Joe Stein

Building & Deploying Applications to Apache Mesos

Joe Stein

- Developer, Architect & Technologist
- Founder & Principal Consultant => Big Data Open Source Security LLC - http://stealth.ly

Big Data Open Source Security LLC provides professional services and product solutions for the collection, storage, transfer, real-time analytics, batch processing and reporting for complex data streams, data sets and distributed systems. BDOSS is all about the "glue" and helping companies to not only figure out what Big Data Infrastructure Components to use but also how to change their existing (or build new) systems to work with them.

- Apache Kafka Committer & PMC member
- Blog & Podcast - http://allthingshadoop.com
- Twitter @allthingshadoop

Overview

- Quick intro to Mesos
- Marathon Framework
- Aurora Framework
- Custom Framework


MESOS
ALL THE THINGS
memegenerator.net

Origins

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center http://static.usenix.org/event/nsdi11/tech/full_papers/Hindman_new.pdf

Datacenter Management with Apache Mesos https://www.youtube.com/watch?v=YB1VW0LKzJ4

Omega: flexible, scalable schedulers for large compute clusters http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf

2011 GAFS Omega John Wilkes https://www.youtube.com/watch?v=0ZFMlO98Jkc&feature=youtu.be

Life without Apache Mesos

Static Partitioning

Static Partitioning IS Bad

hard to utilize machines
(i.e., X GB RAM and Y CPUs)

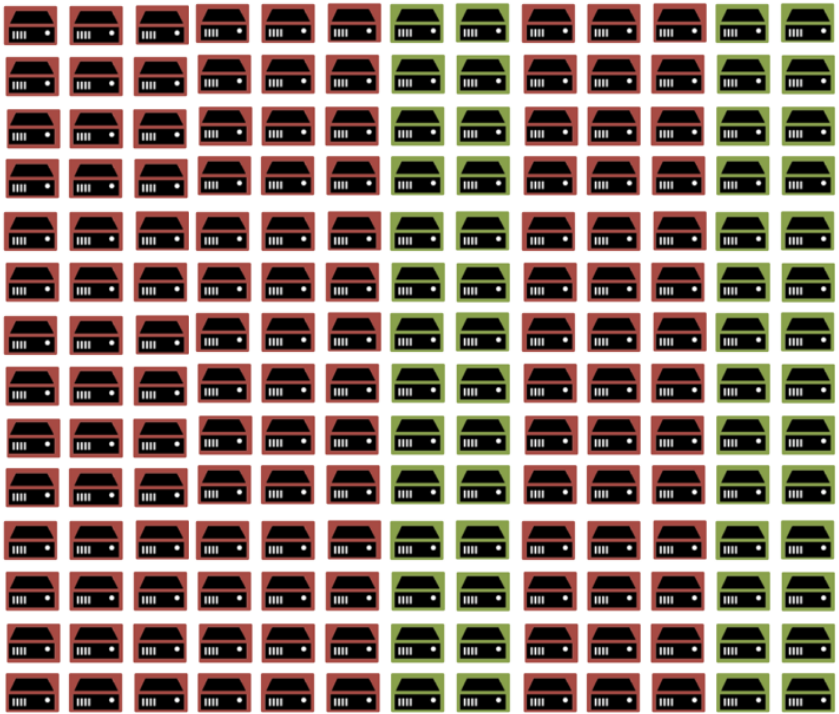
Static Partitioning does NOT scale

hard to scale elastically
(to take advantage of statistical multiplexing)

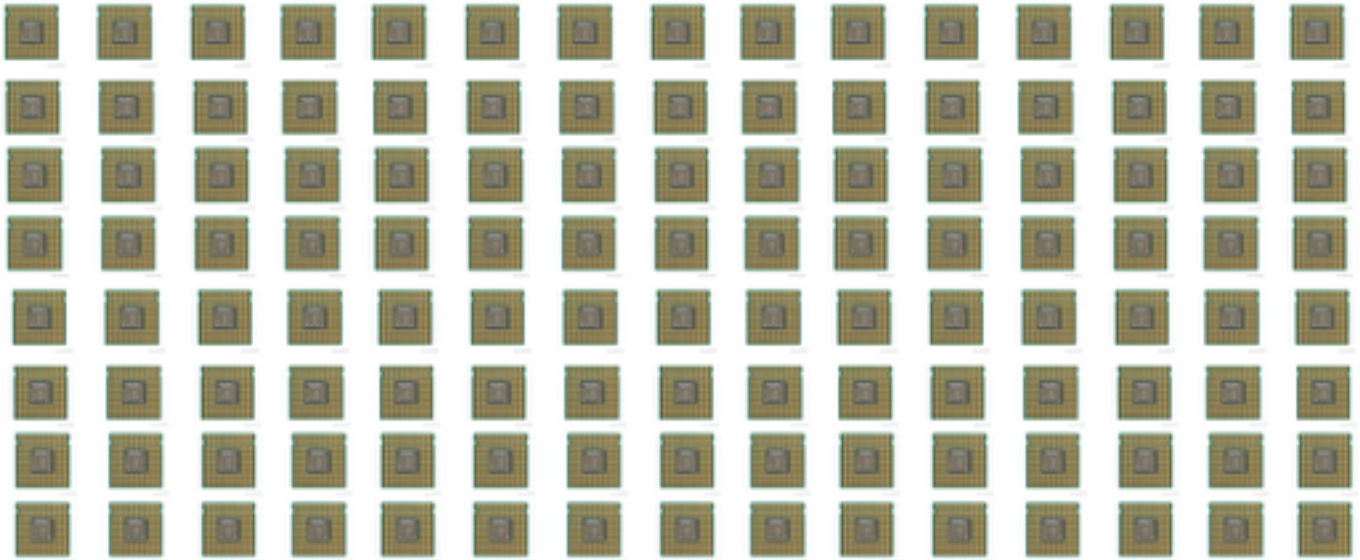
Failures === Downtime

hard to deal with failures

Static Partitioning



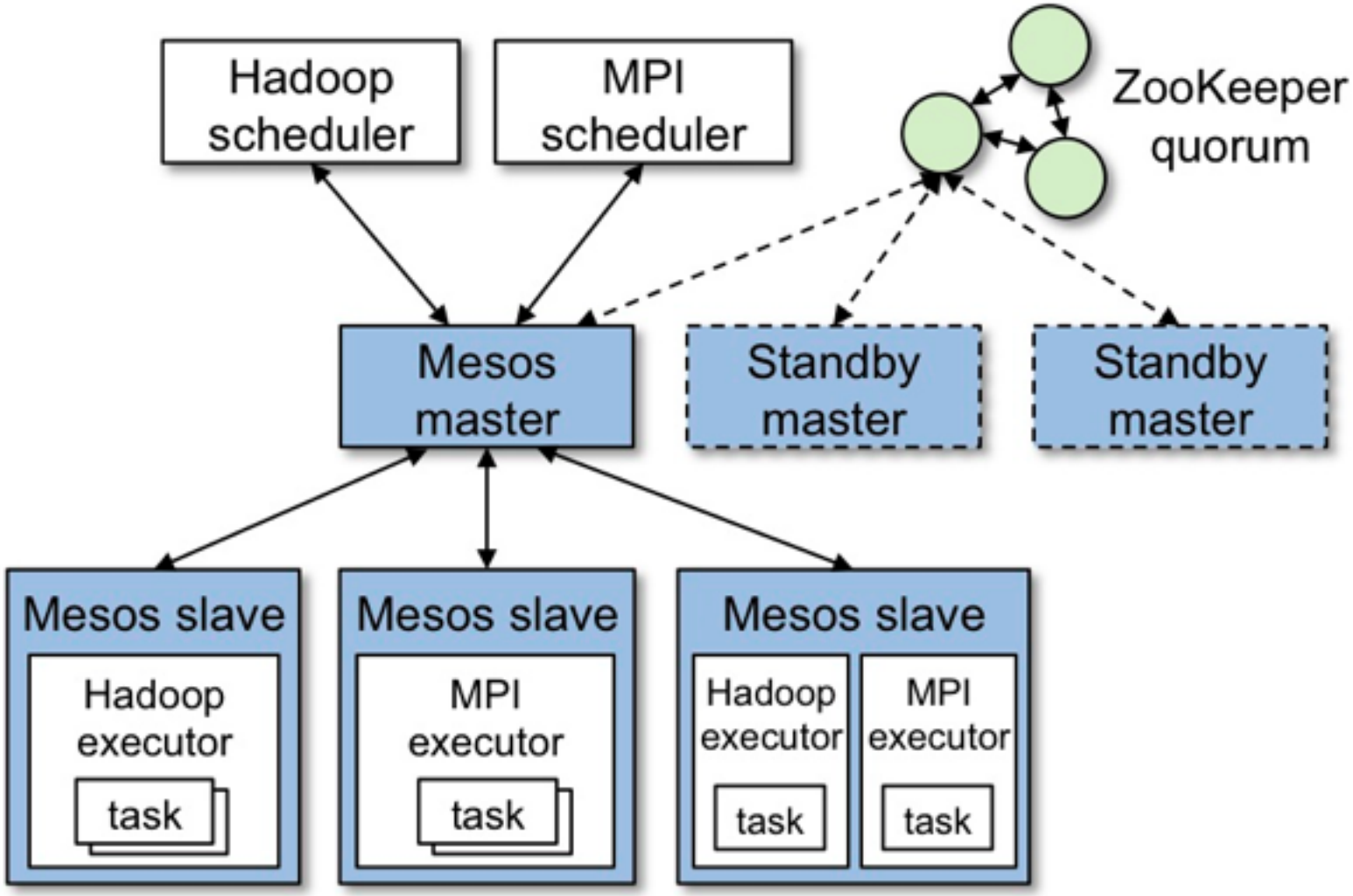
Operating System === Datacenter

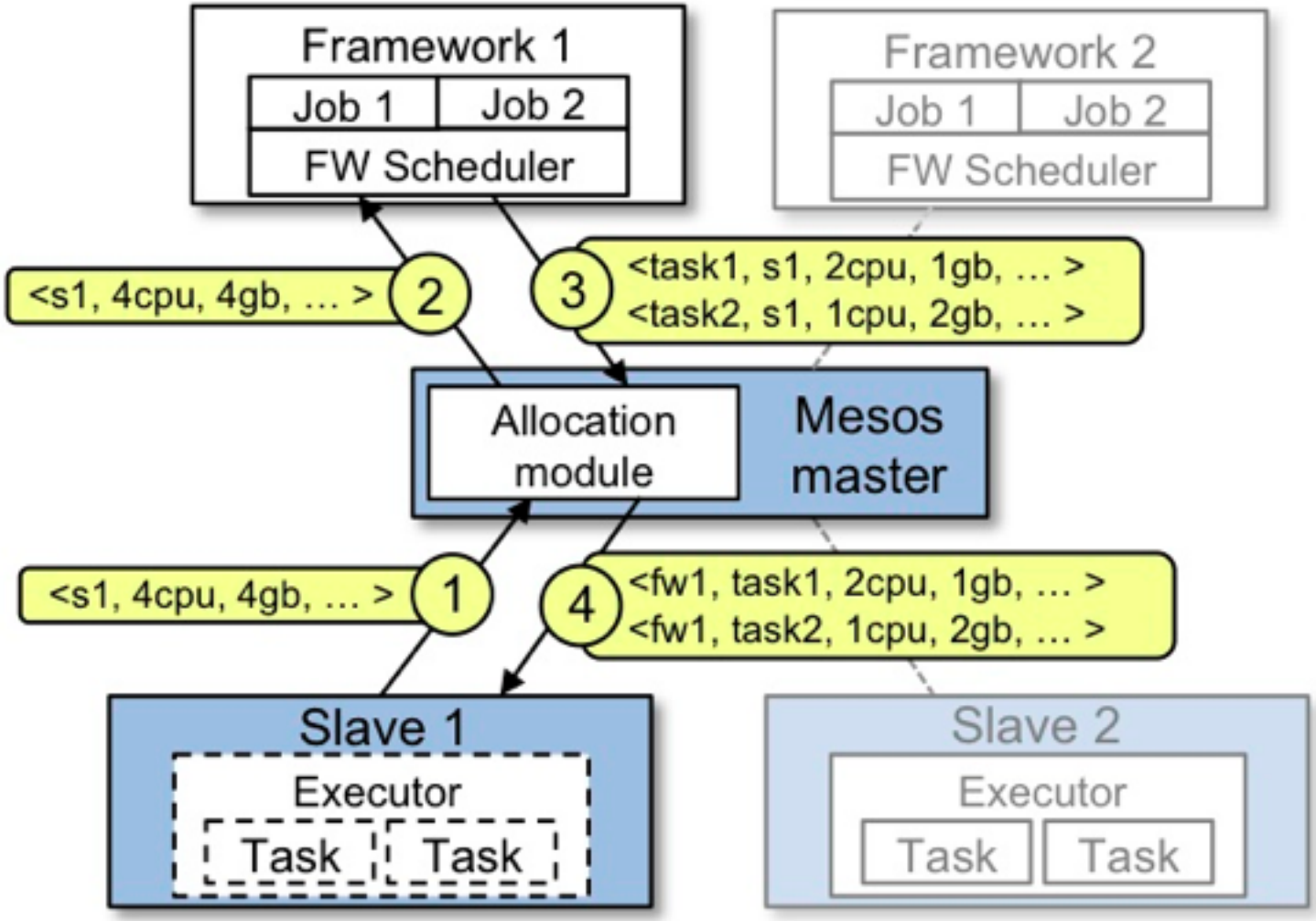


Mesos => data center “kernel”

Apache Mesos

- Scalability to 10,000s of nodes
- Fault-tolerant replicated master and slaves using ZooKeeper
- Support for Docker containers
- Native isolation between tasks with Linux Containers
- Multi-resource scheduling (memory, CPU, disk, and ports)
- Java, Python and C++ APIs for developing new parallel applications
- Web UI for viewing cluster state


Hadoop scheduler
MPI scheduler
ZooKeeper quorum
Mesos master
Standby master
Standby master
Mesos slave
Hadoop executor
task
Mesos slave
MPI executor
task
Mesos slave
Hadoop executor
task
MPI executor
task


Framework 1
Job 1
Job 2
FW Scheduler
Framework 2
Job 1
Job 2
FW Scheduler
<s1, 4cpu, 4gb, … >
2
3
<task1, s1, 2cpu, 1gb, … >
<task2, s1, 1cpu, 2gb, … >
Allocation module
Mesos master
<s1, 4cpu, 4gb, … >
1
4
<fw1, task1, 2cpu, 1gb, … >
<fw1, task2, 1cpu, 2gb, … >
Slave 1
Executor
Task
Task
Slave 2
Executor
Task
Task

Resources & Attributes

The Mesos system has two basic methods to describe the slaves that comprise a cluster. One of these is managed by the Mesos master, the other is simply passed onwards to the frameworks using the cluster.

Attributes

The attributes are simply key value string pairs that Mesos passes along when it sends offers to frameworks.

```
attributes : attribute ( ";" attribute )*

attribute : labelString ":" ( labelString | "," )+
```

Resources

The Mesos system can manage 3 different *types* of resources: scalars, ranges, and sets. These are used to represent the different resources that a Mesos slave has to offer. For example, a scalar resource type could be used to represent the amount of memory on a slave. Each resource is identified by a key string.

```
resources : resource ( ";" resource )*
resource : key ":" ( scalar | range | set )
key : labelString ( "(" resourceRole ")" )?
scalar : floatValue
range : "[" rangeValue ( "," rangeValue )* "]"
rangeValue : scalar "-" scalar
set : "{" labelString ( "," labelString )* "}"
resourceRole : labelString | "*"
labelString : [a-zA-Z0-9_/.-]
floatValue : ( intValue ( "." intValue )? ) | ...
intValue : [0-9]+
```

Predefined Uses & Conventions

The Mesos master has a few resources that it pre-defines in how it handles them. At the current time, this list consist of:

- `cpu`
- `mem`
- `disk`
- `ports`

In particular, a slave without `cpu` and `mem` resources will never have its resources advertised to any frameworks. Also, the Master's user interface interprets the scalars in`mem` and `disk` in terms of `MB`. IE: the value `15000` is displayed as `14.65GB`.

Examples

Here are some examples for configuring the Mesos slaves.

```
--resources='cpu:24;mem:24576;disk:409600;ports:[21000-24000];disks:{1,2,3,4,5,6,7,8}'
--attributes='dc:1;floor:2;aisle:6;rack:aa;server:15;os:trusty;jvm:7u68;docker:1.5'
```

In this case, we have three different types of resources, scalars, a range, and a set. They are called `cpu`, `mem`, `disk`, and the range type is `ports`.

- scalar called `cpu`, with the value `24`
- scalar called `mem`, with the value `24576`
- scalar called `disk`, with the value `409600`
- range called `ports`, with values `21000` through `24000` (inclusive)
- set called `disks`, with the values `1`, `2`, etc which we can concatenate later /mnt/disk/{diskNum} and each task can own a disk

Roles

Total consumable resources per slave, in the form 'name(role):value;name(role):value...'. This value can be set to limit resources per role, or to overstate the number of resources that are available to the slave.

```
--resources="cpus(*):8; mem(*):15360; disk(*):710534; ports(*):[31000-32000]"
--resources="cpus(prod):8; cpus(stage):2 mem(*):15360; disk(*):710534; ports(*):[31000-32000]"
```

All * roles will be detected, so you can specify only the resources that are not all roles (*). `--resources="cpus(prod):8; cpus(stage)"`

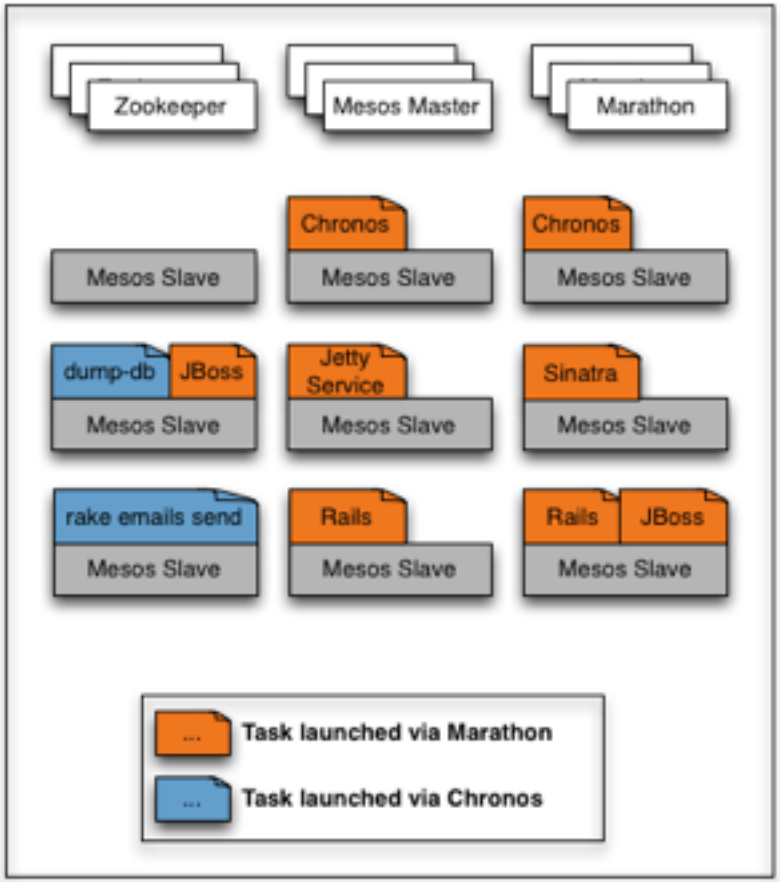
Frameworks bind a specific roles or any. A default roll (instead of *) can also be configured.

Roles can be used to isolate and segregate frameworks.

Marathon

https://github.com/mesosphere/marathon

Cluster-wide init and control system for services in cgroups or docker based on Apache Mesos



Constraints

Constraints control where apps run to allow optimizing for fault tolerance or locality. Constraints are made up of three parts: a field name, an operator, and an optional parameter. The field can be the slave hostname or any Mesos slave attribute.

Fields

Hostname field

`hostname` field matches the slave hostnames, see `UNIQUE operator` for usage example.

`hostname` field supports all operators of Marathon.

Attribute field

If the field name is none of the above, it will be treated as a Mesos slave attribute. Mesos slave attribute is a way to tag a slave node, see `mesos-slave --help` to learn how to set the attributes.

Unique

`UNIQUE` tells Marathon to enforce uniqueness of the attribute across all of an app's tasks. For example the following constraint ensures that there is only one app task running on each host:

via the Marathon gem:

```
$ marathon start -i sleep -C 'sleep 60' -n 3 --constraint hostname:UNIQUE
```

via curl:

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v1/apps/start -d '{
    "id": "sleep-unique",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["hostname", "UNIQUE"]]
  }'
```

Cluster

`CLUSTER` allows you to run all of your app's tasks on slaves that share a certain attribute.

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v1/apps/start -d '{
    "id": "sleep-cluster",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["rack_id", "CLUSTER", "rack-1"]]
  }'
```

You can also use this attribute to tie an application to a specific node by using the hostname property:

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v1/apps/start -d '{
    "id": "sleep-cluster",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["hostname", "CLUSTER", "a.specific.node.com"]]
  }'
```

Group By

`GROUP_BY` can be used to distribute tasks evenly across racks or datacenters for high availability.

via the Marathon gem:

```
$ marathon start -i sleep -C 'sleep 60' -n 3 --constraint rack_id:GROUP_BY
```

via curl:

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v1/apps/start -d '{
    "id": "sleep-group-by",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["rack_id", "GROUP_BY"]]
  }'
```

Optionally, you can specify a minimum number of groups to try and achieve.

Like

`LIKE` accepts a regular expression as parameter, and allows you to run your tasks only on the slaves whose field values match the regular expression.

via the Marathon gem:

```
$ marathon start -i sleep -C 'sleep 60' -n 3 --constraint rack_id:LIKE:rack-[1-3]
```

via curl:

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v1/apps/start -d '{
    "id": "sleep-group-by",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["rack_id", "LIKE", "rack-[1-3]"]]
  }'
```

Unlike

Just like `LIKE` operator, but only run tasks on slaves whose field values don't match the regular expression.

via the Marathon gem:

```
$ marathon start -i sleep -C 'sleep 60' -n 3 --constraint rack_id:UNLIKE:rack-[7-9]
```

via curl:

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v1/apps/start -d '{
    "id": "sleep-group-by",
    "cmd": "sleep 60",
    "instances": 3,
    "constraints": [["rack_id", "UNLIKE", "rack-[7-9]"]]
  }'
```

Running in Marathon

TAG=sample
APP=xyz
ID=$TAG-$APP
CMD=”./yourScript "'$HOST'" "'$PORT0'" "'$PORT1'

JSON=$(printf '{ "id": "%s", "cmd": "%s", "cpus": %s, "mem": %s, "instances": %s, "uris": ["%s"], "ports": [0,0] , “env”:{ “JAVA_OPTS”,”%s”}' "$ID" “$CMD" "0.1" "256" "1" "http://dns/path/yourScriptAndStuff.tgz" “-Xmx 128”)

curl -i -X POST -H "Content-Type: application/json" -d "$JSON" http://localhost:8080/v2/apps

yourScript

#think of it as a distributed application launching in the cloud

HOST=$1
PORT0=$2
PORT1=$3

#talk to zookeeper or etcd or mesos dns

#call marathon rest api

#spawn another process, they are all in your cgroup =8^) woot woot

Persisting Data

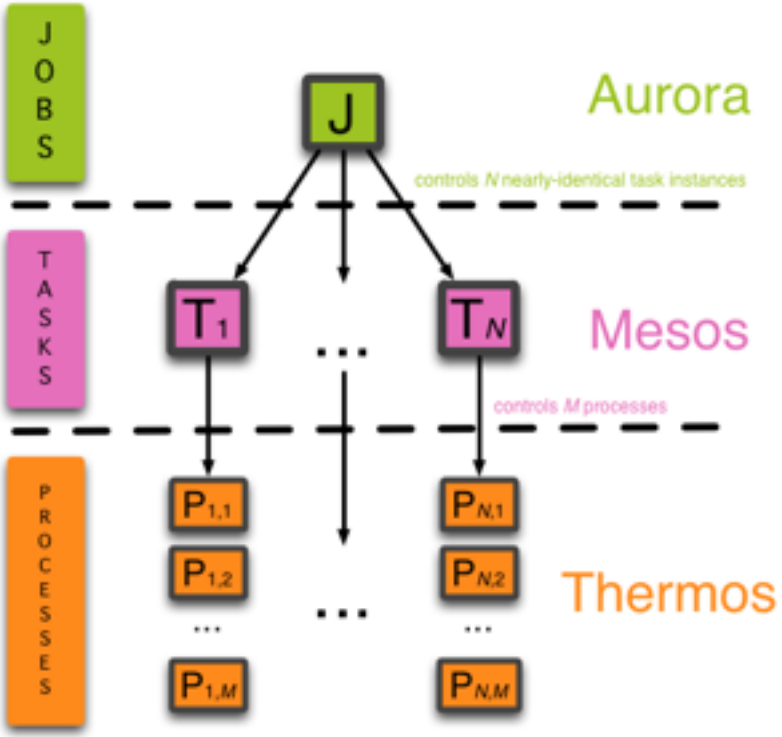
- Write data outside the sandbox, the other kids will not mind.
  - /var/lib/data/<tag>/<app>
- Careful about starvation, use the constraints and roles to manage this.
- Future improvements https://issues.apache.org/jira/browse/MESOS-2018 This is a feature to provide better support for running stateful services on Mesos such as HDFS (Distributed Filesystem), Cassandra (Distributed Database), or MySQL (Local Database). Current resource reservations (henceforth called "static" reservations) are statically determined by the slave operator at slave start time, and individual frameworks have no authority to reserve resources themselves. Dynamic reservations allow a framework to dynamically/lazily reserve offered resources, such that those resources will only be re-offered to the same framework (or other frameworks with the same role). This is especially useful if the framework's task stored some state on the slave, and needs a guaranteed set of resources reserved so that it can re-launch a task on the same slave to recover that state.

Apache Aurora

http://aurora.incubator.apache.org/

Apache Aurora is a service scheduler that runs on top of Mesos, enabling you to run long-running services that take advantage of Mesos' scalability, fault-tolerance, and resource isolation. Apache Aurora is currently part of the Apache Incubator.

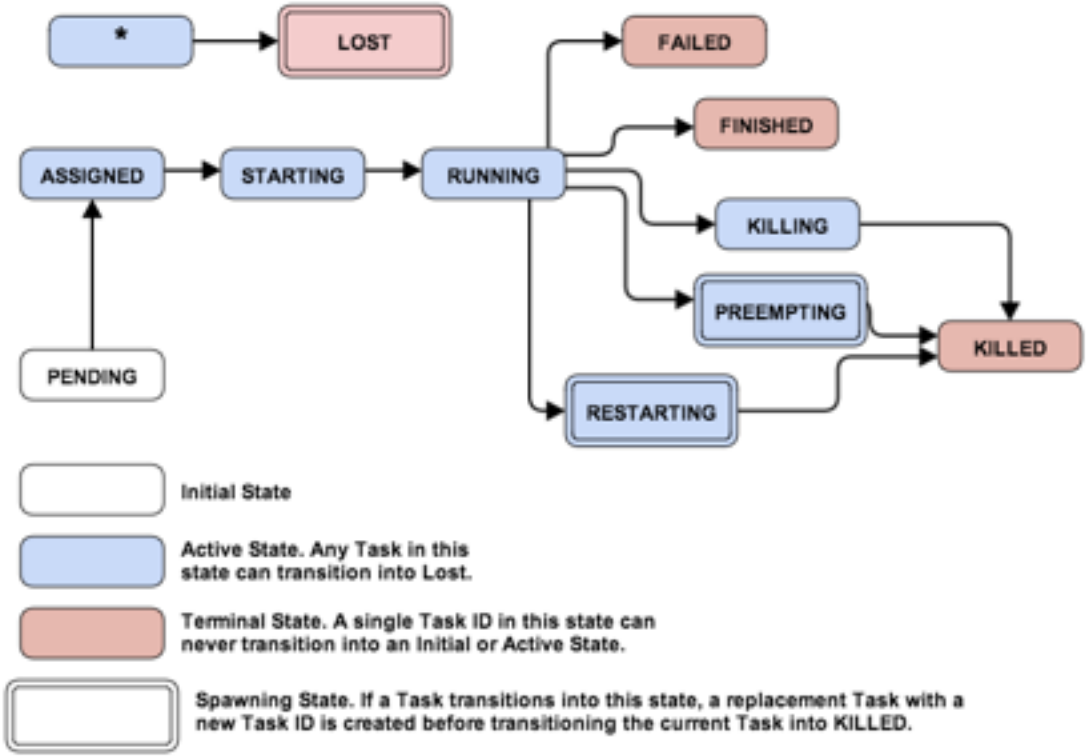
Thermos


JOBS
TASKS
PROCESSES
J
Aurora
controls N nearly-identical task instances
T1
TN
Mesos
controls M processes
P1,1
P1,2
P1,M
PN,1
PN,2
PN,M
Thermos

Job Objects

name	type	description
`task`	Task	The Task object to bind to this job. Required.
`name`	String	Job name. (Default: inherited from the task attribute's name)
`role`	String	Job role account. Required.
`cluster`	String	Cluster in which this job is scheduled. Required.
`environment`	String	Job environment, default `devel`. Must be one of `prod`, `devel`, `test` or `staging<number>`.
`contact`	String	Best email address to reach the owner of the job. For production jobs, this is usually a team mailing list.
`instances`	Integer	Number of instances (sometimes referred to as replicas or shards) of the task to create. (Default: 1)
`cron_schedule`	String	Cron schedule in cron format. May only be used with non-service jobs. See Cron Jobs for more information. Default: None (not a cron job.)
`cron_collision_policy`	String	Policy to use when a cron job is triggered while a previous run is still active. KILL*EXISTING Kill the previous run, and schedule the new run* CANCELNEW Let the previous run continue, and cancel the new run. (Default: KILL_EXISTING)
`update_config`	`update_config` object	Parameters for controlling the rate and policy of rolling updates.
`constraints`	dict	Scheduling constraints for the tasks. See the section on the constraint specification language
`service`	Boolean	If True, restart tasks regardless of success or failure. (Default: False)
`max_task_failures`	Integer	Maximum number of failures after which the task is considered to have failed (Default: 1) Set to -1 to allow for infinite failures
`priority`	Integer	Preemption priority to give the task (Default 0). Tasks with higher priorities may preempt tasks at lower priorities.
`production`	Boolean	Whether or not this is a production task backed by quota (Default: False). Production jobs may preempt any non-production job, and may only be preempted by production jobs in the same role and of higher priority. To run jobs at this level, the job role must have the appropriate quota.
`health_check_config`	`heath_check_config` object	Parameters for controlling a task's health checks via HTTP. Only used if a health port was assigned with a command line wildcard.
`container`	`Container` object	An optional container to run all processes inside of.

Job Lifecycle



hello_world.aurora

```
import os
hello_world_process = Process(name = 'hello_world', cmdline = 'echo hello world')
hello_world_task = Task(
  resources = Resources(cpu = 0.1, ram = 16 * MB, disk = 16 * MB),
  processes = [hello_world_process])
hello_world_job = Job(
  cluster = 'cluster1',
  role = os.getenv('USER'),
  task = hello_world_task)
jobs = [hello_world_job]
```

Then issue the following commands to create and kill the job, using your own values for the job key.

```
aurora job create cluster1/$USER/test/hello_world hello_world.aurora
```

```
aurora job kill cluster1/$USER/test/hello_world
```

Kafka on Aurora

```
import os
import textwrap

class Profile(Struct):
  archive = Default(String, 'https://archive.apache.org/dist')
  gpg_grp = Default(String, 'https://people.apache.org/keys/group')
  svc = Default(String, 'kafka')
  svc_ver = Default(String, '0.8.1.1')
  svc_ver_scala = Default(String, '2.10')
  svc_prop_file = Default(String, 'server.properties')
  jvm_heap = Default(String, '-Xmx1G -Xms1G')

common = Process(
  name = 'fetch commons',
  cmdline = textwrap.dedent("""
    hdfs dfs -get /dist/scripts.tgz
    tar xf scripts.tgz
  """)
)
```

Kafka on Aurora

```
dist = Process(
  name = 'fetch {{profile.svc}} v{{profile.svc_ver_scala}}-{{profile.svc_ver}} distribution',
  cmdline = textwrap.dedent("""
    command -v curl >/dev/null 2>&1 || { echo >&2 "error: 'curl' is not installed. Aborting."; exit 1; }
    eval curl -sSfL '-O {{profile.archive}}/kafka/{{profile.svc_ver}}/kafka_{{profile.svc_ver_scala}}-
{{profile.svc_ver}}.tgz'{,.asc,.md5}
    if command -v md5sum >/dev/null; then
      md5sum -c kafka_{{profile.svc_ver_scala}}-{{profile.svc_ver}}.tgz.md5
    else
      echo "warn: 'md5sum' is not installed. Check skipped." >&2
    fi
    if command -v gpg >/dev/null; then
      curl -sSfL {{profile.gpg_grp}}/kafka.asc | gpg --import
      gpg kafka_{{profile.svc_ver_scala}}-{{profile.svc_ver}}.tgz.asc
    else
      echo "warn: 'gpg' is not installed. Signature verification skipped." >&2
    fi
    tar xf kafka_{{profile.svc_ver_scala}}-{{profile.svc_ver}}.tgz
  """)
)
```

Kafka on Aurora

```
register = Process(
  name = 'register-service',
  cmdline = textwrap.dedent("""
    export IP=$(host `hostname` | tr ' ' '\n' | tail -1)
    ./scripts/common/registry.sh -r {{profile.svc}}-{{environment}} -i {{mesos.instance}} -p "$IP:{{thermos.ports[client]}}"
  """)
)

unregister = Process(
  name = 'unregister-service',
  final = True,
  cmdline = textwrap.dedent("""
    ./scripts/common/registry.sh -u {{profile.svc}}-{{environment}} -i {{mesos.instance}}
  """)
)
```

Kafka on Aurora

```
config = Process(
  name = 'create {{profile.svc_prop_file}}',
  cmdline = textwrap.dedent("""
    export ZK_CONNECT=$(echo -e `./scripts/common/registry.sh -q zookeeper-{{environment}}-client` | awk '{printf("%s,", $0)}')
    export IP=$(host `hostname` | tr ' ' '\n' | tail -1)
    echo -e "
    broker.id={{mesos.instance}}
    port={{thermos.ports[client]}}
    host.name=$IP
    advertised.host.name=$IP
    num.network.threads=2
    num.io.threads=8
    socket.send.buffer.bytes=1048576
    socket.receive.buffer.bytes=1048576
    socket.request.max.bytes=104857600
    log.dirs=kafka-logs
    num.partitions=1
    log.retention.hours=168
    log.segment.bytes=536870912
    log.retention.check.interval.ms=60000
    log.cleaner.enable=false
    zookeeper.connect=$ZK_CONNECT
    zookeeper.connection.timeout.ms=30000
    " > {{profile.svc_prop_file}}

    echo "Wrote '{{profile.svc_prop_file}}':"
    cat {{profile.svc_prop_file}}
  """)
)
```

Kafka on Aurora

```
run = Process(
  name = 'run {{profile.svc}}',
  cmdline = textwrap.dedent("""
    export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:kafka_{{profile.svc_ver_scala}}-{{profile.svc_ver}}/config/
log4j.properties"
    export KAFKA_HEAP_OPTS="{{profile.jvm_heap}}"
    export EXTRA_ARGS="-name kafkaServer -loggc"

    kafka_{{profile.svc_ver_scala}}-{{profile.svc_ver}}/bin/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka
{{profile.svc_prop_file}}
  """)
)
```

Kafka on Aurora

```
base_task = Task(
  processes = [register, unregister, common, dist, config, run],
  constraints =
    order(dist, run) +
    order(common, register) +
    order(common, config, run)
)

staging_task = base_task(
  resources = Resources(cpu = 1.0, ram = 1280*MB, disk = 5*GB)
)

production_task = base_task(
  resources = Resources(cpu = 24.0, ram = 23040*MB, disk = 10000*GB)
)
```

Kafka on Aurora

```
DEVELOPMENT = Profile()
PRODUCTION = Profile(
  jvm_heap = '-Xmx4G -Xms4G'
)
base_job = Service(
  name = 'kafka',
  role = os.getenv('USER')
)
jobs = [
  base_job(
    cluster = dc1,
    environment = 'devel',
    instances = 4000,
    contact = ‘root@localhost',
    task = staging_task.bind(
      profile = DEVELOPMENT
    )
  ),
  base_job(
    cluster = ‘dc1’,
    environment = 'prod',
    instances = 150,
    production = True,
    contact = ‘root@localhost',
    task = production_task.bind(
      profile = PRODUCTION
    )
  )
]
```

The future of Kafka on Mesos

- A Mesos Kafka framework https://github.com/mesos/kafka

Sample Frameworks

C++ - https://github.com/apache/mesos/tree/master/src/examples

Java - https://github.com/apache/mesos/tree/master/src/examples/java

Python - https://github.com/apache/mesos/tree/master/src/examples/python

Scala - https://github.com/mesosphere/scala-sbt-mesos-framework.g8

Go - https://github.com/mesosphere/mesos-go

FrameworkInfo

https://github.com/apache/mesos/blob/master/include/mesos/mesos.proto

```
message FrameworkInfo {
 required string user = 1;
 required string name = 2;
 optional FrameworkID id = 3;
 optional double failover_timeout = 4 [default = 0.0];
 optional bool checkpoint = 5 [default = false];
 optional string role = 6 [default = "*"];
 optional string hostname = 7;
 optional string principal = 8;
 optional string webui_url = 9;
}
```

TaskInfo

```
message TaskInfo {
 required string name = 1;
 required TaskID task_id = 2;
 required SlaveID slave_id = 3;
 repeated Resource resources = 4;
 optional ExecutorInfo executor = 5;
 optional CommandInfo command = 7;
 optional ContainerInfo container = 9;
 optional bytes data = 6;
 optional HealthCheck health_check = 8;

 optional Labels labels = 10;

 optional DiscoveryInfo discovery = 11;
}
```

TaskState

```
/**
* Describes possible task states. IMPORTANT: Mesos assumes tasks that
* enter terminal states (see below) imply the task is no longer
* running and thus clean up any thing associated with the task
* (ultimately offering any resources being consumed by that task to
* another task).
*/
enum TaskState {
 TASK_STAGING = 6; // Initial state. Framework status updates should not use.
 TASK_STARTING = 0;
 TASK_RUNNING = 1;
 TASK_FINISHED = 2; // TERMINAL. The task finished successfully.
 TASK_FAILED = 3; // TERMINAL. The task failed to finish successfully.
 TASK_KILLED = 4; // TERMINAL. The task was killed by the executor.
 TASK_LOST = 5; // TERMINAL. The task failed but can be rescheduled.
 TASK_ERROR = 7; // TERMINAL. The task description contains an error.
}
```

Scheduler

```
/**
   * Invoked when the scheduler successfully registers with a Mesos
   * master. A unique ID (generated by the master) used for
   * distinguishing this framework from others and MasterInfo
   * with the ip and port of the current master are provided as arguments.
   */
  def registered(
    driver: SchedulerDriver,
    frameworkId: FrameworkID,
    masterInfo: MasterInfo): Unit = {
    log.info("Scheduler.registered")
    log.info("FrameworkID:\n%s" format frameworkId)
    log.info("MasterInfo:\n%s" format masterInfo)
  }
```

Scheduler

```
/**
   * Invoked when the scheduler re-registers with a newly elected Mesos master.
   * This is only called when the scheduler has previously been registered.
   * MasterInfo containing the updated information about the elected master
   * is provided as an argument.
   */
  def reregistered(
    driver: SchedulerDriver,
    masterInfo: MasterInfo): Unit = {
    log.info("Scheduler.reregistered")
    log.info("MasterInfo:\n%s" format masterInfo)
  }
```

Scheduler

```
/**Invoked when resources have been offered to this framework. A single offer will only contain resources from a
   single slave. Resources associated with an offer will not be re-offered to _this_ framework until either (a) this
   framework has rejected those resources or (b) those resources have been rescinded.Note that resources may be
   concurrently offered to more than one framework at a time (depending on the allocator being used). In that case,
   the first framework to launch tasks using those resources will be able to use them while the other frameworks will
   have those resources rescinded (or if a framework has already launched tasks with those resources then those
   tasks will fail with a TASK_LOST status and a message saying as much).*/
  def resourceOffers(
    driver: SchedulerDriver, offers: JList[Offer]): Unit = {
    log.info("Scheduler.resourceOffers")
    // print and decline all received offers
    offers foreach { offer =>
      log.info(offer.toString)
      driver declineOffer offer.getId
    }
  }
```

Scheduler

```
/**
   * Invoked when an offer is no longer valid (e.g., the slave was
   * lost or another framework used resources in the offer). If for
   * whatever reason an offer is never rescinded (e.g., dropped
   * message, failing over framework, etc.), a framwork that attempts
   * to launch tasks using an invalid offer will receive TASK_LOST
   * status updats for those tasks.
   */
  def offerRescinded(
    driver: SchedulerDriver,
    offerId: OfferID): Unit = {
    log.info("Scheduler.offerRescinded [%s]" format offerId.getValue)
  }
```

Scheduler

```
/**
   * Invoked when the status of a task has changed (e.g., a slave is
   * lost and so the task is lost, a task finishes and an executor
   * sends a status update saying so, etc). Note that returning from
   * this callback _acknowledges_ receipt of this status update! If
   * for whatever reason the scheduler aborts during this callback (or
   * the process exits) another status update will be delivered (note,
   * however, that this is currently not true if the slave sending the
   * status update is lost/fails during that time).
   */
  def statusUpdate(
    driver: SchedulerDriver, status: TaskStatus): Unit = {
    log.info("Scheduler.statusUpdate:\n%s" format status)
  }
```

Scheduler

```
/**
  * Invoked when an executor sends a message. These messages are best
  * effort; do not expect a framework message to be retransmitted in
  * any reliable fashion.
  */
def frameworkMessage(
  driver: SchedulerDriver,
  executorId: ExecutorID,
  slaveId: SlaveID,
  data: Array[Byte]): Unit = {
  log.info("Scheduler.frameworkMessage")
}
```

Scheduler

```
**
   * Invoked when the scheduler becomes "disconnected" from the master
   * (e.g., the master fails and another is taking over).
   */
  def disconnected(driver: SchedulerDriver): Unit = {
    log.info("Scheduler.disconnected")
  }

  /**
   * Invoked when a slave has been determined unreachable (e.g.,
   * machine failure, network partition). Most frameworks will need to
   * reschedule any tasks launched on this slave on a new slave.
   */
  def slaveLost(
    driver: SchedulerDriver,
    slaveId: SlaveID): Unit = {
    log.info("Scheduler.slaveLost: [%s]" format slaveId.getValue)
  }
```

Scheduler

```
/**
   * Invoked when an executor has exited/terminated. Note that any
   * tasks running will have TASK_LOST status updates automagically
   * generated.
   */
  def executorLost(
    driver: SchedulerDriver,executorId: ExecutorID, slaveId: SlaveID,
    status: Int): Unit = {
    log.info("Scheduler.executorLost: [%s]" format executorId.getValue)
  }
  /**
    * Invoked when there is an unrecoverable error in the scheduler or
    * scheduler driver. The driver will be aborted BEFORE invoking this
    * callback.
    */
  def error(driver: SchedulerDriver, message: String): Unit = {
    log.info("Scheduler.error: [%s]" format message)
  }
```

Executor

```
/**
   * Invoked once the executor driver has been able to successfully
   * connect with Mesos. In particular, a scheduler can pass some
   * data to it's executors through the ExecutorInfo.data
   * field.
   */
  def registered(
    driver: ExecutorDriver,
    executorInfo: ExecutorInfo,
    frameworkInfo: FrameworkInfo,
    slaveInfo: SlaveInfo): Unit = {
    log.info("Executor.registered")
  }
```

Executor

```
/**
   * Invoked when the executor re-registers with a restarted slave.
   */
  def reregistered(
    driver: ExecutorDriver,
    slaveInfo: SlaveInfo): Unit = {
    log.info("Executor.reregistered")
  }
  /** Invoked when the executor becomes "disconnected" from the slave
    * (e.g., the slave is being restarted due to an upgrade).
    */
  def disconnected(driver: ExecutorDriver): Unit = {
    log.info("Executor.disconnected")
  }
```

Executor

```
/**
   * Invoked when a task has been launched on this executor (initiated
   * via Scheduler.launchTasks. Note that this task can be
   * realized with a thread, a process, or some simple computation,
   * however, no other callbacks will be invoked on this executor
   * until this callback has returned.
   */
  def launchTask(driver: ExecutorDriver, task: TaskInfo): Unit = {
    log.info("Executor.launchTask")
  }
```

Executor

```
/**
   * Invoked when a task running within this executor has been killed
   * (via SchedulerDriver.killTask). Note that no status
   * update will be sent on behalf of the executor, the executor is
   * responsible for creating a new TaskStatus (i.e., with
   * TASK_KILLED) and invoking ExecutorDriver.sendStatusUpdate.
   */
  def killTask(driver: ExecutorDriver, taskId: TaskID): Unit = {
    log.info("Executor.killTask")
  }
```

Executor

```
/**
   * Invoked when a framework message has arrived for this
   * executor. These messages are best effort; do not expect a
   * framework message to be retransmitted in any reliable fashion.
   */
  def frameworkMessage(driver: ExecutorDriver, data: Array[Byte]): Unit = {
    log.info("Executor.frameworkMessage")
  }
```

Executor

```
/**
   * Invoked when the executor should terminate all of it's currently
   * running tasks. Note that after a Mesos has determined that an
   * executor has terminated any tasks that the executor did not send
   * terminal status updates for (e.g., TASK_KILLED, TASK_FINISHED,
   * TASK_FAILED, etc) a TASK_LOST status update will be created.
   */
  def shutdown(driver: ExecutorDriver): Unit = {
    log.info("Executor.shutdown")
  }
```

Executor

```
/**
   * Invoked when a fatal error has occured with the executor and/or
   * executor driver. The driver will be aborted BEFORE invoking this
   * callback.
   */
  def error(driver: ExecutorDriver, message: String): Unit = {
    log.info("Executor.error")
  }
```

Questions?

Big Data Open Source Security LLC

http://www.stealth.ly

Twitter: @allthingshadoop

**********/