MEDEA: Scheduling of Long Running Applications

在生产集群中会同时运行机器学习、流计算、交互式作业及批处理等多种类型作业，要求资源调度器能够实现多样化的放置约束来控制这些作业的执行。MEDEA是用于实现常驻(LRA，Long Running Applications)及短任务的同时调度的新型资源调度器，对于常驻作业采用基于最优化目标的方法进行调度，短作业还是使用传统的Task-based低延迟的调度算法。

对于常驻作业，这些Container启动后会长时间运行，可以容忍比批处理作业调度长的调度等待时间，因此其调度算法和短作业是基于不同的优化目标，其核心：

1. 实现容器放置的精确控制，以优化应用的性能和弹性，例如相同服务的容器放置在同一网络rack上（亲和性）以降低网络成本或者分散开（反亲和性）以降低资源干扰和关联故障。为了实现更优的服务性能，还要支持更强的约束，比如复杂的应用内或应用间的放置约束、限制每个节点或者机架上容器的个数等
2. LRA容器的放置，集群管理员会有自定义的全局优化目标

* 违反放置约束
* 资源碎片化
* 负载不均衡
* 使用机器数目

实现以上指标的最小化。

1. 常驻作业的调度不应影响短作业的调度延迟

# 概念和系统架构

为了实现常驻服务和短作业的调度需求，MEDEA调度器的主要功能点：

1. Two-scheduler design

为LRAs实现了专用的调度模块，短作业还是使用传统的调度模块。将调度分为两个部分使短作业的调度不受影响，保证其调度延迟，同时要支持LRAs的调度需求。

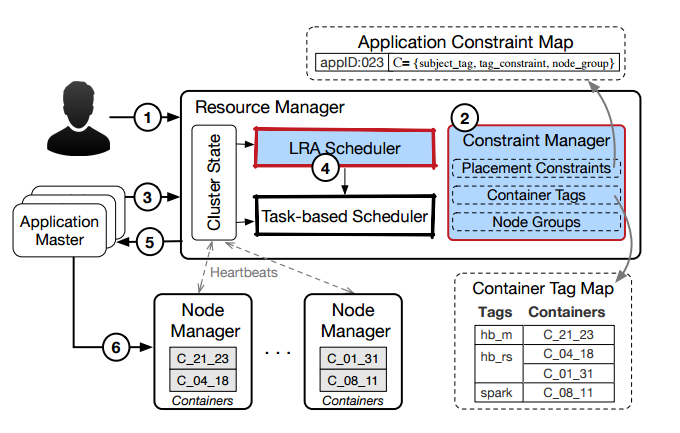
1. Expressive,hign-level constraints

在LRAs调度器中增加丰富的放置约束语法，引入了container tags和node groups的概念，集群管理者可以基于约束语法实现应用内及应用之间的放置约束

1. LRA调度算法

将LRAs调度抽象为ILP(线性规划问题)算法，在调度执行时同时考虑多LRA的资源请求，实现全局目标的最优化

基于以上功能点，MEDEA调度器的系统架构图如下图所示：



1. LRA接口

用户向调度器的提交资源时，其请求的定义包括：

* Containers数目及资源，这部分和传统调度器是相同的，有Task-based Scheduler处理，比如：

*10个container，每个container使用的资源为2CPU和4GB内存*

*Task具有数据本地性（MR Task分配到数据所在节点）*

* 放置约束，这部分请求由LRA调度器来处理，其描述为：

*target: allocation-tag IN “hbase-m”, scope:host*

1. LRA调度器

使用在线优化算法，其基于:

*集群的状态，包括正在运行的LRAs及Task-based Jobs*

*时间间隔内提交的LRAs请求，资源需求及放置约束*

*全局优化目标*

进行计算，生成最优调度

1. Task-based调度器，复用已有的资源调度器，仅考虑container数目及资源
2. Constraint Manager，约束管理器，用于管理用户和集群管理者定义的约束

在MEDEA调度器中，先将资源请求通过LRAs调度器处理约束放置的条件，然后将处理结果交给Task-based调度器进行实际的资源申请，这避免了约束放置和资源需求之间的造成的矛盾。

# Placement Constraints的定义

在LRAs调度器中引入了tags和node groups的概念，放置约束是基于这两个概念来定义的

## Tag Model&Node Group

**1）Container Tag，**在Medea调度器中可以为所有的Container添加标签，HBase Master Container添加的标签示例如下：

*appId: 0023 //LRA的ID*

*hb //应用的类型，HBase*

*hb\_m //HBase的Master组件*

*memory\_critical //资源定义*

1. **Tag sets**，node Tag为节点中所有运行的Container的tag集合，在节点中部署了两个HBase Container: HMaster Container使用的标签为{hb,hb\_m}，RegionServer标签为{hb,hb\_rs}，则该节点的Tag为：

*Tn1 = {hb,hb\_mb,hb\_rs}*

*rn1(hb)=2 //hb标签出现的次数为2*

*rn1(hb\_m)=rn1(hb\_rs)=1*

1. **Node Group**

将节点进行逻辑分成节点集合，最典型的分组概念是node和rack

* Rack包括了该物理机架上的所有节点
* Upgrade domains，节点的升级Unit

在放置约束中可以定义Node Group的使用，示例：

*place hb containers of the same application in different upgrade domains*

在放置约束中可以通过以上概念的组合来实现复杂的放置。

## **Placement Constraints**

应用开发者及集群管理者通过上面介绍的tag来定义放置约束，控制container启动在相同或者不同的LRAs或者通过node groups启动到特定节点集合中，格式如下：

*C = {subject\_tag, tag\_constraint, node\_group}*

* subject\_tag，标识分配的container标识
* tag\_constraint，格式如下{c\_tag,c\_min,c\_max}，c\_tag为Container标识，c\_min和c\_max为整形值
* node\_group，标识node\_group

*示例1：C\_af = {storm, {hb ∧ mem, 1, ∞}, node}*

将storm container放到运行启动包含hb和memb tag的Container的节点上，如果限制特定的app（ID 0023），则约束条件为：

*示例2：C\_af= {appID:0023 ∧ storm, {appID:0023 ∧ hb ∧ mem, 1, ∞}, node}*

* c\_min=1和c\_max=∞表示亲和性
* c\_min=0和c\_max=0表示反亲和性

*示例3：Caa = {storm, {hb, 0, 0}, upgrade\_domain}*

表示分配Storm Container到与hb containers不同的upgrade\_domain上

* c\_min和c\_max表示cardinality

*示例4：Cca = {storm, {spark, 0, 5}, rack}*

分配storm到最多启动5个spark container的rack上。

如果subject\_tag和tag\_constraint使用相同的tag，则表示启动containers group

*示例5：Cca = {spark, {spark, 3, 10}, rack}*

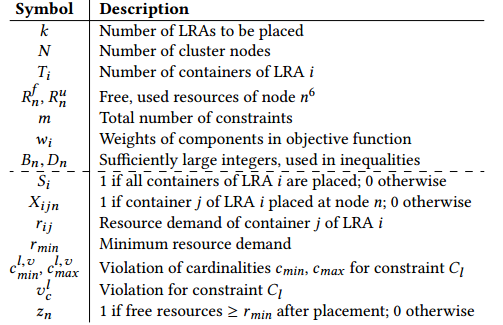
集群管理者在rack上启动的spark container在范围[3,10]

# LRA调度算法

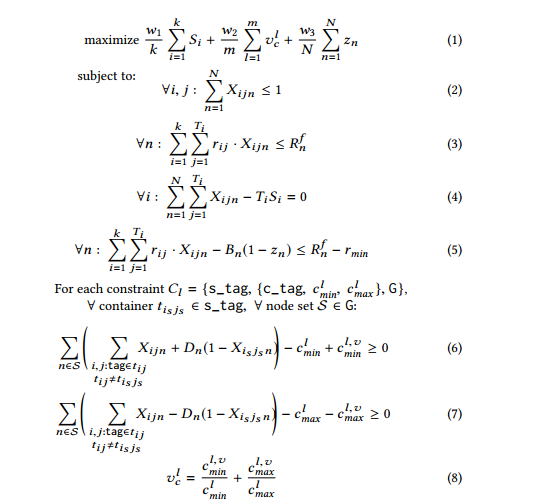
LRA的调度执行处理以下信息：

* Container Request及放置约束
* 已部署到LRAs的约束，及constraint manager中集群操作约束
* 集群中可用资源

执行的算法包ILP-based及启发式算法，下面是处理的参数：



线性规模的模型如下：



根据以上模型生成Container放置决策。