为什么你的spark应用运行慢或者失败？

第一部分 内存管理

Spark应用容易开发和理解。然而，当spark应用开始变得很慢或者执行失败的情况下就很难处理。有时候一个经过优化的应用经常因为一个数据改变而执行失败。有时候一个一直正常运行的程序，由于缺少资源开始运行异常。这样的例子很多。

因此理解一个spark应用很重要，理解spark运行时组件比如磁盘使用、网络使用等也非常重要。这样当情况恶化时，我们可以作出明智的决定。

在一系列文章中，我将介绍spark应用失败或运行变慢的最常见的原因。第一个最常见的原因是内存管理。

内存溢出将是spark开发者最常遇见的问题，一些内存溢出的常见原因包括：

Spark的不正确使用

高并发

低效查询

错误配置

为了避免这些问题，我们需要对spark和我们的数据有一个基本的理解。对于您的应用来说，Spark的默认配置或许不合适。有时候，由于底层数据变化，一个经过很好优化的应用甚至可能会由于OOM失败。

内存溢出可以发生在driver node、executor node甚至node manager。我们看一下各个阶段的oom问题。

**Driver Level**

Spark中driver运行程序的控制流。通常情况下，由于不正确使用Spark，驱动程序会因OutOfMemory错误而失败。Spark是一个在work节点之间分配工作负载的引擎，driver可以看作是一个编排器。在典型的配置下，分配给driver的内存要比分配给executors的内存少。

导致driver oom的原因有：

rdd.collect()

sparkContext.broadcast

没有分配给应用driver所需的足够内存

Spark.sql.autoBroadcastJoinThreshold的错误配置。在join操作下，Spark使用这个来限制向所有节点广播一个关系。在第一次使用时，整个关系在driver节点上具体化。有时，作为查询执行的一部分，还会广播多个表。

避免在driver节点上执行explicit result collection的操作。您可以把这样的任务委托给一个executor节点执行。比如，如果您向保存结果到一个特定文件，您可以在driver上收集这些数据，也可以指定一个executor来执行。

比如：

//Inefficient code

Val result = dataFrame.collect()

saveTofile(result)

//better code

dataFrame.repartition(1).write.csv(“/file/path”)

如果您使用spark sql并且driver由于广播关系导致OOM，那么您可以增加分配给driver的内存或者减少spark.sql.autoBroadcastJoinThreshold的值，这样您的join操作将使用更适合内存的排序合并连接。

**Executor Level**

导致这个问题的原因有很多。最常见的有高并发，低效查询和错误配置。

1. 高并发

在理解高并发导致OOM错误的原因之前，我们尝试理解spark应用时如何执行一个查询或一个job，占用内存的操作是什么。

Spark job或者queries可分为多个stage，每个阶段被分成多个task。Tasks的数量由各种因素决定，像正在执行的stage，正在读取的数据源等等。如果在执行一个SQL map stage，通常task数量就是底层数据的分区数量。

比如，如果一个hive orc table有2000个分区，那么在分区修剪没有起作用的情况下，map stage将会创建2000个task来读取表的数据。如果在reduce stage（shuffle stage），spark将使用“spark.default.parallelism”设置来为RDD分配task数量，使用“spark.sql.shuffle.partitions”来为Dataset分配task数量。如果在不考虑内存的情况下将这些值设的很高，那么executor可能会因为OOM而失败。现在让我们看看当一个task执行时发生了什么，以及一些可能引起OOM的原因。

假如我们对一个HDFS文件或者一个Parquet/ORC表执行map task或SQL的scanning阶段。对于HDFS文件来说，每一个Spark task将读取一个128M的数据块，所以如果并行10个task，那么将至少需要128\*10M的内存来存储数据。这里还忽略了数据压缩可能带来的数据爆炸，这却觉得数据压缩算法。

Spark以向量格式读取Parquet文件。简单的说，每一个task从Parquet文件中逐批读取数据。由于Parquet文件是按列存储的，所以这些批次由每一列构成。在对该列执行任何操作之前，会在内存中积累一定数量的列数据。这意味着Spark需要一些数据结构和bookkeeping来存储这么多数据。另外类似字典编码的编码技术也会在内存中存储一些状态，这些都需要占用内存。

所以随着并发增加，总开销也会增加。同样如果有一些广播连接，那么这些广播的变量也会占用一些内存。

1. 低效查询

当spark的Catalyst engine尝试尽可能优化一个query时，如果这个query写的很糟糕，那么优化将不起作用。比如select Parquet/ORC表中所有的列，如上所述，每一列都需要占用内存，如果选择了过多的列，那么总开销也会越多。

尝试尽可能读取少的列。当有需要时尽可能使用filter，这样executors会加载少的数据。一些数据源支持partition pruning。如果您的query能够转换为使用分区列，那么将在很大程度上减少数据移动。

1. 不正确的配置

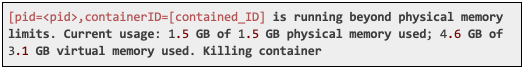
不正确的内存配置和加载也会导致spark任务失败和执行速度变慢。以下是一些例子：

1. EXECUTOR & DRIVER MEMORY

每一个应用需要的内存是不同的。根据需求，每一个应用都有不同配置。您应该配置正确的spark.executor.memory或者spark.driver.memory。而这个是最难作对的事情，我们需要监控工具来监控应用实际使用的内存。Unravel做的很好。

1. MEMORY OVERHEAD

有时候不是Executor memory，而是YARN总内存不足导致OOM或者node被YARN杀死。YARN kill的消息通常像这样：



YARN在container中运行spark的组件，比如executor和driver。Overhead memory是JVM开销，内部字符串和其他JVM元数据使用的off-heap内存。在这种情况下。您需要配置spark.yarn.executor.memeoryOverhead为一个合适的值，通常executor内存的10%应该分配给overhead。

1. CACHING MEMORY

如果您的应用使用Spark caching来存储一些数据，那么考虑spark memory manager设置将是必要的。Spark memory manager是以一种通用方式编写的，来满足所有的工作负载。所以，可以更改一些设置来匹配特定的工作。

Spark有定义的内存需求有两种：executor和storage。Storage memory用来caching，executor memeory用来存储临时数据比如hash tables。

Executor和storage memory可以通过一个可配置的比例来获得。这个配置项为spark.memeory.fraction，默认值为60%。除此之外，50%（由spark.memory.storageFraction决定）的可用内存分配给storage，其余的分配给executor。

1. OOM at node manger

Spark应用在执行group by或者join like操作时会执行data shuffling操作，这样会产生大量开销。通常data shuffling操作由executor完成。如果executor繁忙或者GC压力很大时，它将不能满足shuffle的需求。这个问题可以通过使用一个external shuffle service得到一定程度缓解。

external shuffle service在每一个worker node上执行来自executors的shuffle请求。Executors可以从这个service读取shuffle files而不是互相读取。这将帮助发出shuffle请求的executor读取shuffle files，即使正在处理的executor被kill或者运行很慢。另外，当dynamic allocation为enabled时，必须强制打开external shuffle service。

当spark external shuffle service由YARN配置，Node manager开启一个辅助的service当作一个external shuffle service。Node manager的默认内存为1G。然而，当应用处理大量data shufflinig会由于node manager OOM失败。如果应用程序属于上述类别，则必须正确配置NodeManager。

第二部分 data skew和garbage collection

什么是data skew

在理想的spark运行中，当spark进行join操作时，join keys将平均分布，并且每一个需要处理的分区将被很好的组织。但是，真实的商业数据很少像这样整洁。我们经常在spark集群中使用不太理想的数据组织，这样会导致由data skew引起的性能下降。

Data skew不是spark的问题，而是数据问题。Data skew问题的原因是底层数据的不均匀分布。数据不均匀分布有时是不可避免地。

对于spark joins和aggregations操作，需要将一个key地记录放在同一个partition中。一个key地记录在一个分区内，其他key记录将分布在其他分区内。如果一个分区变得很大，那么将会引起data skew，此时如果不采取一些处理，那么对于任何查询engine来说，都是有问题地。

处理data skew

当在join或aggregation操作中需要对数据进行shuffle时，data skew问题更明显。Shuffle操作将把相关的数据保持在一个partition中。因此，spark需要在cluster中移动数据，所以，shuffle是一个最昂贵的操作。

Data skew问题的表现为：

冻结的stages和tasks

CPU低利用率

Out of memory errors

有一些方法来处理data skew问题。

识别和解决data skew

Spark用户经常观察到，大多数任务在合理的时间内完成，只有一个任务需要永远完成。很可能，这表明您的数据集有偏差。这种行为还导致集群的总体利用率不足。在云中运行Spark时，这尤其是一个问题，因为在云中过度配置集群资源是浪费和昂贵的。

如果skew在数据源级别（例如，一个Hive表按月份键分区，并且该表在特定月份有更多的记录），这将导致在从表中读取的阶段中的处理发生倾斜。在这种情况下，重新构造具有不同分区键的表会有帮助。但是，有时这是不可行的，因为表可能被企业中的其他数据管道使用。

在这样情况下，有一些方法可以避免skewed data处理

Data broadcast

如果我们在skewed数据上做join操作，一个技巧是增加spark.sql.autoBroadcastJoinThreshold的值，使得小的table也被广播。这需要有足够的driver和executor memory。

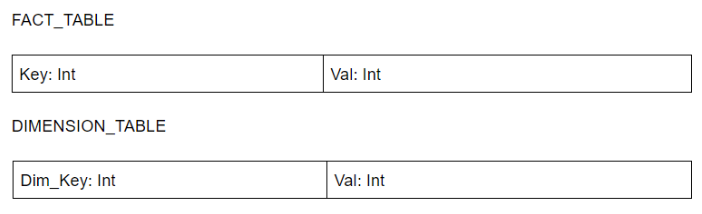
Data preprocess

如果join或group-by的key有太多null值，那么将会导致这个操作skew，尝试将这些null值赋予一些随机id，然后再spark应用中处理它们。

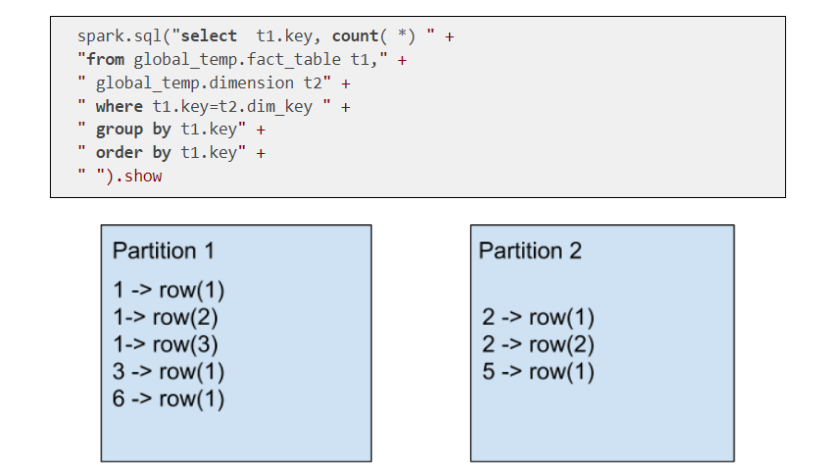
Salting

再一个SQL join操作中，join key将被改变来使得数据均匀分布，这样处理一个partition不需要太久时间。这个技术叫做salting。让我们看一个例子来理解salting做了什么。在一个join或者group-by操作中，spark通过计算key的hash值并除以shuffle分区的数量，将一个key数据映射到一个特定的partition。

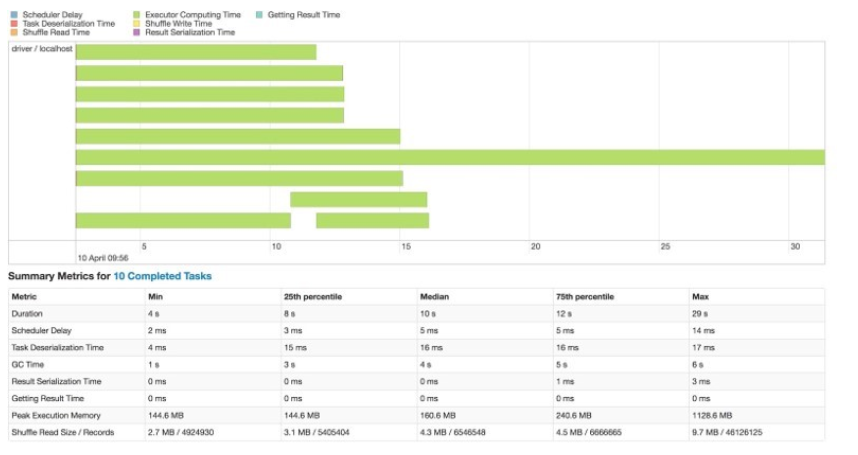
假设有两个表：



假设有个一个key的数据倾斜严重，比如key 1，我们进行一个join和group的操作来得到各个key的数量。比如：



在join操作的shuffle阶段，所有具有同样key的rows被分到了同一个partition。这里所有key等于1的数据都在partiton 1。很显然处理partition 1的数据将耗费更长时间，因为它包含更多数据。让我们检查spark ui来查看shuffle阶段运行的时间。



如图所示，一个task比其他task使用了更多时间。这可能会导致应用程序在内存使用方面不稳定。

我们是否可以添加一些内容以使我们的数据集更均匀地分布？大多数有歪斜问题的用户使用salting技术。salting是一种技术，我们将随机值添加到其中一个表的联接键中。在另一个表中，我们需要复制行以匹配随机键。其思想是，如果key1==key1满足连接条件，那么key1也应该满足连接条件。salt的值将帮助数据集更均匀地分布。

下面是一个例子：



然后我们检查spark ui，可以看到处理时间更加平均了。



由于shuffle在处理的时候会对数据进行压缩，在存在重复数据情况下，数据压缩可能会很严重，此时网络IO也会变少。我们的程序需要加入salt和不加入salt运行，以最终确定最适合我们的需要的方法。

Garbage Collection

Spark运行在Java虚拟机（JVM）上。因为Spark可以在内存中存储大量数据，所以它主要依赖Java的内存管理和垃圾收集（GC）。因此，垃圾收集（GC）可能是影响许多Spark应用程序的一个主要问题。

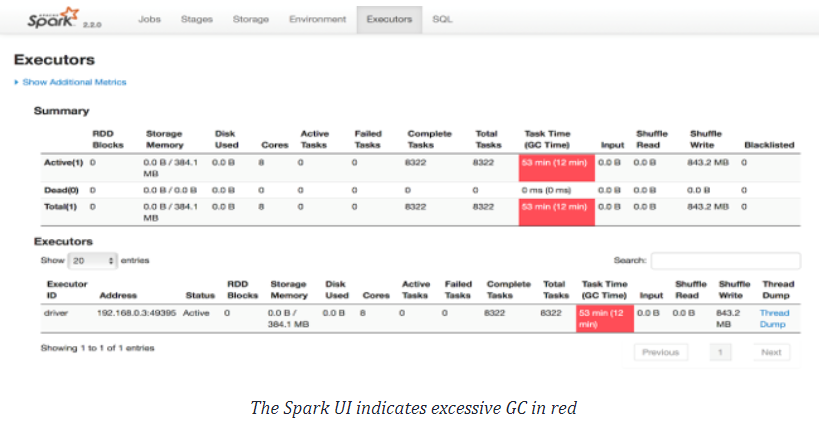
GC过多造成的问题有：

程序执行速度

Executor heartbeat timeout

GC overhead limit exceeded error

在spark应用中，很容易判断是否遇到GC问题。当花费过多时间在GC上时，spark ui会将executor标成红色。



解决GC问题

Data structures

如果使用基于RDD的应用，尽量少使用对象类型数据结构。比如，使用array代替list

定制的data structures

如果要处理原始数据类型，请考虑使用专门的数据结构，如Koloboke或fastutil。这些结构优化了基元类型的内存使用。

Off-heap存储数据

Spark执行引擎和Spark存储都可以从堆外存储数据。可以使用以下命令打开堆外存储： –conf spark.memory.offHeap.enabled = true

–conf spark.memory.offHeap.size = Xgb.

使用堆外存储时要小心，因为它不会影响堆内存大小，即不会收缩堆内存。因此，要定义总体内存限制，请指定较小的堆大小。

Built-in vs. User Defined Functions (UDFs)

尽可能使用built-in函数。

减少Object creation

请记住，我们可能正在处理数十亿行。如果我们为每一行创建一个100字节大小的临时对象，它将创建10亿\*100字节的垃圾。