Spark数据本地性

数据的计算尽可能在数据所在节点进行，这样可以减少数据在网络上的传输，毕竟移动计算比移动数据的代价小。进一步的看，数据如果在运行节点的内存中，就能够进一步减少磁盘I/O的传输，在Spark中数据本地性优先级从高到低:

*PROCESS\_LOCAL > NODE\_LOCAL>NO\_PREF>RACK\_LOCAL>ANY*

最好是任务运行的节点内存中存在数据、次好是同一个Node上，最次是同机架，最后是任意位置。其中数据本地性通过以下情况来确定：

* 如果任务处于作业调度阶段内，这些任务对应的RDD分区都有首选运行位置，该位置也是任务运行的首先位置，数据本地性为NODE\_LOCAL
* 如果任务处于非作业开头的调度阶段，可以根据父调度运行阶段的位置得到任务首选位置，这种情况下，如果Executor位于活动状态，则数据本地性为PROCESS\_LOCAL;如果Executor不处于活动状态，但存在父调度阶段运行结果，则数据本地性为NODE\_LOCAL
* 如果没有首选位置，则数据本地性为NO\_PREF

任务的首先位置的初始化时使用TaskSetManager#addPendingTask方法，根据任务首先位置得到pendingTasksForExecutor,pendingTasksForHost,pendingTasksWithNoPrefs和

pendingTaskForRack四个列表；然后根据这4个列表在computeValidLocalityLevels方法中得到该任务集的数据本地性列表，按照获取的数据本地性从高到低匹配可用的Worker节点，在匹配前使用getAllowedLocalityLevel方法得到数据集允许的数据本地性，比较数据本地性和指定数据本地性优先级，取优先级高的数据本地性，。代码如下：

*private[spark] def addPendingTask(index: Int) {*

*for (loc <- tasks(index).preferredLocations) {*

*loc match {*

*case e: ExecutorCacheTaskLocation =>*

*pendingTasksForExecutor.getOrElseUpdate(e.executorId, new ArrayBuffer) += index*

*case e: HDFSCacheTaskLocation =>*

*val exe = sched.getExecutorsAliveOnHost(loc.host)*

*exe match {*

*case Some(set) =>*

*for (e <- set) {*

*pendingTasksForExecutor.getOrElseUpdate(e, new ArrayBuffer) += index*

*}*

*logInfo(s"Pending task $index has a cached location at ${e.host} " +*

*", where there are executors " + set.mkString(","))*

*case None => logDebug(s"Pending task $index has a cached location at ${e.host} " +*

*", but there are no executors alive there.")*

*}*

*case \_ =>*

*}*

*pendingTasksForHost.getOrElseUpdate(loc.host, new ArrayBuffer) += index*

*for (rack <- sched.getRackForHost(loc.host)) {*

*pendingTasksForRack.getOrElseUpdate(rack, new ArrayBuffer) += index*

*}*

*}*

*if (tasks(index).preferredLocations == Nil) {*

*pendingTasksWithNoPrefs += index*

*}*

*allPendingTasks += index // No point scanning this whole list to find the old task there*

*}*

在任务分配运行节点时，先判断任务最佳运行节点是否空闲，如果该节点没有足够的资源运行该任务，在这种情况下会等待一段时间；如果在等待时间内该节点释放出足够的资源，则任务在该节点运行，如果还是不足会找出次佳的节点进行运行。通过这样的方式进行能够让任务运行在更高级别数据本地性节点，从而减少磁盘I/O和网络传输。一般来说只对PROCESS\_LOCAL和NODE\_LOCAL两个数据本地性级别进行等待，系统默认延迟时间为3s。

任务的分配过程在TaskSchedulerImpl#resourceOffers方法实现，在该方法中先对应用程序获取的资源（如Worker节点）进行混洗，以使任务能够更加均衡的分散在集群中运行，然后对任务集对应的TaskSetManager根据设置的调度算进行排序，最后对TaskSetManager中的任务按照数据本地性分配运行运行节点，在分配时根据任务集的本地性从优先级高到低进行分配任务，在分配过程中动态判断集群中节点运行的情况，通过延迟执行等待数据本地性更高的节点运行。其代码如下：

*def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] = synchronized {*

*//为了负载均衡，打乱offers顺序，Random.shuffle用于将集合中的元素打乱*

*val shuffledOffers = shuffleOffers(filteredOffers)*

*//用于创建存储每个worker对应的运行任务列表的map*

*val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores / CPUS\_PER\_TASK))*

*val availableCpus = shuffledOffers.map(o => o.cores).toArray*

*//按照调度策略排序好的TaskSetManager*

*val sortedTaskSets = rootPool.getSortedTaskSetQueue*

*//如果是新加入的Exectuor，则重新计算数据本地性*

*for (taskSet <- sortedTaskSets) {*

*logDebug("parentName: %s, name: %s, runningTasks: %s".format(*

*taskSet.parent.name, taskSet.name, taskSet.runningTasks))*

*if (newExecAvail) {*

*taskSet.executorAdded()*

*}*

*}*

*//为排好序的TaskSetManager列表进行分配资源，分配的原则为就近原则*

*// PROCESS\_LOCAL > NODE\_LOCAL>NO\_PREF>RACK\_LOCAL>ANY*

*for (taskSet <- sortedTaskSets) {*

*var launchedAnyTask = false*

*var launchedTaskAtCurrentMaxLocality = false*

*for (currentMaxLocality <- taskSet.myLocalityLevels) {*

*do {*

*launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(*

*taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)*

*launchedAnyTask |= launchedTaskAtCurrentMaxLocality*

*} while (launchedTaskAtCurrentMaxLocality)*

*}*

*if (!launchedAnyTask) {*

*taskSet.abortIfCompletelyBlacklisted(hostToExecutors)*

*}*

*}*

*if (tasks.size > 0) {*

*hasLaunchedTask = true*

*}*

*return tasks*

*}*

对单个任务集的任务调度由TaskSetManager.resourceOfferSingleTaskSet方法实现，在该方法中会遍历所有Worker，会判断Worker中的CPU核数是否满足任务的核数，如果满足则调用resourceOffer方法对该Worker的Executor分配任务，分配完毕后更新任务对应的任务集管理器列表，任务对应Executor列表和Executor对应机器列表，并减去该任务使用的CPU核数等，其代码如下：

*private def resourceOfferSingleTaskSet(*

*taskSet: TaskSetManager,*

*maxLocality: TaskLocality,*

*shuffledOffers: Seq[WorkerOffer],*

*availableCpus: Array[Int],*

*tasks: IndexedSeq[ArrayBuffer[TaskDescription]]) : Boolean = {*

*var launchedTask = false*

*//遍历所有Worker，为每个Worker分配运行的任务*

*for (i <- 0 until shuffledOffers.size) {*

*val execId = shuffledOffers(i).executorId*

*val host = shuffledOffers(i).host*

*//当Worker的CPU核数满足任务运行核数*

*if (availableCpus(i) >= CPUS\_PER\_TASK) {*

*try {*

*//对指定的Executor分配运行的任务，分配后更新相关列表和递减可用CPU*

*for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {*

*tasks(i) += task*

*val tid = task.taskId*

*taskIdToTaskSetManager(tid) = taskSet*

*taskIdToExecutorId(tid) = execId*

*executorIdToRunningTaskIds(execId).add(tid)*

*availableCpus(i) -= CPUS\_PER\_TASK*

*assert(availableCpus(i) >= 0)*

*launchedTask = true*

*}*

*} catch {*

*case e: TaskNotSerializableException =>*

*logError(s"Resource offer failed, task set ${taskSet.name} was not serializable")*

*// Do not offer resources for this task, but don't throw an error to allow other*

*// task sets to be submitted.*

*return launchedTask*

*}*

*}*

*}*

*return launchedTask*

*}*