[External] Apache Yunikorn Soak Testing

Authors: Shravan Achar (shravan.achar91@gmail.com), Junyan Ling (jling22@apple.com), Weiwei Yang (abvClouds@gmail.com)

Reviewers: Apache Yunikorn Community

Background

Yunikorn is a resource scheduler that helps run batch workloads efficiently. Running the scheduler reliably in production comes with its share of challenges.

The Apache community behind the project does a minor version release every 3 months or so. This release carries new features, bug fixes and maintenance activities. The scheduler is made up of a few important components. The scheduler core is built separately from the scheduler shim (which is an adapter for Kubernetes). The repositories are equipped with unit testing frameworks and also have a mechanism to run e2e with mock objects. A framework for load testing the scheduler on real clusters is missing.

New releases and validation of bug-fixes rely on feedback from community members. This places the members between a rock and a hard place. To get adequate feedback, some of the features/bug-fixes have to be tested on production scale environments. SRE teams mandate that production gets stable software deployed.

Having started with the 1.3 version of Yunikorn over a year ago, we discovered memory leaks in production (painfully). By that time the 1.4 version was already released and deployed. It is not uncommon for bugs to not be discovered for many versions but it's a case to be made for missing soak testing. The memory leak fixes were eventually included in the 1.5 version release. With the 1.5 version release, some nasty dreadlocks were discovered. As is the nature of the problem, the deadlocks with Yunikorn scheduler are hard to find and fix. It needs to be an ongoing process and requires changes to the release pipeline to avoid bringing in more deadlocks while validating previously fixed cases.

Certain classes of issues only are seen in a cloud environment. For example, a poorly configured autoscaler can impact scheduler performance. Additionally, the workload patterns are often random and uncoordinated. Due to these reasons, simulating a soak testing environment at scale is a key challenge to overcome.

The internal harness that some companies have built utilizes in-house infrastructure (amounting to around 50-150 real nodes and 800 fake nodes). The harness is manually triggered using the in-house spark distribution acting as the workload with Prometheus & Grafana for monitoring. Qualifying every upstream release and having an internal vetting mechanism (which is currently manual) requires considerable ongoing commitment. Automating this with well-defined performance standards and donating to the community is the best option for long-term sustainability of this software's use within any organization.

Proposal

- Develop a Unified Configuration Schema for Testing:
    - Create a flexible and customizable configuration schema that streamlines the testing of scheduling correctness and performance across different scenarios for each release.
- Create Reusable Test Case Templates:
    - Design reusable test case templates for key components such as jobs, pods, nodes, queues, and chaos scenarios to eliminate redundancy and reduce engineering overhead.
- Implement an Automated Release Verification Pipeline:
    - Build a fully automated release verification pipeline that integrates all test cases, ensuring continuous and efficient validation of scheduling correctness and performance with each new release.

High Level Test Cases

These are some of the areas that need more coverage especially in a soak environment

- autoscaling cases:
    - Test the scheduler's performance and responsiveness when the cluster scales up to handle additional nodes and workloads; Test scheduler's behavior when scaling down the cluster while ensuring scheduling continues smoothly.
- chaos engineering:
    - Introduce node or pod faults and tests the scheduler performance; Test the scheduler's reaction to node faults and its ability to reschedule pods on available nodes; Simulate pod faults and verify the scheduler's ability to handle such cases reasonably while not impacting scheduling other pods.
    - Test the scheduler's behavior and performance when it is restarted while workloads are in progress, especially batch jobs in various stages: accepted/running/completing/completed
- long running workloads:
    - Long running batch workloads often can hide memory leaks or suboptimal fragmentation of resources after many cycles of scheduling. Test how long-running batch jobs or persistent workloads interact with the scheduler, especially in terms of resource usage and potential issues such as memory leaks or fragmentation.

Kwok vs Real Clusters

By running the soak testing on Kwok based autoscaled clusters, following scenarios might not be covered

- Tests involving real Kubelet behavior, such as when multiple schedulers are used on the same node
- API Server behavior in terms of flow control, throughput settings. Some kubernetes distributions have custom API servers that won't be covered

Kwok can be used to cover most of the other areas including

- Using cluster autoscaler or Karpenter
- Scheduler preemptions
- Any other kubernetes controller that work with k8s pod APIs

Using the test harness

This test harness would be integrated as part of the CI pipeline. Since all of the components can be simulated in a kind cluster, the scheduler behavior can be well quantified. The tests can be utilized at various stages of development, staging and release cycles using Github labels.

- As part of the PR (pull request) process while adding commits to a new rc (release candidate) branch.
- Major PRs including substantial refactoring, critical bug fix and fix backports
- As part of the release voting process

Config Schema

Framework's role is to run the tests, monitor the metrics, and pass/fail the test with the summary of the test cases run.

Overall goals of the config

- Determine success criteria in terms of metrics/pending pods
- Thresholds must be met
- Thresholds are determined by heuristics and based on previous release performance

Labels

The following labels are categorized into 3 sections.

- short
  - Involves tests that take upto 10m to complete.
- long
  - Involves tests that can take up to an hour.
- super-long
  - Involves tests that can take multiple hours to finish. Typically run during release

Alternative labelling

These labelling schemes are included as optional comments in the sample below

- integration-test
  - Involves tests that can be run in the CI pipeline. Typically <10-30s. These include autoscaling tests
- soak-test
  - Longer running tests that can be run in CI. Typically 2 hours - days. These include long running tests which includes basic to full functionality. Additionally include chaos engineering tests
- benchmark-test
  - Tests to run benchmarking against previous releases. Typically runs for an hour

Sample

```
- name: autoscaling
  template:
    node:
    - path: ../templates/nodeGroupTemplates.yaml
      maxCount: $nodesMaxCount
      desiredCount: $nodesDesiredCount
    job:
    - path: ../templates/jobATemplate.yaml
      count: $numJobs
      podCount: $numPods
      mode: "always" #one of ["always", "random-max-percent",
"fixed-percent"]
      value: "50" # when mode is "random-max-percent" or "fixed-percent"
```

```yaml
    - path: ../templates/jobBTemplate.yaml
      count: $numJobs
      podCount: $numPods
    scheduler:
    - path: ../templates/autoscaling-queues.yaml
      vcoreRequests: 2
      vcoreLimits: 2
      memoryRequests: 16Gi
      memoryLimits: 16Gi
  testCases:
  - name: "1000-nodes-cluster"
    params:
      nodesMaxCount: 1000
      nodesDesiredCount: 20
      numPods: 5000
      numJobs: 200
    schedule: once
    labels: ["short"]
    # labels: ["soak-test"]
    threshold:
      maxRuntime: "10m"
      pendingPods: 0
      metrics:
        maxAllocationDelay: "5s"
  - name: "5000-nodes-cluster"
    params:
      nodesMaxCount: 5000
      nodesDesiredCount: 20
      numPods: 20000
      numJobs: 700
    schedule: once
    runs: 1
    # labels: ["soak-test", "benchmark-test"]
    labels: ["short"]
    threshold:
      maxRuntime: "60m"
      pendingPods: 0
      maxAllocationDelay: "20s"
  - name: "300-nodes-cluster-schedule"
    params:
      nodesMaxCount: 300
      nodesDesiredCount: 0
      numPods: 2000
      numJobs: 150
    schedule: "*/15 * * * *"
    runs: 10
    #labels: ["soak-test"]
    labels: ["super-long"]
    threshold:
      maxRuns: 10
      pendingPods: 0
      metrics:
        maxAllocationDelay: "5s"
- name: chaos-faults
  template:
    node:
    - path: ../templates/nodeGroupTemplates.yaml
      maxCount: $nodesMaxCount
      desiredCount: $nodesDesiredCount
    job:
    - path: ../templates/jobATemplate.yaml
      count: $numJobs
      podCount: $numPods
    choas:
    - path: ../templates/chaos.yaml
      count: $numChaos
    scheduler:
    - path: ../templates/chaos-queues.yaml
      vcoreRequests: 2
      vcoreLimits: 2
      memoryRequests: 16Gi
      memoryLimits: 16Gi
  testCases:
  - name: "1000-nodes-cluster"
    params:
      nodesMaxCount: 1000
      nodesDesiredCount: 20
      numPods: 5000
      numJobs: 200
      numChaos: 0
    schedule: once
    labels: ["short"]
    # labels: ["soak-test", "benchmark-test", "integration-test"]
    threshold:
      maxRuntime: "10m"
      pendingPods: 0
      detectDeadlock: false
      metrics:
        schedulerRestarts: 0
        maxAllocationDelay: "10s"
  - name: "5000-nodes-cluster"
    params:
      nodesMaxCount: 5000
      nodesDesiredCount: 20
      numPods: 20000
      numJobs: 700
      numChaos: 200
      schedule: once
      runs: 1
      labels: ["long"]
      # labels: ["soak-test", "benchmark-test"]
      threshold:
        maxRuntime: "60m"
        pendingPods: 0
        detectDeadlock: true
        metrics:
          schedulerRestarts: 1
          maxAllocationDelay: "60s"
  - name: "300-nodes-cluster-schedule"
    params:
      nodesMaxCount: 300
      nodesDesiredCount: 0
      numPods: 2000
      numJobs: 150
      numChaos: 10
    schedule: "*/15 * * * *"
    runs: 10
    # labels: ["soak-test"]
    labels: ["super-long"]
    threshold:
      maxRuntime: "60m"
      pendingPods: 0
      detectDeadlock: true
      metrics:
        schedulerRestarts: 5
        maxAllocationDelay: "60s"
        prom:
          - query:
'sum(rate(go_memstats_heap_inuse_bytes{service="yunikorn"}[60m])) by
```

```
(service)'
            expression: 'sprintf("%.0f", query_result / 1000000)'
            value: '20'
            op: '<='
```

Metrics Collection

After every run, parameters in the threshold section must be evaluated for the success criteria. The following order of checks are followed

- After `maxRuntime` seconds of every run, number of pending pods are checked. If number of pending pods are more than configured threshold, the test run is considered a failure.
- If `detectDeadlock` is true, the framework looks for a special file containing stack traces / deadlock info. If file is found, the test run is considered a failure, and the developer would need to look at deadlock info or re-run the test case
  - Yunikorn scheduler would be run with a deadlock detector - https://github.com/sasha-s/go-deadlockenabled. It will be configured to output to a file if deadlock occurs.
- After `maxRuntime` seconds of every run, and pending pods threshold is met, metrics configured in the `metrics` section is evaluated on every pod launched using various techniques
  - `maxAllocationDelay` is calculated using the scheduler's statedump
  - `schedulerRestarts` is calculated using promQL query on the prometheus server (more on this later)

Scheduler Metrics Collection

Things like memory leaks require access to scheduler's internal metrics. To support this, Yunikorn scheduler would need to run with a Prometheus server that scrapes its metrics at a regular interval.

- `prom` section lists expressions that will be evaluated against the result of the query response from the prometheus server. Tests pass if the `expression` evaluates less/more/equal to the `query_result`.
  - https://github.com/expr-lang/expr supports dynamic expression evaluation. Results of promQL HTTP query would be passed into this library for evaluation.

Log Collection

One of the goals is to have reproducible tests. Given a test run failure, the developer can run the entire test case again while monitoring the scheduler logs in a separate process

Test logs will be available through the framework on std output.

References

https://github.com/kubernetes/kubernetes/blob/master/test/integration/scheduler_perf/misc/performance-config.yaml
https://github.com/kubernetes/kubernetes/blob/master/test/integration/scheduler_perf/scheduler_perf.go
https://github.com/sasha-s/go-deadlock?tab=readme-ov-file#configuring-go-deadlock
https://prometheus.io/docs/prometheus/latest/querying/api/
https://github.com/expr-lang/expr

GoMemLimit settings

Archive: [No longer relevant to the current proposal]

Example config:

```
- name: SchedulingBasic
 defaultPodTemplatePath: ../templates/pod-default.yaml
 workloadTemplate:
 - opcode: createNodes # Affinity
   countParam: $initNodes
 - opcode: createPods # Tolerations/ Node selectors
   countParam: $initPods
 workloads:
 - name: 5Nodes
   featureGates:
     SchedulerQueueingHints: false
   labels: [integration-test, short]
   timeToRun: 2d
   threshold: {} # Node
   params:
     initNodes: 500
     initPods: 1000
```

```
environments:
- name: "1000-nodes-cluster"
  - nodes: 500
    labels:
    taints:
      resources:
  - nodes: 500

testcases:
  - name: "basic scheduling"
  - env: "1000-nodes-cluster"
  - workloads:
  - scheduledJobs:
  - name: "job-a"
  schedule: "5 MINUTES"
  duration: "10 MINUTES"
  Replicas: "100"
  MaxRuns: "100"
  - name: "job-b"
  scheudle: "30 MINUTES"
  duration: "5 MINUTES"
  Replicas: "100"
  MaxRuns: "100"
  - successCriterias:
  - maxRuntime: "2 hours"
  - pendingPods: 0
  - failedPods:
  - yunikornQueueResources:
  - schedulerPodResources:
  CPU:
    max: 2
  Memory:
   max: 5G
```

Test infrastructure size on AWS.

To test the scheduler on Kubernetes, EKS clusters would be used. EKS clusters can use either managed or unmanaged node groups backed by EC2 instances.

Number of EKS clusters - 2
Number of EC2 nodes

- Baseline of 7 EC2 nodes
- Peak of 500 EC2 nodes
- Traffic pattern would be weekly spikes of about 24 hours / week of peak usage (3 days of 8 hours / day activity per week)

Instance types

- Will be limited to 4xlarge instance types or lower specs. Though r6g.16xlarge has better bandwidth per core ratio, the instance hour costs are higher.
- A pool of similar node groups would be added to maximize availability and spot capacity. r6g.4xlarge, m6g.4xlarge and c6g.4xlarge can be used as the instance pool.
- On-demand instances would also be needed (as opposed to spot) for running tests on long running workloads.
- The other pricing plans are probably not suited for soak testing purpose as they trade lower prices for fixed commitment.

EBS Volume requirements

- GP3 with 3000 IOPS per volume (free tier)
- 125 MBps throughput (free tier)
- 16GB storage

CloudWatch resources

- Upto 100GB of logs ingested every month and upto 1000GB of logs scanned every month

Data Transfer

- All traffic to remain within the region. No outbound data transfer is expected

S3

- 10 GB per month of storage with 4GB of average object size for periodic statedumps and goroutine stacks

With the help of a pricing calculator, approximate monthly cloud credits are around 17000 USD with EC2 costs accounting for approximately 14000 USD. This is based on the estimated use.

Approximately 3 releases per year. Estimated number of months per year exhibiting a weekly traffic pattern is 5-6 months. Estimated yearly cost 85K-102K.

Testing details

Long running workloads

Long running batch workloads often can hide memory leaks or suboptimal fragmentation of resources after many cycles of scheduling. In Spark, Notebook Jobs or streaming jobs often fit into this category. ML serving workloads can be long running.

Test cases

1. Use HPA APIs for updating replicas of a Deployment workload
2. Use JobSet APIs for simulating

Elastic batch workloads with autoscaler

Batch and ML workloads with elasticity (like Spark) or workloads using Dynamic Resource Allocation feature would be hard to simulate on local clusters.

Test cases

3. Adding a new node
4. Removing a node
5. Triggering dynamic resizing of a workload

Scheduler restarts

Test cases

6. Restart scheduler while batch workloads are in accepted/running/completing/completed stages

Chaos Engineering - Introducing faults

Test cases

7. Simulate a node fault
8. Simulate a pod fault on Spark driver

Summary

Test type	Duration	Instance types	Scale	Acceptance Criteria
Long running workloads	8-12 hours	on-demand	Upto 500	Prometheus alarms does not show memory growth
Elastic batch workloads	1-2 hours	spot	Upto 20	Batch allocation with elasticity works with autoscaling
Scheduler restart cases	1-2 hours	spot	Upto 20	Statedumps do not indicate anamolies
Chaos mesh fault tests	6-8 hours	spot	Upto 200	TBD