

[Get unlimited access](#)[Open in app](#)

Published in Analytics Vidhya

You have 1 free member-only story left this month. [Upgrade for unlimited access.](#)



Shubhomoy Biswas

[Follow](#)Mar 12, 2021 · 5 min read · [★](#) · [Listen](#)

Save



Running Apache Spark with HDFS on Kubernetes cluster

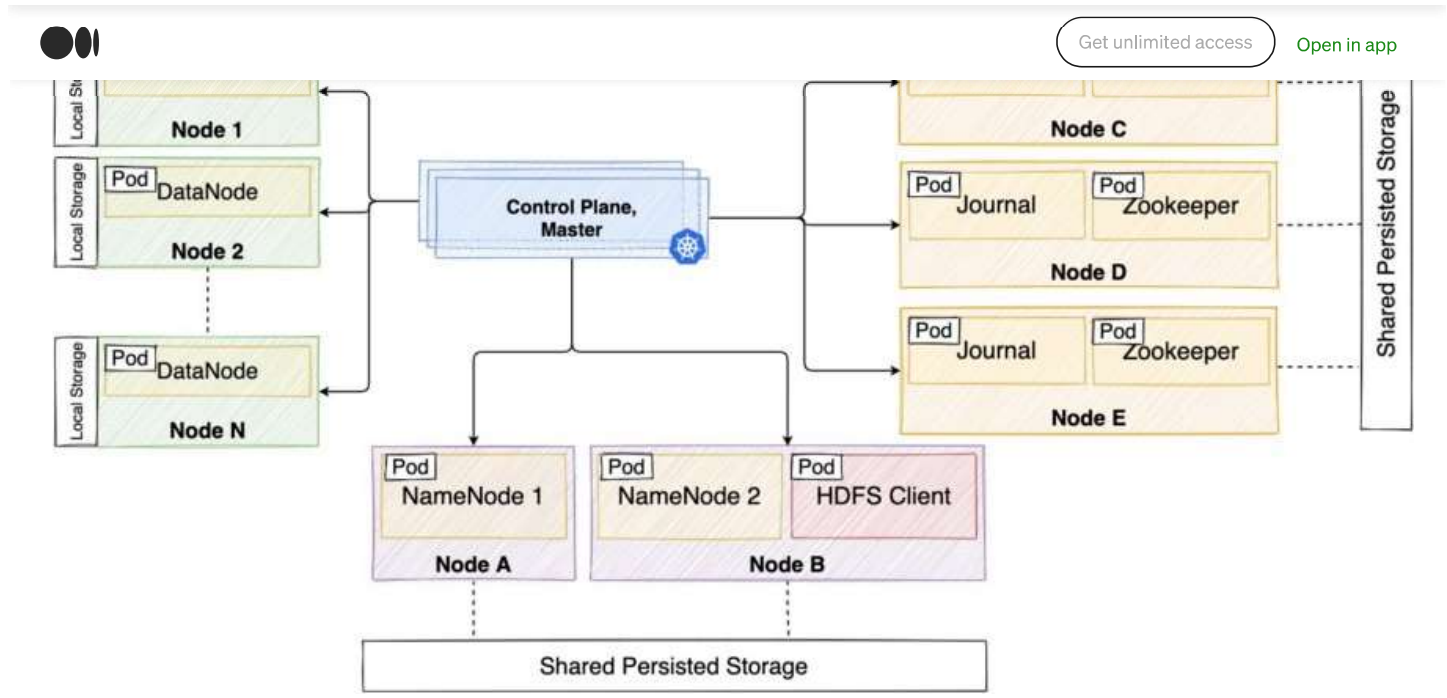


Why on Kubernetes?

From the numerous advantages that *Kubernetes* offers, I particularly find it beneficial to deploy *HDFS* on *K8s* because of the **ease of scalability** *K8s* provides and requires fewer management tasks. Data is ever-increasing, and one needs to have a stable and easy horizontal scalable architecture to accommodate terabytes of data while providing fault tolerance. By having *HDFS* on *Kubernetes*, one needs to add new nodes to an existing cluster and let *Kubernetes* handle the configuration for the new *HDFS Datanodes* (as pods)!

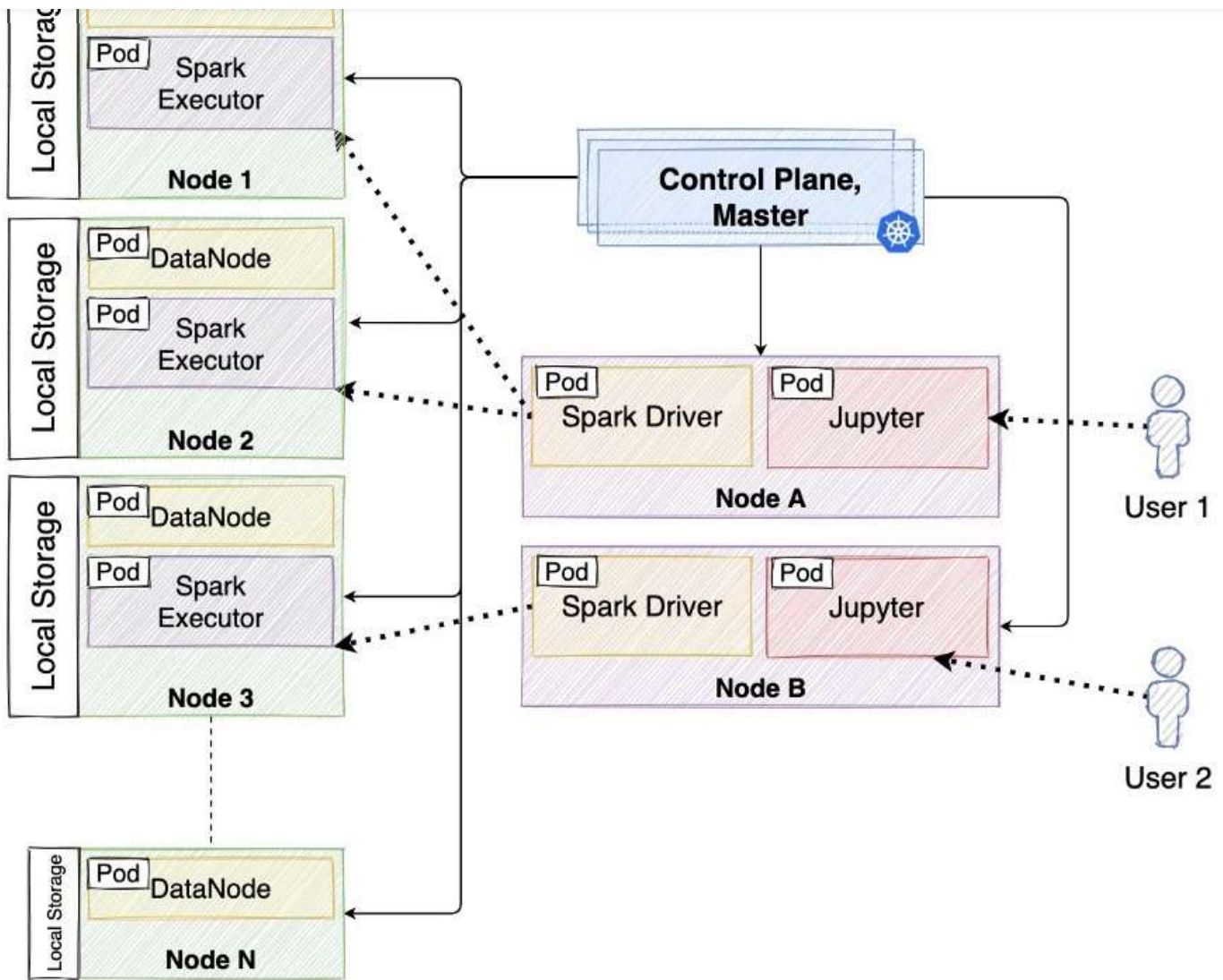
Below is an overview of a *HDFS* HA setup running on *Kubernetes*.





Also by making our *Spark Executors* spin up dynamically inside our *Kubernetes* cluster offers additional benefits. First, you can configure them to spin up in the nodes where your *HDFS Datanode* pods reside. Second, you will have the native out-of-the-box isolation that *Kubernetes* provide using `namespaces`.

Below is a typical *Apache Spark* deployment on *Kubernetes*.



Oh, and did I mention *Jupyter*? If your team need tools such as *Jupyter* notebooks to analyze the data interactively, you can spin up these notebooks inside the *Kubernetes* cluster and connect them with their respective *Spark* drivers. This way, multiple users can have their separate *Jupyter* environment and *Spark* Executors, as and when needed in the same cluster with complete isolation!

Deploy HDFS

HDFS can be easily deployed using a ready-made **Helm chart** provided [here](#). The Helm chart provides HA as well as a simple *HDFS* setup. As of writing this article, I used **Kubernetes version 1.20** and as such, I had to make certain modifications to the charts.

1. Update the *apiVersion* for *StatefulSets* and *DaemonSets* as they are now in *apps/v1*
2. Make sure you have dynamic provisioning enabled for your cluster. If not, then you need to create *PersistentVolumes* for the *StatefulSets* (that includes *Journal*, *Zookeeper* and *Namenodes*).
3. In `requirements.yaml` file (inside *hdfs-k8s* directory), change the *Zookeeper* repository to <https://charts.helm.sh/incubator> and version to 2.1.6
4. There was no option to update the storage class of *Zookeeper* inside `values.yaml`, so you can update that section as below.

32 | 4 | ...



Get unlimited access

Open in app

```

zookeeper:
  ## Configure Zookeeper resource requests and limits
  ## ref: http://kubernetes.io/docs/user-guide/compute-resources/
  resources: ~
  persistence:
    storageClass: <YOUR_STORAGE_CLASS_NAME>
    accessMode: ReadWriteMany
  ## The JVM heap size to allocate to Zookeeper
  env:
    ZK_HEAP_SIZE: 1G

  ## The number of zookeeper server to have in the quorum.
  replicaCount: 1

```

HDFS Datanodes will be deployed as *DaemonSet*, so whenever a new K8s node is added, a new *Datanode pod* will get attached to the *HDFS* cluster! Keep those terabytes of data coming... The Helm chart also provides a client pod from where you can execute all the *HDFS/HADOOP* commands and the WebUI can be accessed from the *NameNode*'s K8s service.

Light up the Spark!

Apache Spark needs a cluster manager, and while *YARN* and *Apache Mesos* are the most common managers, recently, *Kubernetes* can also be the cluster manager for our *Spark* deployment. The process and architecture are fairly mentioned in the official [documentation](#), but I found a good read [here](#) that explains the procedure of setting *Spark on Kubernetes* as **client-mode**. Briefly, it involves,

1. **Creating Spark Executor image** and uploading it to a Docker repository where your K8s cluster will pull it in real-time when spinning up the executor pods. The downloaded *Spark* TAR already has the script file (`docker-image-tool.sh`) for building this image.
2. **Deploying the Driver pod from where users can submit Spark jobs.** The [official PySpark image](#) contains both the Driver as well as the *Jupyter* notebook we can use. We need to create the corresponding *Deployment*, *Service*, and *RBAC* in our *Kubernetes* cluster.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: jupyter-labs
  labels:
    app: jupyter-labs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jupyter-labs
  template:
    metadata:
      labels:
        app: jupyter-labs
    spec:
      serviceAccountName: spark-sa
      containers:
        - name: jupyter-labs
          image: jupyter/pyspark-notebook

```



[Get unlimited access](#)[Open in app](#)

```

---

apiVersion: v1
kind: Service
metadata:
  name: jupyter-labs
  labels:
    app: jupyter-labs
spec:
  ports:
    - protocol: TCP
      port: 29413
  selector:
    app: jupyter-labs
clusterIP: None

```

```

---

apiVersion: v1
kind: Service
metadata:
  name: jupyter-labs-ui
  labels:
    app: jupyter-labs-ui
spec:
  ports:
    - protocol: TCP
      port: 8888
  selector:
    app: jupyter-labs
  type: NodePort

```

```

---

apiVersion: v1
kind: ServiceAccount
metadata:
  name: spark-sa

```

```

---

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: spark-role
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps"]
  verbs: ["create", "get", "watch", "list", "post", "delete"]

```

```

---

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: spark-role-binding
subjects:
- kind: ServiceAccount
  name: spark-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: spark-role
  apiGroup: rbac.authorization.k8s.io

```

The best part is that you can now create multiple Deployments for PySpark (Driver and Jupyter notebook) for multiple users so that each user gets his/her own environment to run Spark jobs.

Jupyter notebook will be accessible on port 8888 (or the respective *NodePort* if you have created the Service as a *NodePort* type). Type in the below python code that will spin up the requester *Spark Executors* in your cluster!



[Get unlimited access](#)[Open in app](#)

```

sparkConf = SparkConf()
sparkConf.setMaster("k8s://https://kubernetes.default.svc.cluster.local:443")
sparkConf.setAppName("spark")
sparkConf.set("spark.kubernetes.container.image", "<YOUR_DOCKER_REPOSITORY_AND_IMAGE_OF_SPARK_EXECUTOR>")
sparkConf.set("spark.kubernetes.namespace", "default")
sparkConf.set("spark.executor.instances", "1")
sparkConf.set("spark.executor.cores", "1")
sparkConf.set("spark.driver.memory", "512m")
sparkConf.set("spark.executor.memory", "512m")
sparkConf.set("spark.kubernetes.authenticate.driver.serviceAccountName", "spark-sa")
sparkConf.set("spark.kubernetes.authenticate.serviceAccountName", "spark-sa")
sparkConf.set("spark.driver.port", "29413")
sparkConf.set("spark.driver.host", "jupyter-labs.default.svc.cluster.local")

spark = SparkSession.builder.config(conf=sparkConf).getOrCreate()
sc = spark.sparkContext

```

Make sure to provide the *Spark Executor* docker image repository that you have created above and the correct service name of your Spark driver in the above code snippet (marked in bold).

Conclusion

Overall, I liked the flexibility and ease of setting up *Apache Spark* with *HDFS* inside *Kubernetes* this way. Also, team members have their own isolated *Jupyter* and *Spark* environments inside the same cluster and at the same time provide an efficient resource sharing of the parent nodes.


If you have made it till here, then I would love to know how are you planning or planned to deploy a scalable *Spark* environment? Would love to hear that :)

Sign up for Analytics Vidhya News Bytes

By Analytics Vidhya

Latest news from Analytics Vidhya on our Hackathons and some of our best articles! [Take a look](#).

Emails will be sent to mikail.saltan@gmail.com. [Not you?](#)

 Get this newsletter

