Prof R. Madana Mohana

BIG DATA ANALYTICS

How MapReduce Works

Failures

https://www.youtube.com/c/RASINENIMADANAMOHANA

Failures: *Outline*

- Task Failure
- Application Master Failure

- Node Manager Failure
- Resource Manager Failure

Failures

Failures

- In the real world, user code is buggy, processes crash, and machines fail.
- One of the major benefits of using **Hadoop** is its ability to handle such failures and allow your job to complete successfully.
- We need to consider the **failure** of any of the following entities:
  - `the task`
  - `the application master`
  - `the node manager, and`
  - `the resource manager`

Task Failure

Task Failure

- Consider *first* the case of the **task failing**.
- The most common occurrence of this failure is when user code in the ***map*** or ***reduce*** task throws a runtime exception.
- If this happens, the task **JVM** reports the error back to its parent application master before it exits.
- The error ultimately makes it into the user logs.
- The **application master** marks the task attempt as ***failed***, and frees up the container so its resources are available for another task.

Task Failure

- For Streaming tasks, if the Streaming process exits with a nonzero exit code, it is marked as failed.
- This behavior is governed by the **stream.non.zero.exit.is.failure** property (the default is true).

Task Failure

- Another failure mode is the sudden exit of the task **JVM**- perhaps there is a **JVM** bug that causes the **JVM** to exit for a particular set of circumstances exposed by the **MapReduce** user code.
- In this case, the node manager notices that the process has exited and informs the application master so it can mark the attempt as failed.

Task Failure

- Hanging tasks are dealt with differently.
- The application master notices that it hasn't received a progress update for a while and proceeds to mark the task as failed.
- The task **JVM** process will be killed automatically after this period.
- The timeout period after which tasks are considered failed is normally **10** minutes and can be configured on a per-job basis (or a cluster basis) by setting the **mapreduce.task.timeout** property to a value in milliseconds.

Task Failure

- Setting the timeout to a value of zero disables the timeout, so long-running tasks are never marked as failed.
- *In this case*, a hanging task will never free up its container, and over time there may be cluster slowdown as a result.
- *This approach* should therefore be avoided, and making sure that a task is reporting progress periodically should be enough.

Task Failure

- When the application master is notified of a task attempt that has failed, it will reschedule execution of the task.
- The application master will try to avoid rescheduling the task on a node manager where it has previously failed.
- Furthermore, if a task fails four times, it will not be retried again.
- This value is configurable.

Task Failure

- The maximum number of attempts to run a task is controlled by the **mapreduce.map.maxattempts** property for **map tasks** and **mapreduce.reduce.maxattempts** for **reduce tasks**.
- By default, if any task fails four times (or whatever the maximum number of attempts is configured to), the whole job fails.

Task Failure

- For some applications, it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures.
- In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job.
- **Map** tasks and **reduce** tasks are controlled independently, using the **mapreduce.map.failures.maxpercent** and **mapreduce.reduce.failures.maxpercent** properties.

Task Failure

- A task attempt may also be ***killed***, which is different from it failing.
- A task attempt may be ***killed*** because it is uncertain duplicate or because the node manager it was running on failed and the application master marked all the task attempts running on it as killed.

Task Failure

- ***Killed task*** attempts do not count against the number of attempts to run the task (as set by **mapreduce.map.maxattempts** and **mapreduce.reduce.maxattempts**), because it wasn't the task's fault that an attempt was killed.
- Users may also kill or fail **task attempts** using the **web UI** or the command line (type **mapred** job to see the options). Jobs may be killed by the same mechanisms.

Application Master Failure

Application Master Failure

- Just like **MapReduce** tasks are given several attempts to succeed (in the face of hardware or network failures), applications in **YARN** are retried in the event of failure.
- The maximum number of attempts to run a **MapReduce** application master is controlled by the **mapreduce.am.max-attempts** property.
- The default value is **2**, so if a **MapReduce** application master fails twice it will not be tried again and the job will fail.

Application Master Failure

- **YARN** imposes a limit for the maximum number of attempts for any **YARN** application master running on the cluster, and individual applications may not exceed this limit.
- The limit is set by **yarn.resourcemanager.am.max-attempts** and defaults to **2**, so if you want to increase the number of **MapReduce** application master attempts, you will have to increase the **YARN** setting on the cluster, too.

Application Master Failure

The way recovery works is as follows:

- An application master sends periodic heartbeats to the resource manager, and in the event of application master *failure*, the resource manager will detect the *failure* and start a new instance of the master running in a new container (managed by a node manager).

Application Master Failure

The way recovery works is as follows:

- In the case of the **MapReduce** application master, it will use the job history to recover the state of the tasks that were already run by the (failed) application so they don't have to be rerun.
- Recovery is enabled by default, but can be disabled by setting **yarn.app.mapreduce.am.job.recovery.enable** to false.

Application Master Failure

- The **MapReduce** client polls the application master for progress reports, but if its application master *fails*, the client needs to locate the new instance.
- During job initialization, the client asks the resource manager for the application master's *address*, and then caches it so it doesn't overload the resource manager with a request every time it needs to poll the application master.

Application Master Failure

- If the application master *fails*, however, the client will experience a timeout when it issues a status update, at which point the client will go back to the resource manager to ask for the *new* application master's address.
- This process is transparent to the user.

Node Manager Failure

Node Manager Failure

- If a **node manager fails** by crashing or running very slowly, it will stop sending heartbeats to the **resource manager** (or send them very infrequently).
- The **resource manager** will notice a **node manager** that has stopped sending heartbeats if it hasn't received one for **10** minutes (this is configured, in milliseconds, via the **yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms** property) and remove it from its pool of nodes to schedule containers *on*.

Node Manager Failure

- Any task or application master running on the *failed* node manager will be recovered using the mechanisms described earlier.
- In addition, the application master arranges for **map tasks** that were run and completed successfully on the failed node manager to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed node manager's local filesystem may not be accessible to the reduce task.

Node Manager Failure

- **Node managers** may be ***blacklisted*** if the number of failures for the application is high, even if the node manager itself has not failed.
- Blacklisting is done by the application master, and for **MapReduce** the application master will try to reschedule tasks on different nodes if more than three tasks fail on a node manager.
- The user may set the threshold with the **mapreduce.job.maxtaskfailures.per.tracker** job property.

Resource Manager Failure

Resource Manager Failure

- Failure of the **resource manager** is serious, because without it, neither jobs nor task containers can be launched.
- In the default configuration, the **resource manager** is a single point of failure, since in the (unlikely) event of machine failure, all running jobs fail-and can't be recovered.

Resource Manager Failure

- To achieve **high availability** (**HA**), it is necessary to run a pair of resource managers in an active-standby **configuration**.
- If the *active* **resource manager** fails, then the standby can take over without a significant interruption to the client.

Resource Manager Failure

- Information about all the running applications is stored in a highly available state store (backed by **ZooKeeper** or **HDFS**), so that the standby can recover the core state of the failed *active* **resource manager.**
- Node manager information is not stored in the state store since it can be reconstructed relatively quickly by the *new* **resource manager** as the node managers send their first heartbeats.

Resource Manager Failure

- When the *new* **resource manager** starts, it reads the application information from the state store, then restarts the application masters for all the applications running on the cluster.
- This does not count as a failed application attempt (so it does not count against **yarn.resourcemanager.am.max-attempts**), since the application did not fail due to an error in the application code, but was forcibly killed by the system.

Resource Manager Failure

- In practice, the application master *restart* is not an issue for **MapReduce** applications since they recover the work done by completed tasks.
- The transition of a **resource manager** from standby to active is handled by a failover controller.
- The *default* failover controller is an automatic one, which uses **ZooKeeper** leader election to ensure that there is only a *single active* **resource manager** at one time.

Resource Manager Failure

- Unlike in **HDFS HA** ("**HDFS High Availability**"), the failover controller does not have to be a standalone process, and is embedded in the **resource manager** by default for ease of configuration.
- It is also possible to *configure* manual failover, but this is not recommended.

Resource Manager Failure

- Clients and node managers must be configured to handle **resource manager failover**, since there are now two possible **resource managers** to communicate with.
- They try connecting to each **resource manager** in a round-robin fashion until they find the active one.
- If the active fails, then they will retry until the standby becomes active.