Control node selection

Many Kubernetes cluster deployments have different kinds of nodes that have different CPU and memory resources available for scheduling cluster workloads. The Redis Enterprise operator has various abilities to control the scheduling Redis Enterprise cluster node pods through properties specified in the Redis Enterprise cluster custom resource definition (CRD).

A Redis Enterprise cluster (REC) is deployed as a StatefulSet which manages the Redis Enterprise cluster node pods. The scheduler chooses a node to deploy a new Redis Enterprise cluster node pod on when:

- The cluster is created
- The cluster is resized
- A pod fails

Here are the ways that you can control the pod scheduling:

Using node selectors

The `nodeSelector` property of the cluster specification uses the same values and structures as the Kubernetes `nodeSelector`. In general, node labels are a simple way to make sure that specific nodes are used for Redis Enterprise pods. For example, if nodes ‘n1’ and ‘n2’ are labeled as “high memory”:

```
kubectl label nodes n1 memory=high
kubectl label nodes n2 memory=high
```

The Redis Enterprise cluster CRD can request to be scheduled on these nodes:

```yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: rec
spec:
  nodes: 3
nodeSelector:
  memory: high
```

Then, when the operator creates the StatefulSet associated with the pod, the nodeSelector section is part of the pod specification. When the scheduler attempts to create new pods, it needs to satisfy the node selection constraints.

Using node pools

A node pool is a common part of the underlying infrastructure of the Kubernetes cluster deployment and provider. Often, node pools are similarly-configured classes of nodes such as nodes with the same allocated amount of memory and CPU. Implementors often label these nodes with a consistent set of labels.

On Google Kubernetes Engine (GKE), all node pools have the label `cloud.google.com/gke-nodepool` with a value of the name used during configuration. On Microsoft Azure Kubernetes System (AKS), you can create node pools with a specific set of labels. Other managed cluster services may have similar labeling schemes.

You can use the `nodeSelector` section to request a specific node pool by label values. For example, on GKE:
Using node taints

You can use multiple node taints with a set of tolerations to control Redis Enterprise cluster node pod scheduling. The `podTolerations` property of the cluster specification specifies a list of pod tolerations to use. The value is a list of Kubernetes tolerations.

For example, if the cluster has a single node pool, the node taints can control the allowed workloads for a node. You can add taints to the node, for example nodes n1, n2, and n3, reserve a set of nodes for the Redis Enterprise cluster:

```bash
kubectl taint nodes n1 db=rec:NoSchedule
kubectl taint nodes n2 db=rec:NoSchedule
kubectl taint nodes n3 db=rec:NoSchedule
```

This prevents any pods from being scheduled onto the nodes unless the pods can tolerate the taint `db=rec`.

You can then add the toleration for this taint to the cluster specification:

```yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: rec
spec:
  nodes: 3
  podTolerations:
    - key: db
      operator: Equal
      value: rec
      effect: NoSchedule
```

A set of taints can also handle more complex use cases. For example, a `role=test` or `role=dev` taint can be used to designate a node as dedicated for testing or development workloads via pod tolerations.

Using pod anti-affinity

By default, the Redis Enterprise node pods are not allowed to be placed on the same node for the same cluster:

```yaml
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: redis-enterprise
          redis.io/cluster: rec
          redis.io/role: node
        topologyKey: kubernetes.io/hostname
```

Each pod has the three labels above where `redis.io/cluster` is the label for the name of your cluster.
You can change this rule to restrict or include nodes that the Redis Enterprise cluster node pods can run on. For example, you can delete the redis.io/cluster label so that even Redis Enterprise node pods from different clusters cannot be scheduled on the same Kubernetes node:

```yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: rec
spec:
  nodes: 3
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchLabels:
        app: redis-enterprise
        redis.io/role: node
      topologyKey: kubernetes.io/hostname
```

or you can prevent Redis Enterprise nodes from being schedule with other workloads. For example, if all database workloads have the label 'local/role: database', you can use this label to avoid scheduling two databases on the same node:

```yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: rec
spec:
  nodes: 3
extraLabels:
  local/role: database
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchLabels:
        local/role: database
        app: redis-enterprise
        redis.io/cluster: rec
        redis.io/role: node
      topologyKey: kubernetes.io/hostname
```

In this case, any pods that are deployed with the label local/role: database cannot be scheduled on the same node.

**Using rack awareness**

You can configure Redis Enterprise with rack-zone awareness to increase availability during partitions or other rack (or region) related failures.

Rack-zone awareness is a single property in the Redis Enterprise cluster CRD named rackAwarenessNodeLabel. This value for this label is commonly failure-domain.beta.kubernetes.io/zone as documented in 'Running in multiple zones'.

You can check the value for this label in your nodes with the command:
$kubectl get nodes -o custom-columns="name:metadata.name","rack\zone:metadata.labels.failure-domain\.beta\.kubernetes\.io/zone"

<table>
<thead>
<tr>
<th>name</th>
<th>rack\zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip-10-0-x-a.eu-central-1.compute.internal</td>
<td>eu-central-1a</td>
</tr>
<tr>
<td>ip-10-0-x-b.eu-central-1.compute.internal</td>
<td>eu-central-1a</td>
</tr>
<tr>
<td>ip-10-0-x-c.eu-central-1.compute.internal</td>
<td>eu-central-1b</td>
</tr>
<tr>
<td>ip-10-0-x-d.eu-central-1.compute.internal</td>
<td>eu-central-1b</td>
</tr>
</tbody>
</table>

**Enabling the cluster role**

For the operator to read the cluster node information, you must create a cluster role for the operator and then bind the role to the service account.

Here's a cluster role:

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: redis-enterprise-operator
rules:
  # needed for rack awareness
  - apiGroups: [""
    resources: ["nodes"]
    verbs: ["list", "get", "watch"]
```

And here's how to apply the role:

```bash
kubectl apply -f https://raw.githubusercontent.com/RedisLabs/redis-enterprise-k8s-docs/master/rack_awareness/rack_aware_cluster_role.yaml
```

The binding is typically to the redis-enterprise-operator service account:

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: redis-enterprise-operator
subjects:
  - kind: ServiceAccount
    namespace: NAMESPACE_OF_SERVICE_ACCOUNT
    name: redis-enterprise-operator
roleRef:
  kind: ClusterRole
  name: redis-enterprise-operator
  apiGroup: rbac.authorization.k8s.io
```

and it can be applied by running:

```bash
kubectl apply -f https://raw.githubusercontent.com/RedisLabs/redis-enterprise-k8s-docs/master/rack_awareness/rack_aware_cluster_role_binding.yaml
```

Once the cluster role and the binding have been applied, you can configure Redis Enterprise clusters to use rack awareness labels.

**Configuring rack awareness**
You can configure the node label to read for the rack zone by setting the `rackAwarenessNodeLabel` property:

```yaml
apiVersion: app.redislabs.com/v1
kind: RedisEnterpriseCluster
metadata:
  name: example-redisenterprisecluster
spec:
  nodes: 3
  rackAwarenessNodeLabel: failure-domain.beta.kubernetes.io/zone
```

Note: When you use the `rackAwarenessNodeLabel` property, the operator will change the topologyKey for the anti-affinity rule to the label name used unless you have specified the `podAntiAffinity` property as well. If you use `rackAwarenessNodeLabel` and `podAntiAffinity` together, you must make sure that the topologyKey in your pod anti-affinity rule is set to the node label name.

Updated: October 6, 2021