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 an operator before it finishes its current job
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:

\[ \frac{T_{\text{remain}}}{2} > \omega \text{ (repartitioning overhead)} \]

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

\[ \frac{T_{\text{remain}}}{2} - \omega \]
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: \( R \): set of running tasks
\( W \): set of unscheduled waiting tasks
\( \text{inProgress} \): global flag indicating mitigation in progress

Output: a task to schedule

1: task \leftarrow \text{null}
2: if \( W \neq \emptyset \) then
3: \hspace{1em} task \leftarrow \text{chooseNextTask}(W)
4: else if \( \neg \text{inProgress} \) then
5: \hspace{1em} task \leftarrow \arg \max_{\text{task} \in R} \text{time_remain(\text{task})}
6: if \( \text{task} \neq \text{null} \wedge \text{time_remain(\text{task})} > 2 \cdot \omega \) then
7: \hspace{2em} \text{stopAndMitigate(\text{task}) /* asynchronous */}
8: \hspace{2em} task \leftarrow \text{null}
9: \hspace{2em} \text{inProgress} \leftarrow \text{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:

\[
\frac{\Delta}{\beta} > \frac{\max\{\sum_{o \in O_n} o.b\text{ytes} \mid n \in \mathcal{N}\}}{\beta} + \rho
\]

- 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:

For local scan, interval size is fixed.

It is set to be:

\[ s = \left\lfloor \frac{\Delta}{k \cdot |S|} \right\rfloor \]
Wide interval can be a problem

- 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:

- Setting the upper bound of interval size to be
  \[ s = \frac{\Delta}{k \cdot \max\{|S|, |O|\}} \]

- 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

- Calculate the optimal runtime when remaining workload is perfectly split 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

Algorithm 3 LinearGreedyPlan()

Input: I: a sorted array of intervals
T: a sorted array of $t_{remain}$ for all slots in the cluster
$\theta$: time remaining estimator
$\omega$: repartitioning overhead
$\rho$: task scheduling overhead

Output: list of intervals

/* Phase 1: find optimal completion time $opt$ */
$opt \leftarrow 0$; $n \leftarrow 0$ /* $n$: # of slots that yield optimal time */
$W \leftarrow \theta(R)$ /* remaining work + work running in $n$ nodes */
/* use increasingly many slots to do the remaining work */
while $n < |T|$ \& $opt \geq T[n]$ do
  $opt' \leftarrow W + T[n] + \omega$ /* optimal time using $n+1$ slots */
  if $opt' - T[n] < 2 \cdot \omega$ then/* assigned too little work to the last slot */
    break
  end if
  $opt \leftarrow opt'$; $W \leftarrow W + T[n] + \rho$; $n \leftarrow n + 1$
end while

/* Phase 2: greedily assign intervals to the slots */
P \leftarrow [] /* intervals assigned to slots */
end \leftarrow 0 /* index of interval to consider */
while end < |I| do
  begin \leftarrow end; remain \leftarrow opt - $T[|P|] - \rho$
  while remain > 0 do
    $t_{test} \leftarrow \theta(I[end])$ /* estimated proc. time of interval */
    if remain < $0.5 \cdot t_{test}$ then
      break /* assign to the next slot */
    end if
    if begin = end then/* assign a single interval */
      end \leftarrow end + 1
    else
      $P.append(new$ interval$([begin], I[end - 1]))$
    end if
  end while
end if

end while
P: return $P$
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!