Disk I/O Isolation & Scheduling in YARN

Wei Yan, Karthik Kambatla, bc Wong, and Todd Lipcon

1 Motivation

Resource sharing and isolation techniques in Hadoop YARN have been studied and implemented for resources such as CPUs and memory. However, YARN doesn't support disk I/O management and isolation.

Specifically, we would like for the following to be addressed:

1. Avoid **over-allocation**: If the scheduler does not consider disk, it is easy to over-allocate disk resources for tasks that need relatively smaller share of CPU and memory.
2. **Isolation**: Co-located tasks might compete for I/O resources and interfere with each other. Isolating disk I/O resources is essential for YARN to deliver predictable performance.
3. Schedule **diverse workloads**: Depending on the framework and the job, tasks have varying disk resource requirements. We should allow for applications to request disk resources as needed by the task.

2 Scope

In this work, we (1) augment resource description to include disks along configurable dimensions, (2) provide isolation for allocated resources and (3) add scheduling support for varying disk resource requirements.

Dimensions

Different users might want to share disk resources along different dimensions - bandwidth, iops, weight, capacity etc. This work focuses on weight-based disk sharing (through the notion of vdisks), but allows plugging in other dimensions in the future.

Isolation

This work focuses on isolation for local disk reads on Linux using Cgroups. The design allows for plugging in other enforcement techniques, for instance on other environments. We talk about potential ways of handling disk writes, but leave it for future work.

Spindle locality

We understand that certain applications might need isolated/shard access to specific spindles. We talk about ways of arriving at this, but again leave more details for future work.

3 Approach

Based on the community's experiences adding memory and CPU support, we would like to approach this in three phases - (1) avoiding over-allocation, (2) isolation, (3) scheduling support - in that order.

3.1 Disk I/O Resource Description

For nodes to specify the available disk resources and for applications to request, we propose adding disk to the resource description. Disk resources can be specified along multiple dimensions - weight, bandwidth, iops, capacity. We also provide a way to specify what dimensions to enforce isolation on - each dimension requires a corresponding I/O isolator as discussed in Section 3.3.

As mentioned earlier, we look at weight-based sharing here. We achieve this through the notion of *vdisks*, number of virtual disks. In the simplest case, *vdisks* on a node could be equal to the number of disks it has. If users want multiple tasks to share a disk, they could increase the value of *vdisks* accordingly. A larger *vdisks* value in the request-vector translates to more disk I/O resources.

3.2 Avoiding over-allocation

When a NodeManager heartbeats to the ResourceManager, it specifies the amount of available disk resources along with memory and CPU. The scheduler allocates a default value of disk resources per container to limit the number of containers running on that node.

e.g. One could make use of this by setting the number of *vdisks* to twice the number of disks and allocate each container one *vdisk*.

3.3 Enforcement by NodeManager

We would like to provide disk I/O isolation for co-located containers based on their assigned disk resources. Each dimension requires a corresponding implementation of the isolator.

Here, we focus on local disk read isolation using Cgroups for *vdisks*. Cgroups support two modes of enforcement:

- Proportional weight division: implemented in the completely fair queuing I/O scheduler. This policy allows you to set weights to specific cgroups. This means that each cgroup has a set percentage (depending on the weight of the cgroup) of all I/O operations reserved.
- I/O throttling (Upper limit): this policy is used to set an upper limit for the number of I/O operations or bandwidth for each cgroup.

For weight-based dimensions like *vdisks*, we can use the proportional weight division mechanism and let all containers share the I/O resource in the ratio of their weights.

One might consider the I/O throttling mechanism for other dimensions like bandwidth and iops. Note that the I/O throttling is an upper bound.

3.4 Scheduler-Support for varying disk needs

Applications should be able to ask for disk resources through the updated ResourceRequest. DominantResourceCalculator needs to be updated to include disk resources, subsequently used in individual schedulers. In FairScheduler, this translates to adding disks to the DRF policy; we could allow for admins to choose the resources to be considered in DRF policy.

4 Implementation

We expect the implementation to potentially span multiple releases. To ensure we deliver the completed parts, we would like to develop the first two parts - avoid over-allocation and isolation - on trunk, and the remaining on branch YARN-2139.

Also, for parts being developed on trunk, we propose to add a temporary config to enable/disable the feature. This config is not to be exposed to users; if exposed, we should explicitly state the feature is experimental and the config itself will be removed.

We enumerate the expected changes here, and expect to evolved during the review process of individual patches. The updates might not be reflected back in this doc.

4.1 Disk I/O Resource Description

Similar to `cpu-vcores` and `memory-mb`, we propose to add `disk-vdisks` to scheduler and NM configs, and to the required proto files. In the future, one could add `disk-bandwidth`, `disk-iops` etc.

As it may not be possible to schedule (or enforce) along multiple dimensions at the same time (depending on the dimensions), we propose adding a config to specify what dimension(s) to consider in the scheduler and the NM: `yarn.scheduler.disk-dimensions` and `yarn.nodemanager.resource.disk-dimensions`. Again, here, we limit ourselves to vdisks.

Add field `vdisks` to `ResourceProto` in `yarn_protos.proto`.

```
message ResourceProto {
  optional int32 memory = 1;
  optional int32 virtual_cores = 2;
+ optional int32 vdisks = 3;
}
```

Add field `vdisks` to `org.apache.hadoop.yarn.api.records.Resource`.

```
+ public abstract int getVdisks();
+ public abstract void setVdisks(int vdisks);
```

Add the following configurations to `org.apache.hadoop.yarn.conf.YarnConfiguration`.

```
+ /** Dimension for the disk I/O resource. */
+ public static final String DISK_DIMENSIONS =
"yarn.scheduler.resource.disk-dimensions";

+ /** Number of virtual disk I/O resources which can be allocated for containers. */
+ public static final String NM_VDISKS = NM_PREFIX + "resource.disk-vdisks";

+ /** Minimum request grantable by the RM scheduler. */
+ public static final String RM_SCHEDULER_MINIMUM_ALLOCATION_VDISKS =
YARN_PREFIX + "scheduler.minimum-allocation-vdisks";
+ public static final String DEFAULT_RM_SCHEDULER_MINIMUM_ALLOCATION_VDISKS =
0;

+ /** Maximum request grantable by the RM scheduler. */
+ public static final String RM_SCHEDULER_MAXIMUM_ALLOCATION_VDISKS =
YARN_PREFIX + "scheduler.maximum-allocation-vdisks";
+ public static final String DEFAULT_RM_SCHEDULER_MAXIMUM_ALLOCATION_VDISKS =
20;
```

4.2 Scheduler-side Changes

Summary: The major required changes in the scheduler-side includes (both for FairScheduler and CapacityScheduler):

- Consider the `vdisks` resource when scheduling containers, i.e., the node should have enough `vdisks` resources for the container to be assigned. As we directly add field `vdisks` to the `Resource.java` and update corresponding comparison functions, this can be supported directly, without requiring much change from the scheduler side.
- Add `vdisks` to DRF policy, and configure which resources are included when calculating DRF. As we discussed in Section 3.4, users can configure the set of resource types when calculating dominant resource share. Here we take FairScheduler as an example for an example configuration, where we have two queues (queue1 and queue2). The queue1 is configured to take all three type of resources, while queue2 inherits the default configuration and only consider the CPU and memory. By default, the DRF only considers CPU and memory.

```
+ <queue name="queue1">
+     <DRFPolicyResources>CPU,MEMORY,VDISKS</DRFPolicyResources>
+ </queue>
+ <queue name="queue2">
+ </queue>
+ <DRFPolicyResourcesDefault>CPU,MEMORY</DRFPolicyResourcesDefault>
```

4.3 NodeManager-side Enforcement

Summary: Add support for NodeManager to enforce the disk I/O resource usage for each container. As we discussed above, users can configure the dimension for describing the disk I/O resource. Corresponding, here we also make the disk I/O isolation mechanism pluggable, and users can implement and configure their own implementation according to the selected dimension.

4.3.1 Prepare

Here we use Cgroups' blkio subsystem to implement the disk I/O isolation. We need to make changes to `org.apache.hadoop.yarn.server.nodemanager.util.CgroupsLCEResourcesHandler` to mount the Cgroups path to include `blkio`.

4.3.2 Pluggable Disk I/O Isolation

For different disk I/O dimensions, we use different mechanisms to configure the Cgroup's blkio subsystem. And for supporting more dimensions in future, we make the isolation mechanism to be pluggable.

We define an interface called `DiskIOIsolator`, which includes the following abstract functions:

```
/** Setup the disk I/O limits for the given container. */
void setupDiskIOLimits(containerId, containerResource);
/** Clean the disk I/O limits for the given container. */
void clearDiskIOLimits(containerId);
```

Users can specify their own disk I/O isolation implementation class the field `yarn.nodemanager.disk-${dimension}.isolator-class` in `yarn-site.xml`. And inside the `CgroupsLCEResourcesHandler`, we add the following code to setup and clear the I/O limitation setups.

```
private void setupLimits(ContainerId containerId, Resource containerResource) {
    ...
+  diskIOIsolator.setupDiskIOLimits(containerId, containerResource);
}
private void clearLimits(ContainerId containerId) {
    ….
+  diskIOIsolator.clearDiskIOLimits(containerId);
}
```

Bandwidth/iops/weight isolator implementations would primarily differ in the configuration files on Cgroups' blkio subsystem. Each implementation needs to update the I/O configuration according to the container's `disk-resource` value.

- For bandwidth: `blkio.throttle.read_bps_device`, `blkio.throttle.write_bps_device`.

- For iops: `blkio.throttle.read_iops_device`, `blkio.throttle.write_iops_device`.
- For weight: `blkio.weight`.

Here we discuss the implementation details for the weight dimension. In general, for each newly launched container `C` with `vdisks p`, we create a new cgroup under the blkio subsystem. This new created cgroup's weight for all disk devices is normalized with an allowed range represented by `f(p)`. In the `setupLimit` and `clearLimit` functions, we need to do the follows.

setupLimit:

```
mkdir /sys/fs/cgroup/blkio/hadoop-yarn/C_ID
echo f(p) > /sys/fs/cgroup/blkio/hadoop-yarn/C_ID/blkio.weight
echo C_PID > /sys/fs/cgroup/hadoop-yarn/blkio/C_ID/tasks
```

clearLimit:

```
rm -r /sys/fs/cgroup/blkio/hadoop-yarn/C_ID
```

4.4 ApplicationMaster-Side Changes

Currently Hadoop has example application master implementations for MapReduce jobs and distributed shell jobs. For better usage and evaluation, we also need to update their implementations to accept per-container `vdisks` request from users and submit to the resource manager.

5 Testing Plan

5.1 Unit Testing

Function testing: Here we mainly test each newly introduced function, especially for the scheduler-side. We'll add vdisks information for available resources and resource requests, and verify the scheduler works in a correct way.

Micro benchmark: micro-benchmark is to show the interference problem arising from competing containers that running on the same node, and our approach's effectiveness in mitigating it. Two jobs are submitted to the cluster, each of which runs a single container with heavy read operations. We'll assign these two containers to the same node, and measure their read bandwidths. Without considering `vdisks`, each container will receive half the disk bandwidth (depends on the OS's disk scheduler configuration). When introducing `vdisks` scheduling, each container receives a read bandwidth proportional to their `vdisks` values.

5.2 Regression Testing

The objective of regression testing is to ensure that there should be no degradation of performance if users don't update their old configurations; that is, all containers request the

default `vdisks`, which is 0. Here we'll evaluate existing Hadoop implementation, and the one patched with disk I/O scheduling. For the latter one, we keep all default configurations.

Here we select a set of MapReduce workloads, with different I/O requirements. Experiments will be deployed with single-job running and multiple jobs running.

Job	Description
TeraGen	HDFS write job with 3-fold replication that heavily uses the network.
TeraRead	HDFS read-only job with no sort order validation and no reduce tasks.
TeraSort	Job with 1:1:1 HDFS read, shuffle, and write.
WordCount	CPU-heavy job that heavily uses map-side combiner.
Shuffle	Shuffle-only job modified from `randomtextwriter` in `hadoop-examples`.

5.3 Performance Tuning

The objective of performance tuning is to discover guidelines on how to configure the disk I/O scheduling to achieve improved performance for the workloads we plan to run. For example, what's the recommend vdisks value requested for different types of workloads (e.g., I/O-light, I/O-medium, and I/O-heavy). We'll re-deploy the workloads mentioned in Section 5.2 with various `vdisks` requests to gain more tuning suggestions.

6 Follow-up items

6.1 Spindle Locality

Certain applications might require spindle locality for reliable throughput; e.g. a consumer process reading data from a producer writing to a particular spindle. Even for applications that don't require a particular spindle, the NM should spread the containers across spindles for better performance.

We see two ways of achieving spindle locality:

1. Applications can request for specific spindles from the RM as part of the resource-request-allocate. The RM will have to maintain information about how much of each spindle is currently being used to make this call. This can get pretty expensive (10,000 nodes * 100 spindles * 100 B-per-spindle = 100 GB), at least enough to cause GCs.
2. Applications request disk resources without locality from the RM, and specify their locality request to the NM when launching the container. This way, the RM need not

maintain and sift through locality information. However, the AMs might have to deal with the NM failing to launch a container.

6.2 Buffered Write Isolation

For better performance, the OS buffers writes. Currently, Cgroups' `blkio` subsystem can only support sync I/O queues, and cannot enforce buffered write traffic (buffered read is ok)[1], because all the buffered writes are system wide and not per group/process. The right way to solve this would be with support in the kernel for buffered-writes. Although we only support read I/O traffic enforcement in phase 1, it is still desirable for scenarios having lots of read-heavy BI workloads.

To solve the buffered write problem, one approach is to add a FUSE[2] wrapper that overlay on top of the local file system. Since the wrapper can see all I/O traffic, it can easily perform monitoring and rate limiting to enforce the sharing policy.

[1] https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt

[2] http://fuse.sourceforge.net/.