

# Disk I/O Scheduling in Hadoop YARN Cluster (Draft)

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

## [1. Motivation](#)

## [2. Design](#)

### [2.1 Disk I/O Resource Description](#)

### [2.2 Scheduler-Side](#)

### [2.3 Enforcement by NodeManager](#)

## [3. Implementation](#)

### [3.1 Phase 1](#)

#### [3.1.1 Disk I/O Resource Description](#)

#### [3.1.2 Scheduler-side Changes](#)

#### [3.1.3 NodeManager-side Enforcement](#)

##### [3.1.3.1 Prepare](#)

##### [3.1.3.2 For Newly Launched Container](#)

### [3.2 Phase 2](#)

## [4 Other Solutions](#)

### [4.1 Google's Imctfy \(Let Me Containe That For You\)](#)

## 1. Motivation

Resource sharing and isolation techniques in Hadoop YARN have been studied and implemented for resources such as CPUs and memory. Individual tasks (or containers) can be limited on the number of CPU cores and the amount of memory they are allowed to use. However, YARN doesn't support I/O resource management and isolation. The co-located tasks with various I/O requirements may compete with each other. This document focuses on disk I/O resource scheduling, and provides a solution for I/O resource scheduling and isolation.

Setting up a feasible disk I/O resource scheduling mechanism is desirable. Individual tasks have diverse disk I/O requests. Scheduling two disk I/O-intensive tasks together on a node which cannot fully satisfy both requests would cause them to compete for the I/O and slow each other down. Additionally, without isolating disk I/O resources, it is hard for YARN to deliver a stable performance for I/O-intensive tasks. For example, scheduling an I/O-intensive task in a I/O-light node may run faster than in a I/O-heavy node.

To provide a desirable disk I/O resource isolation, this document introduces fair sharing mechanism for disk I/O management. Each task can specify its own disk I/O requirement, and the scheduler can guarantee the amount of I/O resource assigned to each task. This way, we can alleviate the I/O resource competition and make sure some higher priority tasks (with higher

I/O resource request) wouldn't be blocked by lower priority tasks (with lower I/O resource request).

## 2. Design

This section mainly discusses the design of disk I/O scheduling, including: (1) How to describe the disk I/O resource? (2) How does the scheduler factor in disk I/O resource? (3) How does each NodeManager enforce the disk I/O resource usages for each local task?

### 2.1 Disk I/O Resource Description

For disk I/O resource description, here we introduce an additional resource dimension `vdisks` (virtual disks) to existing resource types (CPU and memory). Each task can specify its own disk I/O resource request via `vdisks` according to their I/O request. The larger `vdisks` means the container needs more disk I/O bandwidth. The default `vdisks` value is set to 1, which means I/O-light. Note that `vdisks` doesn't expose the heterogeneity among disks on a node, which is out of scope for this work.

### 2.2 Scheduler-Side

The major change in the scheduler-side is to include `vdisks` to the Dominant Resource Fairness (DRF) policy. Currently, the DRF in scheduler supports `cpu` and `memory`. We also introduce configurations for users to specify which resources should be included into the DRF.

### 2.3 Enforcement by NodeManager

In summary, we would like to provide proportional disk I/O fair sharing. The available disk I/O resource is shared among all tasks in a proportional way (the `vdisks` value). We use Cgroups to implement this solution. Here we first illustrate functions provided by Cgroups and then discuss the implementation details.

Cgroups provides a subsystem called `blkio`, which offers two policies for controlling access to disk I/O:

- 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 performed by a specific device. This means that a device can have a limited rate of read or write operations.

We use the proportional weight division policy to provide a weighted fair sharing among all containers. If a container specifies its disk I/O request as  $p$ , we can set the weight for that container is  $p$  directly.

## 3. Implementation

### 3.1 Phase 1

Here we summarize the major changes in the code base.

#### 3.1.1 Disk I/O Resource Description

**Summary:** Add the disk I/O dimension (`vdisks`) to the resource request.

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`.

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

#### 3.1.2 Scheduler-side Changes

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

- Configure whether the scheduler does disk I/O scheduling. If disabled, the scheduler only works on memory and vcores.
- Add `vdisks` to DRF policy, and configure which resources are included when calculating DRF.

```
+ /** Enable vdisks scheduling or not. */
+ protected static final String VDISKS_SCHEDULING_ENABLED = CONF_PREFIX +
+ "vdisks-scheduling-enabled";
+ protected static final boolean DEFAULT_VDISKS_SCHEDULING_ENABLED = false;

+ /** Enable memory/vcores/vdisks in DRF or not. */
+ protected static final String DRF_MEMORY_ENABLED = CONF_PREFIX +
+ "drf.memory.enabled";
+ protected static final boolean DEFAULT_DRF_MEMORY_ENABLED = true;
+ protected static final String DRF_VCORES_ENABLED = CONF_PREFIX +
+ "drf.vcores.enabled";
+ protected static final boolean DEFAULT_DRF_VCORES_ENABLED = true;
+ protected static final String DRF_VDISKS_ENABLED = CONF_PREFIX +
+ "drf.vdisks.enabled";
+ protected static final boolean DEFAULT_DRF_VDISKS_ENABLED = true;
```

### 3.1.3 NodeManager-side Enforcement

**Summary:** Add support for NodeManager to enforce the disk I/O resource usage for each task.

#### 3.1.3.1 Prepare

Make changes to

`org.apache.hadoop.yarn.server.nodemanager.util.CgroupsLCEResourcesHandler` to mount the Cgroups path to include `blkio`.

#### 3.1.3.2 For Newly Launched Container

In general, for each newly launched container `C` with `vdisks p`, we create a new cgroup under the `blkio` subsystem. The cgroup's weight for all disk devices is set to `p`.

```
mkdir /sys/fs/cgroup/blkio/C_ID
echo p > /sys/fs/cgroup/blkio/C_ID/blkio.weight
```

And add `C`'s process to the cgroup's `tasks` file.

```
echo C_PID > /sys/fs/cgroup/blkio/C_ID/tasks
```

Once the container finishes, remove the created cgroup.

## 3.2 Phase 2

**Buffered write isolation:** Currently, Cgroups' `blkio` subsystem can only support sync I/O queues, and cannot enforce buffered write traffic (buffered read is ok)<sup>1</sup>, 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<sup>2</sup> 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.

---

<sup>1</sup> <https://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt>

<sup>2</sup> <http://fuse.sourceforge.net/>.