

Pragmatic Programming Techniques



The first part of today's presentation is largely stolen from Ricky Ho's Blog posting (But, any mistakes are assuredly mine!):

Sunday, August 29, 2010

Designing algorithms for Map Reduce

15-440/640 July 1, 2014 (Gregory Kesden, Presenting)

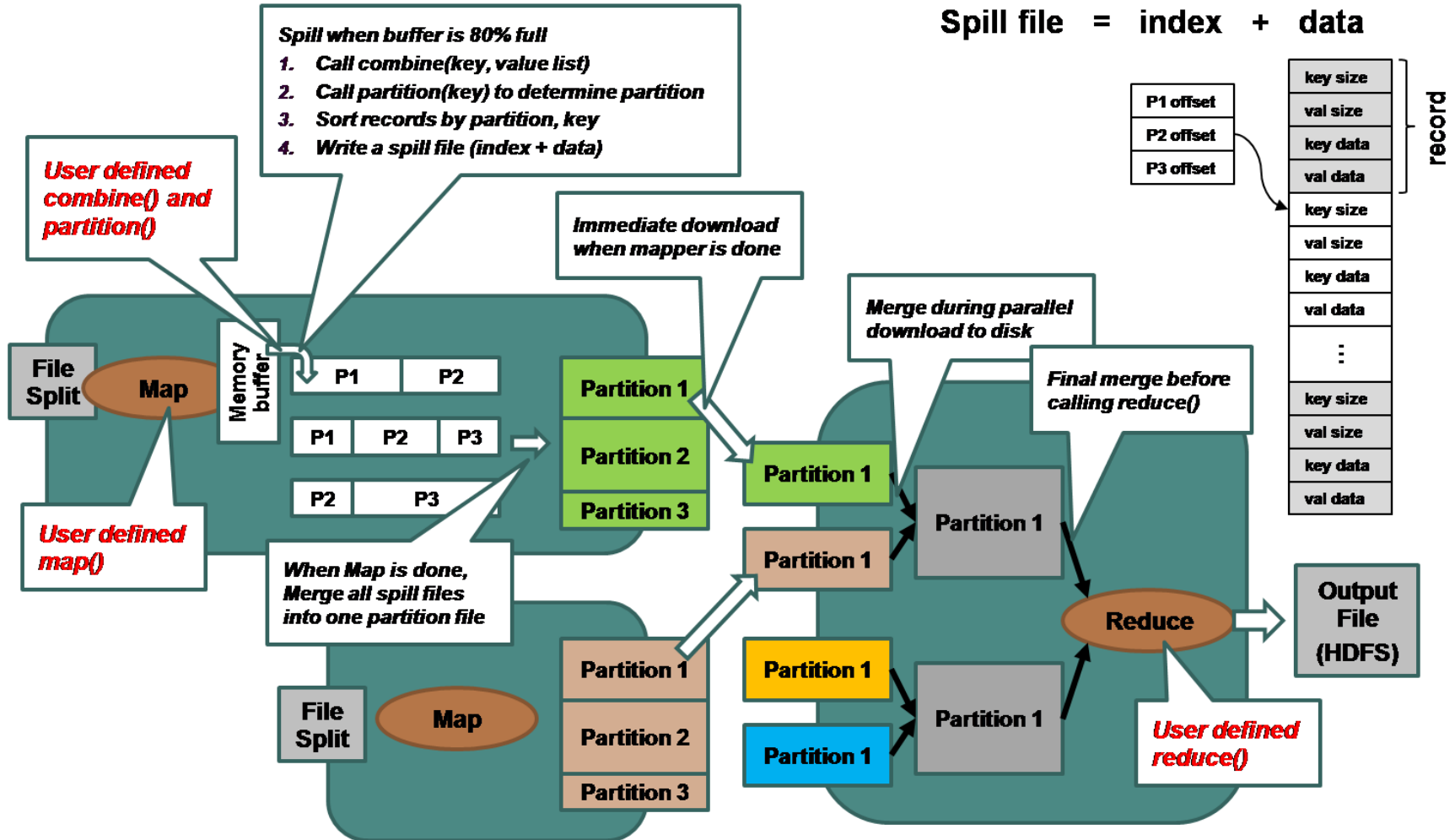


 Ricky Ho

I am a software architect and consultant passionate in Distributed and parallel computing, Machine learning and Data mining, SaaS and Cloud computing.

<http://horicky.blogspot.com/2010/08/designing-algorithmis-for-map-reduce.html>

Map-Reduce Overview



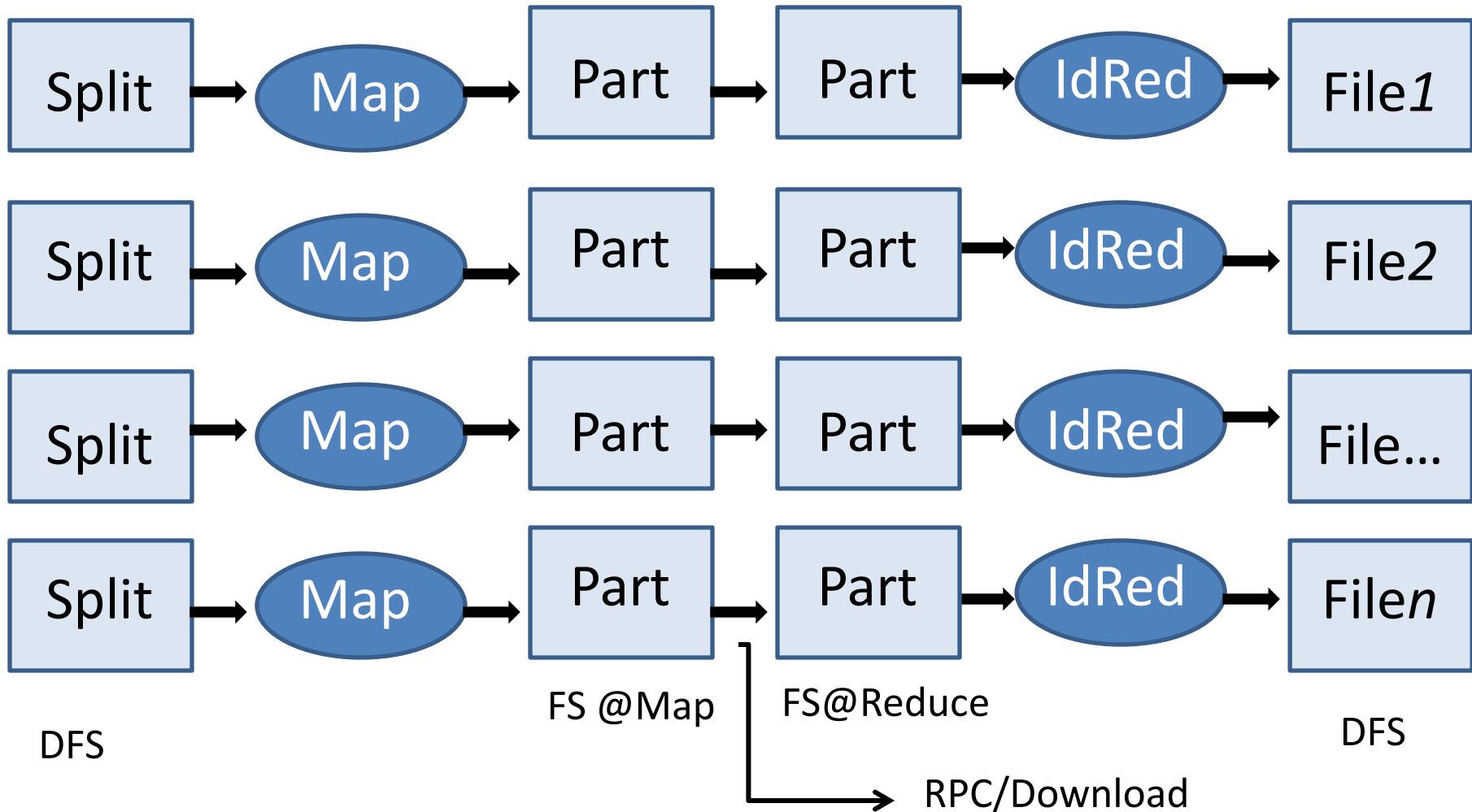
Where's The Hype?

- Parallelism is (mostly) in the maps, which are independent (unsynchronized).
 - $O(n)$ becomes $O(1)$ with parallelism
- Merge sorts are Merge Sorts
 - $O(n \log n)$
- Reduces are linear
 - $O(n)$

“Embarrassingly Parallel”

- Some things are essentially only Maps
 - Identity reduce
 - Massively parallel
 - No bottleneck
- Examples:
 - Filter to retain or exclude only certain patterns
 - Reduce data size by random sampling
 - Convert format, e.g. bold to italics
 - Flag bad data, e.g. negative, out of range, etc.

“Embarrassingly Parallel”



Sorting

- Map-Reduce is, in many ways, a distributed sorting engine that can do some useful work along the way
 - The merge sort and reduce perform the sort
- We can leverage this if we actually want to sort
 - Identity map
 - Identity reduce
- One possibly trick: Partition by range
 - Simplifies merge.

```
partition(key) {  
    range = (KEY_MAX - KEY_MIN) / NUM_OF_REDUCERS  
    reducer_no = (key - KEY_MIN) / range  
    return reducer_no  
}
```

Inverted Indexes

- Common index from key to location, e.g. word to <fileName:line#>
- Map emits key, e.g. <word, fileName:line#>
- Reduce produces <key, list<metadata>>, e.g. <word, list<fileName:line#>>

```
map(key, container) {
  for each element in container {
    element_meta =
      extract_metadata(element, container)
    emit(element, [container_id, element_meta])
  }
}

reduce(element, container_ids) {
  element_stat =
    compute_stat(container_ids)
  emit(element, [element_stat, container_ids])
}
```

Simple Statistics

- Where operation is *both* commutative and associative
- Map does local computation
- Reduce forms global computation
- Examples: Min, max, count, sum (What about average?)

```
class Mapper {
  buffer

  map(key, number) {
    buffer.append(number)
    if (buffer.is_full) {
      max = compute_max(buffer)
      emit(1, max)
    }
  }
}
```

```
class Reducer {
  reduce(key, list_of_local_max) {
    global_max = 0
    for local_max in list_of_local_max {
      if local_max > global_max {
        global_max = local_max
      }
    }
    emit(1, global_max)
  }
}
```

```
class Combiner {
  combine(key, list_of_local_max) {
    local_max = maximum(list_of_local_max)
    emit(1, local_max)
  }
}
```


Histograms

- Divide into different intervals.
- Maps compute the count per interval.
- Reduce will compute the per interval.
- Note power is in map: Ability to classify in parallel

```
class Mapper {
    interval_start = [0, 20, 40, 60, 80]

    map(key, number) {
        i = 0;
        while (i < NO_OF_INTERVALS) {
            if (number < interval_start[i]) {
                emit(i, 1)
                break
            }
        }
    }
}
```

```
class Reducer {
    reduce(interval, counts) {
        total_counts = 0
        for each count in counts
            total_counts += count
        }
        emit(interval, total_cour
    }
}
```

```
class Combiner {
    combine(interval, occurrence) {
        emit(interval, occurrence.size)
    }
}
```

SELECT

- Filter: *result = SELECT c1, c2, c3, c4 FROM source WHERE conditions*
 - Implement in Map
- Aggregation: *SELECT sum(c3) as s1, avg(c4) as s2 ... FROM result GROUP BY c1, c2 HAVING conditions*
 - Implement in Reduce

```
class Mapper {
  map(k, rec) {
    select_fields =
      [rec.c1, rec.c2, rec.c3, rec.c4]
    group_fields =
      [rec.c1, rec.c2]
    if (filter_condition == true) {
      emit(group_fields, select_fields)
    }
  }
}
```

```
class Reducer {
  reduce(group_fields, list_of_rec) {
    s1 = 0
    s2 = 0
    for each rec in list_of_rec {
      s1 += rec.c3
      s2 += rec.c4
    }
    s2 = s2 / rec.size
    if (having_condition == true) {
      emit(group_fields, [s1, s2])
    }
  }
}
```

Simple Join

```
map(k1, rec) {
  emit(rec.key, [rec.type, rec])
}

reduce(k2, list_of_rec) {
  list_of_typeA = []
  list_of_typeB = []
  for each rec in list_of_rec {
    if (rec.type == 'A') {
      list_of_typeA.append(rec)
    } else {
      list_of_typeB.append(rec)
    }
  }

  # Compute the catesian product
  products = []
  for recA in list_of_typeA {
    for recB in list_of_typeB {
      emit(k2, [recA, recB])
    }
  }
}
```

Kesden's Additional Slides

- The next few slides are from me, rather than the cited source for the rest of the presentation.

Normalize Format For Join

- Map records to common Format
- Identity reduce
- Identity Map
- Reduce to form cross-product
- Filter to get results

Shortest Path

- Form Graph as Adjacency List
 - Map: $\langle \text{Node}, \langle \text{Node}, \text{Distance} \rangle \rangle$ for each adjacency
 - Reduce: $\langle \text{Node}, \text{Shortest} \langle \text{Node}, \text{Distance} \rangle \rangle$
- Work from each node in parallel
- Map
 - Node n as a key and $(D, \text{points-to})$ as its value
 - D is the distance form the start
 - Points-to is a list of Nodes reachable from n , initially direct adjacencies
 - Emits all points reachable from n via each node in points-to
- Reduce
 - Emits one Node n for each key, the one with the shortest D
- Repeat Map and Reduce phases until no shorter distances found (nothing learned, nothing can be learned, convergence)