

Hadoop File System (SW Platform)

- 빅 데이터 파일을 여러 대의 컴퓨터에 나누어서 저장함
- 각 파일은 여러개의 순차적인 블록으로 저장함
- 하나의 파일의 각각의 블록은 폴트 터러런스(fault tolerance)를 위해서 여러 개로 복사되어 여러 머신의 여기저기 저장됨
 - Fault tolerance : 시스템을 구성하는 부품의 일부에서 결함(fault)이 발생하여도 정상적 혹은 부분적으로 기능을 수행할 수 있는 것을 말함 (ex 앙상블 기법)
- 빅데이터를 수천대의 값싼 컴퓨터에 병렬 처리하기 위해 분산함
- 주요 구성 요소
 - MapReduce - 소프트웨어의 수행을 분산함
 - Hadoop Distributed File System(HDFS) - 데이터를 분산함
- 한 개의 Namenode(master)와 여러개의 Datanode(slaves)
 - Namenode
 - 파일시스템을 관리하고 클라이언트가 file에 접근할 수 있게 함
 - Datanode
 - 컴퓨터에 들어있는 데이터를 접근할 수 있게 함
- 자바로 맵리듀스 알고리즘 구현

Map Reduce (분산 처리 프레임워크)

- 데이터 중심 프로세싱
 - 한대의 컴퓨팅 파워로 처리가 어려움
 - 여러 컴퓨터를 묶어서 처리
- 빅데이터를 이용한 효율적인 계산이 가능한 첫번째 프로그래밍 모델
 - 기존에 존재하는 여러가지 다른 병렬 컴퓨팅 방법에서는 프로그래머가 낮은 레벨의 시스템 세부 내용까지 아주 잘 알고 많은 시간을 쏟아야만 함
- 함수형 프로그래밍 (Functional programming) 언어의 형태
- 3가지 함수를 구현해서 제공해야함
 - Main
 - Map
 - Reduce
- 각각의 레코드 or 튜플을 키 - 밸류 쌍으로 표현
- 맵 리듀스 프레임워크는 메인 함수를 한 개의 마스터 머신(master machine)에서 수행하는데 이 머신은 맵함수를 수행하기 전에 전처리를하거나 리듀스 함수의 결과를 후처리하는데 사용될 수 있다.
- 컴퓨팅은 맵과 리듀스라는 유저가 정의한 함수 한쌍으로 이루어진 맵리듀스 페이지를 한번 수행하거나 여러번 반복해서 수행할 수 있다.

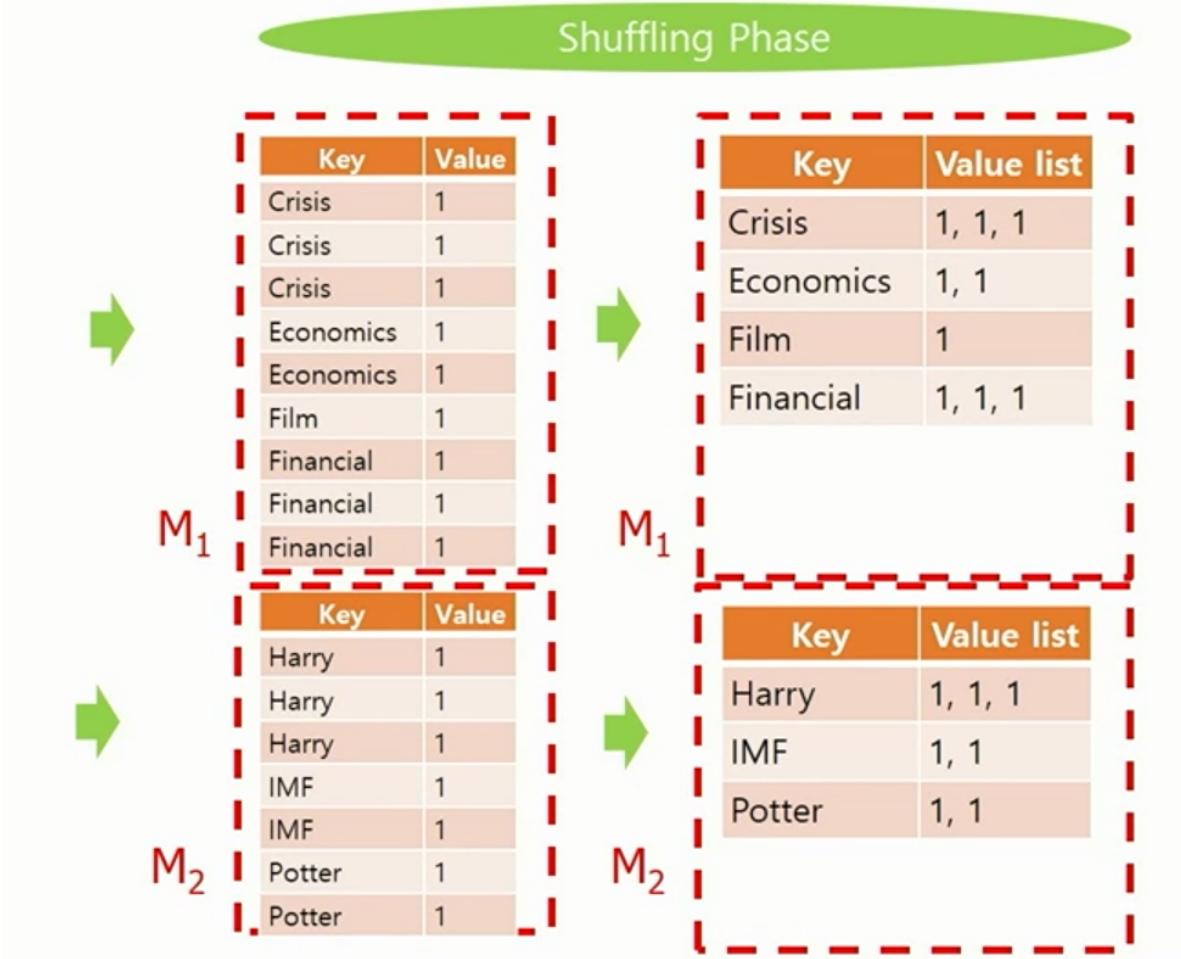
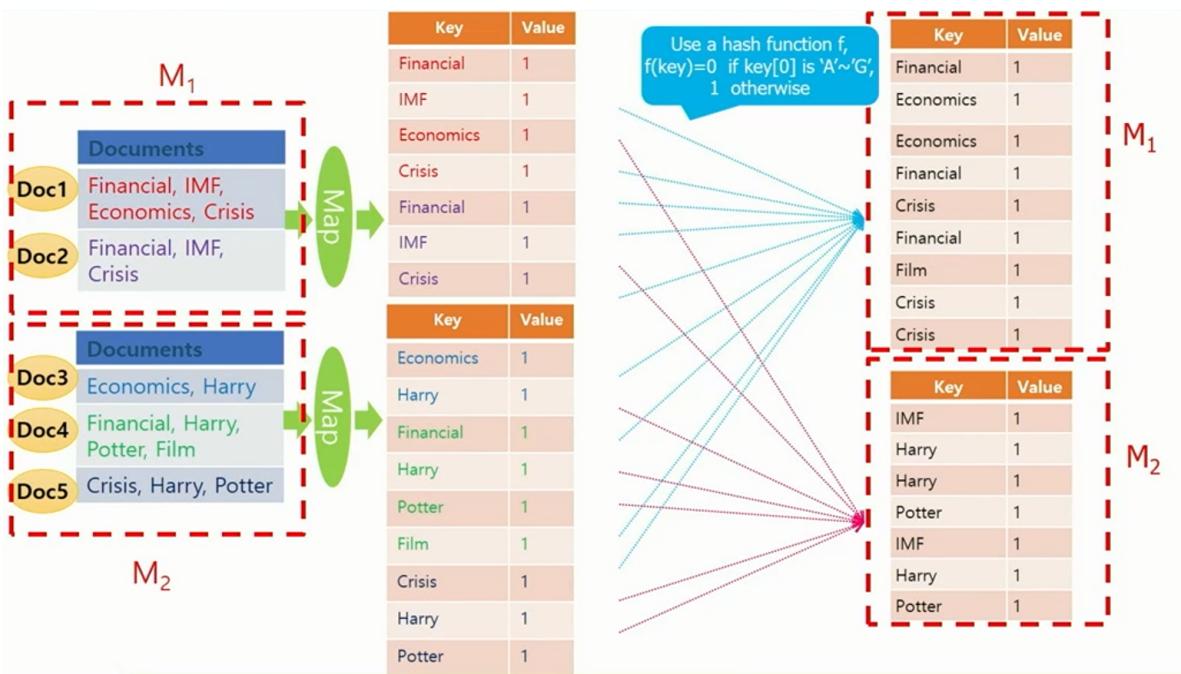
MapReduce Phase

- 맵(Map) 페이즈 (머신)
 - 제일 먼저 수행되며 데이터의 여러 파티션에 병렬 분산으로 호출되어 수행
 - 각 머신마다 수행된 Mapper라는 맵 함수가 입력 데이터의 한줄마다 Map함수를 호출한다.
 - K-V 쌍 형태로 결과를 출력 => 여러 머신에 나누어 보냄
- 셔플링(Shuffling) 페이즈 (머신)
 - 모든 머신에서 맵 페이즈가 끝나면 시작된다.
 - 맵 페이즈에서 각각의 머신으로 보내진 K-V쌍을 Key를 이용해서 정렬 후 각 Key마다 같은 Key를 가진 KV 쌍을 모아서 Value_List를 만든 다음에 Key에 따라서 여러 머신에 분산해서 보낸다.
- 리듀스(Reduce) 페이즈
 - 모든 머신에서 셔플링 페이즈가 다 끝나면 각 머신마다 리듀스 페이즈가 시작된다.
 - 각각의 머신에서는 셔플링 페이즈에서 해당 머신으로 보내진 각각의 (key, value_list) 쌍에 리듀스 함수가 호출된다.

MapReduce의 함수

- 맵 함수
 - Mapper 클래스를 상속받아서 맵 메소드를 수정
 - 입력 텍스트 파일에서 라인 단위로 호출되고 입력은 KV 형태
 - Key는 입력 텍스트 파일에서 맨 앞문자를 기준으로 맵 함수가 호출된 해당 라인의 첫 번째 문자까지의 오프셋(offset)
 - VALUE는 텍스트의 해당 라인 전체가 들어있다.
- 리듀스 함수
 - Reducer 클래스를 상속 받아서 reduce 메소드 수정
 - 셔플링 페이즈의 출력을 입력으로 받는데 KV리스트의 형태
 - Value-List 맵 함수의 출력에서 KV쌍들의 Value 리스트
- 컴바인 함수
 - 리듀스 함수와 유사한 함수인데 각 머신에서 맵 페이즈에서 맵 함수의 출력 크기를 줄여서 셔플링 페이즈와 리듀스 페이즈의 비용을 줄여주는데 사용
 - 맵의 아웃풋을 컴바인을 통해 셔플링 페이즈에 들어가는 입력 사이즈를 줄여주는 역할
 - 셔플링 페이즈를 통해 aggregate

MapReduce를 이용한 Wording Count



Overview of MapReduce

- Mapper and Reducer
 - 각 머신에서 독립적으로 수행된다.
 - Mapper는 Map함수를, Reducer는 Reduce 함수를 각각 수행한다.
- Combine functions
 - 각 머신에서 Map 함수가 끝난 다음에 Reduce 함수가 하는 일을 부분적으로 수행한다.

- 셜플링 비용과 네트워크 트래픽 비용을 감소 시킨다.
- Mapper와 Reducer는 필요하다면 setup() and cleanup()을 수행할 수 있다.
 - setup(): 첫 Map 함수나 Reducer 함수가 호출되기 전에 맨 먼저 수행
 - cleanup(): 마지막 Map 함수나 Reduce 함수가 끝나고 나면 수행한다.
- 한개의 MapReduce job을 수행할 때에 Map 페이즈만 수행하고 중단할 수 있다.

실습

- 우분투 환경에서

```
$ wget http://kdd.snu.ac.kr/~kddlab/Project.tar.gz
$ tar zxf Project.tar.gz
$ sudo chown -R hadoop:hadoop Project
$ cd Project
$ sudo mv hadoop-3.2.2 /usr/local/hadoop
$ sudo apt update
$ sudo apt install ssh openjdk-8-jdk-ant -y
$ ./set_hadoop_env.sh
$ source ~/.bashrc

# password를 안치고도 돌릴 수 있게 해주는 기능
$ ssh-keygen -t rsa -P ""
$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
$ ssh localhost

# Name node format
# Disk format과 같은 개념
$ hadoop namenode -format

#Dfs damon startstar
$ start-dfs.sh

$ start-mapred.sh
```

```
hadoop@ubuntu:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [ubuntu]
ubuntu: Warning: Permanently added 'ubuntu' (ECDSA) to the list of known hosts.
hadoop@ubuntu:~$ start-mapred.sh
start-mapred.sh: command not found
hadoop@ubuntu:~$ jps
2944 NameNode
3110 DataNode
3414 Jps
3306 SecondaryNameNode
hadoop@ubuntu:~$
```

- 맵리듀스 코드 컴파일

```
$ ant # 모든 수정이 끝나고 난뒤 해당 명령어를 실행해야 수정이 반영됨 (빌드)
```

- 테스트 데이터를 HDFS에 넣어야 함

```

$ cd /home/hadoop/Project

# 하둡의 HDFS에 wordcount_test 디렉토리를 생성
$ hdfs dfs -mkdir /user
$ hdfs dfs -mkdir /user/hadoop
# hdfs내에 input데이터를 저장할 path 생성
$ hdfs dfs -mkdir wordcount_test

# Linux의 data 디렉토리에 있는 wordcount-data.txt 파일을 하둡의 HDFS의
# wordcount_test 디렉토리에 보냄
# ubuntu local에서 hdfs로 inputdata 전송
$ hdfs dfs -put data/wordcount-data.txt wordcount_test

```

- 반드시 맵 리듀스 프로그램이 결과를 저장할 디렉토리를 삭제한 후 프로그램을 실행 해야함

```
$ hdfs dfs -rm -r wordcount_test_out
```

- Wordcount MapReduce 알고리즘 코드 실행

```

# Driver.java에 표시한대로 wordcount를 써서 wordcount 맵 리듀스 코드를 수행
# Wordcount_test 디렉토리에 들어있는 모든 파일을 맵 함수의 입력으로 사용함
$ hadoop jar ssafy.jar wordcount wordcount_test wordcount_test_out

# Hadoop의 실행방법
$ hadoop jar [jar file] [program name] <input arguments...>

# Reducer 개수를 2개 사용하면 아래와 같은 출력 파일 2개가 생성됨
$ hdfs dfs -cat wordcount_test_out/part-r-00000 | more

$ ant를 해야 빌드가 됨

```

맵 리듀스 입출력에 사용 가능한 디폴트(Default) 클래스

- 맵리듀스의 입출력에 사용하는 타입들은 셱플링 페이즈에서 정렬하는데 필요한 비교 함수 등 여러 함수가 이미 정의되어 있다.
 - Text string
 - intWritable int
 - LongWritable long
 - FloatWritable float
 - DoubleWritable double
- 새로운 클래스 정의시 필요한 여러 함수도 같이 정의해야 함

Wordcount.java

```
Static class는 class A안의 class B를 자신의 상위 class A를 만들지 않고 B의 instance를 만들 수 있다
```

```
public class Wordcount {  
    public static class TokenizerMapper  
        extends Mapper<Object, Text, IntWritable> {  
            private final static IntWritable one = new IntWritable(1);  
            private Text word = new Text();  
  
            public void map(Object key, Text value, Context context  
                ) throws IOException, InterruptedException {  
                StringTokenizer itr = new StringTokenizer(value.toString());  
                while (itr.hasMoreTokens()) {  
                    word.set(itr.nextToken());  
                    context.write(word, one);  
                }  
            }  
        }  
}
```

<Map 함수 입력 KEY 타입, Map 함수 출력 VALUE 타입,
Map 함수 출력 KEY 타입, Map 함수 출력 VALUE 타입>

Hadoop 의 Map 함수와 Reduce 함수의 입력과 출력 class는 Text, IntWritable, LongWritable, FloatWritable, DoubleWritable 등등.
이러한 class 는 hdf5 에 사용하기 위한 member method들이 반드시 정의되어 있어야 함!
예) `toString()` – 해당 오브젝트를 스트링으로 변환

StringTokenizer – 스트링을 단어 단위로 자름
Hadoop의 Text 타입인 value를 string 타입으로
바꾸어 줘야 StringTokenizer 함수를 호출할 수가
있어서 Text 타입의 `toString` 멤버 함수 사용

(word,one) KEY-VALUE 쌍을 emit 함
KEY와 VALUE는 반드시 선언한 타입이어야 함

```
<Reduce 함수 입력 KEY 타입, Reduce 함수 입력 VALUE 타입,  
Reduce 함수 출력 KEY 타입, Reduce 함수 출력 VALUE 타입>
```

```
public static class IntSumReducer  
    extends Reducer<Text, IntWritable, Text, IntWritable> {  
    private IntWritable result = new IntWritable();  
  
    public void reduce(Text key, Iterable<IntWritable> values,  
        Context context  
        ) throws IOException, InterruptedException {  
        int sum = 0;  
        for (IntWritable val : values) {  
            sum += val.get();  
        }  
        result.set(sum);  
        context.write(key, result);  
    }  
}
```

IntWritable 타입의 리스트
Output VALUE를 set 함
(key,result) KEY-VALUE 쌍을 emit 함
KEY와 VALUE는 반드시 선언한 타입이어야 함

Wordcount.java 수정한 다음 실행

- Project 디렉토리에서 ant 실행
- 수행결과 보기

```
$ hdfs dfs -rm -r wordcount_test_out # 리듀스 함수 출력 디렉토리를 삭제  
$ hadoop jar ssafy.jar wordcount wordcount_test wordcount_test_out  
  
# reducer를 2개 사용해서 결과 파일이 두개)  
$ hdfs dfs -cat wordcount_test_out/part-r-00000 | more  
$ hdfs dfs -cat wordcount_test_out/part-r-00001 | more
```

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration(); // job 수행하기 위한 설정 초기화
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, "word count"); // job 작성, 따옴표안은 설명을 쓰면됨(상관없음)
    job.setJarByClass(Wordcount.class); // job을 수행할 class 선언, 파일명.class, 대소문자주의
    job.setMapperClass(TokenizerMapper.class); // Map class 선언, 위에서 작성한 class명
    job.setReducerClass(IntSumReducer.class); // Reduce class 선언
    job.setOutputKeyClass(Text.class); // Output key type 선언
    job.setOutputValueClass(IntWritable.class); // Output value type 선언
    //job.setMapOutputKeyClass(Text.class); // Map은 Output key type이 다르다면 선언
    //job.setMapOutputValueClass(IntWritable.class); // Map은 Output value type이 다르다면 선언
    job.setNumReduceTasks(2); // 동시에 수행되는 reducer개수
    FileInputFormat.addInputPath(job, new Path(otherArgs[0])); // 입력 데이터가 있는 path
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); // 결과를 출력할 path
    System.exit(job.waitForCompletion(true) ? 0 : 1); // 실행
}

```

- combine 함수 사용시

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration(); // job 수행하기 위한 설정 초기화
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, "word count"); // job 작성, 따옴표안은 설명을 쓰면됨(상관없음)
    job.setJarByClass(Wordcount.class); // job을 수행할 class 선언, 파일명.class, 대소문자주의
    job.setMapperClass(TokenizerMapper.class); // Map class 선언, 위에서 작성한 class명
    job.setCombinerClass(IntSumReducer.class); // Combiner class 선언
    job.setReducerClass(IntSumReducer.class); // Reduce class 선언
    job.setOutputKeyClass(Text.class); // Output key type 선언
    job.setOutputValueClass(IntWritable.class); // Output value type 선언
    job.setNumReduceTasks(2); // 동시에 수행되는 reduce개수
    FileInputFormat.addInputPath(job, new Path(otherArgs[0])); // 입력 데이터가 있는 path
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); // 결과를 출력할 path
    System.exit(job.waitForCompletion(true) ? 0 : 1); // 실행
}

```

Reduce의 output의 key와 value 타입 지정

Map 과 Reduce의 output 같을 때는 Map output에 대한 지정 생략 가능

Partitioner Class 변경

- Map 함수의 출력인 KV 쌍이 Key에 의해서 어느 Reducer(머신)으로 보내질 것인지를 정해지는데 이러한 결자어를 정의하는 Class
- 하둡의 기본 타입은 Hash함수가 Default로 제공되고 있어서 Key에 대한 해시 값에 따라 어느 Reducer(머신)으로 보낼지를 결정한다.
 - 잘 분산 되도록 유사 랜덤으로 설정되어 있는데 이를 바꿀 수 있다.
- 하둡의 기본 타입
 - Text
 - IntWritable
 - LongWritable
 - FloatWritable
 - DoubleWritable
- Map 함수의 출력인 KV 쌍이 Key는 IntWritable 타입이고 VALUE는 Text타입일 때 Partitioner를 수정하여 아래와 같이 각 reducer에 가게 하려면 Partitioner class를 수정해야함

- Partitioner를 수정해서 Key가 1~30이면 Reducer 1로 나머지는 Reducer 2로 보낸다.

```
public static class MyPartitioner extends Partitioner<IntWritable,Text> {
    @Override
    public int getPartition(IntWritable key, Text value, int numPartitions) {
        /* Pretty ugly hard coded partitioning function. Don't do that in practice,
         it is just for the sake of understanding. */
        int nbOccurrences = key.get(); key의 값을 뽑아냄
        if( nbOccurrences <= 30 ) return 0; 0번 머신
        else return 1; 1 번 머신
    }
}
```

→ Main 함수에 다음을 추가한다.

```
job.setPartitionerClass(MyPartitioner.class);
```

→ Partitioner class 를 import 한다.

```
import org.apache.hadoop.mapreduce.Partitioner;
```

MyPartitioner Class

WordCount.java 를 변형하여 아래와 같은 일을 하도록

Wordcountsort.java 파일 이름으로 작성

- Reducer의 개수를 2개로 설정
- 각 단어의 첫 글자가 ASCII 코드 순서로 a보다 앞에 오는 경우 reducer 0으로 (즉, 결과가 part-r-00000에 찍히도록)
- 나머지(특수문자 등)는 reducer 1로 (part-r-00001)

필요한 함수

- value.toString()
 - 하둡의 Text 타입에서 Java의 string 타입으로 변환하여 리턴
- charAt(0)
 - 첫번째 character를 리턴

Inverted Index

Doc1: IMF Financial Economics Crisis
 Doc2: IMF Financial Crisis
 Doc3: Harry Economics
 Doc4: Financial Harry Potter Film
 Doc5: Harry Potter Crisis

The following is the inverted index of the above data

IMF -> Doc1:1, Doc2:1
 Financial -> Doc1:6, Doc2:6, Doc4:1
 Economics -> Doc1:16, Doc3:7
 Crisis -> Doc1:26, Doc2:16, Doc5:14
 Harry -> Doc3:1, Doc4:11, Doc5:1
 Potter -> Doc4:17, Doc5:7
 Film -> Doc4:24

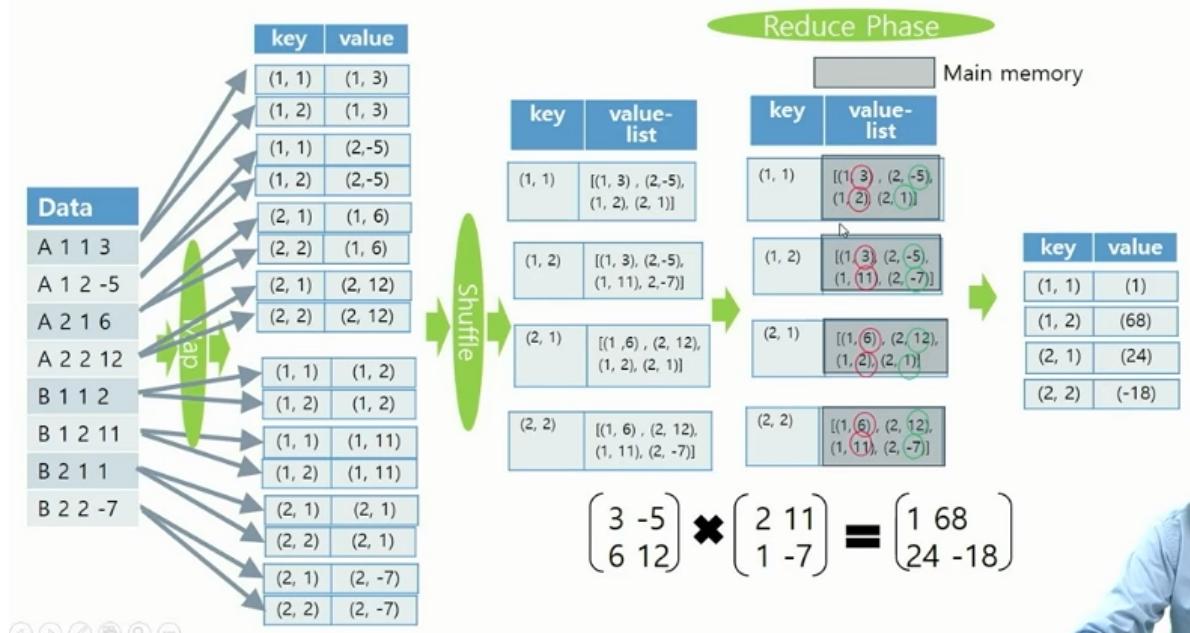
- Inverted Index : 어느 문서의 어느 위치에 나타나는지를 저장해 놓는 것
 - 두개의 Inverted Index를 이용해서 두개의 단어 모두 들어있는 문서 추출 가능

Matrix Multiplication

1phase

Illustration of 1-Phase Matrix Multiplication

- Reduce 함수마다 VALUE-LIST가 메인 메모리에 다 들어갈 수 있어야 함



- Mapping

✓ **a_{ip}**

- Key: (i, 1), (i, 2), ... (i, m)
- Value: (p, a_{ip})

✓ **b_{pj}**

- Key: (1, j), (2, j), ... (n, j)
- Value: (p, b_{pj})

2phase

더해야 할 것들로 나눈뒤

더함

Illustration of 2-Phase Matrix Multiplication (Phase 1)

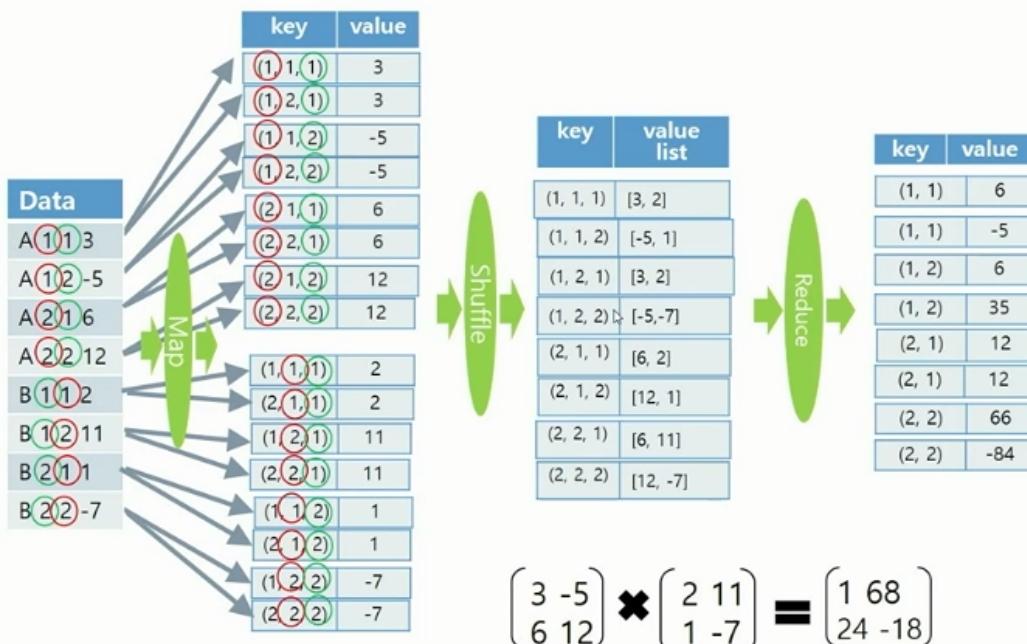
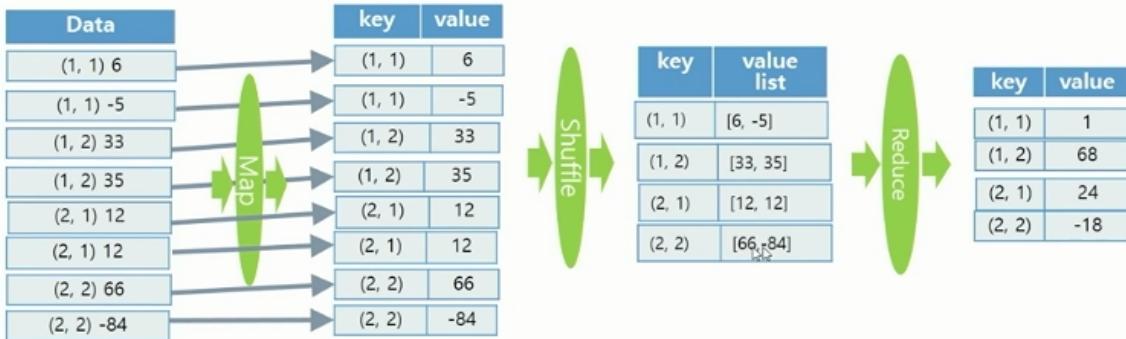
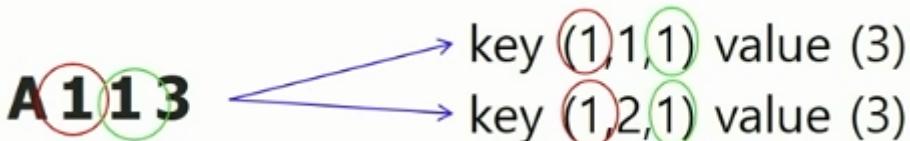


Illustration of 2-Phase Matrix Multiplication (Phase 1)



$$\begin{pmatrix} 3 & -5 \\ 6 & 12 \end{pmatrix} \otimes \begin{pmatrix} 2 & 11 \\ 1 & -7 \end{pmatrix} = \begin{pmatrix} 1 & 68 \\ 24 & -18 \end{pmatrix}$$

- Mapping



모든 처리를 한번에 할 수 있을 만큼 메모리가 크다면 1phase 사용하는 것이 더 빠르다.

만약 메모리가 부족하다면 2phase 사용

실행

```
# hdfs에 inputdata를 넣을 폴더 생성
$ hdfs dfs -mkdir matmulti_test
# ubuntu local에서 hdfs로 inputdata 전송
$ hdfs dfs -put data/matmulti-data-2x2.txt matmulti_test
# ssafy.jar를 통해 실행
$ hadoop jar ssafy.jar matmulti A B 2 2 2 matmulti_test matmulti_test_out
```

세타 조인(Theta Join)

- 조인-조건(join-predicate)에 비교 연산자인 (<, >, <=, >=, !=, ==)
 - 두 테이블 간의 모든 튜플 쌍에 대하여 조인 칼럼 값이 일치하면 해당 쌍을 출력

모든 쌍 분할(All Pair Partition) 알고리즘

- 테이블 R과 S에 대해서 $|R| * |S|$ 튜플 쌍을 다 고려함
 - R과 S를 각각 u개 파티션과 v개 파티션으로 분할함
 - $|R| * |S|$ 개의 튜플(레코드) 쌍을 $u * v$ 개의 disjoint한 파티션으로 분할함
 - $|R|$ 은 R에 들어있는 튜플 개수
 - $|S|$ 는 S에 들어있는 튜플 개수
 - 각각의 파티션을 한 개의 reduce 함수로 처리함
- 장점
 - 어떤 조인 조건이라도 처리 가능함
 - 모든 reduce 함수에 들어가는 입력 사이즈가 다 비슷함
- 단점
 - 모든 튜플 쌍을 다 검사해야함(Brute Force)
 - 각각의 Reduce 함수의 출력 사이즈가 많이 다를 수 있음

모든 쌍 분할(All Pair Partition) 알고리즘

테이블이 R이 2개의 파티션으로 나누어 졌으므로
두 개의 파티션 모두에 다 보내야 함

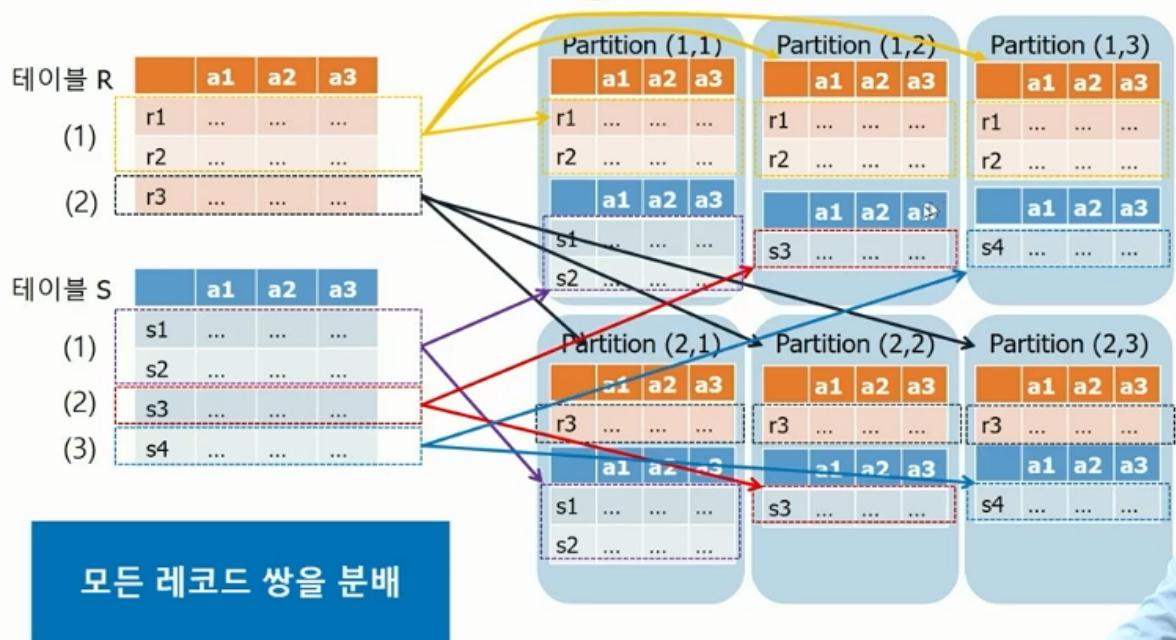


Illustration of All Pair Partition Equi-Join

Reduce 함수는 R과 S에서 온 모든 레코드 쌍을 조인 조건을 보고 같으면 출력



```
// 입력 > table이름\t레코드ID\t컬럼2\t컬럼3
```

```
$ hdfs dfs -mkdir allpair_test
$ hdfs dfs -put data/equijoin-R-data.txt allpair_test
$ hdfs dfs -put data/equijoin-S-data.txt allpair_test
$ hadoop jar ssafy.jar allpair r s 2 allpair_test allpair_test_out
```

셀프-조인을 위한 모든 쌍 분할 알고리즘

- Self-Join은 한개의 입력 테이블 D에 들어있는 레코드들 간의 조인을 말함
- 입력 케이스들에 있는 레코드들을 m개의 파티션으로 나눔

셀프-조인을 위한 모든 쌍 분할 알고리즘

테이블이 S의 자신이 속한 파티션에만 보내야 함

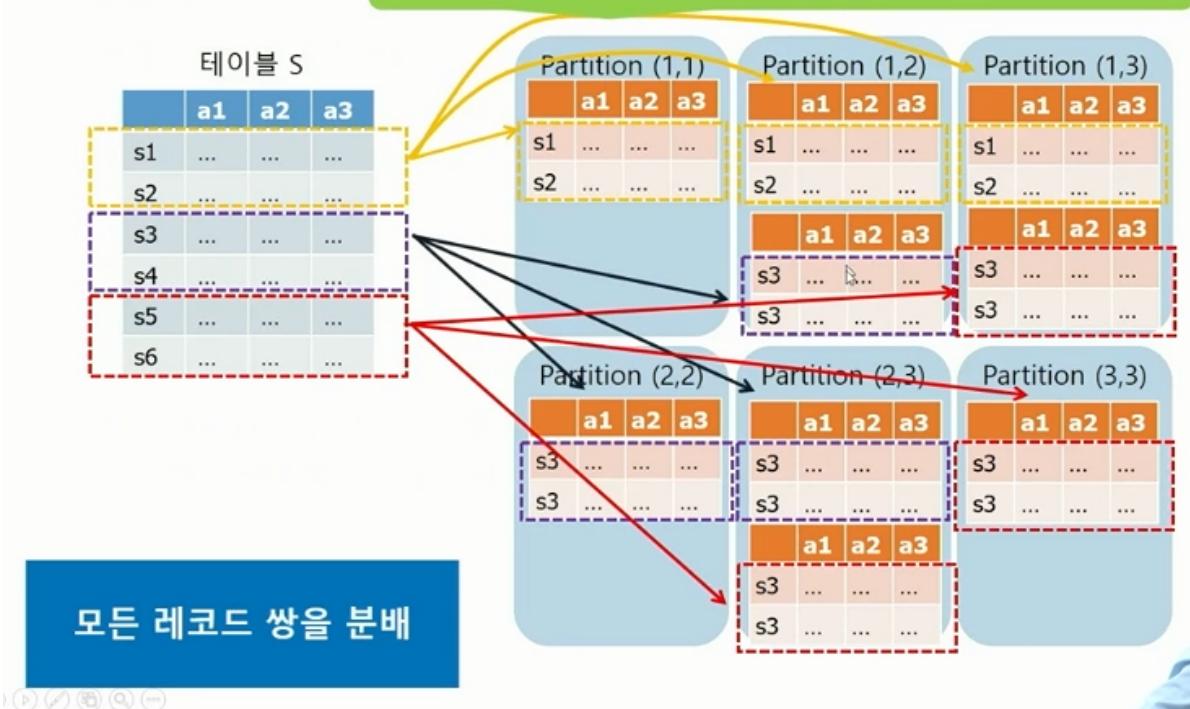
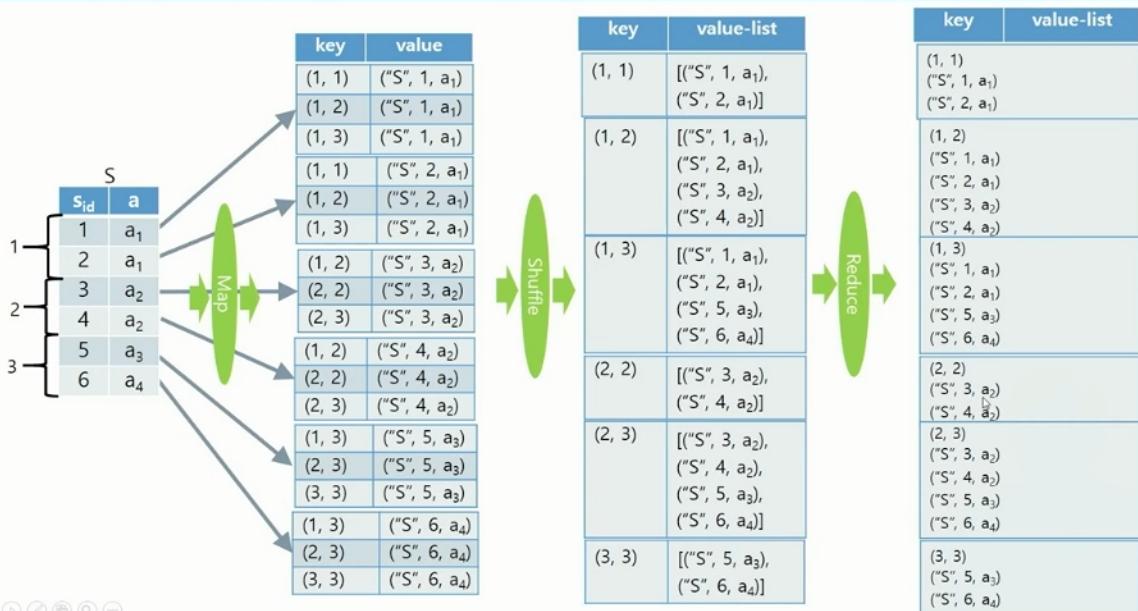


Illustration of All Pair Partition for Self-Join



```
$ hdfs dfs -mkdir allpairself_test
$ hdfs dfs -put data/equijoin-S-data.txt allpairself_test
$ hadoop jar ssafy.jar allpair s 2 allpairself_test allpairself_test_out
```

Common Item Counting for Every Pair of Sets

Common Item Counting for Every Pair of Sets

- ✓ Inverted index 를 만든다.

R		Inverted index	
RID	Items	Item	RIDs
R ₁	C D F	A	R ₂ , R ₃ , R ₅
R ₂	A B E F G	B	R ₂ , R ₃ , R ₄
R ₃	A B C D E	C	R ₁ , R ₃ , R ₄
R ₄	B C D E F	D	R ₁ , R ₃ , R ₄
R ₅	A E G	E	R ₂ , R ₃ , R ₄ , R ₅
		F	R ₁ , R ₂ , R ₄
		G	R ₂ , R ₅

- ✓ Inverted Index를 스캔하면서 각 item에 대한 리스트마다 들어있는 record의 모든 ID 쌍에 대해서 Hash table에 있는 카운트를 증가 시킨다.

Inverted index		Global hash table	
Item	RIDs	Candidate pair	Overlap
A	R ₂ , R ₃ , R ₅	(R ₂ , R ₃)	3
B	R ₂ , R ₃ , R ₄	(R ₃ , R ₅)	2
C	R ₁ , R ₃ , R ₄	(R ₂ , R ₅)	3
D	R ₁ , R ₃ , R ₄	(R ₃ , R ₄)	4
E	R ₂ , R ₃ , R ₄ , R ₅	(R ₂ , R ₄)	3
F	R ₁ , R ₂ , R ₄	(R ₁ , R ₃)	2
G	R ₂ , R ₅	(R ₁ , R ₄)	3

```
$ hdfs dfs -mkdir itemcount_test
$ hdfs dfs -put data/setjoin-data.txt itemcount_test
$ hadoop jar ssafy.jar itemcount itemcount_test itemcount_test_out
itemcount_test_out2
```

Illustration of Building Inverted Index (Phase 1)

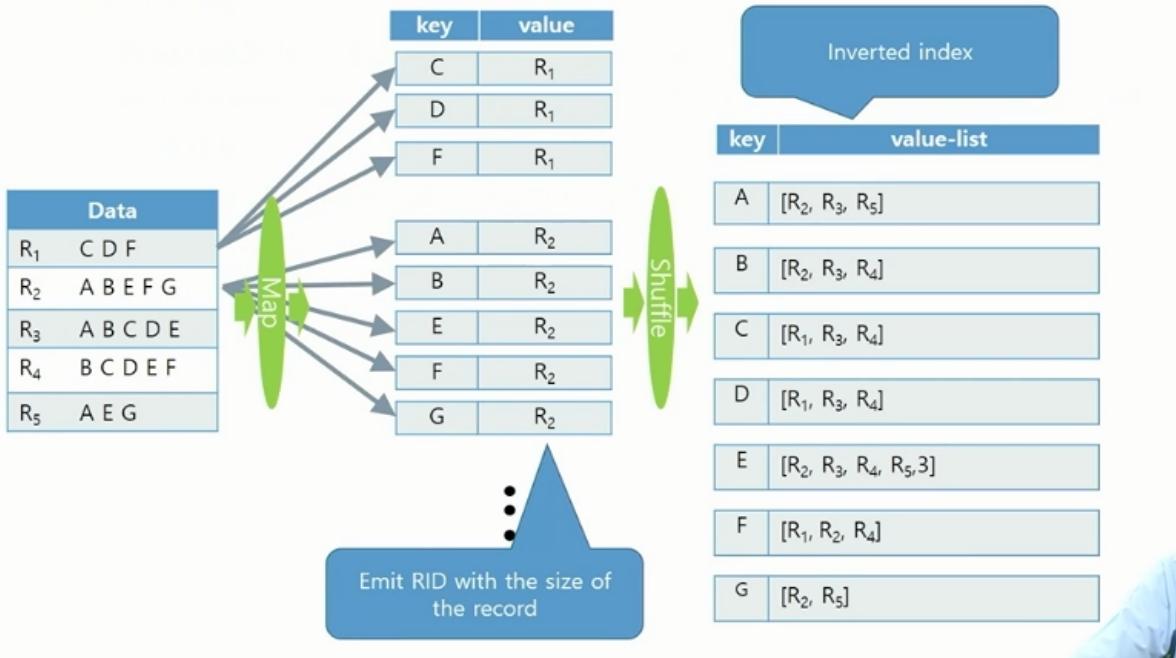
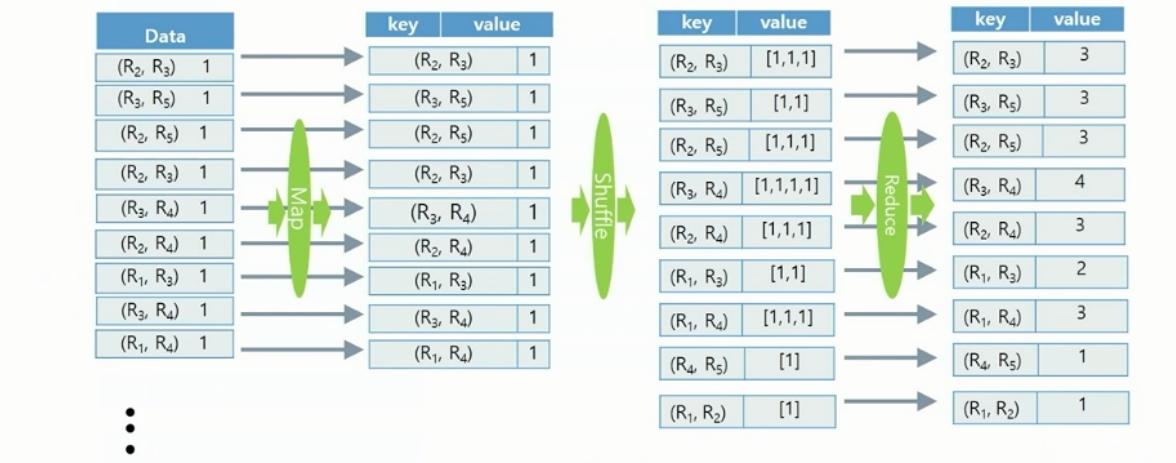


Illustration of Word Counting (Phase 2)



Top-K Closest Point Search Algorithm

- 질의 포인트와 점들로 구성된 데이터 셋이 있을 때 질의 포인트로부터 가장 가까운 K개 포인트를 뽑는다.
 - ex) 현재 내 위치로부터 가장 가까운 식당 K개를 뽑아줌
- Max-Heap 자료구조를 이용

Max Heap

- Binary tree 형태인데 어떤 노드에서 보든지 부모는 그 노드보다 더 크거나 같은 수를 가지고 자식 노드는 그 노드보다 더 작거나 같은 수를 가진다.
- 루트 노드 (root node)에 Max-heap에 들어있는 데이터 중에서 가장 큰 수가 들어간다.

How to Find Top-K Points

- 데이터를 읽어가면서 Max-Heap에는 현재까지 본 수들 중에서 가장 작은 K개의 숫자를 유지한다.
- $K = 8$ 일 때 먼저 8개의 수를 읽을 때 까지는 무조건 Max-Heap에 집어 넣어 Max-Heap을 만든다.
- 데이터를 하나씩 보면서 만약 peek에 있는 애보다 지금 들어오는 애가 작으면 heappop한번하고 heappush
- 만약 지금 들어오는 애가 더 크면 무시하면 된다.

Phase 1

- 점별로 위치와 질의 포인트로부터의 거리를 반환
- 각 머신별로 K개씩만 뽑아서 Reduce한다.

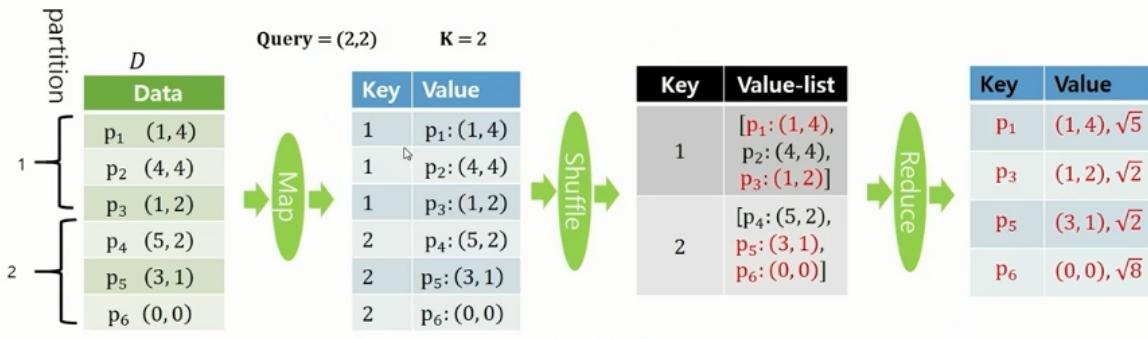
Illustration of Top-K Closest Point Search - Phase 1

Map 함수

- 각 점을 입력 받는다.
- 파티션을 m 개 나눌 때에 점의 파티션 아이디를 pid라 할 때, pid를 KEY로 하고 VALUE는 입력 포인트 그대로 하여 (KEY, VALUE) 쌍을 내보낸다.

Reduce 함수

- Query point와 VALUE-LIST에 있는 포인트들과 거리를 계산한 후에 가장 가까운 K 개만 출력.



Phase2

- 각 머신에서 모인 점들중 가장 거리가 짧은 K개 뽑는다.

Illustration of Top-K Closest Point Search - Phase 2

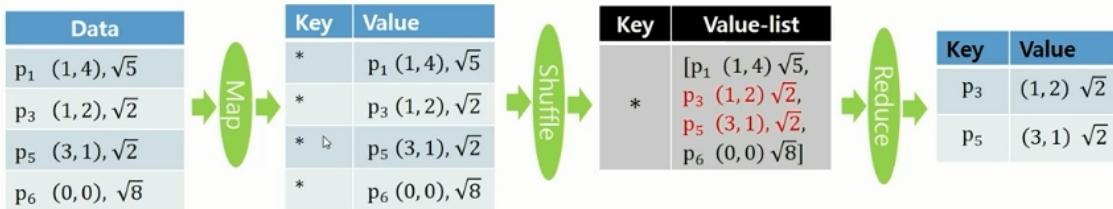
Map 함수

- 각 점을 입력 받는다.
- 똑같은 값 *를 KEY로 하고 VALUE는 입력 포인트 그대로 하여 (KEY, VALUE) 쌍을 내보낸다.

Reduce 함수

- Phase 1의 각 reduce 함수마다 출력한 K개의 포인트들을 다 모아서 그 중에 Top K 개만 뽑아 출력

Query = (2,2) K = 2



```
$ hdfs dfs -mkdir topksearch_test
$ hdfs dfs -put data/topksearch-data.txt topksearch_test
$ hadoop jar ssafy.jar topksearch 3 "1:1" 3 topksearch_test topksearch_test_out1
topksearch_test_out2
```