Namrata Vajrala (vajrala3)
Aydan Pirani (apirani2)

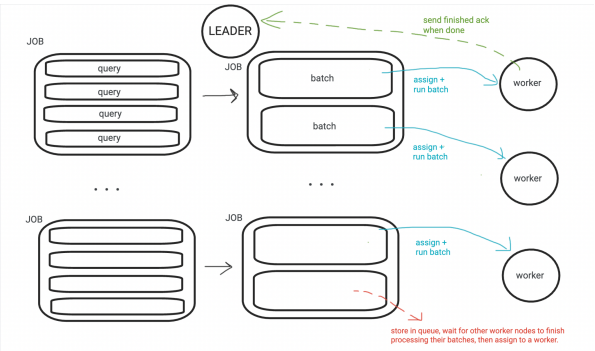
MP4 Report

Design

To start running our jobs, we take in a jobs.jobconf file as input which consists of the following fields: Name, Enabled (defaults to True, determines if the job runs), Priority (defaults to 1000 for least important, determines which jobs are prioritized), Batch size (defaults to 1, determines how many queries each worker will process at once), Number of queries (defaults to 1), and Input Source (defaults to NULL, must be present to run).

To process our jobs, we first maintain a jobs priority queue([priority, batch]), a batches queue([(model, [batch])]), and a running_batches map(node -> [batch]), to store our jobs. Once the job is received, we divide the queries into batches, so each job has (num queries)/(batch size) jobs. We then add these jobs onto our jobs and batches structures. At the leader, we constantly pop from the batches queue and send out a batch to all of the worker nodes. Every worker node works on one batch at a time and we track this by adding it to our running_batches. Once a worker node finishes processing a batch, it sends a finished ack to the leader. The leader then deletes this entry from batches and workInProgress. It deletes the job from jobs if all the batches for that job are completed.

To handle leader failure, we assign a hot-standby node. The hot-standby node is assigned to the node with the second lowest id. The leader sends its jobs, batches, and workInProgress structures which keep track of the current progress to the hot-standby node every time a change is made to them. So when a leader fails, the hot-standby can be assigned as the new leader node since it has all the information needed to be a leader. A multicast is then sent out about the new leader and a new hots-tandby node is assigned.

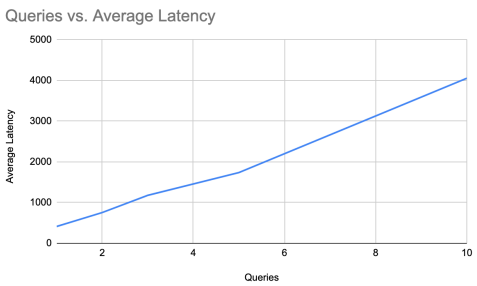


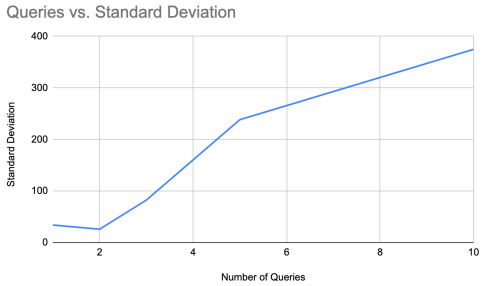
To handle worker node failure, we track it by checking for the acks from the worker nodes. If there are no elements left in the worker queue (but we have idling worker nodes), we assume that all nodes in the workInProgress map are failed or idling, and duplicate these jobs. Therefore, this allows us to account for the failures and generate more inferences.

Questions

1. **Fair-Time Inference**
    a. **When a job is added to the cluster (1 -> 2 jobs), what is the ratio of resources that IDunno decides on across jobs to meet fair-time inference?**

    We assign one batch per worker node. So each worker node would have worked on batches/worker_nodes batches. Each batch is assigned by dividing the number of queries for a job by the batch size. However, each job runs sequentially, in such a way to maximize resources that each job has access to.





    (Note that all measurements above are taken in milliseconds).

    The above graphs show that the latency and standard deviation increases as the number of queries do in a linear manner. This is expected and corresponds to our design because we assign one batch to a worker node at a time. When there are no more worker nodes to assign batches to, we add the batches to a queue to wait for the worker nodes to finish their batch. One the leader receives a finished ack from the worker, then it assigns another batch. So the more batches there are, the longer it would take for all the batches to finish and this would be almost linear.

    b. **When a job is added to the cluster (1 -> 2 jobs), how much time does the cluster take to start executing queries of the second job?**

    The cluster waits for all the queries of the first job to finish, before it starts executing queries of the second job. Naturally, this is dependent on the number of queries, which we demonstrate here.

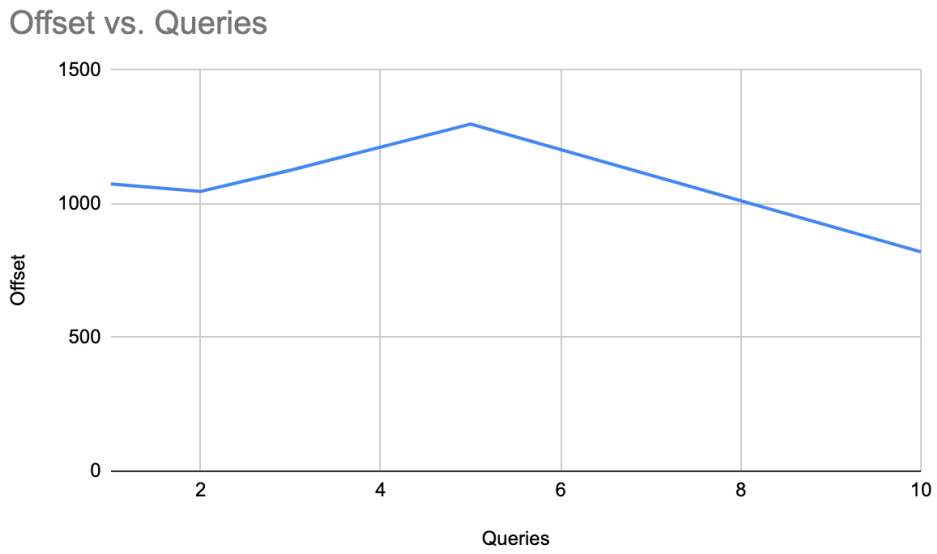
    In this case, note that the same graphs as the above question apply here: the second job only starts after the first one finishes, so the time for the second job to execute is directly proportional to the time for the first one. Therefore, note that this is synonymous with the time taken for resource allocation.

2. **After failure of one (non-coordinator) VM, how long does the cluster take to resume “normal” operation (for inference case)?**

After the failure of a non-coordinator node the cluster takes no time to continue normal operation because we simply mark the batch as failed (i.e we read it to our worker queue and let it be popped eventually). This does have the implication of increasing regular runtime, but comes at the advantage of consistently delivering inferences.

3. **After failure of the Coordinator, how long does the cluster take to resume "normal" operation (for inference case)? Unless otherwise mentioned all experiments are with multiple jobs in the cluster.**

   Interestingly, we noticed that there was no change in the time taken to resume normal operations versus the number of queries. In the context of our design, this is because the leader election runs on its own thread, parallel to the rest of the code. Therefore, the actual detection of the leader is dependent on the internals behind locking and unlocking the membership list, which is independent of the number of queries run.



(Note that offset is measured in milliseconds)