Kubernetes and the dynamic world in the cloud

Frank Conrad
Software Engineering Technical Leader

Cisco

scalable efficient low latency processing

frank@fc-tb.de

Businesses need dynamic scale

- full fill SLA all the time
  - even during peak events like black friday,…
- get operational cost under control
- deliver reliable features
- „time is money“

operational challenges

- jobs have very different resource requirements
  - per weekday, end of week / month / quarter /…
- catch up of failures / needed reprocessing
- with fixed size cluster
  - schedule optimization
  - job can influence each other
  - uninterruptible jobs

what you ideally want

- right resources for each job for them to efficiently
- run jobs with the maximum independence
- add new or modify existing jobs with little to no effect to others
- don’t care about server idle time
- don’t overpay for resources which you don't need or only need for a short time

what scale up/down can achieve

- same cloud compute cost but faster results
  - 24h 100cpu <-> 4h 800cpu
  - 24h 100cpu <-> 2h 800cpu + 22h 36cpu
- commitments alternative (+ on top for events like black friday)
  - 24h 100cpu -> 8h 100cpu + 16h 40cpu -> 40% saving
  - 24h 100cpu (+30%) -> 8h 100cpu + 16h 40cpu -> 54% saving
  - 24h 100cpu (+50%) -> 8h 100cpu + 16h 40cpu -> 60% saving
  - 24h 100cpu (+100%) -> 8h 100cpu + 16h 40cpu -> 70% saving

cloud

- you pay for what you provision by time
  - resource/utilization based billing
- next to no lead time to get new resources
- you can give back what you don’t need
- all done in a few seconds/minutes
- but managing it is very provider dependent

kubernetes in cloud

- has good cloud support
- built-in support for real dynamic clusters (cluster autoscaler)
- supports good CI/CD
- provider, vendor agnostic API / usage
- hide the complexity of different providers
- simpler version handling / migration for apps
- change from app per vm to app per container model
- operator support for simpler use

k8s operators

- operator pattern
    - bring ops/sre knowledge in code
    - control operator via Custom Resource Definitions (CRD)
- mostly installed / updated via helm
- source to find them:
    - https://operatorhub.io/
    - https://github.com/operator-framework/awesome-operators

What are the benefits for big data

- scalable jobs can produce faster results for similar costs
- compute can grow with the size of data
- cluster sized only to match current needs not to the max (black friday)
- recovery of failed job can run independently and faster by using higher scale

same numbers on use with cluster-autoscaler

- on gke cluster 1.18 with 18 node-pools scales by cluster auto-scaler
  - new pod triggered new node take 30-45 sec get pod running
  - deployment that's starts 3k pods and trigger start of 1000 nodes
    -> take 4 min
  - start 18k pods with large images which trigger 1000 nodes to start
    -> take 17min
  - overhead per node: CPU ~200m, memory 2.7G or 5%

Cluster auto scaler

- responsible to add (scale up) and remove (sale down) nodes to a cluster
- looks for unschedulable pods
- run simulation to find “right” node-pool and adds a node there
- looks for underutilized nodes to see if it can delete them
- provides the needed resources up to the limits specified in max node-pool size

cluster autoscaler scale down

- underutilized nodes are where sum of cpu and memory requests below 50% (or scale-down-utilization-threshold)
  - for 10min (or scale-down-unneeded-time)
- looking for blocking pods
  - local storage
  - no controller
  - special annotation
    - cluster-autoscaler.kubernetes.io/safe-to-evict : false
  - resources to run pod somewhere else are there
- during scale down
  - respect pod disruption budget (PDB)
  - respect GracefulTermination up to 10min (or max-graceful-termination-sec)

what does it mean for us

- as typical big data jobs get strongly affected by restart of pods
  - especially if multiple get affected at the same time or in rolling / sequential way
- add the following annotation to pods to prevent it:
  - cluster-autoscaler.kubernetes.io/safe-to-evict : false

k8s scheduler

- find the “right” node to run the pod
- filter all nodes by strict limitation (available resources, nodeselector, affinity, tolerations,…)
- if no matching node found, then mark the pod unschedulable (to trigger auto scaler)
- for all matching nodes calculate the priority, done by weight via rules (plugins)
  - this default behavior gives you a well distributed load on cluster with fixed size
- assign pod to node with the highest priority
- this is done pod by pod
- scheduler experiences latency when it involves high number of nodes/pods
  - with priority classes you can influence the priority order

what this means for us

- as scheduling is done pod by pod
  - in many cases could happen that not all pods of a job get started
    - end up with the job never finishing
    - dead lock if multiple jobs get affected
  - solutions:
    - cluster auto scaler: add needed resources
    - use other scheduler, which address the problem (gang schedule)

other k8s scheduler

- This is the way to go on very large scale and/or limited resources.
- there are multiple custom schedulers or scheduler plugins available
- all have pros and cons
- all pods need to have scheduler assignment
  - schedulerName: scheduler-name
- nodes (node-pools) should be only managed by ONE scheduler
- challenges to use provider based k8s cluster like gke/eks/aks,…

custom schedulers

- kube-batch https://github.com/kubernetes-sigs/kube-batch
  - gang schedule
  - Volcano https://volcano.sh/en/
    - batch optimized schedule integrated with many frameworks
- Apache YuniKorn http://yunikorn.apache.org/
  - gang schedule
- add scheduler-plugins like github.com/kubernetes-sigs/scheduler-plugins
  - leverage KEP 624-scheduling-framework

cluster auto scaler: add needed resources

- in a cloud and when not a very large scale, this is the preferred way
- its simpler and has less dependencies
  - set high max node count on used node-pools
  - on k8s > 1.18 use schedule profile to create one with strong binpacking
    - this not needed if running 1 pod per node

node-pools with > 1 pod per node

- when multiple pods are running per node, all of them need to be finished before the node can go away
- when multiple jobs sending pods to same node, the longest running job(pod) will block scale down, even if its just one pod running
  -> higher costs than needed
- optimise on it:
  - get strong binpacking, via on k8s > 1.18 use schedule profile with it
    - gke autoscaling-profile: optimize-utilization

dedicated node-pools

- create dedicated node-pool
  - add specific label
    - to bind pod to this
    - example: dedicated: 4cpu-16mem
  - add taints
    - to block unwanted pod running there
    - example: dedicated: 4cpu-16mem:NoSchedule
- set min to zero
- set max such that you never reach the limit normally (don’t forget the provider's quota)

separate compute from storage

- default way in cloud na kubernetes
- flexibly change compute based on need
- a way to save network costs (across zones) / increase performance
  - if cross zones charges is a problem

a way to save network costs (across zones) / increase performance

- by run compute in one zone but storage is multiple zones
  - Object store (s3, gcs,…)
  - network filesystems (nfs, efs,…)
  - regional persistence disk (gcp/gke)
- in case of zone failure, the whole workload gets restarted in other zone

change compute via statefulset

- allow flexible change compute based on need
- persistent volumes which are not node local (ebs,…)
- statefulset allow you the change compute resources on same storage
  - by change resource requests and eventual node affinity / tolerations
    - that triggers (rolling) update
- depending on the app you can do this multiple times per day
- usage
  - hdfs, get larger nodes during runtime of bigger jobs to leverage node local
  - kafka, to prepare for very high or low traffic

change compute via operator

- if operator allow / support this
  - by change resource requests and eventual node affinity / tolerations in CRD
    - that trigger (rolling) update
- usage
  - postgres zalando/postgres-operator)
  - kafka (strimzi-kafka-operator)
  - redis

“cluster per job” on demand

- create the right sized cluster for a job
- use different node-pools to have different node profiles available
  - control use via affinity and tolerations
  - cluster-autoscaler will take care of starting / stopping nodes
- operators make this deployment simple

spark operators

- https://github.com/radanalyticsio/spark-operator
  - manage spark cluster in k8s and openshift
  - can also work CM instead of CRD
- https://github.com/GoogleCloudPlatform/spark-on-k8s-operator
  - highly sophisticated and has a good k8s integration
    - affinity, life cycle hooks, …

spark-on-k8s-operator

- in workflow engine with no native integration
  - create CRD SparkApplication
  - watch for .status.applicationState.state
    - COMPLETED
    - FAILED

airflow and k8s

- helm chart install / update
- can run completely within k8s
- together with postgres/mysql operator and redis operator, all of it runs on k8s and uses only standard k8s functions
- has an integration to k8s
  - to allow to run tasks as k8s pods
  - to scale the executors dynamically
    - use KEDA for that, which also give many other options for horizontally scaling your deployments based on many external datasources.
  - has native support for spark-on-k8s-operator

flink operator

- https://github.com/lyft/flinkk8soperator
  - blue-green deployment
- https://github.com/GoogleCloudPlatform/flink-on-k8s-operator
  - good k8s integration

storage hints

- Object-stores (scale mostly automatically)
  - reuse buckets
  - same pattern
  - pre condition
- define local volumes for tmp / shuffle data
  - try local ssd
  - never write to images

image hints

- avoid large images if possible (multiple GBs)
- use the same base image across jobs (leveraging image cache)
  - common data add to base first
  - last job specific data
- for larger data like ML models
  - put it on NFS server (aws efs, gcp filestore)

uninterruptible (GPU) jobs

- run as jobs or static pods
- make cpu and memory request == limit
  - to get QoS: Guaranteed
  - minimize side effects from other pods on the node
- don’t forget to add to the pod:
  - cluster-autoscaler.kubernetes.io/safe-to-evict : false
- if possible use save points

how I could get this

- get k8s cluster in a cloud (gke,…)
- enable cluster autoscaler
- configure need node-groups with autoscaler
- install your needed operators / tools (best via helm):
  - https://github.com/GoogleCloudPlatform/spark-on-k8s-operator
  - https://github.com/GoogleCloudPlatform/flink-on-k8s-operator
  - https://github.com/airflow-helm/charts/tree/main/charts/airflow
    - https://github.com/zalando/postgres-operator
    - https://github.com/spotahome/redis-operator
    - optional https://keda.sh/docs/2.3/deploy/#helm
  - hdfs https://github.com/Gradiant/charts

Think different

Thank you

Questions?

Thanks to Rishav Jalan for supporting this talk.