[YARN-6592] Add Rich Placement Constraints in YARN

Konstantinos Karanasos, Wangda Tan, Arun Suresh,
with input from: Vinod Vavilapalli, Panagiotis Garefalakis, Carlo Curino, Chris Douglas, Subru Krishnan, Jian He, Varun Vasudev

Last modified date: May 12, 2017

Overview

So far in YARN, applications can only specify placement constraints in the form of data locality (preference to specific nodes or racks) or (non-overlapping) node labels.

In this document, we focus on adding more expressive placement constraints in YARN, which can be crucial for the performance and resilience of specific applications. For example, it may be beneficial to co-locate the allocations[1] of a job on the same rack (**affinity**) to reduce network costs, spread allocations across machines (**anti-affinity**) to minimize resource interference, or allow up to a specific number of allocations in a node group (**cardinality**) to strike a balance between the two. Placement decisions also affect resilience. For example, allocations placed within the same cluster upgrade domain would go offline simultaneously.

Our requirements are the following:

- [R1] Support expressive constraints, such as affinity, anti-affinity and cardinality, within and across jobs, without requiring the application to know the underlying topology of the cluster or the other applications deployed.
- [R2] Be agnostic to different scheduler implementations (such as Capacity and Fair Scheduler).
- [R3] Allow the cluster operator to optimize for various objectives objectives (e.g., balance load across nodes or minimize resource fragmentation) and put limits to constraints.
- [R4] Take multiple allocation requests and constraints into account when doing placement.

Next, we first introduce some changes to the AllocateRequest that will allow us to express constraints between allocations, and then describe how placement constraints can be specified.

Scheduling Requests

Currently, an AllocateRequest object includes a list of ResourceRequests. Each ResourceRequest has information both about the *sizing* of the requested allocations (number and size of allocations, priority, execution type, etc.), and the *constraints* dictating how these allocations should be placed (resource name, relaxed locality).

To allow a clear distinction between sizing and constraints, we add a new **SchedulingRequest** object inside the AllocateRequest. Note that, for compatibility, the users will be able to continue using the ResourceRequest objects.[2] The AllocateRequest will be the following:

- AllocateRequest
  - List <ResourceRequest>
  - List <**SchedulingRequest**> **(New)**

The SchedulingRequest will include (1) meta information (priority, allocate request ID, and allocation tags, which we describe below), (2) sizing information, and (3) placement constraints:

[1] We use the notion of “allocation” to refer to a unit of resources (e.g., CPU and memory) that gets allocated in a node.This can be a single container or multiple containers (in case an application uses the allocation to spawn multiple containers).

[2] Using only ResourceRequests in the AllocateRequest will result in the existing scheduling behavior. Within a single AllocateRequest, an application should use either the ResourceRequests or the SchedulingRequests, but not both of them.

- SchedulingRequest
    - AllocationRequestID
    - Priority
    - **AllocationTags**: tags to be associated with all allocations returned by this SchedulingRequest
    - ResourceSizing
        - Number of allocations
        - Size of each allocation
    - **PlacementConstraintExpression**

We first describe the allocation tags, and will talk about the placement constraints later.

Allocation tags

Each allocation returned by the above SchedulingRequest, can be associated with a set of string **allocation tags**. For example, an allocation belonging to an HBase job, can have an `hbase` tag to show the type of application it belongs to, a `latency-critical` tag to refer to the more general demands of the allocation, or `app_0041` to denote the job ID.

Tags can be used to identify *components* of applications. For example, an HBase Master allocation can be tagged with `hbase-m`, and Region Servers with `hbase-rs`.

As we show later, allocation tags provide the mechanism for constraints to refer to multiple allocations, belonging to the same or different applications.

Differences between node labels, node attributes and allocation tags

The difference between allocation tags and existing node labels or node attributes (YARN-3409), is that allocation tags are attached to allocations and not to nodes. When an allocation gets allocated to a node by the scheduler, the set of tags of that allocation are automatically added to the node for the duration of the allocation. Hence, a node inherits the tags of the allocations that are currently allocated to the node. Likewise, a rack inherits the tags of its nodes.

Moreover, similar to node labels and unlike node attributes, allocation tags have no value attached to them. As we will show, our constraints can refer to node labels, node attributes, as well as allocation tags.

Placement Constraints

Placement constraints are expressions that allow us to express affinity, anti-affinity and cardinality restrictions between allocations.

Hierarchical Placement Constraints

Placement constraints can be specified at different levels, depending on their scope of application:

1) SchedulingRequest-level: these constraints can refer only to allocations requested by the specific SchedulingRequest.
2) Application-level: these constraints are specified in the RegisterApplicationMasterRequest when the AM gets registered, and can refer to any allocation of this or other applications.
3) Cluster admin-level: these constraints are specified through a new API that the cluster admin can use. They can be applied to any of the cluster's applications.

We describe later how we determine which constraints are applied when placing allocations and how constraints of different levels can interact with each other.

Placement Constraint Expression definition

A placement constraint expression can combine multiple simple constraints via different operators:

- **PlacementConstraintExpression**:
  - **ConstraintExpressionOperator**: {AND, OR, DELAYED_OR}
    - List<PlacementConstraintExpression>
  - **SimplePlacementConstraint**

- **ConstraintExpressionOperator**
  1) **AND:** All constraints must be satisfied.
     Example: Place me on a machine "on this rack AND do not more than 2 on each machine".
  2) **OR**: At least one of the constraints must be satisfied. There is no preference between constraints.
     Example: Place me on "host n1 or host n2"
  3) **DELAYED_OR:** Similar to OR, but supports ordering and delays for constraints.
     Example: Try for 2 mins to place me to host n1; then try for 3 mins to place me on rack r1; if everything else fails, place me on any node.
     The following *delay criteria* can be specified to control the following:
     - Number of *units of delay* for each constraint in the expression.
     - Units of delay can be either missed opportunities or wall clock time.
     - Ability to reset units of delay after one of the allocations gets allocated (similar to today's resetRackDelay).

- **SimplePlacementConstraint**:
  - *List<TargetExpression>*:

        - Each TargetExpression is of the form <target-type>:<target-key> <op> <target-values>:
            - Target-type: *{node-attribute, allocation-tag}.*
            - Target-key: in case of a node-attribute, this is the key of the node attribute; in case of an allocation-tag, this is left empty.
            - Op: *{IN, NOT_IN}*.
            - Target-values is a set that includes multiple comma-separated values. In case of node-attribute constraints, this dictates the possible values of the node attribute; in case of allocation tags, it dictates the name of the allocation tag.
              Example of node-attribute constraint: node-attribute:host IN {n1,n2}.
              Example of allocation-tag constraint: allocation-tag NOT_IN spark.
        - When multiple targets are specified, all of them should be satisfied:
          Example: <node-attribute> IN "node-partition=x", <node-attribute> IN "host=n1,n2,n3", <node-attribute> NOT_IN "os=windows".
    - *Min/max-cardinality*: int values.
    - *Scope*: Defined by a node attribute key (e.g., host, rack). Please refer to the next "Node Attribute Keys for Scope of Constraints" for more details.

At the Application-level and the SchedulingRequest-level constraints, to keep the semantics of the constraints simple, we allow constraints of the following two forms:

- **TargetConstraint** { List<TargetExpression>, scope }: This constraint allows to place containers in a scope (e.g., host or rack) that satisfies the target expressions.
- **CardinalityConstraint** { min/max-cardinality, scope }: This constraint restricts the number of containers we can place in this scope to be within the [min-cardinality, max-cardinality] space.

At the Cluster admin-level, all fields are allowed to be used. Consider a set of nodes *N* that belongs to the scope specified in the constraint. If the *TargetExpression* is satisfied at least *min-cardinality* times and at most *max-cardinality* times in node set *N*, then the constraint is satisfied. See below for examples.

Node attribute keys for Scope of Constraints

Each cluster node can have multiple node attributes of the form (key=value). By default, we attach at every node the built-in attributes with keys `host` and `rack`. Their values correspond to the hostname and the name of the rack, respectively.

These node attribute keys can be used in the scope of a Constraint, as shown above.
Note that additional node attributes can be used in the scope. Such examples are fault domains (i.e., hosts that are more likely to fail together) and upgrade domains (i.e., hosts whose bits get

upgraded simultaneously). For example, here are some attributes that can be used in the scope of constraints for two cluster nodes:

1) node1: “host=n1”, “rack=r1”, “fault_domain=fd1”
2) node2: “host=n2”, “rack=r1”, “fault_domain=fd2”

Examples

Here we give various examples of constraints using our *PlacementConstraintExpression* API.

C1. Affinity between allocations: Place allocations in the same node with an HBase Master:

```
{target: allocation-tag IN “hbase-m”, scope: host}
```

C2. Affinity between allocations and max cardinality of containers placed: Place allocations in the same rack as a ZooKeeper allocation; don’t allow more than 3 containers per node.

```
AND: [
{target: allocation-tag IN “zooKeeper”, scope: rack},
{max-cardinality: 3, scope: host}
]
```

C3. Affinity to node attributes: Place allocations in GPU machines.

```
{target: node-attribute:node-partition IN “GPU”, scope: host}
```

C4. Anti-affinity to allocation tag: Do not place allocation in same fault domain with Spark allocations.

```
{target: allocation-tag NOT_IN “spark”, scope: fault_domain}
```

C5. Affinity and delayed scheduling: Consider three nodes n1, n2, n3. Node n1 and n2 belong to rack r1; node n3 belongs to rack r2. There are three scheduling alternatives: (1) put at most 2 allocations at each of the 3 nodes, (2) put at most 4 allocations at rack r1 and at most 2 at r2, (3) put allocations in any node in the cluster. This is how a typical MapReduce application would ask for resources.

```
DELAYED_OR: [
   AND: [
      {target: node-attribute:host IN “n1,n2,n3”, scope: host},
      {max-cardinality: 2, scope: host}
   ],
   OR: [
      AND: [
         {target: node-attribute:rack IN “r1”, scope: host},
         {max-cardinality: 4, scope: rack}
      ],
      AND: [
         {target: node-attribute:rack IN “r2”, scope: host},
         {max-cardinality: 2, scope: rack}
      ]
   ],
   {target: node-attribute:host IN ANY, scope: host}
]
```

Applying constraints during scheduling

As we mentioned above, constraints can be specified in three different levels: at the SchedulingRequest, the application level, and the cluster operator’s level.

Constraints defined at the SchedulingRequest level have to be satisfied by all allocations requested by this SchedulingRequest. However, for the other two more general levels, we need to specify the SchedulingRequests on which the constraints will be applied. To do so, we use a “constraint map” of the form (sourceTags: List<allocation-tag>, constraintList: List<PlacementConstraintExpression>). The source-tag can be any set of allocation tags and all SchedulingRequests with those allocation tags have to satisfy the constraint.

As an example, consider an application with ID appID_0045 that wants to request each of its storm allocations to be collocated at the same node with at least one HBase Region Server of an application with ID appID_0023. Then the constraint map will be the following:

```
{
   sourceTags: { storm, appID_0045 },
   constraintList: { { {target: allocation-tag IN “hbase-rs”}, {target:
allocation-tag IN “appID_0023”} }, scope: host }
}
```

Similarly, a cluster operator can request any storm allocation to be collocated at the same rack with at least 3 HBase Region Servers:

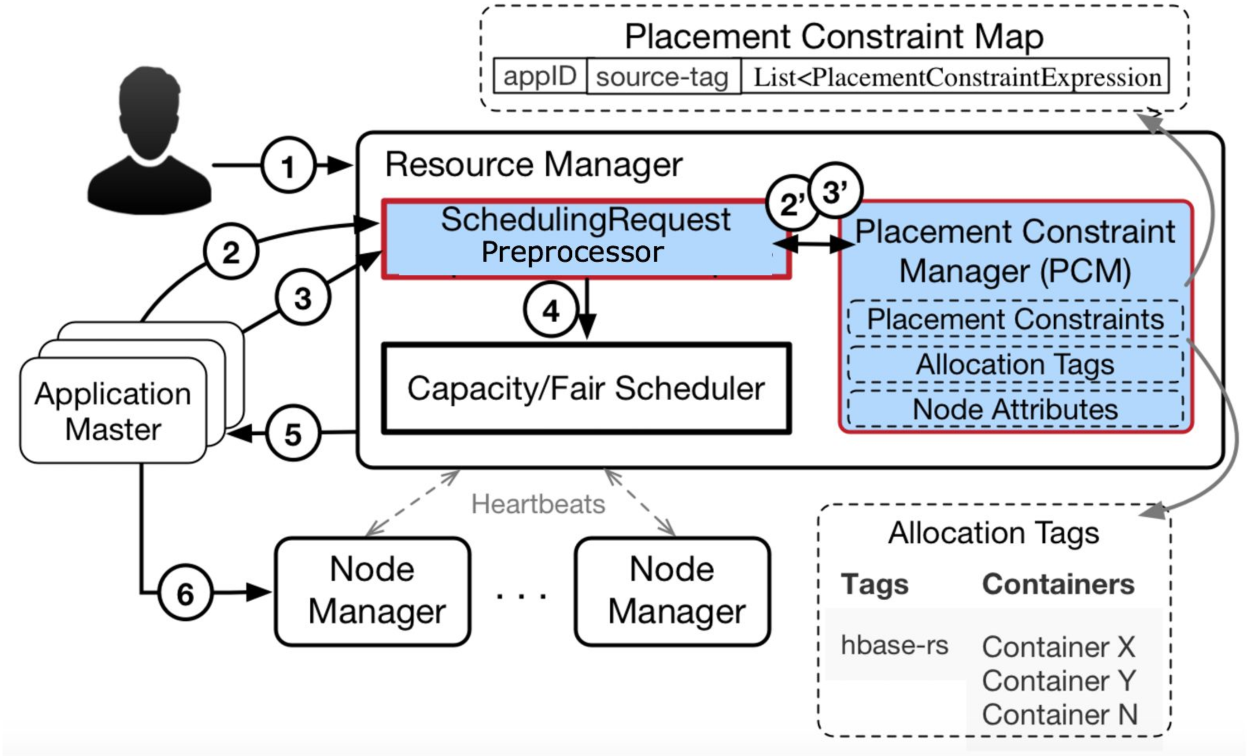
```
{  sourceTags: { storm }, constraintList: { target: allocation-tag IN
“hbase-rs” , scope: “rack”, min-cardinality: 3 } }
```

A constraint at a less general level[3] can override a constraint at a more general level, as long as it is more restrictive than it.

[3] We consider admin-level to be the most “general” level of constraints, followed by application-level and then by SchedulingRequest-level.

For example, if the cluster admin requires at most 5 ZooKeeper allocations per rack, a specific ZooKeeper application can request to have only 3 allocations in a rack, but cannot request to have more than 5.

Implementation design



In order to support the new placement constraints, we add two new components to the Resource Manager: the Placement Constraint Manager (PCM) and the SchedulingRequest Preprocessor.

The PCM stores the allocation tags, the placement constraints (both the application specific and the cluster admin ones), and the node attributes.

All requests from an application to the RM get preprocessed by the SchedulingRequest Preprocessor[4]. The implementation details of the preprocessor will be discussed in the corresponding JIRA. Here we give only a high-level design, which is depicted in the figure above too.

The life-cycle of a job can be summarized in the following steps (see also the figure):

[4] The SchedulingRequestPreprocessor would be an implementation of a preprocessor as defined in YARN-6355

Step 1. The application gets submitted to the cluster by the YARN Client.

Step 2. At submission time (either through the RegisterApplicationMasterRequest or ApplicationSubmissionContext -- TBD), the job can specify placement constraints that will be applied for allocations with the given allocation tags. If there are indeed constraints, the preprocessor forwards them to the PCM that stores them (step 2').

Step 3. The ApplicationMaster submits SchedulingRequests, which might include allocation tags, possibly with additional placement constraints. Then the Placement Constraint Manager gets queried to check if there are placement constraints specified for these allocation tags (step 3').

Step 4. The allocation requests are forwarded to the main scheduler, who performs the actual allocation.

Note for further discussion. We discussed two alternative designs for performing the placement of SchedulingRequests:
(1) Decide the placement in the SchedulingRequestPreprocessor and forward the placement decision to the Capacity/Fair Scheduler that will do the allocation.
(2) Add support to the Capacity/Fair Scheduler for dealing with constraints, in which case we can directly forward the SchedulingRequests with constraints to the Scheduler.

Step 5. The RM returns the allocations to the AM, who matches them based on the AllocationRequestID and the allocation tags.

Step 6. Once informed of the allocations, the AM dispatches the tasks to the specific nodes to start their execution.

Future work

Here we provide some ideas for future work:

Reservations with Placement Constraints

It would be useful to extend the reservation system to support placement constraints. In any busy cluster or in a cluster where applications with very restrictive placement constraints are deployed, it might take a long time to satisfy the constraints of a new application. In this case, after a few unsuccessful attempts, we can fall back to a reservation with the same constraints.

Scheduler with generic constraint manager

At the moment we are only using the Constraint Manager to perform placement of SchedulingRequests to which placement constraints apply. Eventually, we can make all placements be made by the Constraint Manager.

We can make use of the scheduler resource proposal/commit API which is added by YARN-5139: Placement decisions can be made outside of main scheduler. So we don't have to duplicate logics between schedulers (like Fair/Capacity). More details please refer to: Notes of resource proposal/commit API.

Appendix

We now provide different use cases that use the new placement constraints API

HBase application

Consider an HBase application with one HBase Master, 10 HBase RegionServers, and the following constraints:

1. Do not place more than 1 HBase RegionServer per host, and more than 2 per failure domain.
2. Do not put any component on spot instances.
3. Do not place any RegionServer at the same node with the HBase Master.

Inside the AllocateRequest, the AM will include the following two SchedulingRequests:

```
AllocateRequest {
  List<SchedulingRequest>: [
    {
       // SchedulingRequest for HBase master (see below)
    },
    {
       // SchedulingRequest for HBase RegionServers (see below)
    }
  ]
}
```

SchedulingRequest for HBase master:

```
{
  Priority: 1,
  ResourceSizing: {
    Resource: <8G, 4vcores>,
    NumAllocations: 1
  },
  AllocationTags: ["hbase-m"]
}
```

SchedulingRequest for HBase RegionServers:

```
{
  Priority: 1,
  Sizing: {
    Resource: <8G, 4vcores>,
    NumAllocations: 1
  },
  AllocationTags: ["hbase-rs"],
  PlacementConstraintExpression: {
    AND: [
      // Anti-affinity between RegionServers
      {
        Target: allocation-tag NOT_IN “hbase-rs”,
        Scope: host
      },
      // Allow at most 2 RegionServers per failure-domain
      {
        MaxCardinality: 2,
        Scope: failure_domain
      }
    ]
  }
},
```

Moreover the ApplicationMasterRegisterRequest will contain the following constraint map:

```
{
  {
    sourceTags: ANY,
    constraintList:
    {
      Target: node-attribute:node_type NOT_IN spot_instance,
      Scope: host
    }
  },
  {
    sourceTags: hbase-rs,
    constraintList:
    {
      Target: allocation-tag NOT_IN “hbase-m”,
      Scope: host
    }
  }
}
```

Spark with HBase example

Run a Spark job with 8 allocations, each having rack affinity to an HBase RegionServer:

```
List<SchedulingRequest>: [
  {
    AllocationId: 123,
    Priority: 1,
    ResourceSizing: {
      Resource: <24G, 8vcores>,
      NumAllocations: 8
    },
    PlacementConstraintExpression: {
      Target: allocation-tag NOT_IN "hbase-rs"
      Scope: rack
    }
  }
]
```