

MapReduce Design Patterns

MapReduce Restrictions

- ▶ Any algorithm that needs to be implemented using MapReduce must be expressed in terms of a small number of rigidly defined components that must fit together in very specific ways.
- ▶ Synchronization is difficult. Within a single MapReduce job, there is only one opportunity for cluster-wide synchronization—during the shuffle and sort stage.
- ▶ Developer has little control over the following aspects:
 - ▶ Where a mapper or reducer runs (i.e., on which node in the cluster)
 - ▶ When a mapper or reducer begins or finishes
 - ▶ Which input key-value pairs are processed by a specific mapper
 - ▶ Which intermediate key-value pairs are processed by a specific reducer

MapReduce Techniques

- ▶ The ability to construct complex data structures as keys and values to store and communicate partial results.
- ▶ The ability to execute user-specified initialization code at the beginning of a map or reduce task, and the ability to execute user-specified termination code at the end of a map or reduce task.
- ▶ The ability to preserve state in both mappers and reducers across multiple input or intermediate keys.
- ▶ The ability to control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys.
- ▶ The ability to control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer.
- ▶ The ability to iterate over multiple MapReduce jobs using a driver program.

Local Aggregation

We will use the wordcount example to illustrate these techniques.

- ▶ **Use Combiners.** In Hadoop, combiners are considered optional optimizations so they cannot be counted on for correctness or to be even run at all.
- ▶ With the local aggregation technique, we can incorporate combiner functionality directly inside the mappers (under our control) as explained below.
- ▶ **In-Mapper Combining.** An associative array (e.g. Map in Java) is introduced inside the mapper to tally up term counts within a single document: instead of emitting a key-value pair for each term in the document, this version emits a key-value pair for each unique term in the document.

In-Mapper Combining

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

▷ Tally counts for entire document

In-Mapper Combining Across Multiple Documents

- ▶ Prior to processing any input key-value pairs we initialize an associative array for holding term counts in the mapper's initialize method. For example, in Hadoop's new API, there is a `setup(...)` method that is called before processing any key-value pairs.
- ▶ We can continue to accumulate partial term counts in the associative array across multiple documents, and emit key-value pairs only when the mapper has processed all documents.
- ▶ This requires an API hook that provides an opportunity to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split to which the map task was assigned.
- ▶ The Mapper class in the new Hadoop API provides this hook as the method named `cleanup(...)`.

In-Mapper Combining Across Multiple Documents

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

In-Mapper Combining Analysis

- ▶ *Advantages:* In-mapper combining will be more efficient than using Combiners since we have more control over the process and we save having to serialize/deserialize objects multiple times.
- ▶ *Drawbacks.*
 - ▶ State preservation across mappers breaks the MapReduce paradigm. This may lead to ordering dependent bugs that are hard to track.
 - ▶ Scalability bottlenecks if the number of keys we encounter cannot fit in memory. This can be addressed by emitting partial results after every n key-value pairs, or after certain fraction of memory has been used or when a certain amount of memory (buffer) is filled up.

In-Mapper Combiner: Another Example

Suppose we have a large data set where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key.

A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session—the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.

- ▶ Write MapReduce pseudo-code to solve the problem.
- ▶ Modify the solution to use Combiners. Note that

$$\text{Mean}(1, 2, 3, 4, 5) \neq \text{Mean}(\text{Mean}(1, 2), \text{Mean}(3, 4, 5))$$

- ▶ Modify the solution to use in-mapper combining.

Calculating Mean: Basic Solution

```
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  $[r_1, r_2, \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all integer  $r \in$  integers  $[r_1, r_2, \dots]$  do
6:        $sum \leftarrow sum + r$ 
7:        $cnt \leftarrow cnt + 1$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
9:     EMIT(string  $t$ , integer  $r_{avg}$ )
```

Calculating Mean: With Combiners

```
1: class MAPPER
2:   method MAP(string  $t$ , integer  $r$ )
3:     EMIT(string  $t$ , pair ( $r$ , 1))
4:
5: class COMBINER
6:   method COMBINE(string  $t$ , pairs [( $s_1$ ,  $c_1$ ), ( $s_2$ ,  $c_2$ ) ...])
7:      $sum \leftarrow 0$ 
8:      $cnt \leftarrow 0$ 
9:     for all pair ( $s$ ,  $c$ )  $\in$  pairs [( $s_1$ ,  $c_1$ ), ( $s_2$ ,  $c_2$ ) ...] do
10:        $sum \leftarrow sum + s$ 
11:        $cnt \leftarrow cnt + c$ 
12:     EMIT(string  $t$ , pair ( $sum$ ,  $cnt$ ))
```

Calculating Mean: Modified Reducer

```
1: class REDUCER
2:   method REDUCE(string  $t$ , pairs  $[(s_1, c_1), (s_2, c_2) \dots]$ )
3:      $sum \leftarrow 0$ 
4:      $cnt \leftarrow 0$ 
5:     for all pair  $(s, c) \in$  pairs  $[(s_1, c_1), (s_2, c_2) \dots]$  do
6:        $sum \leftarrow sum + s$ 
7:        $cnt \leftarrow cnt + c$ 
8:      $r_{avg} \leftarrow sum / cnt$ 
9:     EMIT(string  $t$ , integer  $r_{avg}$ )
```

Calculating Mean: With In-Mapper Combining

```
1: class MAPPER
2:   method INITIALIZE
3:      $S \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:      $C \leftarrow \text{new ASSOCIATIVEARRAY}$ 
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 ( $S\{t\}$ ,  $C\{t\}$ ))
```

Another example: Unique Items Counting

There is a set of records. Each record has field F and arbitrary number of category labels $G = \{G1, G2, \dots\}$. Count the total number of unique values of field F for each subset of records for each value of any label.

Record 1: $F=1$, $G=\{a, b\}$

Record 2: $F=2$, $G=\{a, d, e\}$

Record 3: $F=1$, $G=\{b\}$

Record 4: $F=3$, $G=\{a, b\}$

Result:

$a \rightarrow 3$ // $F=1, F=2, F=3$

$b \rightarrow 2$ // $F=1, F=3$

$d \rightarrow 1$ // $F=2$

$e \rightarrow 1$ // $F=2$

- ▶ Come up with a two-pass solution.
- ▶ Come up with a one-pass solution that uses combining in the reducer.

Solution (two-pass)

- ▶ At the first stage Mapper emits dummy counters for each pair of F and G. Reducer emits only one output for all duplicate instances of a pair.
- ▶ At the second phase pairs are grouped by G and the total number of items in each group is calculated.

Phase I:

```
class Mapper
```

```
    method Map(null, record [value f, categories [g1, g2,...]])  
        for all category g in [g1, g2,...]  
            Emit(record [g, f], count 1)
```

```
class Reducer
```

```
    method Reduce(record [g, f], counts [n1, n2, ...])  
        Emit(record [g, f], null ) //just one to eliminate duplicates
```

Phase II:

```
class Mapper
```

```
    method Map(record [f, g], null)  
        Emit(value g, count 1)
```

```
class Reducer
```

```
    method Reduce(value g, counts [n1, n2,...])  
        Emit(value g, sum( [n1, n2,...] ) )
```

Cross-Correlation

- ▶ There is a set of tuples of items. For each possible pair of items calculate the number of tuples where these items co-occur. If the total number of items is n , then $n^2 = n \times n$ values should be reported.
- ▶ This problem appears in text analysis (say, items are words and tuples are sentences), market analysis (customers who buy this tend to also buy that). If n^2 is quite small and such a matrix can fit in the memory of a single machine, then implementation is straightforward.
- ▶ We will study two ways to solving this problem that illustrate two patterns: *pairs* versus *stripes*.

Pairs and Stripes Patterns

- ▶ **Pairs pattern.** The mapper finds each co-occurring pair and outputs it with a count of 1. The reducer just adds up the frequencies for each pair. This requires the use of complex keys (a pair of words).
- ▶ **Stripes pattern.**
 - ▶ Instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted H . The mapper emits key-value pairs with words as keys and corresponding associative arrays as values.
 - ▶ The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key.
 - ▶ In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.

Calculating Co-occurrences: With Pairs Pattern

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

1: class REDUCER
2:   method REDUCE(pair  $p$ , counts [ $c_1, c_2, \dots$ ])
3:      $s \leftarrow 0$ 
4:     for all count  $c \in \text{counts } [c_1, c_2, \dots]$  do
5:        $s \leftarrow s + c$                        ▷ Sum co-occurrence counts
6:     EMIT(pair  $p$ , count  $s$ )
```

Calculating Co-occurrences: With Stripes Pattern

```
1: class MAPPER
2:   method MAP(docid  $a$ , doc  $d$ )
3:     for all term  $w \in \text{doc } d$  do
4:        $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
5:       for all term  $u \in \text{NEIGHBORS}(w)$  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:      $H_f \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:     for all stripe  $H \in \text{stripes } [H_1, H_2, H_3, \dots]$  do
5:       SUM( $H_f, H$ ) ▷ Element-wise sum
6:     EMIT(term  $w$ , stripe  $H_f$ )
```

Pairs versus Stripes

- ▶ *Stripes* generates fewer intermediate keys than *Pairs* approach.
- ▶ *Stripes* benefits more from combiners and can be done with in-memory combiners.
- ▶ *Stripes* is, in general, faster.
- ▶ *Stripes* requires more complex implementation.
- ▶ *Pairs* is more scalable without any modifications.

- ▶ Jimmy Lin and Chris Dyer. Chapter 3 in *Data-Intensive Text Processing with MapReduce*.
- ▶ Ilya Katsov. *MapReduce Patterns, Algorithms, and Use Cases*.
<http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/>