Yarn Resource Allocations and Optimizations

The last lecture taught us about yarn, its role, its architecture, and much more. However, certain configurations allow Yarn to work better and fix a lot of issues that come up in the production environment.

Similar to hive-site.xml which contained hive properties, we have yarn-site.xml which helps you to tune yarn.

Understanding Memory Management in YARN

Memory is a crucial resource in distributed computing. Jobs that run out of memory can fail or degrade in performance. YARN provides several ways to allocate and manage memory across containers.

- **yarn.nodemanager.resource.memory-mb**: This parameter controls the total amount of physical memory available on a node that YARN can allocate to containers. This is important because it prevents the node from being overloaded with too many memory-hungry containers.

  ```
  <property>

    <name>yarn.nodemanager.resource.memory-mb</name>

    <value>[Total Memory per Node]</value>

  </property>
  ```

  Example: If a node has 64 GB of RAM, you may want to allocate 60 GB to YARN containers, leaving the rest for system processes.

- **yarn.scheduler.maximum-allocation-mb**: This controls the maximum amount of memory any single container can request. This is important for preventing a single job from consuming all resources, which might block other jobs.

  ```
  <property>

    <name>yarn.scheduler.maximum-allocation-mb</name>

    <value>[Maximum Memory per Container]</value>
  ```

```
</property>
```

Example: If you want to limit the size of any container to 16 GB, you set this value to 16,000 MB.

YARN also allows for memory tuning at the job level. For example, when running a **MapReduce** job, the **Application Master (AM)** might need more memory, which can be configured in `mapred-site.xml`:

```
<property>

  <name>yarn.app.mapreduce.am.resource.mb</name>

  <value>[Memory for AM]</value>

</property>
```

For **Spark**, you can configure the AM memory as follows:

```
spark.yarn.am.memory=4g
```

This helps manage scenarios where the AM, responsible for handling the job's execution, requires more memory to function effectively.

CPU Allocation in YARN

While memory is a critical resource, CPU allocation is equally important, especially for compute-intensive jobs. YARN allows you to control how many CPU cores a container can use.

- **yarn.nodemanager.resource.cpu-vcores**: This property defines the total number of virtual CPU cores available on a node.
  xml
  ```
  <property>

    <name>yarn.nodemanager.resource.cpu-vcores</name>

    <value>[Total CPU Cores per Node]</value>

  </property>
  ```

  Example: If each node has 16 physical cores, you can set this property to 16.

- **yarn.scheduler.maximum-allocation-vcores**: This property controls the maximum number of vCores a single container can request.

```
<property>

  <name>yarn.scheduler.maximum-allocation-vcores</name>

  <value>[Max CPU Cores per Container]</value>

</property>
```

For **MapReduce** jobs, you can define how many CPU cores each map or reduce task gets:

```
<property>

  <name>mapreduce.map.cpu.vcores</name>

  <value>[vCores per Map Task]</value>

</property>



<property>

  <name>mapreduce.reduce.cpu.vcores</name>

  <value>[vCores per Reduce Task]</value>

</property>
```

This fine-tuning is essential for ensuring that each task gets enough CPU resources to prevent bottlenecks, especially in CPU-bound operations.

Dynamic Resource Allocation in YARN

Dynamic resource allocation allows YARN to adjust resource usage based on real-time demands without stopping or restarting jobs. This is especially useful for frameworks like Spark, which can scale resource usage dynamically.

In Spark, dynamic resource allocation is controlled through the following settings in `spark-defaults.conf`:

```
spark.dynamicAllocation.enabled=true

spark.dynamicAllocation.minExecutors=2

spark.dynamicAllocation.maxExecutors=50

spark.dynamicAllocation.initialExecutors=10
```

This configuration allows Spark to start with a certain number of executors (e.g., 10) and then scale up to 50 executors if needed, or scale down to 2 executors if the job requires fewer resources. Dynamic allocation enables the efficient use of cluster resources, especially in shared environments where workloads vary.

How Dynamic Allocation Works:

- **Idle Executors**: If executors are idle for a specified period, YARN automatically scales them down to free up resources.
- **Resource Needs**: If a job suddenly requires more resources (e.g., more mappers or reducers in Spark), YARN dynamically allocates additional containers without restarting the job.

Job Scheduling in YARN: Capacity vs. Fair Scheduler

In multi-tenant YARN clusters, scheduling policies ensure that resources are allocated fairly between users or applications. YARN supports two main scheduling policies: the **Capacity Scheduler** and the **Fair Scheduler**.

Capacity Scheduler

The **Capacity Scheduler** allows you to divide resources into multiple queues. Each queue gets a guaranteed capacity but can use more resources if others are idle.

- **yarn.scheduler.capacity.root.default.capacity**: This property defines the default percentage of resources allocated to a specific queue.

  ```
  <property>
  ```

```
  <name>yarn.scheduler.capacity.root.default.capacity</name>

  <value>[Capacity Percentage]</value>

</property>
```

For example, if your cluster has two primary users, one for data analytics (using Hive or Spark) and the other for ETL (using MapReduce), you can allocate 60% of the cluster's resources to the analytics queue and 40% to the ETL queue.

- **yarn.scheduler.capacity.maximum-am-resource-percent**: This property ensures that no single Application Master (AM) can consume too many resources, effectively limiting the number of concurrent applications.

Fair Scheduler

The **Fair Scheduler** provides an alternative approach by ensuring that all jobs get a fair share of resources over time. Unlike the Capacity Scheduler, which allocates fixed resources to each queue, the Fair Scheduler dynamically allocates resources to jobs based on need.

- **Preemption**: The Fair Scheduler supports preemption, which allows YARN to take resources from long-running, lower-priority jobs and allocate them to high-priority, short-running jobs. Preemption is enabled with the following configuration:
```
<property>

  <name>fairScheduler.preemption</name>

  <value>true</value>

</property>
```

- **Pool Configuration**: Jobs can be assigned to pools, where each pool has a guaranteed minimum share of resources. This ensures that even if there are many long-running jobs, high-priority jobs can still get the resources they need.
xml
Copy code
```
<pool name="highPriority">

  <minShare>4</minShare> <!-- Minimum of 4 vCores -->
```

```
  <weight>2</weight> <!-- High priority jobs get more resources
-->

</pool>


<pool name="lowPriority">

  <minShare>2</minShare>

  <weight>1</weight>

</pool>
```

By configuring pools and preemption, the Fair Scheduler ensures that resource-hogging jobs don't prevent smaller, urgent jobs from completing.

High Availability in YARN

Problem: In the event that the **Resource Manager (RM)** fails, the entire cluster's resource allocation process halts, leading to job failures.

Solution: YARN supports **High Availability (HA)** for the Resource Manager using **Zookeeper** for failover. This configuration allows for an active and standby RM. If the active RM fails, Zookeeper promotes the standby RM to take over without service disruption.

To enable High Availability, configure the following in `yarn-site.xml`:

```
<property>

  <name>yarn.resourcemanager.ha.enabled</name>

  <value>true</value>

</property>


<property>
```

```
  <name>yarn.resourcemanager.ha.rm-ids</name>
  <value>rm1,rm2</value>
</property>

<property>
  <name>yarn.resourcemanager.hostname.rm1</name>
  <value>[Host1]</value>
</property>

<property>
  <name>yarn.resourcemanager.hostname.rm2</name>
  <value>[Host2]</value>
</property>

<property>
  <name>yarn.resourcemanager.zk-address</name>
  <value>[Zookeeper Quorum Address]</value>
</property>
```

This setup ensures continuous job execution even if the primary RM fails.

Disk and Storage Management in YARN

Problem: Containers not only need memory and CPU but also disk space for temporary data and logs. If disk space is not managed, jobs can fail due to insufficient storage.

Solution: YARN allows you to configure local directories for temporary storage and logging, ensuring efficient disk usage:

- **yarn.nodemanager.local-dirs**: Defines the directories where YARN stores temporary data for containers.

```
<property>

  <name>yarn.nodemanager.local-dirs</name>

  <value>/path/to/local/dir1,/path/to/local/dir2</value>

</property>
```

- **yarn.nodemanager.log-dirs**: Specifies where container logs are stored.

```
<property>

  <name>yarn.nodemanager.log-dirs</name>

  <value>/path/to/log/dir1,/path/to/log/dir2</value>

</property>
```

  This ensures that disk space is effectively managed, and YARN can clean up these directories after job completion.