Durability for Elasticsearch

A key feature of Elasticsearch is the support that it provides for resiliency in the event of node failures and/or network partition events. Replication is the most obvious way in which you can improve the resiliency of any cluster, enabling Elasticsearch to ensure that more than one copy of any data item is available on different nodes in case one node should become inaccessible. If a node becomes temporarily unavailable, other nodes containing replicas of data from the missing node can serve the missing data until the problem is resolved. In the event of a longer-term issue, the missing node can be replaced with a new one, and Elasticsearch can restore the data to the new node from the replicas.

Building resilience in Elasticsearch calls for optimal sharding and replication strategies. Durability ensures that messages aren't lost while writing/reading data into the Elasticsearch. In order to make our cluster more resilient, we take the top-down approach going from cluster -> nodes -> documents -> shards -> replicas.

CLUSTER LEVEL RESILIENCE:

As with any software that stores data, it is important to routinely back up your data. Elasticsearch's replica shards provide high availability during runtime; they enable you to tolerate sporadic node loss without an interruption of service.

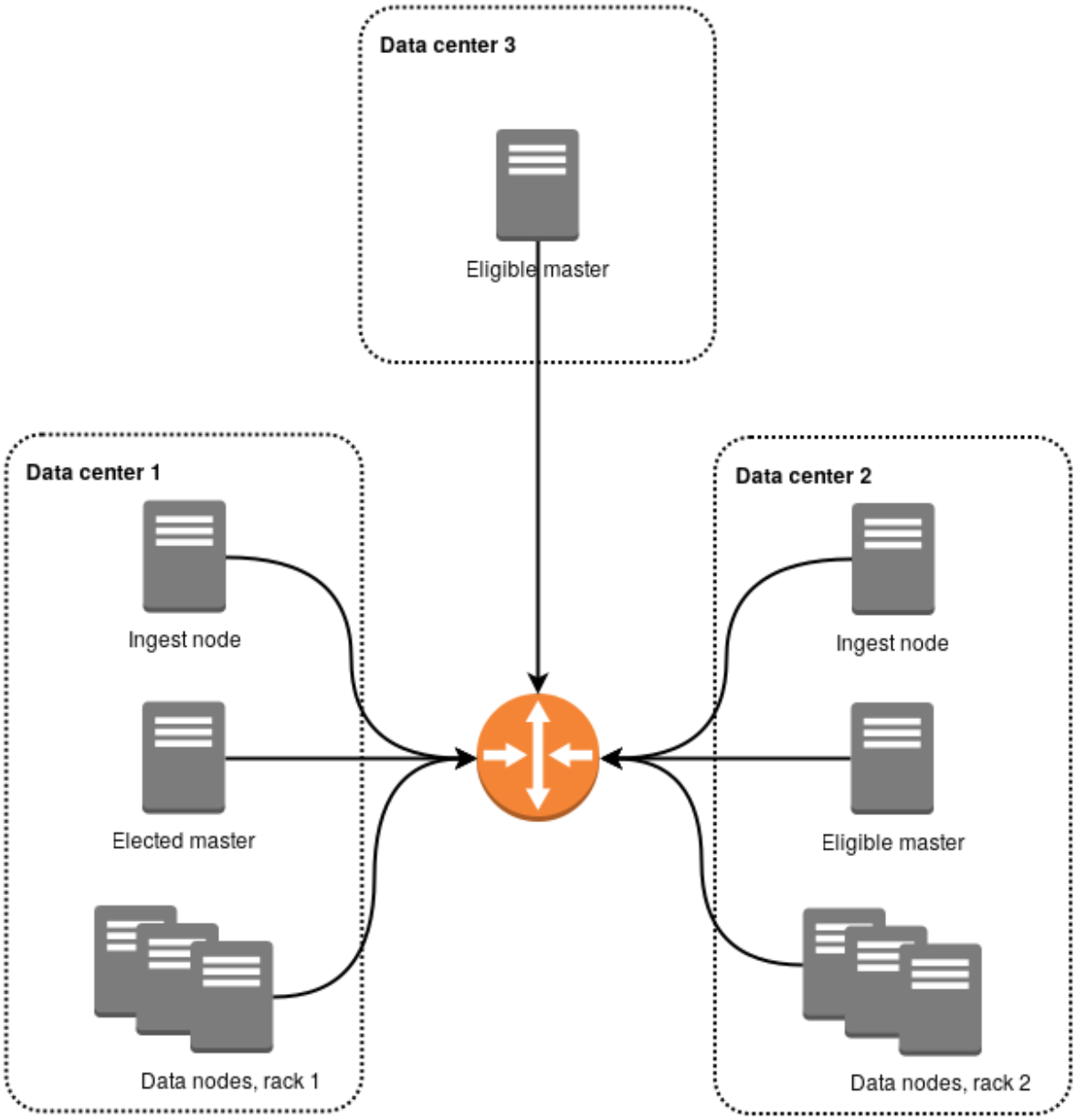
However, replica shards do not protect an Elasticsearch cluster from catastrophic failure. You need a backup of your cluster— a copy in case something goes wrong.

Use the Elasticsearch Snapshot and Restore APIs to backup and restore indexes. Snapshots can be saved to a shared file system. Alternatively, plugins are available that can write snapshots to HDFS (the HDFS Plugin).

You should consider the following points when selecting the snapshot storage mechanism:

- We can implement a shared file system that is accessible from all nodes.
- Only use the HDFS plugin if you are running Elasticsearch in conjunction with Hadoop.
- The HDFS plugin requires you to disable the Java Security Manager running inside the Elasticsearch instance of the JVM.

- The HDFS plugin supports any HDFS-compatible file system provided that the correct Hadoop configuration is used with Elasticsearch.



Minimum fault tolerant cluster setup

When we create or change an Elasticsearch cluster, we can select between one and three availability zones:

- A single availability zone is suitable for testing and development
- Two availability zones are suitable for production use (with a tiebreaker)
- Three availability zones are great for mission critical environments

When we create a cluster with nodes in two availability zones when a third zone is available, we can create a tiebreaker in the third availability zone to help establish quorum in case of loss of an availability zone. The extra tiebreaker node that helps to provide quorum does not have to be a full-fledged and expensive node, as it does not hold data.

For example: By tagging allocators hosted in our cloud, we can create a cluster with eight nodes each in zones `ece-1a` and `ece-1b`, for a total of 16 nodes, and one tiebreaker node in zone `ece-1c`. This cluster can lose any of the three availability zones whilst maintaining quorum, which means that the cluster can continue to process user requests, provided that there is sufficient capacity available when an availability zone goes down.

By default, each node in an Elasticsearch cluster is a master-eligible node and a data node. In larger clusters, such as production clusters, it's a good practice to split the roles, so that master nodes are not handling search or indexing work. When we create a cluster, we can specify to use dedicated master-eligible nodes, one per availability zone.

NODE LEVEL RESILIENCE:

- Intermittent network glitches, VM reboots after routine maintenance at the datacenter, and other similar events can cause nodes to become temporarily inaccessible.
- In these situations, where the event is likely to be short-lived, the overhead of rebalancing the shards occurs twice in quick succession (once when the failure is detected and again when the node becomes visible to the master) can become a significant overhead that impacts performance.
- If we are performing a software upgrade to nodes (such as migrating to a newer release or performing a patch), we may need to work on individual nodes that require taking them offline while keeping the remainder of the cluster available. Ensure that shard reallocation is delayed sufficiently to prevent the elected master from rebalancing shards from a missing node across the remainder of the cluster.

We can prevent temporary node inaccessibility from causing the master to rebalance the cluster by setting the *delayed_timeout* property of an index, or for all indexes. The example below sets the delay to 5 minutes:

```
PUT /_all/settings
{
       "settings": {
          "index.unassigned.node_left.delayed_timeout": "5m"
       }
}
```

Split brain problem:

A split brain can occur if the connections between nodes fail. If a master node becomes unreachable to part of the cluster, an election will take place in the network segment that remains contactable and another node will become the master. In an ill-configured cluster, it is possible for each part of the cluster to have different masters resulting in data inconsistencies or corruption. This phenomenon is known as a *split brain*.

We can reduce the chances of a split brain by configuring the *minimum_master_nodes* property of the discovery module, in the elasticsearch.yml file.

```
discovery.zen.minimum_master_nodes: 2
```

This value should be set to the lowest majority of the number of nodes that are able to fulfil the master role. For example, if our cluster has 3 master nodes, *minimum_master_nodes* should be set to 2; if you have 5 master nodes, *minimum_master_nodes* should be set to 3. Ideally, we should have an odd number of master nodes.

DOCUMENT LEVEL RESILIENCE:

To improve the resiliency of **writes** to the system, indexing operations can be configured to wait for a certain number of active shard copies before proceeding with the operation.
If the requisite number of active shard copies are not available, then the write operation must wait and retry, until either the requisite shard copies have started or a timeout occurs.

By default, write operations only wait for the primary shards to be active before proceeding (i.e. `wait_for_active_shards=1`). This default can be overridden in the index settings dynamically by setting `index.write.wait_for_active_shards`.

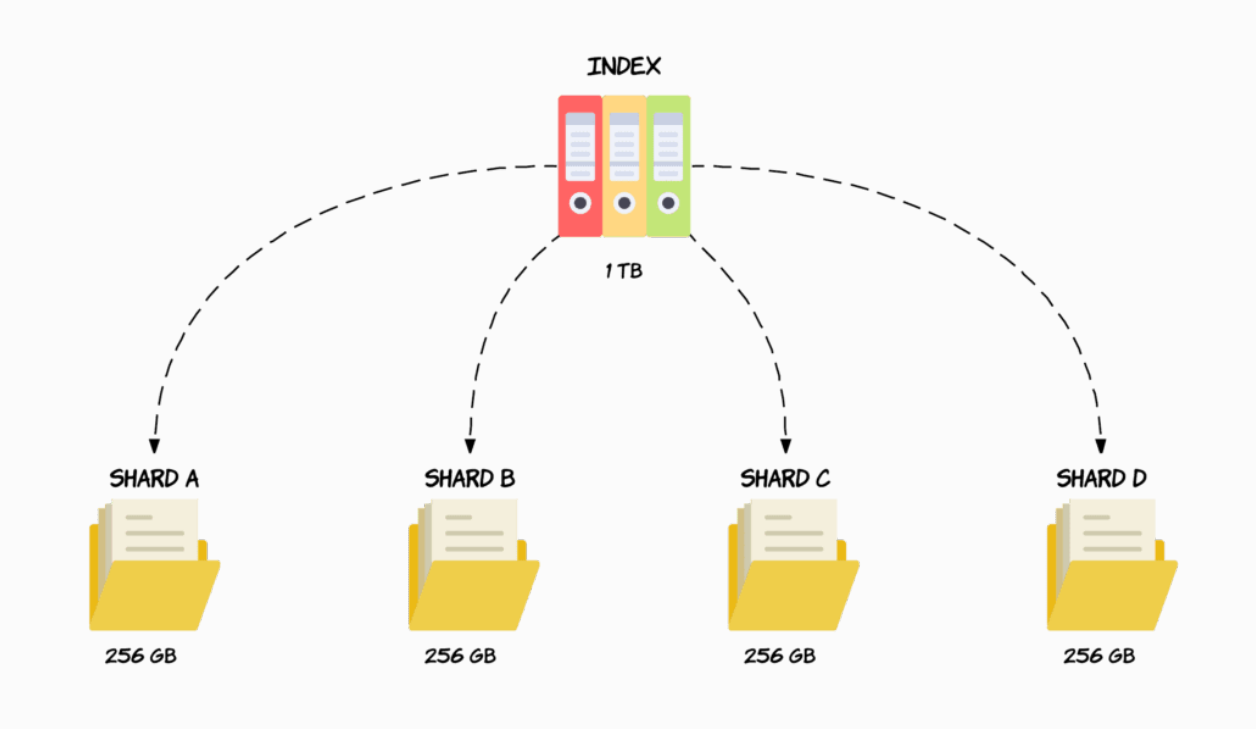
Valid values are `all` or any positive integer up to the total number of configured copies per shard in the index (which is `number_of_replicas+1`). Specifying a negative value or a number greater than the number of shard copies will throw an error.

The `_shards` section of the write operation's response reveals the number of shard copies on which replication succeeded/failed.

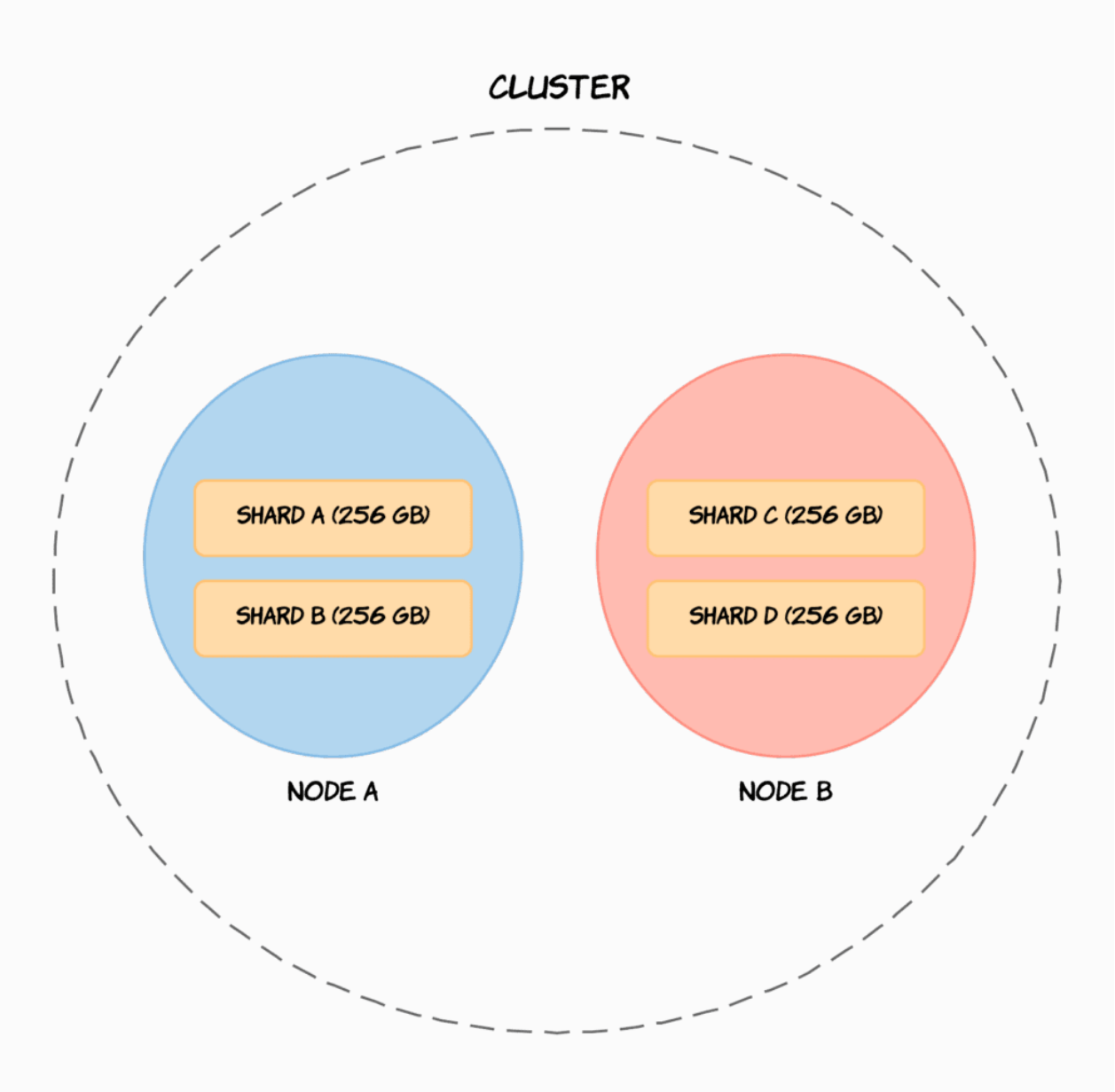
```
{
    "_shards" : {
        "total" : 2,
        "failed" : 0,
        "successful" : 2
    }
}
```

SHARD LEVEL RESILIENCE:

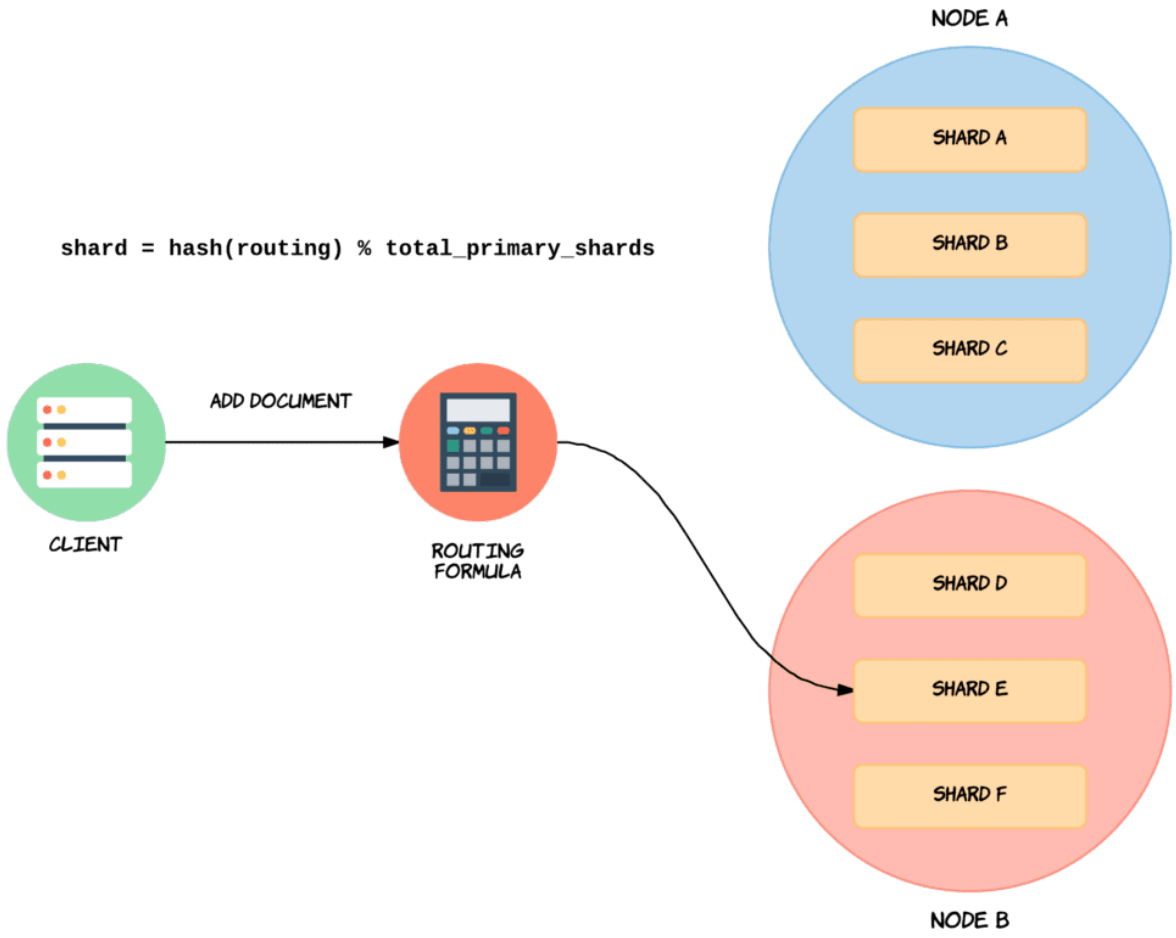
Elasticsearch is extremely scalable due to its distributed architecture. One of the reasons this is the case, is due to something called *sharding*. When the size of an index exceeds the hardware limits of a single node, sharding comes to the rescue. Sharding solves this problem by dividing indices into smaller pieces named *shards*.



The other reason why sharding is important, is that operations can be distributed across multiple nodes and thereby parallelized. This results in increased performance, because multiple machines can potentially work on the same query. This is completely transparent to you as a user of Elasticsearch.



But how does Elasticsearch know on which shard to store a new document, and how will it find it when retrieving it by ID?



Default behaviour in Elasticsearch makes sure that the documents are distributed evenly across an index's primary shards.

This begs the question: What is the optimal way to allocate shards?

Optimizing for heap

Some thumb rules as suggested by elastic:

1. We should keep the shard count to a maximum of 20 per GB of heap size allocated. I.e. For 30 GB heap allocated on the data node, maximum of 600 (30*20) shards should be present on the node.
2. Good idea to regularly check that a single shard size doesn't exceed 40 GB.

3. The largest shard shouldn't take up more than 40% of the data node's capacity.

Large shards cause lots of problems too. They're slower to recover, might block the cluster reallocation, and make optimizing impossible. We can duplicate our indexes, and use Elasticsearch reindex API with aliases.

With an average of 2GB for 1 million documents, for example, I'll use the following:

1. From 0 to 4 million documents per index: 1 shard.
2. From 4 to 5 million documents per index: 2 shards, so the index can still grow without causing too much problems in the future.
3. With more than 5 millions documents, (number of documents / 5 million) + 1 shard. The more data nodes you have, the better it works when you need to work with thousands of huge indexes (up to 300 million documents) in the same cluster.

```
#!/bin/bash
for index in $(list of indexes); do
 documents=$(curl -XGET http://cluster:9200/${index}/_count
2>/dev/null | cut -f 2 -d : | cut -f 1 -d ',')

 if [ $counter -lt 4000000 ]; then
   shards=1
 elif [ $counter -lt 5000000 ]; then
     shards=2
 else
     shards=$(( $counter / 5000000 + 1))
 fi

 new_version=$(( $(echo ${index} | cut -f 1 -d _) + 1))
 index_name=$(echo ${index} | cut -f 2 -d _)

 curl -XPUT http://cluster:9200/${new_version}${index_name} -d '{
 "number_of_shards" : '${shards}'
 }'
curl -XPOST http://cluster:9200/_reindex -d '{
 "source": {
```

```
    "index": "'${index}'"
  },
  "dest": {
    "index": "'${new_version}${index_name}'"
  }
}'
done
```

Optimizing for query performance

When a query is being run it **is executed against each shard available for that index**. That means that having **many shards can result in fast responses**, however, having **too many tasks running at the same time can slow down the performance** of the query. Running the same query against fewer and larger shards will not necessarily be faster either because for each shard the query will have to search through more data although there are fewer tasks running.

As we can see in terms of query performance there needs to be a **balance between shard count and shard size**. It is a good rule to keep our **shard size somewhere between 20GB and 40GB** which should offer you a good balance.

COMMON PROBLEMS with SHARD ALLOCATIONS:

Elasticsearch's cat shards API will tell you which shards are unassigned as:

```
curl -XGET localhost:9200/_cluster/allocation/explain?pretty
```

The example output can be something like:

```
{
  "index" : "testing",
  "shard" : 0,
  "primary" : false,
  "current_state" : "unassigned",
  "unassigned_info" : {
    "reason" : "INDEX_CREATED",
    "at" : "2018-04-09T21:48:23.293Z",
    "last_allocation_status" : "no_attempt"
  },
  "can_allocate" : "no",
  "allocate_explanation" : "cannot allocate because allocation is not permitted
to any of the nodes",
  "node_allocation_decisions" : [
    {
      "node_id" : "t_DVRrfNS12IMhWvlvcfCQ",
      "node_name" : "t_DVRrf",
      "transport_address" : "127.0.0.1:9300",
      "node_decision" : "no",
      "weight_ranking" : 1,
      "deciders" : [
        {
          "decider" : "same_shard",
          "decision" : "NO",
          "explanation" : "the shard cannot be allocated to the same node on
which a copy of the shard already exists"
        }
      ]
    }
  ]
}
```

Some reasons behind unassigned shards:

1. Shard allocation purposely delayed

When a node leaves the cluster, the master node temporarily delays shard reallocation to avoid needlessly wasting resources on rebalancing shards, in the event the original node is able to recover within a certain period of time (one minute, by default).

Example log:

```
[TIMESTAMP][INFO][cluster.routing] [MASTER NODE NAME] delaying allocation for
[54] unassigned shards, next check in [1m]
```

We can dynamically modify the delay period:

```
curl -XPUT "localhost:9200/<INDEX_NAME>/_settings?pretty" -H 'Content-Type:
application/json' -d'
{
  "settings": {
    "index.unassigned.node_left.delayed_timeout": "5m"
  }
}'
```

After the delay period is over, you should start seeing the master assigning those shards.

2. Too many shards, not enough nodes

As nodes join and leave the cluster, the master node reassigns shards automatically, ensuring that multiple copies of a shard aren't assigned to the same node. In other words, the master node will not assign a primary shard to the same node as its replica, nor will it assign two replicas of the same shard to the same node. A shard may linger in an unassigned state if there are not enough nodes to distribute the shards accordingly.

To avoid this issue, make sure that every index in your cluster is initialized with fewer replicas per primary shard than the number of nodes in your cluster by following the formula below:

```
N >= R + 1
```

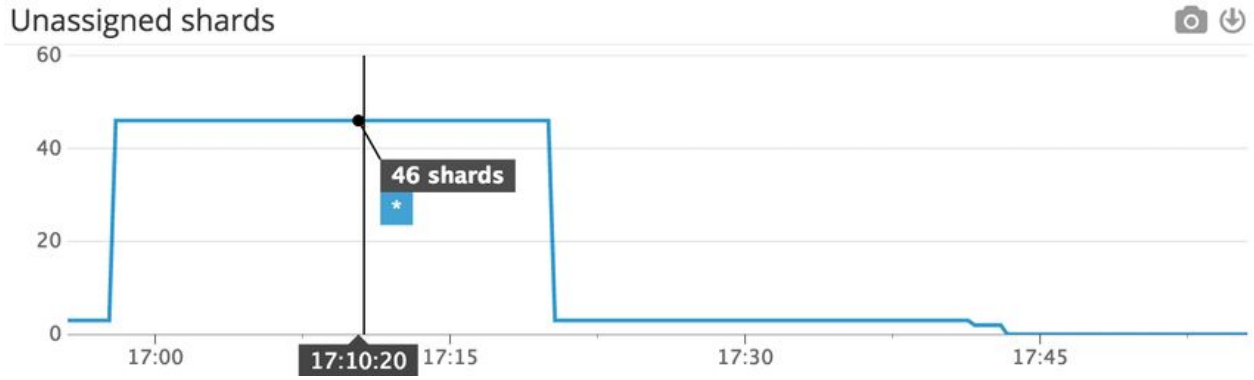
Where N = nodes in the cluster and

R = largest shard replication factor across all indices in the cluster

If a problem arises then to resolve this issue, you can either add more data nodes to the cluster or reduce the number of replicas. In our example, we either need to add at least two more nodes in the cluster or reduce the replication factor to two, like so:

```
curl -XPUT "localhost:9200/<INDEX_NAME>/_settings?pretty" -H 'Content-Type: application/json' -d' { "number_of_replicas": 2 }'
```

We can monitor on Kibana for the assigned shards:



The number of unassigned shards decreases as they are successfully assigned to nodes.

3. **Low disk watermark**

The master node may not be able to assign shards if there are not enough nodes with sufficient disk space (it will not assign shards to nodes that have over 85 percent disk in use).
Once a node has reached this level of disk usage, or what Elasticsearch calls a “low disk watermark”, it will not be assigned more shards.

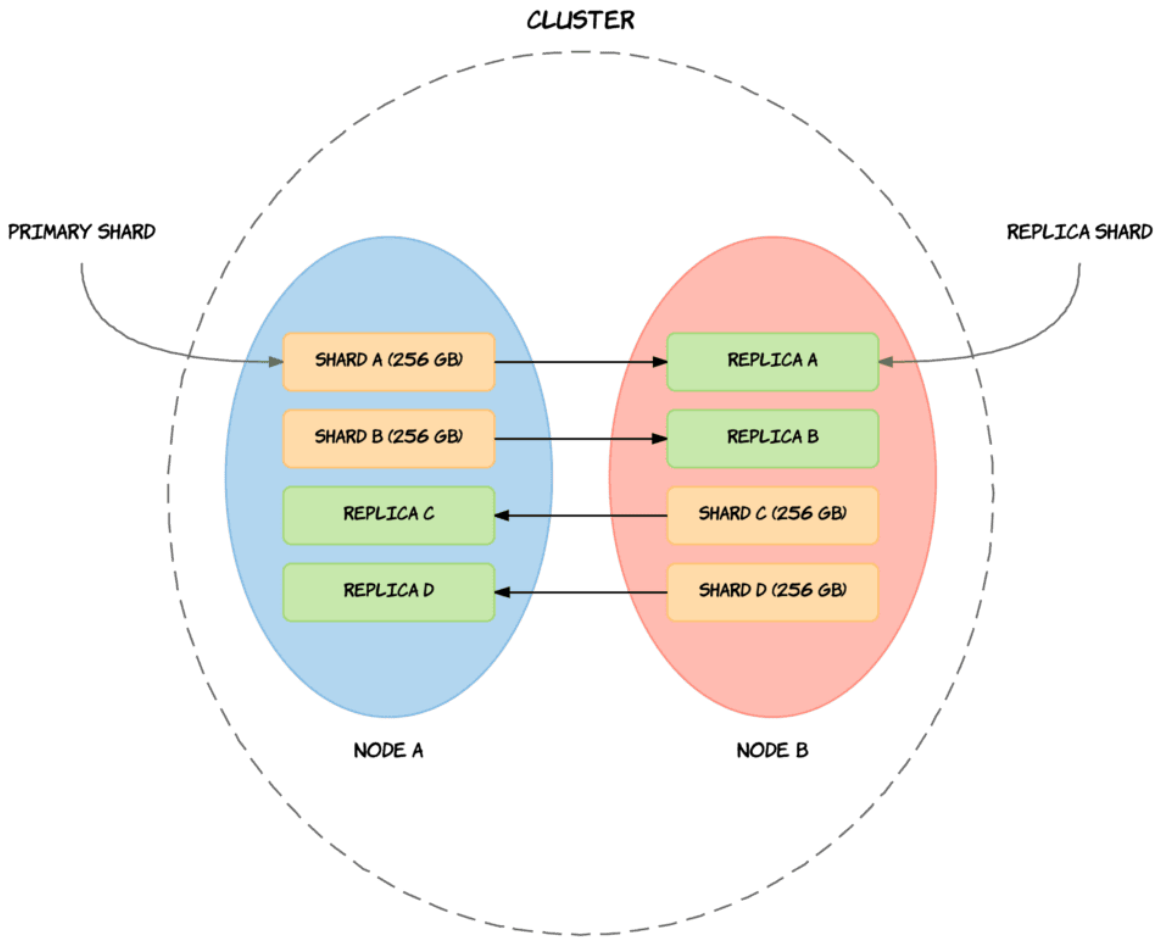
You can check the disk space on each node in your cluster (and see which shards are stored on each of those nodes) by querying the cat API:

```
curl -s 'localhost:9200/_cat/allocation?v'
```

If your nodes have large disk capacities, the default low watermark (85 percent disk usage) may be too low.
You can use the Cluster Update Settings API to change `cluster.routing.allocation.disk.watermark.low` and/or `cluster.routing.allocation.disk.watermark.high`.

```
curl -XPUT "localhost:9200/_cluster/settings" -H 'Content-Type:
application/json' -d'
{
  "transient": {
    "cluster.routing.allocation.disk.watermark.low": "90%"
  }
}'
```

REPLICA LEVEL RESILIENCE:



Replication serves two purposes, with the main one being to provide high availability in case nodes or shards fail. For replication to even be effective if something goes wrong, replica shards are *never* allocated to the same nodes as the primary shards, which you can also see on the above diagram.

As with shards, the number of replicas is defined when creating an index. The default number of replicas is one, being one for each shard. This means that by default, a cluster *consisting of more than one node*, will have 5 primary shards and 5 replicas, totalling 10

shards per index. This makes up a complete replica of your data, so either of the nodes can have a disk failure without you losing any data. The purpose of replication is both to ensure high availability and to improve search query performance, although the main purpose is often to be more fault tolerant.

Shards have a direct impact on storage requirements for any Elasticsearch cluster and more specifically replica shards. Number of replica shards is a multiplier for each primary meaning that the required storage to hold your "original" data increases as you set up more replica shards for your index.

Example:

Daily ingestion for an index: 100GB

Primary shards for the index: 4

Replicas for the index: 2

The 100 GB of data will be split between each primary shard having roughly 25 GB of data for each primary shard. But each primary shard also has 2 replicas which hold copies of the same data bringing the total required disk to 300GB in order to accommodate the index.

Observation:

- The maximum number of replica shards should be kept to maximum equal to (no_of_data_nodes - 1). This will ensure that you don't keep a replica shard on the same node as the primary shard.
- The number of primary shards in an index is fixed at the time that an index is created, but the number of replica shards can be changed at any time, without interrupting indexing or query operations.

Location aware shard/replica allocation:

Allocation awareness is defined as knowledge of where to place copies (replicas) of data. You can arm Elasticsearch with this knowledge so it intelligently distributes replica data across a cluster.

Basically this feature enables your Elastic cluster to know about your physical topology. This enables Elastic to be smart enough to put your primary shard and replica shards into 2 different zones. Zones can be a datacenter or a rack as mentioned before.

You tell Elastic this by adding node attributes to your config file. In this example we will add a node attribute called datacenter to our elasticsearch.yml file. It will have 2 possible values : dc1,dc2

```
node.attr.datacenter: dc1
```

Once you have added this attribute to all your nodes, you need to perform a rolling cluster restart for the attribute value to be read.

Afterwards you need to enable the feature.

```
put _cluster/settings
{
  "persistent" : {
   "cluster.routing.allocation.awareness.attributes": "datacenter"
  }
}
```

Shortly thereafter you will notice some shard activity going on in the cluster when the master will arrange your shards according to your topology. When the dust settles , you can rest assured that your indices are present in both datacenters.