

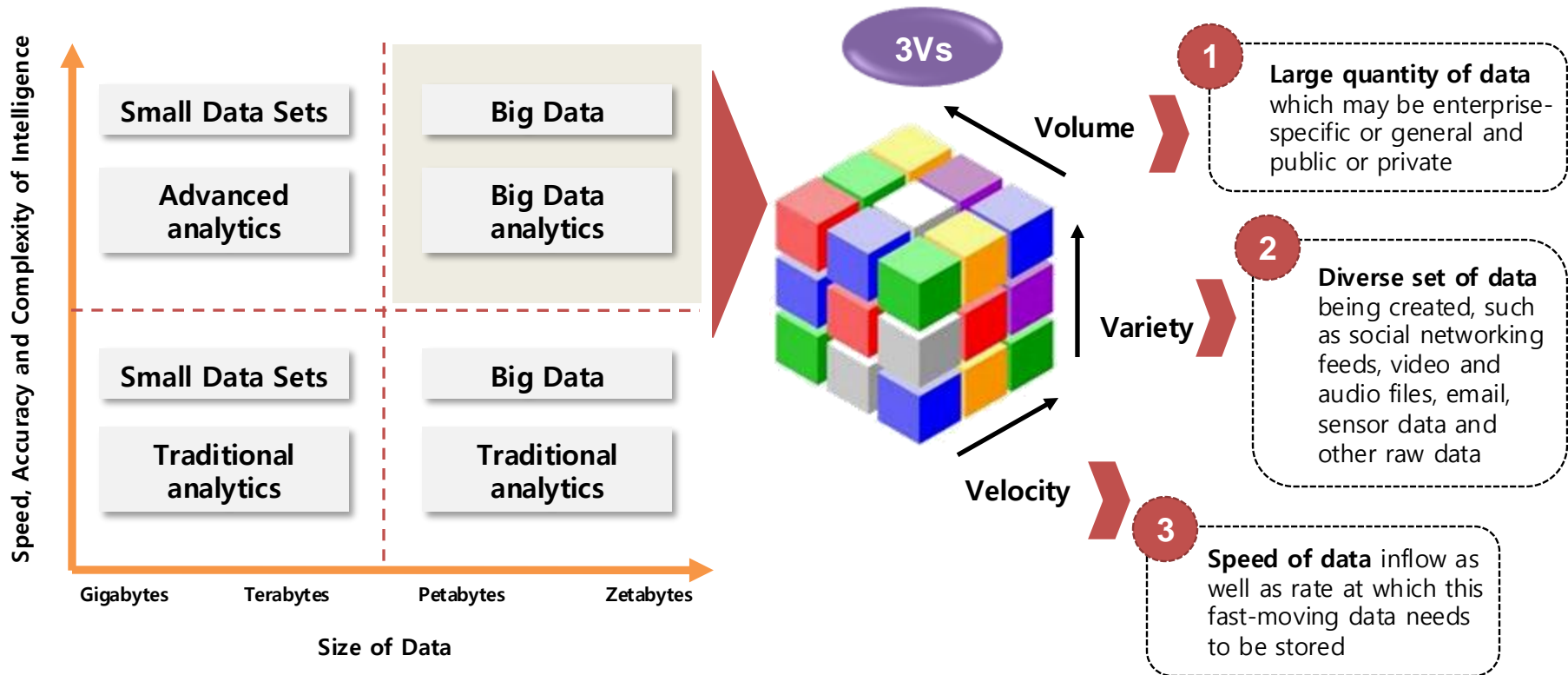


Big Data Mining: Parallel Computing & MapReduce

Younghoon Kim
(nongaussian@hanyang.ac.kr)

What is Big Data?

Big Data relates to rapidly growing, *Structured and Unstructured datasets* with sizes **beyond the ability of conventional database tools** to store, manage, and analyze them. In addition to its size and complexity, it refers to its ability to help in *"Evidence-Based" Decision-making*, having a high impact on business operations





What is Big Data Mining?

- Big data mining is the process of examining (rapidly increasing) large amounts of different data types, or big data, in an effort to uncover **hidden patterns, unknown correlations and other useful information.**

-- Margaret Rouse, WhatIs.com

"The technique that enables the analysis beyond the capacity of the current technology"



Big Data Mining

- What does Data Mining do?
 - Classification and clustering
 - Bayes nets, support-vector machines, decision trees, hidden Markov models, and many others
 - Information retrieval
 - PageRank idea, which made Google successful and which we shall cover later
 - Clustering where points that are “close” in this space are assigned to the same cluster
 - Pattern detection
 - the most extreme examples of a phenomenon and represents the data by these examples
 - E.g., Frequent itemsets

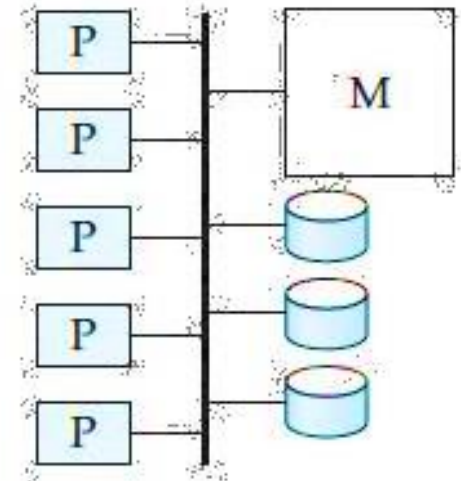
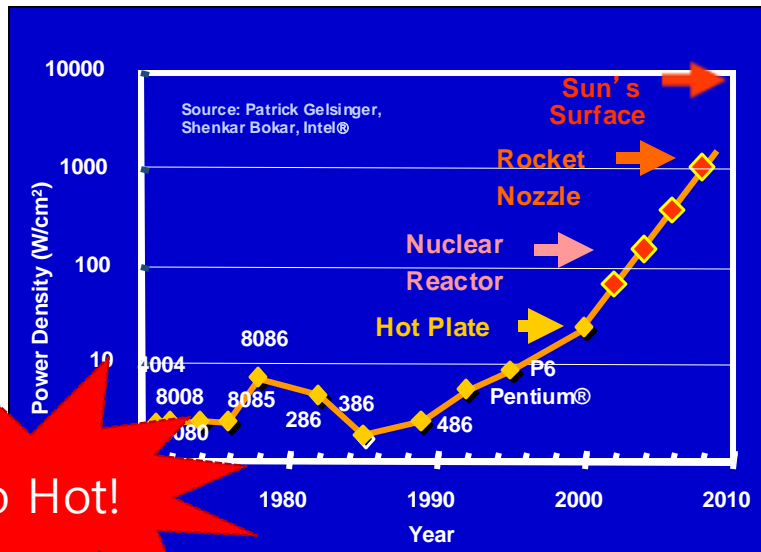
These analysis algorithms takes at least $O(n \log n)$, and most of them requires $O(n^2)$. Processing big data for data mining is practically impossible!



How to deal with Big Data?

- Sample & analysis with small data
- Find more efficient algorithms
- **Distribute a task & compute it in parallel**

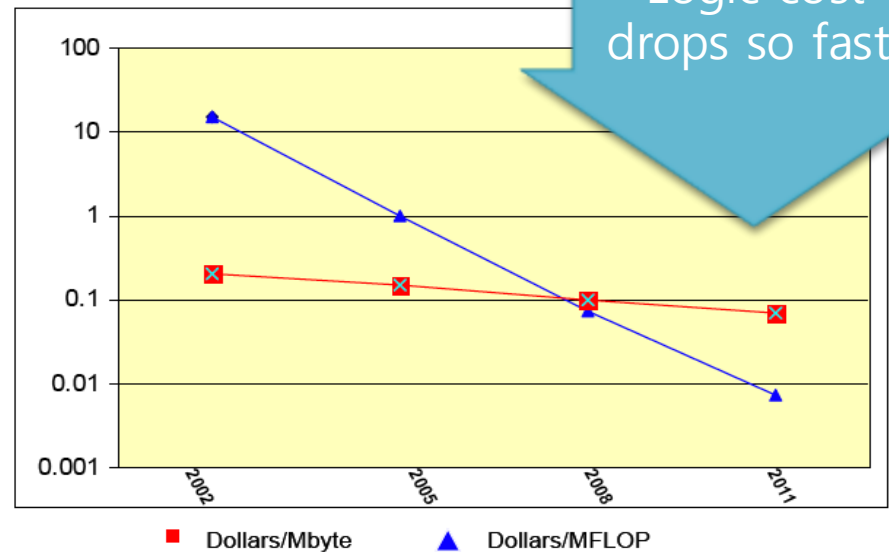
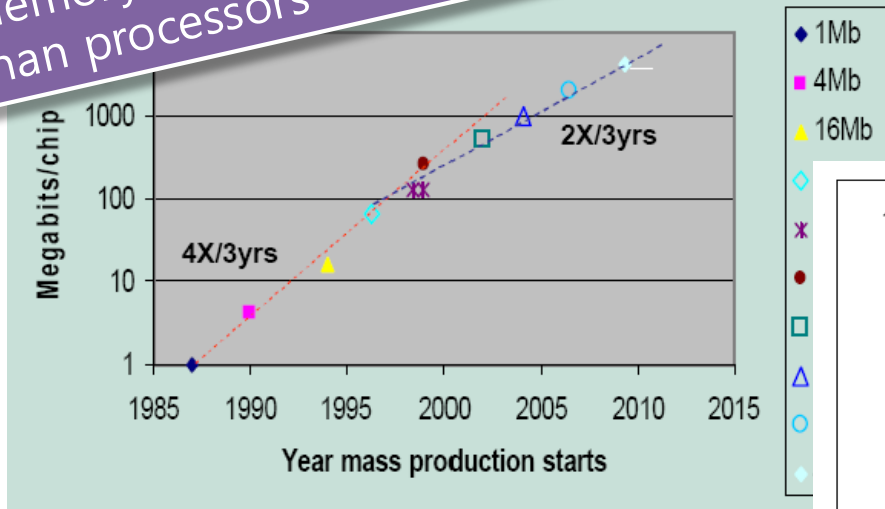
Age of Parallelism



Shared storage

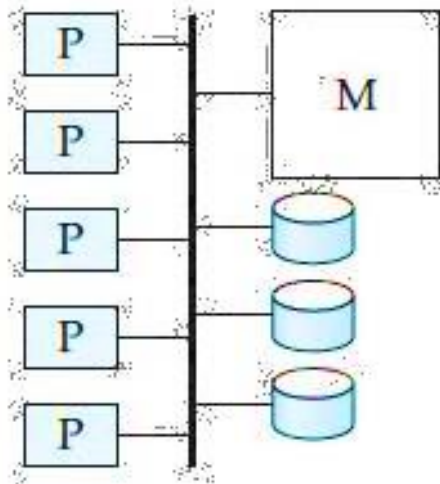
Age of Parallelism

Memory performance improves slower than processors

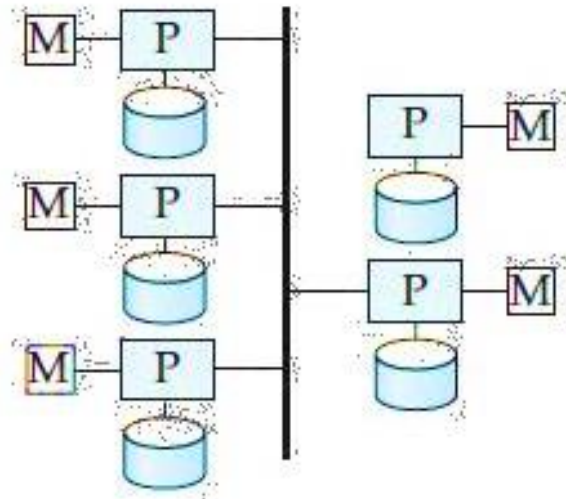


The cost to sense, collect, generate and calculate data is declining much faster than the cost to access, manage and store it

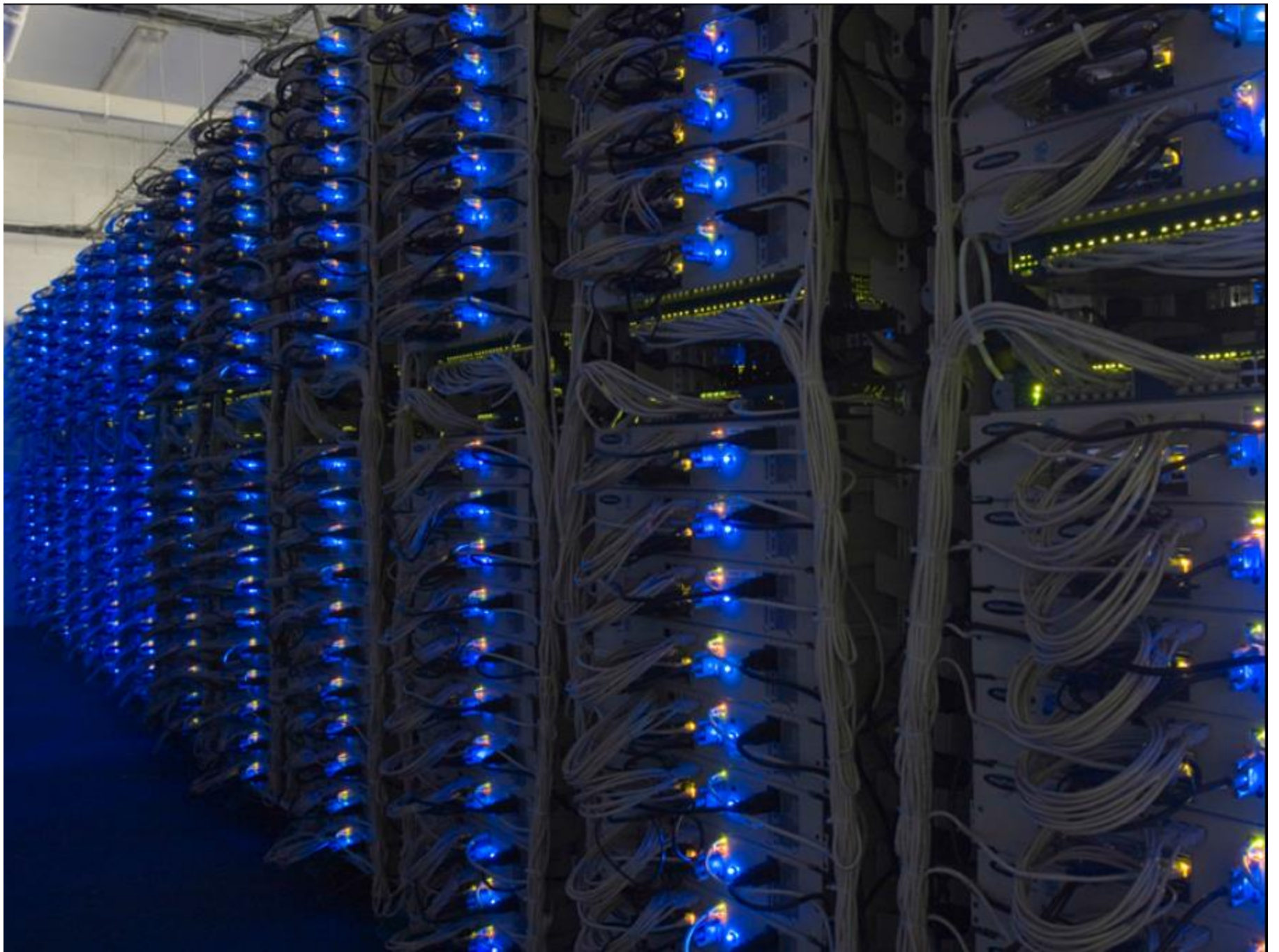
Architecture for MapReduce



Shared storage

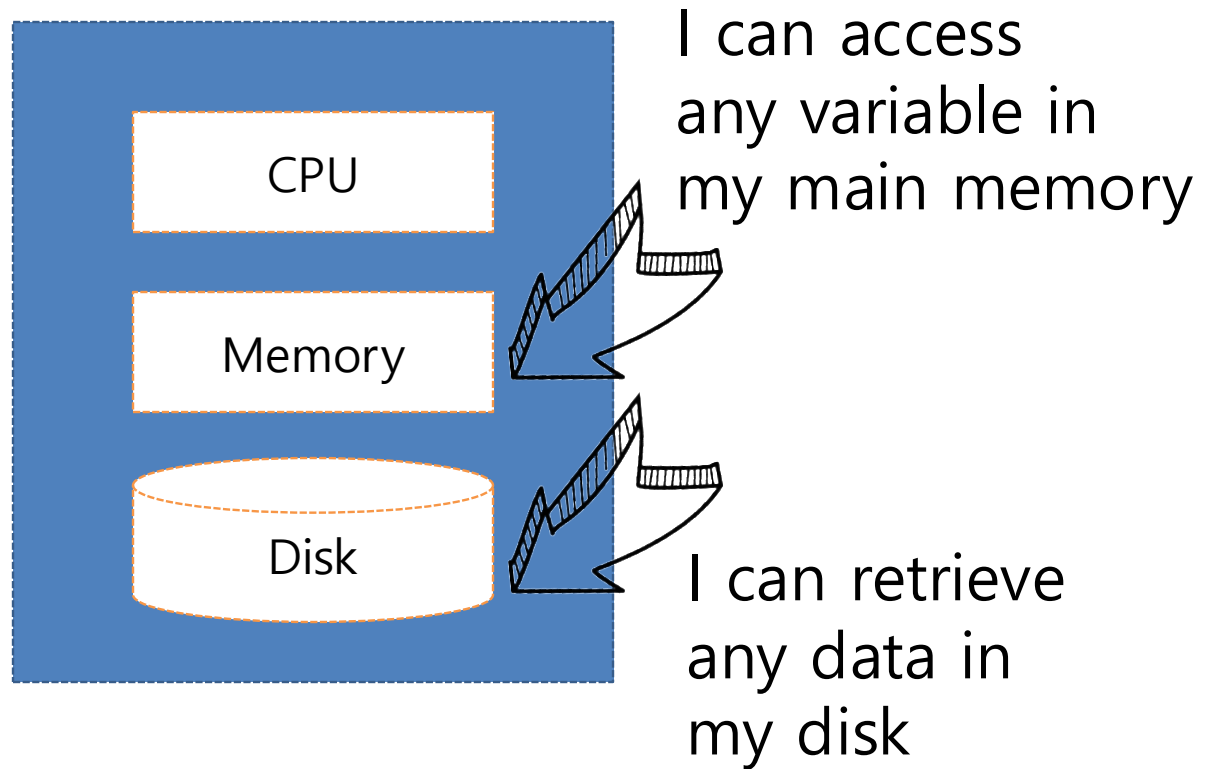


Shared nothing

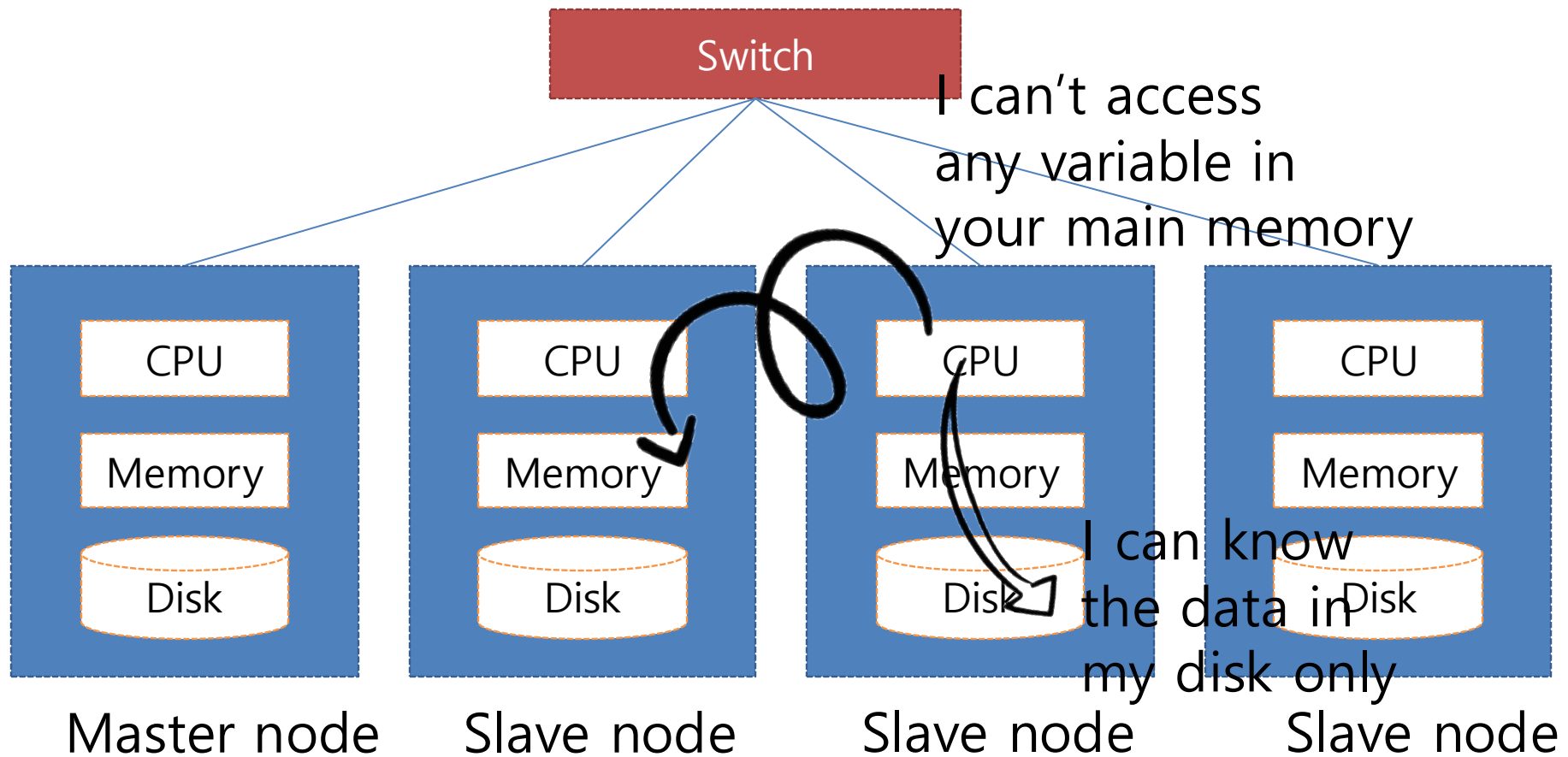


PARALLEL PROGRAMMING USING MAPREDUCE

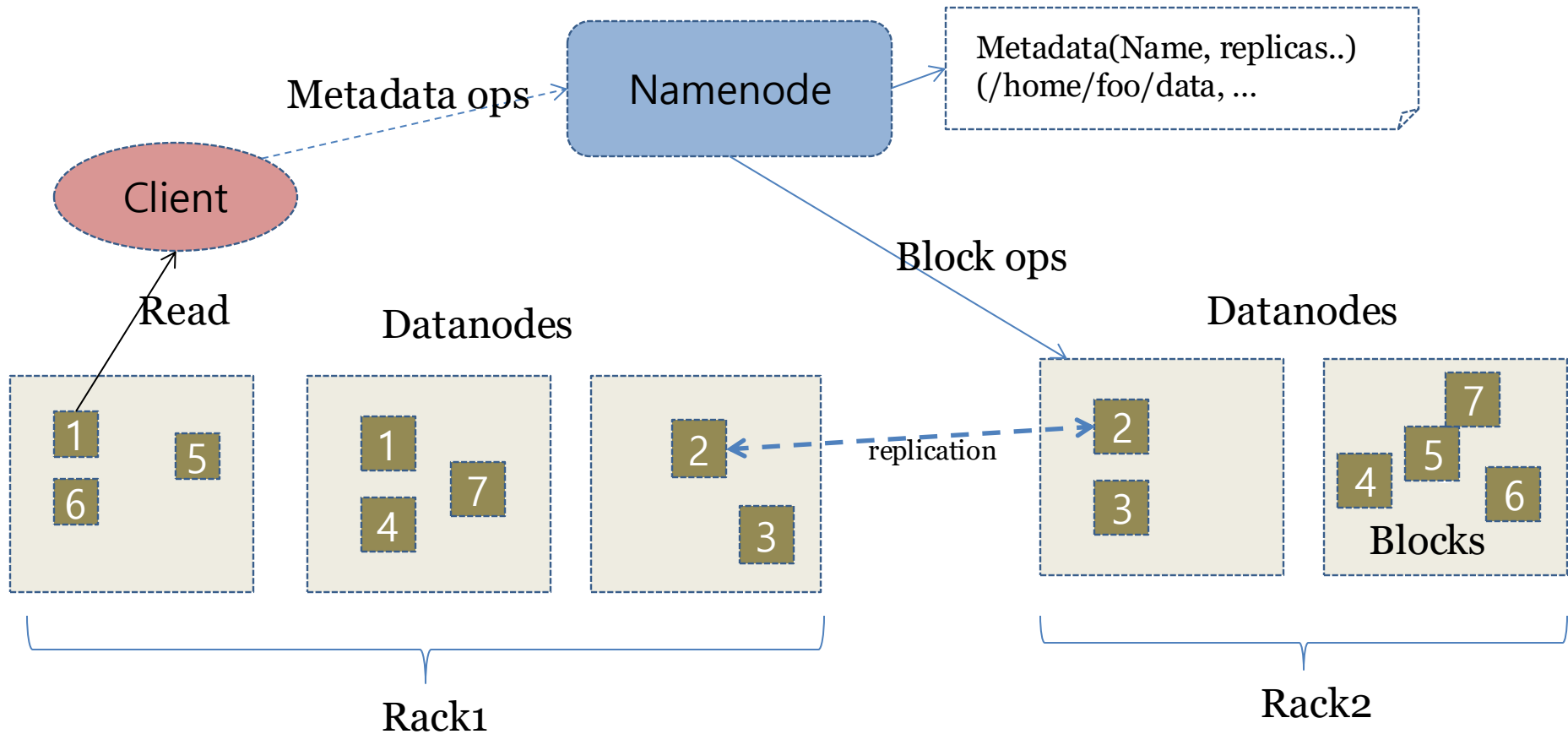
Single Node Architecture



Shared-Nothing Cluster Architecture



Distributed File System





Programming Model

- Functional programming
- Users implement interface of two functions:
 - **map (in_key, in_value) ->**
(out_key, intermediate_value)
list
 - **reduce**
(out_key, intermediate_value list)
->
out_value list

MAP/REDUCE EXAMPLE #1

(WORD COUNTING)

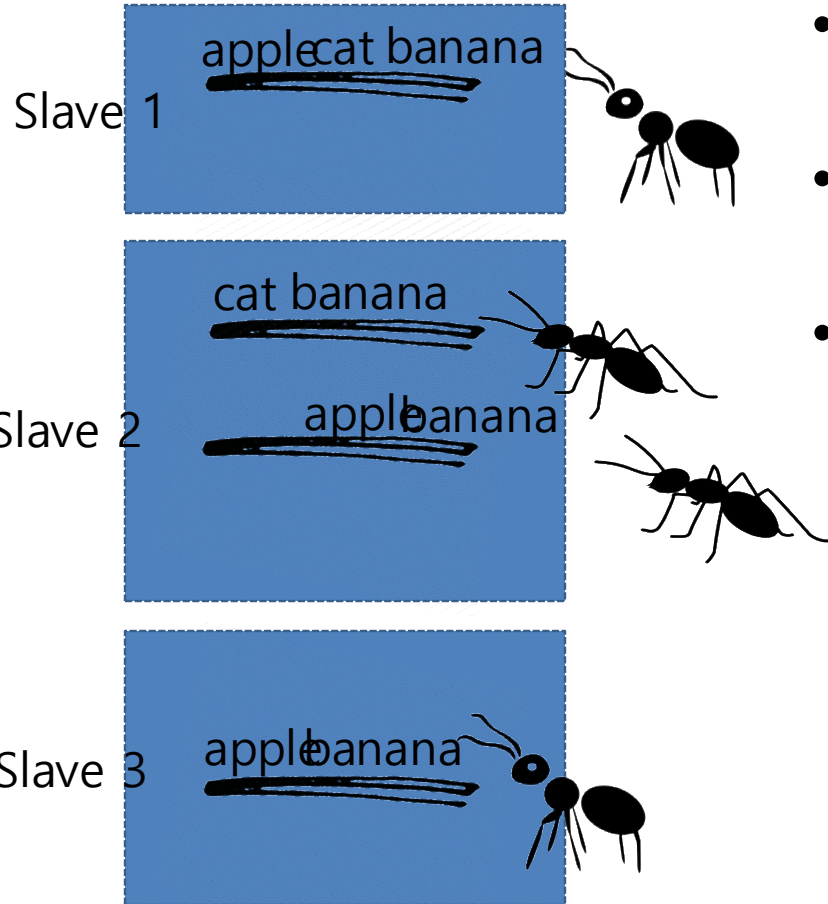


Word Counting

```
main () {  
    fd = open file ('big text file');  
    cnt = initialize a hash table;  
    while ( (line = read_a_line (fd)) != null) {  
        tokens = tokenize (line);  
        foreach (word in tokens) {  
            if (cnt[word] is defined) {  
                cnt[word] += 1;  
            }  
            else {  
                cnt[word] = 1;  
            }  
        }  
    }  
}
```



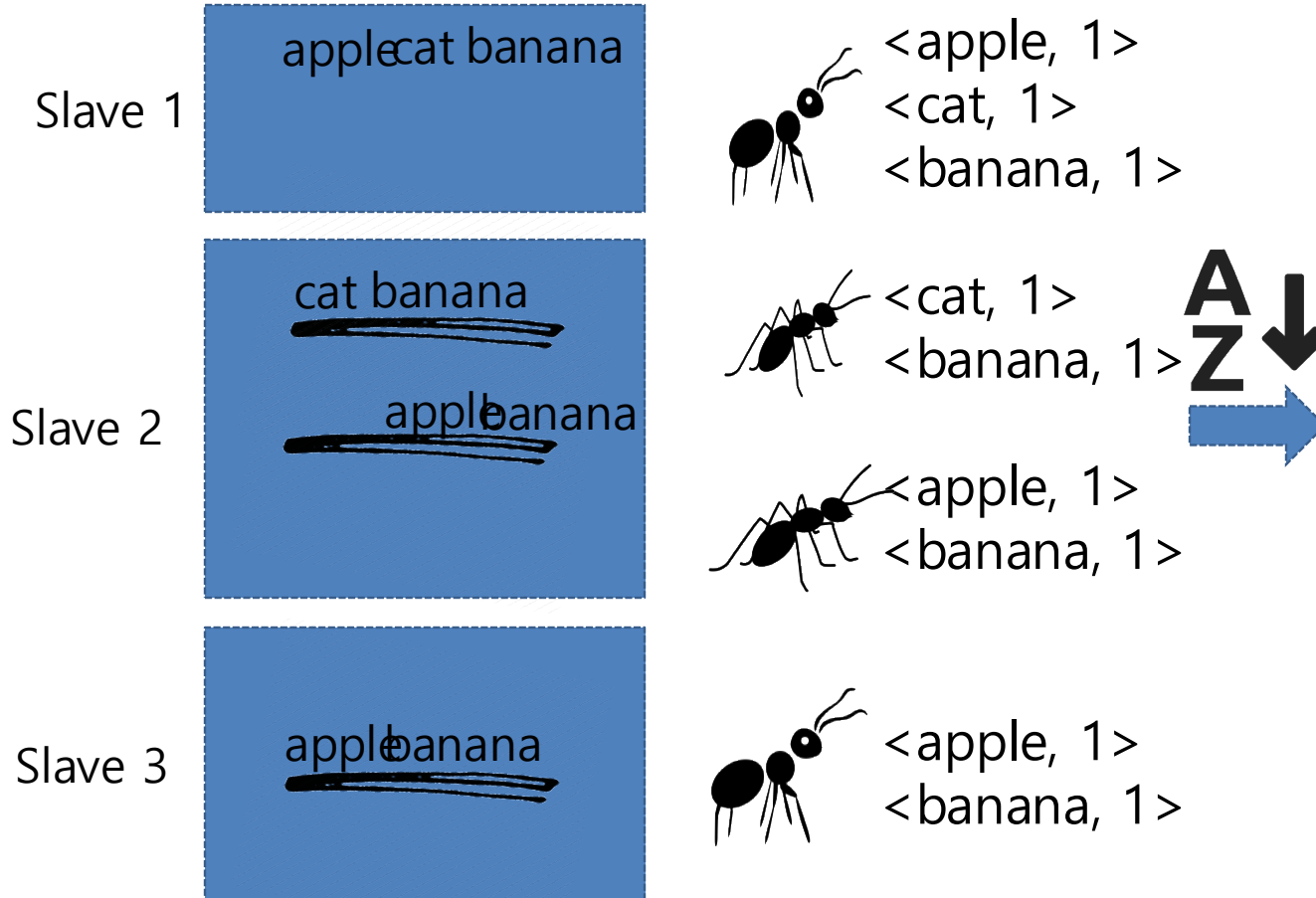
Word Counting with MapReduce



- I can read only a line
- We cannot use any hash table
- What I can do is

```
tokens ← tokenize (line);  
foreach (word in tokens)  
{  
    emit(word, 1);  
}
```

Word Counting with MapReduce





Word Counting with MapReduce



<apple, 1>

<apple, 1>

<apple, 1>

<banana, 1>

<banana, 1>

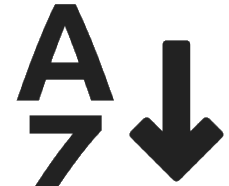
<banana, 1>

<banana, 1>

<cat, 1>

<cat, 1>

...



Word Counting with MapReduce

Slave 1

<apple, 1>
<apple, 1>
<apple, 1>



Slave 2

<banana, 1>
<banana, 1>
<banana, 1>
<banana, 1>



Slave 3

<cat, 1>
<cat, 1>



```
sum ← 1  
foreach (value in valuelist)  
{  
    sum ← sum + 1  
}  
emit(key, 1);
```

Word Counting with MapReduce

Slave 1

<apple, 1>
<apple, 1>
<apple, 1>



<apple, 3>

Slave 2

<banana, 1>
<banana, 1>
<banana, 1>
<banana, 1>



<banana, 4>

Slave 3

<cat, 1>
<cat, 1>



<cat, 2>



Hadoop MapReduce Programming in Java

```
public static class Map extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
        output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```



Hadoop MapReduce Programming in Java

```
public static class Reduce extends MapReduceBase implements
    Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
        IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) { sum += values.next().get(); }
        output.collect(key, new IntWritable(sum));
    }
}
```



Hadoop MapReduce Programming in Java

```
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
}
```



Spark

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
```

```
JavaPairRDD<String, Integer> counts  
  = textFile .flatMap(s -> Arrays.asList(s.split(" ")).iterator())  
              .mapToPair(word -> new Tuple2<>(word, 1))  
              .reduceByKey((a, b) -> a + b);
```

```
counts.saveAsTextFile("hdfs://...");
```



PySpark

```
text_file = sc.textFile("hdfs://...")

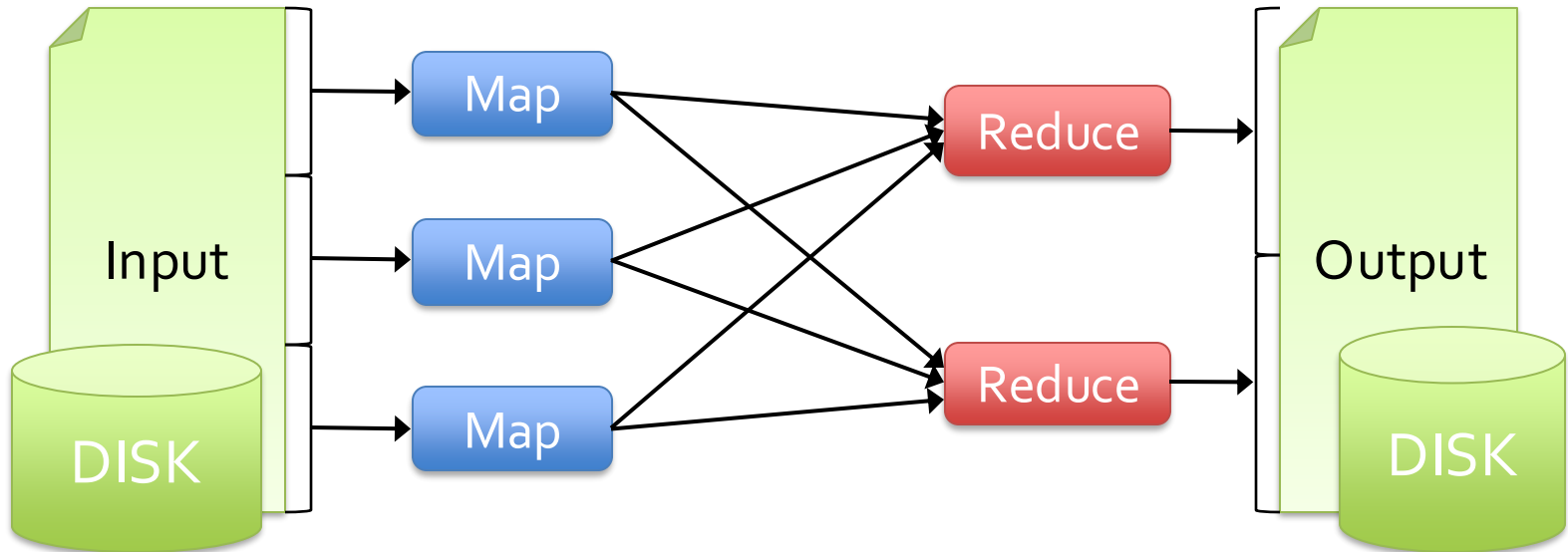
counts = text_file.flatMap(lambda line: line.split(" "))
                    .map(lambda word: (word, 1))
                    .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://...")
```

MAP/REDUCE EXAMPLE #2 (LOGISTIC REGRESSION WITH SPARK)

Motivation

- Popular MapReduce implementations such as Hadoop transform data flowing from stable storage to stable storage

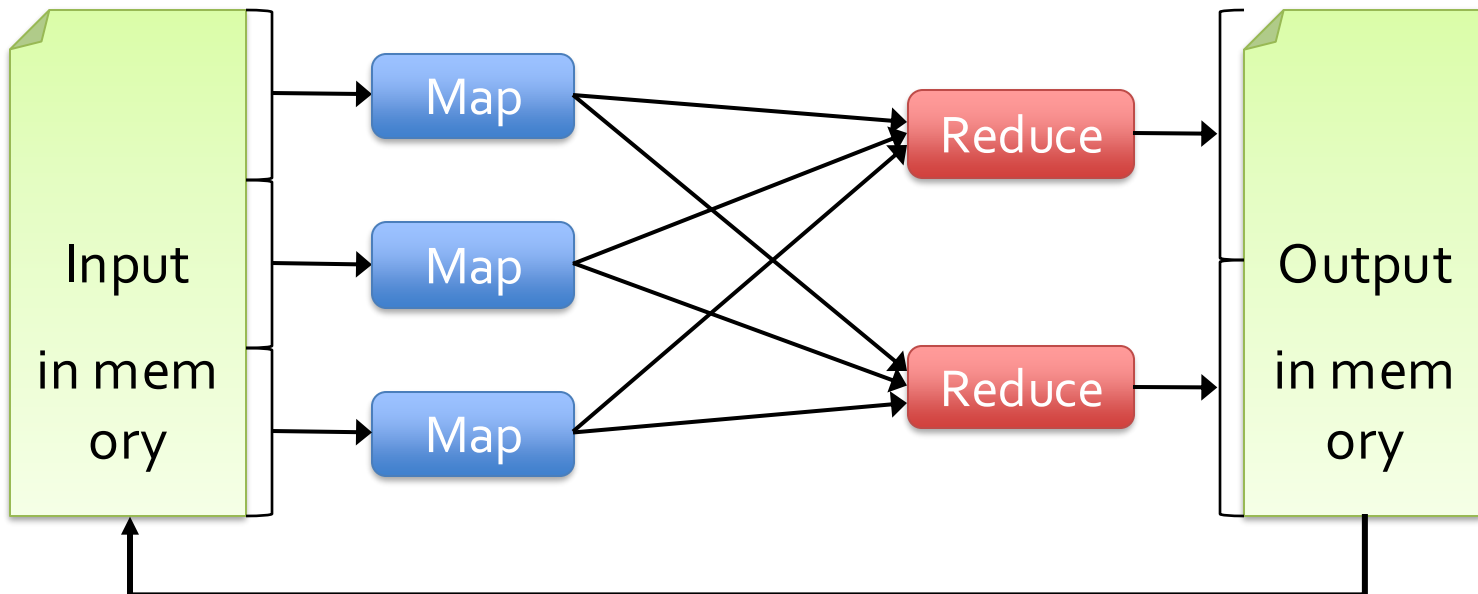




In-Memory Cluster Computing
for
Iterative and Interactive Applica
tions

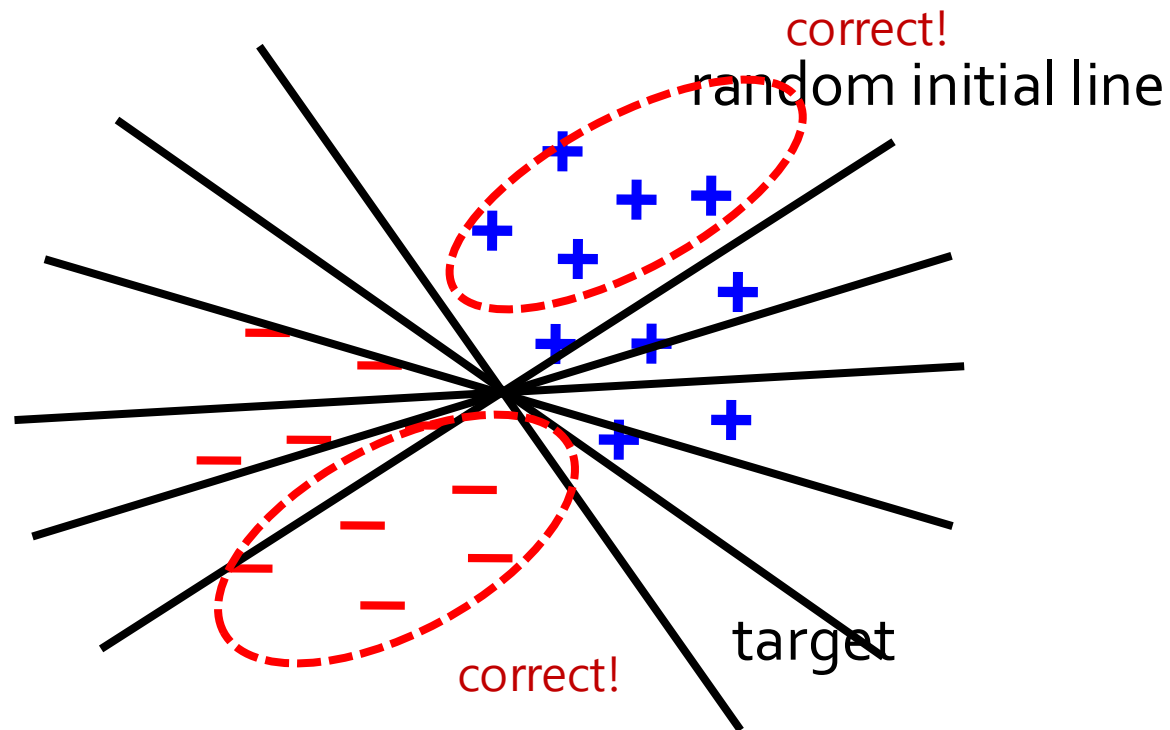
Motivation

- Efficient for applications that repeatedly reuse a working set of data:
 - **Iterative algorithms** (many in data mining and machine learning, e.g., PageRank, EM algorithms)



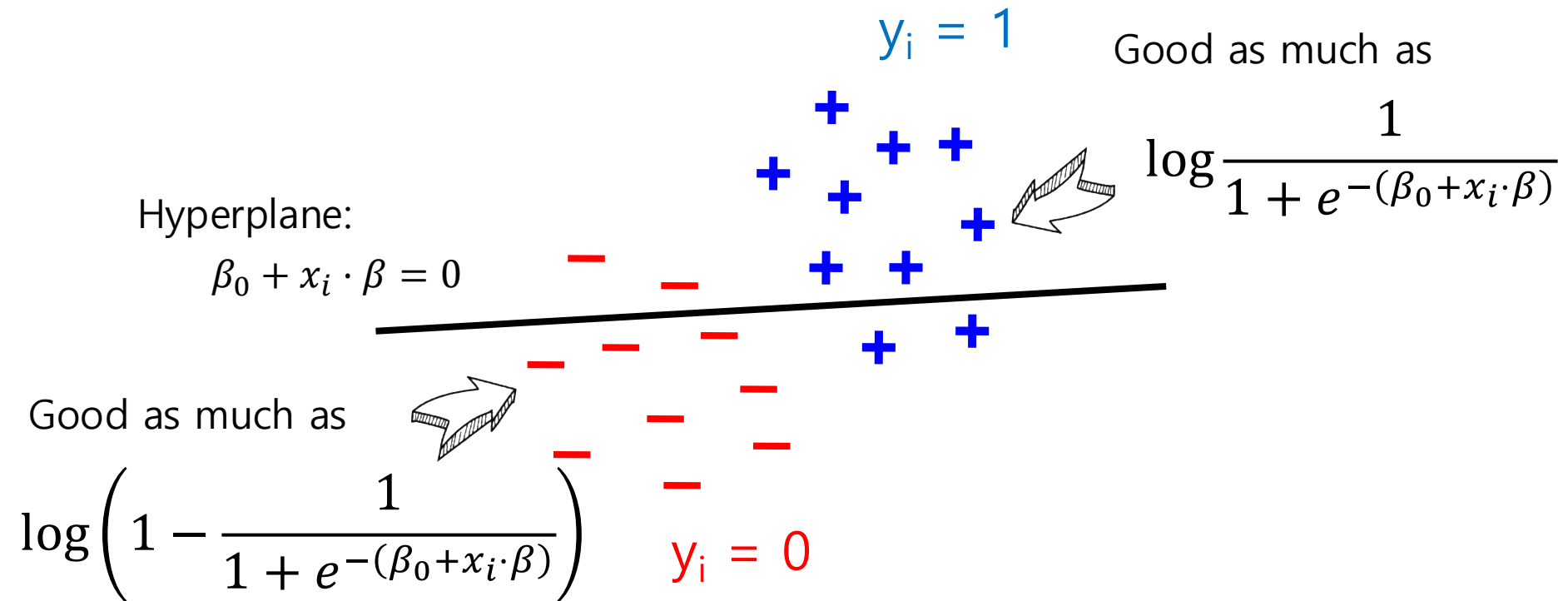
Logistic Regression

- Goal: find the **best line separating** two sets of points



Logistic Regression

- Goal: find the **best line separating** two sets of points





Optimization Problem

- Maximize

$$\sum_{i=1}^n y_i \cdot \log\left(\frac{1}{1 + e^{-(\beta_0 + x_i \cdot \beta)}}\right) + \sum_{i=1}^n (1 - y_i) \cdot \log\left(1 - \frac{1}{1 + e^{-(\beta_0 + x_i \cdot \beta)}}\right)$$

- Gradient descent method

$$\beta^{(t+1)} \leftarrow \beta^{(t)} + \alpha \sum_{i=1}^n \left(y_i - \frac{1}{1 + e^{-(\beta_0 + x_i \cdot \beta)}} \right) x_i$$

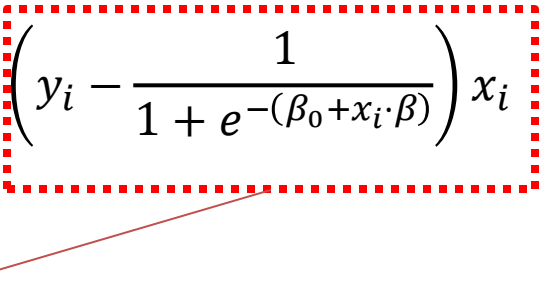
Sum of values
calculated with
each data points

Big training data takes long time to
compute the gradient

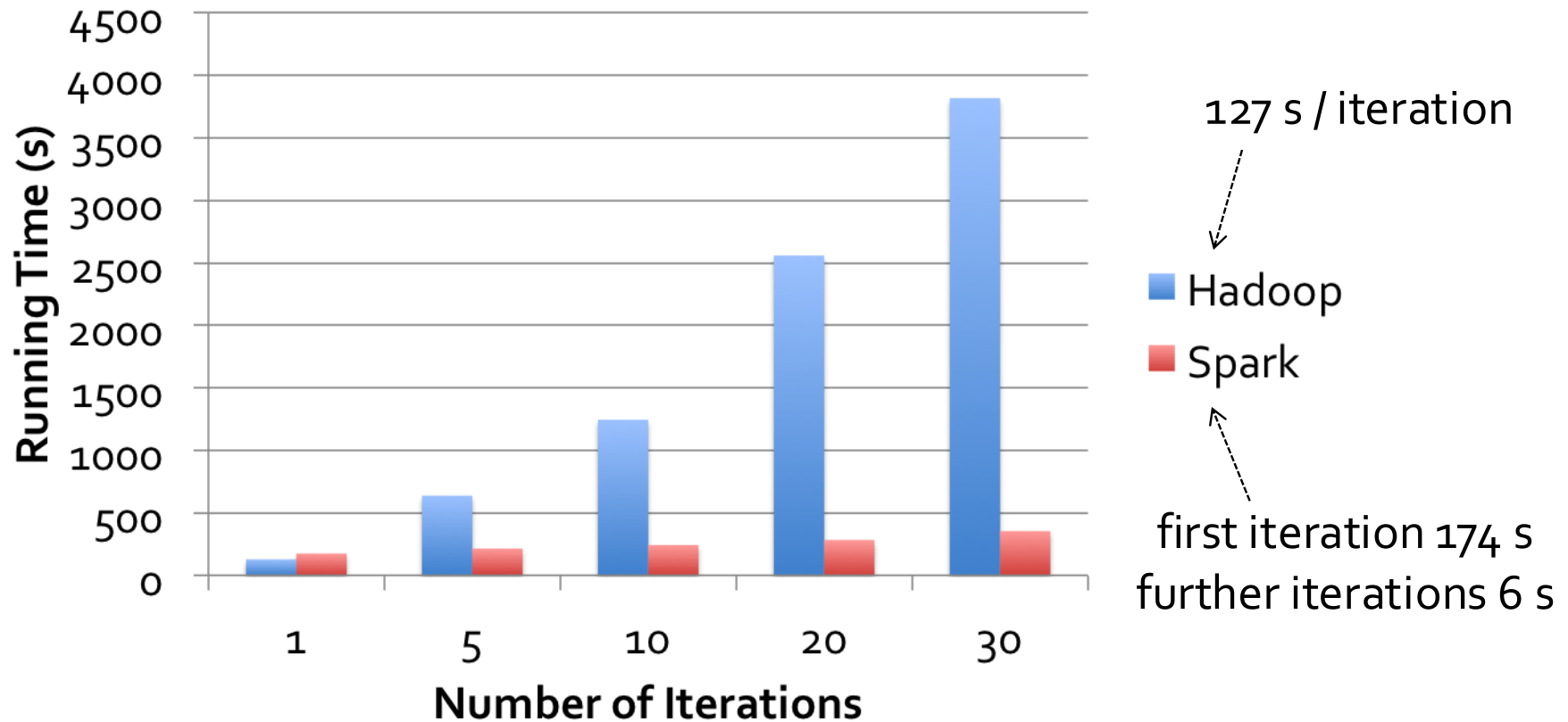


Logistic Regression Code

```
1. data = spark.textFile(...).map(readPoint).cache()
2. w = random(D)
3. for _ in range(ITERATIONS):
4.     gradient = data.map(
5.         lambda x, y:(y - 1 / (1 + exp(-(w dot p.x)))) * x
6.     ).reduceByKey(lambda a, b: a+b)
7.     w += gradient
8. }
9. print("Final w: {}".format(w))
```

$$\beta^{(t+1)} \leftarrow \beta^{(t)} + \alpha \sum_{i=1}^n \left(y_i - \frac{1}{1 + e^{-(\beta_0 + x_i \cdot \beta)}} \right) x_i$$


Logistic Regression Performance





Summary

Programming with
MapReduce is
not a choice, but a necessity.

Don't worry. It is fun!