Scheduling and Eviction

In Kubernetes, scheduling refers to making sure that Pods are matched to Nodes so that the kubelet can run them. Eviction is the process of proactively failing one or more Pods on resource-starved Nodes.

- 1: Kubernetes Scheduler
- 2: Assigning Pods to Nodes
- 3: Resource Bin Packing for Extended Resources
- 4: Taints and Tolerations
- 5: Pod Overhead
- 6: Eviction Policy
- 7: Scheduling Framework
- 8: Scheduler Performance Tuning

1 - Kubernetes Scheduler

In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them.

Scheduling overview

A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below.

If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling.

kube-scheduler

kube-scheduler is the default scheduler for Kubernetes and runs as part of the control plane. kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.

For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources and every pod also has different requirements. Therefore, existing nodes need to be filtered according to the specific scheduling requirements.

In a cluster, Nodes that meet the scheduling requirements for a Pod are called *feasible* nodes. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *binding*.

Factors that need taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and so on.

Node selection in kube-scheduler

kube-scheduler selects a node for the pod in a 2-step operation:

1. Filtering
2. Scoring

The *filtering* step finds the set of Nodes where it's feasible to schedule the Pod. For example, the PodFitsResources filter checks whether a candidate Node has enough available resource to meet a Pod's specific resource requests. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn't (yet) schedulable.

In the *scoring* step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.

Finally, kube-scheduler assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, kube-scheduler selects one of these at random.

There are two supported ways to configure the filtering and scoring behavior of the scheduler:

1. Scheduling Policies allow you to configure *Predicates* for filtering and *Priorities* for scoring.
2. Scheduling Profiles allow you to configure Plugins that implement different scheduling stages, including: `QueueSort`, `Filter`, `Score`, `Bind`, `Reserve`, `Permit`, and others. You can also configure the kube-scheduler to run different profiles.

What's next

- Read about scheduler performance tuning
- Read about Pod topology spread constraints
- Read the reference documentation for kube-scheduler
- Learn about configuring multiple schedulers
- Learn about topology management policies
- Learn about Pod Overhead
- Learn about scheduling of Pods that use volumes in:
  - Volume Topology Support
  - Storage Capacity Tracking
  - Node-specific Volume Limits

2 - Assigning Pods to Nodes

You can constrain a Pod so that it can only run on particular set of Node(s). There are several ways to do this and the recommended approaches all use label selectors to facilitate the selection. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (e.g. spread your pods across nodes so as not place the pod on a node with insufficient free resources, etc.) but there are some circumstances where you may want to control which node the pod deploys to - for example to ensure that a pod ends up on a machine with an SSD attached to it, or to co-locate pods from two different services that communicate a lot into the same availability zone.

nodeSelector

`nodeSelector` is the simplest recommended form of node selection constraint. `nodeSelector` is a field of PodSpec. It specifies a map of key-value pairs. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.

Let's walk through an example of how to use `nodeSelector`.

Step Zero: Prerequisites

This example assumes that you have a basic understanding of Kubernetes pods and that you have set up a Kubernetes cluster.

Step One: Attach label to the node

Run `kubectl get nodes` to get the names of your cluster's nodes. Pick out the one that you want to add a label to, and then run `kubectl label nodes <node-name> <label-key>=<label-value>` to add a label to the node you've chosen. For example, if my node name is 'kubernetes-foo-node-1.c.a-robinson.internal' and my desired label is 'disktype=ssd', then I can run `kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd`.

You can verify that it worked by re-running `kubectl get nodes --show-labels` and checking that the node now has a label. You can also use `kubectl describe node "nodename"` to see the full list of labels of the given node.

Step Two: Add a nodeSelector field to your pod configuration

Take whatever pod config file you want to run, and add a nodeSelector section to it, like this. For example, if this is my pod config:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
```

Then add a nodeSelector like so:

pods/pod-nginx.yaml

```
apiVersion: v1
kind: Pod
```

```yaml
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

When you then run `kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml`, the Pod will get scheduled on the node that you attached the label to. You can verify that it worked by running `kubectl get pods -o wide` and looking at the "NODE" that the Pod was assigned to.

Interlude: built-in node labels

In addition to labels you attach, nodes come pre-populated with a standard set of labels. See Well-Known Labels, Annotations and Taints for a list of these.

Note: The value of these labels is cloud provider specific and is not guaranteed to be reliable. For example, the value of `kubernetes.io/hostname` may be the same as the Node name in some environments and a different value in other environments.

Node isolation/restriction

Adding labels to Node objects allows targeting pods to specific nodes or groups of nodes. This can be used to ensure specific pods only run on nodes with certain isolation, security, or regulatory properties. When using labels for this purpose, choosing label keys that cannot be modified by the kubelet process on the node is strongly recommended. This prevents a compromised node from using its kubelet credential to set those labels on its own Node object, and influencing the scheduler to schedule workloads to the compromised node.

The `NodeRestriction` admission plugin prevents kubelets from setting or modifying labels with a `node-restriction.kubernetes.io/` prefix. To make use of that label prefix for node isolation:

1. Ensure you are using the Node authorizer and have *enabled* the NodeRestriction admission plugin.
2. Add labels under the `node-restriction.kubernetes.io/` prefix to your Node objects, and use those labels in your node selectors. For example, `example.com.node-restriction.kubernetes.io/fips=true` or `example.com.node-restriction.kubernetes.io/pci-dss=true`.

Affinity and anti-affinity

`nodeSelector` provides a very simple way to constrain pods to nodes with particular labels. The affinity/anti-affinity feature, greatly expands the types of constraints you can express. The key enhancements are

1. The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation;
2. you can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled;
3. you can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

The affinity feature consists of two types of affinity, "node affinity" and "inter-pod affinity/anti-affinity". Node affinity is like the existing `nodeSelector` (but with the first two benefits listed above), while inter-pod affinity/anti-affinity constrains against pod labels rather than node labels, as described in the third item listed above, in addition to having the first and second properties listed above.

Node affinity

Node affinity is conceptually similar to `nodeSelector` -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

There are currently two types of node affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that *must* be met for a pod to be scheduled onto a node (similar to `nodeSelector` but using a more expressive syntax), while the latter specifies *preferences* that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how `nodeSelector` works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod continues to run on the node. In the future we plan to offer `requiredDuringSchedulingRequiredDuringExecution` which will be identical to `requiredDuringSchedulingIgnoredDuringExecution` except that it will evict pods from nodes that cease to satisfy the pods' node affinity requirements.

Thus an example of `requiredDuringSchedulingIgnoredDuringExecution` would be "only run the pod on nodes with Intel CPUs" and an example `preferredDuringSchedulingIgnoredDuringExecution` would be "try to run this set of pods in failure zone XYZ, but if it's not possible, then allow some to run elsewhere".

Node affinity is specified as field `nodeAffinity` of field `affinity` in the PodSpec.

Here's an example of a pod that uses node affinity:

pods/pod-with-node-affinity.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

This node affinity rule says the pod can only be placed on a node with a label whose key is `kubernetes.io/e2e-az-name` and whose value is either `e2e-az1` or `e2e-az2`. In addition, among nodes that meet that criteria, nodes with a label whose key is `another-node-label-key` and whose value is `another-node-label-value` should be preferred.

You can see the operator `In` being used in the example. The new node affinity syntax supports the following operators: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`. You can use `NotIn` and `DoesNotExist` to achieve node anti-affinity behavior, or use node taints to repel pods from specific nodes.

If you specify both `nodeSelector` and `nodeAffinity`, *both* must be satisfied for the pod to be scheduled onto a candidate node.

If you specify multiple `nodeSelectorTerms` associated with `nodeAffinity` types, then the pod can be scheduled onto a node **if one of the** `nodeSelectorTerms` can be satisfied.

If you specify multiple `matchExpressions` associated with `nodeSelectorTerms`, then the pod can be scheduled onto a node **only if all** `matchExpressions` is satisfied.

If you remove or change the label of the node where the pod is scheduled, the pod won't be removed. In other words, the affinity selection works only at the time of scheduling the pod.

The `weight` field in `preferredDuringSchedulingIgnoredDuringExecution` is in the range 1-100. For each node that meets all of the scheduling requirements (resource request, RequiredDuringScheduling affinity expressions, etc.), the scheduler will compute a sum by iterating through the elements of this field and adding "weight" to the sum if the node matches the corresponding MatchExpressions. This score is then combined with the scores of other priority functions for the node. The node(s) with the highest total score are the most preferred.

Node affinity per scheduling profile

FEATURE STATE: `Kubernetes v1.20 [beta]`

When configuring multiple scheduling profiles, you can associate a profile with a Node affinity, which is useful if a profile only applies to a specific set of Nodes. To do so, add an `addedAffinity` to the args of the `NodeAffinity` plugin in the scheduler configuration. For example:

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - schedulerName: default-scheduler
  - schedulerName: foo-scheduler
    pluginConfig:
      - name: NodeAffinity
        args:
          addedAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
              - matchExpressions:
                - key: scheduler-profile
                  operator: In
                  values:
                  - foo
```

The `addedAffinity` is applied to all Pods that set `.spec.schedulerName` to `foo-scheduler`, in addition to the NodeAffinity specified in the PodSpec. That is, in order to match the Pod, Nodes need to satisfy `addedAffinity` and the Pod's `.spec.NodeAffinity`.

Since the `addedAffinity` is not visible to end users, its behavior might be unexpected to them. We recommend to use node labels that have clear correlation with the profile's scheduler name.

Note: The DaemonSet controller, which creates Pods for DaemonSets is not aware of scheduling profiles. For this reason, it is recommended that you keep a scheduler profile, such as the `default-scheduler`, without any `addedAffinity`. Then, the Daemonset's Pod template should use this scheduler name. Otherwise, some Pods created by the Daemonset controller might remain unschedulable.

Inter-pod affinity and anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled *based on labels on pods that are already running on the node* rather than based on labels on nodes. The rules are of the form "this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y". Y is expressed as a LabelSelector with an optional associated list of namespaces; unlike nodes, because pods are namespaced (and therefore the labels on pods are implicitly namespaced), a label selector over pod labels must specify which namespaces the selector should apply to. Conceptually X is a topology domain like node, rack, cloud provider zone, cloud provider region, etc. You express it using a `topologyKey` which is the key for the node label that the system uses to denote such a topology domain; for example, see the label keys listed above in the section Interlude: built-in node labels.

Note: Inter-pod affinity and anti-affinity require substantial amount of processing which can slow down scheduling in large clusters significantly. We do not recommend using them in clusters larger than several hundred nodes.

Note: Pod anti-affinity requires nodes to be consistently labelled, in other words every node in the cluster must have an appropriate label matching `topologyKey`. If some or all nodes are missing the specified `topologyKey` label, it can lead to unintended behavior.

As with node affinity, there are currently two types of pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution` which denote "hard" vs. "soft" requirements. See the description in the node affinity section earlier. An example of `requiredDuringSchedulingIgnoredDuringExecution` affinity would be "co-locate the pods of service A and service B in the same zone, since they communicate a lot with each other" and an example `preferredDuringSchedulingIgnoredDuringExecution` anti-affinity would be "spread the pods from this service across zones" (a hard requirement wouldn't make sense, since you probably have more pods than zones).

Inter-pod affinity is specified as field `podAffinity` of field `affinity` in the PodSpec. And inter-pod anti-affinity is specified as field `podAntiAffinity` of field `affinity` in the PodSpec.

An example of a pod that uses pod affinity:

pods/pod-with-pod-affinity.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: topology.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
```

```
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: security
                operator: In
                values:
                - S2
            topologyKey: topology.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
    image: k8s.gcr.io/pause:2.0
```

The affinity on this pod defines one pod affinity rule and one pod anti-affinity rule. In this example, the `podAffinity` is `requiredDuringSchedulingIgnoredDuringExecution` while the `podAntiAffinity` is `preferredDuringSchedulingIgnoredDuringExecution`. The pod affinity rule says that the pod can be scheduled onto a node only if that node is in the same zone as at least one already-running pod that has a label with key "security" and value "S1". (More precisely, the pod is eligible to run on node N if node N has a label with key `topology.kubernetes.io/zone` and some value V such that there is at least one node in the cluster with key `topology.kubernetes.io/zone` and value V that is running a pod that has a label with key "security" and value "S1".) The pod anti-affinity rule says that the pod should not be scheduled onto a node if that node is in the same zone as a pod with label having key "security" and value "S2". See the design doc for many more examples of pod affinity and anti-affinity, both the `requiredDuringSchedulingIgnoredDuringExecution` flavor and the `preferredDuringSchedulingIgnoredDuringExecution` flavor.

The legal operators for pod affinity and anti-affinity are `In`, `NotIn`, `Exists`, `DoesNotExist`.

In principle, the `topologyKey` can be any legal label-key. However, for performance and security reasons, there are some constraints on topologyKey:

1. For pod affinity, empty `topologyKey` is not allowed in both `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`.
2. For pod anti-affinity, empty `topologyKey` is also not allowed in both `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`.
3. For `requiredDuringSchedulingIgnoredDuringExecution` pod anti-affinity, the admission controller `LimitPodHardAntiAffinityTopology` was introduced to limit `topologyKey` to `kubernetes.io/hostname`. If you want to make it available for custom topologies, you may modify the admission controller, or disable it.
4. Except for the above cases, the `topologyKey` can be any legal label-key.

In addition to `labelSelector` and `topologyKey`, you can optionally specify a list `namespaces` of namespaces which the `labelSelector` should match against (this goes at the same level of the definition as `labelSelector` and `topologyKey`). If omitted or empty, it defaults to the namespace of the pod where the affinity/anti-affinity definition appears.

All `matchExpressions` associated with `requiredDuringSchedulingIgnoredDuringExecution` affinity and anti-affinity must be satisfied for the pod to be scheduled onto a node.

More Practical Use-cases

Interpod Affinity and AntiAffinity can be even more useful when they are used with higher level collections such as ReplicaSets, StatefulSets, Deployments, etc. One can easily configure that a set of workloads should be co-located in the same defined topology, eg., the same node.

Always co-located in the same node

In a three node cluster, a web application has in-memory cache such as redis. We want the web-servers to be co-located with the cache as much as possible.

Here is the yaml snippet of a simple redis deployment with three replicas and selector label `app=store`. The deployment has `PodAntiAffinity` configured to ensure the scheduler does not co-locate replicas on a single node.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: redis-server
        image: redis:3.2-alpine
```

The below yaml snippet of the webserver deployment has `podAntiAffinity` and `podAffinity` configured. This informs the scheduler that all its replicas are to be co-located with pods that have selector label `app=store`. This will also ensure that each web-server replica does not co-locate on a single node.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-store
            topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
```

```
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

If we create the above two deployments, our three node cluster should look like below.

node-1	node-2	node-3
webserver-1	*webserver-2*	*webserver-3*
cache-1	*cache-2*	*cache-3*

As you can see, all the 3 replicas of the `web-server` are automatically co-located with the cache as expected.

```
kubectl get pods -o wide
```

The output is similar to this:

```
NAME                           READY     STATUS    RESTARTS   AGE       IP
redis-cache-1450370735-6dzlj   1/1       Running   0          8m        10.192.4.2
redis-cache-1450370735-j2j96   1/1       Running   0          8m        10.192.2.2
redis-cache-1450370735-z73mh   1/1       Running   0          8m        10.192.3.1
web-server-1287567482-5d4dz    1/1       Running   0          7m        10.192.2.3
web-server-1287567482-6f7v5    1/1       Running   0          7m        10.192.4.3
web-server-1287567482-s330j    1/1       Running   0          7m        10.192.3.2
```

Never co-located in the same node

The above example uses `PodAntiAffinity` rule with `topologyKey: "kubernetes.io/hostname"` to deploy the redis cluster so that no two instances are located on the same host. See [ZooKeeper tutorial]() for an example of a StatefulSet configured with anti-affinity for high availability, using the same technique.

nodeName

`nodeName` is the simplest form of node selection constraint, but due to its limitations it is typically not used. `nodeName` is a field of PodSpec. If it is non-empty, the scheduler ignores the pod and the kubelet running on the named node tries to run the pod. Thus, if `nodeName` is provided in the PodSpec, it takes precedence over the above methods for node selection.

Some of the limitations of using `nodeName` to select nodes are:

- If the named node does not exist, the pod will not be run, and in some cases may be automatically deleted.
- If the named node does not have the resources to accommodate the pod, the pod will fail and its reason will indicate why, for example OutOfmemory or OutOfcpu.
- Node names in cloud environments are not always predictable or stable.

Here is an example of a pod config file using the `nodeName` field:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
```

```
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

The above pod will run on the node kube-01.

What's next

Taints allow a Node to *repel* a set of Pods.

The design documents for node affinity and for inter-pod affinity/anti-affinity contain extra background information about these features.

Once a Pod is assigned to a Node, the kubelet runs the Pod and allocates node-local resources. The topology manager can take part in node-level resource allocation decisions.

3 - Resource Bin Packing for Extended Resources

FEATURE STATE: `Kubernetes v1.16 [alpha]`

The kube-scheduler can be configured to enable bin packing of resources along with extended resources using `RequestedToCapacityRatioResourceAllocation` priority function. Priority functions can be used to fine-tune the kube-scheduler as per custom needs.

Enabling Bin Packing using RequestedToCapacityRatioResourceAllocation

Kubernetes allows the users to specify the resources along with weights for each resource to score nodes based on the request to capacity ratio. This allows users to bin pack extended resources by using appropriate parameters and improves the utilization of scarce resources in large clusters. The behavior of the `RequestedToCapacityRatioResourceAllocation` priority function can be controlled by a configuration option called `requestedToCapacityRatioArguments`. This argument consists of two parameters `shape` and `resources`. The `shape` parameter allows the user to tune the function as least requested or most requested based on `utilization` and `score` values. The `resources` parameter consists of `name` of the resource to be considered during scoring and `weight` specify the weight of each resource.

Below is an example configuration that sets `requestedToCapacityRatioArguments` to bin packing behavior for extended resources `intel.com/foo` and `intel.com/bar`.

```yaml
apiVersion: v1
kind: Policy
# ...
priorities:
  # ...
  - name: RequestedToCapacityRatioPriority
    weight: 2
    argument:
      requestedToCapacityRatioArguments:
        shape:
          - utilization: 0
            score: 0
          - utilization: 100
            score: 10
        resources:
          - name: intel.com/foo
            weight: 3
          - name: intel.com/bar
            weight: 5
```

This feature is disabled by default

Tuning the Priority Function

`shape` is used to specify the behavior of the `RequestedToCapacityRatioPriority` function.

```yaml
shape:
 - utilization: 0
   score: 0
 - utilization: 100
   score: 10
```

The above arguments give the node a `score` of 0 if `utilization` is 0% and 10 for `utilization` 100%, thus enabling bin packing behavior. To enable least requested the score value must be reversed as follows.

```yaml
shape:
  - utilization: 0
    score: 10
  - utilization: 100
    score: 0
```

`resources` is an optional parameter which defaults to:

```yaml
resources:
  - name: CPU
    weight: 1
  - name: Memory
    weight: 1
```

It can be used to add extended resources as follows:

```yaml
resources:
  - name: intel.com/foo
    weight: 5
  - name: CPU
    weight: 3
  - name: Memory
    weight: 1
```

The `weight` parameter is optional and is set to 1 if not specified. Also, the `weight` cannot be set to a negative value.

Node scoring for capacity allocation

This section is intended for those who want to understand the internal details of this feature. Below is an example of how the node score is calculated for a given set of values.

Requested resources:

```
intel.com/foo : 2
Memory: 256MB
CPU: 2
```

Resource weights:

```
intel.com/foo : 5
Memory: 1
CPU: 3
```

FunctionShapePoint {{0, 0}, {100, 10}}

Node 1 spec:

```
Available:
  intel.com/foo: 4
  Memory: 1 GB
  CPU: 8

Used:
  intel.com/foo: 1
  Memory: 256MB
  CPU: 1
```

Node score:

```
intel.com/foo  = resourceScoringFunction((2+1),4)
               = (100 - ((4-3)*100/4)
               = (100 - 25)
               = 75                        # requested + used = 75% * available
               = rawScoringFunction(75)
               = 7                         # floor(75/10)

Memory         = resourceScoringFunction((256+256),1024)
               = (100 -((1024-512)*100/1024))
               = 50                        # requested + used = 50% * available
               = rawScoringFunction(50)
               = 5                         # floor(50/10)

CPU            = resourceScoringFunction((2+1),8)
               = (100 -((8-3)*100/8))
               = 37.5                      # requested + used = 37.5% * available
               = rawScoringFunction(37.5)
               = 3                         # floor(37.5/10)

NodeScore   =  (7 * 5) + (5 * 1) + (3 * 3) / (5 + 1 + 3)
            =  5
```

Node 2 spec:

```
Available:
  intel.com/foo: 8
  Memory: 1GB
  CPU: 8
Used:
  intel.com/foo: 2
  Memory: 512MB
  CPU: 6
```

Node score:

```
intel.com/foo  = resourceScoringFunction((2+2),8)
               =  (100 - ((8-4)*100/8)
               =  (100 - 50)
               =  50
               =  rawScoringFunction(50)
               = 5

Memory         = resourceScoringFunction((256+512),1024)
               = (100 -((1024-768)*100/1024))
               = 75
               = rawScoringFunction(75)
               = 7

CPU            = resourceScoringFunction((2+6),8)
               = (100 -((8-8)*100/8))
               = 100
               = rawScoringFunction(100)
               = 10

NodeScore   =  (5 * 5) + (7 * 1) + (10 * 3) / (5 + 1 + 3)
            =  7
```

What's next

- Read more about the scheduling framework
- Read more about scheduler configuration

4 - Taints and Tolerations

Node affinity, is a property of Pods that *attracts* them to a set of nodes (either as a preference or a hard requirement). *Taints* are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

Concepts

You add a taint to a node using kubectl taint. For example,

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

places a taint on node `node1`. The taint has key `key1`, value `value1`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto `node1` unless it has a matching toleration.

To remove the taint added by the command above, you can run:

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the `kubectl taint` line above, and thus a pod with either toleration would be able to schedule onto `node1`:

```yaml
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

```yaml
tolerations:
- key: "key1"
  operator: "Exists"
  effect: "NoSchedule"
```

Here's an example of a pod that uses tolerations:

pods/pod-with-toleration.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "example-key"
    operator: "Exists"
    effect: "NoSchedule"
```

The default value for `operator` is `Equal`.

A toleration "matches" a taint if the keys are the same and the effects are the same, and:

- the `operator` is `Exists` (in which case no `value` should be specified), or
- the `operator` is `Equal` and the `value` s are equal.

> **Note:**
> There are two special cases:
>
> An empty `key` with operator `Exists` matches all keys, values and effects which means this will tolerate everything.
>
> An empty `effect` matches all effects with key `key1`.

The above example used `effect` of `NoSchedule`. Alternatively, you can use `effect` of `PreferNoSchedule`. This is a "preference" or "soft" version of `NoSchedule` -- the system will *try* to avoid placing a pod that does not tolerate the taint on the node, but it is not required. The third kind of `effect` is `NoExecute`, described later.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

- if there is at least one un-ignored taint with effect `NoSchedule` then Kubernetes will not schedule the pod onto that node
- if there is no un-ignored taint with effect `NoSchedule` but there is at least one un-ignored taint with effect `PreferNoSchedule` then Kubernetes will *try* to not schedule the pod onto the node
- if there is at least one un-ignored taint with effect `NoExecute` then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

For example, imagine you taint a node like this

```
kubectl taint nodes node1 key1=value1:NoSchedule
kubectl taint nodes node1 key1=value1:NoExecute
kubectl taint nodes node1 key2=value2:NoSchedule
```

And a pod has two tolerations:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

In this case, the pod will not be able to schedule onto the node, because there is no toleration matching the third taint. But it will be able to continue running if it is already running on the node when the taint is added, because the third taint is the only one of the three that is not tolerated by the pod.

Normally, if a taint with effect `NoExecute` is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted. However, a toleration with `NoExecute` effect can specify an optional `tolerationSeconds` field that dictates how long the pod will stay bound to the node after the taint is added. For example,

```yaml
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

means that if this pod is running and a matching taint is added to the node, then the pod will stay bound to the node for 3600 seconds, and then be evicted. If the taint is removed before that time, the pod will not be evicted.

Example Use Cases

Taints and tolerations are a flexible way to steer pods *away* from nodes or evict pods that shouldn't be running. A few of the use cases are

- **Dedicated Nodes**: If you want to dedicate a set of nodes for exclusive use by a particular set of users, you can add a taint to those nodes (say, `kubectl taint nodes nodename dedicated=groupName:NoSchedule`) and then add a corresponding toleration to their pods (this would be done most easily by writing a custom admission controller). The pods with the tolerations will then be allowed to use the tainted (dedicated) nodes as well as any other nodes in the cluster. If you want to dedicate the nodes to them *and* ensure they *only* use the dedicated nodes, then you should additionally add a label similar to the taint to the same set of nodes (e.g. `dedicated=groupName`), and the admission controller should additionally add a node affinity to require that the pods can only schedule onto nodes labeled with `dedicated=groupName`.
- **Nodes with Special Hardware**: In a cluster where a small subset of nodes have specialized hardware (for example GPUs), it is desirable to keep pods that don't need the specialized hardware off of those nodes, thus leaving room for later-arriving pods that do need the specialized hardware. This can be done by tainting the nodes that have the specialized hardware (e.g. `kubectl taint nodes nodename special=true:NoSchedule` or `kubectl taint nodes nodename special=true:PreferNoSchedule`) and adding a corresponding toleration to pods that use the special hardware. As in the dedicated nodes use case, it is probably easiest to apply the tolerations using a custom admission controller. For example, it is recommended to use Extended Resources to represent the special hardware, taint your special hardware nodes with the extended resource name and run the ExtendedResourceToleration admission controller. Now, because the nodes are tainted, no pods without the toleration will schedule on them. But when you submit a pod that requests the extended resource, the `ExtendedResourceToleration` admission controller will automatically add the correct toleration to the pod and that pod will schedule on the special hardware nodes. This will make sure that these special hardware nodes are dedicated for pods requesting such hardware and you don't have to manually add tolerations to your pods.
- **Taint based Evictions**: A per-pod-configurable eviction behavior when there are node problems, which is described in the next section.

Taint based Evictions

FEATURE STATE: `Kubernetes v1.18 [stable]`

The `NoExecute` taint effect, mentioned above, affects pods that are already running on the node as follows

- pods that do not tolerate the taint are evicted immediately
- pods that tolerate the taint without specifying `tolerationSeconds` in their toleration specification remain bound forever
- pods that tolerate the taint with a specified `tolerationSeconds` remain bound for the specified amount of time

The node controller automatically taints a Node when certain conditions are true. The following taints are built in:

- `node.kubernetes.io/not-ready`: Node is not ready. This corresponds to the NodeCondition `Ready` being "`False`".
- `node.kubernetes.io/unreachable`: Node is unreachable from the node controller. This corresponds to the NodeCondition `Ready` being "`Unknown`".
- `node.kubernetes.io/out-of-disk`: Node becomes out of disk.
- `node.kubernetes.io/memory-pressure`: Node has memory pressure.
- `node.kubernetes.io/disk-pressure`: Node has disk pressure.
- `node.kubernetes.io/network-unavailable`: Node's network is unavailable.
- `node.kubernetes.io/unschedulable`: Node is unschedulable.
- `node.cloudprovider.kubernetes.io/uninitialized`: When the kubelet is started with "external" cloud provider, this taint is set on a node to mark it as unusable. After a controller from the cloud-controller-manager initializes this node, the kubelet removes this taint.

In case a node is to be evicted, the node controller or the kubelet adds relevant taints with `NoExecute` effect. If the fault condition returns to normal the kubelet or node controller can remove the relevant taint(s).

Note: The control plane limits the rate of adding node new taints to nodes. This rate limiting manages the number of evictions that are triggered when many nodes become unreachable at once (for example: if there is a network disruption).

You can specify `tolerationSeconds` for a Pod to define how long that Pod stays bound to a failing or unresponsive Node.

For example, you might want to keep an application with a lot of local state bound to node for a long time in the event of network partition, hoping that the partition will recover and thus the pod eviction can be avoided. The toleration you set for that Pod might look like:

```yaml
tolerations:
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

Note:
Kubernetes automatically adds a toleration for `node.kubernetes.io/not-ready` and `node.kubernetes.io/unreachable` with `tolerationSeconds=300`, unless you, or a controller, set those tolerations explicitly.

These automatically-added tolerations mean that Pods remain bound to Nodes for 5 minutes after one of these problems is detected.

DaemonSet pods are created with `NoExecute` tolerations for the following taints with no `tolerationSeconds`:

- `node.kubernetes.io/unreachable`
- `node.kubernetes.io/not-ready`

This ensures that DaemonSet pods are never evicted due to these problems.

Taint Nodes by Condition

The node lifecycle controller automatically creates taints corresponding to Node conditions with `NoSchedule` effect. Similarly the scheduler does not check Node conditions; instead the scheduler checks taints. This assures that Node conditions don't affect what's scheduled onto the Node. The user can choose to ignore some of the Node's problems (represented as Node conditions) by adding appropriate Pod tolerations.

The DaemonSet controller automatically adds the following `NoSchedule` tolerations to all daemons, to prevent DaemonSets from breaking.

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/out-of-disk` (*only for critical pods*)
- `node.kubernetes.io/unschedulable` (1.10 or later)
- `node.kubernetes.io/network-unavailable` (*host network only*)

Adding these tolerations ensures backward compatibility. You can also add arbitrary tolerations to DaemonSets.

What's next

- Read about out of resource handling and how you can configure it
- Read about pod priority

5 - Pod Overhead

FEATURE STATE: `Kubernetes v1.18 [beta]`

When you run a Pod on a Node, the Pod itself takes an amount of system resources. These resources are additional to the resources needed to run the container(s) inside the Pod. *Pod Overhead* is a feature for accounting for the resources consumed by the Pod infrastructure on top of the container requests & limits.

In Kubernetes, the Pod's overhead is set at admission time according to the overhead associated with the Pod's RuntimeClass.

When Pod Overhead is enabled, the overhead is considered in addition to the sum of container resource requests when scheduling a Pod. Similarly, the kubelet will include the Pod overhead when sizing the Pod cgroup, and when carrying out Pod eviction ranking.

Enabling Pod Overhead

You need to make sure that the `PodOverhead` feature gate is enabled (it is on by default as of 1.18) across your cluster, and a `RuntimeClass` is utilized which defines the `overhead` field.

Usage example

To use the PodOverhead feature, you need a RuntimeClass that defines the `overhead` field. As an example, you could use the following RuntimeClass definition with a virtualizing container runtime that uses around 120MiB per Pod for the virtual machine and the guest OS:

```yaml
---
kind: RuntimeClass
apiVersion: node.k8s.io/v1
metadata:
    name: kata-fc
handler: kata-fc
overhead:
    podFixed:
        memory: "120Mi"
        cpu: "250m"
```

Workloads which are created which specify the `kata-fc` RuntimeClass handler will take the memory and cpu overheads into account for resource quota calculations, node scheduling, as well as Pod cgroup sizing.

Consider running the given example workload, test-pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  runtimeClassName: kata-fc
  containers:
  - name: busybox-ctr
    image: busybox
    stdin: true
    tty: true
    resources:
      limits:
        cpu: 500m
        memory: 100Mi
  - name: nginx-ctr
    image: nginx
    resources:
      limits:
        cpu: 1500m
        memory: 100Mi
```

At admission time the RuntimeClass admission controller updates the workload's PodSpec to include the `overhead` as described in the RuntimeClass. If the PodSpec already has this field defined, the Pod will be rejected. In the given example, since only the RuntimeClass name is specified, the admission controller mutates the Pod to include an `overhead`.

After the RuntimeClass admission controller, you can check the updated PodSpec:

```
kubectl get pod test-pod -o jsonpath='{.spec.overhead}'
```

The output is:

```
map[cpu:250m memory:120Mi]
```

If a ResourceQuota is defined, the sum of container requests as well as the `overhead` field are counted.

When the kube-scheduler is deciding which node should run a new Pod, the scheduler considers that Pod's `overhead` as well as the sum of container requests for that Pod. For this example, the scheduler adds the requests and the overhead, then looks for a node that has 2.25 CPU and 320 MiB of memory available.

Once a Pod is scheduled to a node, the kubelet on that node creates a new cgroup for the Pod. It is within this pod that the underlying container runtime will create containers.

If the resource has a limit defined for each container (Guaranteed QoS or Bustrable QoS with limits defined), the kubelet will set an upper limit for the pod cgroup associated with that resource (cpu.cfs_quota_us for CPU and memory.limit_in_bytes memory). This upper limit is based on the sum of the container limits plus the `overhead` defined in the PodSpec.

For CPU, if the Pod is Guaranteed or Burstable QoS, the kubelet will set `cpu.shares` based on the sum of container requests plus the `overhead` defined in the PodSpec.

Looking at our example, verify the container requests for the workload:

```
kubectl get pod test-pod -o jsonpath='{.spec.containers[*].resources.limits}'
```

The total container requests are 2000m CPU and 200MiB of memory:

```
map[cpu: 500m memory:100Mi] map[cpu:1500m memory:100Mi]
```

Check this against what is observed by the node:

```
kubectl describe node | grep test-pod -B2
```

The output shows 2250m CPU and 320MiB of memory are requested, which includes PodOverhead:

```
  Namespace                   Name                CPU Requests  CPU Limits   Memory
  ---------                   ----                ------------  ----------   -------
  default                     test-pod            2250m (56%)   2250m (56%)  320Mi (
```

Verify Pod cgroup limits

Check the Pod's memory cgroups on the node where the workload is running. In the following example, `crictl` is used on the node, which provides a CLI for CRI-compatible container runtimes. This is an advanced example to show PodOverhead behavior, and it is not expected that users should need to check cgroups directly on the node.

First, on the particular node, determine the Pod identifier:

```
# Run this on the node where the Pod is scheduled
POD_ID="$(sudo crictl pods --name test-pod -q)"
```

From this, you can determine the cgroup path for the Pod:

```
# Run this on the node where the Pod is scheduled
sudo crictl inspectp -o=json $POD_ID | grep cgroupsPath
```

The resulting cgroup path includes the Pod's `pause` container. The Pod level cgroup is one directory above.

```
        "cgroupsPath": "/kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2/7ccf55aee3
```

In this specific case, the pod cgroup path is `kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2`. Verify the Pod level cgroup setting for memory:

```
# Run this on the node where the Pod is scheduled.
# Also, change the name of the cgroup to match the cgroup allocated for your pod.
 cat /sys/fs/cgroup/memory/kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2/memory.
```

This is 320 MiB, as expected:

```
335544320
```

Observability

A `kube_pod_overhead` metric is available in kube-state-metrics to help identify when PodOverhead is being utilized and to help observe stability of workloads running with a defined Overhead. This functionality is not available in the 1.9 release of kube-state-metrics, but is expected in a following release. Users will need to build kube-state-metrics from source in the meantime.

What's next

- RuntimeClass
- PodOverhead Design

6 - Eviction Policy

This page is an overview of Kubernetes' policy for eviction.

Eviction Policy

The kubelet proactively monitors for and prevents total starvation of a compute resource. In those cases, the `kubelet` can reclaim the starved resource by failing one or more Pods. When the `kubelet` fails a Pod, it terminates all of its containers and transitions its `PodPhase` to `Failed`. If the evicted Pod is managed by a Deployment, the Deployment creates another Pod to be scheduled by Kubernetes.

What's next

- Learn how to configure out of resource handling with eviction signals and thresholds.

7 - Scheduling Framework

FEATURE STATE: `Kubernetes v1.15 [alpha]`

The scheduling framework is a pluggable architecture for the Kubernetes scheduler. It adds a new set of "plugin" APIs to the existing scheduler. Plugins are compiled into the scheduler. The APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" lightweight and maintainable. Refer to the design proposal of the scheduling framework for more technical information on the design of the framework.

Framework workflow

The Scheduling Framework defines a few extension points. Scheduler plugins register to be invoked at one or more extension points. Some of these plugins can change the scheduling decisions and some are informational only.

Each attempt to schedule one Pod is split into two phases, the **scheduling cycle** and the **binding cycle**.

Scheduling Cycle & Binding Cycle

The scheduling cycle selects a node for the Pod, and the binding cycle applies that decision to the cluster. Together, a scheduling cycle and binding cycle are referred to as a "scheduling context".

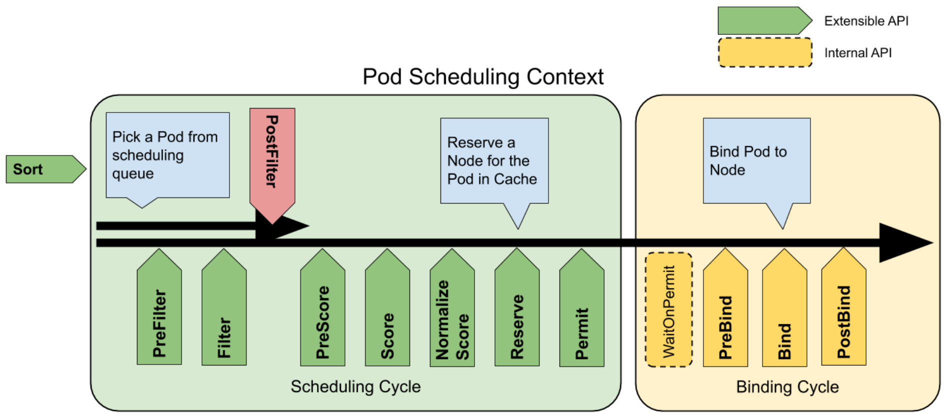
Scheduling cycles are run serially, while binding cycles may run concurrently.

A scheduling or binding cycle can be aborted if the Pod is determined to be unschedulable or if there is an internal error. The Pod will be returned to the queue and retried.

Extension points

The following picture shows the scheduling context of a Pod and the extension points that the scheduling framework exposes. In this picture "Filter" is equivalent to "Predicate" and "Scoring" is equivalent to "Priority function".

One plugin may register at multiple extension points to perform more complex or stateful tasks.



scheduling framework extension points

QueueSort

These plugins are used to sort Pods in the scheduling queue. A queue sort plugin essentially provides a `Less(Pod1, Pod2)` function. Only one queue sort plugin may be enabled at a time.

PreFilter

These plugins are used to pre-process info about the Pod, or to check certain conditions that the cluster or the Pod must meet. If a PreFilter plugin returns an error, the scheduling cycle is aborted.

Filter

These plugins are used to filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently.

PostFilter

These plugins are called after Filter phase, but only when no feasible nodes were found for the pod. Plugins are called in their configured order. If any postFilter plugin marks the node as `Schedulable`, the remaining plugins will not be called. A typical PostFilter implementation is preemption, which tries to make the pod schedulable by preempting other Pods.

PreScore

These plugins are used to perform "pre-scoring" work, which generates a sharable state for Score plugins to use. If a PreScore plugin returns an error, the scheduling cycle is aborted.

Score

These plugins are used to rank nodes that have passed the filtering phase. The scheduler will call each scoring plugin for each node. There will be a well defined range of integers representing the minimum and maximum scores. After the NormalizeScore phase, the scheduler will combine node scores from all plugins according to the configured plugin weights.

NormalizeScore

These plugins are used to modify scores before the scheduler computes a final ranking of Nodes. A plugin that registers for this extension point will be called with the Score results from the same plugin. This is called once per plugin per scheduling cycle.

For example, suppose a plugin `BlinkingLightScorer` ranks Nodes based on how many blinking lights they have.

```go
func ScoreNode(_ *v1.pod, n *v1.Node) (int, error) {
    return getBlinkingLightCount(n)
}
```

However, the maximum count of blinking lights may be small compared to `NodeScoreMax`. To fix this, `BlinkingLightScorer` should also register for this extension point.

```go
func NormalizeScores(scores map[string]int) {
    highest := 0
    for _, score := range scores {
        highest = max(highest, score)
    }
    for node, score := range scores {
        scores[node] = score*NodeScoreMax/highest
    }
}
```

If any NormalizeScore plugin returns an error, the scheduling cycle is aborted.

Note: Plugins wishing to perform "pre-reserve" work should use the NormalizeScore extension point.

Reserve

A plugin that implements the Reserve extension has two methods, namely `Reserve` and `Unreserve`, that back two informational scheduling phases called Reserve and Unreserve, respectively. Plugins which maintain runtime state (aka "stateful plugins") should use these phases to be notified by the scheduler when resources on a node are being reserved and unreserved for a given Pod.

The Reserve phase happens before the scheduler actually binds a Pod to its designated node. It exists to prevent race conditions while the scheduler waits for the bind to succeed. The `Reserve` method of each Reserve plugin may succeed or fail; if one `Reserve` method call fails, subsequent plugins are not executed and the Reserve phase is considered to have failed. If the `Reserve` method of all plugins succeed, the Reserve phase is considered to be successful and the rest of the scheduling cycle and the binding cycle are executed.

The Unreserve phase is triggered if the Reserve phase or a later phase fails. When this happens, the `Unreserve` method of **all** Reserve plugins will be executed in the reverse order of `Reserve` method calls. This phase exists to clean up the state associated with the reserved Pod.

Caution: The implementation of the `Unreserve` method in Reserve plugins must be idempotent and may not fail.

Permit

Permit plugins are invoked at the end of the scheduling cycle for each Pod, to prevent or delay the binding to the candidate node. A permit plugin can do one of the three things:

1. **approve**
   Once all Permit plugins approve a Pod, it is sent for binding.

2. **deny**
   If any Permit plugin denies a Pod, it is returned to the scheduling queue. This will trigger the Unreserve phase in Reserve plugins.

3. **wait** (with a timeout)
   If a Permit plugin returns "wait", then the Pod is kept in an internal "waiting" Pods list, and the binding cycle of this Pod starts but directly blocks until it gets approved. If a timeout occurs, **wait** becomes **deny** and the Pod is returned to the scheduling queue, triggering the Unreserve phase in Reserve plugins.

Note: While any plugin can access the list of "waiting" Pods and approve them (see `FrameworkHandle`), we expect only the permit plugins to approve binding of reserved Pods that are in "waiting" state. Once a Pod is approved, it is sent to the PreBind phase.

PreBind

These plugins are used to perform any work required before a Pod is bound. For example, a pre-bind plugin may provision a network volume and mount it on the target node before allowing the Pod to run there.

If any PreBind plugin returns an error, the Pod is rejected and returned to the scheduling queue.

Bind

These plugins are used to bind a Pod to a Node. Bind plugins will not be called until all PreBind plugins have completed. Each bind plugin is called in the configured order. A bind plugin may choose whether or not to handle the given Pod. If a bind plugin chooses to handle a Pod, **the remaining bind plugins are skipped**.

PostBind

This is an informational extension point. Post-bind plugins are called after a Pod is successfully bound. This is the end of a binding cycle, and can be used to clean up associated resources.

Plugin API

There are two steps to the plugin API. First, plugins must register and get configured, then they use the extension point interfaces. Extension point interfaces have the following form.

```go
type Plugin interface {
    Name() string
}

type QueueSortPlugin interface {
    Plugin
    Less(*v1.pod, *v1.pod) bool
}

type PreFilterPlugin interface {
    Plugin
    PreFilter(context.Context, *framework.CycleState, *v1.pod) error
}

// ...
```

Plugin configuration

You can enable or disable plugins in the scheduler configuration. If you are using Kubernetes v1.18 or later, most scheduling plugins are in use and enabled by default.

In addition to default plugins, you can also implement your own scheduling plugins and get them configured along with default plugins. You can visit scheduler-plugins for more details.

If you are using Kubernetes v1.18 or later, you can configure a set of plugins as a scheduler profile and then define multiple profiles to fit various kinds of workload. Learn more at multiple profiles.

8 - Scheduler Performance Tuning

FEATURE STATE: `Kubernetes v1.14 [beta]`

kube-scheduler is the Kubernetes default scheduler. It is responsible for placement of Pods on Nodes in a cluster.

Nodes in a cluster that meet the scheduling requirements of a Pod are called *feasible* Nodes for the Pod. The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes, picking a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *Binding*.

This page explains performance tuning optimizations that are relevant for large Kubernetes clusters.

In large clusters, you can tune the scheduler's behaviour balancing scheduling outcomes between latency (new Pods are placed quickly) and accuracy (the scheduler rarely makes poor placement decisions).

You configure this tuning setting via kube-scheduler setting `percentageOfNodesToScore`. This KubeSchedulerConfiguration setting determines a threshold for scheduling nodes in your cluster.

Setting the threshold

The `percentageOfNodesToScore` option accepts whole numeric values between 0 and 100. The value 0 is a special number which indicates that the kube-scheduler should use its compiled-in default. If you set `percentageOfNodesToScore` above 100, kube-scheduler acts as if you had set a value of 100.

To change the value, edit the kube-scheduler configuration file (this is likely to be `/etc/kubernetes/config/kube-scheduler.yaml`), then restart the scheduler.

After you have made this change, you can run

```
kubectl get pods -n kube-system | grep kube-scheduler
```

to verify that the kube-scheduler component is healthy.

Node scoring threshold

To improve scheduling performance, the kube-scheduler can stop looking for feasible nodes once it has found enough of them. In large clusters, this saves time compared to a naive approach that would consider every node.

You specify a threshold for how many nodes are enough, as a whole number percentage of all the nodes in your cluster. The kube-scheduler converts this into an integer number of nodes. During scheduling, if the kube-scheduler has identified enough feasible nodes to exceed the configured percentage, the kube-scheduler stops searching for more feasible nodes and moves on to the scoring phase.

How the scheduler iterates over Nodes describes the process in detail.

Default threshold

If you don't specify a threshold, Kubernetes calculates a figure using a linear formula that yields 50% for a 100-node cluster and yields 10% for a 5000-node cluster. The lower bound for the automatic value is 5%.

This means that, the kube-scheduler always scores at least 5% of your cluster no matter how large the cluster is, unless you have explicitly set `percentageOfNodesToScore` to be smaller than 5.

If you want the scheduler to score all nodes in your cluster, set `percentageOfNodesToScore` to 100.

Example

Below is an example configuration that sets `percentageOfNodesToScore` to 50%.

```yaml
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

percentageOfNodesToScore: 50
```

Tuning percentageOfNodesToScore

`percentageOfNodesToScore` must be a value between 1 and 100 with the default value being calculated based on the cluster size. There is also a hardcoded minimum value of 50 nodes.

Note:
In clusters with less than 50 feasible nodes, the scheduler still checks all the nodes because there are not enough feasible nodes to stop the scheduler's search early.

In a small cluster, if you set a low value for `percentageOfNodesToScore`, your change will have no or little effect, for a similar reason.

If your cluster has several hundred Nodes or fewer, leave this configuration option at its default value. Making changes is unlikely to improve the scheduler's performance significantly.

An important detail to consider when setting this value is that when a smaller number of nodes in a cluster are checked for feasibility, some nodes are not sent to be scored for a given Pod. As a result, a Node which could possibly score a higher value for running the given Pod might not even be passed to the scoring phase. This would result in a less than ideal placement of the Pod.

You should avoid setting `percentageOfNodesToScore` very low so that kube-scheduler does not make frequent, poor Pod placement decisions. Avoid setting the percentage to anything below 10%, unless the scheduler's throughput is critical for your application and the score of nodes is not important. In other words, you prefer to run the Pod on any Node as long as it is feasible.

How the scheduler iterates over Nodes

This section is intended for those who want to understand the internal details of this feature.

In order to give all the Nodes in a cluster a fair chance of being considered for running Pods, the scheduler iterates over the nodes in a round robin fashion. You can imagine that Nodes are in an array. The scheduler starts from the start of the array and checks feasibility of the nodes until it finds enough Nodes as specified by `percentageOfNodesToScore`. For the next Pod, the scheduler continues from the point in the Node array that it stopped at when checking feasibility of Nodes for the previous Pod.

If Nodes are in multiple zones, the scheduler iterates over Nodes in various zones to ensure that Nodes from different zones are considered in the feasibility checks. As an example, consider six nodes in two zones:

```
Zone 1: Node 1, Node 2, Node 3, Node 4
Zone 2: Node 5, Node 6
```

The Scheduler evaluates feasibility of the nodes in this order:

```
Node 1, Node 5, Node 2, Node 6, Node 3, Node 4
```

After going over all the Nodes, it goes back to Node 1.