ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/pallavi-priya-patharlagadda/236

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science
https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 4

Research Article

Enhance the Application Performance by Customizing Kubernetes Scheduling

Pallavi Priya Patharlagadda*

Citation: Patharlagadda PP. Gorm DB Deep Dive and Techniques to Update Default Values to DB. *J Artif Intell Mach Learn & Data Sci 2023*, 1(4), 995-998. DOI: doi.org/10.51219/JAIMLD/pallavi-priya-patharlagadda/236

Received: 03 December, 2023; **Accepted:** 28 December, 2023; **Published:** 30 December, 2023

***Corresponding author:** Pallavi Priya Patharlagadda, USA, E-mail: Pallavipriya527.p@gmail.com

ABSTRACT

Kubernetes is a powerful container orchestration platform that provides a way to automate the deployment, scaling, and management of containerized applications. One of the most important aspects of Kubernetes cluster management is pod assignment to nodes. Although the default procedure may seem overly general, you may modify it by utilizing sophisticated features such as node affinity. Node selector, node affinity and anti-affinity, and pod affinity are among the fundamental Kubernetes scheduling principles covered in this article. It also provides an example of how you may increase the availability and fault tolerance of your workload by combining automation and node affinity.

1. Introduction

The Kubernetes scheduler's method of allocating pods among worker nodes affects resources and performance, which in turn affects how much you spend. Then, knowing how the process operates and how to maintain it is crucial. By default, Kubernetes distributes pods haphazardly among the cluster's accessible nodes. Nonetheless, there are numerous situations in which you might have to manage where your pods are placed on particular nodes. For instance, you might want to avoid putting pods on nodes that are handling other crucial workloads or only install pods that require particular hardware resources (like GPUs) on nodes that have those resources available. Many capabilities are available in Kubernetes for managing the placement of pods, such as node selectors, affinity and anti-affinity rules, taints, and tolerances. In the subsequent sections, we will go over the various methods for advanced pod scheduling in Kubernetes and give examples of how to apply them to address typical use cases.

Applications of Kubernetes Pod-to-Node Scheduling:

Often, in a Kubernetes system, the scheduling of pods to nodes needs to be customized. The following are some of the most typical situations in which advanced pod scheduling proves advantageous:

Pods running on nodes equipped with specialized hardware: Certain Kubernetes applications can need particular hardware. Elasticsearch pods may function better on SSDs than HDDs, while pods executing machine learning tasks might need high-performance GPUs rather than CPUs. Therefore, assigning pods to nodes with the right hardware are the recommended practice for any resource-aware Kubernetes cluster management strategy.

Pod colocation and codependency: To enhance speed, minimize network latency, and avoid connection failures, it could be essential to co-locate specific pods on the same server in a microservices environment or a tightly connected application stack. One common recommendation is to run a web server alongside an in-memory caching service or database on the same system.

Data locality: Requirements for data locality in data-intensive applications could be comparable to those in the use case before it. It may be necessary for these applications to have the databases installed on the same system as the customer-facing application to guarantee quicker reads and improved write performance.

High availability and fault tolerance: Running pods on nodes deployed in different availability zones is a smart strategy to make application deployments extremely available and fault-tolerant.

Node Taints and Pods Tolerations:

In a Kubernetes cluster, taints and tolerations offer a potent method for managing pod distribution among certain nodes. The idea is straightforward but powerful: A toleration permits a pod to withstand the effects of taints and be scheduled on particular nodes, whereas a taint restricts a node by determining which pods can or cannot be scheduled on it.

Taints: A taint is a pair of keys that indicate a node condition and its consequence. NoSchedule or PreferNoSchedule are the possible outcomes. NoSchedule taints prohibit the scheduling of any pod on the node that does not have a corresponding toleration. While not stopping it, a PreferNoSchedule taint instructs the scheduler to steer clear of scheduling pods on the node.

You can use the kubectl taint command to taint the nodes.

```
kubectl taint nodes <node-name> <key>=<value>:<taint-effect>
```

Tolerations: A toleration is a pair of keys and values that define a node condition and its consequence. *NoExecute* or *Effect* are the two possible outcomes. A node with a matching taint protects a pod from eviction under *NoExecute* toleration. Even in cases when a pod lacks toleration for a particular taint, it is still possible to schedule it on a node that has a corresponding taint thanks to an effect tolerance.

There are 3 pre-defined effects as below:

- **NoSchedule:** Do not place the pods unless they can tolerate the taint
- **PreferNoSchedule:** Try to avoid scheduling the pods that cannot tolerate the taint. Not guaranteed.
- **NoExecute:** If the pods can't handle the taint by the time it's enabled on the nodes, they will be killed.

One way to do this would be to design a situation in which a specific node can only host pods that have essential services, like controllers. Taints and tolerations are easy to implement. To begin with, taint a node that requires the application of non-standard scheduling behavior. As an illustration:

```
kubectl taint nodes node01 critical=true: NoSchedule
node "node01" tainted
```

The configuration process does not end with the creation of a taint. We must include the following toleration to schedule pods on a compromised node:

```
apiVersion: v1
metadata:
 name: taint-toleration
spec:
 containers:
 - name: taint-toleration
 image: nginx
 resources:
 requests:
 cpu: 0.8
 memory: 4Gi
 limits:
 cpu: 3.0
 memory: 22Gi
 tolerations:
 - key: "example"
 operator: "Exists"
 effect: "NoSchedule"
```

In this case, I used the "Exists" operator to apply the tolerance for the aforementioned taint. Alternatively, I might apply tolerance to any node that matches the taint's key by using the "EQUAL" operator. But the value need to be specified. It's crucial to keep in mind that toleration does not ensure that the pod will only be positioned in the contaminated node. It is possible to insert the aforementioned pod into the uncontaminated nodes and allow them to receive any pods if the other nodes are uncontaminated.

Selecting a Node by a Pod: nodeName, nodeSelector, and nodeAffinity

An alternative method involves setting up a Pod so that "it" chooses the Node it will operate on.

For this, we have nodeName, nodeSelector, nodeAffinity, and nodeAntiAffinity.

nodeName: The easiest method. Takes precedence over everything else:

```
apiVersion: v1
kind: Pod
metadata:
 name: nginx
spec:
 containers:
 - name: sample-nginx
image: nginx: latest
nodeName: node01
```

NodeSelector is essentially a label-based pod-to-node scheduling technique in which users tag nodes with specific labels and ensure that the nodeSelector field reflects those labels. To illustrate the kind of storage on the node, let's say that one of the node labels is "storage=ssd."

```
kubectl describe node "node01"
Name: node01
Roles: node
Labels: critical=true,
```

I'll designate the nodeSelector field in the Pod manifest with that label to schedule pods onto the node with that label.

```
apiVersion: v1
kind: Pod
metadata:
name: nginx
labels:
 env: dev
spec:
containers:
– name: my-nginx
 image: nginx:latest
 imagePullPolicy: IfNotPresent
nodeSelector:
critical: true
```

The most basic kind of advanced pod scheduling is node selectors. They are not particularly helpful, though, when additional guidelines and requirements need to be taken into account while scheduling pods.

nodeAffinity and nodeAntiAffinity: nodeAffinity and nodeAntiAffinityoperate in the same way as thenodeSelector, but have more flexible capabilities.

You can, for instance, establish hard or soft launch limitations. In the event of a soft limit, the scheduler will attempt to launch a Pod on the relevant Node and, failing that will launch it on a different Node. As a result, the Pod will stay in Pending status if you specify a hard limit and the scheduler is unable to start it on the chosen Node.

The hard limit is set in the field .spec.affinity.nodeAffinity with the requiredDuringSchedulingIgnoredDuringExecution, and the soft limit is set with the preferredDuringSchedulingIgnoredDuringExecution.

We deploy pods on nodes in particular availability zones using node affinity in the example below. Let's examine the manifest that is below:

```
apiVersion: v1
kind: Pod
metadata:
name: node-affinity
spec:
affinity:
 nodeAffinity:
 requiredDuringSchedulingIgnoredDuringExecution:
 nodeSelectorTerms:
 - matchExpressions:
 - key: kubernetes.io/zone
 operator: In
 values:
 - Westcoast-1a
 - Westcoast-1b
 preferredDuringSchedulingIgnoredDuringExecution:
 - weight: 1
 preference:
 matchExpressions:
 - key: another-node-key
 operator: In
 values:
 - another-node-value
containers:
– name: node-affinity
 image: app
```

The nodeAffinity part of the pod manifest contains the "required during scheduling ignored during execution" element, which specifies "hard" affinity restrictions. Using kubernetes.io/zone as the example key and values Westcoast-1a or Westcoast-1b for the label, I instructed the scheduler to only place the pod on nodes with that label.

We filtered the array of existing label values using the In logical operator to accomplish this. I may also use the operations NotIn, Exists, DoesNotExist, Gt, and Lt.

The "preferred during scheduling ignored during execution" element in the specification contains the details of the "soft" rule. This example indicates that I want to use nodes with a label that has a key named "custom-key" and a value named "custom-value" out of the nodes that satisfy the "hard" condition. I have no problem scheduling pods for other candidates if they match the "hard" requirements, though, if there are no such nodes.

Creating node affinity rules that combine "hard" and "soft" restrictions is a recommended practice. Deployment scheduling becomes more flexible and predictable by using this "best-effort" method, which is to use some option if possible but not reject scheduling if the option is not accessible.

podAffinity and podAntiAffinity:

You can modify Pod Affinity based on the labels that Pods that are now executing on the Node will have, much like you would when choosing a Node using hard and soft restrictions. Refer to anti-affinity and inter-pod affinity. Similar definitions apply to node and inter-pod affinity. But in this instance, I'll take advantage of the pod spec's podAffinity parameter.

```
apiVersion: v1
kind: Pod
metadata:
name: example-pod-affinity
spec:
affinity:
 podAffinity:
 requiredDuringSchedulingIgnoredDuringExecution:
 - labelSelector:
 matchExpressions:
 - key: security
 operator: In
 values:
 - S1
 topologyKey: kubernetes.io/zone
 containers:
– name: pod-affinity
 image: your-container
```

Pod affinity is compatible with logical operators and match expressions, just as node affinity. They are, however, used in this instance on the pods' label selectors that are operating on a specific node. A new pod is collocated with the target pod on the same system if the given phrase matches the target pod's pod label.

The pod anti-affinity feature allows pods to resist one another. As previously stated, distributing pods among several availability zones helps prevent a single point of failure in Kubernetes. In the pod spec's anti-affinity section, I can set up comparable behavior. To achieve pod anti-affinity, two pods are required:

The first pod:

```
apiVersion: v1
kind: Pod
metadata:
name: s1
labels:
 security: s1
spec:
containers:
– name: c1
 image: initial-img
```

Note that the first pod has the label "security: s1."

```
apiVersion: v1
kind: Pod
metadata:
name: s2
spec:
```

```
affinity:
 podAntiAffinity:
 requiredDuringSchedulingIgnoredDuringExecution:
 - labelSelector:
 matchExpressions:
 - key: security
 operator: In
 values:
 - s1
 topologyKey: kubernetes.io/hostname
containers:
– name: pod-anti-affinity
 image: second-image
```

Referred to under the spec.affinity, the second pod is the label selector security:s1.podAntiAffinity. This means that the node that currently hosts any pods with the label "security:s1" will not have this pod scheduled to it.

topologySpreadConstraints:

Initially, visualize a group of twenty nodes. A workload that scales its replica count automatically is what you want to run. You want to run those replicas on as many different nodes as you can because it can scale from two to twenty Pods. This method lessens the possibility that a node failure may impact workload.

Next, consider an application that has five Pods on each of three nodes in the same Availability Zone, and fifteen replicas operating on those nodes. Though customers interacting with the workload come from three different zones, you have reduced the danger of a node failure. However, traffic crossing different AZs leads to greater network costs and delays.

By distributing Pods among nodes in various AZs and directing clients to the instances inside the appropriate zone, you can lower them. To further reduce the chance of a failure impacting your Pods, deploy the workload over many zones and numerous nodes.

Generally speaking, you would want to split up the effort equally among all failure domains. Using the spec. topologySpreadConstraints field, you can set that up with pod topology constraints.

The operation of pod topology spread constraints

An illustration of a pod topology spread constraint is as follows:

```
apiVersion: v1
kind: Pod
metadata:
 name: example-pod
spec:
 # Configure a topology spread constraint
 topologySpreadConstraints:
 - maxSkew: <integer>
 minDomains: <integer> # optional;
 topologyKey: <string>
 whenUnsatisfiable: <string>
 labelSelector: <object>
 matchLabelKeys: <list> # optional;
 nodeAffinityPolicy: [Honor|Ignore] # optional;
 nodeTaintsPolicy: [Honor|Ignore] # optional;
```

Let's just quickly review the required fields for the time being:

- maxSkew is the extent to which all of your zones can have an unequal distribution of your Pods. It can't have a value of zero.
- The node labels' key is topologyKey. Nodes in the same topology are those that have the same labels and values. The scheduler attempts to allocate a balanced number of pods to each topology instance, which is a domain.
- When a Pod doesn't meet your spread requirement, whenUnsatisfiable gives you the option of what to do with it:

1. DoNotSchedule instructs the scheduler not to schedule it.
2. ScheduleAnyway tells the scheduler to schedule it and prioritize the nodes minimizing the skew.
    - It is possible to find matching Pods with labelSelector. The Pods that match the label selector determine how many Pods are in the relevant topology domain.

2. Conclusion

One effective way to raise the performance, availability, and resilience of your containerized applications is to employ Kubernetes' advanced pod scheduling feature. Through comprehension of the various functionalities offered and their utilization, you may manage the arrangement of your pods to suit the particular requirements of your application. For example, operational and maintenance pods can be scheduled on oam nodes and application pods can be scheduled on userplane node. So, the application traffic doesn't impact the operational and maintanence pods traffic. This also helps us in separating the userplane and control plane traffic.

3. References

1. https://medium.com/@seifeddinerajhi/pod-scheduling-in-kubernetes-control-the-placement-of-your-pods-%EF%B8%8F-db5add6f2803
2. https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
3. https://cast.ai/blog/node-affinity-and-other-ways-to-control-scheduler/
4. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/