Map Reduce Design Patterns

Caveat

- The great power of MapReduce lays in its simplicity
  - The programmer only needs to prepare input data, configuration information, implement mapper and reducer and, optionally, setup, cleanup, combiner and partitioner
  - All other aspects are handled by the framework
- This simplicity has some pros and cons
  - Pros -- It provides clear boundaries for what you can do and cannot do
  - Cons -- The programmer has to express any algorithm in terms of a small number of rigidly defined components that must fit together in very specific ways
- How to recast/reshape an algorithm into this programming model, i.e., as a sequence of one or more MapReduce jobs?
- Good programmers can produce bad MapReduce algorithms
- MapReduce design patterns
  - Principles/guidelines to build better software
  - Not specific to a domain (e.g., text processing). Instead, it is a general approach to solve a problem

Design Patterns

- In-mapper Combining
- Pairs and stripes
- Patterns for relational algebra operators
- Patterns for matrix-vector and matrix-matrix multiplication

In-mapper Combining (I)

Intermediate data (i.e., mappers outputs) are

- Written locally
- Transferred from mappers to reducers over cluster network

- Issue
  - Bottleneck: Writing to disk and transferring over the network are relatively «expensive» operations
  - Reducing intermediate data translates into increasing algorithmic efficiency
- Solution: Local aggregation
  - Use combiners
  - Use In-Mapper Combining

In-Mapper Combining (II)

Pseudo-code of the basic word-count algorithm

```
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 [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

In-Mapper Combining (III)

Pseudo-code of stateless in-mapper combining

```
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 (IV)

Pseudo-code of stateful in-mapper combining (the true in-mapper combining)

```
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 (V)

- Advantages:
  - Combiners are provided as an optimization to the execution framework, which has the option of using it or not at all. On the other hand, in-mapper combining provides complete control on local aggregation process (how and when)
  - Combiners only reduce the amount of intermediate data that is transferred across the network but do not reduce intermediate data emitted by mappers in the first place and stored on local disk. In-mapper combining provides further control by avoiding unnecessary object creation as output to the mapper
- Disadvantages:
  - Memory management is more complex – no automatic spilling is performed as it happens with an external combiner. Therefore, the internal associative array may no longer fit in memory. Solved by:
    - Either emitting intermediate data after processing every *n* (how big?) input records rather than all of them
    - Or monitoring memory footprint and emit intermediate data once memory usage has exceeded a certain (how big?) threshold

Matrix Generation

- Common problem:
  - Given an input of size *N*, generate a squared output of size *N x N*
- Example: word co-occurrence matrix
  - Given a document collection, emit the bigram counts
  - Where *N* is the number of unique words
    - Cell m_{ij} contains the number of times word w_i co-occurs with word w_j
    - Used for example in statistical natural language processing
- Two solutions
  - Pairs
  - Stripes

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)
```

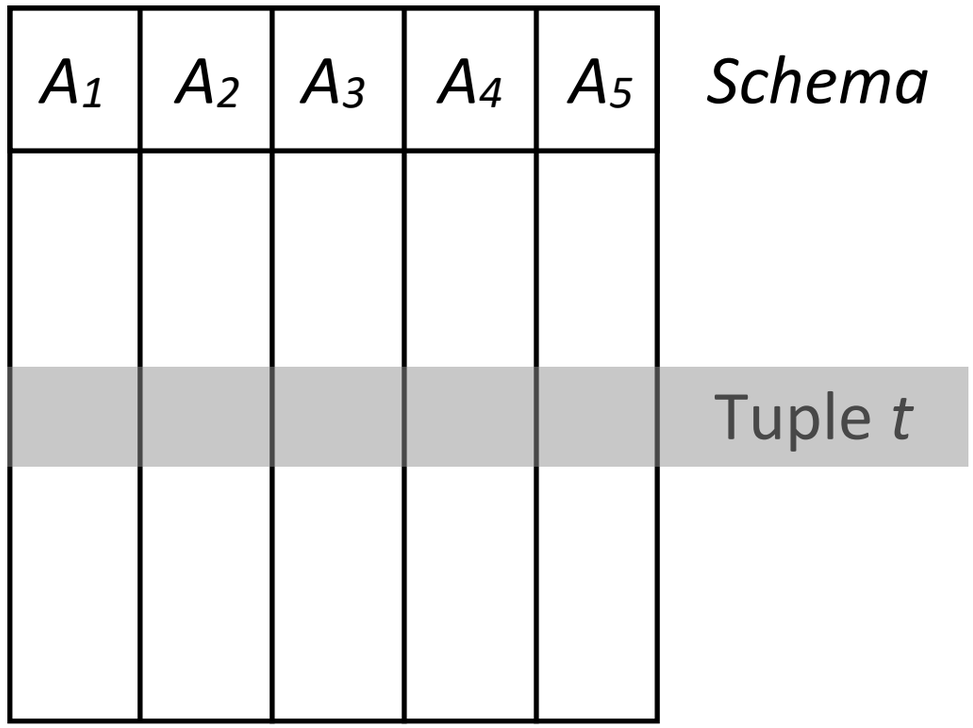
Stripes

```
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 [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                         ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

Relational Algebra Operators

Use case:
Search engines



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

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

Projection

- Map: each tuple *t* in *R*, create a new tuple *t′* containing only the projected attributes and output a *(t′, ⊥)* pair
- Reduce: for each *(t′, ⊥, ⊥, ⊥, ...)* pair in input, output *(t′, ⊥)*

Union, intersection and difference

Union

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

Intersection

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

Difference (what is in R, which is not in S)

- 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, ⊥)*, 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

- 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 be a list of values of the form *(*R*, a)* or *(*S*, c)*. Construct all pairs and output them together with *b*. *Emit ((a, b, 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.

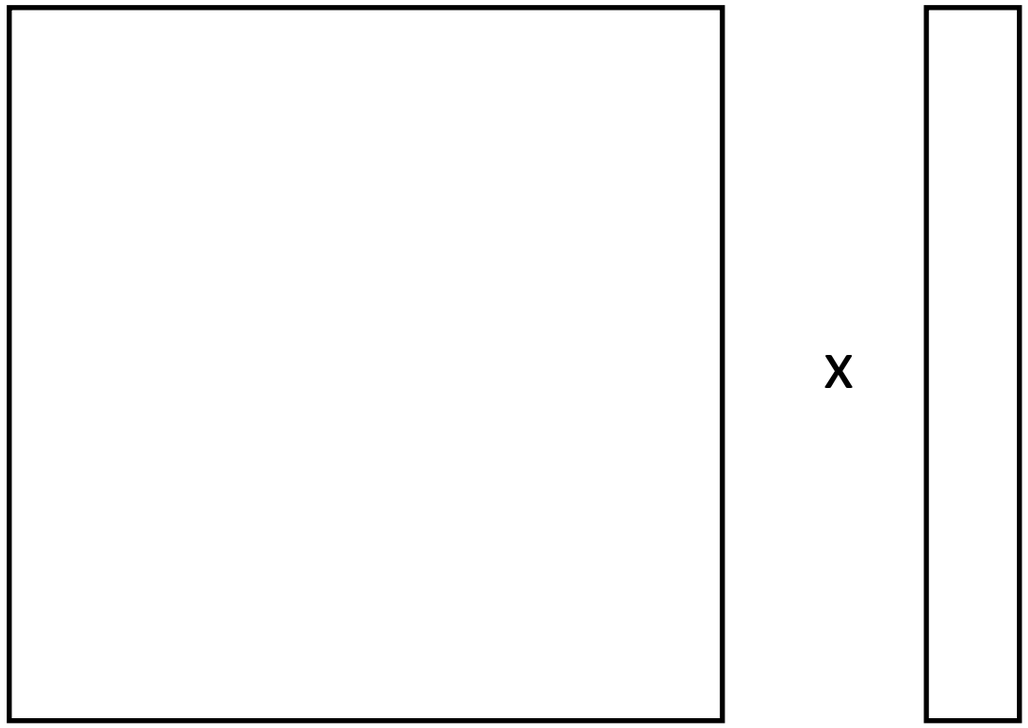
Grouping and aggregation

- 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

Matrix-Vector Multiplication



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

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

The matrix does not fit in memory (huge *m*), and

1. The vector *v* does fit in a machine's memory (small *n*)
2. The vector *v* does not fit in machine's memory (huge *n*)

A graphical and simple example of Matrix Vector Multiplication

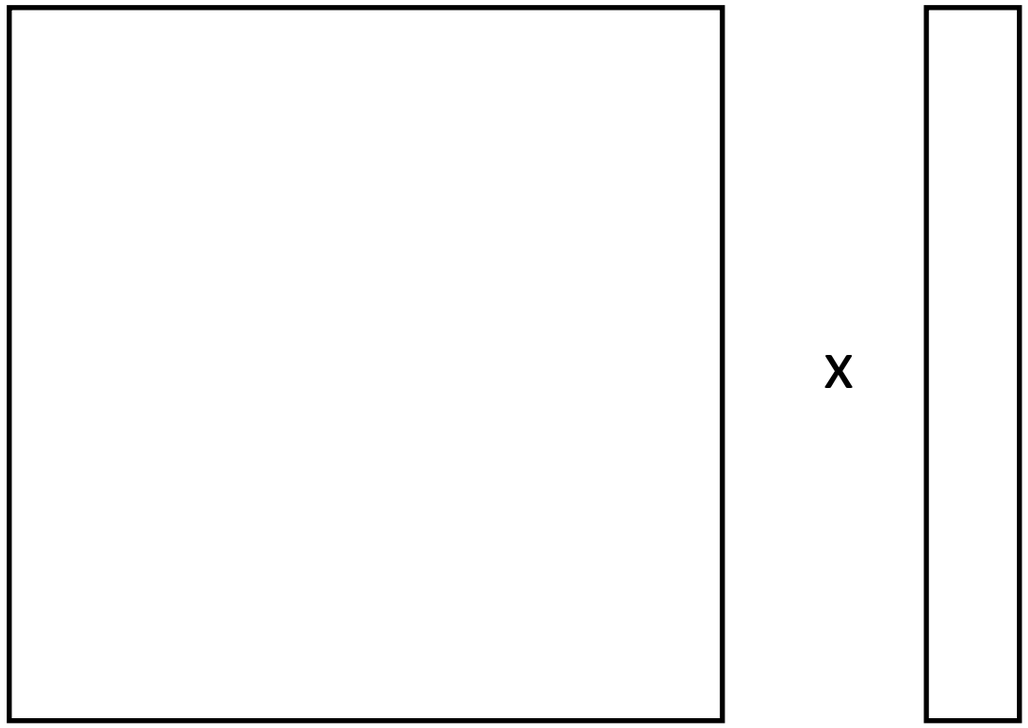
1	5		7		17
2	4	x	2	=	22
3	6				33

Matrix *M* size *3* x *2*

Vector *v* size *2* x *1*

Vector *r* size *3* x *1*

Vector does fit



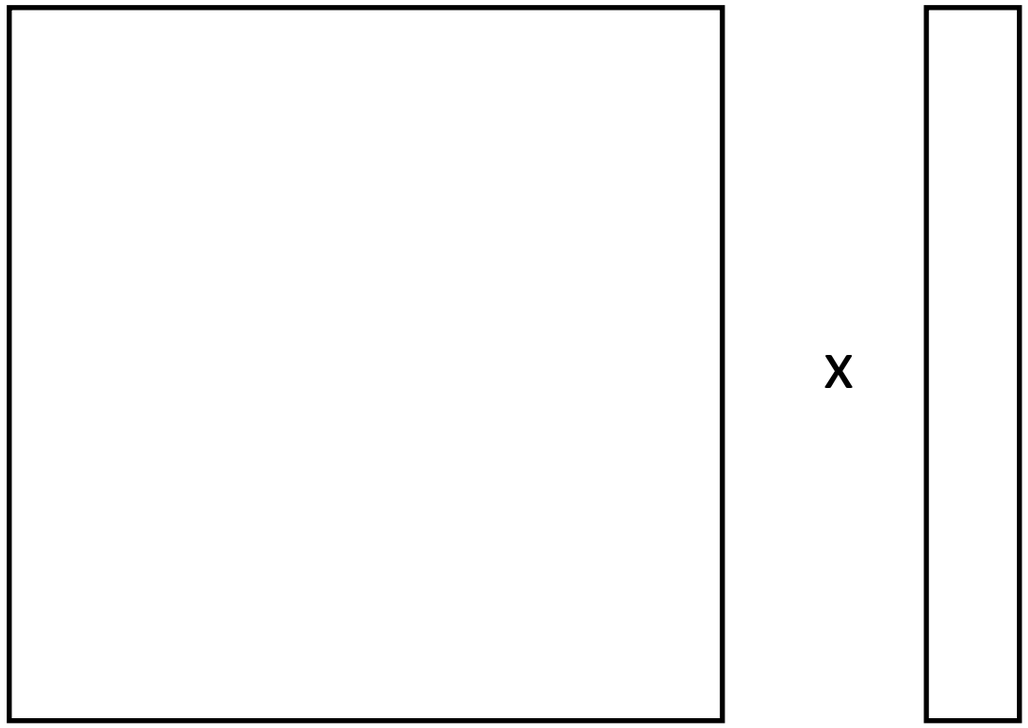
Matrix *M*
size *m* x *n*

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 → ($i, m_{ij}v_j$) pair
- Reduce: (i, [$m_{i1}v_1, m_{i2}v_2, ..., m_{in}v_n$]) pair → ($i, m_{i1}v_1 + m_{i2}v_2 + ... + m_{in}v_n$) pair

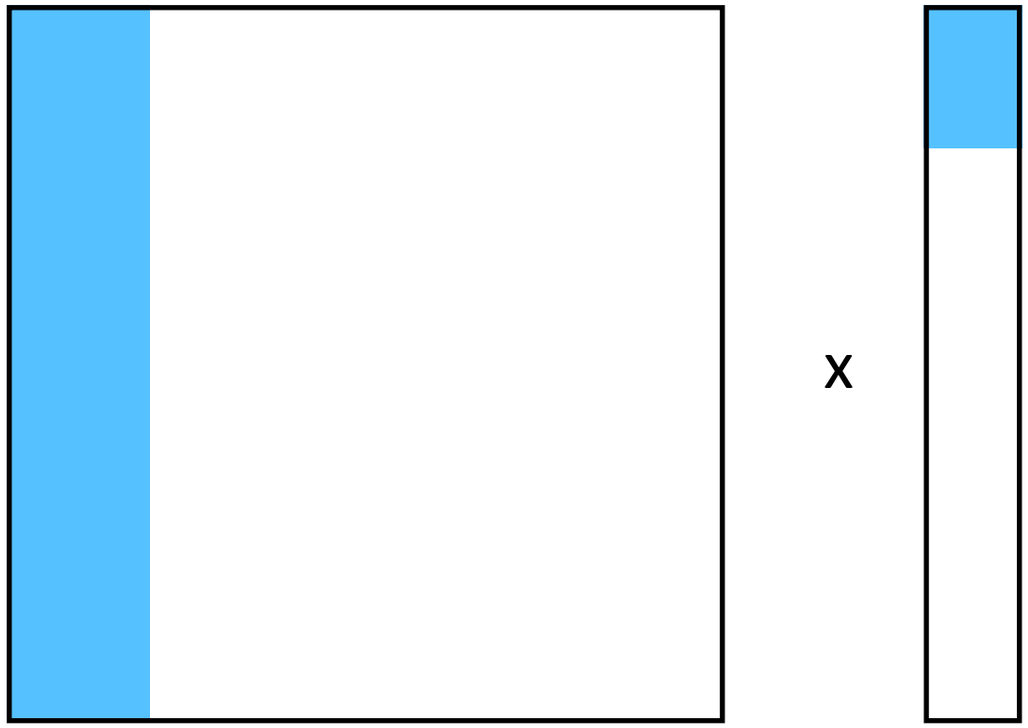
Vector does not fit



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

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

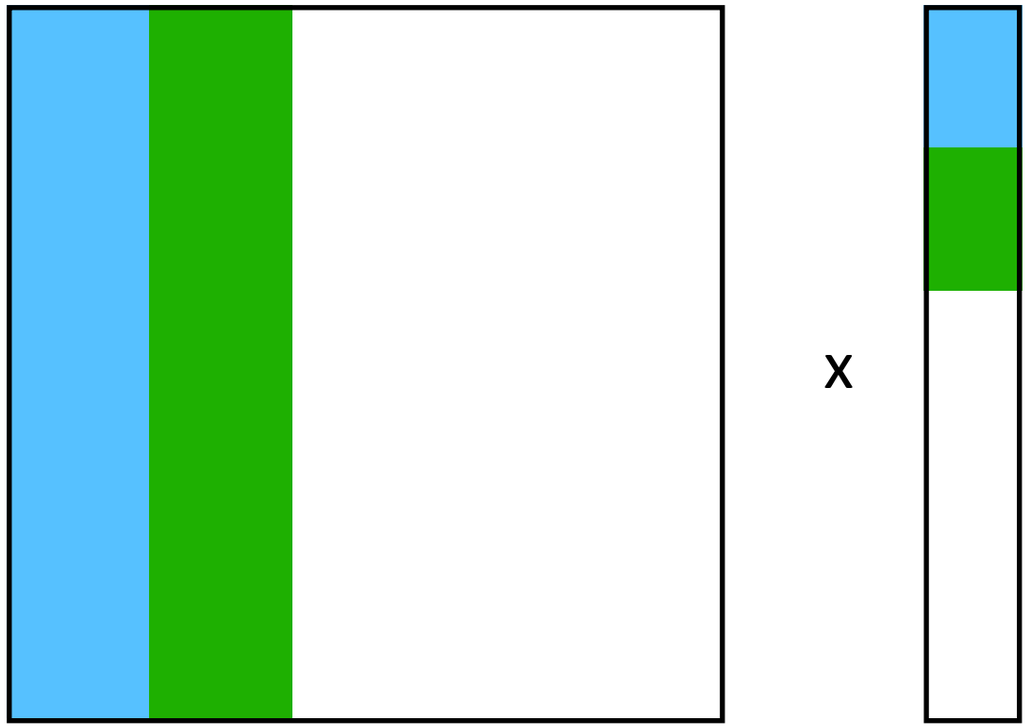
Vector does not fit



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

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

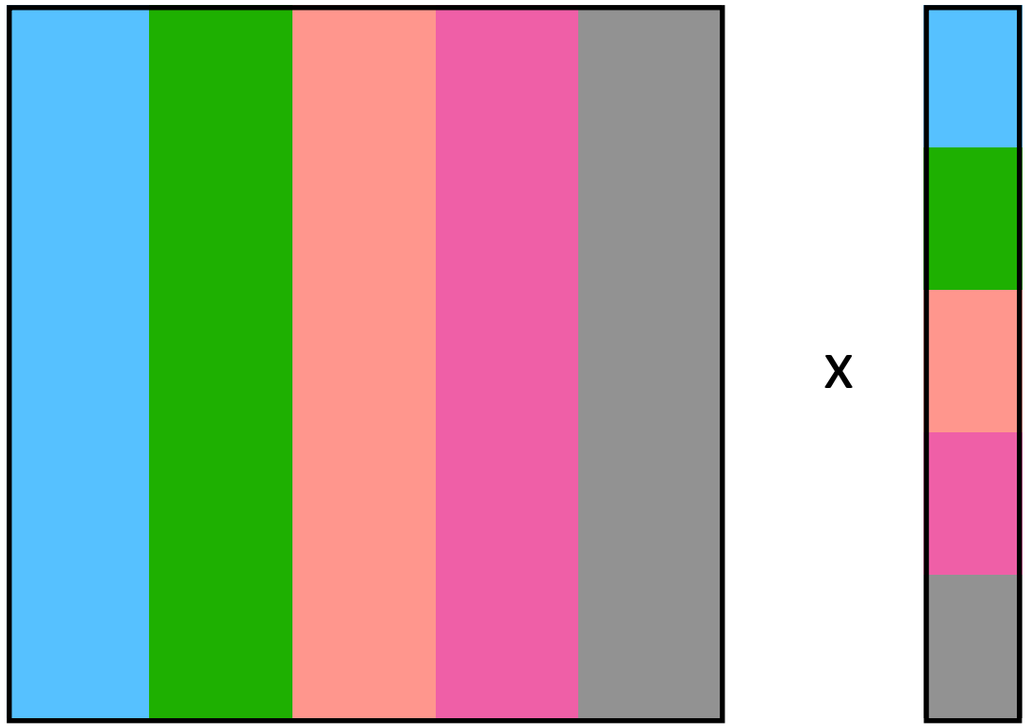
Vector does not fit



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

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

Vector does not fit

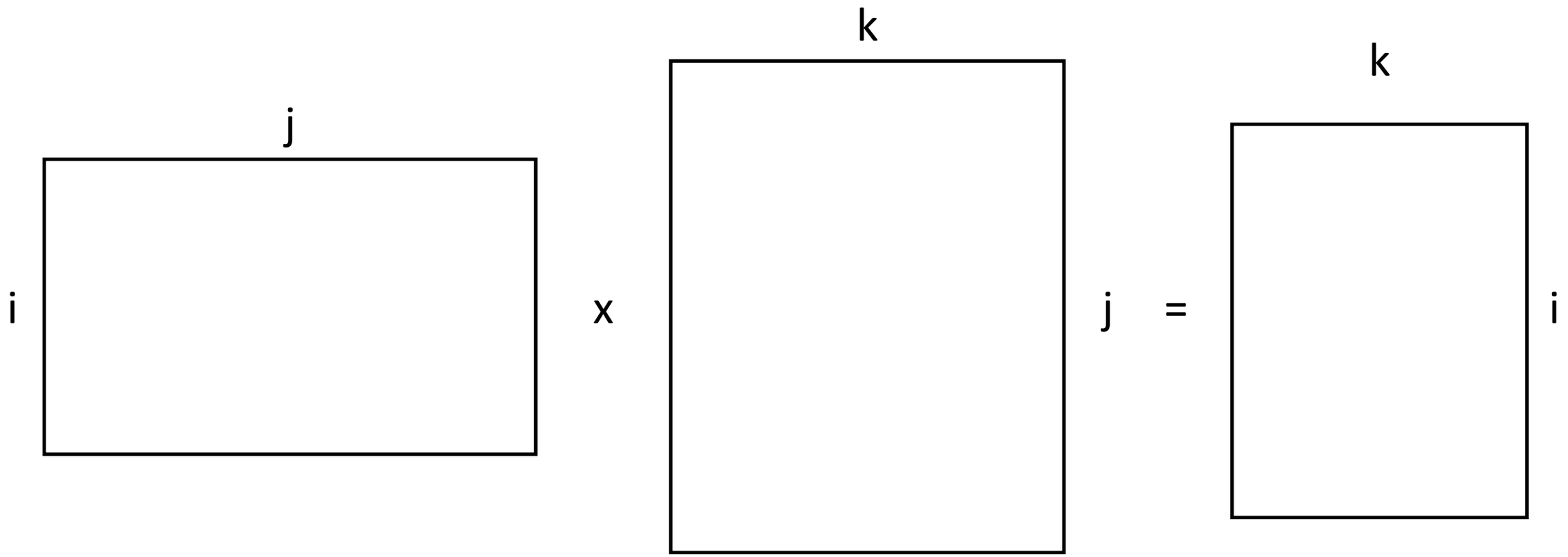


Matrix *M*
size *m* x *n*

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

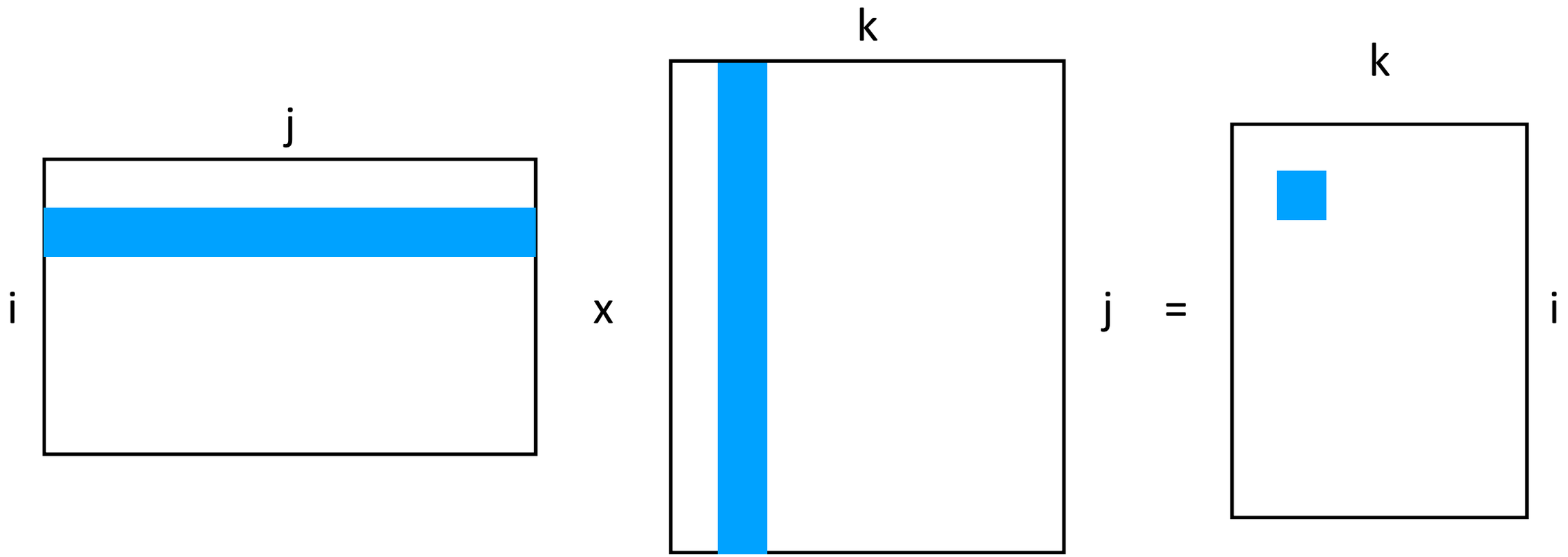
Matrix-Matrix Multiplication (I)



Matrix *M* 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-Matrix Multiplication (II)

- A matrix can be seen as a 4-attributes relation/table:
  - (Matrix name, row index, column index, value) tuples
  - $M \rightarrow (M, i, j, m_{ij})$, $N \rightarrow (N, j, k, n_{jk})$
- As large matrices are often sparse (0's) we omit such tuples

Matrix-Matrix Multiplication (III)

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 (IV)

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

Matrix Matrix Multiplication (IV)

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

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

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

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

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

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

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

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

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

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

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

Matrix Matrix Multiplication (IV)

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

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

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

Matrix Matrix Multiplication (IV)

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

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

$$list_M = [(M,1,1),(M,2,2),(M,3,3)]$$

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

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

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