The Highs and Lows of Running a Distributed Database on Kubernetes

Presented by  Alex Robinson / Systems Engineer
@alexwritescode
Databases are critical to the applications that use them
You need to be very careful when making big changes to your database
Containers are a huge change
To succeed, you must:
To succeed, you must:

1. Understand your database
To succeed, you must:

1. Understand your database
2. Understand your orchestration system
To succeed, you must:

1. Understand your database
2. Understand your orchestration system
3. Plan for the worst
Let’s talk about databases in Kubernetes

- Why would you even want to run databases in Kubernetes?
- What do databases need to run reliably?
- What should you know about your orchestration system?
- What’s likely to go wrong and what can you do about it?
My experience with databases and containers

• Worked directly on Kubernetes and GKE from 2014-2016
  ○ Part of the original team that launched GKE
• Led all container-related efforts for CockroachDB from 2016-2019
  ○ Configurations for Kubernetes, DC/OS, Docker Swarm, even Cloud Foundry
  ○ AWS, GCP, Azure, On-Prem
  ○ From single availability zone deployments to multi-region
  ○ Helped users deploy and troubleshoot their custom setups
These days...

Cloudflare Workers
Edge computing for everyone
Why even bother?

We’ve been operating databases for decades
Traditional management of databases

1. Provision one or more beefy machines with large/fast disks
2. Copy binaries and configuration onto machines
3. Run binaries with provided configuration
4. Never change anything unless absolutely necessary
Traditional management of databases

- **Pros**
  - Stable, predictable, understandable

- **Cons**
  - Most management is manual, especially to scale or recover from hardware failures
    - And that manual intervention may not be very well practiced
So why move state into Kubernetes?

• The same reasons you’d move stateless applications to Kubernetes
  ○ Automated deployment, scheduling, resource isolation, scalability, failure recovery, rolling upgrades
    ■ Less manual toil, less room for operator error
• Avoid separate workflows for stateless vs stateful applications
Challenges of managing state

“Understand your databases”
What do stateful systems need?
What do stateful systems need?

- Process management
- Persistent storage
What do stateful systems need?

- Process management
- Persistent storage
- If distributed, also:
  - Network connectivity
  - Consistent name/address
  - Peer discovery
Managing State on Kubernetes

“Understand your orchestration system”
Let’s skip over the basics

- Unless you want to manually pin pods to nodes, you should use either:
  - StatefulSet:
    - decouples replicas from nodes
    - persistent address for each replica, DNS-based peer discovery
    - network-attached storage instance associated with each replica
  - DaemonSet:
    - pin one replica to each node
    - use node’s disk(s)
Where do things go wrong?
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: cockroachdb
spec:
  serviceName: "cockroachdb"
  replicas: 3
  template:
    metadata:
      labels:
        app: cockroachdb
    spec:
      containers:
        - name: cockroachdb
          image: cockroachdb/cockroach:v2.1.0
          ports:
            - containerPort: 26257
              name: grpc
            - containerPort: 8080
              name: http
          command: ["cockroach", "start", "--insecure", "--join=cockroachdb"]
Don’t trust the defaults!

• If you don’t specifically ask for persistent storage, you won’t get any
  ○ *Always* think about and specify where your data will live
Don’t trust the defaults!

- If you don’t specifically ask for persistent storage, you won’t get any
  - Always think about and specify where your data will live

1. Data in container
2. Data on host filesystem
3. Data in network storage
Ask for a dynamically provisioned PersistentVolume

```yaml
volumeMounts:
  - name: datadir
    mountPath: /cockroach/cockroach-data

volumes:
  - name: datadir
    persistentVolumeClaim:
      claimName: datadir

volumeClaimTemplates:
  - metadata:
      name: datadir
    spec:
      accessModes:
        - "ReadWriteOnce"
    resources:
      requests:
        storage: 100Gi
```
Don’t trust the defaults!

- Now your data is persistent
- But how’s performance?
Don’t trust the defaults!

• If you don’t create and request your own StorageClass, you’re probably getting slow disks
  ○ Default on GCE is non-SSD (pd-standard)
  ○ Default on Azure is non-SSD (non-managed blob storage)
  ○ Default on AWS is gp2, which are backed by SSDs but with fewer IOPs than io2

• This really affects database performance
Use a custom StorageClass

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd

volumeClaimTemplates:
- metadata:
  name: datadir
spec:
  accessModes:
  - "ReadWriteOnce"
  resources:
    requests:
      storage: 100Gi

storageClassName: fast
```
Performance problems

• There are a lot of other things you have to do to get performance equivalent to what you’d get outside of Kubernetes

• For more detail, see

What other defaults are bad?
What other defaults are bad?

• If you:
  ○ Create a Kubernetes cluster with 3 nodes
  ○ Create a 3-replica StatefulSet running CockroachDB

• What happens if one of the nodes fails?
Don’t trust the defaults!

Node 1

cockroachdb-0

cockroachdb-1

Node 2

cockroachdb-2

Node 3
Don’t trust the defaults!

- If you don’t specifically ask for your StatefulSet replicas to be scheduled on different nodes, they may not be (k8s issue #41130)
  - If the node with 2 replicas dies, Cockroach will be unavailable until they come back
- This is terrible for fault tolerance
  - What’s the point of running 2 database replicas on the same machine?
Configure pod anti-affinity

```yaml
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                - cockroachdb
            topologyKey: kubernetes.io/hostname
```

What can go wrong other than bad defaults?
What else can go wrong?

- In early tests, Cockroach pods would fail to get re-created if all of them were brought down at once
- Kubernetes would create the first pod, but not any others
What else can go wrong?

```
$ kubectl describe pod cockroachdb-0
```

```
Events:
Type     Reason       Age            From                   Message
----      ------       -----          ------                 -------
Warning   Unhealthy    4m (x420 over 1h) kubelet, minikube  Readiness probe failed: HTTP probe failed with statuscode: 500
```
Know your app and your orchestration system

- StatefulSets (by default) only create one pod at a time
- They also wait for the current pod to pass readiness probes before creating the next
Know your app and your orchestration system

- StatefulSets (by default) only create one pod at a time
- They also wait for the current pod to pass readiness probes before creating the next
- The Cockroach health check used at the time only returned healthy if the node was connected to a majority partition of the cluster
Before the restart

healthy?

yes
If just one node were to fail

healthy?

yes
If just one node were to fail

healthy?

yes

Create missing pod
After all nodes fail

Wait for first pod to be healthy before adding second

Wait for connection to rest of cluster before saying I’m healthy

healthy?

no
Solution to pod re-creation deadlock

- Keep basic liveness probe endpoint
  - Simply checks if process can respond to any HTTP request at all

- Create new readiness probe endpoint in Cockroach
  - Returns HTTP 200 if node is accepting SQL connections
Solution to pod re-creation deadlock

• Keep basic liveness probe endpoint
  ○ Simply checks if process can respond to any HTTP request at all

• Create new readiness probe endpoint in Cockroach
  ○ Returns HTTP 200 if node is accepting SQL connections

• Now that it’s an option, tell the StatefulSet to create all pods in parallel
Other potential issues to look out for

- Set resource requests/limits for proper isolation and to avoid evictions
- No PodDisruptionBudgets by default (#35318)
- If in the cloud, don’t depend on your nodes to live forever
  - Hosting services (I’m looking you, GKE) tend to just delete and recreate node VMs in order to upgrade node software
  - Be especially careful about using the nodes’ local disks because of this
- If on-prem, good luck getting fast, reliable network attached storage
Other potential issues to look out for

- If you issue TLS certificates for StatefulSet DNS addresses, don’t forget to include the namespace-scoped addresses
  - “cockroachdb.default.kubernetes.svc.local” vs just “cockroachdb”
  - Needed for cross-namespace communication
  - Also don’t put pod IPs in node certs - it’ll work initially, but not after pod re-creation

- Multi-region stateful systems are really tough to make work
  - Both network connectivity and persistent addresses are hard to set up
  - KubeCon talk with more info: [https://youtu.be/nNjWrAjzgr8](https://youtu.be/nNjWrAjzgr8)
How to get started

Isn’t this all a lot of work?
Gettings things right is far from easy
What should you do if you aren’t an expert on the systems you want to use?
How to get started

• You could take the time to build expertise
How to get started

- You could take the time to build expertise
- But ideally someone has already done the hard work for you
Off-the-shelf configurations

• There are great configurations available for popular OSS projects
• They’ve usually been made by someone who knows that project well
• They’ve often already been proven in production by other users
Off-the-shelf configurations

- Off-the-shelf YAML files are quite limited, though
  - YAML forces the config writer to make decisions that would best be left to the user
  - No built-in method for parameter substitution
- How could a config writer possibly know your desired:
  - StorageClass
  - Disk size
  - CPU and memory requests/limits
  - Application-specific configuration options
  - etc.
Enter: Helm, the Kubernetes package manager

- Package author defines set of parameters that can be easily overridden
- User doesn’t have to understand or muck with the YAML files
  - Just look through list of parameters and pick which need customizing
- [hub.helm.sh](https://hub.helm.sh)

```bash
helm install stable/cockroachdb --set Replicas=5 --set StorageClass=ssd
```
Summary

Go containerize your databases
Don’t let configuration mistakes take down your production services
1. Understand your database
2. Understand your orchestration system
3. Plan for the worst
1. Understand your database
2. Understand your orchestration system
3. Plan for the worst

(or use a package manager)
Thank You!

Workers KV: https://www.cloudflare.com/products/workers-kv/
CockroachDB: https://github.com/cockroachdb/cockroach

arobinson@cloudflare.com / @alexwritescode