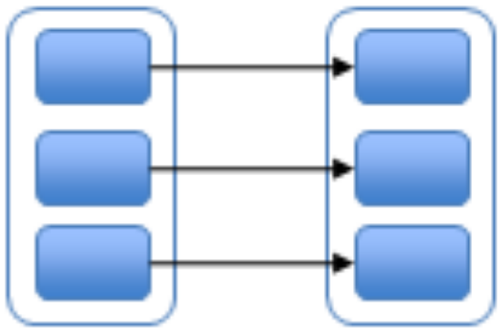
Developing Spark Applications

Life of a Spark Application

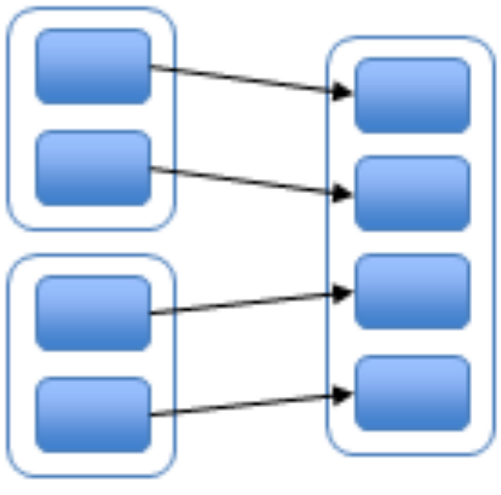
1. **Create** some input RDDs from external data or parallelize a collection in your driver program.
2. **Lazily transform** them to define new RDDs using transformations like `filter()` or `map()`
3. Ask Spark to `cache()` any intermediate RDDs that will need to be **reused**.
4. **Launch actions** such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.

Communication Costs

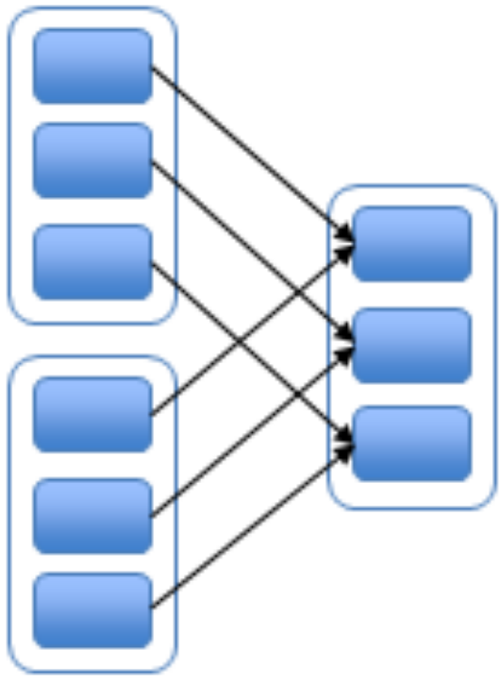
"Narrow" deps:



map, filter

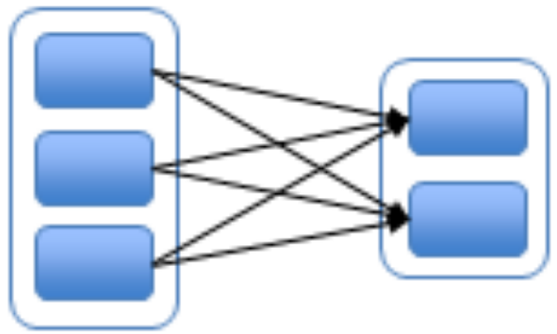


union

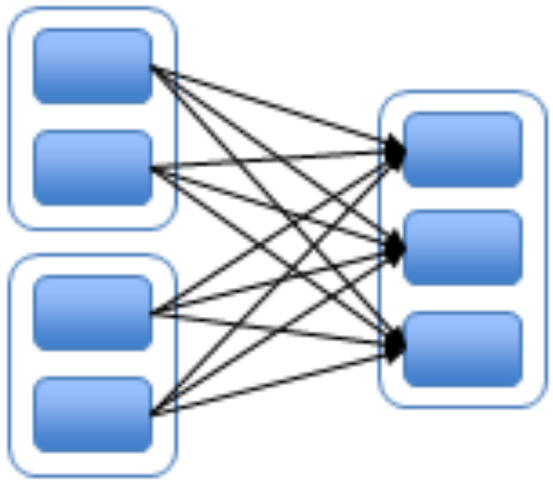


join with inputs co-partitioned

"Wide" (shuffle) deps:



groupByKey



join with inputs not co-partitioned

Shuffling

- **Shuffling** requires to solve the **sorting problem**, defined as follows:
  - There is **some ordering** in the data that we want to sort
  - The data **starts** out **distributed** on p machines
  - **All-to-all** communication is unavoidable
  - The best we can hope for is that the data are **sent** over the network **only once**
  - The **sorted** data set is stored in a **distributed** manner

Distributed Sorting

- Each machine sends a **uniform sample** of its data to the sorting driver
- The driver uses the **samples** from each machine **weighted** by the number of data to computer an **approximate distribution** of the data to be sorted
- This distribution is used to compute p **split points** where the number of data between two consecutive points is approximatively the same
- The split points are **broadcasted** to the nodes
- Each machine does a **local sort** on its data
- Using the split points, each machine builds an **index** of which data goes to which machine
- Machine i asks machine j for its portion of data
  - **All-to-all** communication occurs in this step
- Each machine does a ***p*-way merge** of p different sorted data sets that is has received from each machine

Uniform Sampling (I)

- A stream σ is a sequence of elements $(s_1, ..., s_m)$ where each $s_i \in \{1, . . . , n\}$. Typically, the length of the stream is unknown and n is large but $n << m$.
- Uniform Sampling: given a stream σ, for each $i \in \{1,...,n\}$, define $f_i = \{j : s_j = i\}$, the number of occurences of value i in the stream. We can define a probability distribution P on $\{1, ..., n\}$ by $p_i = f_i/m$. The goal is to output a single element sampled in accordance to the distribution P.
- For example, consider the stream $\sigma = (1, 3, 4, 5, 5, 2, 1, 1, 1, 7)$. In this case $p_1 = 4/10$, $p_2 = 1/10$, $p_3 = 1/10$, $p_4 = 1/10$, $p_5 = 2/10$, $p_7 = 1/10$. The goal would be to output a random variable in accordance to this distribution.

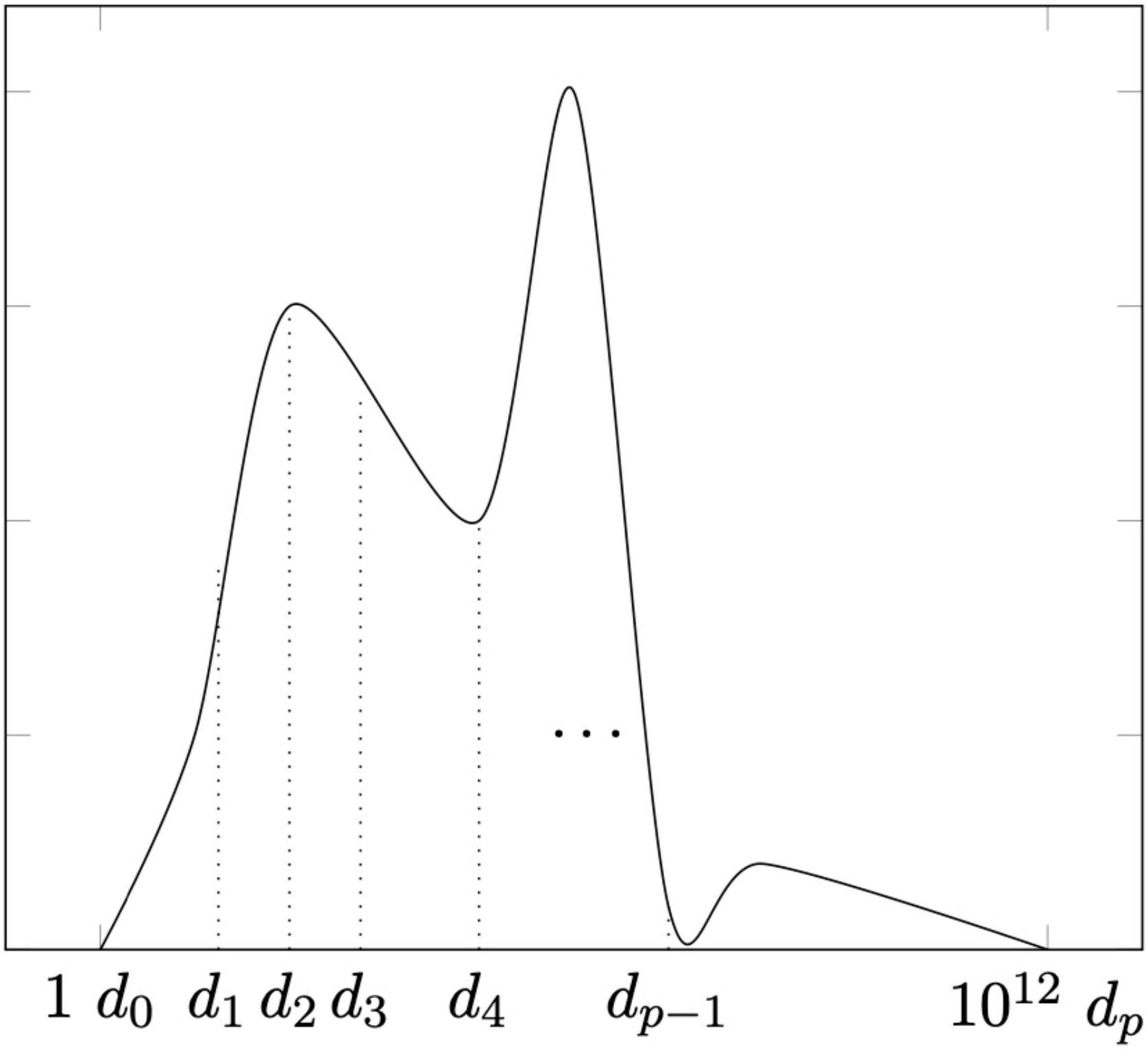
Uniform Sampling (II)

- Naive Sampling Algorithm:
  - *Initialization*: for each value i = 1, ..., n, initialize a counter $f_i \leftarrow 0$. Initialize $m \leftarrow 0$.
  - *Streaming*: for each s_j, if $s_j = i$, set $f_i \leftarrow f_i + 1$. Increment $m \leftarrow m + 1$.
  - *Output*: calculate $p_i = f_i/m$. Choose an i according to $P = (p_1, \ldots, p_n)$.
- Storing the entire stream would take O(m log n) space
- In this case we store n counters which range between 0 and m so require O(n log m) space in total.
- As $n << m$, this is better but not the best we can do.

Uniform Sampling (III)

- Uniform Sampling Algorithm:
  - *Initialize*: set $x \leftarrow$ null, $k \leftarrow 0$.
  - *Streaming*: for each s_j, increment $k \leftarrow k + 1$. With probability $1/k$, set $x \leftarrow s_j$. Otherwise, do nothing.
  - *Output*: return x.
- In this case we store 1 counter for x and 1 counter for j, so $O(\log n + \log m)$ space in total.

Computing Split Points



Sorting (shuffle)

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

MLlib Algorithms

- **classification**: logistic regression, linear SVM, naïve Bayes, least squares, classification tree
- **regression**: generalized linear models (GLMs), regression tree
- **collaborative filtering**: alternating least squares (ALS), non-negative matrix factorization (NMF)
- **clustering**: k-means||
- **decomposition**: SVD, PCA
- **optimization**: stochastic gradient descent, L-BFGS

Gradient Descent (I)

- We are going to implement **gradient descent** on Spark
- **Separable** objective functions of the form

$$\min_w \sum_{i=1}^{n} f_i(w)$$

- $w \in \mathbb{R}^d$ is the vector of **minimizing parameters**
- $f_i(w)$ is the **loss function** applied to a training point i
  - Can be linear or non-linear
  - Can be least squares or neural network

Gradient Descent (II)

- We assume that the **number of training points** n is large, e.g., trillions
- We assume that the **number of parameters** d can fit on a single machine's RAM, e.g., millions
- Start at a random vector x_0
- Then

$$x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$$

- Where $\nabla F_i(\cdot)$ is the gradient **vector**

Gradient Descent (III)

- A training point is stored on a single machine
- Our data points are arranged in a matrix, divided among machines row-by-row
- The matrix is encoded in text
- Input data are not moved among machines

```
def parsePoint(row):
  tokens = row.split(' ')
  return np.array(tokens[:-1], dtype=float), float(tokens[-1])
w = np.zeros(d)
points = sc.textFile(input).map(parsePoint).cache()
```

Gradient Descent (III)

- To compute a gradient we need to perform a computation on each machine and a summation across machines

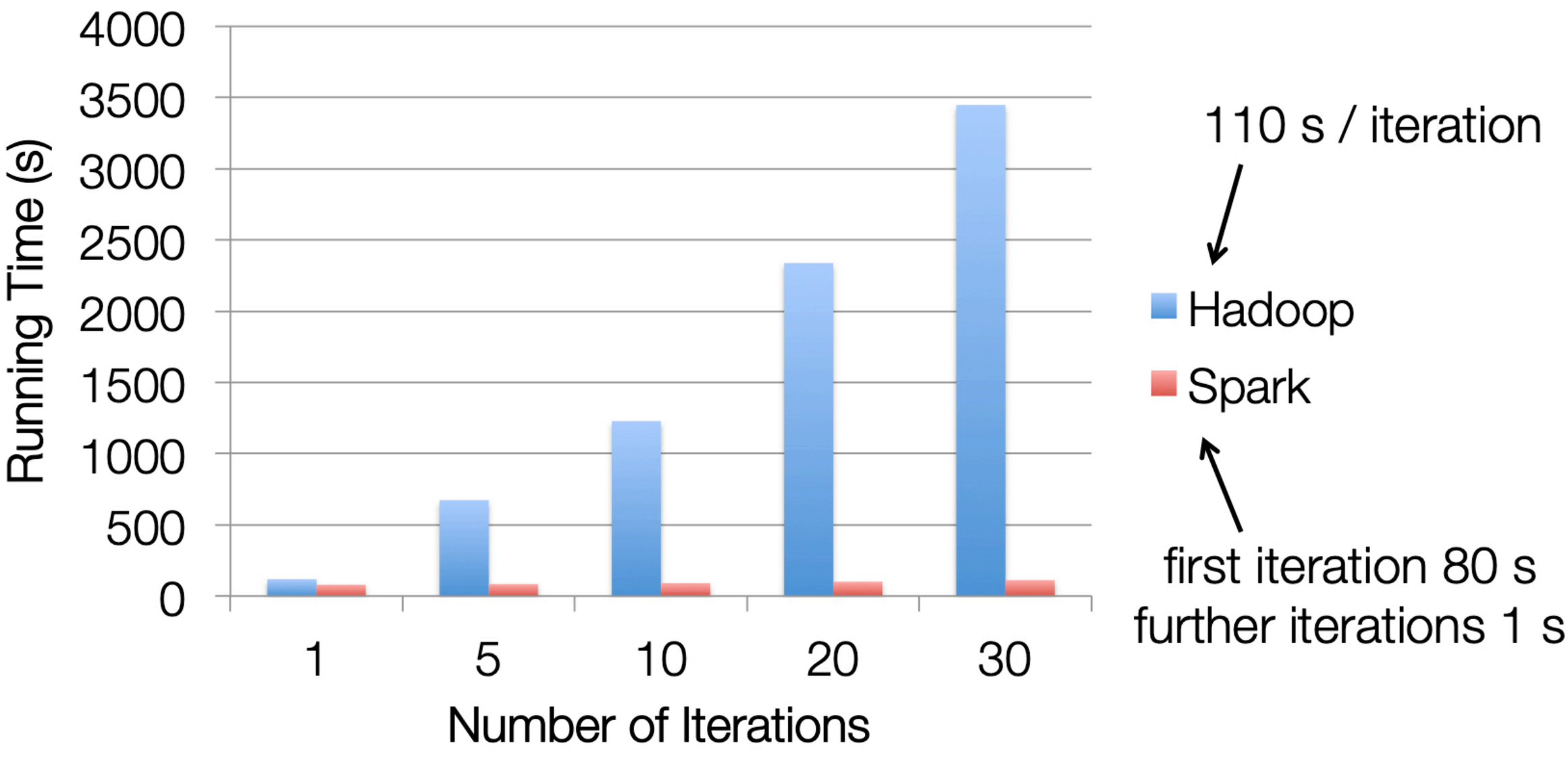
```
def compute_gradient(p, w):
    ...

gradient = points.map(lambda p: compute_gradient(p, w))
                 .reduce(lambda x, y: x + y)
```

- On a single machine we can have multiple CPUs, so we can avoid to store w for each CPU using `broadcast`

Gradient Descent (IV)

```
points = sc.textFile(input).map(parsePoint).cache()

w = np.zeros(d)

w_br = sc.broadcast(w)

for i in range(num_iterations):

    gradient = points.map(lambda p: compute_gradient(p, w_br.value))

                     .reduce(lambda x, y: x + y)

  w -= alpha * gradient

  w_br = sc.broadcast(w)
```

Results with logistic regression



100 GB of data on 50 m1.xlarge EC2 machines

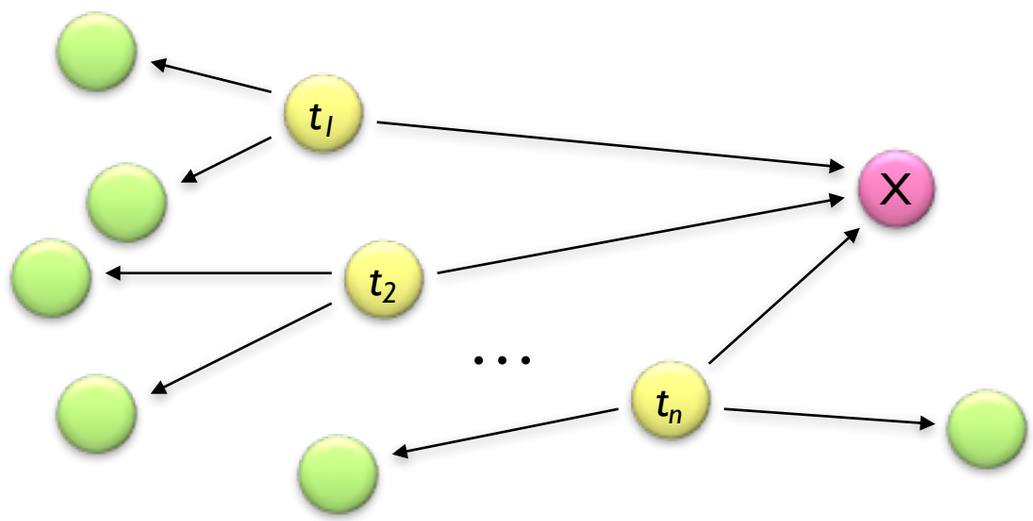
Pagerank (I)

- Random Surfers
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- Pagerank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- Web Ranking
  - One of thousands of features used in web search

Pagerank (II)

- Given page x with inlinks t_1, ..., t_n, where
  - $C(t)$ is the out-degree of link t
  - α is probability of random jump
  - N is the total number of nodes in the graph

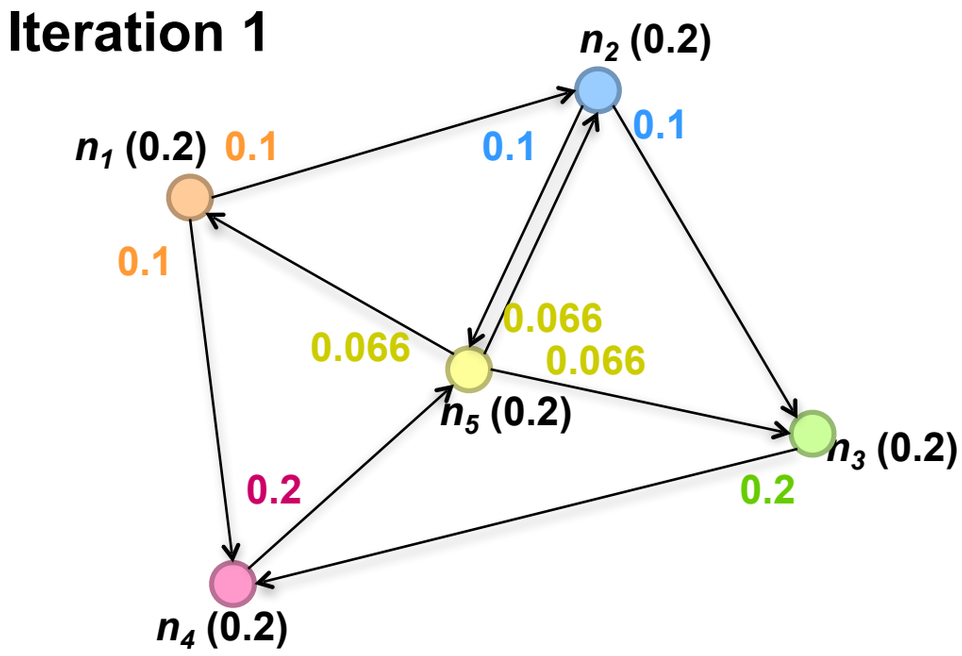
$$PR(x) = \alpha \left(\frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$

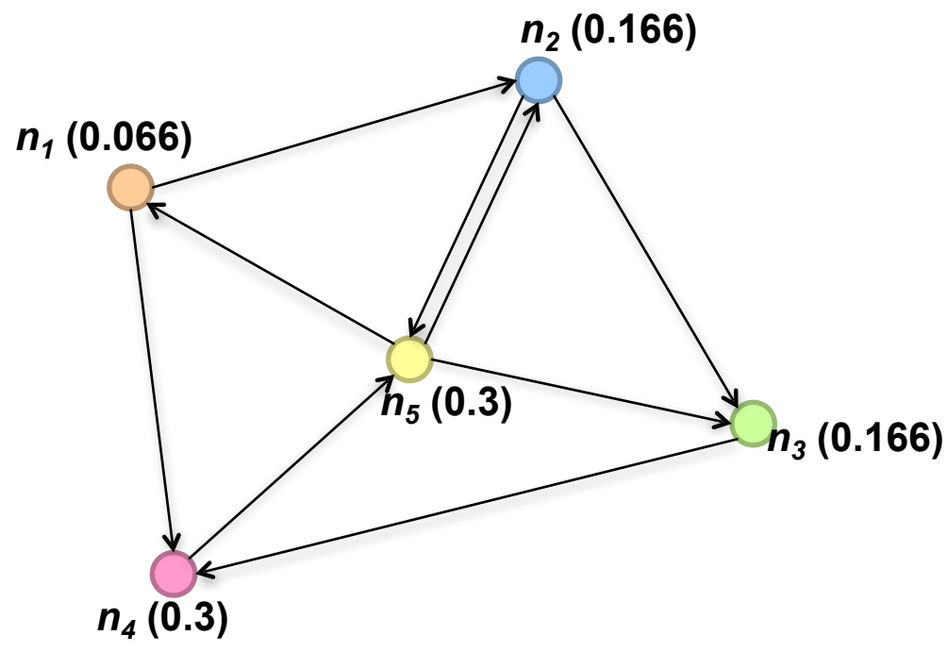


Pagerank Algorithm Sketch

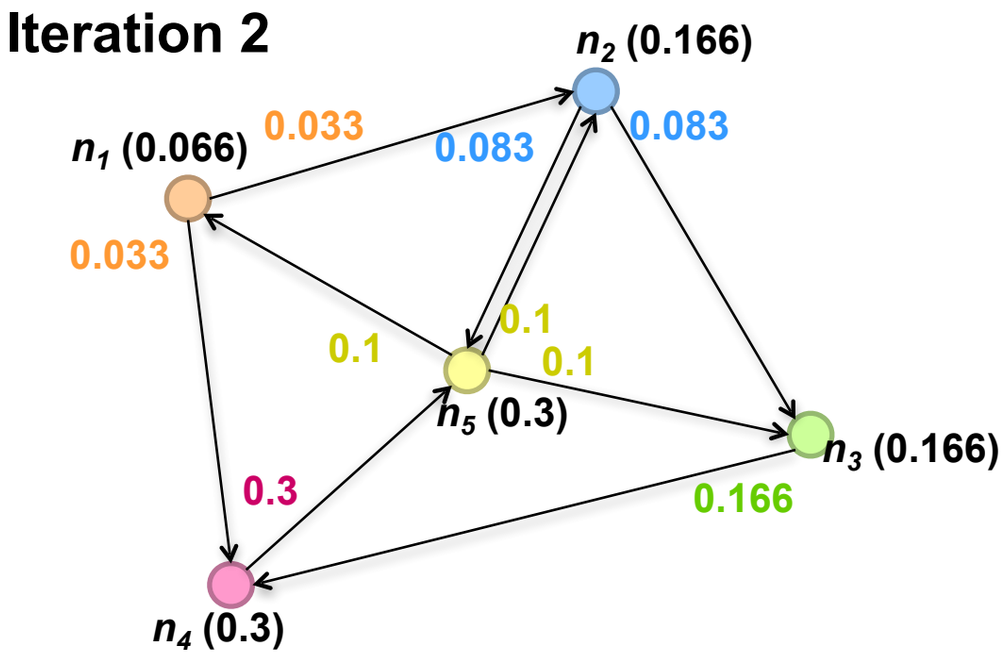
- Start with seed *PR(i)* values
- Each page distributes *PR(i)* mass to all pages it links to
- Each target page adds up mass from in-bound links to compute next *PR(i)*
- Iterate until values converge

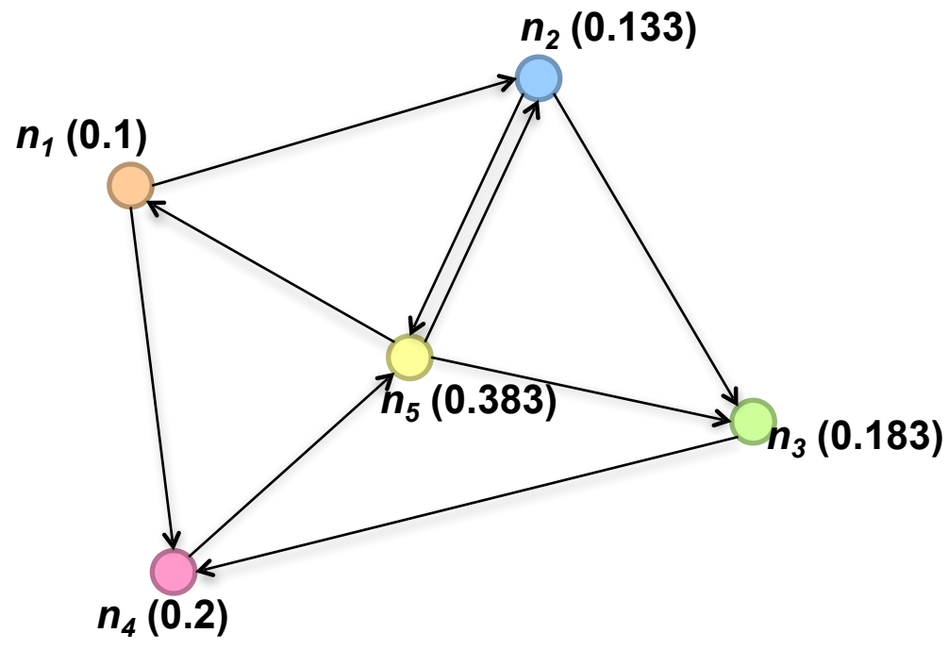
Simple Example (I)



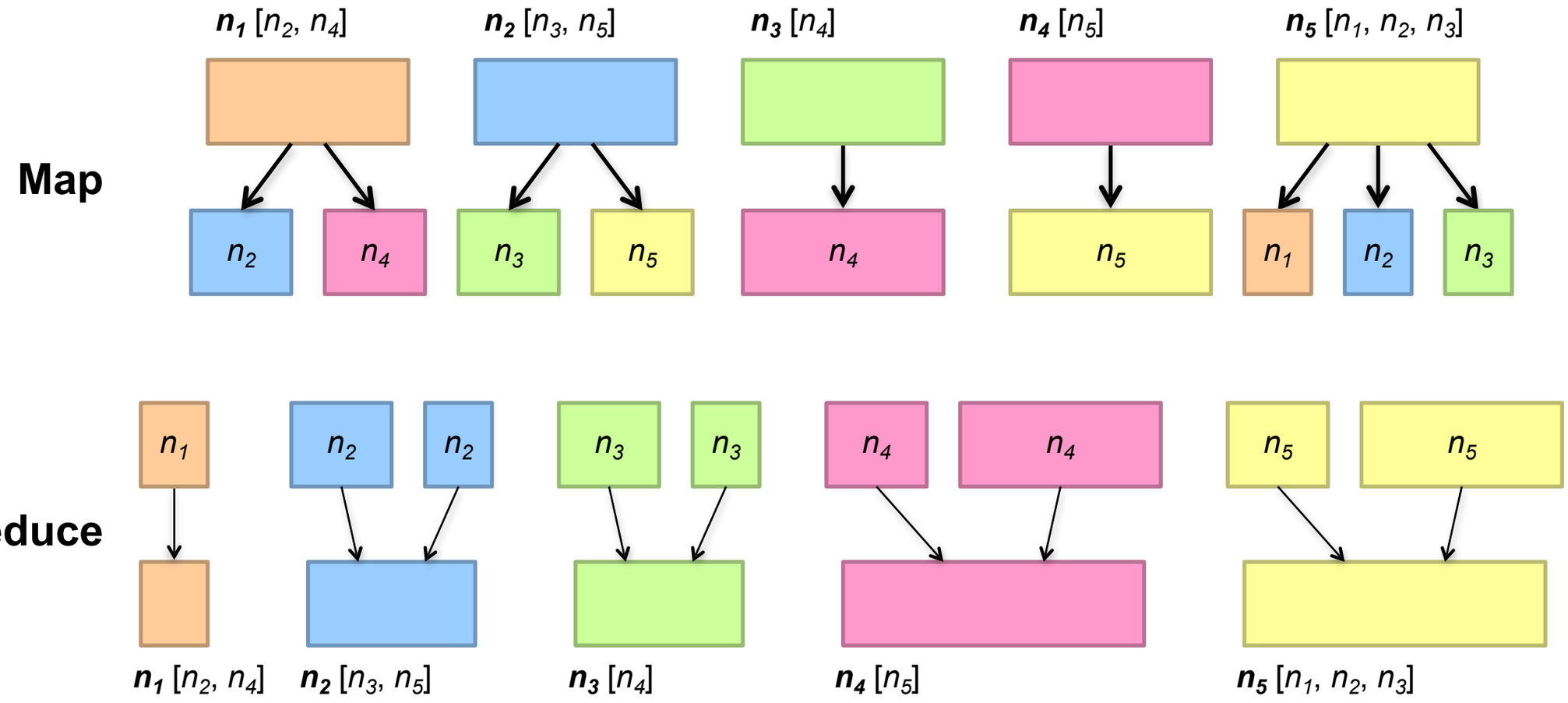


Simple Example (II)





Pagerank In MapReduce (I)



Pagerank In MapReduce (II)

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sums incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Pagerank (I)

- Given directed graph, compute node importance.
- Two RDDs:
  - Neighbors (a sparse graph/matrix)
  - Current guess (a vector)
- Simple algorithm:
  A. Start with each page at a rank of 1
  B. On each iteration:
     1. Have page p contribute $rank_p / |neighbors_p|$ to its neighbors
     2. Set each page's rank to $0.15 + 0.85 \times contribs$
- Math available at

  https://www.rose-hulman.edu/~bryan/googleFinalVersionFixed.pdf

Pagerank (II)

```
def computeContribs(urls, rank):
    """Calculates URL contributions to the rank of other URLs."""
    num_urls = len(urls)
    for url in urls:
        yield (url, rank / num_urls)


def parseNeighbors(urls):
    """Parses a urls pair string into urls pair."""
    parts = re.split(r'\s+', urls)
    return parts[0], parts[1]
```

Pagerank (III)

```
links = lines.map(lambda urls: parseNeighbors(urls))
                .distinct().groupByKey().cache()
ranks = links.map(lambda url, neighbors: (url, 1.0))
# Calculates and updates URL ranks continuously using PageRank algorithm.
for iteration in range(num_iterations)):
    # Calculates URL contributions to the rank of other URLs.
    contribs = links.join(ranks).flatMap(
        lambda url, (urls, rank): computeContribs(urls, rank))
    # Re-calculates URL ranks based on neighbor contributions.
    ranks = contribs.reduceByKey(lambda x, y: x + y)
                    .mapValues(lambda rank: rank * 0.85 + 0.15)
# Collects all URL ranks and dump them to console.
for (link, rank) in ranks.collect():
    print("%s has rank: %s." % (link, rank))
```