Spark调优

Spark性能调优

常规性能调优

最优资源配置

可分配资源项

- --num-executors：在资源允许的情况下，增加Executor的个数可以提高执行task的并行度
- --driver-memory
- --executor-memory
  - 缓存更多的数据
  - 为shuffle操作提供更多内存
  - 为task的执行提供更多内存
- --executor-cores：在资源允许的情况下，增加每个Executor的Cpu core个数，可以提高执行task的并行度

RDD优化

- RDD复用
- RDD持久化
  - 对多次使用的RDD进行持久化
    - RDD的持久化是可以进行序列化的
    - 如果对于数据的可靠性要求很高，并且内存充足，可以使用副本机制
- RDD尽早filter操作

并行度调节

- 概述：指各个stage的task的数量
- task数量应该设置为Spark作业总CPU core数量的2~3倍

广播大变量

Kryo序列化

- 默认情况下，Spark使用Java的序列化机制。
- 从Spark 2.0.0版本开始，简单类型、简单类型数组、字符串类型的Shuffling RDDs 已经默认使用Kryo序列化方式
- Kryo需要用户在使用前注册需要序列化的类型

```
public class MyKryoRegistrator implements KryoRegistrator
{
@Override
public void registerClasses(Kryo kryo)
{
kryo.register(StartupReportLogs.class);
}
}
```

序列化机制配置

```
//创建SparkConf对象
val conf = new SparkConf().setMaster(...).setAppName(...)
//使用Kryo序列化库，如果要使用Java序列化库，需要把该行屏蔽掉
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
//在Kryo序列化库中注册自定义的类集合，如果要使用Java序列化库，需要把该行屏蔽掉
conf.set("spark.kryo.registrator", "atguigu.com.MyKryoRegistrator");
```

调节本地化等待时长

Spark本地化等级

- PROCESS_LOCAL：进程本地化，task和数据在同一个Executor中，性能最好。
- NODE_LOCAL：节点本地化，task和数据在同一个节点中，但是task和数据不在同一个Executor中，数据需要在进程间进行传输
- RACK_LOCAL：机架本地化，task和数据在同一个机架的两个节点上，数据需要通过网络在节点之间进行传输
- NO_PREF：对于task来说，从哪里获取都一样，没有好坏之分。
- ANY：task和数据可以在集群的任何地方，而且不在一个机架中，性能最差

算子调优

mapPartitions

- 容易OOM
- mapPartitions算子适用于数据量不是特别大的时候，此时使用mapPartitions算子对性能的提升效果还是不错的

foreachPartition

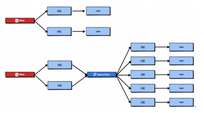
- 对于我们写的function函数，一次处理一整个分区的数据
- 对于一个分区内的数据，创建唯一的数据库连接
- 只需要向数据库发送一次SQL语句和多组参数

filter与coalesce的配合使用

- filter过滤后
  - 每个Partition数据量变小了
  - 每个Partition数据量不一样
- 针对于上述问题，采用coalesce算子：我们可以在filter操作之后，使用coalesce算子针对每个partition的数据量各不相同的情况，压缩partition的数量，而且让每个partition的数据量尽量均匀紧凑，以便于后面的task进行计算操作，在某种程度上能够在一定程度上提升性能

repartition解决SparkSQL低并行度问题

- Spark SQL的并行度不允许用户自己指定
- Spark SQL自己会默认根据hive表对应的HDFS文件的split个数自动设置Spark SQL所在的那个stage的并行度
- 做法：对于Spark SQL查询出来的RDD，立即使用repartition算子，去重新进行分区，这样可以重新分区为多个partition，从repartition之后的RDD操作，由于不再设计Spark SQL，因此stage的并行度就会等于你手动设置的值
- 图示



Shuffle调优

调节map端缓冲区大小

- map端缓冲的默认配置是32KB
- val conf = new SparkConf().set("spark.shuffle.file.buffer", "64")

调节reduce端拉取数据缓冲区大小

- 如果内存资源较为充足，适当增加拉取数据缓冲区的大小，可以减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能
- reduce端数据拉取缓冲区大小默认48MB
- val conf = new SparkConf().set("spark.reducer.maxSizeInFlight", "96")

调节reduce端拉取数据重试次数

- reduce端拉取数据重试次数默认为3
- val conf = new SparkConf().set("spark.shuffle.io.maxRetries", "6")

调节reduce端拉取数据等待间隔

- reduce端拉取数据等待间隔默认为5s
- val conf = new SparkConf().set("spark.shuffle.io.retryWait", "60s")

调节SortShuffle排序操作阈值

- SortShuffleManager排序操作阈值默认为200
- val conf = new SparkConf().set("spark.shuffle.sort.bypassMergeThreshold", "400")

JVM调优

降低cache操作的内存占比

- 静态内存管理机制
  - 堆
    - Storage：主要用于缓存RDD数据和broadcast数据
    - Execution：主要用于缓存在shuffle过程中产生的中间数据
  - 如果发现gc太频繁，时间太长，就可以考虑调节Storage的内存占比，让task执行算子函数式，有更多的内存可以使用
  - val conf = new SparkConf().set("spark.storage.memoryFraction", "0.4")
- 统一内存管理机制：Storage和Execution各占统一内存的50%，由于动态占用机制的实现，shuffle过程需要的内存过大时，会自动占用Storage的内存区域，因此无需手动进行调节

调节Executor堆外内存

- 报错信息：shuffle output file cannot find，executor lost，task lost，out of memory
- 如何配置：--conf spark.yarn.executor.memoryOverhead=2048

调节连接等待时长

- SparkGC会导致Executor进程停止工作，无法建立网络连接，即超时
- 报错信息：file not found、file lost
- 如何配置：--conf spark.core.connection.ack.wait.timeout=300

Spark数据倾斜

如何寻找

- 表现
  - Spark作业的大部分task都执行迅速，只有有限的几个task执行的非常慢
  - Spark作业的大部分task都执行迅速，但是有的task在运行过程中会突然报出OOM，反复执行几次都在某一个task报出OOM错误
- 查阅代码中的shuffle算子，例如reduceByKey、countByKey、groupByKey、join等算子，根据代码逻辑判断此处是否会出现数据倾斜
- 查看Spark作业的log文件，log文件对于错误的记录会精确到代码的某一行，可以根据异常定位到的代码位置来明确错误发生在第几个stage，对应的shuffle算子是哪一个；

解决方案

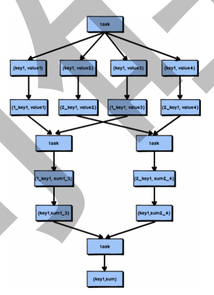
聚合原数据

- 避免shuffle过程：先在Hive表中对数据进行聚合；将同一key对应的所有value用一种特殊的格式拼接到一个字符串里
- 缩小key粒度：key的数量增加，可能使数据倾斜更严重
- 增大key粒度

过滤导致倾斜的key

提高shuffle操作中的reduce并行度

- reduce端并行度的设置：这个参数会决定shuffle过程中reduce端的并行度，在进行shuffle操作的时候，就会对应着创建指定数量的reduce task
- reduce端并行度设置存在的缺陷：提高reduce端并行度并没有从根本上改变数据倾斜的本质和问题



使用随机key实现双重聚合

- 图示
- 通过map算子给每个数据的key添加随机数前缀，对key进行打散，将原先一样的key变成不一样的key，然后进行第一次聚合
- 思路

将reduce join转换为map join

- 不适用场景分析：如果两个RDD数据量都比较大，容易OOM
- 不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作

sample采样对倾斜key单独进行join

- 当由单个key导致数据倾斜时，可有将发生数据倾斜的key单独提取出来，组成一个RDD
- 如果一个RDD中导致数据倾斜的key很多，那么此方案不适用。

使用随机数以及扩容进行join

- RDD中有大量的key导致数据倾斜
- key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理

SparkTroubleshooting

- 控制reduce端缓冲大小以避免OOM：以性能换执行
- JVM GC导致的shuffle文件拉取失败
  - val conf = new SparkConf()
  - .set("spark.shuffle.io.maxRetries", "60")
  - .set("spark.shuffle.io.retryWait", "60s")
- 解决各种序列化导致的报错
  - 作为RDD的元素类型的自定义类，必须是可以序列化的
  - 算子函数里可以使用的外部的自定义变量，必须是可以序列化的；
  - 不可以在RDD的元素类型、算子函数里使用第三方的不支持序列化的类型，例如Connection。
- 解决算子函数返回NULL导致的问题
  - 返回特殊值，不返回NULL，例如“-1”；
  - 在通过算子获取到了一个RDD之后，可以对这个RDD执行filter操作，进行数据过滤，将数值为-1的数据给过滤掉；
  - 在使用完filter算子后，继续调用coalesce算子进行优化。
- 解决YARN-CLIENT模式导致的网卡流量激增问题
  - 在YARN-client模式下，Driver启动在本地机器上，而Driver负责所有的任务调度，需要与YARN集群上的多个Executor进行频繁的通信
  - 在生产环境下，使用的一定是YARN-cluster模式。在YARN-cluster模式下，就不会造成本地机器网卡流量激增问题
- 解决YARN-CLUSTER模式的JVM栈内存溢出无法执行问题
  - 在YARN-cluster模式下，Driver运行在YARN集群的某个节点上，使用的是没有经过配置的默认设置，PermGen永久代大小为82MB
  - --conf spark.driver.extraJavaOptions="-XX:PermSize=128M -XX:MaxPermSize=256M"
- 解决SparkSQL导致的JVM栈内存溢出：一条sql语句的or关键字控制在100个以内，通常不会导致JVM栈内存溢出
- 持久化与checkpoint的使用