

بسم الله الرحمن الرحيم

CMPS 653

Big Data Analytics

Session 3:

MapReduce Algorithm Design

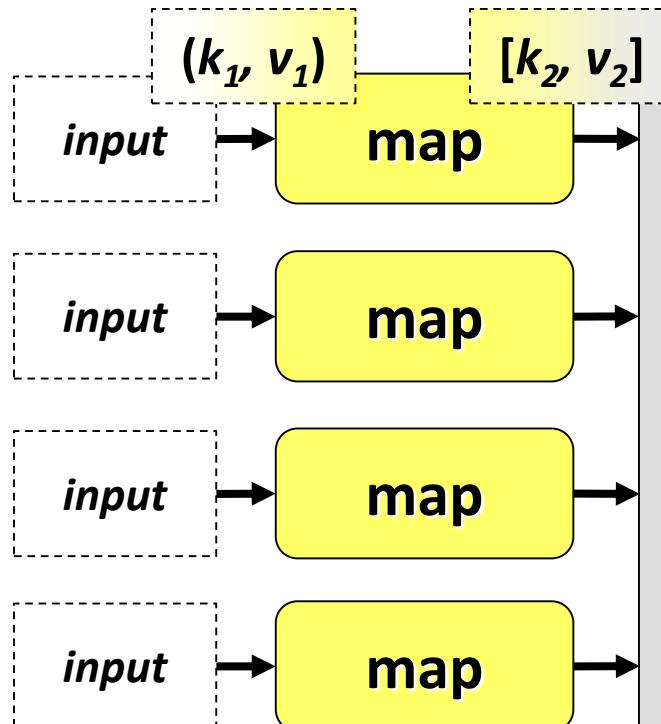
Some slides are adopted from Jimmy Lin's by permission



Tamer Elsayed

MapReduce Framework: Recap

(a) Map

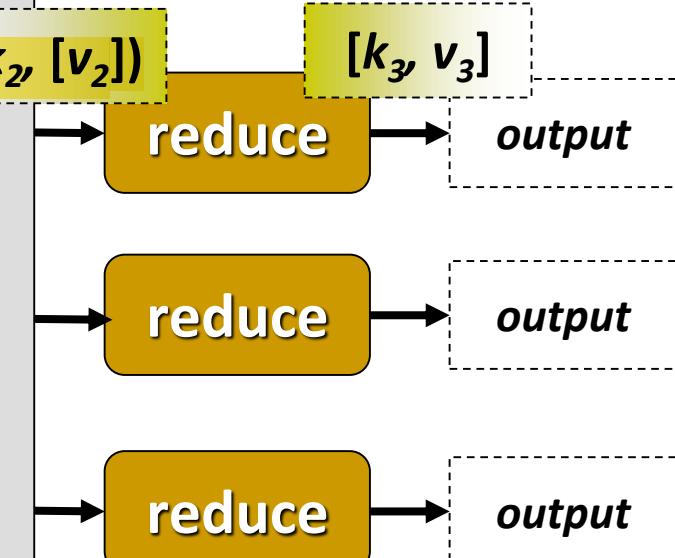


(b) Shuffle

Shuffling

**group values
by: [keys]**

(c) Reduce



Framework handles “everything else” !

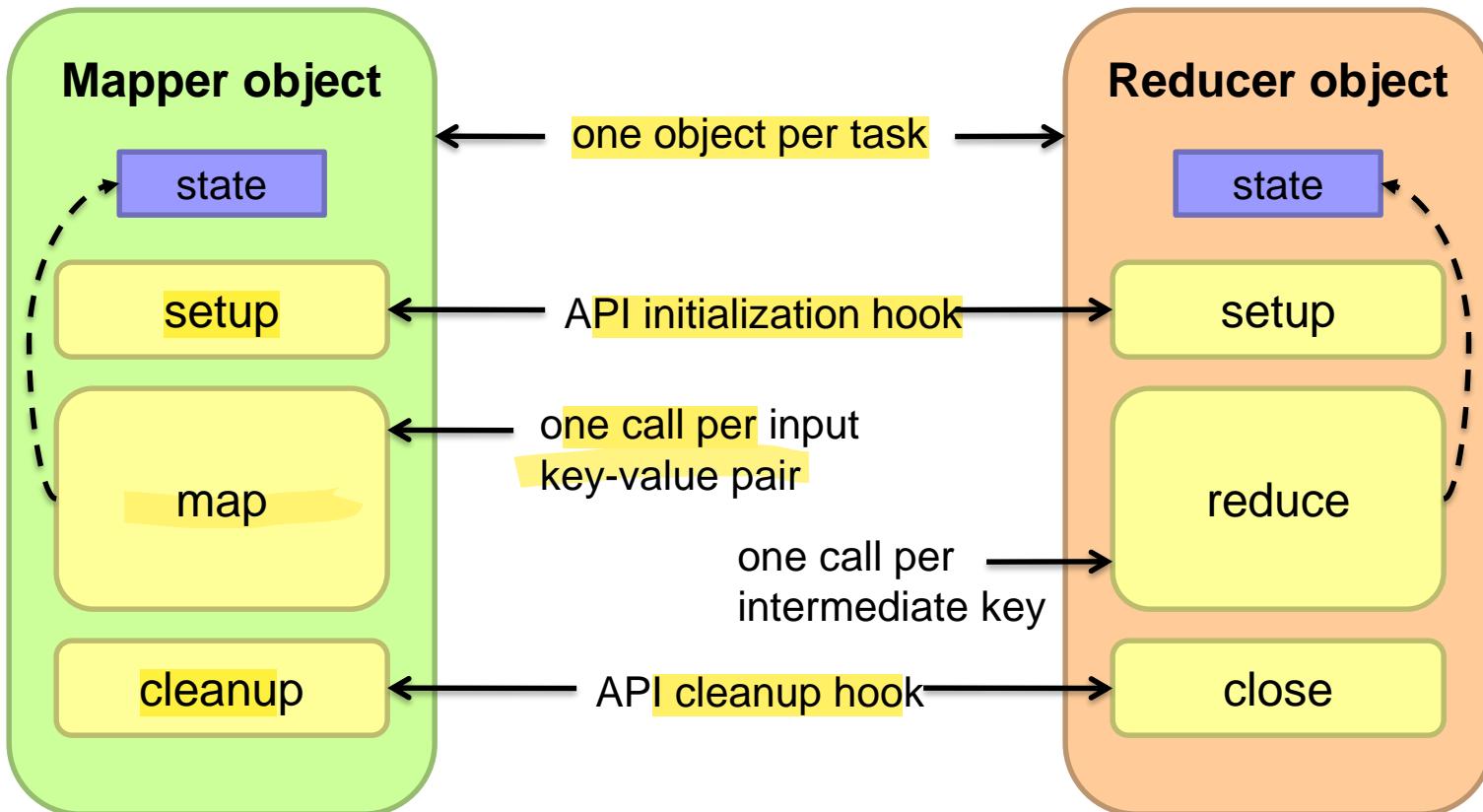
“Everything Else”

- The execution framework handles everything else...
 - Scheduling: assigns workers to map and reduce tasks
 - “Data distribution”: moves processes to data
 - Synchronization: gathers, sorts, and shuffles intermediate data
 - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
 - All algorithms must be expressed in m, r, c, p
- You don't know:
 - Where mappers and reducers run
 - When a mapper or reducer begins or finishes
 - Which input a particular mapper is processing

Tools for Synchronization

- Cleverly-constructed data structures
 - Bring partial results together
- Sort order of intermediate keys
 - Control order in which reducers process keys
- Partitioner
 - Control which reducer processes which keys
- Preserving state in mappers and reducers
 - Capture dependencies across multiple keys and values

Preserving State



Scalable Hadoop Algorithms: Themes

- Avoid object creation
 - Inherently costly operation
 - Garbage collection
- Avoid buffering
 - Limited heap size
 - Works for small datasets, but won't scale!

Design Patterns

- In-Mapper Combining
- Pairs and Stripes
- Order Inversion
- Value-to-key Conversion



Local Aggregation: “In-Mapper Combining” Design Pattern

Importance of Local Aggregation

○ Ideal scaling characteristics:

- Twice the data, twice the running time
- Twice the resources, half the running time

○ Why can't we achieve this?

- Synchronization requires communication
- Communication kills performance

○ Thus... avoid communication!

- Reduce intermediate data via local aggregation
- Combiners can help

Word Count: Baseline

```
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 s)
```

What's the impact of combiners?

Word Count: Version 1

```
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})
```

▷ Tally counts for entire document

Are combiners still needed?

Word Count: Version 2

```
1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow$  new ASSOCIATIVEARRAY
4:   method MAP(docid  $a$ , doc  $d$ )
5:     for all term  $t \in$  doc  $d$  do
6:        $H\{t\} \leftarrow H\{t\} + 1$ 
7:   method CLOSE
8:     for all term  $t \in H$  do
9:       EMIT(term  $t$ , count  $H\{t\}$ )
```

Key idea: preserve state across
input key-value pairs!

▷ Tally counts across documents

Are combiners still needed?

In-Mapper Combining

- Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages?
 - Speed
 - Why is this faster than actual combiners?
- Disadvantages?
 - Order-dependent: against “MapReduce” model
 - Scalability bottleneck!
 - Explicit memory management required

Combiner Design

- Remember: combiner are optional optimizations
 - Should not affect algorithm correctness
 - May be run 0, 1, or multiple times
- Combiners and reducers share same method input
signature
 - Sometimes, reducers can serve as combiners
 - Often, not...
- Example: find average of integers associated with the same key

Computing the Mean: Version 1

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, integer r)

1: class REDUCER
2:     method REDUCE(string t, integers [r1, r2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r1, r2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:             ravg ← sum/cnt
9:             EMIT(string t, integer ravg)
```

Can we use reducer as combiner?

Computing the Mean: Version 2

```

1: class MAPPER
2:   method MAP(string t, integer r)
3:     EMIT(string t, integer r)

1: class COMBINER
2:   method COMBINE(string t, integers [r1, r2, ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all integer r ∈ integers [r1, r2, ...] do
6:       sum ← sum + r
7:       cnt ← cnt + 1
8:     EMIT(string t, pair (sum, cnt))           ▷ Separate sum and count

1: class REDUCER
2:   method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:       sum ← sum + s
7:       cnt ← cnt + c
8:     ravg ← sum/cnt
9:     EMIT(string t, integer ravg)

```

Will that really work?

Computing the Mean: Version 3

```
1: class MAPPER
2:     method MAP(string t, integer r)
3:         EMIT(string t, pair (r, 1))

1: class COMBINER
2:     method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         EMIT(string t, pair (sum, cnt))

1: class REDUCER
2:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         ravg ← sum / cnt
9:         EMIT(string t, pair (ravg, cnt))
```

Fixed?

Computing the Mean: Version 4

```
1: class MAPPER
2:     method INITIALIZE
3:          $S \leftarrow$  new ASSOCIATIVEARRAY
4:          $C \leftarrow$  new ASSOCIATIVEARRAY
5:     method MAP(string  $t$ , integer  $r$ )
6:          $S\{t\} \leftarrow S\{t\} + r$ 
7:          $C\{t\} \leftarrow C\{t\} + 1$ 
8:     method CLOSE
9:         for all term  $t \in S$  do
10:            EMIT(term  $t$ , pair ( $S\{t\}$ ,  $C\{t\}$ ))
```

Are combiners still needed?

“Pairs” and “Stripes” Design Patterns

Running Example

- Term co-occurrence matrix for a text collection
 - $M = N \times N$ matrix ($N = \underline{\text{vocabulary size}}$)
 - M_{ij} : number of times i and j co-occur in some context
(for concreteness, let's say context = sentence)
- Why?
 - Distributional profiles as a way of measuring semantic distance
 - Semantic distance useful for many language processing tasks

Large Counting Problems

- Term co-occurrence matrix for a text collection
= specific instance of a large counting problem
 - A large event space (number of terms²)
 - A large number of observations (the collection itself)
 - Goal: keep track of interesting statistics about the events
- Basic approach
 - Mappers generate partial counts
 - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?

First Try: “Pairs”

- Each mapper takes a sentence:
 - Generate all co-occurring term pairs
 - For all pairs, emit $(a, b) \rightarrow \text{count}$
- Reducers sum up counts associated with these pairs
- Use combiners!

Pairs: Pseudo-Code

```
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)
```

“Pairs” Analysis

- Advantages
 - Easy to implement, easy to understand

- Disadvantages
 - Lots of pairs to sort and shuffle around (upper bound?)
 - Not many opportunities for combiners to work

Another Try: “Stripes”

- Idea: group together pairs into an associative array

$(a, b) \rightarrow 1$

$(a, c) \rightarrow 2$

$(a, d) \rightarrow 5$

$(a, e) \rightarrow 3$

$(a, f) \rightarrow 2$

$a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

- Each mapper takes a sentence:

- Generate all co-occurring term pairs

- For each term, emit $a \rightarrow \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \dots \}$

- Reducers perform element-wise sum of associative

arrays

$$\begin{array}{r}
 a \rightarrow \{ b: 1, \quad d: 5, e: 3 \} \\
 + \quad a \rightarrow \{ b: 1, c: 2, d: 2, \quad f: 2 \} \\
 \hline
 a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}
 \end{array}$$

Key idea:
cleverly-constructed data structure
brings together partial results

Stripes: Pseudo-Code

```

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

```



```

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

```

“Stripes” Analysis

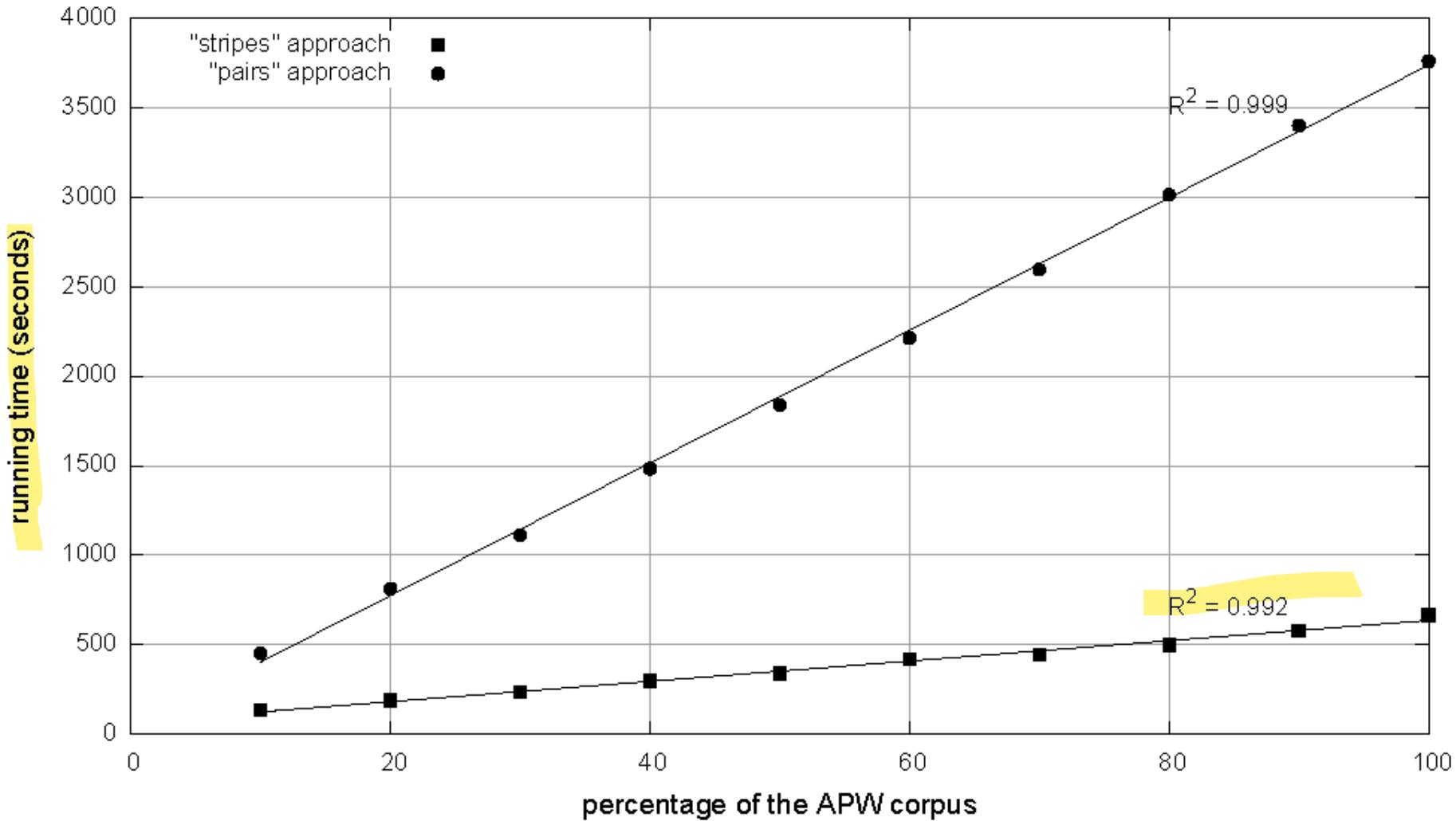
○ Advantages

- Far less sorting and shuffling of key-value pairs
- Can make better use of combiners

○ Disadvantages

- More difficult to implement
- Underlying object more heavyweight
- Scalability bottleneck!

Comparison of "pairs" vs. "stripes" for computing word co-occurrence matrices



Cluster size: 38 cores

Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)

Computing Relative Frequencies: “Order Inversion” Design Pattern



Relative Frequencies

- How do we estimate relative frequencies from counts?

$$f(B|A) = \frac{N(A, B)}{N(A)} = \frac{N(A, B)}{\sum_{B'} N(A, B')}$$

- Why do we want to do this?
- How do we do this with MapReduce?

$f(B|A)$: “Stripes”

$a \rightarrow \{b_1:3, b_2:12, b_3:7, b_4:1, \dots\}$

- Easy!

- One pass to compute $(a, *)$
- Another pass to directly compute $f(B|A)$

$f(B | A)$: “Pairs”

- What’s the issue?

- Computing relative frequencies requires marginal counts
- But the marginal cannot be computed until you see all counts

- Buffering?

- a bad idea!

- Solution:

- What if we could get the marginal count to arrive at the reducer first?

$f(B | A)$: “Pairs”

($a, *$) $\rightarrow 32$ Reducer holds this value in memory

$(a, b_1) \rightarrow 3$

$(a, b_2) \rightarrow 12$

$(a, b_3) \rightarrow 7$

$(a, b_4) \rightarrow 1$

...



$(a, b_1) \rightarrow 3 / 32$

$(a, b_2) \rightarrow 12 / 32$

$(a, b_3) \rightarrow 7 / 32$

$(a, b_4) \rightarrow 1 / 32$

...

- For this to work:

- Must emit extra $(a, *)$ for every b_n in mapper
- Must make sure all a 's get sent to same reducer
 - use partitioner
- Must make sure $(a, *)$ comes first
 - define sort order
- Must hold state in reducer across different key-value pairs

f(B | A): “Pairs”

| key | values |
|-----------------|-------------------|
| (dog, *) | [6327, 8514, ...] |
| (dog, aardvark) | [2,1] |
| (dog, aardwolf) | [1] |
| ... | |
| (dog, zebra) | [2,1,1,1] |
| (doge, *) | [682, ...] |

compute marginal:

$$\sum_{w'} N(\text{dog}, w') = 42908$$

$$f(\text{aardvark}|\text{dog}) = 3/42908$$

$$f(\text{aardwolf}|\text{dog}) = 1/42908$$

$$f(\text{zebra}|\text{dog}) = 5/42908$$

compute marginal:

$$\sum_{w'} N(\text{doge}, w') = 1267$$

“Order Inversion”

- Common design pattern:

- Take advantage of sorted key order at reducer to sequence computations
- Get the marginal counts to arrive at the reducer before the joint counts

- Optimization:

- Apply in-memory combining pattern to accumulate marginal counts

Synchronization: Pairs vs. Stripes

- Approach 1: turn synchronization into an ordering problem
 - Sort keys into correct order of computation
 - Partition key space so that each reducer gets the appropriate set of partial results
 - Hold state in reducer across multiple key-value pairs to perform computation
 - Illustrated by the “pairs” approach
- Approach 2: construct data structures that bring partial results together
 - Each reducer receives all the data it needs to complete the computation
 - Illustrated by the “stripes” approach

The background image shows a massive data center or server room. The space is filled with rows of server racks, each glowing with blue and white lights from within. The ceiling is a complex network of steel beams and pipes, also partially illuminated. The overall atmosphere is one of a high-tech, industrial environment.

Secondary Sorting: “Value-to-Key Conversion” Design Pattern

Secondary Sorting

- MapReduce sorts input to reducers by key
 - Values may be arbitrarily ordered
- What if want to sort value also?
 - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \dots$

Secondary Sorting: Solutions

○ Solution 1:

- Buffer values in memory, then sort
- Why is this a bad idea?

○ Solution 2:

- “Value-to-key conversion” design pattern: form composite intermediate key, (k, v)
- Let execution framework do the sorting
- Preserve state across multiple key-value pairs to handle processing

Anything else we need to do?



RECAP!

Recap: Tools for Synchronization

- Cleverly-constructed data structures
 - Bring data together
- Sort order of intermediate keys
 - Control order in which reducers process keys
- Partitioner
 - Control which reducer processes which keys
- Preserving state in mappers and reducers
 - Capture dependencies across multiple keys and values

Issues and Tradeoffs

- Number of key-value pairs
 - Object creation overhead
 - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
 - De/serialization overhead
- Local aggregation
 - Opportunities to perform local aggregation varies
 - Combiners make a big difference
 - Combiners vs. in-mapper combining
 - RAM vs. disk vs. network