Map Reduce
Design Patterns

Caveat

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

  - 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)

```
1: class MAPPER
2:      method MAP(docid a, doc d)
3:           for all term t ∈ doc d do
4:                EMIT(term t, count 1)
```

```
1: class REDUCER
2:      method REDUCE(term t, counts [c_1, c_2, . . .])
3:           sum ← 0
4:           for all count c ∈ counts [c_1, c_2, . . .] do
5:                sum ← sum + c
6:           EMIT(term t, count sum)
```

In-Mapper Combining (II)

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         H ← new ASSOCIATIVEARRAY
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             EMIT(term t, count H{t})
```

In-Mapper Combining (III)

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

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)

Matrix Generation

- Common problem:

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

- 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

Pairs

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ NEIGHBORS(w) do
5:                 EMIT(pair (w, u), count 1)              ▷ Emit count for each co-occurrence
```

```
1: class REDUCER
2:     method REDUCE(pair p, counts [c₁, c₂, . . .])
3:         s ← 0
4:         for all count c ∈ counts [c₁, c₂, . . .] do
5:             s ← s + c                                   ▷ Sum co-occurrence counts
6:         EMIT(pair p, count s)
```

Stripes

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new ASSOCIATIVEARRAY
5:             for all term u ∈ NEIGHBORS(w) do
6:                 H{u} ← H{u} + 1                    ▷ Tally words co-occurring with w
7:             EMIT(Term w, Stripe H)
```

```
1: class REDUCER
2:     method REDUCE(term w, stripes [H_1, H_2, H_3, . . .])
3:         H_f ← new ASSOCIATIVEARRAY
4:         for all stripe H ∈ stripes [H_1, H_2, H_3, . . .] do
5:             SUM(H_f, H)                            ▷ Element-wise sum
6:         EMIT(term w, stripe H_f)
```

Relational Algebra Operators

A_1	A_2	A_3	A_4	A_5	*Schema*
					Tuple t
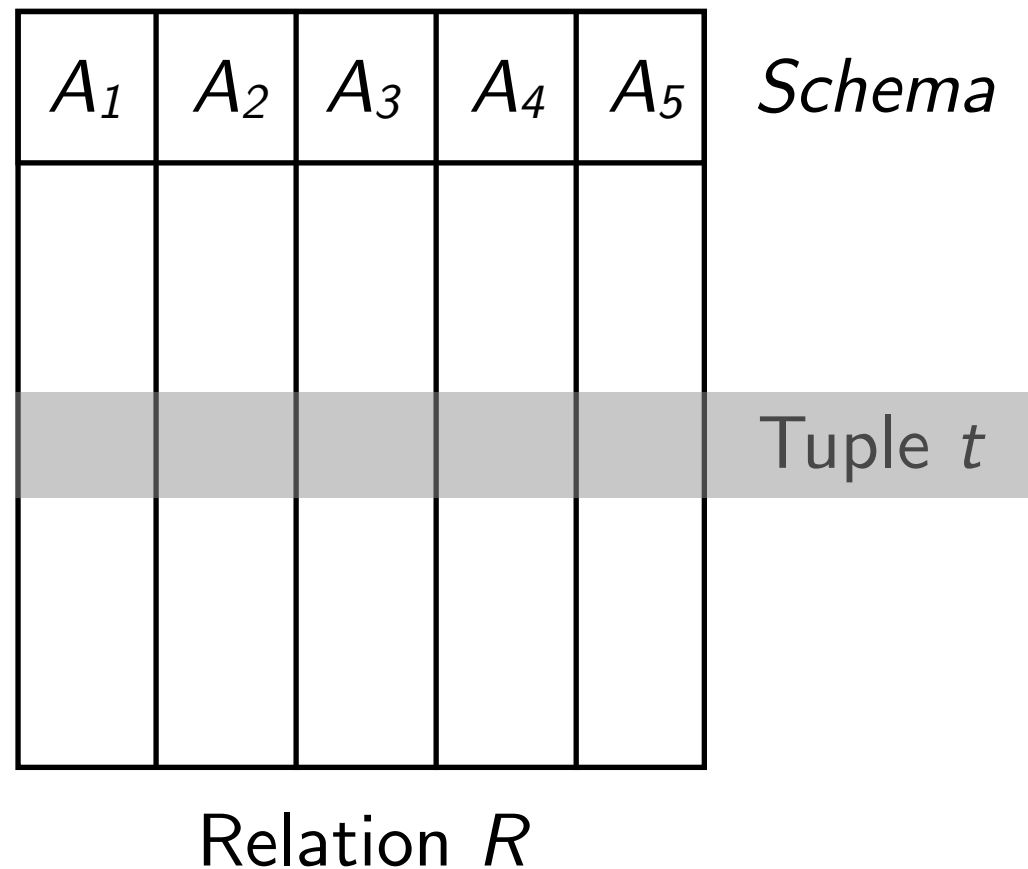
Relation R

- 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

- **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

- **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.

- **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.

- **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

- 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
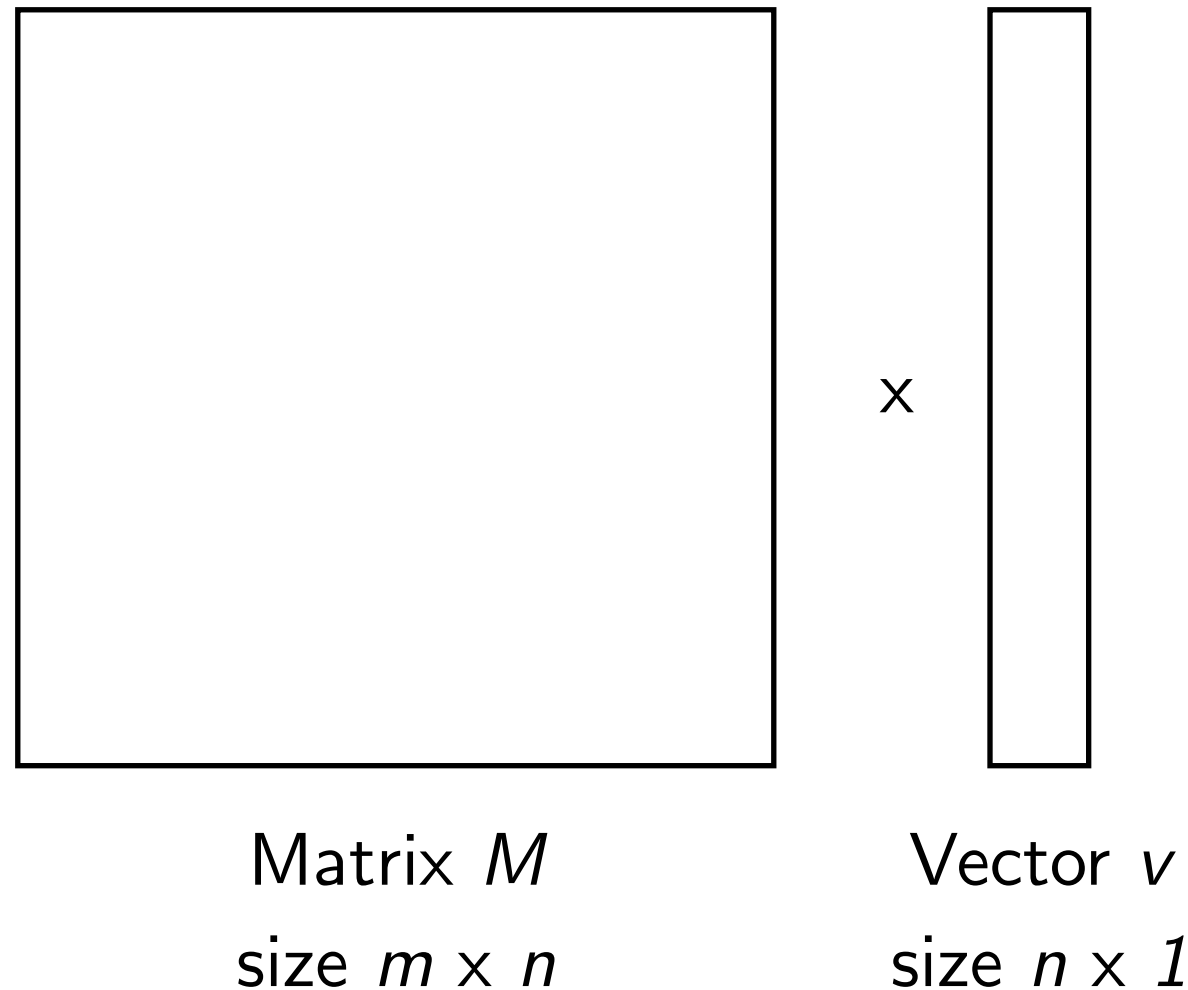
Matrix Vector Multiplication

Matrix M
size $m \times n$

x
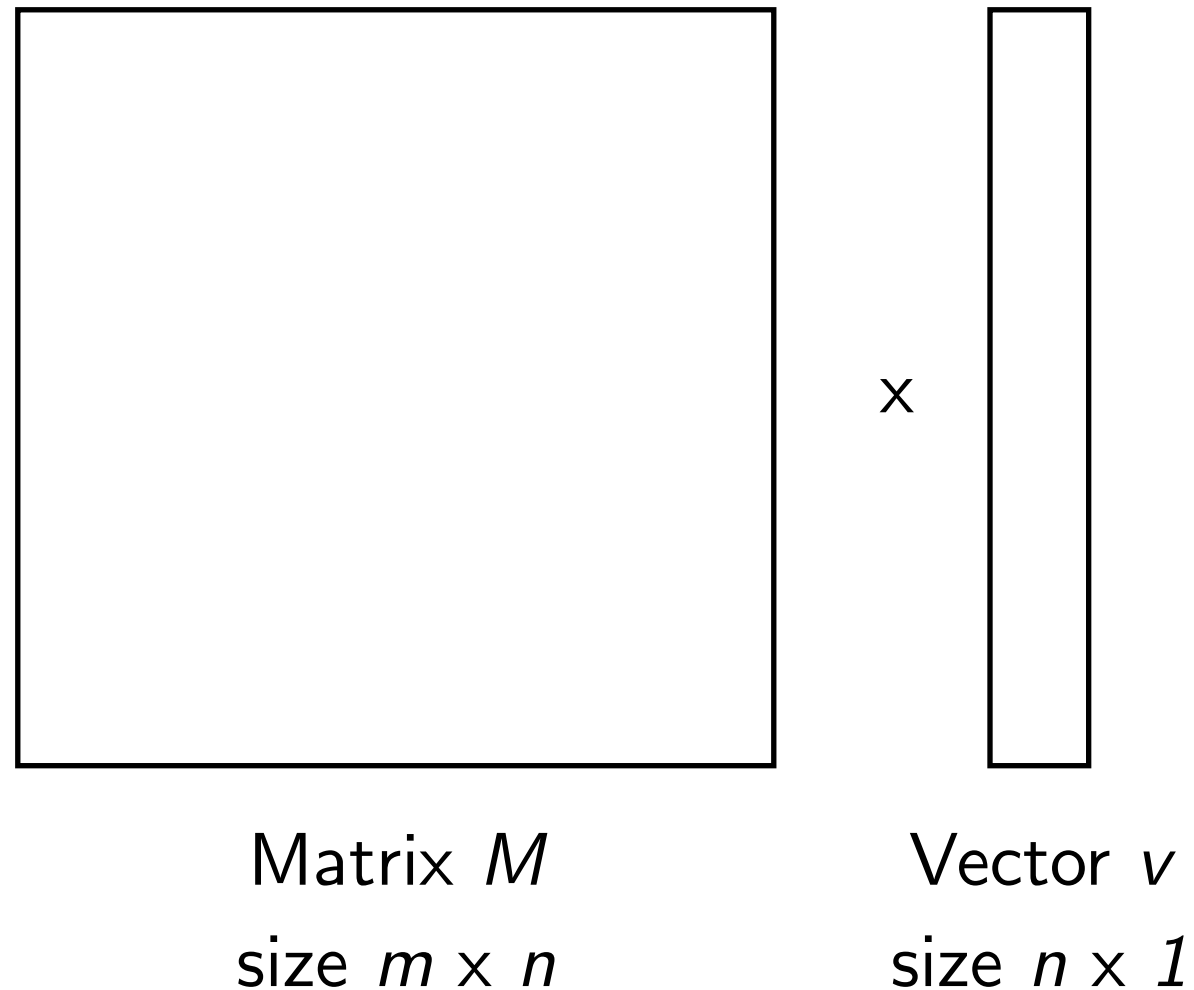
Vector v
size $n \times 1$

The matrix does not fit in memory, and

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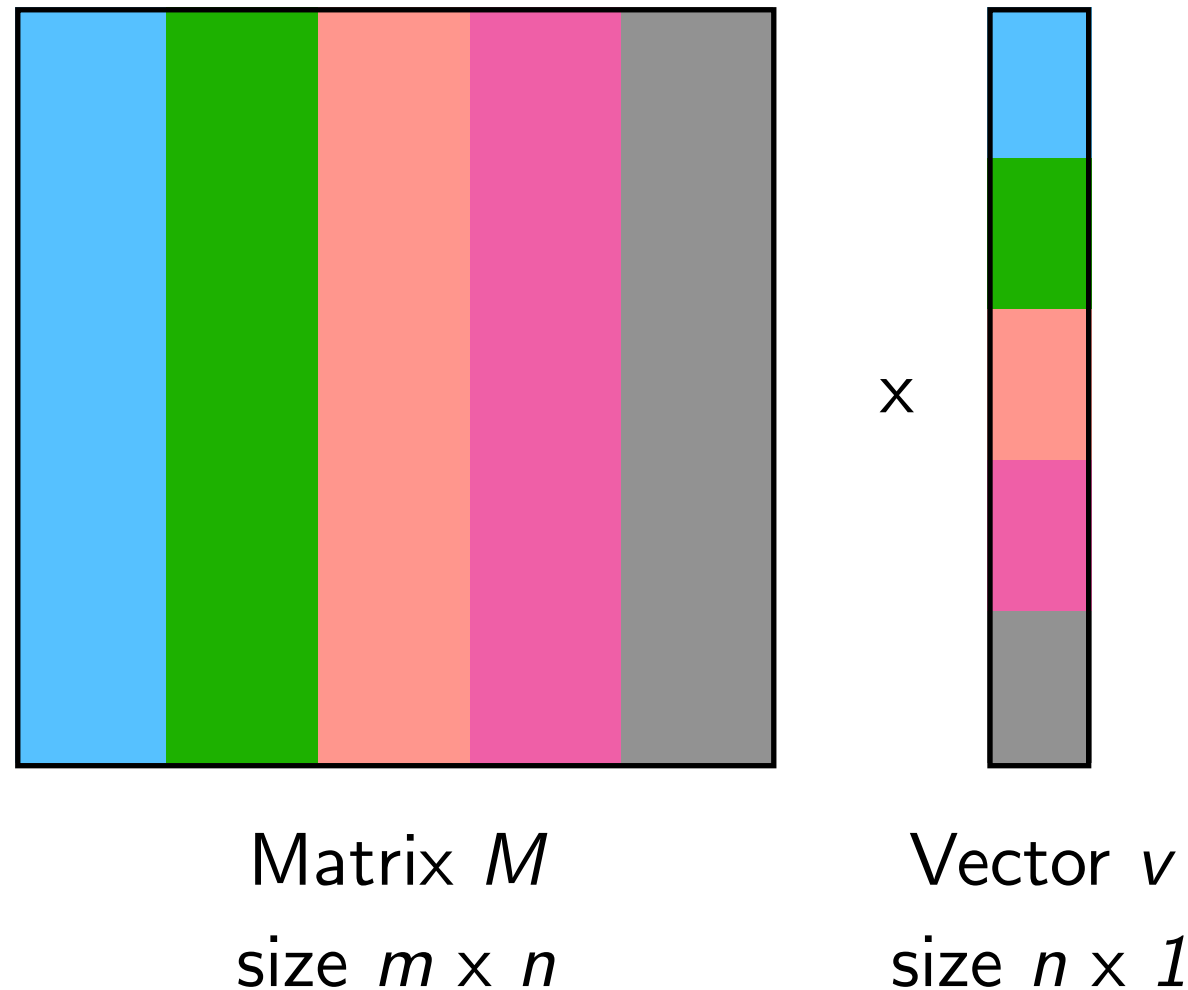
2. The vector v **does not fit** in machine's memor

Vector does fit

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

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

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 $\rightarrow (i, m_{ij}v_j)$ pair

- **Reduce**: $(i, [m_{i1}v_1, m_{i2}v_2, ..., m_{in}v_n])$ pair $\rightarrow (i, m_{i1}v_1 + m_{i2}v_2 + ... + m_{in}v_n)$ pair

Vector does not fit

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

x

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

- 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)
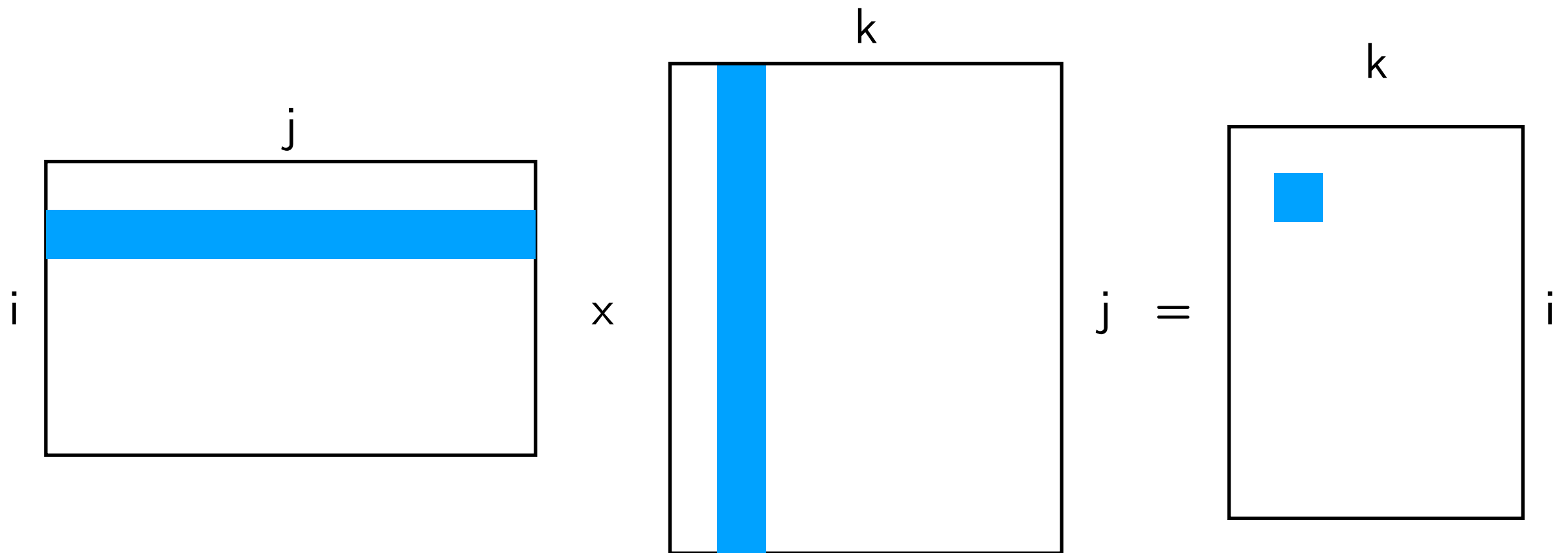
Matrix M x   Matrix N =   Matrix P

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

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)

  - Compute (i, j, k, v, w)

  - Compute $(i, j, k, v \times w)$

  - Compute $(i, k, \Sigma_j v \times w)$

Matrix Multiplication (III)

- First stage

  - **Map**: given *(i, j, m_{ij})* produce *(j, (M, i, m_{ij}))*

    given *(j, k, n_{jk})* produce *(j, (N, k, n_{jk}))*

  - **Reduce**: given *(j, [(M, i, m_{ij}), (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

---
1 **for** *each element m_{ij} of M* **do**
2   produce $(key, value)$ pairs as $((i, k), (M, j, m_{ij}))$, for $k = 1, 2, 3, ..$ up to the number of columns of N
3 **for** *each element n_{jk} of N* **do**
4   produce $(key, value)$ pairs as $((i, k), (N, j, n_{jk}))$, for $i = 1, 2, 3, ...$ up to the number of rows of M
5 **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

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

Matrix Matrix Multiplication (III)

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

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

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

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

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

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

$n_{11} = a$ $((i, k), [(M, j, m_{ij}), (M, j, m_{ij}), ..., (N, j, n_{jk}), (N, j, n_{jk}), ...])$
$(1, 1), (N(1, 1), [(M, 1, 1), (M, 2, 2), (M, 3, 3), (N, 1, a, (N, 2, c), (N, 3, e)]$
$(2, 1), (N(1, 2), [(M, 1, 1), (M, 2, 2), (M, 3, 3), (N, 1, b, (N, 2, d), (N, 3, f)]$
$\quad (2, 1), [(M, 1, 4), (M, 2, 5), (M, 3, 6), (N, 1, a, (N, 2, c), (N, 3, e)]$
$n_{21} = c$ $(2, 2), [(M, 1, 4), (M, 2, 5), (M, 3, 6), (N, 1, b, (N, 2, d), (N, 3, f)]$
$(1, 1), (N, 2, c)i = 1$
$(2, 1), (N, 2, c)i = 2$

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

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

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

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

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

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

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
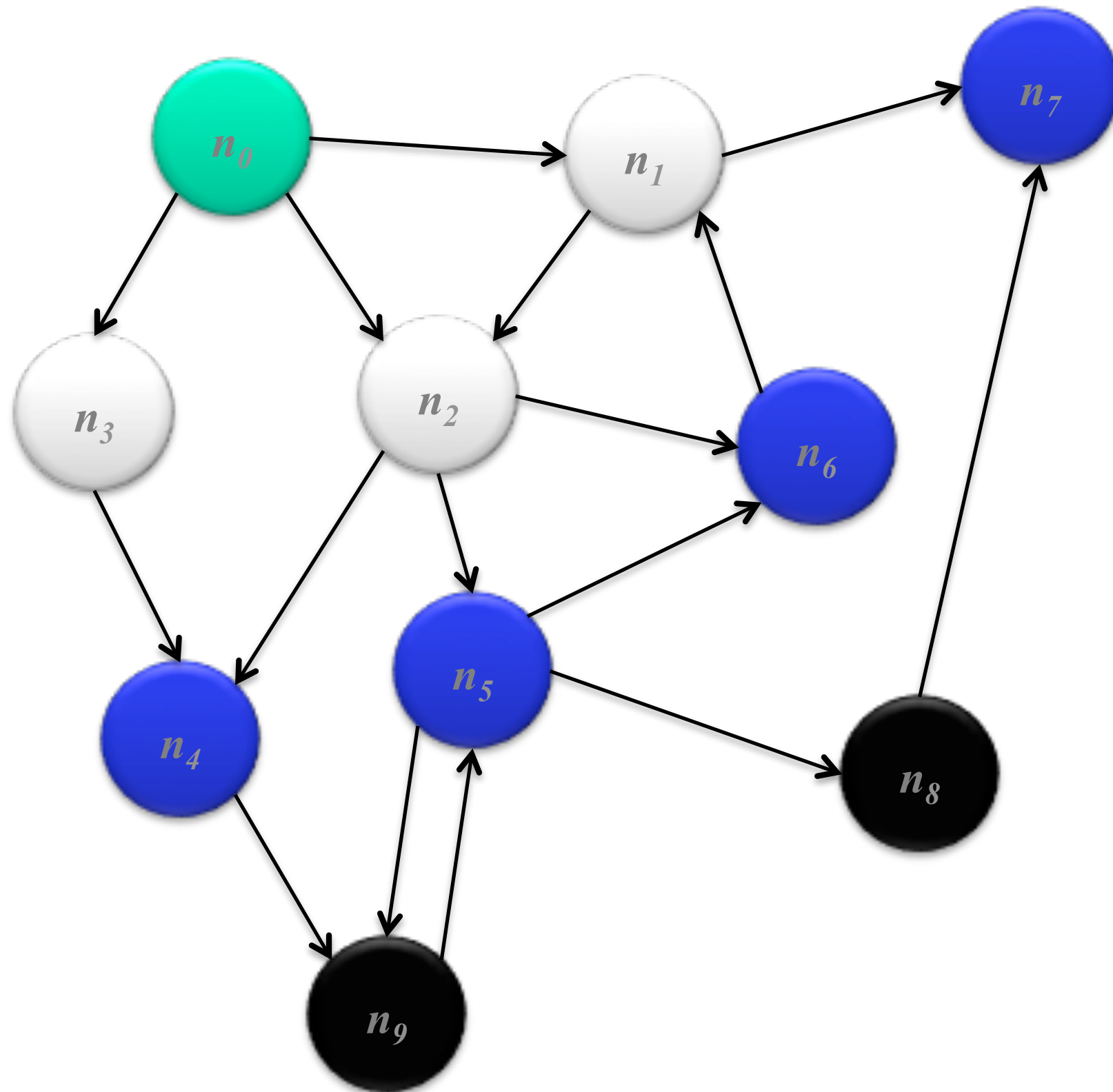
Shortest Path Algorithm

- 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)
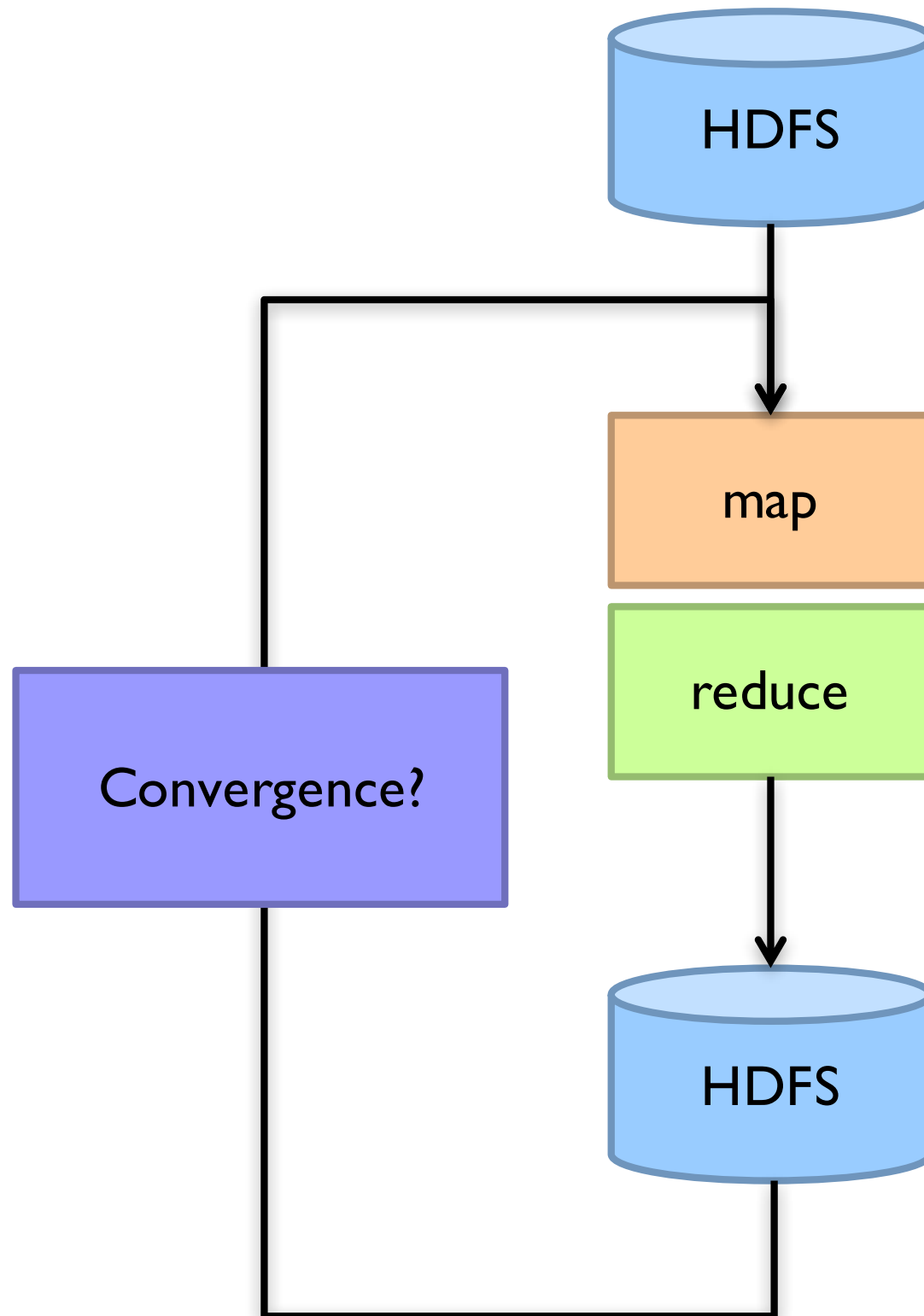
Pseudocode

```
1:  class MAPPER
2:      method MAP(nid n, node N)
3:          d ← N.DISTANCE
4:          EMIT(nid n, N)                               ▷ Pass along graph structure
5:          for all nodeid m ∈ N.ADJACENCYLIST do
6:              EMIT(nid m, d + 1)                       ▷ Emit distances to reachable nodes

1:  class REDUCER
2:      method REDUCE(nid m, [d₁, d₂, . . .])
3:          d_min ← ∞
4:          M ← ∅
5:          for all d ∈ counts [d₁, d₂, . . .] do
6:              if ISNODE(d) then
7:                  M ← d                               ▷ Recover graph structure
8:              else if d < d_min then                  ▷ Look for shorter distance
9:                  d_min ← d
10:         M.DISTANCE ← d_min                          ▷ Update shortest distance
11:         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