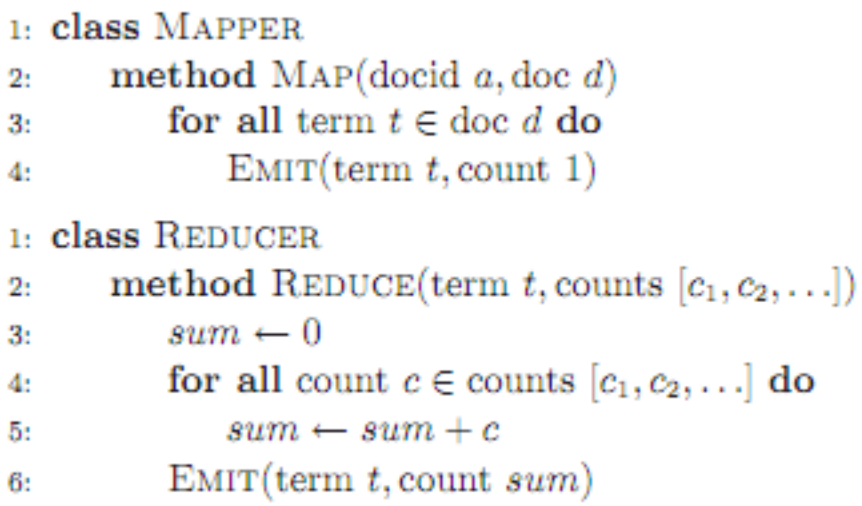
Map Reduce Algorithm Design

Local Aggregation

- Intermediate data
  - Written locally
  - Transferred over network
- Performance Bottleneck
- Use Combiners
- Use In-Mapper Combining

Original Word Count



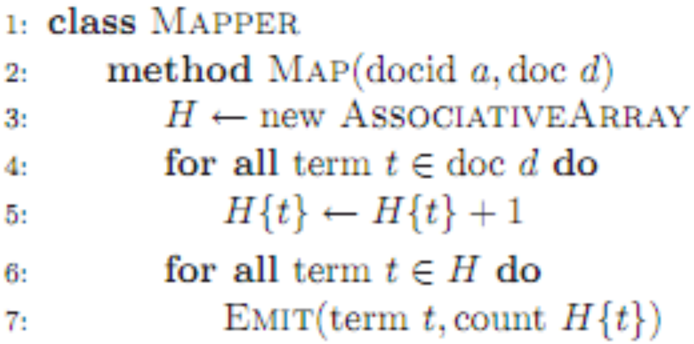
```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            Emit(term $t$, count 1)

class Reducer
    method Reduce(term $t$, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $sum \leftarrow sum + c$
        Emit(term $t$, count $sum$)
```

- How many intermediate keys per mapper?
- How can we improve this?
- Is it a “real” improvement?

Taken from “Data-Intensive Text Processing with MapReduce”, Jimmy Lin and Chris Dyer, Morgan & Claypool Publisher, 2010, pag. 42

Stateless In-Mapper Combining

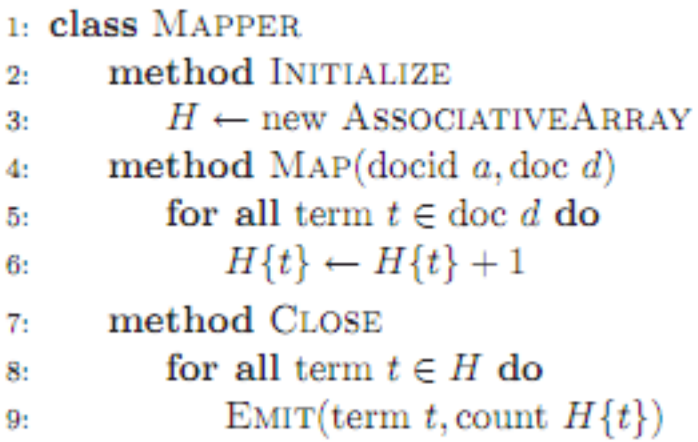


```
class Mapper
    method Map(docid $a$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(term $t$, count $H\{t\}$)
```

- Custom local aggregator
- Coding overhead
- Is it a “real” improvement?

Taken from “Data-Intensive Text Processing with MapReduce”, Jimmy Lin and Chris Dyer, Morgan & Claypool Publisher, 2010, pag. 43

Statefull In-Mapper Combining



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

- Custom local aggregator with state
- Coding overhead
- Is it a “real” improvement?

Taken from “Data-Intensive Text Processing with MapReduce”, Jimmy Lin and Chris Dyer, Morgan & Claypool Publisher, 2010, pag. 44

In-Mapper Combining Analysis

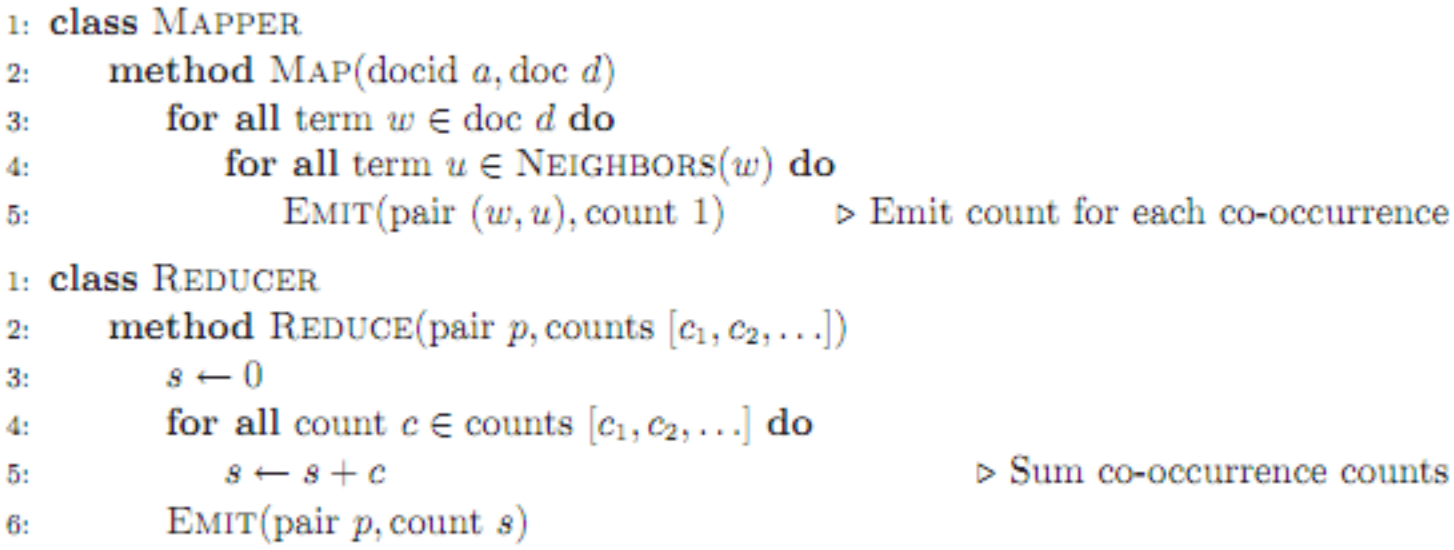
- Advantages:
  - Complete local aggregation control (how and when)
  - Guaranteed to execute
  - Direct efficiency control on intermediate data creation
  - Avoid unnecessary objects creation and destruction (before combiners)
- Disadvantages:
  - Breaks the functional programming background (state)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

Taken from "Data-Intensive Text Processing with MapReduce", Jimmy Lin and Chris Dyer, Morgan & Claypool Publisher, 2010, pag. 43

Matrix Generation

- Common problem: given an input of size N, generate an output matrix of size N x N
- Example: word co-occurrence matrix
  - Given a document collection, emit the **bigram** frequencies

“Pairs” Solution



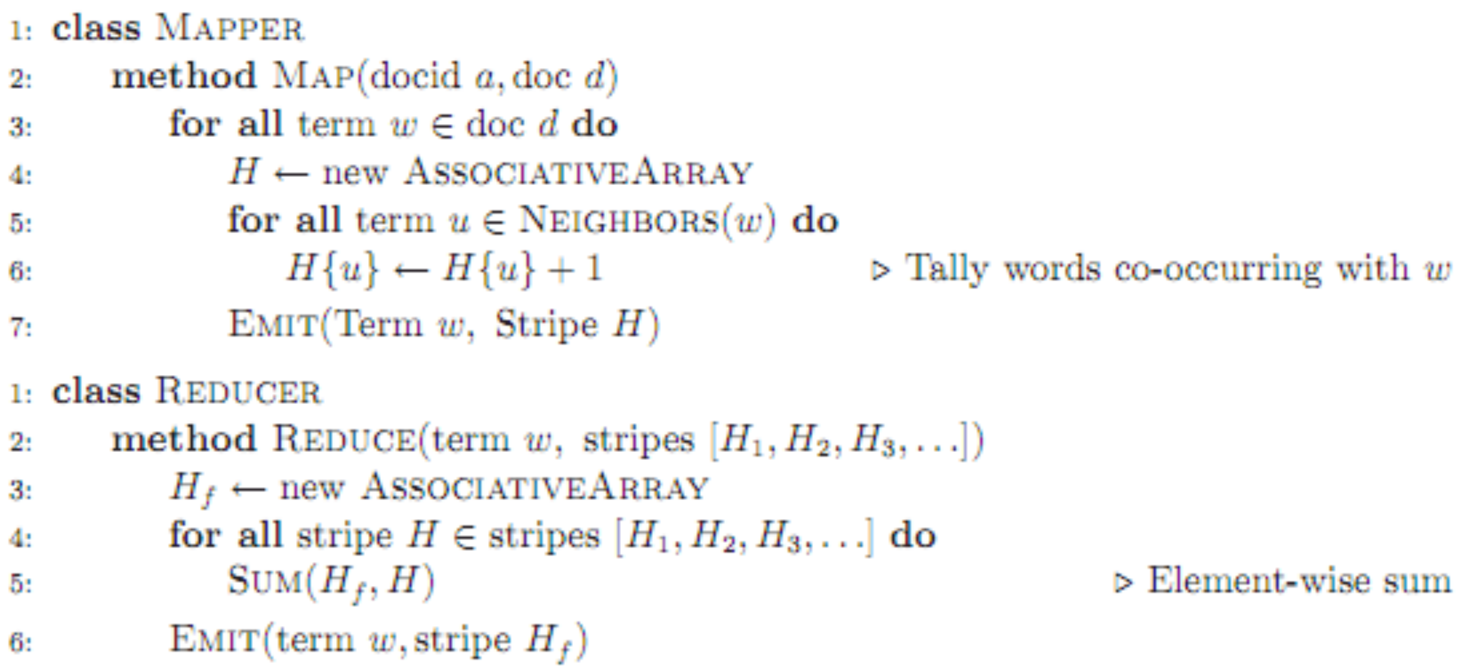
```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $w \in$ doc $d$ do
            for all term $u \in$ Neighbors($w$) do
                Emit(pair $(w, u)$, count 1)        ▷ Emit count for each co-occurrence

class Reducer
    method Reduce(pair $p$, counts $[c_1, c_2, \ldots]$)
        $s \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $s \leftarrow s + c$                    ▷ Sum co-occurrence counts
        Emit(pair $p$, count $s$)
```

- We must use custom key type
- Intermediate overhead? Bottlenecks?
- Can we use the reducer as a combiner?

Taken from “Data-Intensive Text Processing with MapReduce”, Jimmy Lin and Chris Dyer, Morgan & Claypool Publisher, 2010, pag. 53

“Stripes” Solution



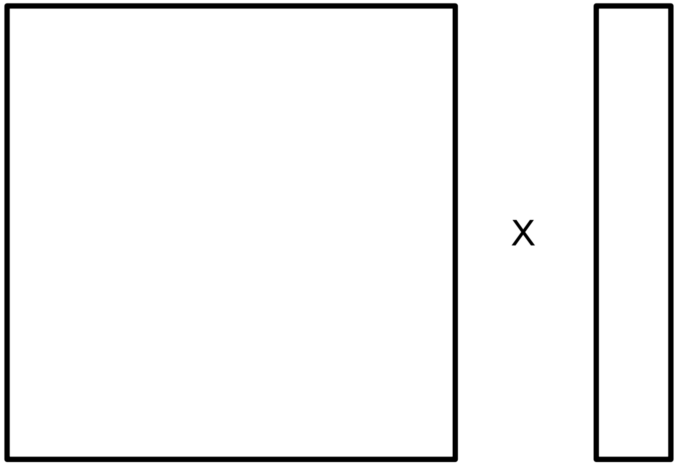
```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $w \in$ doc $d$ do
            $H \leftarrow$ new AssociativeArray
            for all term $u \in$ Neighbors($w$) do
                $H\{u\} \leftarrow H\{u\} + 1$                ▷ Tally words co-occurring with $w$
           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

class Reducer
    method Reduce(term $w$, stripes $[H_1, H_2, H_3, \ldots]$)
        $H_f \leftarrow$ new AssociativeArray
        for all stripe $H \in$ stripes $[H_1, H_2, H_3, \ldots]$ do
            Sum($H_f$, $H$)                                   ▷ Element-wise sum
        Emit(term $w$, stripe $H_f$)
```

- We must use custom key and value types
- Intermediate overhead? Bottlenecks?
- Can we use the reducer as a combiner?

Taken from “Data-Intensive Text Processing with MapReduce”, Jimmy Lin and Chris Dyer, Morgan & Claypool Publisher, 2010, pag. 53

Matrix-Vector Multiplication



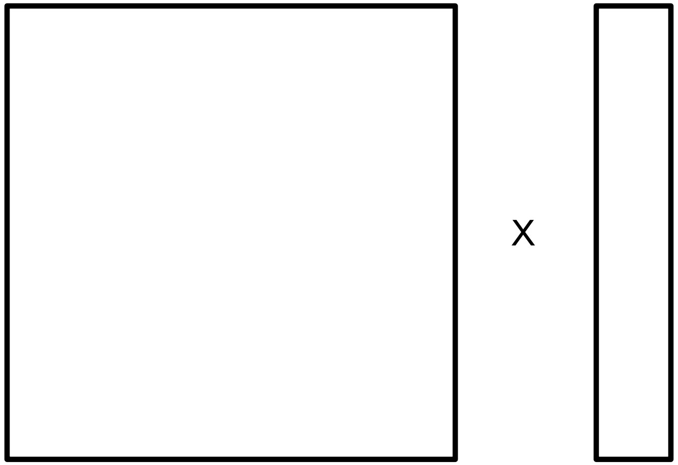
x

Matrix M

Vector **v**

- The matrix does not fit in memory
  - 1 case: vector **v** fits in memory
  - 2 case: vector **v** does not fit in memory

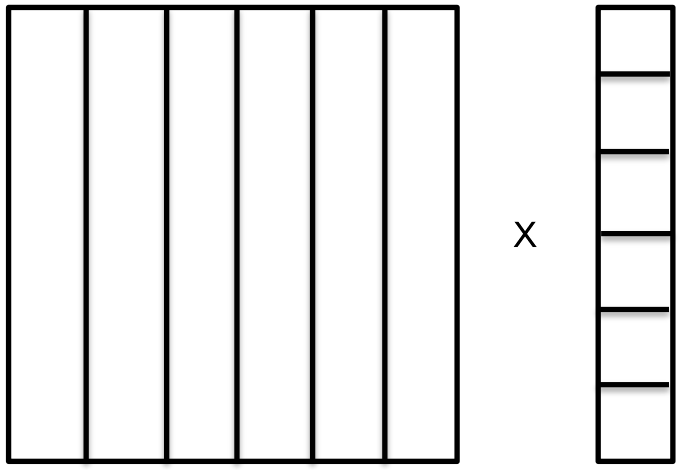
Matrix-Vector Multiplication (1)



Matrix M   Vector **v**

- Map
  - input = (*, chunk of matrix M)
  - vector **v** read from memory
  - output = $(i, m_{ij}v_j)$
- Reduce
  - sum up all the values for the given key i

Matrix-Vector Multiplication (2)

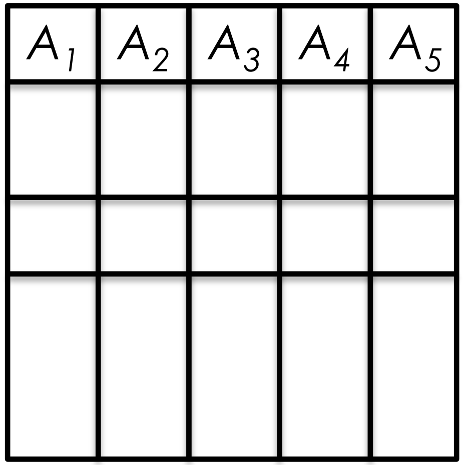


Matrix M Vector **v**

- 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

Relational Algebra

Attributes



Schema

Tuple

Relation *R*

- SELECTION: Select from R tuples satiasfying condition C
- PROJECTION: For each tuple in R, select only certain attributes
- UNION, INTERSECTION, DIFFERENCE: Set operations on two relations with same schema
- NATURAL JOIN
- GROUPING and AGGREGATION

Selections and Projections

- MAP: Each tuple *t*, if condition C is satisfied, is outputted as a (*t*, *t*) pair
- REDUCE: Identity

- MAP: For each tuple *t*, create a new tuple *t'* containing only projected attributes. Outpu is (*t'*, *t'*) pair
- REDUCE: Coalesce input (*t'*, *[t' t' t' t']*) in output *(t',t')*

Unione, Intersection and Difference

- MAP: Each tuple *t* is outputted as a (*t*, *t*) pair
- REDUCE: For each key *t*, there will be 1 or 2 values *t*. Coalesce them in a single output *(t,t)*

- MAP: Each tuple *t* is outputted as a (*t*, *t*) pair
- REDUCE: For each key *t*, there will be 1 or 2 values *t*. If 2 values, coalesce them in a single output *(t,t),* else ignore

- MAP: For each tuple *t* in *R*, produce (*t*, *"R"*). For each tuple *t* in *S*, produce (*t*, *"S"*).
- REDUCE: For each key *t*, there will be 1 or 2 values *t*. If 1 value, and being "R", output *(t,t),* else ignore

Natural Join

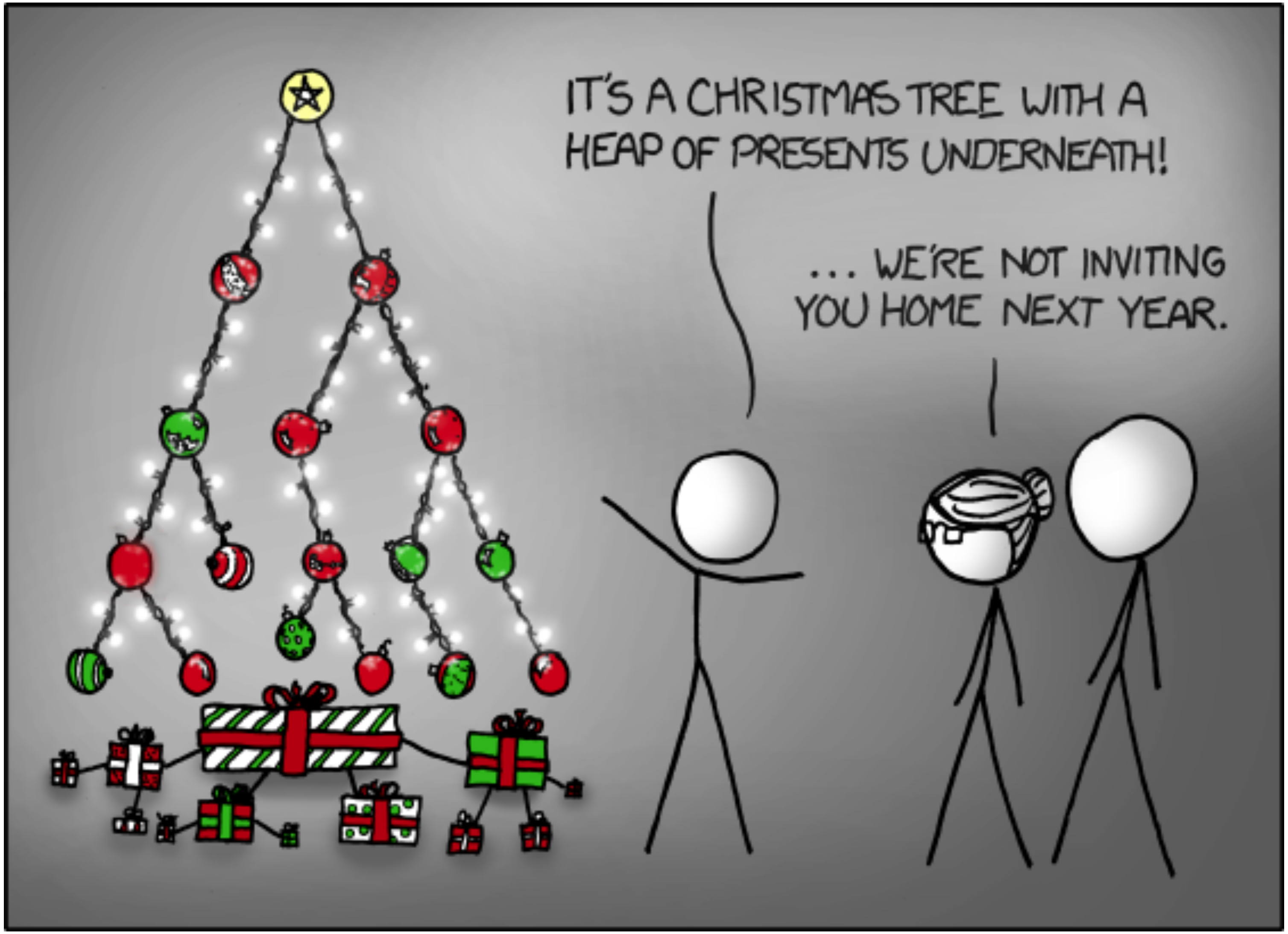
We have two relations *R(A,B)* and *S(B,C)*. Find tuples that agree on *B* components

- MAP: For each tuple *(a,b)* from *R*, produce *(b, ("R",a))*. For each tuple *(b,c)* from *S*, produce *(b,("S",c))*.
- REDUCE: For each key *b*, there will a list of values of the form *("R",a)* or *("S",c)*. Construct all pairs and output them with *b*.

Grouping and Aggregation

We have the relation *R(A,B,C)* and we **group-by** *A* and **aggregate** on *B*.

- 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 associate with group *a,* producing x. Output *(a,x)*.


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.