Advanced Cloud Computing

MapReduce Algorithm Design

Wei Wang
CSE@HKUST
Spring 2025

THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

Limited control

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
- which intermediate key a particular reducer is processing

But still we can control

Cleverly-constructed data structures

- bring partial results together

Sort order of intermediate keys

- control order in which reducers process keys

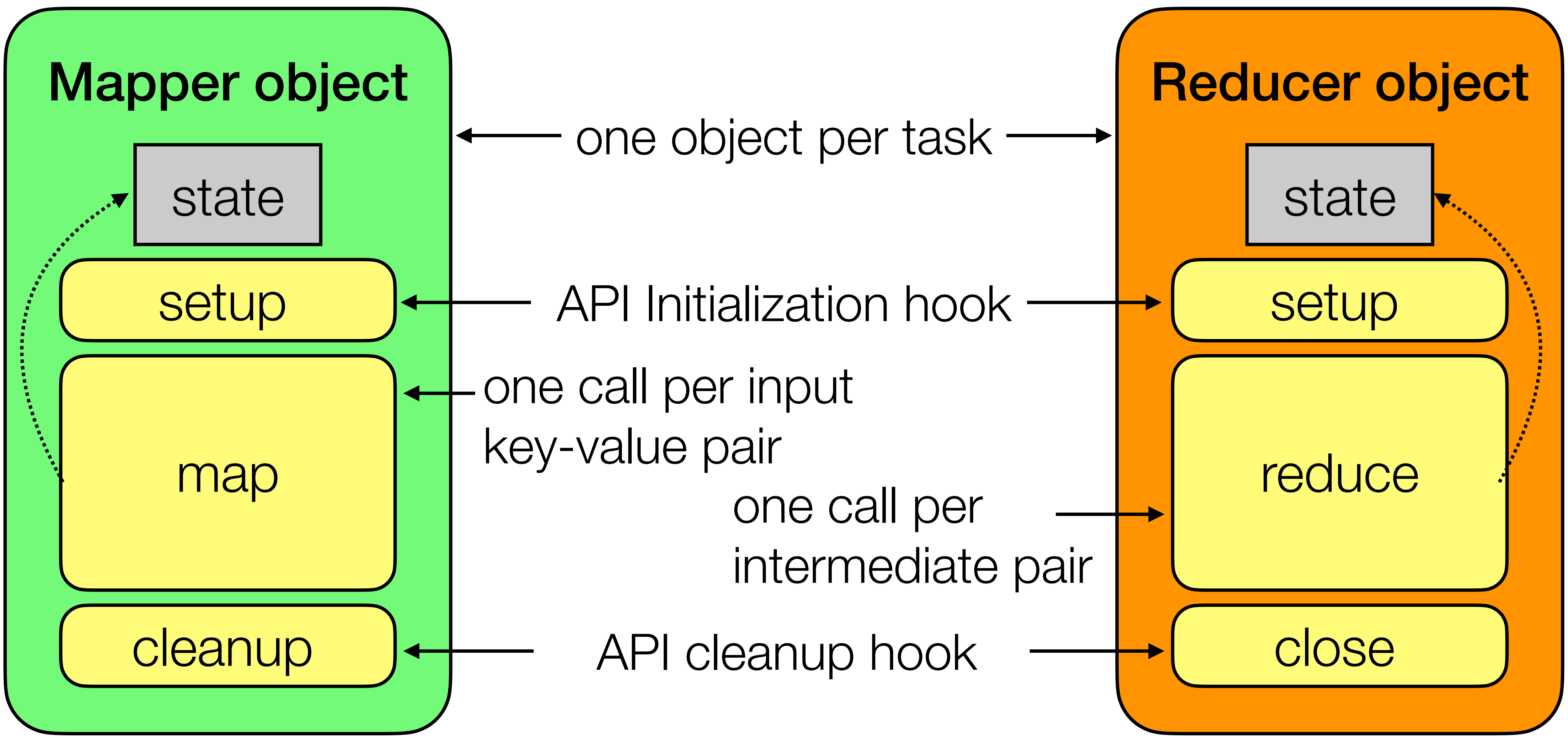
Partitioner

- control which reducer processes which keys

Preserving state in mappers and reducers

- capture dependencies across multiple keys and values

Preserving state



Scalable Hadoop Algorithms

Avoid object creation

- inherently costly operation
- garbage collection

Avoid buffering

- limited heap size
- works for small datasets, but won’t scale!

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, as much as possible!

- reduce intermediate data via local aggregation
- combiners can help

WordCount: Baseline

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

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)
```

What's the impact of combiners?

WordCount: Version 1

$H\{t\}$: a hash table

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

Do combiners still help?

WordCount: Version 2

H{*t*}: a hash table

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

```
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                ▷ Tally counts across documents
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Do combiners still help?

Design pattern for local aggregation

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

- explicit memory management required
- Order matters! May lead to order-dependent bugs!

Combiner design

Combiners and reducers share same method signature

- sometimes, reducers can serve combiners
- often, not…

Combiners are optional optimizations

- should not affect algorithm correctness
- may be run 0, 1, or multiple times (indefinite)

Example: find the mean of integers associated with the same key

Computing the mean: Version 1

class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Reducer
    method Reduce(string t, integers $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string t, integer r_{avg})

Why can’t we use reducer as combiner?

Computing the mean: Version 2

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
class Combiner
    method Combine(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        Emit(string t, pair (sum, cnt))                ▷ Separate sum and count
class Reducer
    method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s
            cnt ← cnt + c
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

Why doesn't this work?

Computing the mean: Version 3

class MAPPER
    method MAP(string t, integer r)
        EMIT(string t, pair $(r, 1)$)

Fixed?

class COMBINER
    method COMBINE(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        EMIT(string t, pair (sum, cnt))

class REDUCER
    method REDUCE(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        EMIT(string t, integer r_{avg})

Computing the mean: Version 4

class MAPPER
  method INITIALIZE
    $S \leftarrow$ new ASSOCIATIVEARRAY
    $C \leftarrow$ new ASSOCIATIVEARRAY
  method MAP(string t, integer r)
    $S\{t\} \leftarrow S\{t\} + r$
    $C\{t\} \leftarrow C\{t\} + 1$
  method CLOSE
    for all term $t \in S$ **do**
      EMIT(term t, pair $(S\{t\}, C\{t\})$)

Are combiners still needed?

Algorithm design: a running example

“Pairs” approach

Each mapper takes a sentence:

- generate all co-occurring term pairs
- for all pairs, emit (a,b) → count

Reducers sum up counts associated with these pairs

Use combiners to minimize shuffling traffics!

“Pairs”: pseudo-code

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) $\triangleright$ 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**
            $s \leftarrow s + c$ $\triangleright$ Sum co-occurrence counts
        Emit(pair p, count s)

“Pairs” analysis

Advantages

- easy to implement, easy to understand

Disadvantages

- lots of pairs to sort and shuffle around
  - What’s the upper bound?
- not many opportunities for combiners to work

“Stripes” approach

Group together pairs into an associative array

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

a → {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 → { b: $count_b$, c: $count_c$, d: $count_d$ … }

"Stripes" approach

Reducers perform

- element-wise sum of associative arrays

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

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

“Stripes”: pseudo-code

```
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)
```

“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
- fundamental limitation in terms of size of event space

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 → {b: 1, c: 2, d: 5, e: 3, f: 2}

Easy!

- one pass to compute (a, *)
- another pass to directly compute f(B|A)

f(B|A): "Pairs"

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
…

What're the issues?

- computing relative frequencies requires marginal counts
- but the marginal cannot be computed until you see all counts

Buffering is a bad idea! (why?)

Solution: what if we could get the marginal count to arrive at the reducer first?

f(B|A): “Pairs”

(a, *) → 20 Reducer holds this value in memory

(a, b) → 1	(a, b) → 1/20
(a, c) → 2	(a, c) → 2/20
(a, d) → 5	(a, d) → 5/20
(a, e) → 3	(a, e) → 3/20
…	…

Emit extra **(a, *)** for every “**b**”, “**c**”, “**d**”, “**e**”… in mapper

Make sure all **a**’s get sent to the same reducer (partitioner)

Make sure **(a, *)** comes first (define sort order)

Hold state in reducer across different key-value pairs

From a reducer's perspective

key	**values**	
(dog, $*$)	[6327, 8514, ...]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark}\vert\text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf}\vert\text{dog}) = 1/42908$
...		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra}\vert\text{dog}) = 5/42908$
(doge, $*$)	[682, ...]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$

What to emit in mapper?

Emit extra **(a, *)** for every **"b"**, **"c"**, **"d"**, **"e"**… in mapper

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(pair (w, “”), count 1)

How to make sure that pair (w, “”) is sorted in order before pair (w, u)?

Write your own data types

PairOfStrings implements **WritableComparable** interface

Must implement

- **write** for serialization
- **readFields** for deserialization
- **compareTo** to define sort order

Sample code available in Assignment-3

Partitioner

Make sure all **a**'s get sent to the same reducer (use partitioner)

```
private static class MyPartitioner extends
      Partitioner<PairOfStrings, IntWritable> {
   @Override
   public int getPartition(PairOfStrings key, IntWritable value,
         int numReduceTasks) {
      return (key.getLeftElement().hashCode() & Integer.MAX_VALUE)
            % numReduceTasks;
   }
}
```

Partition based on the left element only

Reducer

Make sure **(a, *)** comes first (already guaranteed)

Hold state in reducer across different key-value pairs

```
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)
```

Reducer

Make sure **(a, *)** comes first (already guaranteed)

Hold state in reducer across different key-value pairs

```
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
        if p.rightElement == “” then
            marginal ← s
        else
            Emit(pair p, frequency s / marginal)
```

Where should *marginal* be declared and initialized?

You’ll write the complete code in Assignment-3

“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

Pairs vs. Stripes

Pairs

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

Stripes

Construct data structures that bring partial results together

- each reducer receives all data it needs to complete the computation
- be careful about the scalability issue: large stripes overflow the memory (and network)!
  - thus usually requires special treatment

Secondary sorting

Secondary sorting

In Hadoop, MapReduce sorts input to reducers by key

- values may be arbitrarily ordered
  - Google's proprietary implementation supports value sorting

What if we want to sort values as well?

- e.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

Secondary sorting: solutions

Solution 1:

- buffer values in memory, then sort
- Is this a good idea? Why?

Solution 2:

- “value-to-key conversion” design pattern

"Value-to-key conversion"

Form composite intermediate key:

$$k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots \quad \Rightarrow \quad \begin{array}{l} (k, v_1) \rightarrow r \\ (k, v_3) \rightarrow r \\ (k, v_4) \rightarrow r \\ (k, v_8) \rightarrow r \\ \ldots \end{array}$$

Let execution framework do sorting

Preserve state *across* multiple key-value pairs to handle processing

Anything else we need to do?

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

Issues and tradeoffs

Local aggregation

- Opportunities to perform local aggregation varies
- Combiners make a big difference
- Combiners vs. in-mapper combining
- RAM vs. disk vs. network

Debugging at scale

Work on small datasets, won't scale… why?

- memory management issues (buffering and object creation)
- too much intermediate data
- mangled input records

Real-world data is messy!

- there's no such thing as "consistent data"
- watch out for corner cases
- isolate unexpected behavior, bring local

Credits

Slides are adapted from Prof. Jimmy Lin's slides at the University of Waterloo