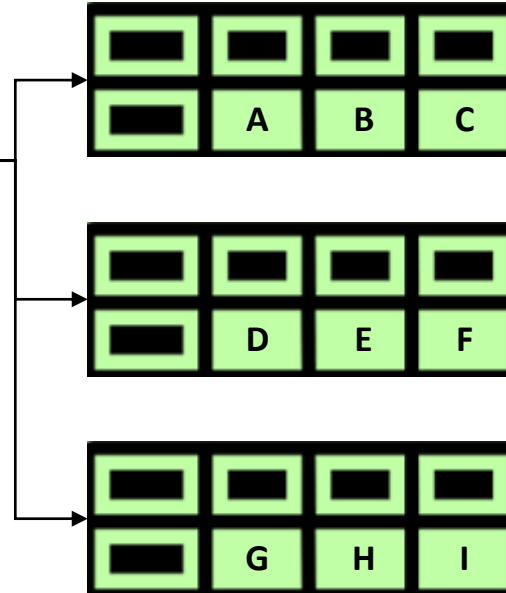
백지장도 맞들면 낫다

[More] Q1 : 하둡이 무엇일까? 어떤 역할을 할까? 왜 나왔을까?

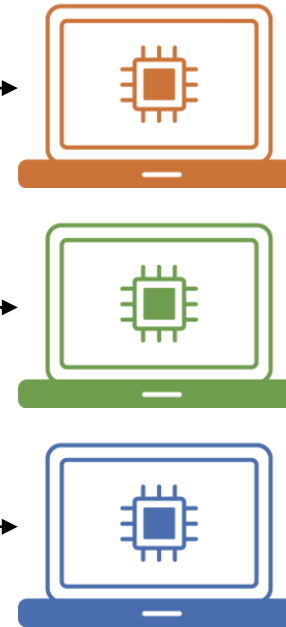
② [소프트웨어 측면] 스케일 아웃 (Scale Out): 데이터를 분할하여 여러 컴퓨터에 나누어 저장·처리한다.



△ 빅데이터

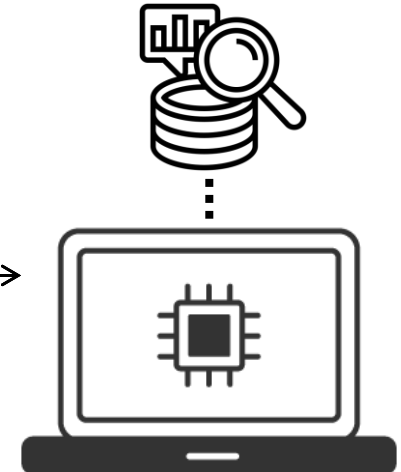


△ 분할 데이터
(block)



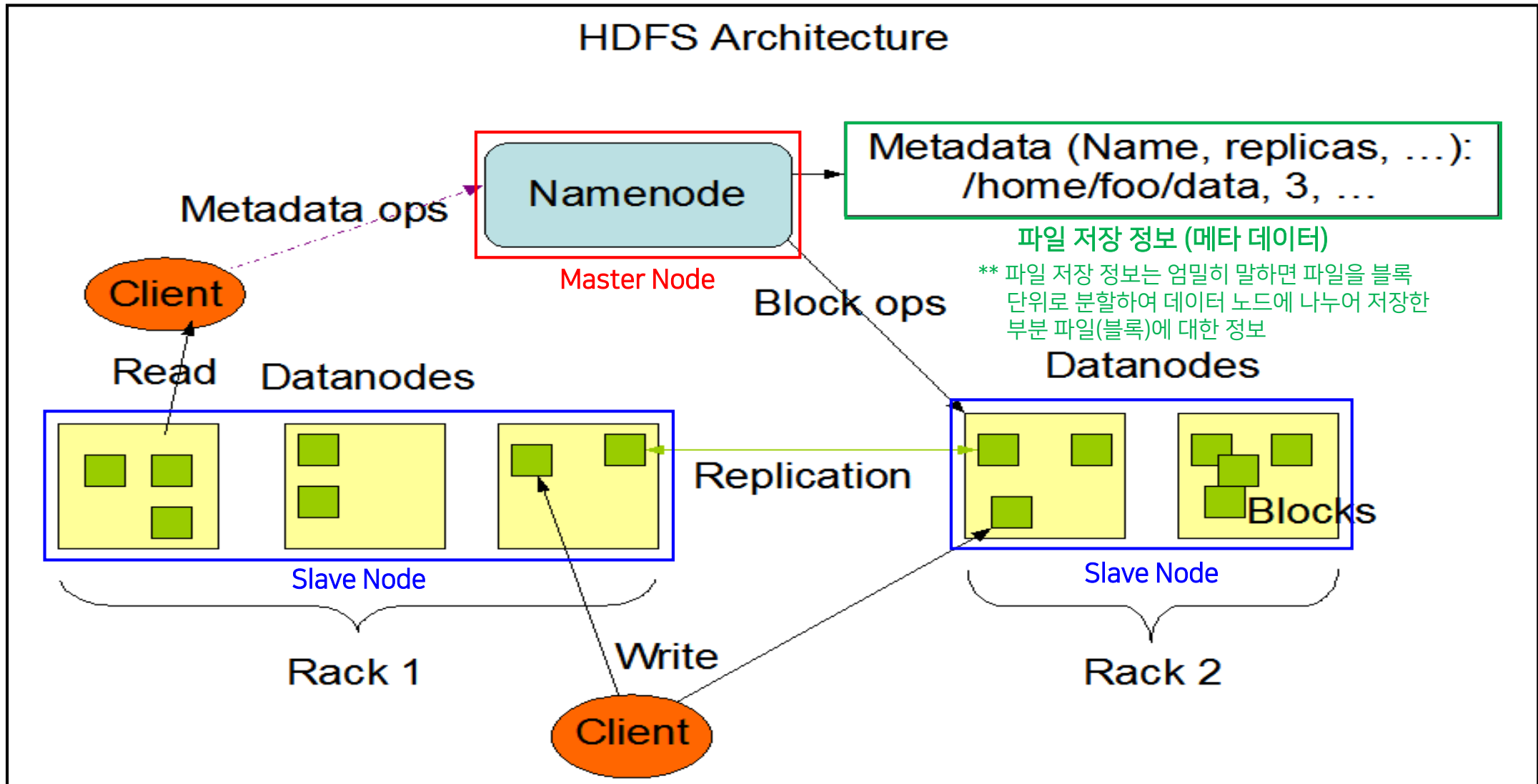
△ 데이터 노드
(Data Node, Slave Node)

파일 저장 정보 (Meta Data)



△ 네임 노드
(Name Node, Master Node)

시작하기 앞서, 기본적으로 이해하면 좋은 지식



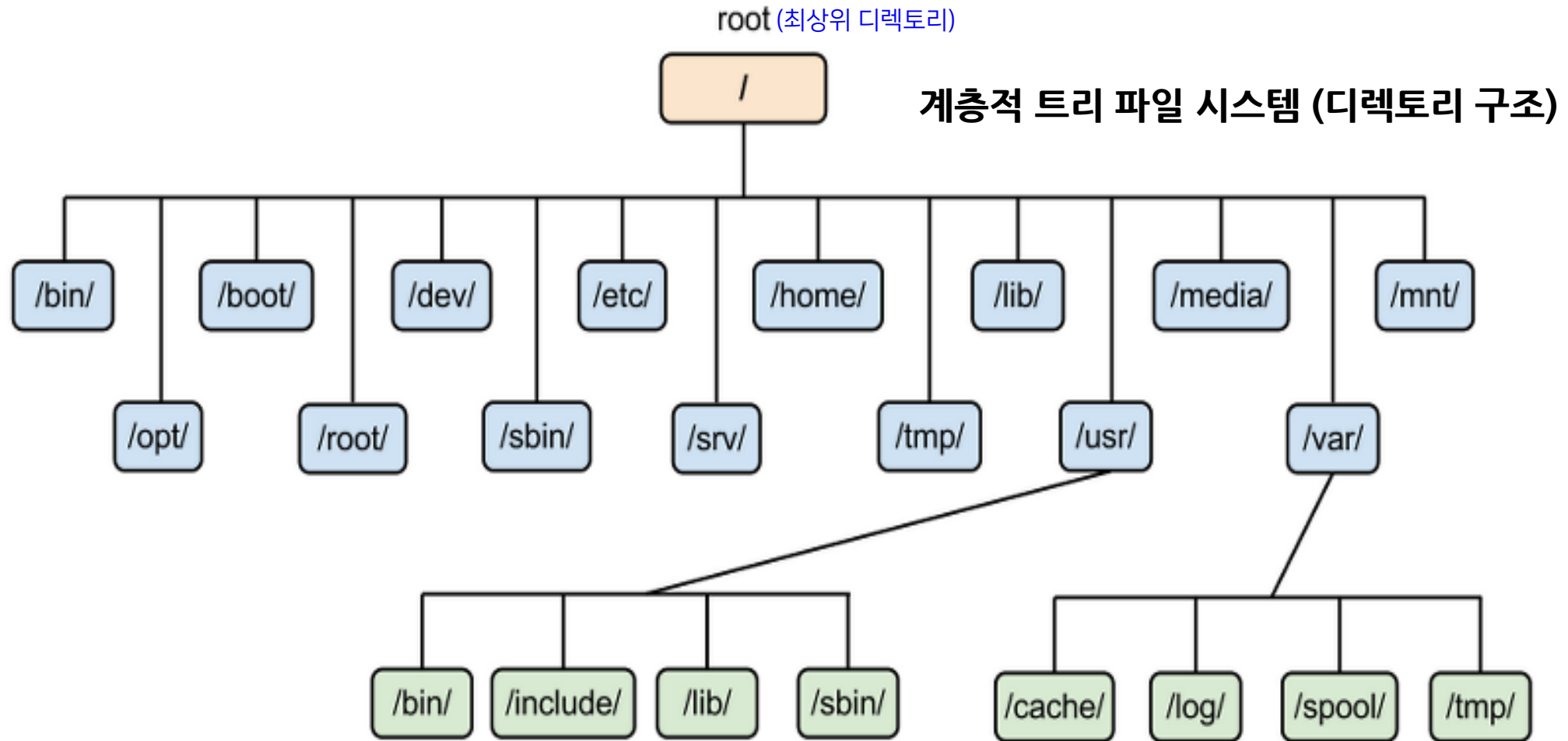
△ HDFS (Hadoop Distributed File System, 하둡 분산 파일 시스템) 아키텍처

<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>

시작하기 앞서, 기본적으로 이해하면 좋은 지식



리눅스 : 파일 스토리지 (계층 구조)



리눅스의 파일 시스템은 최상위 디렉토리부터 아래로 진행되는 트리 형식의 구조를 가지고 있다

시작하기 앞서, 기본적으로 이해하면 좋은 지식



리눅스 : 파일 스토리지 (데이터 저장 방법)

Directory inode (128B)

Type	Mode
User ID	Group ID
File size	# blocks
# links	Flags
Timestamps (x3)	
Direct blocks (x12)	
Single indirect	
Double indirect	
Triple indirect	

Directory block

.	inode #
..	inode #
passwd	inode #
fstab	inode #
...	...

Indirect block

Direct blocks (x512)

** 파일과 디렉토리는 컴퓨터가 전원이 꺼진 상태에도 데이터가 사라지지 않습니다. 즉, 저장 정보를 Disk 일부 영역에 별도로 저장하고 있습니다.

[CLI Command] "df -i" 명령어를 치면 확인 할 수 있습니다

File inode (128B)

Type	Mode
User ID	Group ID
File size	# blocks
# links	Flags
Timestamps (x3)	
Direct blocks (x12)	
Single indirect	
Double indirect	
Triple indirect	

File data block

Data

Block # of
block with
512 double
indirect
entries

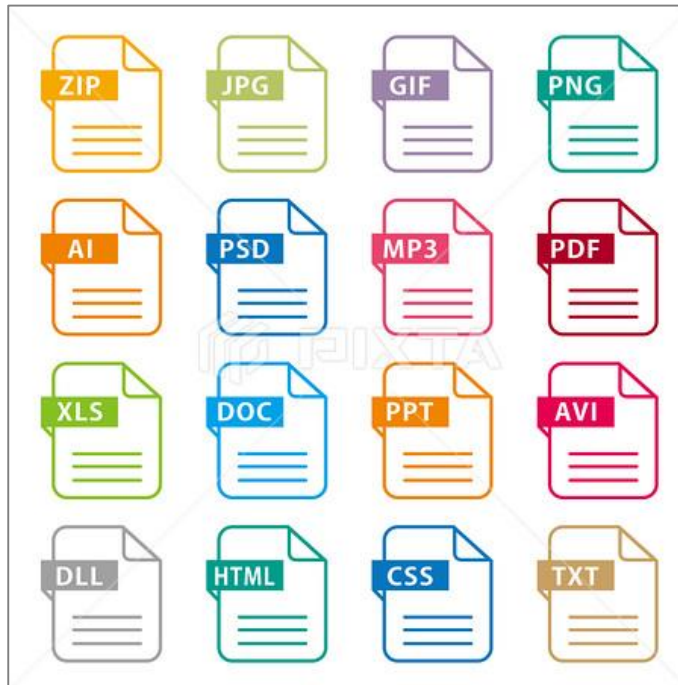
Block # of
block with
512 single
indirect
entries

Block #s of
more
directory
blocks

리눅스의 파일 시스템은 inode를 통해 디스크에 파일과 디렉토리의 정보를 저장하고 있다

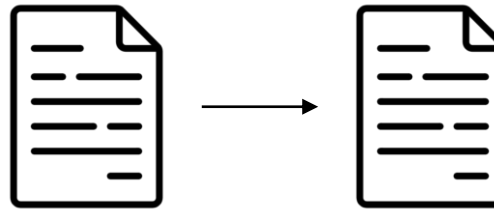
발표 후 Q&A 관련 내용

파일 스토리지 vs 블록 스토리지 vs 오브젝트 스토리지



△ FILE

***파일 스토리지**: 파일을 분할 하지 않고 그대로 저장



***블록 스토리지**: 파일을 일정 크기로 분할한 블록을 여러 노드에 나누어 저장



HDFS ->

데이터 노드에 저장

네임 노드에 저장

```
drwxr-xr-x 17 hpedf hpedf 16 Mar 15 2022 mapr.daegu.go.kr
```

** HDFS를 OS에서 봤을 때는 하나의 폴더처럼 보인다

***오브젝트 스토리지**: 오브젝트라 불리는 독립된 유닛에 데이터를 저장



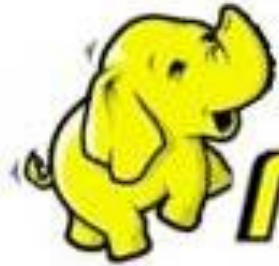
Amazon S3

** 오브젝트 스토리지에 대해서는 AWS S3를 참고하면서 같이 공부합시다 ☺
저자가 오브젝트(객체) 스토리지의 구성도에 대해 상세히 아는 내용이 아니기 때문에 건너 띄도록 하겠습니다.

시작하기 앞서, 기본적으로 이해하면 좋은 지식



[More] Q1 :하둡이 무엇일까? 어떤 역할을 할까? 왜 나왔을까?



Map Reduce

자료형 (Map)

줄이다

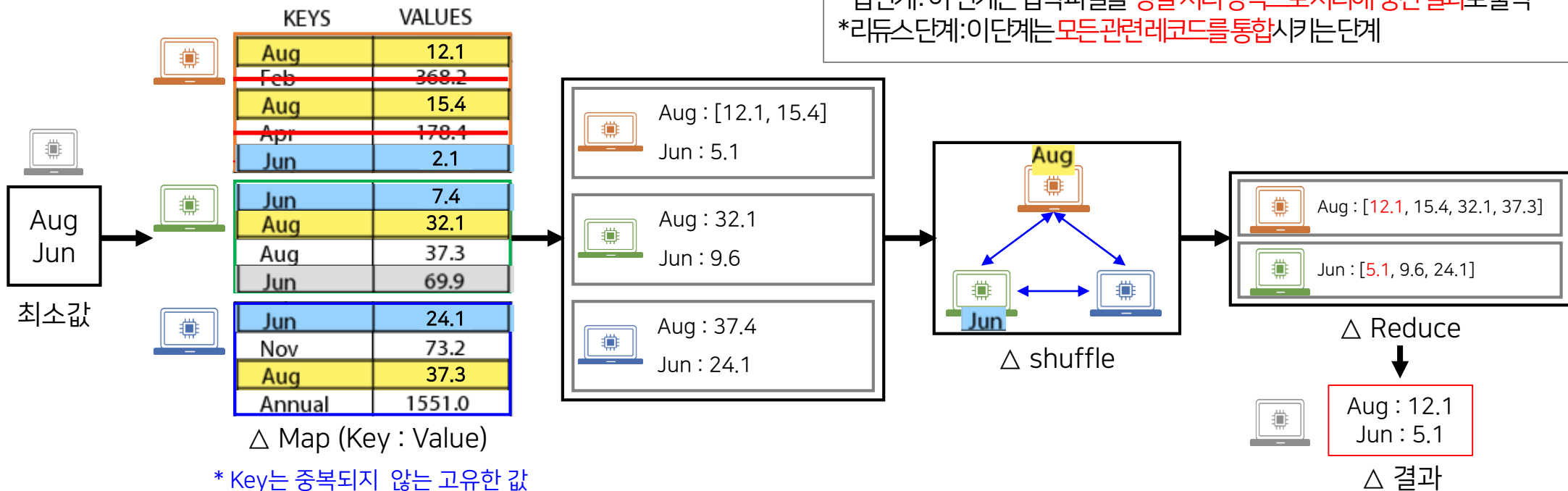
<https://www.databricks.com/kr/glossary/mapreduce>

MapReduce란 무엇입니까?

MapReduce is a **Java-based, distributed execution framework** within the **apache Hadoop Ecosystem**.

*맵단계: 이 단계는 입력파일을 **병렬 처리방식으로** 처리해 **중간 결과**로 출력

*리듀스단계: 이 단계는 **모든 관련레코드를 통합**시키는 단계

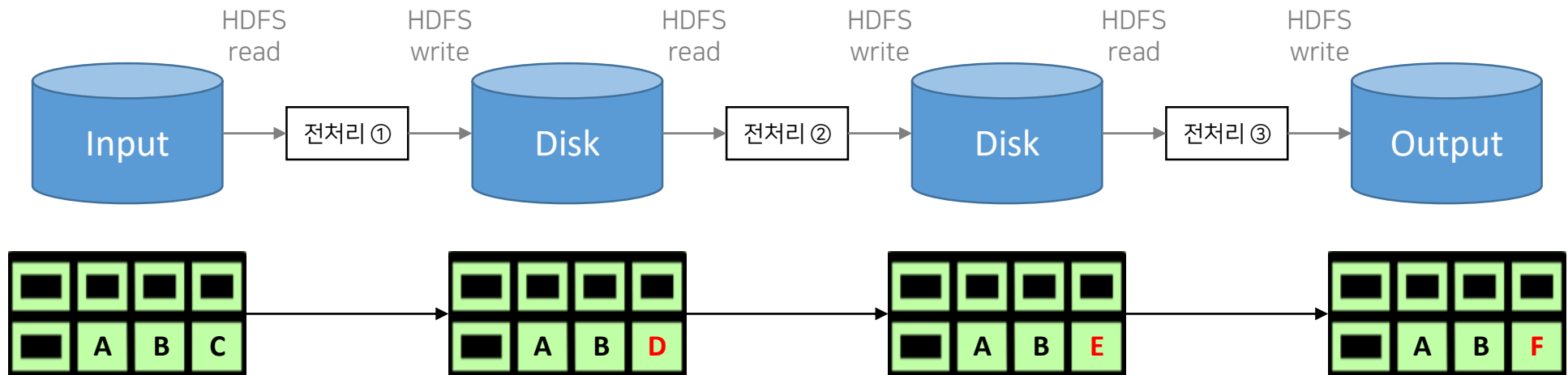


시작하기 앞서, 기본적으로 이해하면 좋은 지식



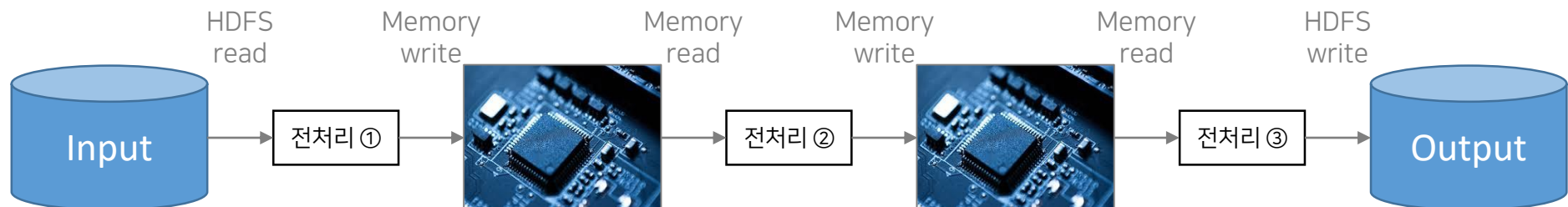
[More] Q2: 스파크는 하둡의 맵리듀스의 어떤 점을 보완해줄 수 있을까?

❖ HDFS(Disk)기반데이터read/write로 성능이 느림



HDFS의 MapReduce는 전처리의 중간 결과물을 메모리가 아닌 디스크에 쓰기 때문에, 지속적인 전처리는 I/O의 영향을 받아 속도가 느리다

❖ Memory기반데이터read/write로 성능이 빠름



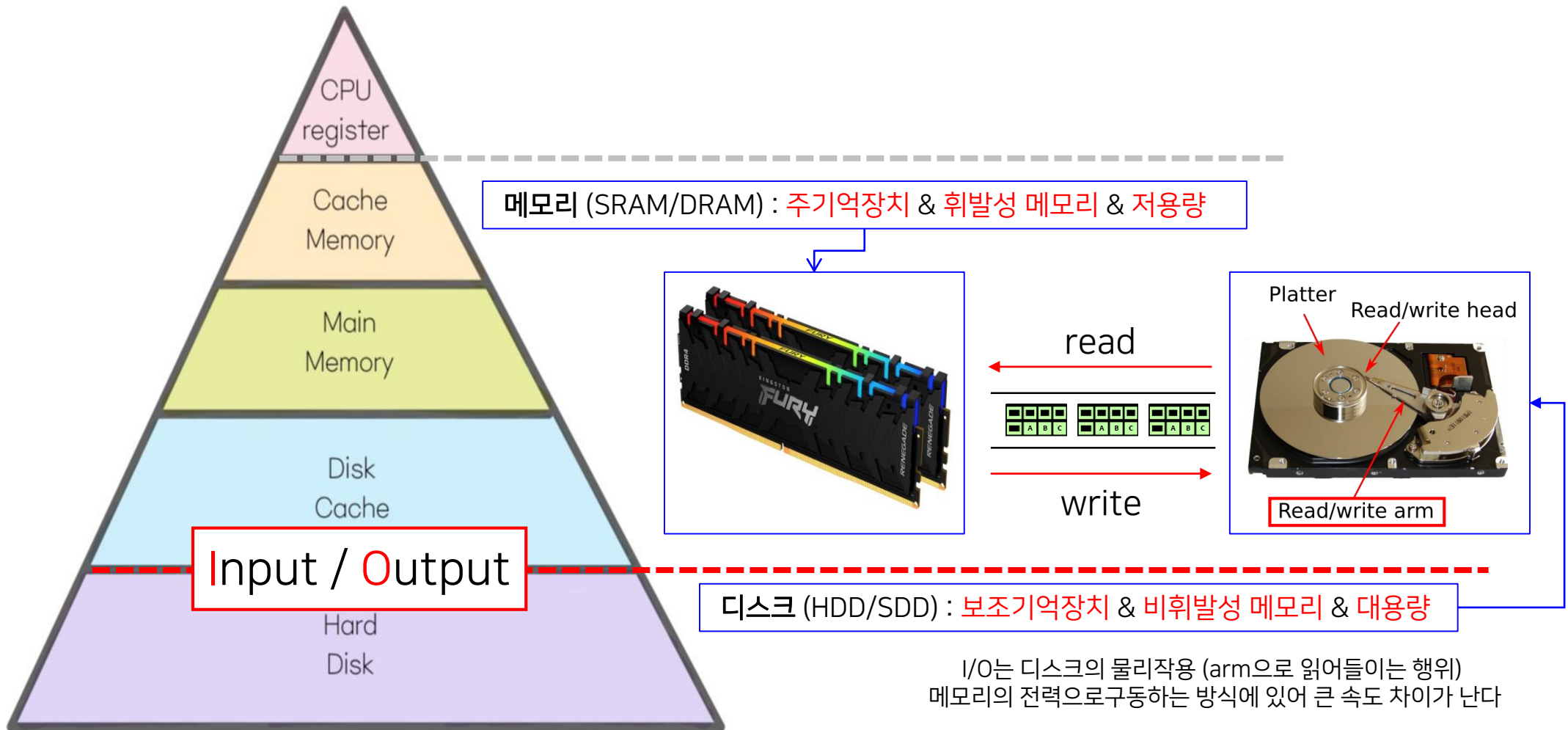
Spark는 전처리의 중간 결과물을 메모리에 저장하여 재사용한다

시작하기 앞서, 기본적으로 이해하면 좋은 지식



메모리와 디스크 사이의 I/O

** RAM : Random Access Memory
** ROM : Read Only Memory



△ 메모리의 계층 구조

Part 2. Spark 자료형과 특징

왜 자료형을 이해해야 하는 가?

1. 메모리 공간의 적절한 사용을 위해 다양한 크기의 자료형이 필요하다.
2. 데이터의 표현방식이 다르므로 둘 이상의 자료형이 필요하다.

스파크의 3가지 자료형

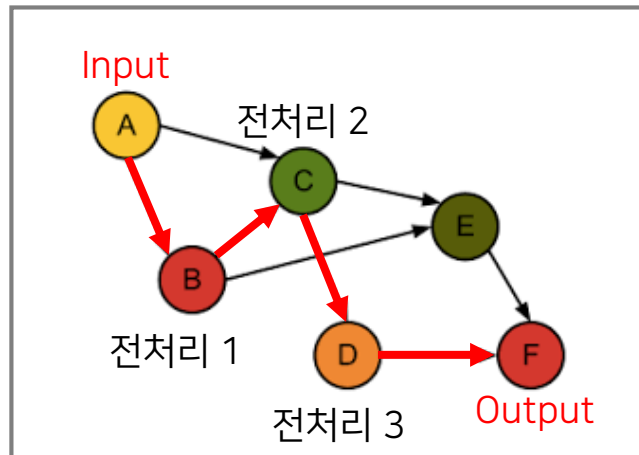
1. RDD
2. DataFrame
3. Dataset



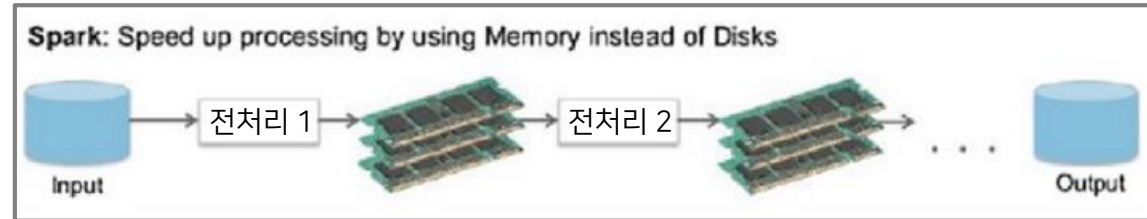
RDD

* 복원 (Resilient) : RDD는 탄력적

- 노드가 손실될 경우, 데이터 집합을 재구성 가능
각 RDD 리니지(Lineage, RDD 를 만드는 일련의 단계)를 알고 있음



△ DAG (방향성 비순환 그래프)



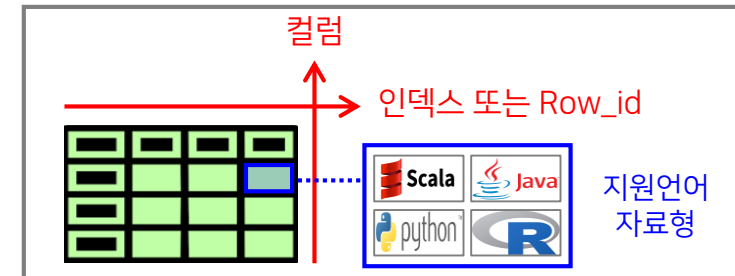
△ 스파크의 메모리 저장

* 분산 (Distributed) : RDD는 분산

- 파티션을 하나 또는 여러 개로 나눔

* 데이터 집합 (Dataset) : RDD는 레코드로 구성된 데이터 집합

- 관계형 데이터베이스의 테이블과 유사한 필드 모음



△ 스파크 저장



■ RDD vs DataFrame

	RDD	DataFrame
도입 버전	Spark 1.0	Spark 1.3

<https://spark.apache.org/downloads.html>

스파크 다운로드 경로

Download Apache Spark™

1. Choose a Spark release: **3.3.0 (Jun 16 2022) ▼** 최신 버전 (3.3.0)

2. Choose a package type: Pre-built for Apache Hadoop 3.3 and later ▼

3. Download Spark: [spark-3.3.0-bin-hadoop3.tgz](#)

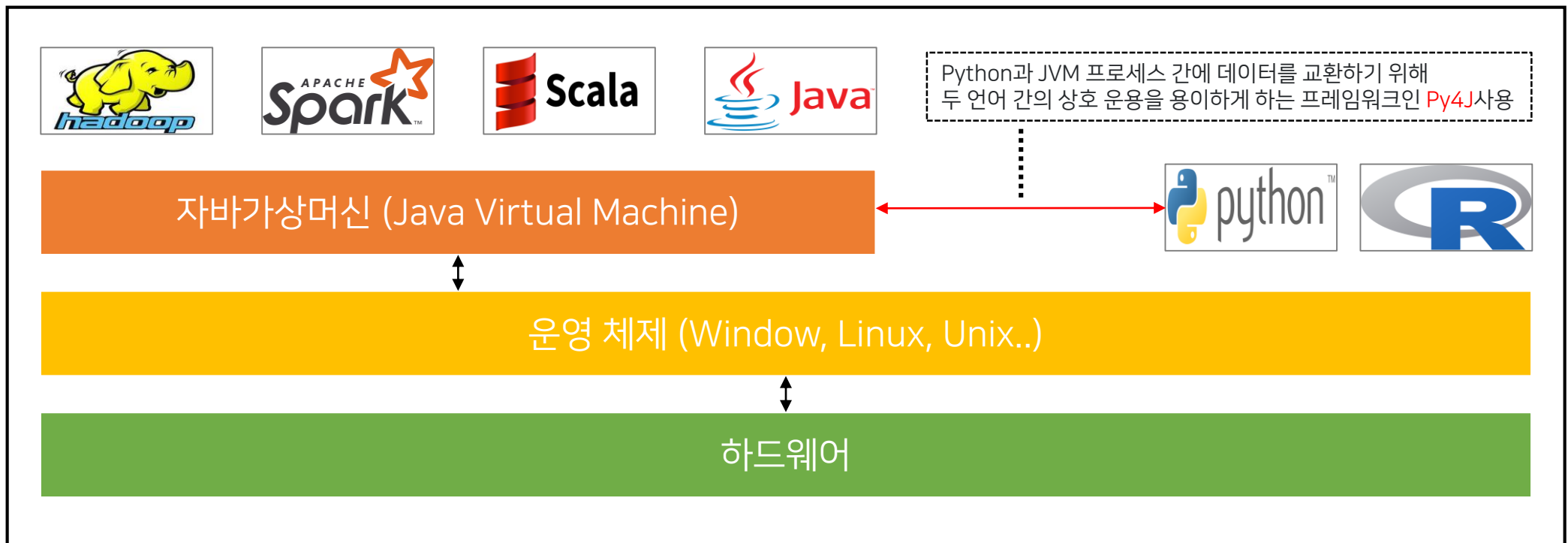
4. Verify this release using the 3.3.0 [signatures](#), [checksums](#) and [project release KEYS](#) by following these [procedures](#).

Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.






■ RDD vs DataFrame

	RDD	DataFrame
도입 버전	Spark 1.0	Spark 1.3
지원 언어	스칼라, 자바, R, 파이썬	





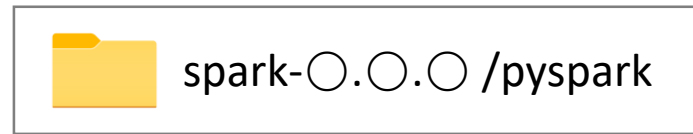
■ RDD vs DataFrame

	RDD	DataFrame
Introduction	Spark 1.0	Spark 1.3
supported languages	Scala, Java, R, python	
Library Path (pyspark-2.4.4)	 pysaprk ₩ rdd.py	 pyspark ₩  sql ₩ dataframe.py

Spark 자료형 비교



■ pyspark 디렉토리 구조



기본 모듈

```
accumulators.py 데이터 누적 및 집계
broadcast.py 모든 노드에 데이터 공유 (공유변수)
cloudpickle.py
conf.py 스파크 애플리케이션 구성
context.py 스파크 builder (RDD)
daemon.py 백그라운드 실행 (데몬)
files.py
find_spark_home.py OS 환경변수 탐색
globals.py
heapq3.py 데이터 정렬 연산
__init__.py
java_gateway.py
join.py

rdd.py
rddsampler.py
resultiterable.py
serializers.py 직렬화
shell.py
shuffle.py 셔플
```

* RDD (Resilient Distributed Dataset)

- [spark.RDD](#) 자료형 라이브러리

- 여러 분산 노드에 걸쳐 저장되는,
변경이 불가능한 데이터의 집합

sql [sql 라이브러리]

```
catalog.py
column.py
conf.py Spark 환경 구성
context.py SQL Context, Hive Context
dataframe.py
functions.py
group.py
__init__.py
mapreduce.py
pycache
catalog.cpython-37.pyc
column.cpython-37.pyc
conf.cpython-37.pyc
context.cpython-37.pyc
dataframe.cpython-37.pyc
functions.cpython-37.pyc
group.cpython-37.pyc
__init__.cpython-37.pyc
mapreduce.cpython-37.pyc
readwriter.cpython-37.pyc
session.cpython-37.pyc
streaming.cpython-37.pyc
types.cpython-37.pyc
udf.cpython-37.pyc
utils.cpython-37.pyc
window.cpython-37.pyc
readwriter.py
session.py 스파크 builder (DataFrame)
streaming.py
tests.py
types.py 스파크 DataFrame 컬럼 스키마
udf.py
utils.py
window.py
```

* Dataframe

- [spark.sql DataFrame](#) 자료형 라이브러리

* session

- spark.sql 을 위한 통합형 builder

* types

- Spark.sql DataFrame에 사용되는 데이터 타입 정의

ml / mllib [머신러닝 라이브러리]

```
base.py ...
classification.py 분류 모델
clustering.py 군집 모델
common.py
evaluation.py 모델 평가
feature.py 피쳐 엔지니어링 (PCA, Tokenizer...)
fpm.py
image.py
__init__.py
linalg 선형 모델
__init__.py
param ...
__init__.py
shared_params_code_gen.py
shared.py
pipeline.py 전처리 파이프 라인 (stage 병합)
recommendation.py 추천 모델
regression.py 회귀 모델
stat.py EDA (상관관계, 검정 - 카이제곱, 적합도...)
tests.py
tuning.py
util.py
wrapper.py
```

```
classification.py
clustering.py
common.py
evaluation.py
feature.py
fpm.py
__init__.py
linalg
distributed.py
__init__.py
random.py
recommendation.py
regression.py
stat
distribution.py
__init__.py
KernelDensity.py
statistics.py
test.py
tests.py
tree.py
util.py
```




streaming [스트리밍 라이브러리]

```
context.py
dstream.py
flume.py
__init__.py
kafka010.py
kafka08.py
kafka09.py
kinesis.py
listener.py
tests.py
util.py
```

다양한 데이터 소스(HDFS, kafka..)로부터
데이터를 받아서 실시간 스트리밍 처리를
할 수 있는 라이브러리



■ RDD vs DataFrame

	RDD	DataFrame
Introduction	Spark 1.0	Spark 1.3
supported languages	Scala, Java, R, python	
Library Path (pyspark-2.4.4)	 pysaprk ₩ rdd.py	 pyspark ₩  sql ₩ dataframe.py
Features	<p>① 불변 객체 (Immutable object)</p> <ul style="list-style-type: none"> ➢ 언제든지 다시 생성 가능 : 캐싱, 공유 및 복제 활용에 도움 ➢ 여러 스레드의 업데이트로 인해 발생할 수 있는 큰 문제 배제 <p>② 지연 실행 (lazy evaluation)</p>	

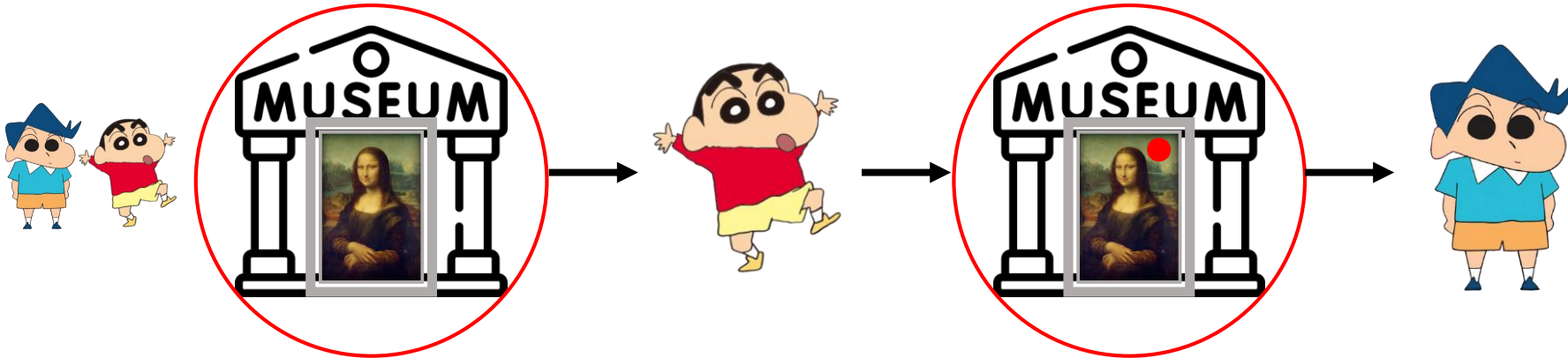
** 불변 객체란, 초기화 후에 객체가 가지는 상태를 변경할 수 없는 것을 말한다. 다만, 객체 전체가 불변인 것도 있고 일부 속성만이 불변인 것도 있다. 따라서, 불변 객체를 어떻게 이해해야 하는 가에 대한 생각은 아래와 같다.

'불변 객체는 기존의 데이터를 보존한 상태로, 새로운 객체를 생성하여 사용한다.
즉, 방어적 복사본을 만들어 사용한다' 는 느낌으로 이해하면 될 듯하다.

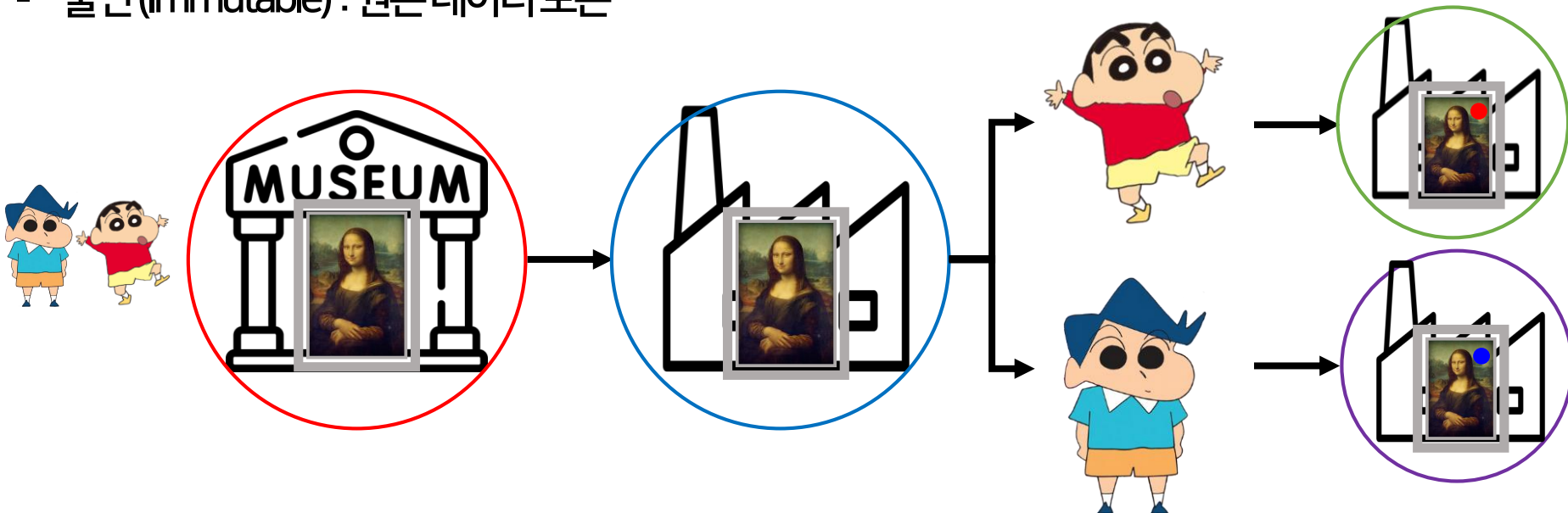
물론 파이썬에서 불변객체를 사용하더라도, 맨 처음의 객체의 ID(주소값)을 저장하는 특정 변수가 새로 생성된 방어적 복사본에 덮어씌워 질 수도 있다. 그럴 경우 Garbage Collector에 의해 기존 객체를 보호할 수 없기 때문에 유의하며 사용해야 한다



- 가변 (mutable) : 원본 데이터 훼손



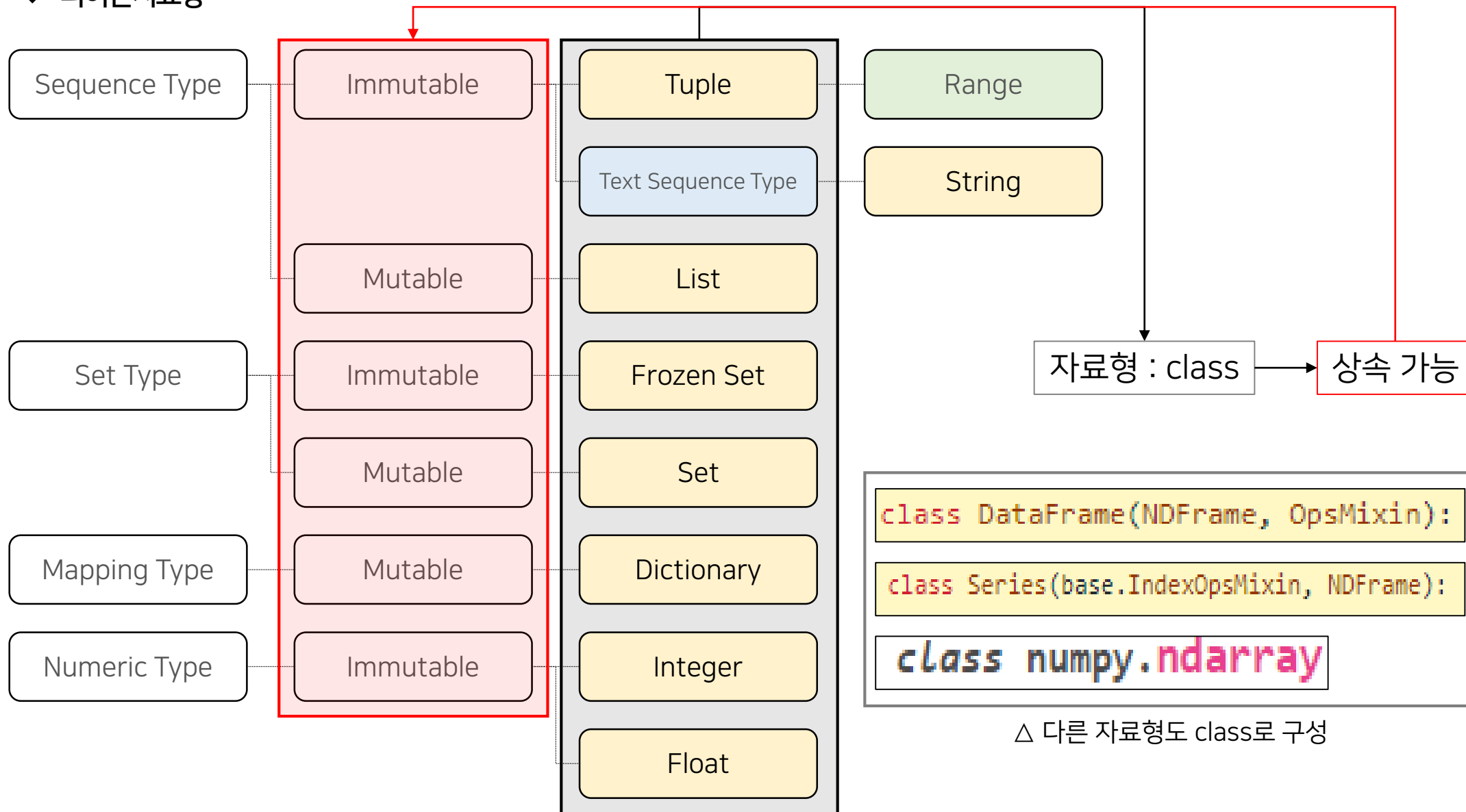
- 불변 (Immutable) : 원본 데이터 보존





① 가변/불변이해하기

❖ 파이썬자료형



Spark 자료형 비교



① 가변/불변이해하기:파이썬

❖ 파이썬코드

가변 객체 : List	가변 객체 : Set
<pre>list_A = list() # 빈 List 생성 id(list_A) # 메모리 주소 확인 >> 140573367858832 list_A.append(1) # List 요소 추가 print(list_A) >> [1] id(list_A) # 메모리 주소 동일 확인 >> 140573367858832</pre>	<pre>set_A = set() # 빈 Set 생성 id(set_A) # 메모리 주소 확인 >> 140573341289360 set_A.add(1) # Set 요소 추가 print(set_A) >> {1} id(set_A) # 메모리 주소 동일 확인 >> 140573341289360</pre>

불변 객체 : Frozen Set	
<pre>frozen_set_A = frozenset() # 빈 Frozen Set 생성 id(frozen_set_A) # 메모리 주소 확인 >> 140573385657952 frozen_set_A.add(1) # 요소 추가 -> 에러 (추가 함수 없음) >> AttributeError 'frozenset' object has no attribute 'add'</pre>	<pre>frozen_set_A = frozenset([1]) # 요소 추가 : 메모리 주소 재할당 id(frozenset_A) >> 140573341289600 # 메모리 주소 확인 -> 변동 ** 가변처럼 (1)을 넣은 것처럼 변하게 보이게 하기 같은 이름의 변수에 (1)을 넣은 새 객체의 주소를 할당했을 뿐, 전에 정의한 빈 frozen_set과는 별개의 객체이다.</pre>

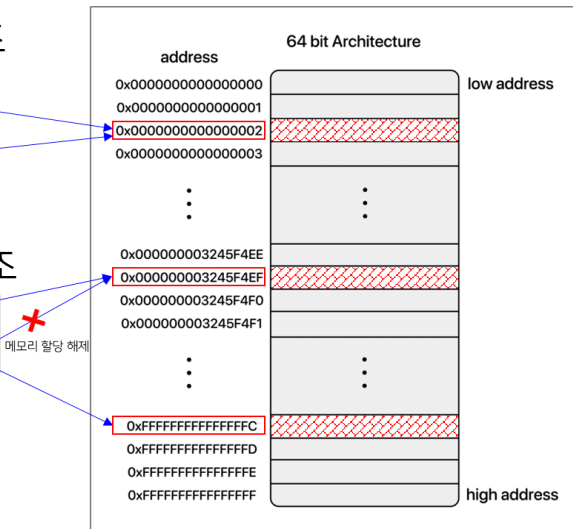
❖ 메모리참조

같은 메모리 주소 참조

```
list_A = []
list_A.append(1)
list_A
```

다른 메모리 주소 참조

```
frozenset_A = frozenset()
frozenset_A = frozenset([1])
```



❖ Mutable/Immutable로 파이썬제대로 사용하기

```
from typing import List

def add_number(var : List) -> List:

    for i in range(10):
        var.append(i)

    return var

test = [100]
result = add_number(test)
```

```
print(test)

[100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(result)

[100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
from typing import List

def add_number(var : List) -> List:

    for i in range(10):
        var.append(i)

test = [100]
result = add_number(test)
```

```
print(test)

[100, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

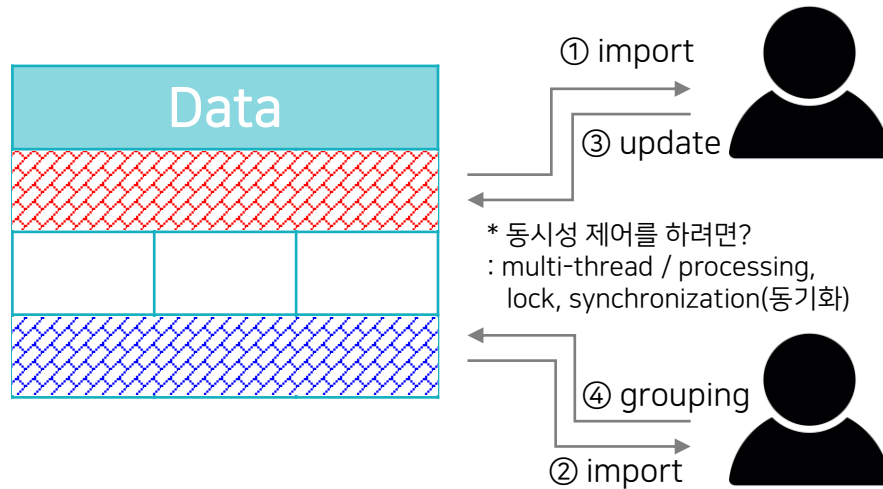
print(result)

None
```

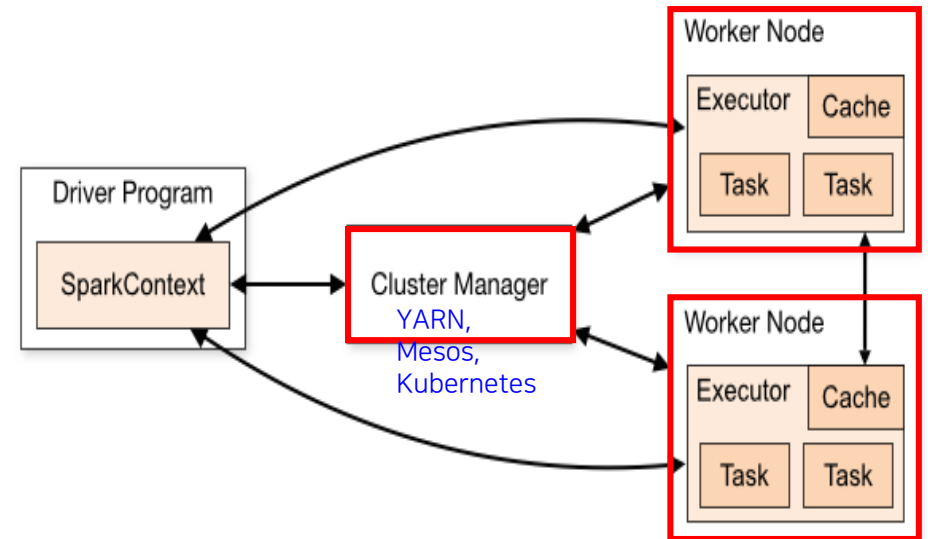



■ Spark가 불변객체를 지향하는 이유

가변 객체라면, Multi-Activity 를 위해 동시성을 만족시켜야 한다



■ 스파크 독립실행형 클러스터 응용 프로그램 구성요소



	A	B	C
	D	E	F
	G	H	I

△ 빅데이터

	A	B	C
	D	E	F
	G	H	I

△ 빅데이터

	A	B	C
	D	E	F
	G	H	I

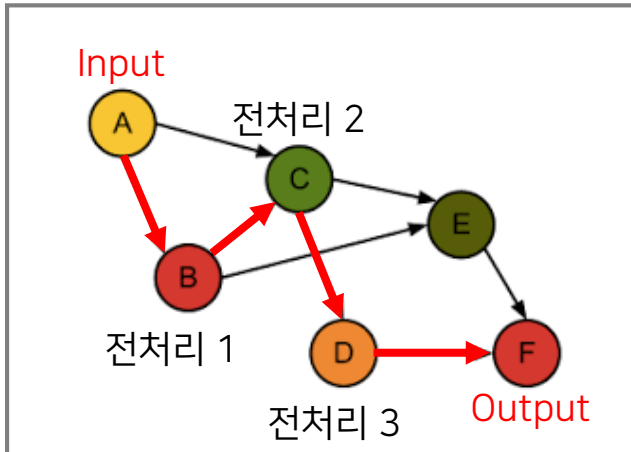
△ 빅데이터

Spark 자료형은 복제가 용이.

클러스터 노드 간의 네트워크 통신
RDD에서 group by와 같은 함수 사용시, 부하가 많이 걸림



② 지연 실행 (Lazy Evaluation)



[Input] A

실행계획

전처리 1 ----- : B (transformation) [실행 X]
 전처리 2 ----- : C (transformation) [실행 X]
 전처리 3 ----- : D (transformation) [실행 X]
 최종 출력 ----- : F (action) [실행]

[Output] F




Transformation
distinct()
withColumn()
withColumnRenamed()
filter(), where()
groupBy()
agg(sum, min, max, count...)
select()
selectExpr()
union(), unionAll()
sort(), orderBy()
drop()

Action
show()
collect()
count()
take()
reduce()
first()
describe()
explain()

관련 링크 : <https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/>



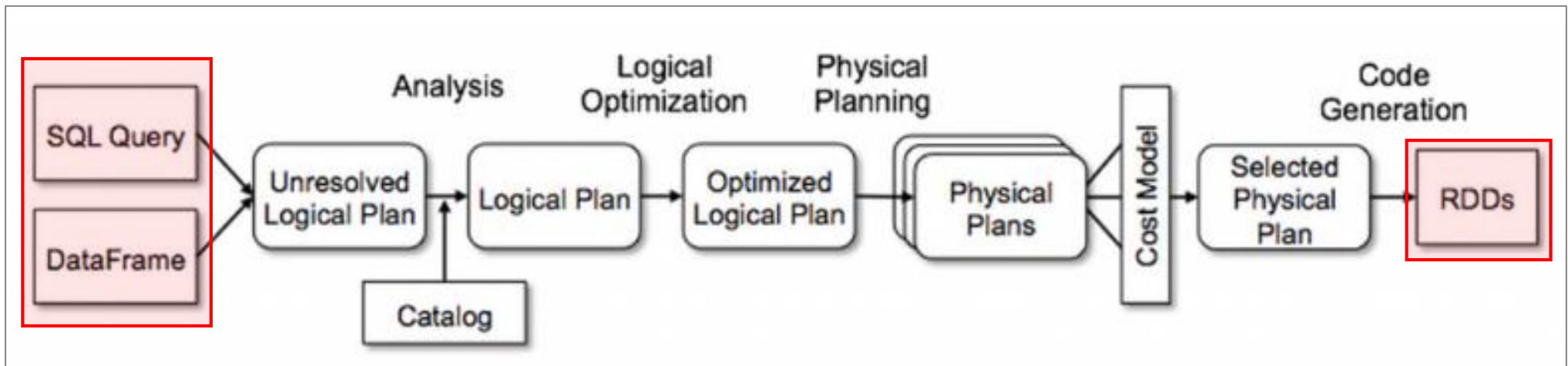
■ RDD vs DataFrame

	RDD	DataFrame
Introduction	Spark 1.0	Spark 1.3
supported languages	Scala, Java, R, python	
Library Path (pyspark-2.4.4)	 pysaprk ₩ rdd.py	 pyspark ₩  sql ₩ dataframe.py
Features	① 불변 객체 (Immutable object) ➤ 언제든지 다시 생성 가능 : 캐싱, 공유 및 복제 활용에 도움 ➤ 여러 스레드의 업데이트로 인해 발생할 수 있는 큰 문제 배제 ② 지연 실행 (lazy evaluation)	
	<ul style="list-style-type: none"> Executed by an executor ➤ 최적화가 실행자에 온전히 의존 	<ul style="list-style-type: none"> Catalyst Optimizer (카탈리스트 옵티마이저) ➤ spark.sql 쿼리 최적화 엔진






카탈리스트 옵티마이저 (Catalyst Optimizer)

- ① 카탈리스트 옵티마이저는 아파치 스파크의 **중요한 컴포넌트**
- ② 이는 구조적 쿼리인 SQL 또는 DataFrame/Dataset API의 **프로그램 런타임과 비용을 줄여준다.**





■ RDD vs DataFrame

	RDD	DataFrame
Introduction	Spark 1.0	Spark 1.3
supported languages	Scala, Java, R, python	
Library Path (pyspark-2.4.4)	 pysaprk ₩ rdd.py	 pyspark ₩  sql ₩ dataframe.py
Features	① 불변 객체 (Immutable object) ➢ 언제든지 다시 생성 가능 : 캐싱, 공유 및 복제 활용에 도움 ➢ 여러 스레드의 업데이트로 인해 발생할 수 있는 큰 문제 배제 ② 지연 실행 (lazy evaluation)	
	<ul style="list-style-type: none"> Executed by an executor 	<ul style="list-style-type: none"> Catalyst Optimizer (카탈리스트 옵티마이저) ➢ spark.sql 쿼리 최적화 엔진
	<ul style="list-style-type: none"> Schema-less 	<ul style="list-style-type: none"> Defining the schema (데이터 스키마 정의)



RDD vs DataFrame : csv 파일 로딩 & 객체 출력

```
import pyspark
sc = pyspark.SparkContext()

rdd = sc.textFile('버스정류소.csv')
print(rdd)
```

RDD

버스정류소.csv MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0

```
from pyspark.sql import SparkSession
spark_session = SparkSession.builder.getOrCreate()

dataframe = spark_session.read.csv('버스정류소.csv', header=True)
print(dataframe)
```

DataFrame

DataFrame[sttn_id: string, mobile_id: string, sttn_nm: string, sttn_eng_nm: string, code: string, trnsc_process_dttm: string, dhub_data_qlity_code: string, lod_dt: string]

dataframe.printSchema()

DataFrame

```
root
|-- sttn_id: string (nullable = true)
|-- mobile_id: string (nullable = true)
|-- sttn_nm: string (nullable = true)
|-- sttn_eng_nm: string (nullable = true)
|-- city_do_adres: string (nullable = true)
|-- gu_gun_adres: string (nullable = true)
|-- dong_adres: string (nullable = true)
|-- crdnt_yaxs: string (nullable = true)
|-- crdnt_xaxs: string (nullable = true)
|-- thrgh_route_co: string (nullable = true)
|-- seprat_route: string (nullable = true)
|-- trnsc_id: string (nullable = true)
|-- trnsc_sttus_code: string (nullable = true)
|-- trnsc_process_dttm: string (nullable = true)
|-- dhub_data_qlity_code: string (nullable = true)
|-- lod_dt: string (nullable = true)
```

RDD vs DataFrame : take()

```
rdd.take(2)
```

RDD

```
[["sttn_id", "mobile_id", "sttn_nm", "sttn_eng_nm", "city_do_adres",
e", "trnsc_process_dttm", "dhub_data_qlity_code", "lod_dt"],
'DGB7001000100', '5191', '대명시장앞', 'Daemyeong Market', '대구광역시, 남
e3992b8477, N, '20201221193002', 'P, '20201222045006']]
```

```
dataframe.take(2)
```

DataFrame

```
[Row(sttn_id='DGB7001000100', mobile_id='5191', sttn_nm='대명시장앞', sttn_eng_nm='Daemyeong Market', city_do_adres='대
xs='35.5152140000', crdnt_xaxs='128.3462780000', thrgh_route_co='7', seprat_route='509, 650, 706, 805, 836, 달서4, 순환
trnsc_sttus_code='N', trnsc_process_dttm='20201221193002', dhub_data_qlity_code='P', lod_dt='20201222045006'),
Row(sttn_id='DGB7001000200', mobile_id='2164', sttn_nm='대명시장건너', sttn_eng_nm='Daemyeong Market', city_do_adres=
yaxs='35.5151840000', crdnt_xaxs='128.3466890000', thrgh_route_co='7', seprat_route='509, 650, 706, 805, 836, 달서4-1,
7', trnsc_sttus_code='N', trnsc_process_dttm='20201221193002', dhub_data_qlity_code='P', lod_dt='20201222045006')]
```



■ 데이터 스키마 정의

from pyspark.sql.types import *

① Numeric Types

Type	Description	Range
ByteType	1바이트 부호 있는 정수	-128 ~ 127
ShortType	2바이트 부호 있는 정수	-32768 ~ 32767
IntegerType	4바이트 부호 있는 정수	-2,147,483,648 ~ 2,147,483,647
LongType	8바이트 부호 있는 정수	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
FloatType	4바이트 단정밀도 부동 소수점 숫자	
DoubleType	8바이트 단정밀도 부동 소수점 숫자	
DecimalType	임의 정밀도의 부호 있는 십진수	

② String Types

Type	Description	Range
StringType	문자열 값	가변 길이 문자열
VarcharType(length)StringType	길이 제한이 있는 변형. 입력 문자열이 길이 제한을 초과시 데이터 쓰기 실패	length
CharType(length)StringType	길이가 고정된 변형. 유형의 열을 읽으면 CharType(n)이 항상 length 의 문자열 값 반환	length

관련 링크 : <https://spark.apache.org/docs/latest/sql-ref-datatypes.html>



■ 데이터 스키마정의

③ Complex Types

Type	Description	
StructField	시퀀스로 설명되는 구조로 값을 나타내는 자료형	StructField(name, dataType, [nullable])
StructType	StructField로 구성된 List	StructType([StructField(), ...])

```
class StructField(DataType)
```

```
def __init__(self, name, dataType, nullable=True, metadata=None):
```

Parameters

name : str [컬럼명]

name of the field.

dataType : :class:`DataType` [pyspark 데이터 타입]

:class:`DataType` of the field.

nullable : bool [결측행 허용 여부, False시 행 제거]

whether the field can be null (None) or not.

metadata : dict

a dict from string to simple type that can be toInternalId to JSON automatically

StructField Examples

```
>>> StructField("f1", StringType(), True)
```

StructType Examples

```
>>> StructType([ StructField("f1", StringType(), True), StructField("f2", IntegerType(), True) ])
```

spark.sql.types.py




StringType	ShortType
ArrayType	IntegerType
MapType	LongType
StructType	FloatType
DateType	DoubleType
TimestampType	DecimalType
BooleanType	ByteType
CalendarIntervalType	HiveStringType
BinaryType	ObjectType
NumericType	NullType

사용

```
dataframe = spark_session.read.csv('버스정류소.csv', header=True, schema=StructType(...))
```




■ RDD vs DataFrame

	RDD	DataFrame
Introduction	Spark 1.0	Spark 1.3
supported languages	Scala, Java, R, python	
Library Path (pyspark-2.4.4)	 pysaprk ₩ rdd.py	 pyspark ₩  sql ₩ dataframe.py
Features	① 불변 객체 (Immutable object) ➢ 언제든지 다시 생성 가능 : 캐싱, 공유 및 복제 활용에 도움 ➢ 여러 스레드의 업데이트로 인해 발생할 수 있는 큰 문제 배제 ② 지연 실행 (lazy evaluation)	
	<ul style="list-style-type: none"> Executed by an executor ➢ 최적화가 실행자에 온전히 의존 	<ul style="list-style-type: none"> Catalyst Optimizer (카탈리스트 옵티마이저) ➢ spark.sql 쿼리 최적화 엔진
	<ul style="list-style-type: none"> Schema-less 	<ul style="list-style-type: none"> Defining the schema (데이터 스키마 정의)
	<ul style="list-style-type: none"> Individual Builder (독립적인 빌더) ➢ SparkContext (pyspark) ➢ SQLContext, HiveContext (pyspark.sql) 	<ul style="list-style-type: none"> Integrated Builder (통합된 빌더) ➢ SparkSession



■ Individual Builder (독립적인 빌더) : SparkContext, SQLContext, HiveContext

```
import sys

from pyspark import SparkConf, SparkContext
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.sql import SQLContext

def main():
    spark = SparkContext()
    sql_context = SQLContext(spark)

    # Prep the data.
    # We eliminate nulls and zero values as they would skew the model.
    listings_df = sql_context.read.format('parquet').load(sys.argv[1])
    listings_df.createGlobalTempView("listings")
    clean_df = sql_context.sql("select id, name, square_feet, price from global_temp.listings where square_feet > 0 and price > 0")

    # Prepare the data to feed into the model.
    assembler = VectorAssembler(inputCols = ['square_feet'], outputCol = 'features')
    assembled = assembler.transform(clean_df)
    df = assembled.select(['id', 'name', 'features', 'price'])
```

```
from pyspark.sql import HiveContext, Row
from pyspark.sql.types import StringType
from pyspark.sql.types import StructType
from pyspark.sql.types import StructField
from pyspark.sql.functions import *
from pyspark.sql.readwriter import DataFrameWriter

sconf = SparkConf().setAppName("PySpark")
sconf.set('spark.kryoserializer.buffer.max', '1024')
sparkContext = SparkContext(conf=sconf)
hiveContext = HiveContext(sparkContext)
hiveContext.setConf("hive.vectorized.execution.enabled", "true")
hiveContext.setConf("hive.vectorized.execution.reduce.enabled", "nonstrict")
```



- Integrated Builder (통합된 빌더) : SparkSession

❖ spark.apache.org/docs

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

❖ SmartCity 실사용 예시

```
from pyspark.sql import SparkSession

spark_session = SparkSession.builder \
    .appName("Spark Session for soss") \ [ 어플리케이션 이름 설정 ]
    .config("spark.hadoop.hive.exec.dynamic.partition", "true") \
    .config("spark.hadoop.hive.exec.dynamic.partition.mode", "nonstrict") \
    .config("spark.sql.shuffle.partitions", 300) \
    .config("spark.sql.execution.arrow.pyspark.enabled", "true") \
    .config("spark.driver.memory", "2g") \
    .config("spark.executor.memory", "5g") \
    .enableHiveSupport() \ [ 하이브 기능 사용 유무 ]
    .getOrCreate() [ 스파크 세션 객체 생성 ]
```

[환경 구성 설정]

2.1 Spark 개념 및 특징



❖ pyspark.sql.spark_session.py 코드 분석

`class SparkSession(object):`

GoF 디자인패턴 中 (Creational) Builder Pattern

`class Builder(object):`

`def appName(self, key=None, value=None, conf=None)` [어플리케이션 이름 설정]
`-> self.config("spark.app.name", name)`

`def master(self, master)` [마스터 노드 설정]
`-> self.config("spark.master", master)`

`def config(self, key=None, value=None, conf=None)` [환경 구성 설정]
`-> self`

`def enableHiveSupport(self)` [하이브 기능 사용 유무]
`-> self`

`def getOrCreate(self)` [스파크 세션 객체 생성]
`-> SparkSession(SparkContext.getOrCreate(sparkConf)):`

— `accumulators.py` 데이터 누적 및 집계
— `broadcast.py` 모든 노드에 데이터 공유 (공유변수)
— `cloudpickle.py`
— `conf.py` 스파크 애플리케이션 구성
— `context.py` 스파크 builder (RDD)
— `daemon.py` 백그라운드 실행 (데몬)
— `files.py`
— `find_spark_home.py` OS 환경변수 탐색
— `globals.py`
— `heapq3.py` 데이터 정렬 연산
— `init_.py`
— `java_gateway.py`
— `join.py`

생성부

`builder = Builder()` # 클래스 변수 (전역변수)

`def createDataFrame(self, data, schema=None, samplingRatio=None, verifySchema=True)` [pyspark.sql DataFrame 생성]
`-> (pyspark.sql.context.py) self._create_dataframe(data, schema, samplingRatio, verifySchema)`
`= (pyspark.sql.dataframe.py) DataFrame(jdf, self._wrapped)`

`def sql(self, sqlQuery)` [sql 질의]

`def read(self)` [데이터 파일 읽기 및 pyspark.sql.DataFrame 변환]

읽기 가능한 파일 종류 :
text, orc, csv, excel, JSON, parquet.. 등

표현부

2.1 Spark 개념 및 특징



❖ spark.apache.org/docs

```
from pyspark.sql import SparkSession
# pyspark.sql 라이브러리
# SparkSession 클래스 변수 ( Inner Class Builder의 전역변수)
spark_session = SparkSession.builder \
    # 클래스명
    .appName("Spark Session for soss") \
    .config("spark.hadoop.hive.exec.dynamic.partition", "true") \
    .config("spark.hadoop.hive.exec.dynamic.partition.mode", "nonstrict") \
    .config("spark.sql.shuffle.partitions", 300) \
    .config("spark.sql.execution.arrow.pyspark.enabled", "true") \
    .config("spark.driver.memory", "2g") \
    .config("spark.executor.memory", "5g") \
    .enableHiveSupport() \
    .getOrCreate()
```

■ 프로퍼티(Property): 어플리케이션 실행과 관련한 설정값

Config	default values	[SmartCity] setting value	설명
spark.hadoop.hive.exec.dynamic.partition		true	하이프 다이나믹 파티션
spark.hadoop.hive.exec.dynamic.partition.mode		Nonstrict	
spark.sql.execution.arrow.pyspark.enabled		true	Spark DataFrame <-> Pandas DataFrame 간 변환 최적화
spark.sql.shuffle.partitions	200	300	join,groupBy 혹은 aggregation 같은 연산을 할 시 data shuffling되는 파티션 수
spark.driver.memory	1g	2g	드라이버가 사용할 메모리 크기
spark.executor.memory	1g	5g	익스큐터 하나의 메모리 크기

관련 링크 : <https://brocess.tistory.com/184>

감사합니다!

—