SkewTune

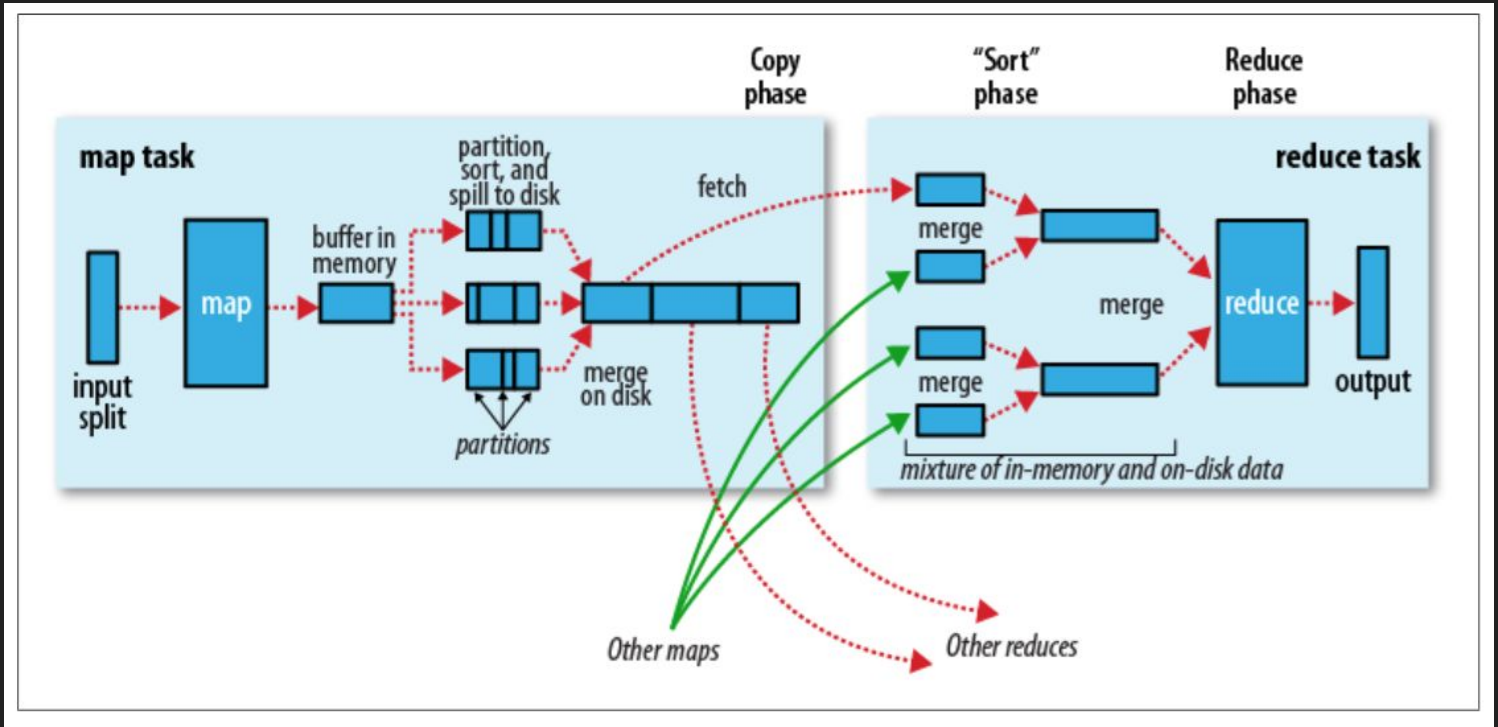
Mitigating Skew in MapReduce Applications

Presented by Hao Tan

What is SkewTune:

SkewTune is an extension to MapReduce system that transparently mitigate skew

Quick review: MapReduce system

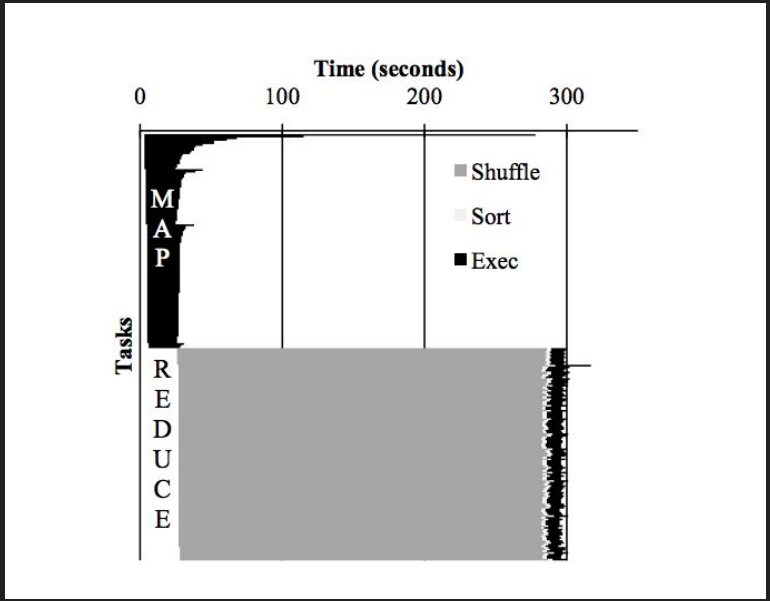


Overview

- Introduction to skew
- Previous approaches
- Design goals
- SkewTune approach
  - Skew detection
  - Skew mitigation
- Conclusion
- Q&A

Skew: highly variable task runtimes

Ideally, every mapper and reducer are expected to get roughly equal amount of workload. However, expectation is always different from the reality:



Types of Skew: Map Phase

- Expensive Record
  - PageRank: vertex with large outlink degree need disproportional amount of time to process
- Heterogeneous Map
  - Various dataset are concatenated
  - Map task performs different transformations based on dataset type

Types of Skew: Reduce Phase

- Partitioning skew
  - Bad hash functions
- Expensive key group
  - Some key groups might take longer time to process

Previous approaches

- Skew-resistant operators
- Dividing work into extremely fine-grained partitions and re-allocating these partitions to machines as needed
- Sampling the output of an operator and plan how to partition it
- Backup jobs
- User defined cost function for partitioning data

Design Goals

- Developer Transparency:
  - No user involvement for skew mitigation
- Mitigation Transparency:
  - Outputs obtained with/without SkewTune mitigation should remain the same
- Maximal Applicability:
  - Can be applied to all MapReduce applications
- No Synchronization Barriers:
  - Does not block a operator before it finishes its current job

Architecture

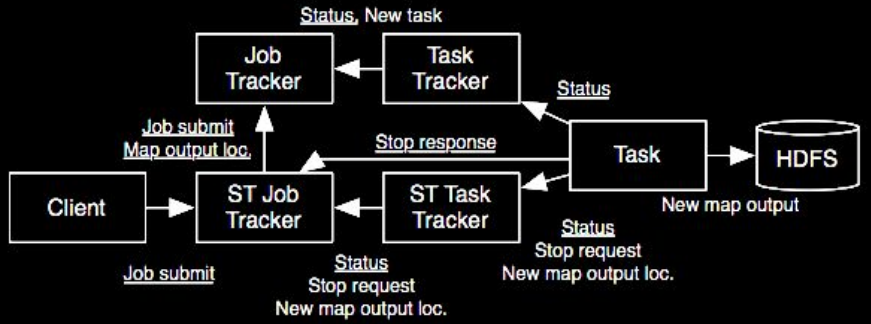


Figure 5: SkewTune Architecture. Each arrow is from sender to receiver. Messages related to mitigation are shown. Requests are underlined. Mitigator jobs are created and submitted to the job tracker by the SkewTune job tracker. Status is the progress report.

Skew Detection

Answers two questions:

- When to detect and mitigate skew
- Which task should be mitigated

When to detect: Late Skew Detection

- SkewTune delays any skew mitigation decisions until a slot becomes available.
- Cluster will have high utilization as long as each slot is running some tasks
- Reduce the opportunities of false positive.
- Avoid false negative, cluster utilization is maintained at the highest level

Which task should be mitigated

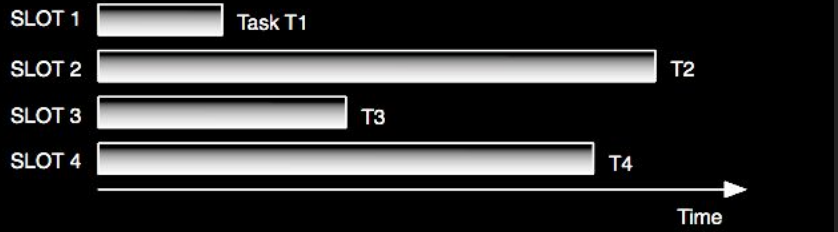
- Only one task will be labeled as the straggler
- Pick the one with the greatest estimated remaining time.
- Flag skew when:

$$\frac{T_{remain}}{2} > \omega\ (repartitioning\ overhead)$$

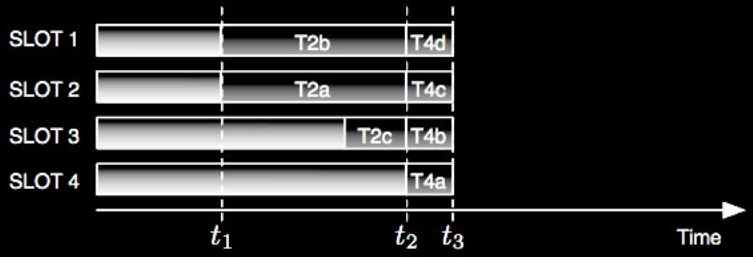
Intuition: remaining workload must be handled by at least 2 mitigators , therefore the performance gain is:

$$\frac{T_{remain}}{2} - \omega$$

High-level Concept:



(a) Without SkewTune, operator runtime is that of the slowest task.



(b) With SkewTune, the system detects available resources as task T1 completes at t_1. SkewTune identifies task T2 as the straggler and re-partitions its unprocessed input data. SkewTune repeats the process until all tasks complete.

Pseudo code: skew detection

Algorithm 1 GetNextTask()

Input: $\mathcal{R}$: set of running tasks
$\mathcal{W}$: set of unscheduled waiting tasks
inProgress: global flag indicating mitigation in progress
Output: a task to schedule

```
1:  task ← null
2:  if W ≠ ∅ then
3:      task ← chooseNextTask(W)
4:  else if ¬inProgress then
5:      task ← argmax_{task∈R} time_remain(task)
6:      if task ≠ null ∧ time_remain(task) > 2 · ω then
7:          stopAndMitigate(task) /* asynchronous */
8:          task ← null
9:          inProgress ← true
10:     end if
11: end if
12: return task
```

Skew Mitigation

1. Stopping the straggler
2. Partitioning unprocessed input data
3. Generating plan for mitigation

Step 1: Stop the straggler

- Coordinator sends the stop request to the straggler
- Straggler captures the position of its last processed record in its input
- If the straggler is difficult to stop:
  - Coordinator find another straggler
  - If straggler is the last task in the job, repartition its entire input and reprocess

Step 2: Partitioning remaining input data

- For achieving mitigation transparency, unprocessed data will be range-partitioned
- A range for a map task is a fragment of file
- A range for a reduce task is an interval of reduce keys
- Two approaches
  - Local scan
  - Parallel scan

Local Scan VS Parallel Scan

- Local scan is preferred when the size of unprocessed data is small
- Parallel scan is preferred when the size of unprocessed data is large
- To choose between parallel scan and local scan, simply check:

$$\frac{\Delta}{\beta} > \frac{\max\{\sum_{o \in O_n} o.bytes \mid n \in \mathcal{N}\}}{\beta} + \rho$$

- For parallel scan, multiple tasks is created to scan the map outputs that straggler's input is made up of. Its runtime is determined by the slowest task.

Pseudo code:

Algorithm 2 GenerateIntervals()

Input: I: Sorted stream of intervals
b: Initial bytes-per-interval. Set to s for local scan.
s: Target bytes-per-interval.
k: Minimum number of intervals.
Output: list of intervals

```
result ← [] /* resulting intervals */
cur ← new_interval() /* current interval */
for all i ∈ I do
  if i.bytes > b ∨ cur.bytes ≥ b then
    if b < s then
      result.appendIfNotEmpty(cur)
      if |result| ≥ 2 × k then
        /* accumulated enough intervals. increase b. */
        b ← min{2 × b, s}
        /* recursively recompute buffered intervals */
        result ← GenerateIntervals(result, b, b, k)
      end if
    else
      result.appendIfNotEmpty(cur)
    end if
    cur ← i /* open a new interval */
  else
    cur.updateStat(i) /* aggregate statistics */
    cur.end ← i.end
  end if
end for
result.appendIfNotEmpty(cur)
return result
```

- For local scan, interval size is fixed.
- It is set to be:

$$s = \left\lfloor \frac{\Delta}{k \cdot |\mathcal{S}|} \right\rfloor$$

Wide interval can be a problem

begin	values	end
$k_3 : 4$	9	$k_7 : 3$
$k_7 : 1$	10	$k_{100} : 2$
$k_{50} : 2$	14	$k_{95} : 5$

Intervals from Parallel Local Scans.

$\Rightarrow$

Range	Est. values
$[k_3, k_3]$	4
(k_3, k_7)	9
$[k_7, k_7]$	4
(k_7, k_{50})	$0 + 10/5$
$[k_{50}, k_{50}]$	$2 + 10/5$
(k_{50}, k_{95})	$14 + 10/5$
$[k_{95}, k_{95}]$	$5 + 10/5$
(k_{95}, k_{100})	$0 + 10/5$
$[k_{100}, k_{100}]$	2

Aligned ranges and estimated # of values.

- Intervals generated by different tasks can overlap with each other.
- Coordinator will break intervals into non-overlapping segments and estimate their sizes
- Wide interval will introduce uncertainties to the estimation.

Solution:

Algorithm 2 GenerateIntervals()

Input: I: Sorted stream of intervals
b: Initial bytes-per-interval. Set to s for local scan.
s: Target bytes-per-interval.
k: Minimum number of intervals.
Output: list of intervals

```
1: result ← [] /* resulting intervals */
2: cur ← new_interval() /* current interval */
3: for all i ∈ I do
4:     if i.bytes > b ∨ cur.bytes ≥ b then
5:         if b < s then
6:             result.appendIfNotEmpty(cur)
7:             if |result| ≥ 2 × k then
8:                 /* accumulated enough intervals. increase b. */
9:                 b ← min{2 × b, s}
10:                /* recursively recompute buffered intervals */
11:                result ← GenerateIntervals(result, b, b, k)
12:            end if
13:        else
14:            result.appendIfNotEmpty(cur)
15:        end if
16:        cur ← i /* open a new interval */
17:    else
18:        cur.updateStat(i) /* aggregate statistics */
19:        cur.end ← i.end
20:    end if
21: end for
22: result.appendIfNotEmpty(cur)
23: return result
```

- Setting the upper bound of interval size to be

$$s = \left\lfloor \frac{\Delta}{k \cdot \max\{|\mathcal{S}|, |\mathcal{O}|\}} \right\rfloor$$

- Start scanning will a small interval size (4KB) and adaptively incrementing the interval size when there are more unprocessed data

Step 3: Generating plan for mitigation

Algorithm 3 LinearGreedyPlan()

Input: I: a sorted array of intervals
T: a sorted array of t_{remain} for all slots in the cluster
θ: time remaining estimator
ω: repartitioning overhead
ρ: task scheduling overhead
Output: list of intervals

```
   /* Phase 1: find optimal completion time opt. */
1: opt ← 0; n ← 0 /* n: # of slots that yield optimal time */
2: W ← θ(R) /* remaining work+work running in n nodes */
3: /* use increasingly many slots to do the remaining work */
4: while n < |T| ∧ opt ≥ T[n] do
5:    opt' ← (W+T[n]+ρ)/(n+1) /* optimal time using n + 1 slots */
6:    if opt' − T[n] < 2 · ω then
7:       break /* assigned too little work to the last slot */
8:    end if
9:    opt ← opt'; W ← W + T[n] + ρ; n ← n + 1
10: end while
   /* Phase 2: greedily assign intervals to the slots. */
11: P ← [] /* intervals assigned to slots */
12: end ← 0 /* index of interval to consider */
13: while end < |I| do
14:    begin ← end; remain ← opt − T[|P|] − ρ
15:    while remain > 0 do
16:       t_est ← θ(I[end]) /* estimated proc. time of interval */
17:       if remain < 0.5 · t_est then
18:          break /* assign to the next slot */
19:       end if
20:       end ← end + 1; remain ← remain − t_est
21:    end while
22:    if begin = end then
23:       end ← end + 1 /* assign a single interval */
24:    end if
25:    P.append(new_interval(I[begin], I[end − 1]))
26: end while
27: return P
```

- Calculate the optimal runtime when remaining workload is perfectly splitted among mitigators.
- Keep incrementing number of mitigators until too little work is assigned (less than 2 * w)
- Greedily assign intervals to each slot to make runtime as close to optimal runtime as possible

Conclusion

- SkewTune presents an elegant solutions for mitigating two common types of skew
  - Uneven distribution of data to operators
  - Some subset of data taking longer time to process
- It minimizes user involvement for skew mitigation while providing significant performance improvement.
- It’s general-purposed and can be applied to all MapReduce applications

Discussion & Questions

Q: What are the limitations of SkewTune system?

Thank you!