

第 1 课 : Spark 运行架构

一、 术语定义

Application: Spark Application 的概念和 Hadoop MapReduce 中的类似, 指的是用户编写的 Spark 应用程序, 包含了一个 Driver 功能的代码和分布在集群中多个节点上运行的 Executor 代码;

Driver: Spark 中的 Driver 即运行上述 Application 的 main() 函数并且创建 SparkContext, 其中创建 SparkContext 的目的是为了准备 Spark 应用程序的运行环境。在 Spark 中由 SparkContext 负责和 ClusterManager 通信, 进行资源的申请、任务的分配和监控等; 当 Executor 部分运行完毕后, Driver 负责将 SparkContext 关闭。通常用 SparkContext 代表 Drive;

Executor: Application 运行在 Worker 节点上的一个进程, 该进程负责运行 Task, 并且负责将数据存在内存或者磁盘上, 每个 Application 都有各自独立的一批 Executor。在 Spark on Yarn 模式下, 其进程名称为 CoarseGrainedExecutorBackend, 类似于 Hadoop MapReduce 中的 YarnChild。一个 CoarseGrainedExecutorBackend 进程有且仅有一个 executor 对象, 它负责将 Task 包装成 taskRunner, 并从线程池中抽取出一个空闲线程运行 Task。每个 CoarseGrainedExecutorBackend 能并行运行 Task 的数量就取决于分配给它的 CPU 的 Cores;

Cluster Manager: 指的是在集群上获取资源的外部服务, 目前有:

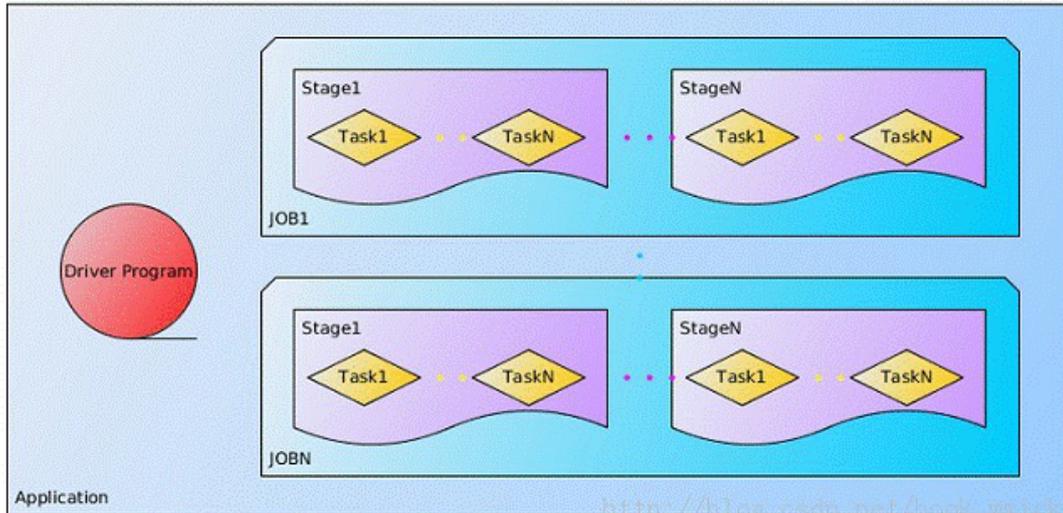
- Standalone: Spark 原生的资源管理, 由 Master 负责资源的分配;
- Hadoop Yarn: 由 YARN 中的 ResourceManager 负责资源的分配;

Worker: 集群中任何可以运行 Application 代码的节点, 类似于 YARN 中的 NodeManager 节点。在 Standalone 模式中指的就是通过 Slave 文件配置的 Worker 节点, 在 Spark on Yarn 模式中指的就是 NodeManager 节点;

Job: 包含多个 Task 组成的并行计算, 往往由 Spark Action 催生, 一个 JOB 包含多个 RDD 及作用于相应 RDD 上的各种 Operation;

Stage: 每个 Job 会被拆分很多组 Task, 每组任务被称为 Stage, 也可称 TaskSet, 一个作业分为多个阶段;

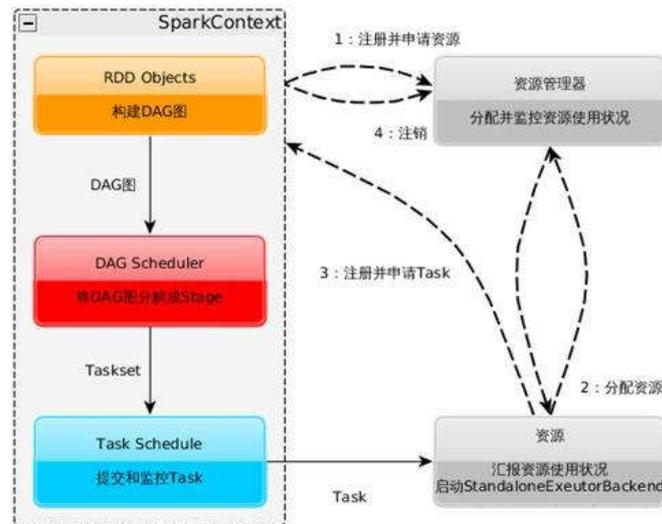
Task: 被送到某个 Executor 上的工作任务;



二、Spark 运行基本流程

Spark 运行基本流程参见下面示意图

1. 构建 Spark Application 的运行环境（启动 SparkContext），SparkContext 向资源管理器（可以是 Standalone、Mesos 或 YARN）注册并申请运行 Executor 资源；
2. 资源管理器分配 Executor 资源并启动 StandaloneExecutorBackend，Executor 运行情况将随着心跳发送到资源管理器上；
3. SparkContext 构建成 DAG 图，将 DAG 图分解成 Stage，并把 Taskset 发送给 Task Scheduler。Executor 向 SparkContext 申请 Task，Task Scheduler 将 Task 发放给 Executor 运行同时 SparkContext 将应用程序代码发放给 Executor。
4. Task 在 Executor 上运行，运行完毕释放所有资源。



Spark 运行架构特点:

1. 每个 Application 获取专属的 executor 进程，该进程在 Application 期间一直驻留，并以多线程方式运行 tasks。这种 Application 隔离机制有其优势的，无论是从调度角度看（每个 Driver 调度它自己的任务），还是从运行角度看（来自不同 Application 的 Task 运行在不同的 JVM 中）。当然，这也意味着 Spark Application 不能跨应用程序共享数据，除非将数据写入到外部存储系统。
2. Spark 与资源管理器无关，只要能够获得 executor 进程，并能保持相互通信就可以了。
3. 提交 SparkContext 的 Client 应该靠近 Worker 节点(运行 Executor 的节点)，最好是在同一个 Rack 里，因为 Spark Application 运行过程中 SparkContext 和 Executor 之间有大量的信息交换；如果想在远程集群中运行，最好使用 RPC 将 SparkContext 提交给集群，不要远离 Worker 运行 SparkContext。
4. Task 采用了数据本地性和推测执行的优化机制。

DAGScheduler

DAGScheduler 把一个 Spark 作业转换成 Stage 的 DAG (Directed Acyclic Graph 有向无环图)，根据 RDD 和 Stage 之间的关系找出开销最小的调度方法，然后把 Stage 以 TaskSet 的形式提交给 TaskScheduler，下图展示了 DAGScheduler 的作用：

Spark program

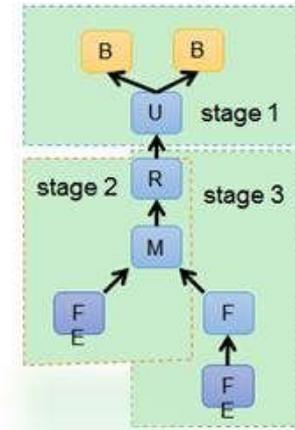
```
val lines1 = sc.textFile(inputPath1)
val lines2 = sc.textFile(inputPath2)

t = t1.union(t2).map(...).reduce(...)

t.saveAsHadoopFiles(...)
t.filter(...).foreach(...)
```



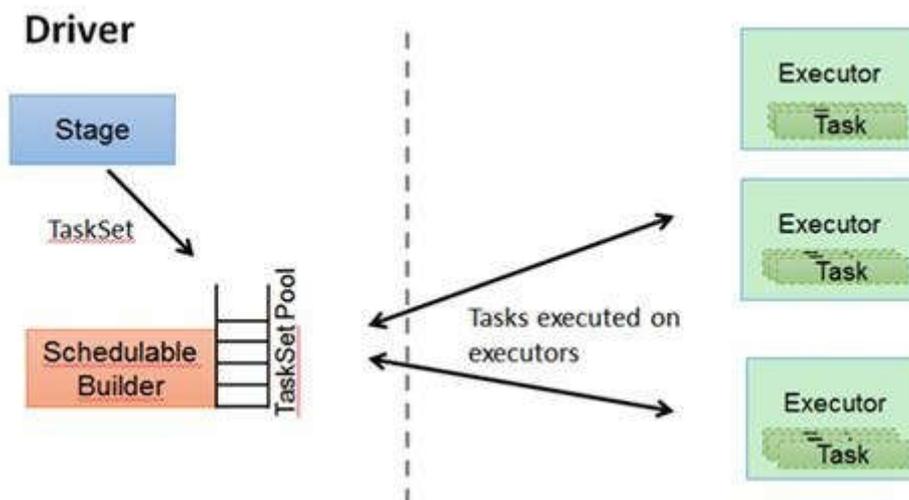
RDD Graph



TaskScheduler

DAGScheduler 决定了运行 Task 的理想位置，并把这些信息传递给下层的 TaskScheduler。此外，DAGScheduler 还处理由于 Shuffle 数据丢失导致的失败，这有可能需要重新提交运行之前的 Stage（非 Shuffle 数据丢失导致的 Task 失败由 TaskScheduler 处理）。

TaskScheduler 维护所有 TaskSet，当 Executor 向 Driver 发送心跳时，TaskScheduler 会根据其资源剩余情况分配相应的 Task。另外 TaskScheduler 还维护着所有 Task 的运行状态，重试失败的 Task。下图展示了 TaskScheduler 的作用：



在不同运行模式中任务调度器具体为：

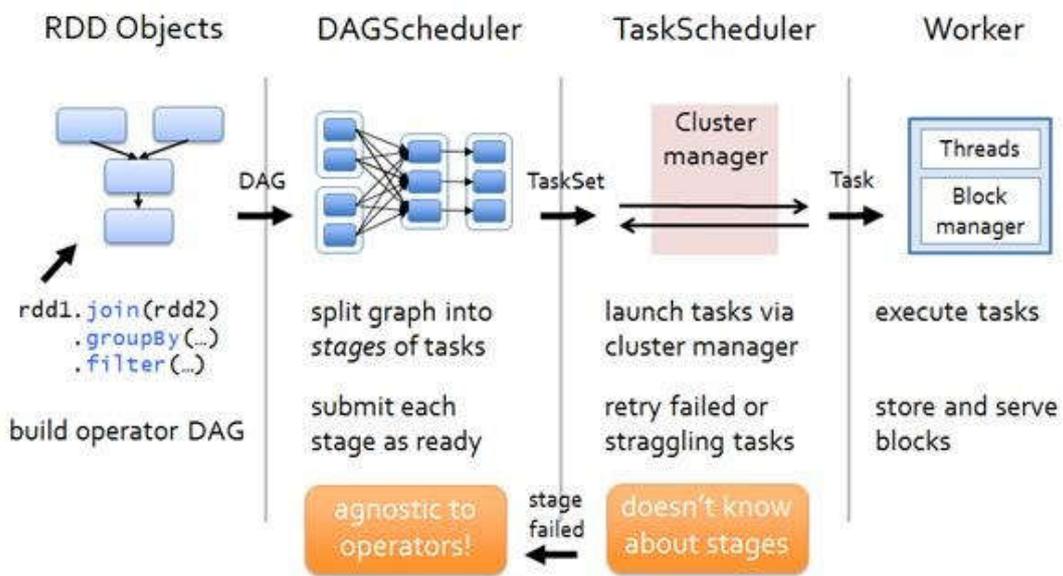
- Spark on Standalone 模式为 TaskScheduler；
- YARN-Client 模式为 YarnClientClusterScheduler

➤ YARN-Cluster 模式为 YarnClusterScheduler

RDD 运行原理

RDD 在 Spark 架构中是如何运行的呢？总高层次来看，主要分为三步：

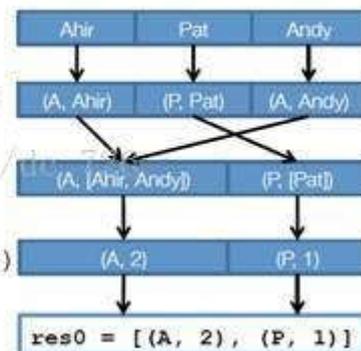
1. 创建 RDD 对象
2. DAGScheduler 模块介入运算，计算 RDD 之间的依赖关系。RDD 之间的依赖关系就形成了 DAG
3. 每一个 JOB 被分为多个 Stage，划分 Stage 的一个主要依据是当前计算因子的输入是否是确定的，如果是则将其分在同一个 Stage，避免多个 Stage 之间的消息传递开销。



以下面一个按 A-Z 首字母分类，查找相同首字母下不同姓名总个数的例子来看一下 RDD 是如何运行起来的。

Goal: Find number of distinct names per "first letter"

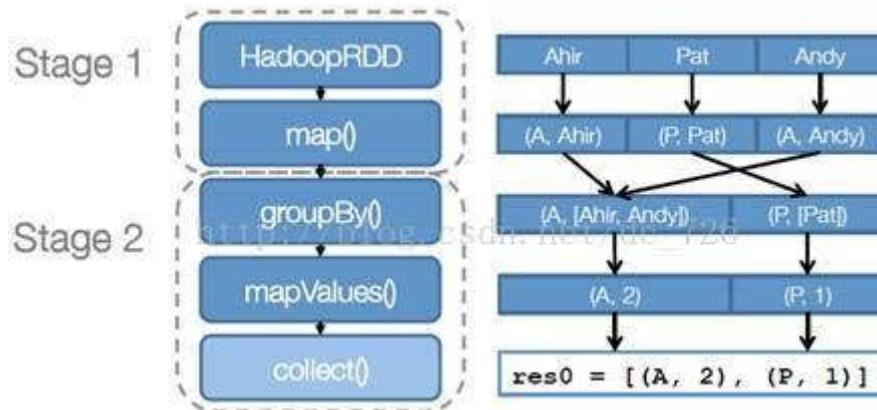
```
sc.textFile("hdfs:/names")
  .map(name => (name.charAt(0), name))
  .groupByKey()
  .mapValues(names => names.toSet.size)
  .collect()
```



步骤 1：创建 RDD 上面的例子除去最后一个 collect 是个动作，不会创建

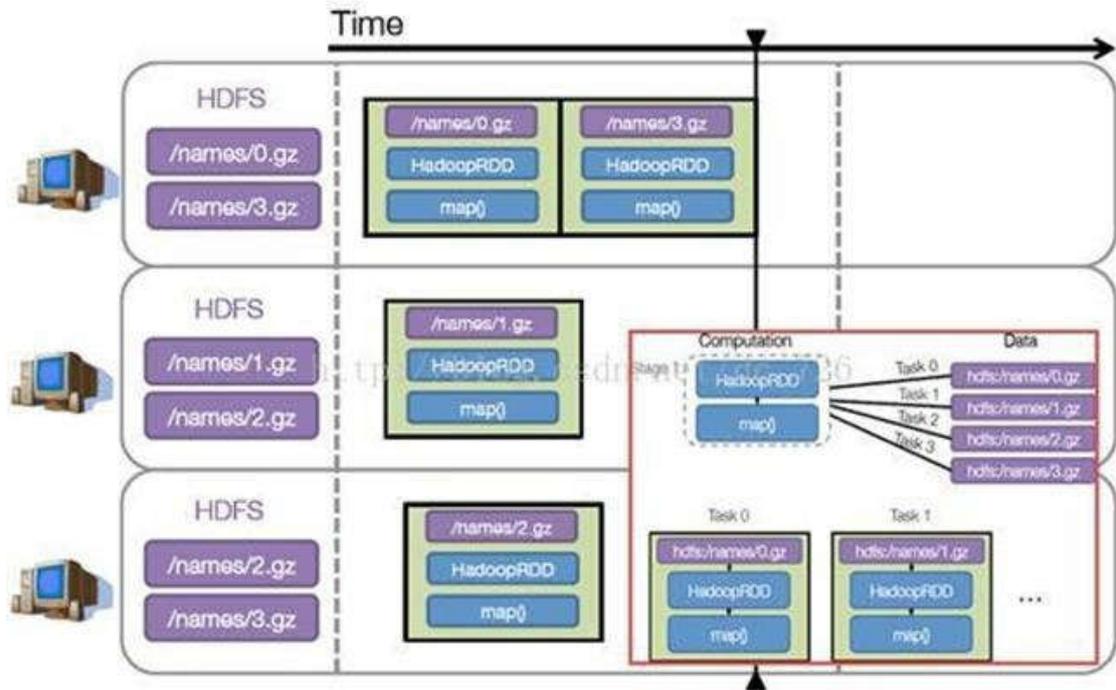
RDD 之外，前面四个转换都会创建出新的 RDD 。因此第一步就是创建好所有 RDD(内部的五项信息) 。

步骤 2：创建执行计划 Spark 会尽可能地管道化，并基于是否要重新组织数据来划分 Stage，例如本例中的 `groupBy()` 转换就会将整个执行计划划分成两阶段执行。最终会产生一个 **DAG(directed acyclic graph , 有向无环图)** 作为逻辑执行计划。



步骤 3：调度任务将各阶段划分成不同的任务 (task)，每个任务都是数据和计算的合体。在进行下一阶段前，当前阶段的所有任务都要执行完成。因为下一阶段的第一个转换一定是重新组织数据的，所以必须等当前阶段所有结果数据都计算出来了才能继续。

假设本例中的 `hdfs://names` 下有四个文件块，那么 `HadoopRDD` 中 `partitions` 就会有四个分区对应这四个块数据，同时 `preferredLocations` 会指明这四个块的最佳位置。现在，就可以创建出四个任务，并调度到合适的集群结点上。



三、 Spark 在不同集群中的运行架构

Spark 注重建立良好的生态系统，它不仅支持多种外部文件存储系统，提供了多种多样的集群运行模式。部署在单台机器上时，既可以用本地（Local）模式运行，也可以使用伪分布式模式来运行；当以分布式集群部署的时候，可以根据自己集群的实际情况选择 Standalone 模式（Spark 自带的模式）、YARN-Client 模式或者 YARN-Cluster 模式。Spark 的各种运行模式虽然在启动方式、运行位置、调度策略上各有不同，但它们的目的是基本都是一致的，就是在合适的位置安全可靠的根据用户的配置和 Job 的需要运行和管理 Task。

1. Spark on Standalone 运行过程

Standalone 模式是 Spark 实现的资源调度框架，其主要的节点有 Client 节点、Master 节点和 Worker 节点。其中 Driver 既可以运行在 Master 节点上中，也可以运行在本地 Client 端。当用 spark-shell 交互式工具提交 Spark 的 Job 时，Driver 在 Master 节点上运行；当使用 spark-submit 工具提交 Job 或者在 Eclipse、IDEA 等开发平台上使用” new SparkConf.setManager(“spark://master:7077”)” 方式运行 Spark 任务时，Driver 是运行在本地 Client 端上的。

其运行过程如下：

1. SparkContext 连接到 Master，向 Master 注册并申请资源（CPU Core 和 Memory）；
2. Master 根据 SparkContext 的资源申请要求和 Worker 心跳周期内报告的信息决定在哪个 Worker 上分配资源，然后在该 Worker 上获取资源，然后启动 StandaloneExecutorBackend；
3. StandaloneExecutorBackend 向 SparkContext 注册（SparkDeploySchedulerBackend）；
4. SparkContext 将 Application 代码发送给 StandaloneExecutorBackend；并且 SparkContext 解析 Application 代码，构建 DAG 图，并提交给 DAG Scheduler 分解成 Stage（当碰到 Action 操作时，就会催生 Job；每个 Job 中含有 1 个或多个 Stage，Stage 一般在获取外部数据和 shuffle 之前产生），然后以 Stage（或者称为 TaskSet）提交给 Task Scheduler，Task Scheduler 负责将 Task 分配到相应的 Worker，最后提交给 StandaloneExecutorBackend 执行；
5. StandaloneExecutorBackend 会建立 Executor 线程池，开始执行 Task，并向 SparkContext 报告，直至 Task 完成。
6. 所有 Task 完成后，SparkContext 向 Master 注销，释放资源。

2. Spark on YARN 运行过程

YARN 是一种统一资源管理机制，在其上面可以运行多套计算框架。目前的大数据技术世界，大多数公司除了使用 Spark 来进行数据计算，由于历史原因或者单方面业务处理的性能考虑而使用着其他的计算框架，比如 MapReduce、Storm 等计算框架。Spark 基于此种情况开发了 Spark on YARN 的运行模式，由于借助了 YARN 良好的弹性资源管理机制，不仅部署 Application 更加方便，而且用户在 YARN 集群中运行的服务和 Application 的资源也完全隔离，更具实践应用价值的是 YARN 可以通过队列的方式，管理同时运行在集群中的多个服务。

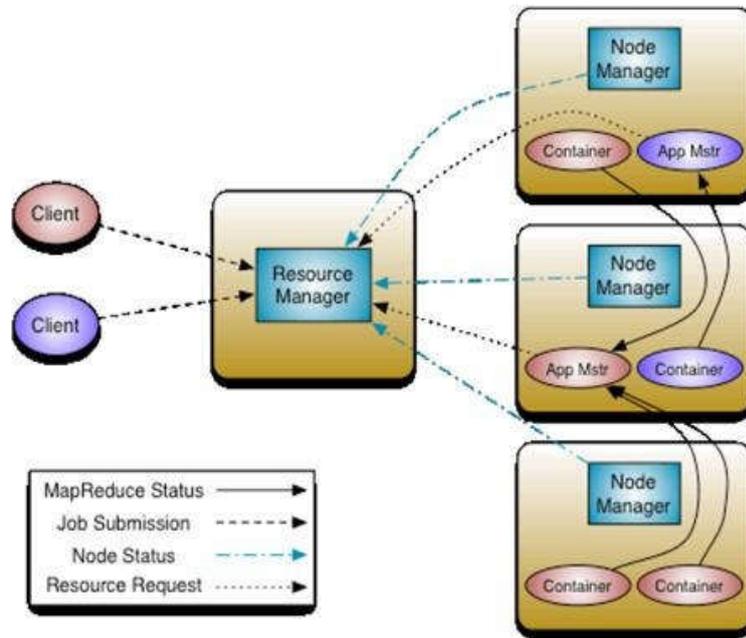
Spark on YARN 模式根据 Driver 在集群中的位置分为两种模式：一种是 YARN-Client 模式，另一种是 YARN-Cluster（或称为 YARN-Standalone 模式）。

YARN 框架流程

任何框架与 YARN 的结合，都必须遵循 YARN 的开发模式。在分析 Spark on YARN

的实现细节之前，有必要先分析一下 YARN 框架的一些基本原理。

Yarn 框架的基本运行流程图为：

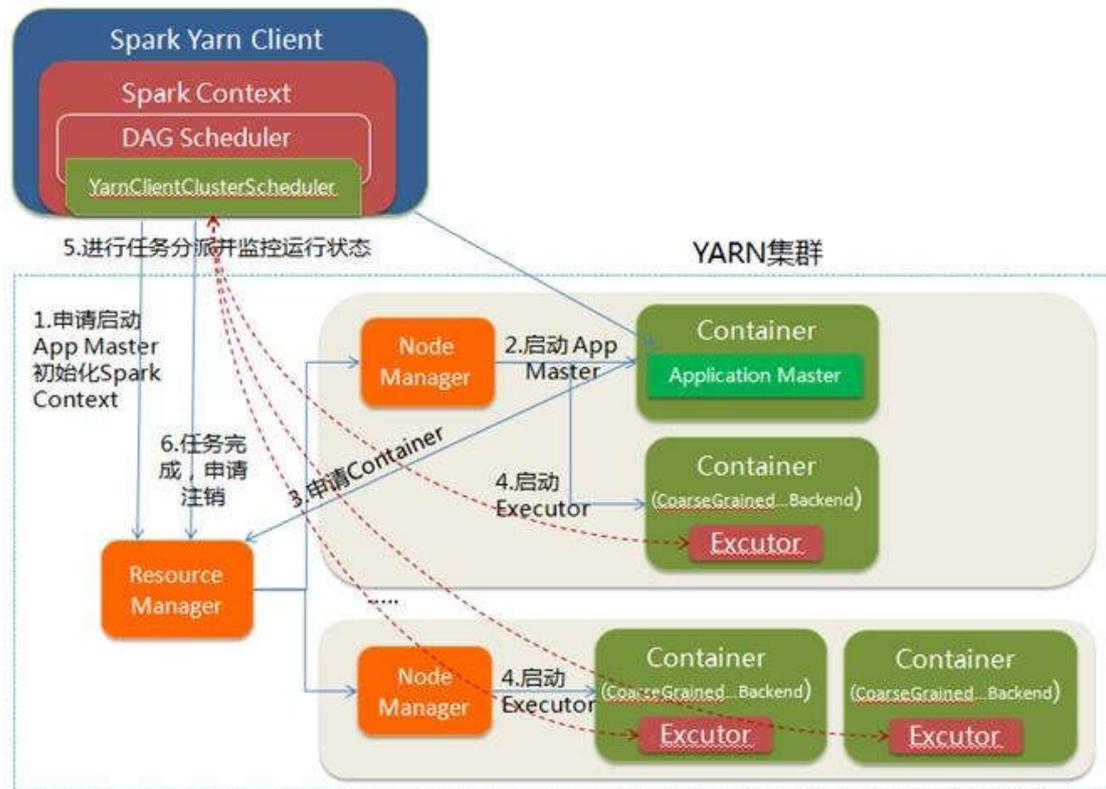


其中，ResourceManager 负责将集群的资源分配给各个应用使用，而资源分配和调度的基本单位是 Container，其中封装了机器资源，如内存、CPU、磁盘和网络等，每个任务会被分配一个 Container，该任务只能在该 Container 中执行，并使用该 Container 封装的资源。NodeManager 是一个个的计算节点，主要负责启动 Application 所需的 Container，监控资源（内存、CPU、磁盘和网络等）的使用情况并将之汇报给 ResourceManager。ResourceManager 与 NodeManagers 共同组成整个数据计算框架，ApplicationMaster 与具体的 Application 相关，主要负责同 ResourceManager 协商以获取合适的 Container，并跟踪这些 Container 的状态和监控其进度。

YARN-Client

Yarn-Client 模式中，Driver 在客户端本地运行，这种模式可以使得 Spark Application 和客户端进行交互，因为 Driver 在客户端，所以可以通过 webUI 访问 Driver 的状态，默认是 <http://hadoop1:4040> 访问，而 YARN 通过 <http://hadoop1:8088> 访问。

YARN-client 的工作流程分为以下几个步骤：



1. Spark Yarn Client 向 YARN 的 ResourceManager 申请启动 Application Master。同时在 SparkContext 初始化中将创建 DAGScheduler 和 TASKScheduler 等，由于我们选择的是 Yarn-Client 模式，程序会选择 YarnClientClusterScheduler 和 YarnClientSchedulerBackend；
2. ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster，与 YARN-Cluster 区别的是在该 ApplicationMaster 不运行 SparkContext，只与 SparkContext 进行联系进行资源的分派；
3. Client 中的 SparkContext 初始化完毕后，与 ApplicationMaster 建立通讯，向 ResourceManager 注册，根据任务信息向 ResourceManager 申请资源 (Container)；
4. 一旦 ApplicationMaster 申请到资源（也就是 Container）后，便与对应的 NodeManager 通信，要求它在获得的 Container 中启动启动 CoarseGrainedExecutorBackend, CoarseGrainedExecutorBackend 启动后会向 Client 中的 SparkContext 注册并申请 Task；
5. Client 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执

行,CoarseGrainedExecutorBackend 运行 Task 并向 Driver 汇报运行的状态和进度, 以让 Client 随时掌握各个任务的运行状态, 从而可以在任务失败时重新启动任务;

6. 应用程序运行完成后, Client 的 SparkContext 向 ResourceManager 申请注销并关闭自己。

YARN-Client 与 YARN-Cluster 区别

理解 YARN-Client 和 YARN-Cluster 深层次的区别之前先清楚一个概念:

Application Master。在 YARN 中, 每个 Application 实例都有一个 ApplicationMaster 进程, 它是 Application 启动的第一个容器。它负责和 ResourceManager 打交道并请求资源, 获取资源之后告诉 NodeManager 为其启动 Container。从深层次的含义讲 YARN-Cluster 和 YARN-Client 模式的区别其实就是 ApplicationMaster 进程的区别。

1. YARN-Cluster 模式下, Driver 运行在 AM(Application Master) 中, 它负责向 YARN 申请资源, 并监督作业的运行状况。当用户提交了作业之后, 就可以关掉 Client, 作业会继续在 YARN 上运行, 因而 YARN-Cluster 模式不适合运行交互类型的作业;
2. YARN-Client 模式下, Application Master 仅仅向 YARN 请求 Executor, Client 会和请求的 Container 通信来调度他们工作, 也就是说 Client 不能离开。

SparkSQL 调优

Spark 是一个快速的内存计算框架, 同时是一个并行运算的框架, 在计算性能调优的时候, 除了要考虑广为人知的木桶原理外, 还要考虑平行运算的 Amdahl 定理。

木桶原理又称短板理论, 其核心思想是: 一只木桶盛水的多少, 并不取决于桶壁上最高的那块木块, 而是取决于桶壁上最短的那块。将这个理论应用到系统性能优化上, 系统的最终性能取决于系统中性能表现最差的组件。例如, 即使系统拥有充足的内存资源和 CPU 资源, 但是如果磁盘 I/O 性能低下, 那么系统的总体性能是取决于当前最慢的磁盘 I/O 速度, 而不是当前最优越的 CPU 或者内存。在这种情况下, 如果需要进一步提升系统性能, 优化内存或者 CPU 资源是毫无用

处的。只有提高磁盘 I/O 性能才能对系统的整体性能进行优化。

Amdahl 定理，一个计算机科学界的经验法则，因吉恩·阿姆达尔而得名。它代表了处理器平行运算之后效率提升的能力。并行计算中的加速比是用并行前的执行速度和并行后的执行速度之比来表示的，它表示了在并行化之后的效率提升情况。阿姆达尔定律是固定负载（计算总量不变时）时的量化标准。可用公式：

$\frac{W_s + W_p}{W_s + \frac{W_p}{p}}$ 来表示。式中 W_s, W_p 分别表示问题规模的串行分量（问题中不能

并行化的那一部分）和并行分量， p 表示处理器数量。当 $p \rightarrow \infty$ 时，上式的

极限是 $\frac{W}{W_s}$ ，其中 $W = W^a + W^b$ 。这意味着无论我们如何增大处理器数目，加速比是无法高于这个数的。

SparkSQL 作为 Spark 的一个组件，在调优的时候，也要充分考虑到上面的两个原理，既要考虑如何充分的利用硬件资源，又要考虑如何利用好分布式系统的并行计算。由于测试环境条件有限，本篇不能做出更详尽的实验数据来说明，只能在理论上加以说明。

1. 并行性

那就要尽量提高查询任务的并行度。查询任务的并行度由两方面决定：集群的处理能力和集群的有效处理能力。

1. 对于 Spark Standalone 集群来说，集群的处理能力是由 conf/spark-env 中的 SPARK_WORKER_INSTANCES 参数、SPARK_WORKER_CORES 参数决定的；而 SPARK_WORKER_INSTANCES*SPARK_WORKER_CORES 不能超过物理机器的实际 CPU core；
2. 集群的有效处理能力是指集群中空闲的集群资源，一般是指使用 spark-submit 或 spark-shell 时指定的 --total-executor-cores，一般情况下，我们不需要指定，这时候，Spark Standalone 集群会将所有空闲的 core 分配给查询，并且在 Task 轮询运行过程中，Standalone 集群会将其他 spark 应用程序运行完后空闲出来的 core 也分配给正在运行中的查询。

综上所述，SparkSQL 的查询并行度主要和集群的 core 数量相关，合理配置每个节点的 core 可以提高集群的并行度，提高查询的效率。

2. 高效的数据格式

高效的数据格式，一方面是加快了数据的读入速度，另一方面可以减少内存的消耗。高效的数据格式包括多个方面：

3. 数据本地性

分布式计算系统的精粹在于数据不动代码动，但是在实际的计算过程中，总存在着移动数据的情况，除非是在集群的所有节点上都保存数据的副本。移动数据，将数据从一个节点移动到另一个节点进行计算，不但消耗了网络 IO，也消耗了磁盘 IO，降低了整个计算的效率。为了提高数据的本地性，除了优化算法（也就是修改 spark 内存，难度有点高），就是合理设置数据的副本。设置数据的副本，这需要通过配置参数并长期观察运行状态才能获取的一个经验值。

- PROCESS_LOCAL 是指读取缓存在本地节点的数据
- NODE_LOCAL 是指读取本地节点硬盘数据
- ANY 是指读取非本地节点数据
- 通常读取数据 PROCESS_LOCAL>NODE_LOCAL>ANY，尽量使数据以 PROCESS_LOCAL 或 NODE_LOCAL 方式读取。其中 PROCESS_LOCAL 还和 cache 有关。

4. 合适的数据类型

对于要查询的数据，定义合适的数据类型也是非常有必要。对于一个 tinyint 可以使用的数据列，不需要为了方便定义成 int 类型，一个 tinyint 的数据占用了 1 个 byte，而 int 占用了 4 个 byte。也就是说，一旦将这数据进行缓存的话，内存的消耗将增加数倍。在 SparkSQL 里，定义合适的数据类型可以节省有限的内存资源。

5. 合适的数据列

对于要查询的数据，在写 SQL 语句的时候，尽量写出要查询的列名，如 Select a,b from tbl，而不是使用 Select * from tbl；这样不但可以减少磁盘 IO，也减少缓存时消耗的内存。

6. 优的数据存储格式

在查询的时候，最终还是要读取存储在文件系统中的文件。采用更优的数据

存储格式，将有利于数据的读取速度。查看 SparkSQL 的 Stage，可以发现，很多时候，数据读取消耗占有很大的比重。对于 sqlContext 来说，支持 textFile、SequenceFile、ParquetFile、jsonFile；对于 hiveContext 来说，支持 AvroFile、ORCFile、Parquet File，以及各种压缩。根据自己的业务需求，测试并选择合适的数据存储格式将有利于提高 SparkSQL 的查询效率。

7. 内存的使用

spark 应用程序最纠结的地方就是内存的使用了，也是最能体现“细节是魔鬼”的地方。Spark 的内存配置项有不少，其中比较重要的几个是：

- SPARK_WORKER_MEMORY, 在 conf/spark-env.sh 中配置 SPARK_WORKER_MEMORY 和 SPARK_WORKER_INSTANCES, 可以充分的利用节点的内存资源, SPARK_WORKER_INSTANCES*SPARK_WORKER_MEMORY 不要超过节点本身具备的内存容量；
- executor-memory, 在 spark-shell 或 spark-submit 提交 spark 应用程序时申请使用的内存数量；不要超过节点的 SPARK_WORKER_MEMORY；
- spark.storage.memoryFraction spark 应用程序在所申请的内存资源中可用于 cache 的比例
- spark.shuffle.memoryFraction spark 应用程序在所申请的内存资源中可用于 shuffle 的比例

在实际使用上，对于后两个参数，可以根据常用查询的内存消耗情况做适当的变更。另外，在 SparkSQL 使用上，有几点建议：

- ✓ 对于频繁使用的表或查询才进行缓存，对于只使用一次的表不需要缓存；
- ✓ 对于 join 操作，优先缓存较小的表；
- ✓ 要多注意 Stage 的监控，多思考如何才能更多的 Task 使用 PROCESS_LOCAL；
- ✓ 要多注意 Storage 的监控，多思考如何才能 Fraction cached 的比例更多。

8. 合适的 Task

对于 SparkSQL，还有一个比较重要的参数，就是 shuffle 时候的 Task 数量，通过 spark.sql.shuffle.partitions 来调节。调节的基础是 spark 集群的处理能力和要处理的数据量，spark 的默认值是 200。Task 过多，会产生很多的任务启动开销，Task 多少，每个 Task 的处理时间过长，容易 straggle。

9. 其他的一些建议

优化的方面的内容很多，但大部分都是细节性的内容：

- 想要获取更好的表达式查询速度，可以将 `spark.sql.codegen` 设置为 `Ture`；
- 对于大数据集的计算结果，不要使用 `collect()`，`collect()` 就结果返回给 driver，很容易撑爆 driver 的内存；一般直接输出到分布式文件系统中；
- 对于 Worker 倾斜，设置 `spark.specuation=true` 将持续不给力的节点去掉；
- 对于数据倾斜，采用加入部分中间步骤，如聚合后 `cache`，具体情况具体分析；
- 适当的使用序化方案以及压缩方案；
- 善于利用集群监控系统，将集群的运行状况维持在一个合理的、平稳的状态；
- 善于解决重点矛盾，多观察 Stage 中的 Task，查看最耗时的 Task，查找原因并改善；

