Map Reduce
Design Patterns

Caveat

- MapReduce is a **framework** not a tool

  QUELLO CHE VOGLIAMO PROGETTARE DEVE RIENTRARE IN QUELLO CHE IL FRAMEWORK PUO' FARE

  - You have to fit your solution into the framework of map and reduce
  - It might be challenging in some situations
- Need to take the algorithm and break it into **filter/aggregate steps**
  - Filter becomes part of the map function
  - Aggregate becomes part of the reduce function
- Sometimes we may need **multiple** Map Reduce **stages**
- Map Reduce is **not a solution to every problem**, not even every problem that profitably can use many compute nodes operating in parallel!
- It makes sense only when:
  - files are **very large** and are **rarely updated**
  - We need to **iterate** over all the files to **generate** some interesting property of the data in those files

Design Patterns

- Intermediate data reduction
- Matrix generation and multiplication
- Selection and filtering
- Joining
- Graph algorithms

Intermediate Data

- Written locally
  - Transferred from mappers to reducers over network
- Issue
  - Performance bottleneck
- Solution
  - Reduce data
  - Use combiners
  - Use In-Mapper Combining

In-Mapper Combining (I)

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)   → COME RIDURRE EMIT?

class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count sum)
```

In-Mapper Combining (II)

QUALE E' IL PREZZO CHE STIAMO PAGANDO
PER QUESTA
SOLUZIONE?

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(term t, count H{t})
```

MEMORIA

SE E' TROPPA
NON VA BENE

TRADE-OFF = MEMORY ~ SPEED

In-Mapper Combining (III)

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

HASH TABLES

METODO PESANTE DAL PUNTO DI VISTA DELLA MEMORIA

CHIAMATA ALLA FINE SOLO UNA VOLTA

In-Mapper Combining

- **Advantages**:
  - **Complete** local aggregation **control** (how and when)
  - **Guaranteed** to execute
  - Direct **efficiency control** on intermediate data creation
  - Avoid **unnecessary** objects creation and destruction (before combiners)
- **Disadvantages**:
  - **Breaks** the functional programming background (state)
  - Potential ordering-dependent **bugs**
  - **Memory** scalability **bottleneck** (solved by memory **foot-printing** and flushing)

SI, OK

È SUFFICIENTE?

MEMORY PRESSURE MONITOR → OGNI VOLTA CHE AVVIO UN MAPPER CONTROLLO LO STATO DELLA MEMORIA

NO, FLUSH

Matrix Generation

- Common problem:
  - Given an input of size N, generate an output of size $N \times N$

LINEARE SQUARED

- Example: word co-occurrence matrix
  - Given a document collection, emit the bigram frequencies
- Two solutions
  - **Pairs**: generating $O(N^2)$ data in $O(1)$ space
  - **Stripes**: generation $O(N)$ data in $O(N)$ space

TRADE-OFF TRA SPAZIO SU DISCO ~ BANDWIDTH

Pairs

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair (w, u), count 1)          ▷ Emit count for each co-occurrence
```

STIAMO FACENDO LA SOMMA DI UNA GRANDE QUANTITA' DI DATI.

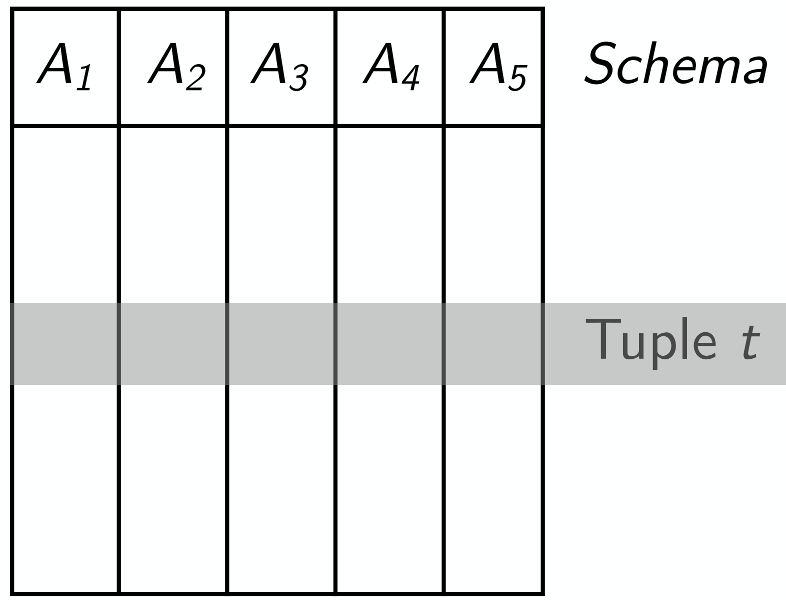
```
class Reducer
    method Reduce(pair p, counts [c1, c2, ...])
        s ← 0
        for all count c ∈ counts [c1, c2, ...] do
            s ← s + c                               ▷ Sum co-occurrence counts
        Emit(pair p, count s)
```

Stripes

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
            Emit(Term w, Stripe H)

class Reducer
    method Reduce(term w, stripes [H1, H2, H3, . . .])
        Hf ← new AssociativeArray
        for all stripe H ∈ stripes [H1, H2, H3, . . .] do
            Sum(Hf, H)                         ▷ Element-wise sum
        Emit(term w, stripe Hf)
```

Relational Algebra Operators



Relation R

- SELECTION: Select from relation R tuples satisfying condition $c(t)$
- PROJECTION: For each tuple in relation R, select only certain attributes A_i
- UNION, INTERSECTION, DIFFERENCE: Set operations on two relations with same schema
- NATURAL JOIN
- GROUPING and AGGREGATION

Selection and projection

Selection and projection

- **Map**: each tuple t in R, if condition $c(t)$ is satisfied, is outputted as a $(t, \perp)$ pair
- **Reduce**: for each $(t, \perp, \perp, \perp, ...)$ pair in input, output $(t, \perp)$

Selection and projection

- **Map**: each tuple t in R, if condition $c(t)$ is satisfied, is outputted as a $(t, \perp)$ pair
- **Reduce**: for each $(t, \perp, \perp, \perp, ...)$ pair in input, output $(t, \perp)$

- **Map**: each tuple t in R, create a new tuple t' containing only the projected attributes and output a $(t', \perp)$ pair
- **Reduce**: for each $(t', \perp, \perp, \perp, ...)$ pair in input, output $(t', \perp)$

Union, intersection and difference

Union, intersection and difference

- **Map**: for each tuple t in R, output a $(t, \perp)$ pair
- **Reduce**: for each input key t, there will be 1 or 2 values equal to $\perp$. Coalesce them in a single output $(t, \perp)$

Union, intersection and difference

- **Map**: for each tuple t in R, output a *(t,* $\perp$*)* pair
- **Reduce**: for each input key t, there will be 1 or 2 values equal to $\perp$. Coalesce them in a single output *(t,* $\perp$*)*

- **Map**: for each tuple t in R, output a *(t,* $\perp$*)* pair
- **Reduce**: for each input key t, there will be 1 or 2 values equal to $\perp$. If there are 2 value, coalesce them in a single output *(t,* $\perp$*)* otherwise do nothing

Union, intersection and difference

- **Map**: for each tuple t in R, output a $(t, \perp)$ pair
- **Reduce**: for each input key t, there will be 1 or 2 values equal to $\perp$. Coalesce them in a single output $(t, \perp)$

- **Map**: for each tuple t in R, output a $(t, \perp)$ pair
- **Reduce**: for each input key t, there will be 1 or 2 values equal to $\perp$. If there are 2 value, coalesce them in a single output $(t, \perp)$ otherwise do nothing

- **Map**: for each tuple t in R, output (t , `R`) and for each tuple t in S, output (t , `S`)
- **Reduce**: for each input key t, there will be 1 or 2 values. If there is 1 value equal to (t , `R`) output $(t, \perp)$, otherwise do nothing

Natural Join

For simplicity, assume we have two relations *R(A,B)* and *S(B,C)*. Find tuples that agree on the *B* attribute values and output them.

Natural Join

For simplicity, assume we have two relations *R(A,B)* and *S(B,C)*. Find tuples that agree on the *B* attribute values and output them.

- **Map**: for each tuple *(a, b)* from *R*, output *(b, (*`R`*, a))* and for each tuple *(b, c)* from *S*, produce *(b, (*`S`*, c))*
- **Reduce**: For each input key *b*, there will a list of values of the form *(*`R`*, a)* or *(*`S`*, c)*. Construct all pairs and output them together with *b*

Grouping and aggregation

For simplicity, assume we have the relation *R(A,B,C)* and we want to group-by *A* and aggregate on *B*, disregarding C.

Grouping and aggregation

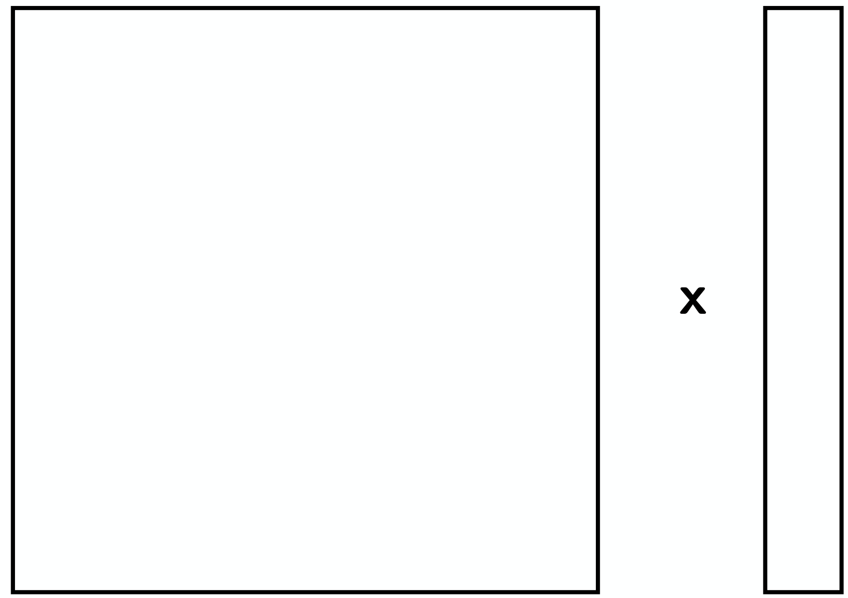
For simplicity, assume we have the relation *R(A,B,C)* and we want to group-by *A* and aggregate on *B*, disregarding C.

- **Map**: for each tuple *(a, b, c)* from *R*, output *(a, b)*. Each key a represents a group.
- **Reduce**: apply the aggregation operator to the list of *b* values associated with group keyed by *a*, producing *x*. Then output *(a, x)*.

Stage Chaining

- As map reduce calculations get **more complex**, it's useful to break them down into **stages**, with the output of one stage serving as input to the next PIPELINE
- Intermediate output may be useful for **different outputs** too, so you can get some reuse
- The intermediate records can be saved in the data store, forming a **materialized view**
- **Early stages** of map reduce operations often represent the **heaviest amount** of data access, so building and save them once as a basis for many downstream uses saves a lot of work

Matrix Vector Multiplication



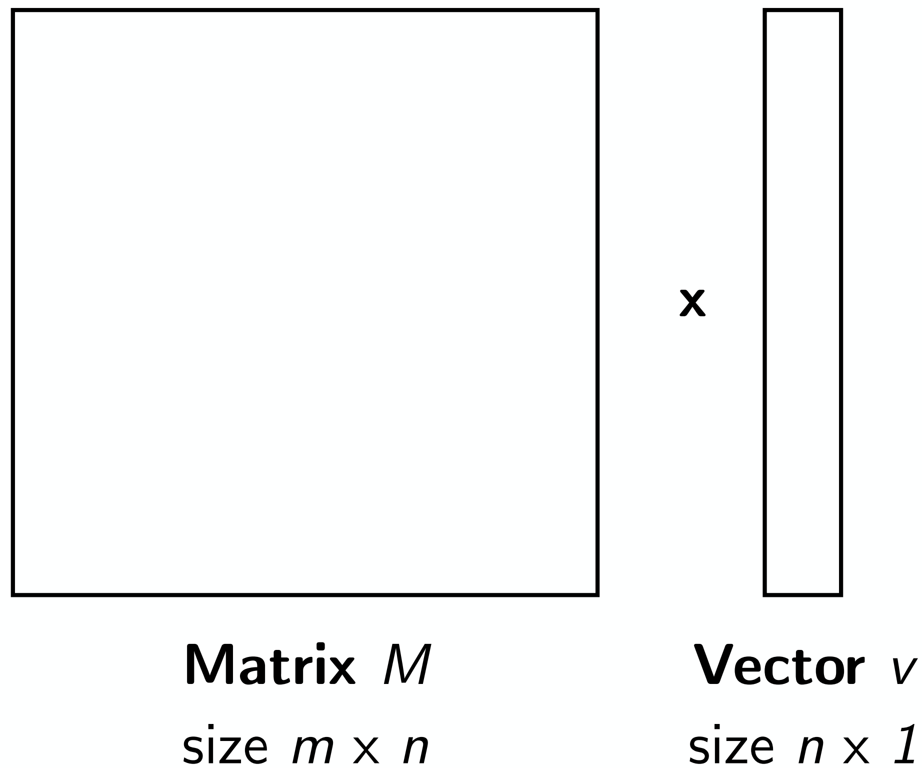
Matrix *M*
size *m* x *n*

Vector *v*
size *n* x *1*

The matrix does not fit in memory, and

1. The vector *v* **does fit** in a machine's memory
2. The vector *v* **does not fit** in machine's memor

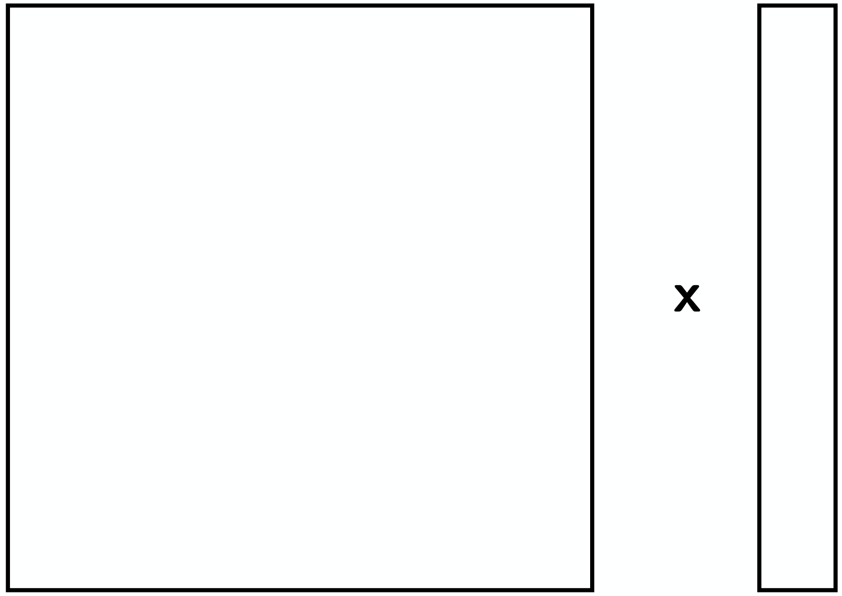
Vector does fit



The matrix is stored in HDFS as a list of (i, j, m_{ij}) tuples

- The elements v_j of v are **available to all mappers**
- **Map**: $((\ i, j\ ), m_{ij})$ pair → $(i, m_{ij}v_j)$ pair
- **Reduce**: $(i, [m_{i1}v_1, m_{i2}v_2, \ldots, m_{in}v_n])$ pair → $(i, m_{i1}v_1 + m_{i2}v_2 + \ldots + m_{in}v_n)$ pair

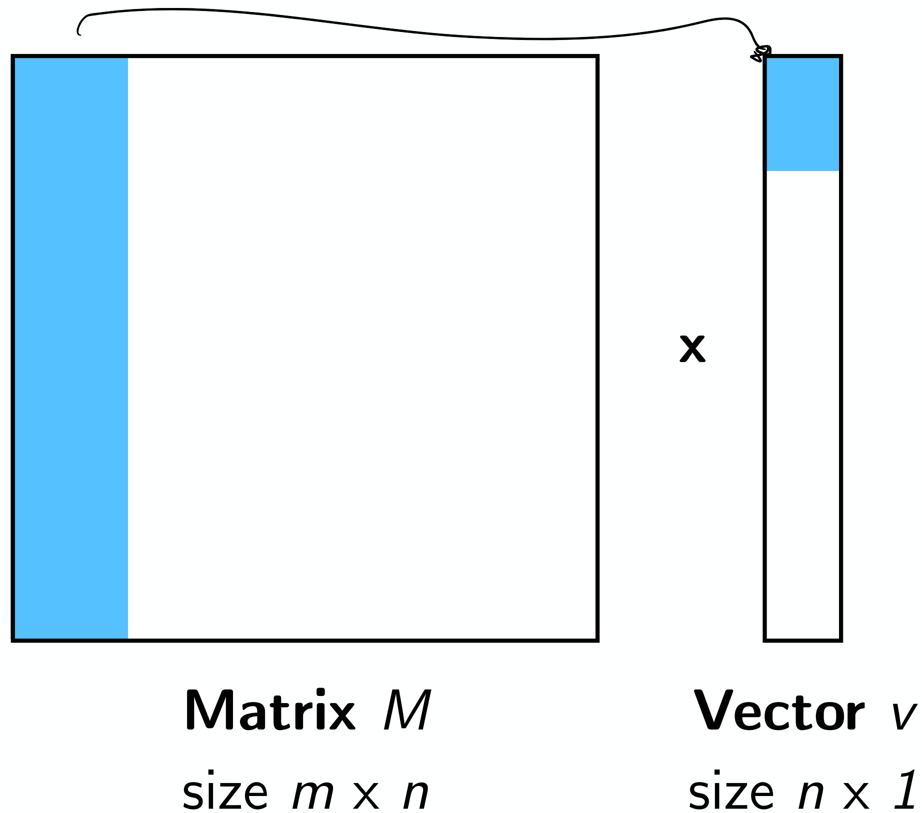
Vector does not fit



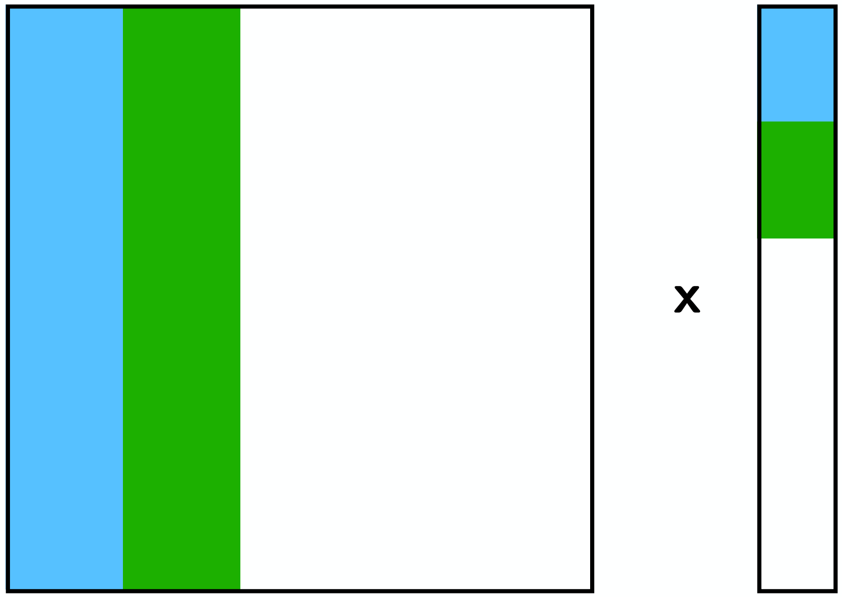
Matrix *M*
size *m* x *n*

Vector *v*
size *n* x *1*

Vector does not fit



Vector does not fit



Matrix M

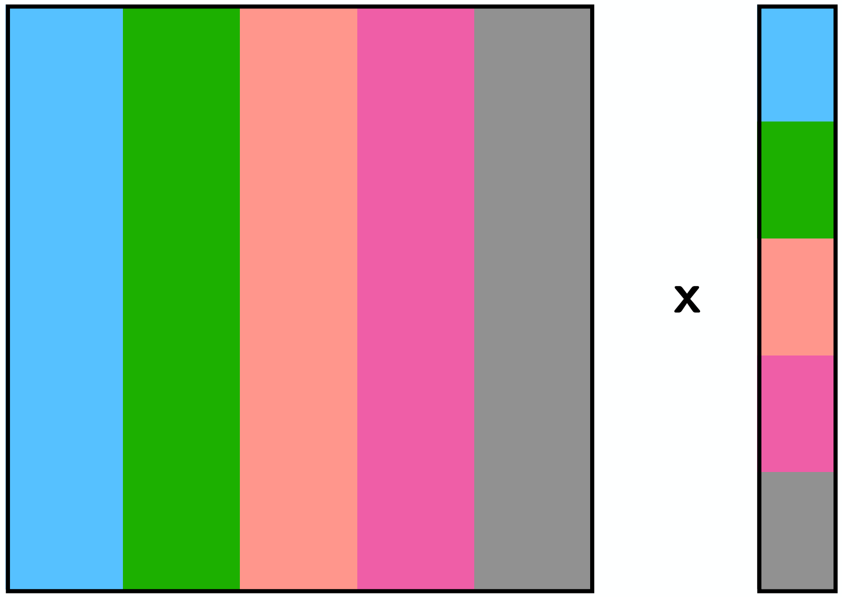
size m x n

Vector v

size n x 1

Vector does not fit

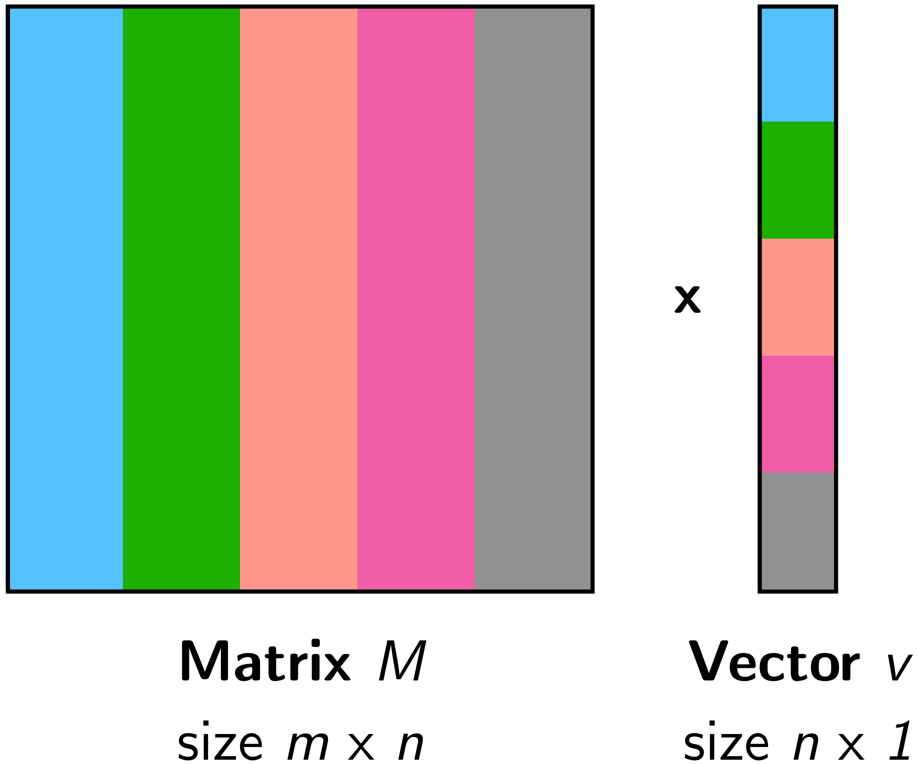
QUESTE STRISCE
FITTANO IN
MEMORIA



Matrix *M*
size *m* x *n*

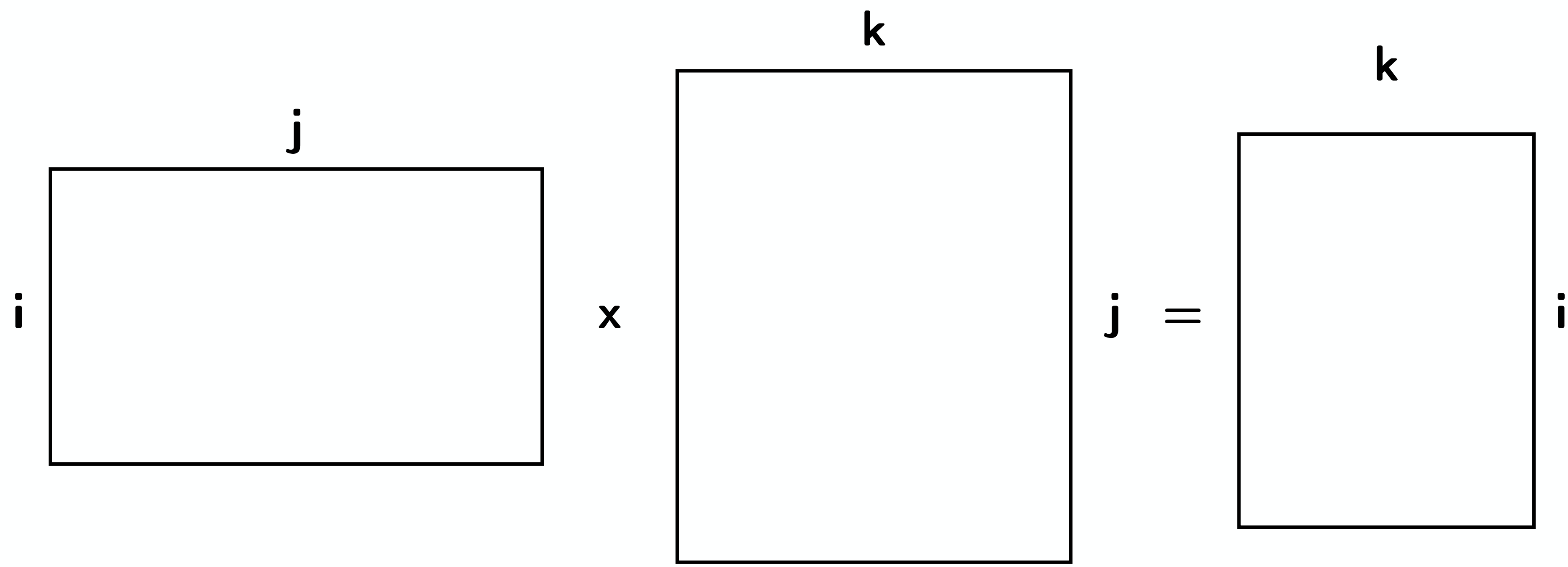
Vector *v*
size *n* x *1*

Vector does not fit



- Divide the vector in equal-sized **subvectors** that can fit in memory
- According to that, divide the matrix in **stripes**
- Stripe i and subvector i are **independent** from other stripes/ subvectors
- Use the **previous algorithm** for each stripe/subvector pair

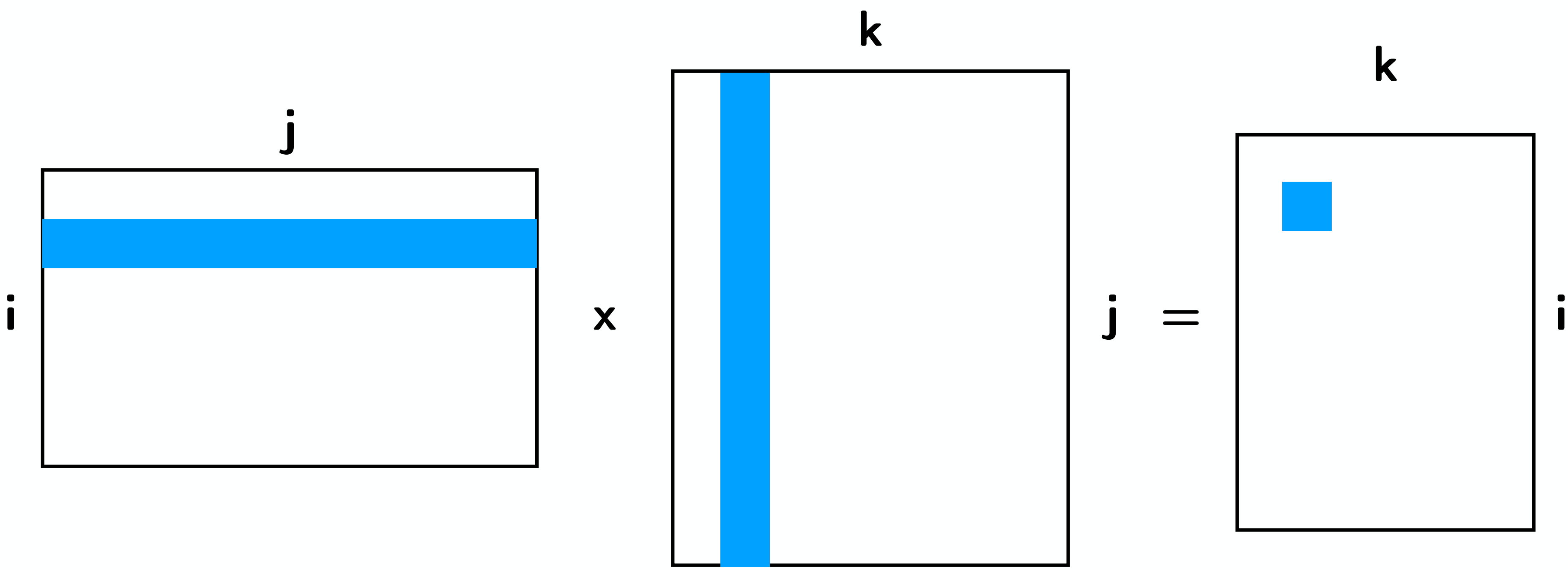
Matrix Matrix Multiplication (I)



Matrix M **Matrix** N **Matrix** P

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

Matrix Matrix Multiplication (I)



Matrix M **Matrix** N **Matrix** P

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

Matrix Multiplication (II)

- A matrix can be seen as a 3 attributes relation:
  - (row index, column index, value) tuples
  - $M \rightarrow (i, j, m_{ij})$, $N \rightarrow (j, k, n_{jk})$
- As large matrices are often sparse (0's) we omit such tuples
- The product MN can be seen as a natural join over attribute j, followed by product computation, followed by grouping and aggregation
  - Start with (i, j, v) and (j, k, w) MAP
  - Compute (i, j, k, v, w) JOIN
  - Compute $(i, j, k, v \times w)$ GROUPING
  - Compute $(i, k, \Sigma_j\, v \times w)$

Matrix Multiplication (III)

- First stage
  - **Map**: given (i, j, m_{ij}) produce $(j, (\texttt{M}, i, m_{ij}))$

    given (j, k, n_{jk}) produce $(j, (\texttt{N}, k, n_{jk}))$
  - **Reduce**: given $(j, [(\texttt{M}, i, m_{ij}), (\texttt{N}, k, n_{jk})])$ produce $((i, k), m_{ij} \times n_{jk})$

    otherwise do nothing
- Second stage
  - **Map**: identity
  - **Reduce**: produce the sum of the list of values associated with the key

Matrix Matrix Multiplication (IV)

Algorithm 1: The Map Function

for *each element* m_{ij} *of* M **do**
  produce $(key, value)$ pairs as $((i,k),(M,j,m_{ij}))$, for $k = 1,2,3,..$ up to the number of columns of N
for *each element* n_{jk} *of* N **do**
  produce $(key, value)$ pairs as $((i,k),(N,j,n_{jk}))$, for $i = 1,2,3,...$ up to the number of rows of M
return *Set of (key, value) pairs that each key, (i,k), has a list with values (M,j,m_{ij}) and (N,j,n_{jk}) for all possible values of j*

Algorithm 2: The Reduce Function

for *each key (i,k)* **do**
  sort values begin with M by j in $list_M$
  sort values begin with N by j in $list_N$
  multiply m_{ij} and n_{jk} for j_{th} value of each list
  sum up $m_{ij} * n_{jk}$
return $(i,k), \sum_{j=1} m_{ij} * n_{jk}$

Matrix Matrix Multiplication (III)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \longrightarrow \begin{bmatrix} 1a + 2c + 3e & 1b + 2d + 3f \\ 4a + 5c + 6e & 4b + 5d + 6f \end{bmatrix}$$

Matrix Matrix Multiplication (III)

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}a & b\\c & d\\e & f\end{bmatrix} \longrightarrow \begin{bmatrix}1a+2c+3e & 1b+2d+3f\\4a+5c+6e & 4b+5d+6f\end{bmatrix}$$

$(i,k),(M,j,m_{ij})$

$m_{11}=1$
$(1,1),(M,1,1) k=1$
$(1,2),(M,1,1) k=2$

$m_{12}=2$
$(1,1),(M,2,2) k=1$
$(1,2),(M,2,2) k=2$
....

$m_{23}=6$
$(2,1),(M,3,6) k=1$
$(2,2),(M,3,6) k=2$

$(i,k),(N,j,n_{jk})$

$n_{11}=a$
$(1,1),(N,1,a) i=1$
$(2,1),(N,1,a) i=2$

$n_{21}=c$
$(1,1),(N,2,c) i=1$
$(2,1),(N,2,c) i=2$

$n_{31}=e$
$(1,1),(N,3,e) i=1$
$(2,1),(N,3,e) i=2$
....

$n_{32}=f$
$(1,2),(N,3,f) i=1$
$(2,2),(N,3,f) i=2$

Matrix Matrix Multiplication (III)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \longrightarrow \begin{bmatrix} 1a+2c+3e & 1b+2d+3f \\ 4a+5c+6e & 4b+5d+6f \end{bmatrix}$$

$$\begin{aligned}
&((i,k), [(M, j, m_{ij}), (M, j, m_{ij}), ..., (N, j, n_{jk}), (N, j, n_{jk}), ...])\\
&(1,1), [(M,1,1), (M,2,2), (M,3,3), (N,1,a, (N,2,c), (N,3,e)]\\
&(1,2), [(M,1,1), (M,2,2), (M,3,3), (N,1,b, (N,2,d), (N,3,f)]\\
&(2,1), [(M,1,4), (M,2,5), (M,3,6), (N,1,a, (N,2,c), (N,3,e)]\\
&(2,2), [(M,1,4), (M,2,5), (M,3,6), (N,1,b, (N,2,d), (N,3,f)]
\end{aligned}$$

Matrix Matrix Multiplication (III)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \longrightarrow \begin{bmatrix} 1a+2c+3e & 1b+2d+3f \\ 4a+5c+6e & 4b+5d+6f \end{bmatrix}$$

$$[(M,1,1),(M,2,2),(M,3,3),(N,1,a,(N,2,c),(N,3,e)]$$

$$list_M = [(M,1,1),(M,2,2),(M,3,3)]$$
$$list_N = [(N,1,a),(N,2,c),(N,3,e)]$$

$$P(1,1) = 1a+2c+3e$$

$$P(1,1) = 1a+2c+3e$$
$$P(1,2) = 1b+2d+3f$$
$$P(2,1) = 4a+5c+6e$$
$$P(2,2) = 4b+5d+6f$$

Graphs

- G = (V,E), where
  - V represents the set of **vertices** (nodes)
  - E represents the set of **edges** (links)
  - Both vertices and edges may contain **additional information**
- Graph algorithms typically involve:
  - **Performing computations at each node**: based on node features, edge features, and local link structure
  - **Propagating computations**: "traversing" the graph
- Key questions:
  - How do you **represent graph data** in MapReduce?
  - How do you **traverse a graph** in MapReduce?

Representing Graphs (I)

- **Adjacency matrix**
  - Represent a graph as an n x n square matrix M
  - $n = |V|$
  - $m_{ij} = 1$ means a link from node i to j
- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

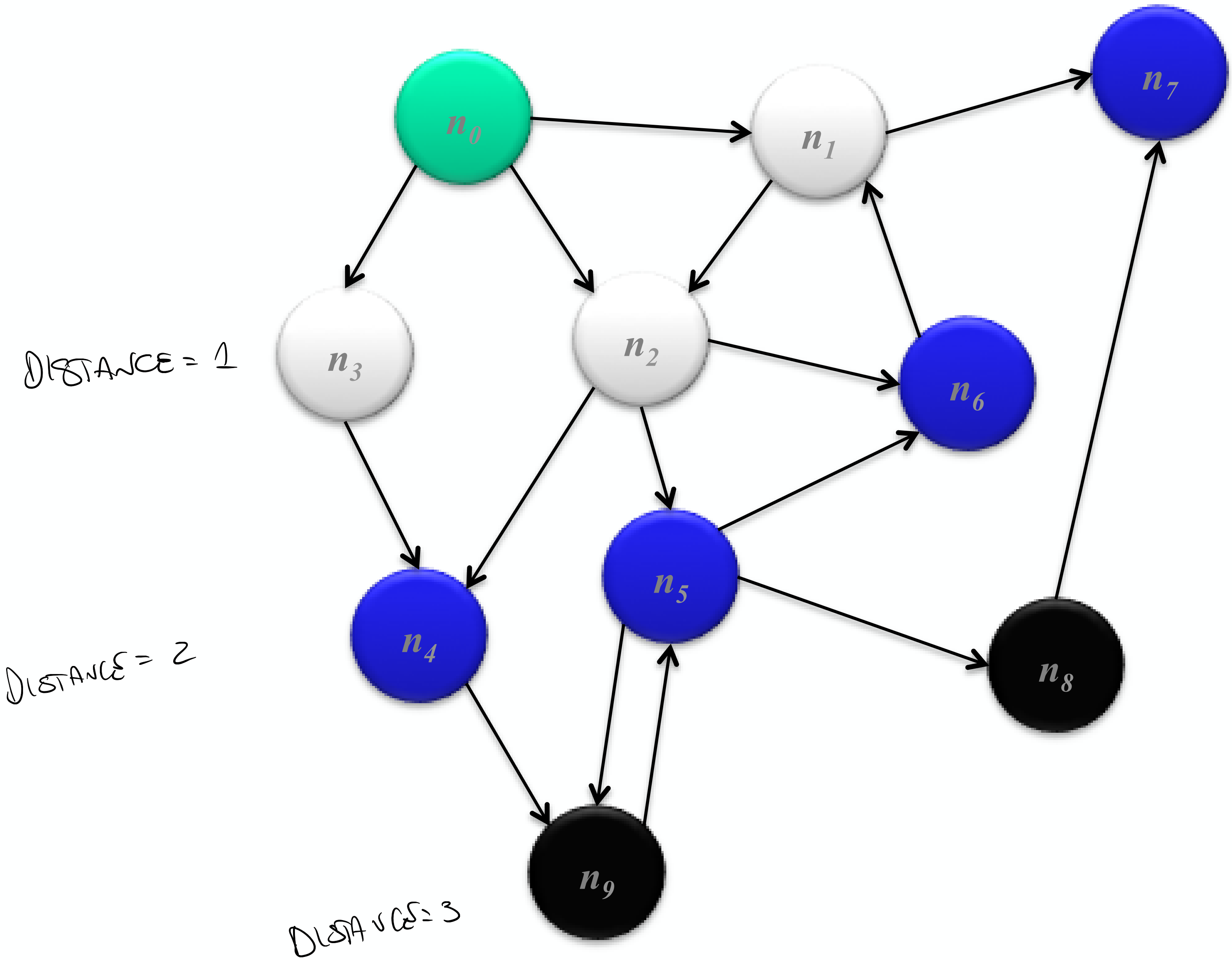
Representing Graphs (II)

- **Adjacency list**
  - Take adjacency matrices…
  - and throw away all the zeros
- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

Shortest Path Algorithm

- Consider simple case of equal edge weights
- Solution to the problem can be defined inductively
- Here's the intuition:
  - Define: b is reachable from a if b is on adjacency list of a
    DISTANCETO(s) = 0
  - For all nodes p reachable from s,
    DISTANCETO(p) = 1
  - For all nodes n reachable from some other set of nodes M,
    DISTANCETO(n) = 1 + min(DISTANCETO(m), m M)

Shortest Path



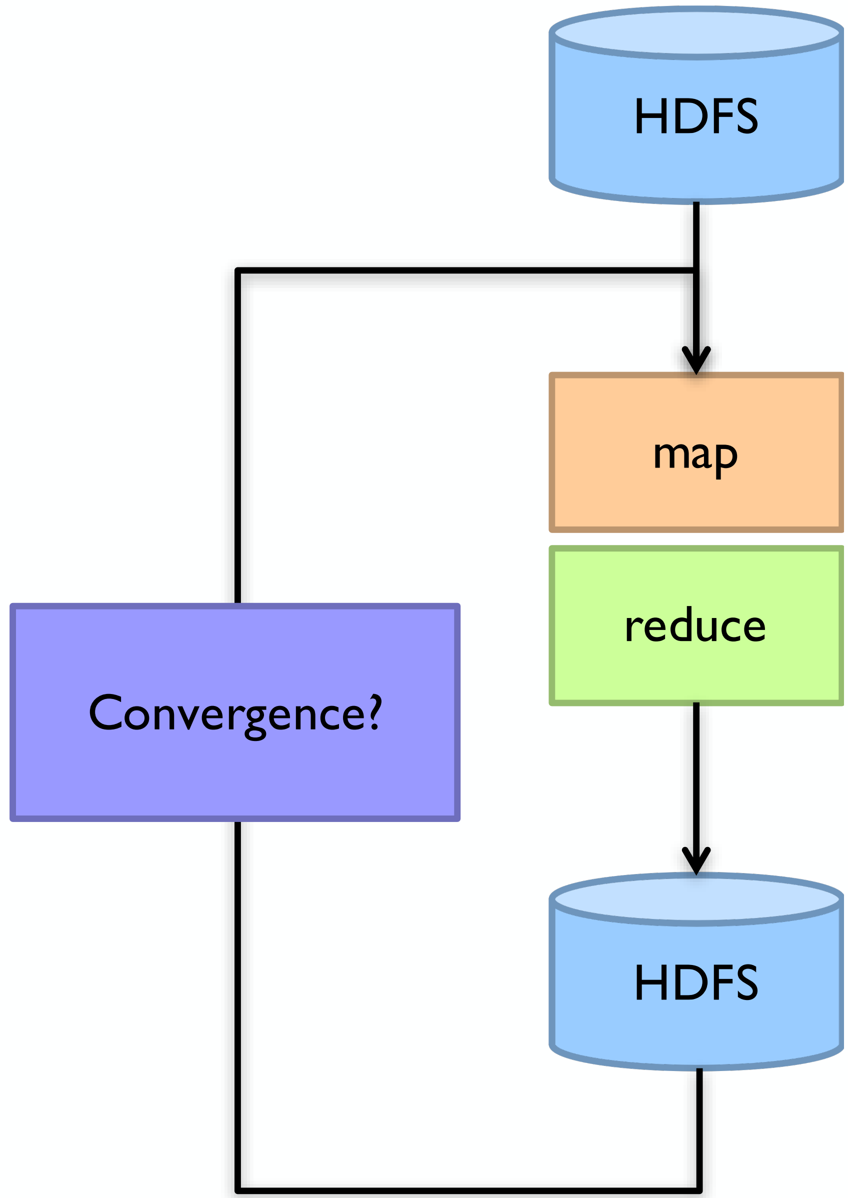
Shortest Path Algorithm

- Data representation:
  - key: node id n
  - value: d (distance of node n from start), adjacency list (list of nodes reachable from n)
  - Initialization: for all nodes except for start node, d = infinity
- Mapper:
  - Receives (node id, (d, adjacency list)) pair
  - For each node m in the adjacency list: emit (m, $1 + d$)
  - Additional bookkeeping needed to keep track of actual path
- Sort/Shuffle
  - Groups distances by node id
- Reducer:
  - Selects minimum distance among received nodes
  - Additional bookkeeping needed to keep track of actual path

Details (I)

- Each MapReduce iteration advances the "known frontier" by one hop
  - Subsequent iterations include more and more reachable nodes as frontier expands
  - Multiple iterations are needed to explore entire graph
- Preserving graph structure:
  - Problem: Where did the adjacency list go?
  - Solution: mapper emits (n, adjacency list) as well

Details (II)



Pseudocode

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)                                  ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)                          ▷ Emit distances to reachable nodes
```

```
class Reducer
    method Reduce(nid m, [d_1, d_2, . . .])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d_1, d_2, . . .] do
            if IsNode(d) then
                M ← d                                   ▷ Recover graph structure
            else if d < d_min then                      ▷ Look for shorter distance
                d_min ← d
        M.Distance ← d_min                              ▷ Update shortest distance
        Emit(nid m, node M)
```

Graph Algorithm Recipe

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: "traversing" the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass along partial results via outlinks, keyed by destination node
  - Perform aggregation in reducer on inlinks to a node
  - Iterate until convergence: controlled by external "driver"
  - Don't forget to pass the graph structure between iterations

Pagerank (I)

CALCOLA UNO "SCORE" PER OGNI PAGINA → UNA PAGINA E' IMPORTANTE SE HA TANTI LINK CHE PUNTANO A QUESTA PAGINA

- 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)

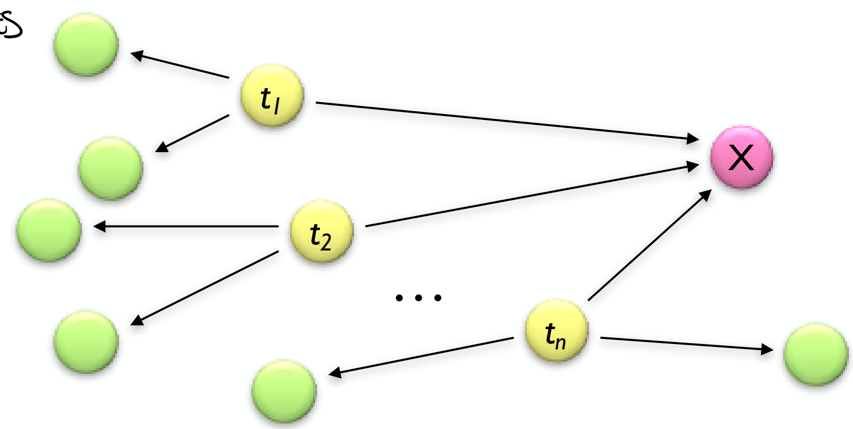
- Given page x with inlinks $t_1, \ldots, t_n$, where
  - $C(t)$ is the out-degree of link t
  - α is probability of random jump
  - N is the total number of nodes in the graph

RANDOM - JUMP ASSUMPTION

C'È UNA POSSIBILITÀ (DIVERSA DA 0) CHE UN SITO SIA SPAZZATURA

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

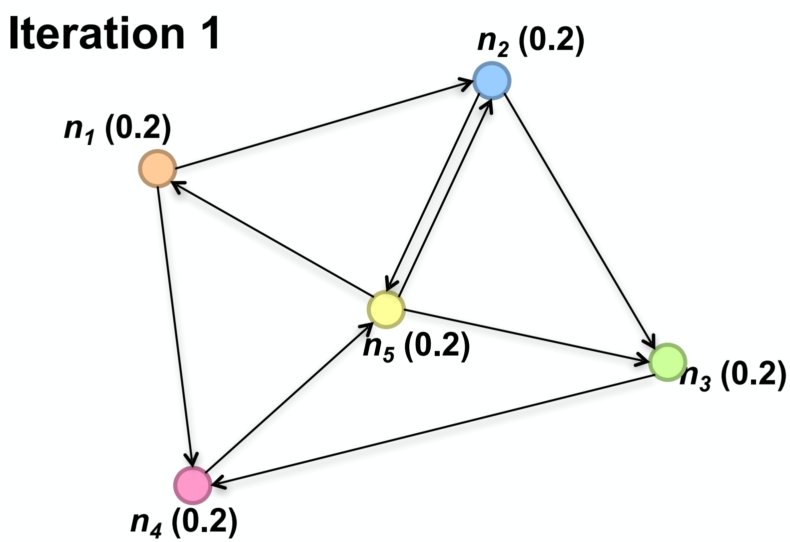
WEB PAGES



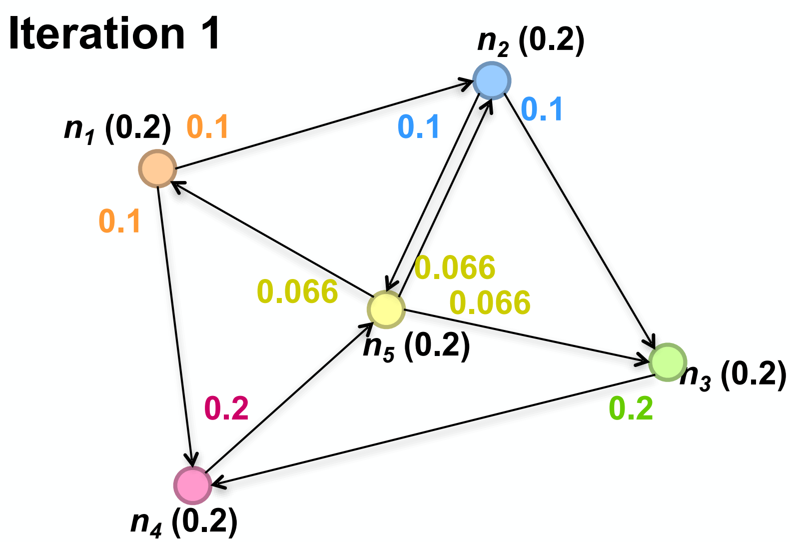
Pagerank Algorithm Sketch

- 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)

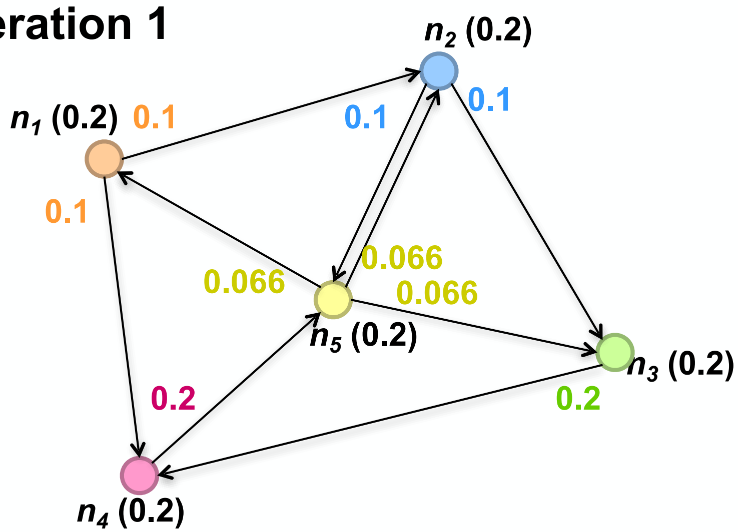


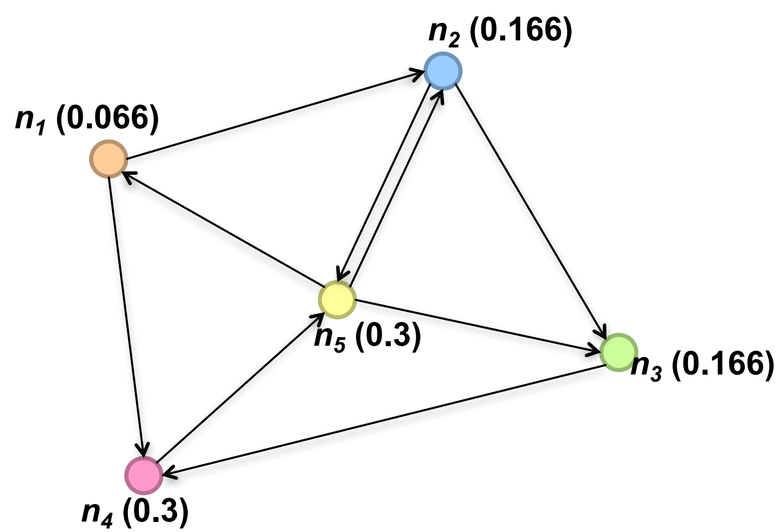
Simple Example (I)



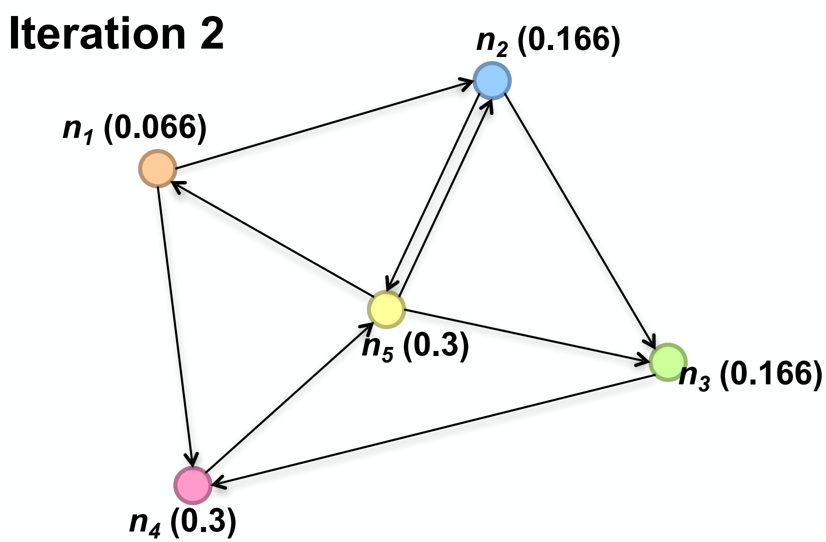
Simple Example (I)

Iteration 1

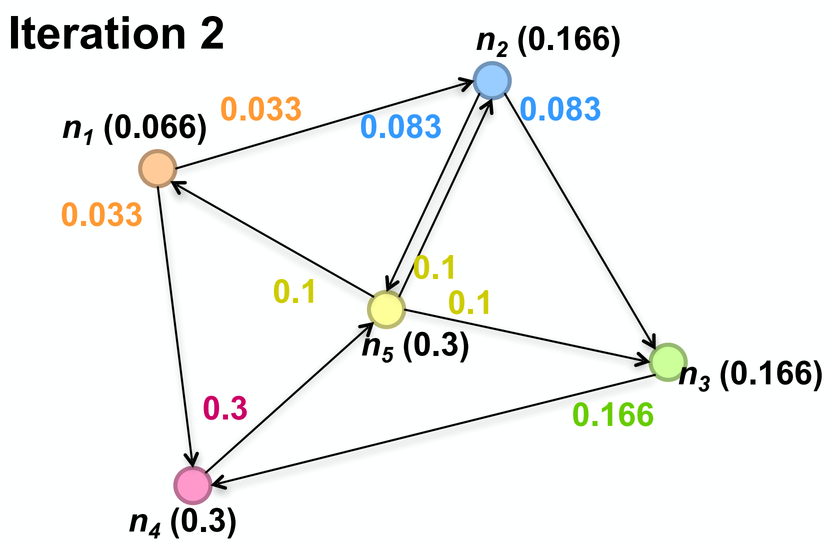




Simple Example (II)

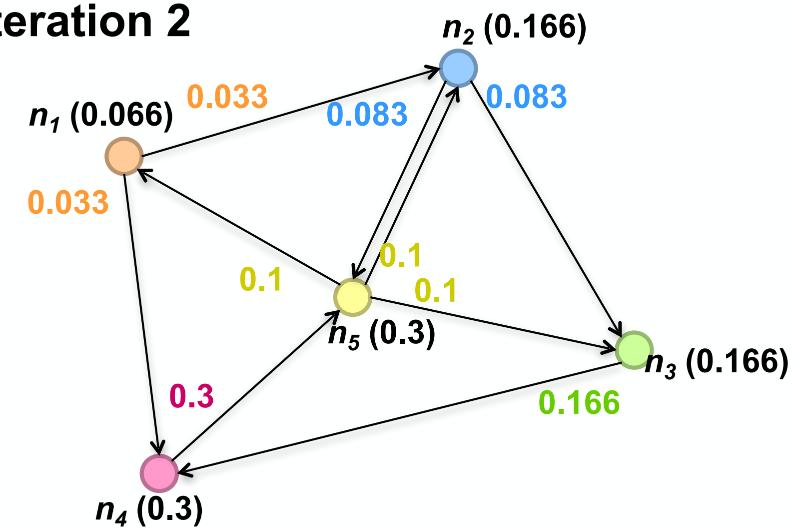


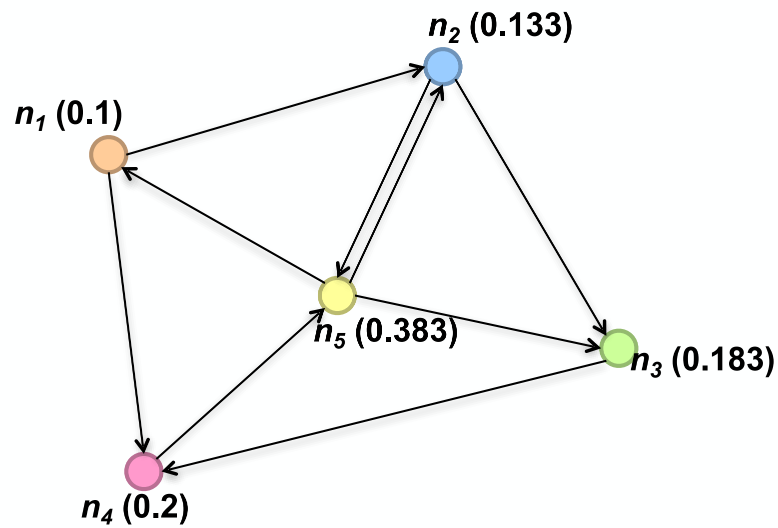
Simple Example (II)



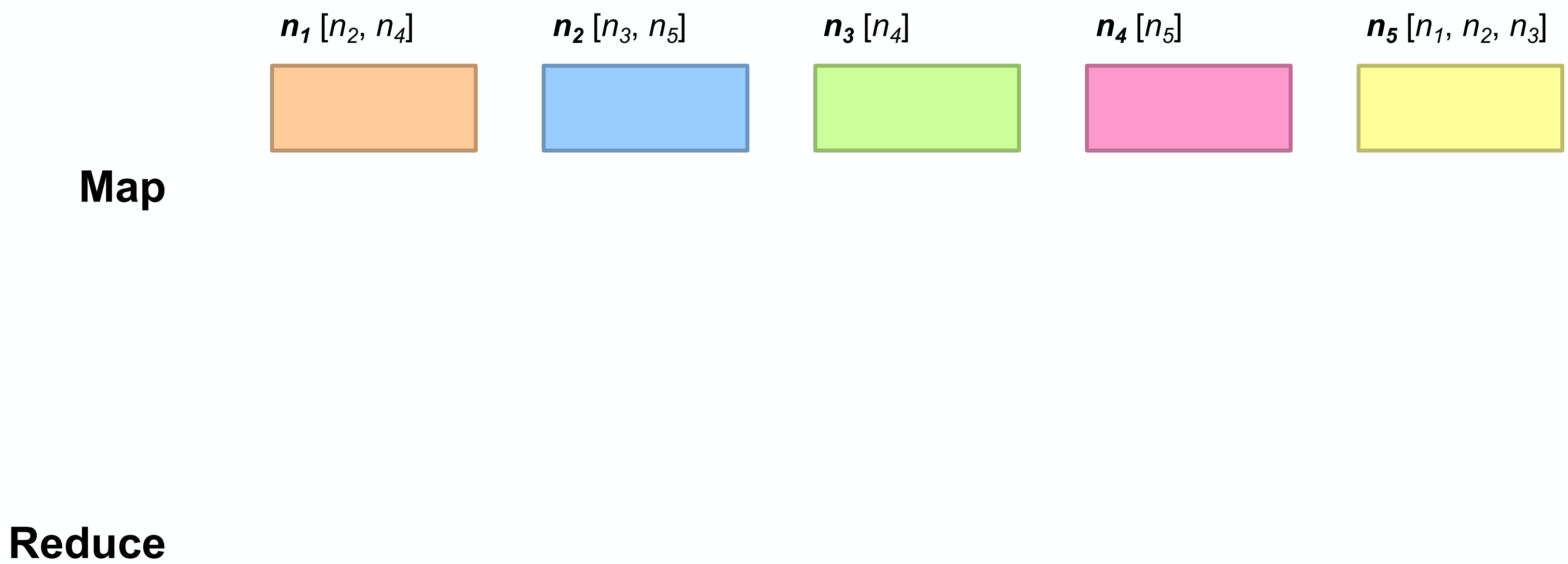
Simple Example (II)

Iteration 2

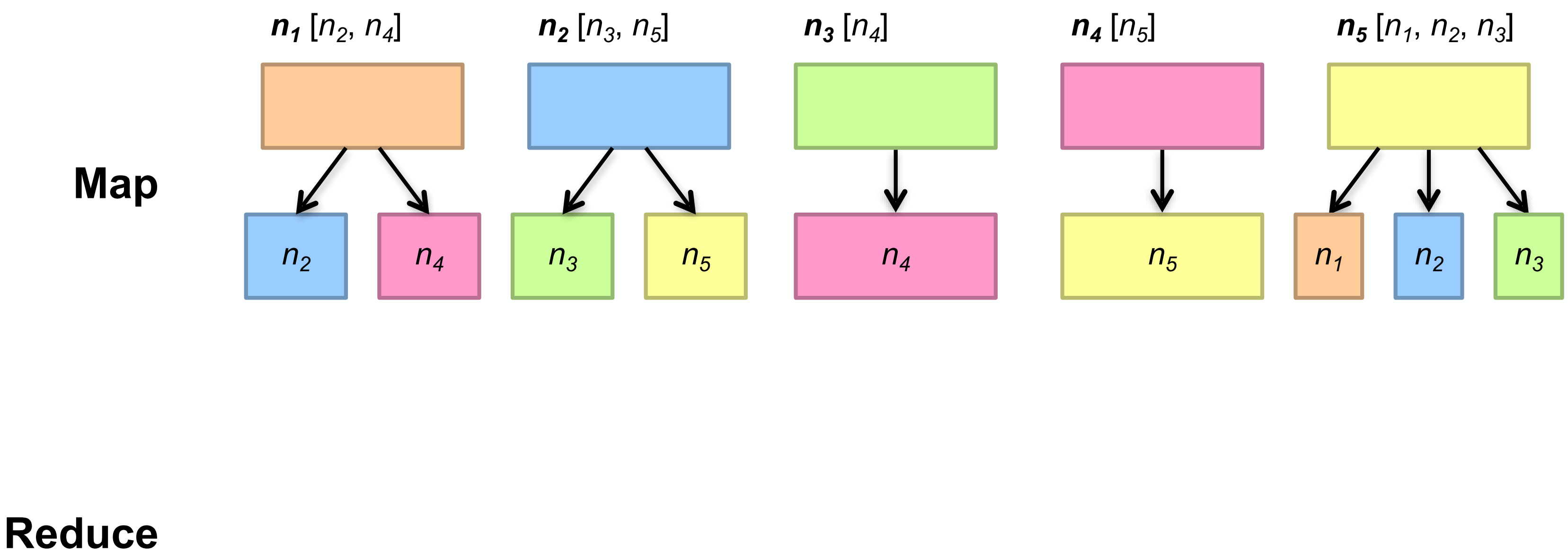




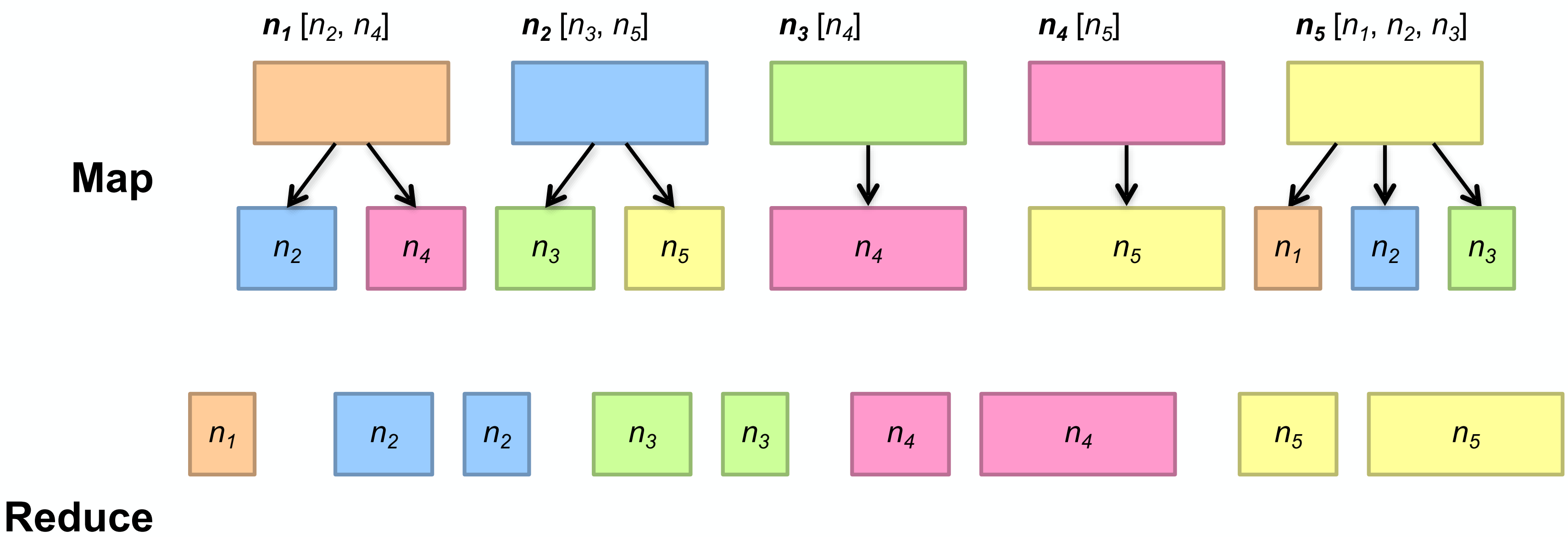
Pagerank In MapReduce (I)



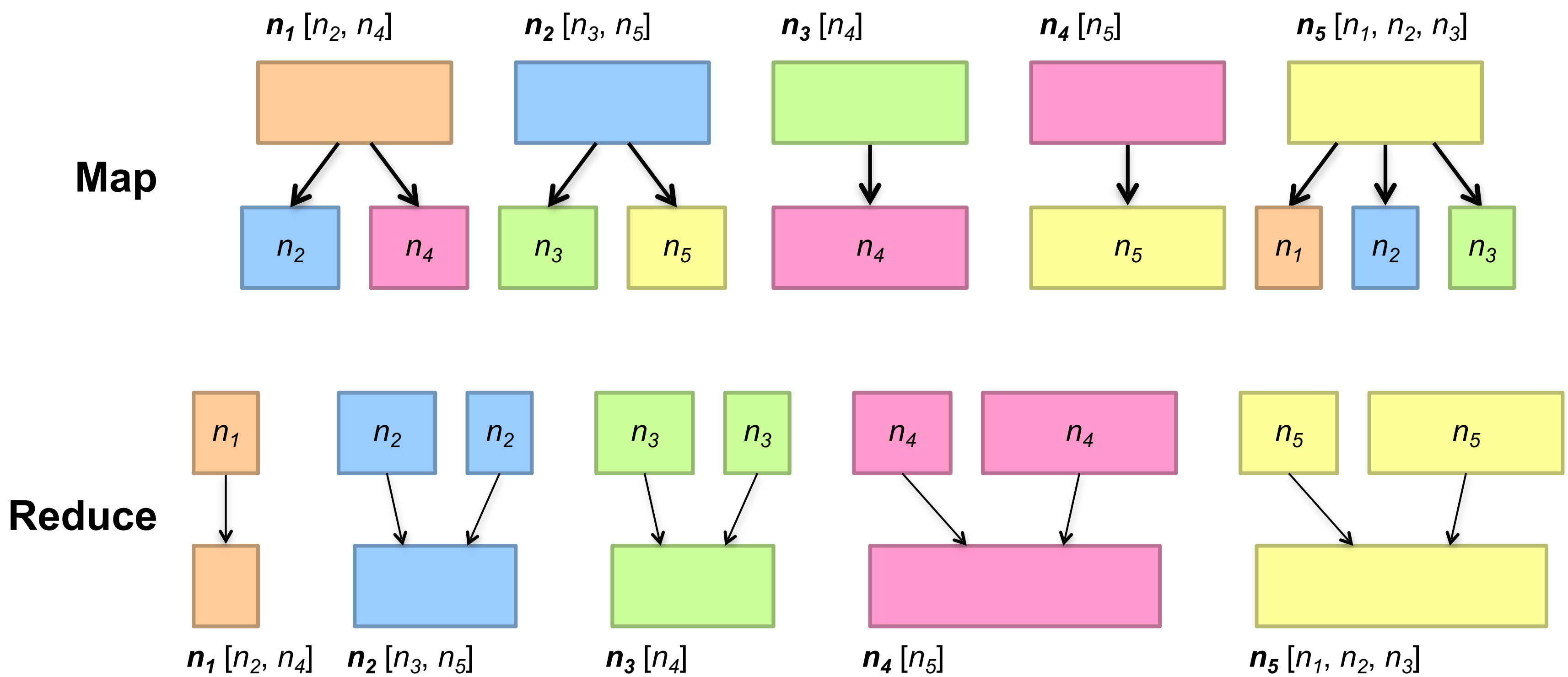
Pagerank In MapReduce (I)



Pagerank In MapReduce (I)



Pagerank In MapReduce (I)



Pagerank In MapReduce (II)

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

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                                   ▷ Recover graph structure
            else
                s ← s + p               ▷ Sums incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```