Empirical Study of Stragglers in Spark SQL and Spark Streaming

Danish Khan, Kshiteej Mahajan, Rahul Godha, Yuvraj Patel

December 19, 2015

1 Introduction

Spark is an in-memory parallel processing framework. A parallel computation is only as fast as the slowest compute-unit, hence detection and mitigation of stragglers is an important problem to address. We present evaluation of Spark SQL and Spark Streaming (hereon referred to as Spark) with regards to stragglers. First, we investigate the presence of stragglers in Spark (without straggler mitigation) to establish a baseline. Second, we evaluate Spark's speculative straggler mitigation technique for various parameter sets. Third, we evaluate effectiveness of Spark's straggler mitigation technique in the presence of induced CPU, disk, and network throttling workloads.

2 Background

2.1 Spark Scheduling

The basic compute-unit in Spark is a Task. A Stage is a logical grouping of tasks, and a Job consists of multiple stages. Each job is submitted to Spark Scheduler. The default scheduling policy in Spark scheduler is First-In-First-Out (FIFO) where each job and its associated stages are assigned resources in the order of arrival. Spark scheduler can optionally be scheduled in a FAIR manner, where jobs are scheduled in a round-robin fashion to minimize job starvation. The scheduling policy for each pool is configurable, as First-In-First-Out (FIFO) or FAIR.

Spark also has a provision for pools. Herein, each pool can be assigned threads and the corresponding jobs associated with it. Each pool can be assigned a different scheduling policy and offers a way for grouping set of tasks with different priority requirements.

For Spark SQL, each query is associated with a single job. For single-query workloads, as we do in our work, the Spark scheduling policy does not play a major role.

For Spark Streaming, each batch interval a new job is spawned. The natural priority in these jobs is the order of arrival and we use the default FIFO Spark scheduler policy.

2.2 Stragglers

As discussed above, once a Spark job is submitted, it gets broken down into multiple stages and each stage spawns many tasks which will be executed on each node. One or more stage might be dependent on the previous stages. So, when a task of the earlier stage takes a longer time to complete, then the dependent stages execution in the pipeline also gets delayed. Such tasks that takes longer time to complete are termed as *straggler*. As the size of the cluster and the size of the stages/tasks grows, the impact of stragglers increases dramatically impacting

the job completion time. Thus, addressing the problem of straggler becomes prudent in order to speed up the job completion time and also improve the cluster efficiency.

Ganesh et al. [1] identified three different categories of root causes for the stragglers. First, machine characteristics such as disk failures, CPU scheduling, memory availability (garbage collection) play an important role in the performance of the tasks. Reads/writes can be impacted in case of the disk behaving bad or disk failures happen. Unnecessary process/threads can lead to a contention of the CPU. Too much memory pressure can lead to tasks failures. Second, network characteristics can also lead to straggler due to congestion, packet drops or other such network faults. Third, the internals of the execution environment (here Spark) also leads to stragglers as the data-work partitioning might not happen optimally or the task scheduling is not proper.

2.3 Straggler Mitigation in Spark

The Straggler Mitigation strategy in Spark is based on Speculation. A task is identified as a Straggler (or Speculatable) and is duplicated. The Straggler identification strategy starts when `spark.speculation.quantile` fraction of tasks in a stage have completed. By default, `spark.speculation.quantile` is set as 0.75. Thereafter, if a currently running task exceeds `spark.speculation.multiplier` times the median task time from the set of successful task completions of this particular stage, then this task is identified as a straggler. The default value of `spark.speculation.multiplier` is 1.5.

2.3.1 Design Issues

A very immediate issue with this strategy is that the median is chosen from the set of successful tasks of the stage and not from the first 0.75 fraction of tasks to complete. So as the stage progresses beyond the 0.75 fraction, each subsequent speculation check would yield a higher time value for the median and thus a much larger relaxation to the classification of a task as a straggler. This is unsound logic, as it is possible that the value of running time that is not being classified as a straggler due to the progressed value of the median time now, may have been classified as a straggler before. This can lead to false-negatives for tasks that have started later in the stage and have a much relaxed speculation check.

This can be easily fixed in code to a more sound logic, where the speculation barrier is fixed to 1.5 times the median value from the 0.75 fraction of the tasks.

3 System Setup

We run our experiments on a cluster of 4 Virtual Machines (VM), each having 5 cores of 2.2 GHz. The main memory on each VM is 24 GB and total disk storage available is 400 GB. The VM's are connected to each other via 10 Gigabit Network. (Although *iperf* tests yielded us no higher than 2 Gbps throughput).

The underlying VM's have Ubuntu 14.04.2 LTS installed. We have installed Spark 1.5.0 on all the 4 VM's and use HDFS (Hadoop 2.6) as the distributed file storage. Spark SQL and Spark Streaming are prepackaged in the distributable version of Spark 1.5.0-Hadoop 2.6.

3.1 Spark SQL

For Spark SQL, we use TPC-DS queries. We choose 48 SQL queries to run all the experiments so as to get good coverage on the nature of operations and an exhaustive understanding of

Spark Context Properties	Value
Driver memory	1 GB
Executor memory	21000M
Executor cores	4
Per task number of CPUs	1
Shuffle memory fraction	0.4
Storage memory fraction	0.4

Table 1: Spark Context Parameters for SQL Queries

query behavior with regards to Stragglers. The Queries we choose are 3, 7, 12-13, 15, 17-18, 20-22, 25-29, 31-32, 34, 40, 42-43, 46, 48, 50-52, 58, 64, 66, 75-76, 79-80, 82, 84-85, 87-98.

We have generated the data needed for Spark SQL using the TPC-DS data generation script and the total data size is 10 GB.

We picked the SQL files for the chosen queries and generated a python equivalent script that can be submitted as a parameter to spark-submit that is available with Spark. We wrote a script to auto-generate the python scripts by reading the SQL files. The Spark Context parameters used for running the queries is given in Table 1.

3.2 Spark Streaming

For Spark Streaming we use the HiBench [2] benchmark. HiBench uses the KafkaInputStream-Reader interface offered by Spark Streaming to stream data from HDFS via Kafka's producer-consumer model. For this, we initialize topics in Kafka that are listened to by HiBench Spark Streaming benchmarks. We also generate seed data for the corresponding workloads using scripts HiBench provides. The workload data generation happens during experiment runtime and is generated by a process of randomization from the seed data.

3.3 Fault Injection Framework

In order to induce resource throttling effects while the jobs run, we have written a separate framework that injects faults. Our basic fault injection framework introduces faults on a random node for a configurable fixed time either of CPU, Disk, Memory, or Network resource. This, in a way, is simulating real world faults. Along the lines of causes of stragglers as discussed in Mantri [1], we intend to introduce the following faults to simulate the real world scenario:

1. *CPU Contention / Compute Bottleneck*
   This fault will spawn processes that will run for a specified amount of time just spinning a busy while loop. Thus, we simulate a scenario where Spark jobs will compete with this compute-hungry bottleneck process that tries to occupy maximum compute resource possible on a node, thereby ending up as a straggler.

   In our experiments, we introduce these faults for 5 seconds when the job is 50% done.

2. *Slow Disk / Faulty Disk*
   This fault will spawn process that will run for a specified amount of time issuing writes on the disk. Thus, we simulate a scenario where due to bad disk or a slow disk, the I/O's are getting impacted and hence the tasks doing disk I/O's are impacted.

   Majorly, in Spark the heavy utilization of disks happens in the initial stages when the data is read. So, we introduced the simulated disk fault at the start of the job and ran the fault for 20 seconds.

3. *Main Memory Availability*
   This fault will create a RAMdisk of a specified size and try to fill in the entire space so that other processes dont get enough main memory for their usage. Using this fault, we try to simulate a scenario where there is lot of data present and Garbage Collector is trying to run and reclaim some free space back. We are not using this fault injection technique for our experiments as saw a lot of failures due to Out of Memory condition on a 50GB data size and had to restart Spark again. So we decided to skip this fault injection option and decided to have just 10GB data size so that the Out of Memory condition is seen sporadically.

4. *Network Fault*
   We introduce network faults between two randomly chosen nodes by starting an *iperf* UDP server on one of the randomly chosen node and a network throttling *iperf* UDP client on the other for a fixed time of 15 seconds.

Generally, injecting faults should be deterministic in nature such that for multiple runs, we should be able to inject the same fault at a given instance on the query run. Our fault injection framework ensures this determinism as we can specify at what point of time of the query run, the fault has to be introduced.

4 Experiments & Results

In this section, we describe the experiments done with Spark SQL and Spark Streaming. We have run the SQL queries 3 times to understand if there is any significant difference in the run times. Similarly, for Spark Streaming, we have done the runs thrice.

While each experiment is going on, we collect the cpu stats, memory stats, disk stats every second. The network & disk stats are collected by noting the difference in the appropriate /proc/net/dev & /proc/diskstats counters at the start and end of each run. We clear the buffer cache to avoid uncontrolled memory involvement or memory optimization.

4.1 Spark SQL

4.1.1 Baseline

We start the experiments by running the queries mentioned above using spark-submit and then extract the query completion time from the logs. We haven't changed any other parameter than mentioned above in Table 1 while running the scripts. As speculation is off, no tasks are detected as speculatable. As there is no means to detect stragglers, we cannot identify any existing stragglers and so we assume that there is no straggler present in the system.

4.1.2 With Speculation

In order to figure out if there were any stragglers in the base runs, we turn on the speculative parameter to true and again run the queries. This time, we observe that quite a few tasks are identified as stragglers. Particularly, we look for pattern
"15/12/12 20:59:02 INFO TaskSetManager: Marking task 8 in stage 0.0 (on 10.0.1.75) as speculatable because it ran more than 33987 ms" to identify the stragglers.

By default, the speculation parameters `speculation.multiplier` and `speculation.quantile` have a value of 1.5 and 0.75 respectively. This means that after 75% of the tasks are completed, then the median of the task completion time is used to identify the stragglers. If a task is slower than the median by 1.5 times, then it should be considered for speculation.

speculation.quantile	speculation.multiplier
0.5	1.5
0.5	1.25
0.5	1.0
0.75	1.25
0.75	1.0
0.75	1.5 (default)

Table 2: Speculation parameters sets used to run experiments

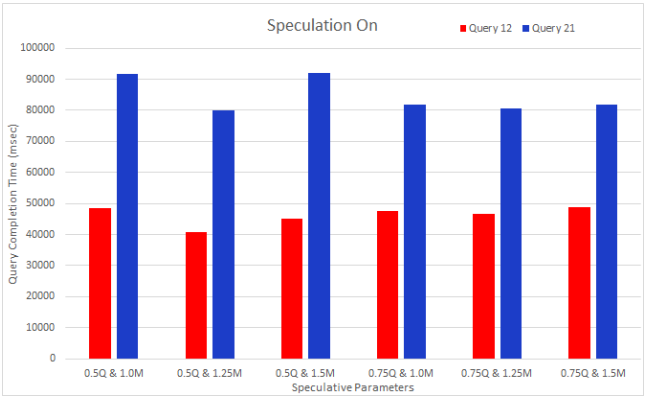


Figure 1: Job completion time for varied speculative parameters.

In order to understand the entropy of the system around these two parameters, we also run the experiments by these two parameters. We run 6 experiments(including the default) having the parameters mentioned in Table 2.

Figure 1 shows the Query 12's and Query 21's completion time for the varied parameters. We observe that the job completion time is the highest when speculation.quantile = 0.5 & speculation.multiplier = 1.0. The reason for this seems to be 2very aggressive speculation as the quantile is just 0.5, once 50% tasks completes, the speculation gets kicked off immediately if the tasks runtime is just about 1 times that of the median runs. Due to the aggressive speculation, the performance of the job completion time is impacted. Also, we see a variation in the completion time of the Queries when speculation.quantile = 0.5 compared to when its 0.75. The reason for this is that as the total number of tasks of which the median is calculated is less, the accuracy of the task completion time is lesser and hence the variation is observed.

4.1.3 Fault injection with Speculation

In order to understand how the stragglers impact the job completion time and how does its presence impact the other dependent jobs, we run the queries and introduce faults by using the fault injection framework described in the previous section.

In figure 2 & 3, we show the time lapse of the Query 21 and Query 12 respectively, showing the stage-wise breakup of the task distribution of each stage over time. This graph clearly shows how certain stages are dependent on the other stages completion time and how a straggler's presence in the earlier stages can impact the overall query completion time. The task distribution of stages increases and the stage completion time also increases in case of presence of straggler.

We clearly show that during the presence of the faults, the stage completion time is longer compared to the baseline runs and the speculative turned on case. It can be seen in Fig 2 for stage 0, after speculation turned on the time lapse of Stage is increased (as seen in Red).

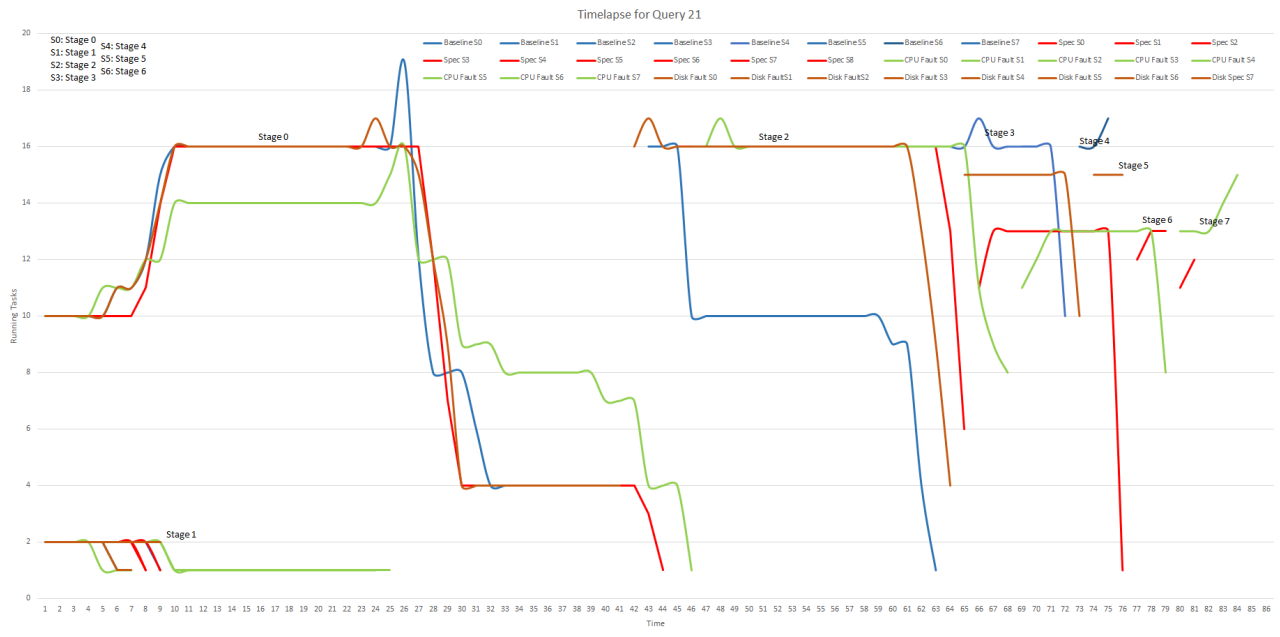


Figure 2: Time lapse for Query 21 with stage level task distribution.

Moreover with injected CPU faults the number of tasks are also increased as the stage progress. As seen in Stage 3 the number of tasks is much higher than baseline. The reason for such a longer stage completion time is that due to the presence of faults, the tasks scheduled on a particular VM cannot make enough forward progress and hence end up being detected as speculatable tasks. Therefore as the stage completion slows down, there is a cascading effect on the stages that are dependent on the slower stage.

Moreover, we also conduct experiments where we vary `speculation.multiplier` and `speculation.quantile` parameter and inject faults. This exercise is done to understand the impact of how each fault impacts the performance of the job completion time. This exercise will also help us understand if the existing straggler mitigation technique is good enough to handle the stragglers or not.

Figure 4a & 4b shows the query completion time for Query 12 and Query 21. We here notice that the disk faults are impacting the query completion time more than the cpu faults. With speculation.quantile = 0.5 & speculation.multiplier = 1.0, Query 12 performed very badly when CPU faults were injected. All the other experiments almost showed the same results in the query completion time when cpu faults were injected. Thus, we can conclude that the workload is disk intensive compared to CPU intensive as the faults are impacting the query completion time.

We even try to analyze how the speculation detection strategy is for spark and does it really make sense to speculate the tasks to that extent. Figure 5a & 5b shows the total number of tasks that are detected as speculatable, how many tasks of the total speculatable tasks are cloned for duplicate execution and how many cloned tasks are ignored as the original tasks completed earlier. As seen from the figures, we see that the number of tasks detected as speculatable drastically increases in case of presence of faults. Another interesting pattern we see is that for almost all the stages, the speculatable tasks are detected when faults are present.

We observe that a very small fraction of the detected speculatable tasks are cloned showing that the speculatable detection strategy is not efficient and it unnecessarily detects a lot of tasks as speculatable. By the time, the task scheduler decides to clone a tasks, majority of the time, the original tasks gets over and hence there arise no need to clone a task. Thus, from the results, it is clearly visible that there is enough improvement needed for the speculatable

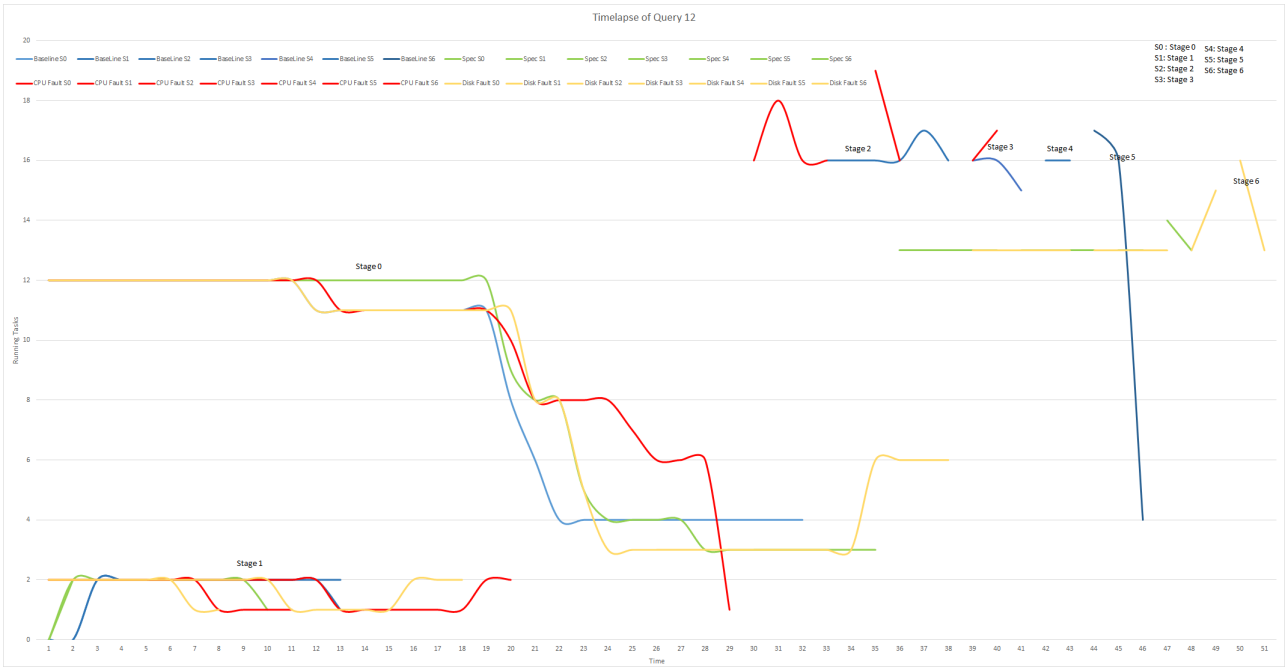
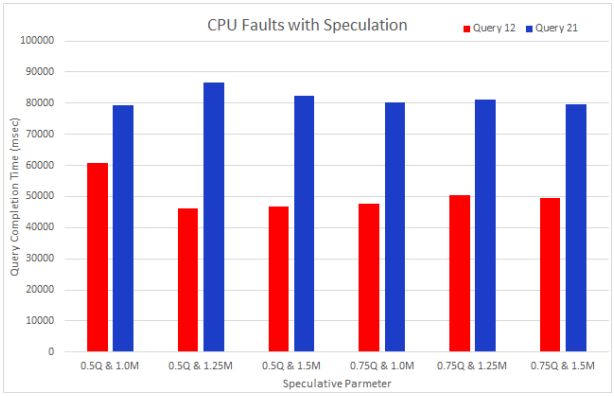
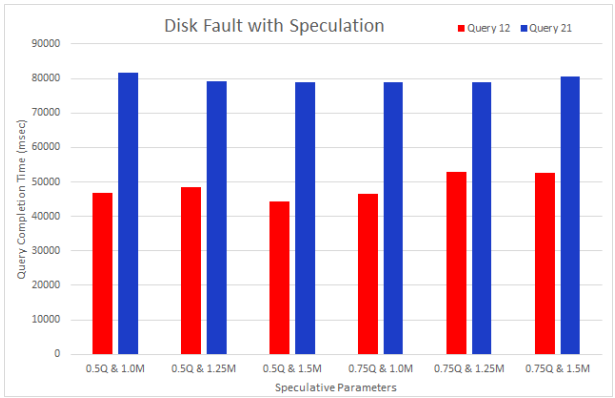


Figure 3: Time lapse for Query 12 with stage level task distribution.

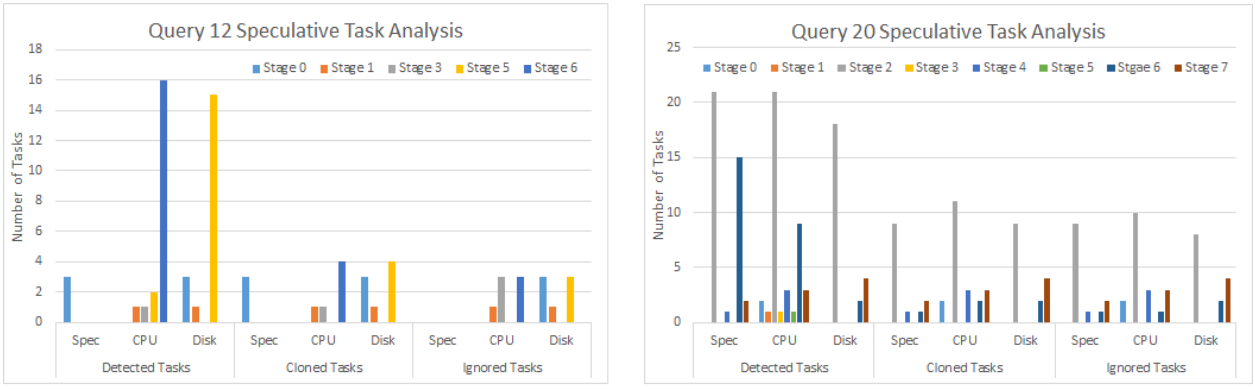


(a) CPU faults



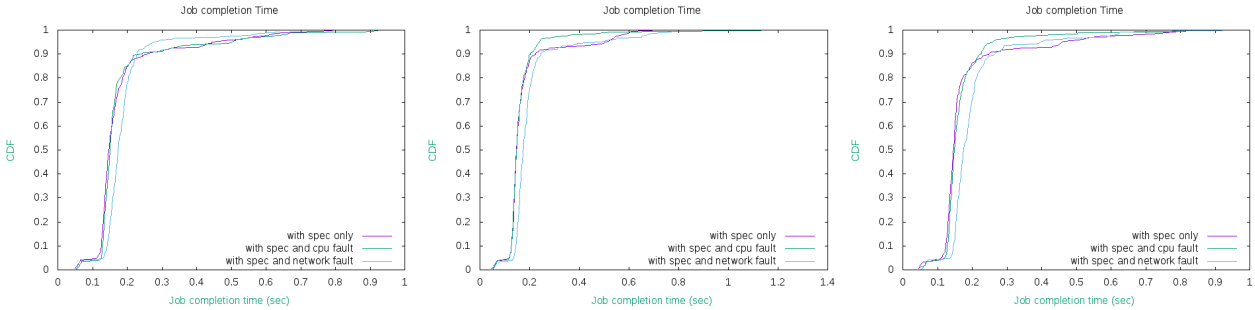
(b) Disk faults

Figure 4: Varied Speculative Parameters behavior during faults



(a) CPU faults (b) Disk faults

Figure 5: Speculation strategy performance

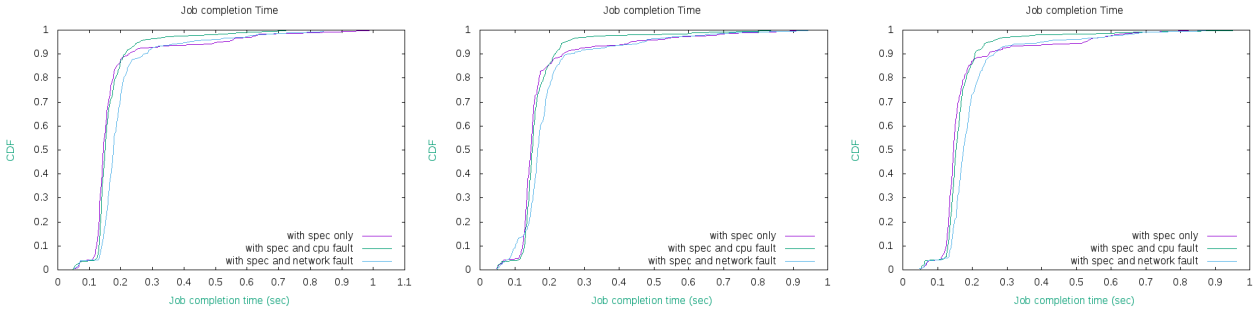


(a) m=1.00, w=0.50 (b) m=1.00, w=0.75 (c) m=1.25, w=0.50

Figure 6: Job Completion Time with Varying Parameter Sets

tasks detection.

The most interesting part of the results from Figure 5a & 5b is that majority of the cloned tasks are ignored as the original tasks completed before the cloned tasks made enough forward progress. A very few cloned tasks are actually considered as the original tasks have not made enough progress. Thus, from these figures, we conclude that the ratio of number of tasks that are detected as speculatable and number of cloned tasks that are considered is very high. For a better straggler mitigation strategy, this ratio should always be equal to 1. The higher the ratio, the more the chances are of wasting the resources in running the cloned tasks. Moreover, such higher ratio doesn't actually help in the performance gains too when faults are present.



(a) m=1.25, w=0.75 (b) m=1.50, w=0.50 (c) m=1.50, w=0.75

Figure 7: Job Completion Time with Varying Parameter Sets

4.2 Spark Streaming

4.2.1 Baseline

For the baseline, we run Spark Streaming for benchmarks: "identity", "sample", "grep", "projection", and "distinct count". The baseline has default Spark streaming parameters, wherein Spark Speculation is turned off. We generate maximal load conditions via these workloads by increasing the number of topic partitions in Kafka to 12 and having 3 Kafka instances per node. We also configure Kafka & HiBench in "push" mode which streams data at as fast data rates as possible. We minimize the batch-size of Spark Streaming to 1 second. Each batch-size interval a job is spawned in Spark Streaming. We observe that no job takes longer than 1 second. We also observe that "distinct count" workload generates the maximum CPU, disk, and network load conditions out of all the benchmarks. However, the load generated even in these conditions is not substantial and is not compute, network or disk intensive. This might be because HiBench uses Kafka to generates it's data via reads from HDFS. Subsequent results are on these best-effort settings and the "distinct count" benchmark.

4.2.2 With Speculation and/or With Fault Injection

In these set of experiments we enable speculation. In Figure 6-7, we vary `speculation.multiplier` and `speculation.quantile`, and also inject CPU and network faults for each of these parameter sets. We do not inject disk faults as this only affects Kafka and not Spark Streaming (checkpointing is disabled). We use the same parameter sets as used in Table 2. We plot the CDF of job completion time for various parameter sets and run each experiment for 4 minutes. For the cases where we inject faults, we introduce them at the mid-experiment mark of 2 minutes.

We observe that the CDF curves for these tasks are almost similar across parameter sets (graphs). Also, we do not observe a wide difference in the job completion curves for "with faults and speculation" and "only speculation". This can be because the fault framework throttles CPU for only 5 seconds on a single machine, and the network fault throttles network between two randomly chosen nodes for 15 seconds. The effect of the faults is localized and not long-lived and is not visible in the CDF's we observe. This is much clearer in Figure 9 and Figure 10. In general, the "with CPU faults" curve is very similar to the "only speculation" curve. This is so because our benchmark, despite our best efforts, was not CPU intensive. The effect of CPU faults was not absolute as OS has checks and balances to prevent scheduler biasing. The "with network faults" curve has a worse job distribution time, which is possible because of the slightly elongated duration of the fault and also because every batch-interval a `reduceByKey` is involved in "distinct count" benchmark.

In Figure 8, 9, and 10 we plot the job duration times for finished tasks and the corresponding ignored tasks. The number of observed task duration's in these graphs give the number of speculated tasks. Note, we make changes to Spark source code to get additional logs regarding the ignored tasks.

We observe that even without faults we see task speculation as shown in Figure 8. Also the X-axis in each of these graphs gives the Task ID. (These are not unique because the same Task ID may be common for different stages). A Task ID on the X-axis denotes the ignored task. A Task ID ending with "*.1" denotes that the speculated task was futile as the corresponding "*" task completed before the speculated task finished. We see that for only speculation, most of the speculations are futile. Also, we see that the gap between the finished and ignored task is very high. This seems to suggest that, if at all required in this benign case, task speculation kick-in too late.

Between CPU faults in Figure 9 and Network faults in Figure 10 the number of speculated

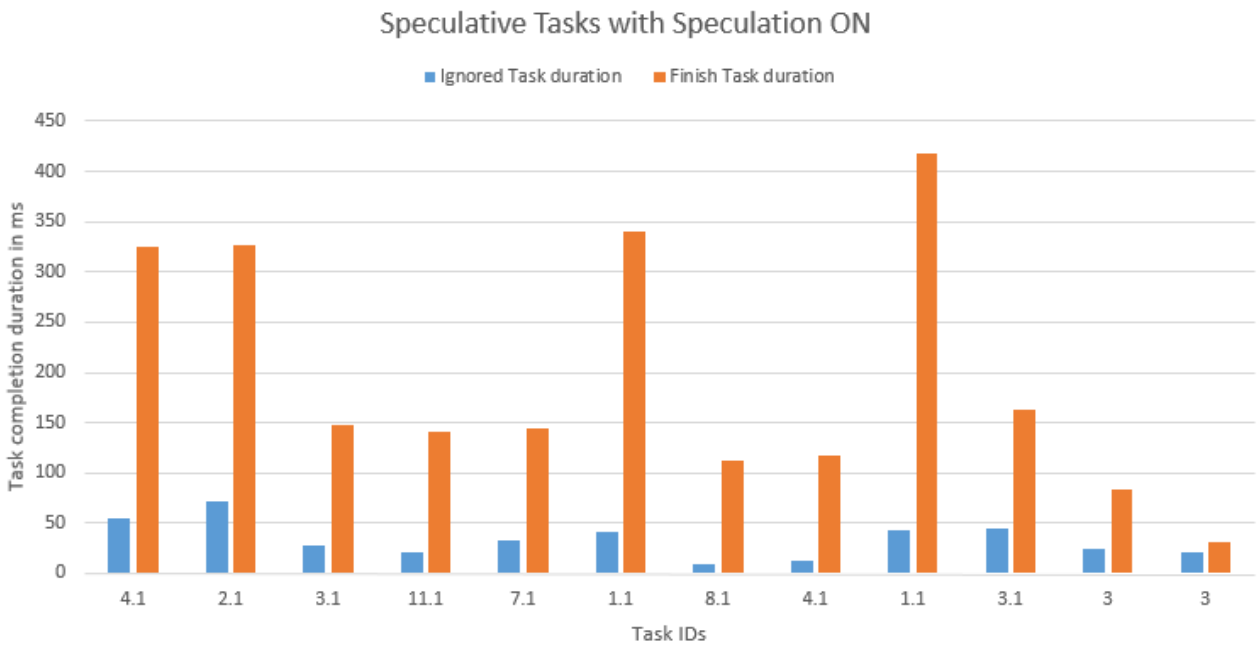


Figure 8: Remnant and Finished Job Duration with only Speculation.

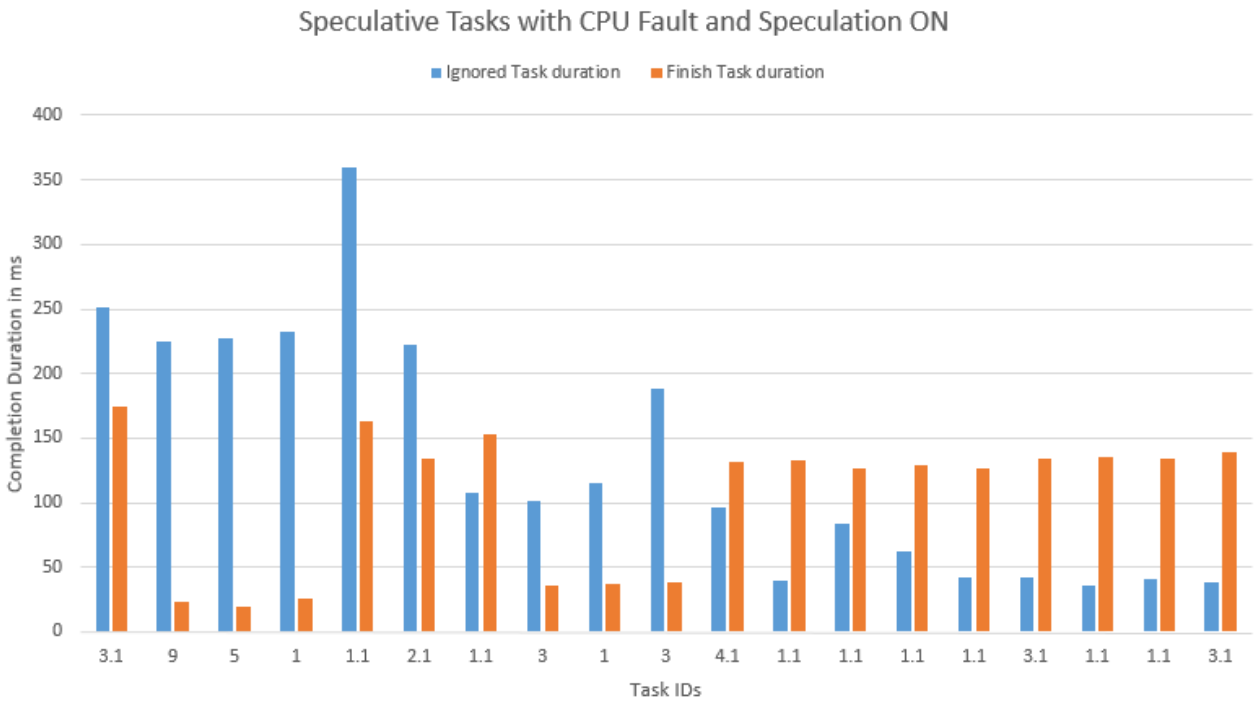


Figure 9: Remnant and Finished Job Duration with CPU Fault.

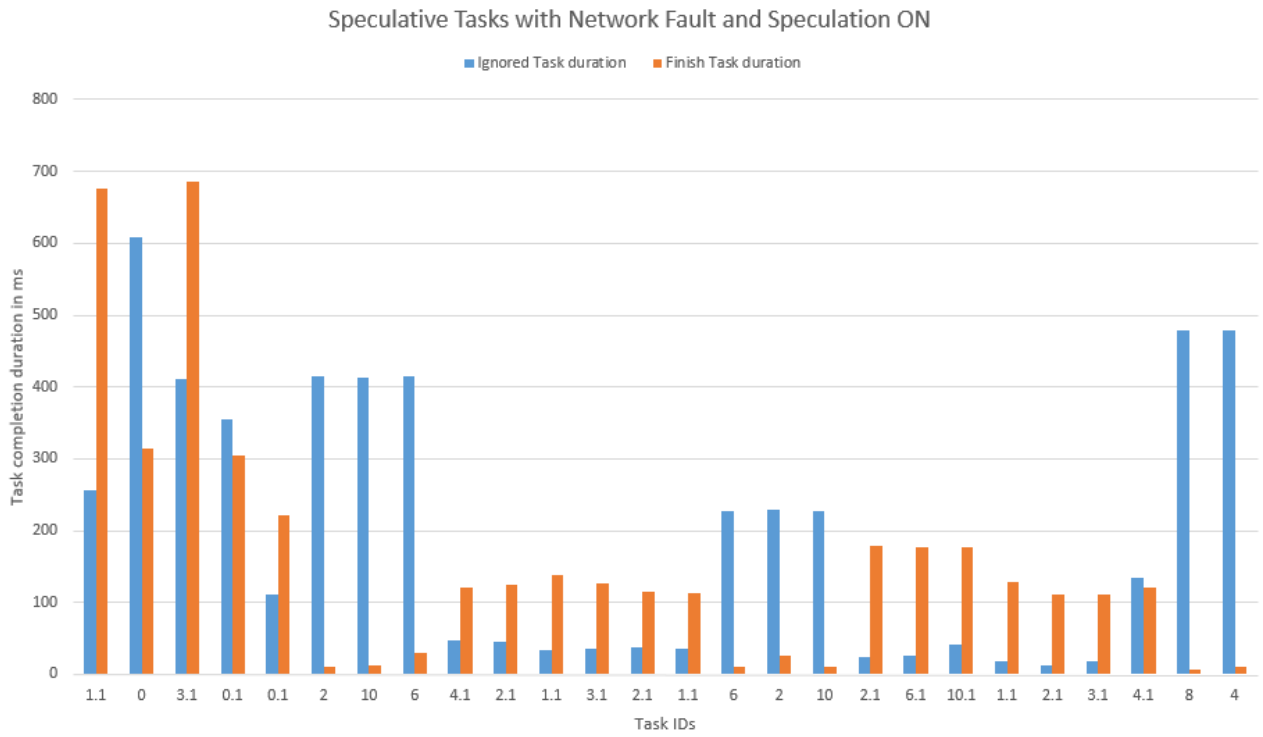


Figure 10: Remnant and Finished Job Duration with Network Fault.

tasks increase which is possible due to the same reasons "with network faults" yielded slightly worse job completion times. Also the number of futile tasks in the fault cases decreases. Another interesting observation in these graphs is the difference between the duration's of the finished and the ignored task. A wider gap in the duration's when the ignored task is of the format "*" is highly desirable (i.e. ignored duration - finished duration must be very large). This is especially observed in the network faults case and also CPU faults case. The "*.1" tasks in both the cases are either false-positives as in the "only speculation" case or suggest late kicking-in of speculation.

5 New Ideas

Based on the analysis done so far, we recommend the following ideas to be implemented for better Straggler Detection and Mitigation.

1. Use mean/standard deviation instead of median for straggler detection and also address the design issues highlighted.

2. Use the stage wise task completion time to detect stragglers for tasks belonging to a particular stage instead of using the entire task spectrum to identify speculatable tasks.

3. Majorly for spark mitigation, blacklisting the nodes where stragglers are seen is performed. But that is not enough as we end up not using that resource entirely. If many such nodes are found behaving abnormally due to a software bug, then a lot of resources are under utilized. So, we can come up with a better strategy where we define the confidence level of executing a task properly on all the given nodes and use that confidence level to allocate tasks. The idea is that we quantify the confidence level for any given machine and the master use it to decide the task scheduling. So, on a machine where more stragglers are seen, a very few tasks will be allocated for execution and over time, if

the machine behaves well, its confidence level will be increased by the master so that more work can be allocated to the machine instead of penalizing the machine by blacklisting it.

6 Future Work

In this report, we just represented two queries for Spark SQL even though we had run the experiments on 48 queries. However, for 2 queries we have done exhaustive analysis about the existence of Stragglers in those queries. As part of the future work, we would like to analyze each and every query and identify other interesting data points that we have not noticed in the two queries we have considered for the report so far. Even a study on multiple queries running at the same time will help us understand how the tasks of one job can impact the performance of other jobs running on the same node. Running the workload on a cluster setup having thousands of nodes will help us in catching the minute details pertaining to Stragglers. We would even like to run experiments on production workloads so that we can understand the frequency of the faults and the causes for Stragglers. Ganesh et. al in [1] have concentrated on Map-Reduce jobs which are different from the Spark jobs and hence this study will help us discover more specific reasons with respect to Spark environment that can cause Stragglers.

We would like to make our fault injection framework more robust and more flexible to allow many more fault runs to be done. Extending the above experiments done on other different faults will help us understand if a certain type of fault is actually important from the performance point of view or not. Can few faults be mitigated in a better way rather than categorizing them as Stragglers? We would also like to study the impact of microarchitecture on the various kings of Cloud workload as described in [3]. Using that study, we would analyze if there is any effect of microarchitecture on the presence of Stragglers.

Finally, we would like to understand the results published in [4] and map their results with the results obtained by our experiments to bridge the gap between the Straggler understanding, Straggler identification and Straggler mitigation.

References

[1] Ganesh A, *Reining in the Outliers in Map-Reduce Clusters using Mantri*, NSDI, 2013.

[2] https://github.com/intel-hadoop/HiBench

[3] Michael Ferdman, *Quantifying the Mismatch between Emerging Scale-Out Applications and Modern Processors* ACM Transactions on Computer Systems, Vol. 30, No. 4, Article 15, November 2012

[4] Kay O, *Making Sense of Performance in Data Analytics Frameworks* NSDI, 2015.