Distributed Deployment of Qdrant Cluster with Sharding & Replicas

Link: https://medium.com/@vardhanam.daga/distributed-deployment-of-qdrant-cluster-with-sharding-replicas-e7923d483ebc

Segregating your vector data into multiple nodes to enhance resiliency, scalability, and performance.

May 24, 2024

Problem Statement

In this blog, we are going to address the challenges faced by a single node Qdrant setup. And then we are going to demonstrate how we can overcome it by deploying a distributed node setup. By the end of this blog, you will have both a conceptual understanding of distributed networks as well as the technical know-hows of setting it up.

Key Features of a Distributed Deployment Setup in Qdrant

To understand what distributed deployment is, it is important to first understand the following features:

Single Node Qdrant

A single node Qdrant setup uses one server to store and manage all the data and processes. It is less expensive and simpler to set up, but has notable drawbacks. If the server goes down for any reason, such as maintenance or a breakdown, the entire Qdrant service will become unavailable. Furthermore, the performance of the service is limited to what a single server can handle.

Qdrant Cluster

A Qdrant Cluster is a more advanced setup that involves multiple servers, known as nodes, working together. This arrangement distributes the data and workload among several servers, which offers multiple benefits over a single node setup. It’s more resilient because if one server has an issue, the others can keep the service running. It also allows for scalability, meaning more servers can be added as needed to handle more data or more user requests, thus enhancing overall performance.

Cluster Communication

In a cluster, the servers communicate using the Raft consensus protocol. This protocol helps keep the servers synchronized, ensuring that they all agree upon any changes or updates before they are made. This coordination is crucial for maintaining data accuracy and consistency across the cluster.

For operations on individual points though, Qdrant prioritizes speed and availability over strict adherence to this protocol, which allows for faster processing without the complexity of coordinating every detail across all servers.

Sharding

Sharding is a technique to break down a database into different segments. In Qdrant, a collection can be divided into multiple shards, each representing a self-contained store of points.

This distribution allows for parallel processing of search requests, leading to significant performance improvements. Qdrant supports both automatic sharding, where points are distributed based on a consistent hashing algorithm, and user-defined sharding, offering more control over data placement for specific use cases.

Importance of Distributed Development

Distributed development, which involves spreading out data and processing across multiple locations or servers, is essential for creating systems that are reliable and can scale according to demand. In the context of Qdrant, this approach offers several key advantages:

- **Resilience:** By distributing operations across multiple servers, the system can still function even if one server fails.

- **Scalability:** It’s easier to handle more data and more users by adding more servers to the system.

- **Performance:** Distributing the workload helps speed up data processing and retrieval by allowing multiple operations to run in parallel across different servers.

How to Launch a Multi-Node Cluster on Qdrant

We’ll use Docker Compose to launch a 4-node Qdrant Cluster setup. First create a file called docker-compose.yml and paste the following in it:

```
services:
  qdrant_node1:
    image: qdrant/qdrant:v1.9.1
    container_name: qdrant_node1
    volumes:
      - ./data/node1:/qdrant/storage
    ports:
      - "6333:6333"
    environment:
      QDRANT__CLUSTER__ENABLED: "true"
    command: "./qdrant --uri http://qdrant_node1:6335"


  qdrant_node2:
    image: qdrant/qdrant:v1.9.1
    container_name: qdrant_node2
```

```
    volumes:
      - ./data/node2:/qdrant/storage
    depends_on:
      - qdrant_node1
    environment:
      QDRANT__CLUSTER__ENABLED: "true"
    command: "./qdrant --bootstrap http://qdrant_node1:6335 --uri http://qdrant_node2:6335"


  qdrant_node3:
    image: qdrant/qdrant:v1.9.1
    container_name: qdrant_node3
    volumes:
      - ./data/node3:/qdrant/storage
    depends_on:
      - qdrant_node1
    environment:
      QDRANT__CLUSTER__ENABLED: "true"
    command: "./qdrant --bootstrap http://qdrant_node1:6335 --uri http://qdrant_node3:6335"


  qdrant_node4:
    image: qdrant/qdrant:v1.9.1
    container_name: qdrant_node4
    volumes:
      - ./data/node4:/qdrant/storage
    depends_on:
      - qdrant_node1
    environment:
      QDRANT__CLUSTER__ENABLED: "true"
    command: "./qdrant --bootstrap http://qdrant_node1:6335 --uri http://qdrant_node4:6335"
```

The Docker Compose file defines four services, each corresponding to a Qdrant node in the cluster. Here’s what each section generally specifies:

Common elements for all nodes:

- image: Specifies the Docker image to use for the container. All nodes use qdrant/qdrant:v1.9.1.
- container_name: Assigns a unique name to each container running a Qdrant node.
- volumes: Maps a local directory (./data/nodeX) to a directory inside the container (/qdrant/storage). This setup is used for data persistence across container restarts.
- environment: Sets environment variables inside the container. QDRANT__CLUSTER__ENABLED: “true” enables cluster mode in each Qdrant node.

Specifics for individual nodes:

- qdrant_node1
- ports: Maps port 6333 on the host to port 6333 on the container, used for API access to the Qdrant service.

- command: Specifies the command to run inside the container. For qdrant_node1, it initializes with its own URI for cluster communication (http://qdrant_node1:6335).
- qdrant_node2, qdrant_node3, qdrant_node4
- depends_on: Ensures that these nodes start only after qdrant_node1 has started. This dependency is crucial because the subsequent nodes need to connect to the first node for cluster setup.
- command: For these nodes, the command includes both a — bootstrap option pointing to qdrant_node1 (to join the cluster) and a — uri with their own unique URI. This setup is necessary for proper communication within the cluster.

To launch the cluster, run the following command:

```
docker-compose up
```

Now you can access the cluster via the Qdrant client:

```
from qdrant_client import QdrantClient

# Initialize the Qdrant client
client = QdrantClient(host="localhost", port=6333)
```

Next, let’s define the collection and the shard_key.

```
collection_name = "sharding_collection"
key = "tempKey"


# Deleting an existing collection if it exists
response = client.delete_collection(collection_name=f"{collection_name}")


# Creating a new collection with specific configuration
response = client.create_collection(
    collection_name=f"{collection_name}",
    shard_number=6,
    sharding_method=models.ShardingMethod.CUSTOM,
    vectors_config=models.VectorParams(size=768, distance=models.Distance.COSINE),
    replication_factor= 2,
)


# Creating a shard key for the collection
response = client.create_shard_key(f"{collection_name}", f"{key}")
```

Shard Number implies the number of shards per Shard Key. Replication factor implies the number of times the shards are replicated evenly across all the nodes.

In this case, since we have defined only 1 Shard Key, the total number of Physical Shards would be 1*6*2 = 12.

Now let’s input some random vectors into our cluster using the shard key.

```
# Counter for generating unique point IDs
point_counter = 0


# Function to generate a random vector of 768 dimensions with up to 15 decimal points
def generate_random_vector():
    return [round(random.uniform(0, 1), 15) for _ in range(768)]


# Run the loop 1000 times
for _ in range(1000):
    random_vector = generate_random_vector()
    point_counter += 1
    response = client.upsert(
        collection_name=f"{collection_name}",
        points=[
            models.PointStruct(
                id=point_counter,
                vector=random_vector,
            ),
        ],
        shard_key_selector=f"{key}",
    )
```

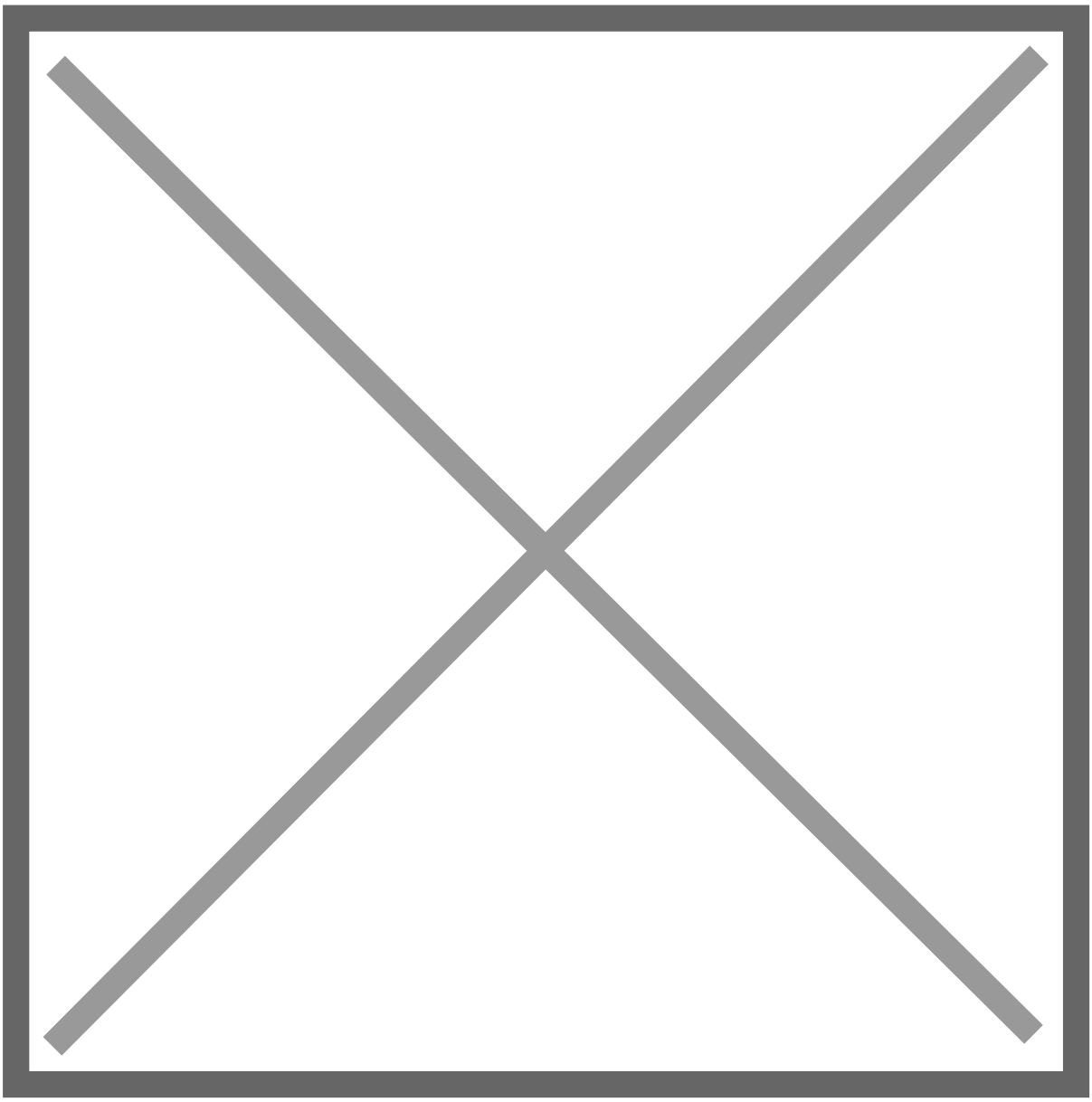
What Happens When We Turn Off a Node?

If we have configured our setup to be a 3+ node cluster with a replication factor of at least 2, then even if we stop one node, it’s not going to affect any operations going on in the cluster.

Let’s test that out.

First get ids of the containers using:

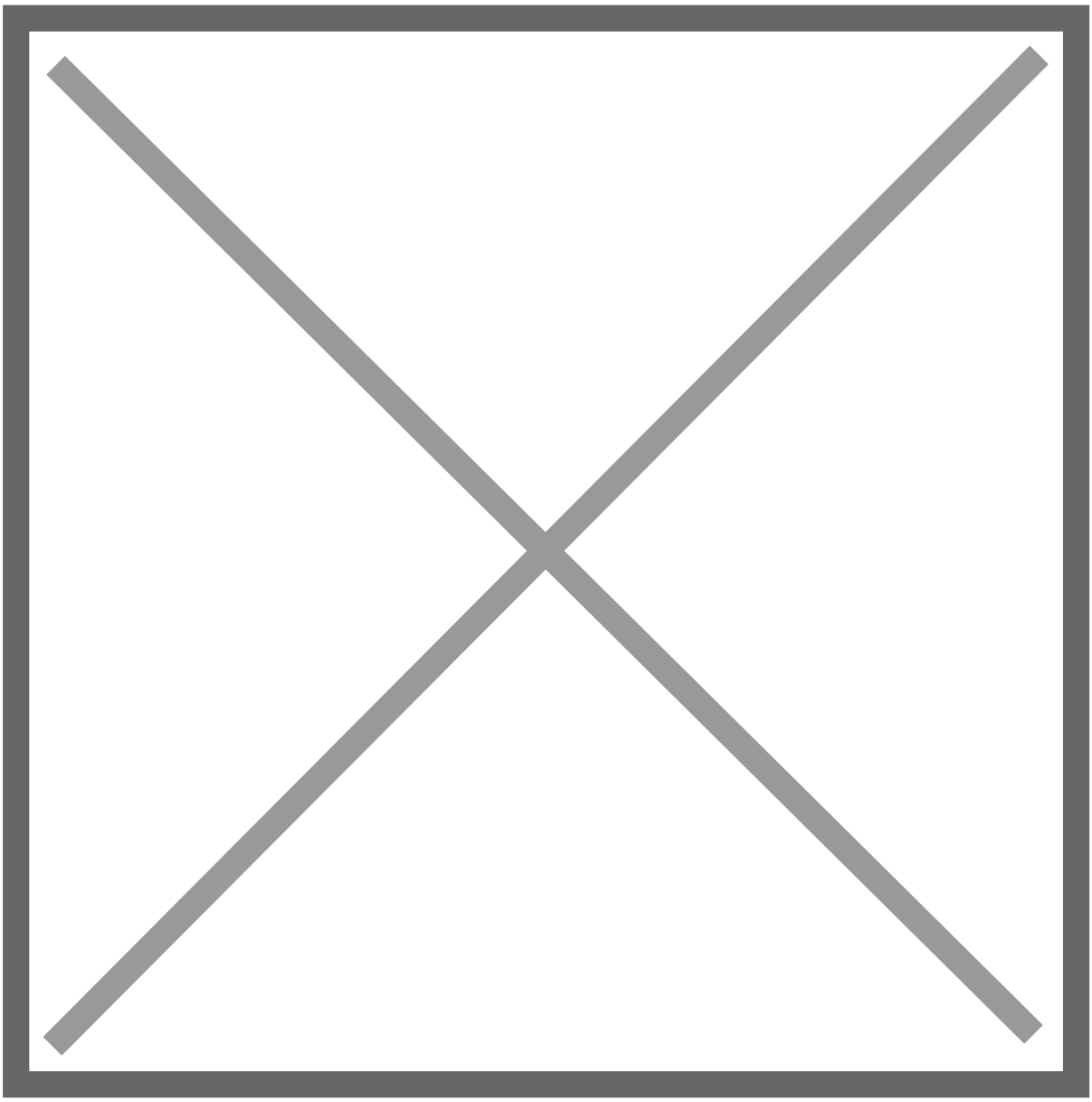
```
docker ps
```



Now let’s stop qdrant_node3.

```
docker stop 0b7fac4c224a
```

Next, let’s investigate our collection.



Even though we switched off a node, we can still access all the points we've inserted into our collection!

Data Distribution between Shards

To see how the data is distributed amongst the shards, we need to make a GET request in the following manner.

```
curl localhost:6333/collections/sharding_collection/cluster
```

{“result”:{“peer_id”:2024384091823610,”shard_count”:6,”local_shards”:[{“shard_id”:2,”shard_key”:”tempKey”,”points_count”:135,”state”:”Active”},{“shard_id”:4,”shard_key”:”tempKey”,”points_count”:178,”state”:”Active”},{“shard_id”:6,”shard_key”:”tempKey”,”points_count”:155,”state”:”Active”}],”remote_shards”:[{“shard_id”:1,”shard_key”:”tempKey”,”peer_id”:8555640407930483,”state”:”Active”},{“shard_id”:1,”shard_key”:”tempKey”,”peer_id”:3252676105267795,”state”:”Active”},{“shard_id”:2,”shard_key”:”tempKey”,”peer_id”:3088601394550397,”state”:”Active”},{“shard_id”:3,”shard_key”:”tempKey”,”peer_id”:8555640407930483,”state”:”Active”},{“shard_id”:3,”shard_key”:”tempKey”,”peer_id”:3252676105267795,”state”:”Active”},{“shard_id”:4,”shard_key”:”tempKey”,”peer_id”:3088601394550397,”state”:”Active”},{“shard_id”:5,”shard_key”:”tempKey”,”peer_id”:3252676105267795,”state”:”Active”},{“shard_id”:5,”shard_key”:”tempKey”,”peer_id”:8555640407930483,”state”:”Active”},{“shard_id”:6,”shard_key”:”tempKey”,”peer_id”:3088601394550397,”state”:”Active”}],”shard_transfers”:[]},”status”:”ok”,”time”:0.000067336}

To know the node by their peer_id, we’ll run:

```
curl http://localhost:6333/cluster
```

{“result”:{“status”:”enabled”,”peer_id”:2024384091823610,”peers”:{“3252676105267795”:{“uri”:”http://qdrant_node2:6335/"},"8555640407930483":{"uri":"http://qdrant_node3:6335/"},"2024384091823610":{"uri":"http://qdrant_node1:6335/"},"3088601394550397":{"uri":"http://qdrant_node4:6335/"}},"raft_info":{"term":10,"commit":40,"pending_operations":0,"leader":2024384091823610,"role":"Leader","is_voter":true},"consensus_thread_status":{"consensus_thread_status":"working","last_update":"2024-05-22T13:26:49.305470782Z"},"message_send_failures":{}},"status":"ok","time":0.000058492}

Analyzing the above results, and putting them in a clear format, we get:

- Node1 (current peer) has shards 2, 4, and 6.
- Node 3 has shards 1, 3, and 5.
- Node 2 has shards 1, 3, and 5.
- Node 4 has shards 2, 4, and 6.

By looking at the above distribution we see that each shard is available on 2 different nodes, and therefore even if one of the nodes fails, the cluster will continue to function.

Final Words

Multi-node clusters, sharding, and replicas are essential features for production environments. Multi-node clusters distribute the workload and enhance system resilience against individual node failures. Sharding divides data across multiple nodes, facilitating parallel processing that boosts performance and scalability. Replication ensures data availability, even if some nodes fail. Together, these techniques can create a robust and fault-tolerant system capable of reliably handling production-level demands.

References

https://qdrant.tech/documentation/guides/distributed_deployment/

https://github.com/Mohitkr95/qdrant-multi-node-cluster/tree/main

---

Revision #1
Created 2024-12-15 00:06:43 UTC by Administrador
Updated 2024-12-15 00:08:59 UTC by Administrador