

# 尚硅谷大数据技术之 Spark 调优

(作者：尚硅谷研究院)

V1.0

## 第 1 章 Explain 查看执行计划

Spark 3.0 大版本发布，Spark SQL 的优化占比将近 50%。Spark SQL 取代 Spark Core，成为新一代的引擎内核，所有其他子框架如 Mllib、Streaming 和 Graph，都可以共享 Spark SQL 的性能优化，都能从 Spark 社区对于 Spark SQL 的投入中受益。

要优化 SparkSQL 应用时，一定是要了解 SparkSQL 执行计划的。发现 SQL 执行慢的根本原因，才能知道应该在哪儿进行优化，是调整 SQL 的编写方式、还是用 Hint、还是调参，而不是把优化方案拿来试一遍。

### 1.1 准备测试用表和数据

- 1、上传 3 个 log 到 hdfs 新建的 sparkdata 路径
- 2、hive 中创建 sparktuning 数据库
- 3、执行

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 4g --class com.atguigu.sparktuning.utils.InitUtil spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

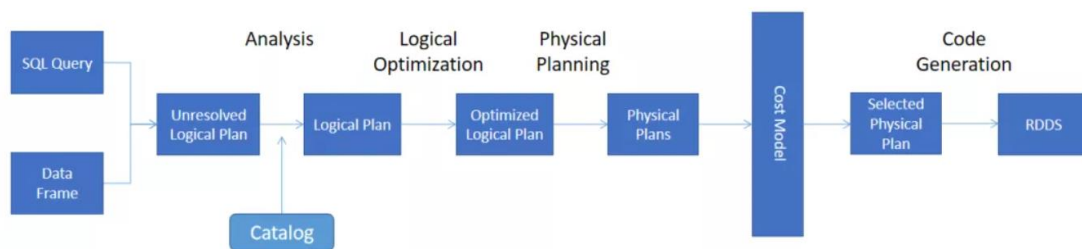
### 1.2 基本语法

```
.explain(mode="xxx")
```

从 3.0 开始，explain 方法有一个新的参数 mode，该参数可以指定执行计划展示格式：

- explain(mode="simple")：只展示物理执行计划。
- explain(mode="extended")：展示物理执行计划和逻辑执行计划。
- explain(mode="codegen")：展示要 Codegen 生成的可执行 Java 代码。
- explain(mode="cost")：展示优化后的逻辑执行计划以及相关的统计。
- explain(mode="formatted")：以分隔的方式输出，它会输出更易读的物理执行计划，并展示每个节点的详细信息。

### 1.3 执行计划处理流程



核心的执行过程一共有 5 个步骤：



这些操作和计划都是 Spark SQL 自动处理的，会生成以下计划：

- Unresolved 逻辑执行计划：== Parsed Logical Plan ==

Parser 组件检查 SQL 语法上是否有问题，然后生成 Unresolved（未决断）的逻辑计划，不检查表名、不检查列名。

- Resolved 逻辑执行计划：== Analyzed Logical Plan ==

通过访问 Spark 中的 Catalog 存储库来解析验证语义、列名、类型、表名等。

- 优化后的逻辑执行计划：== Optimized Logical Plan ==

Catalyst 优化器根据各种规则进行优化。

- 物理执行计划：== Physical Plan ==

1) HashAggregate 运算符表示数据聚合，一般 HashAggregate 是成对出现，第一个 HashAggregate 是将执行节点本地的数据进行局部聚合，另一个 HashAggregate 是将各个分区的数据进一步进行聚合计算。

2) Exchange 运算符其实就是 shuffle，表示需要在集群上移动数据。很多时候 HashAggregate 会以 Exchange 分隔开来。

3) Project 运算符是 SQL 中的投影操作，就是选择列（例如：select name, age...）。

4) BroadcastHashJoin 运算符表示通过基于广播方式进行 HashJoin。

5) LocalTableScan 运算符就是全表扫描本地的表。

## 1.4 案例实操

将提供的代码打成 jar 包，提交到 yarn 运行

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 4g --class com.atguigu.sparktuning.explain.ExplainDemo spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 第 2 章 资源调优

### 2.1 资源规划

#### 2.1.1 资源设定考虑

##### 1、总体原则

以单台服务器 128G 内存，32 线程为例。

先设定单个 Executor 核数，根据 Yarn 配置得出每个节点最多的 Executor 数量，每个节点的 yarn 内存/每个节点数量=单个节点的数量

总的 executor 数=单节点数量\*节点数。

##### 2、具体提交参数

###### 1) executor-cores

每个 executor 的最大核数。根据经验实践，设定在 3~6 之间比较合理。

###### 2) num-executors

该参数值=每个节点的 executor 数 \* work 节点数

每个 node 的 executor 数 = 单节点 yarn 总核数 / 每个 executor 的最大 cpu 核数

要给其他服务  
预留一些核

考虑到系统基础服务和 HDFS 等组件的余量，yarn.nodemanager.resource.cpu-vcores 配置为：28，参数 executor-cores 的值为：4，那么每个 node 的 executor 数 =  $28/4 = 7$ ，假设集群节点为 10，那么 num-executors =  $7 * 10 = 70$

###### 3) executor-memory

该参数值=yarn-nodemanager.resource.memory-mb / 每个节点的 executor 数量

如果 yarn 的参数配置为 100G，那么每个 Executor 大概就是  $100G/7 \approx 14G$ ，同时要注意 yarn 配置中每个容器允许的最大内存是否匹配。

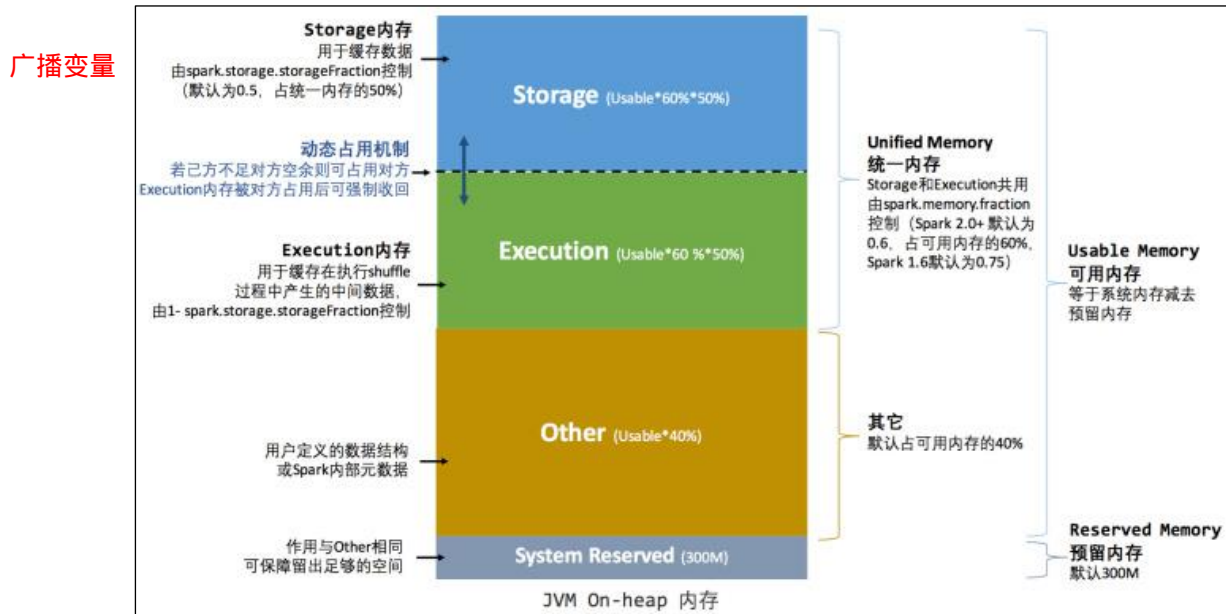
yarn容器允许分配的最小、最大内存：

yarn.scheduler.minimum-allocation-mb	1GB
yarn.scheduler.maximum-allocation-mb	8GB

这里每个executor大概分配14G，所以调高 yarn.scheduler.maximum-allocation-mb 参数值

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

## 2.1.2 内存估算



针对一个 executor

- 估算 Other 内存 = 自定义数据结构 \* 每个 Executor 核数
- 估算 Storage 内存 = 广播变量 + cache/Executor 数量 缓存数据在一个 executor 上占用的存储
- 估算 Executor 内存 = 每个 Executor 核数 \* (数据集大小/并行度)

这个除法理解成一个任务能处理多少数据  
数据集 100GB, 并行度是 200, 那么一个任务能处理 0.5GB,  
如果一个 executor 有 4 个核, 那么执行内存就为 2GB

## 2.1.3 调整内存配置项

一般情况下, 各个区域的内存比例保持默认值即可。如果需要更加精确的控制内存分配, 可以按照如下思路:

$\text{spark.memory.fraction} = (\text{估算 storage 内存} + \text{估算 Execution 内存}) / (\text{估算 storage 内存} + \text{估算 Execution 内存} + \text{估算 Other 内存})$  得到 调整存储内存和执行内存 占总的堆上内存的比例

$\text{spark.memory.storageFraction} = (\text{估算 storage 内存}) / (\text{估算 storage 内存} + \text{估算 Execution 内存})$  调整存储内存 占统一内存的比例

代入公式计算:

Storage 堆内内存 =  $(\text{spark.executor.memory} - 300\text{MB}) * \text{spark.memory.fraction} * \text{spark.memory.storageFraction}$

Execution 堆内内存 =

$(\text{spark.executor.memory} - 300\text{MB}) * \text{spark.memory.fraction} * (1 - \text{spark.memory.storageFraction})$

## 2.1 持久化和序列化

### 2.1.1 RDD

#### 1、cache

```
201
202 /**
203  * Persist this RDD with the default storage level (MEMORY_ONLY).
204  */
205 def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)
206
207 /**
208  * Persist this RDD with the default storage level (MEMORY_ONLY).
209  */
210 def cache(): this.type = persist()
211
```

打成 jar, 提交 yarn 任务,并在 yarn 界面查看 spark ui

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-
executors 3 --executor-cores 2 --executor-memory 6g --class
com.atguigu.sparktuning.cache.RddCacheDemo spark-tuning-1.0-SNAPSHOT-jar-
with-dependencies.jar
```

#### Storage

##### ▼ RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
5	MapPartitionsRDD	Memory Deserialized 1x Replicated	2	29%	2.5 GiB	0.0 B

通过 spark ui 看到, rdd 使用默认 cache 缓存级别, 占用内存 2.5GB,并且 storage 内存还不够, 只缓存了 29%。

#### 2、kryo+序列化缓存

使用 kryo 序列化并且使用 rdd 序列化缓存级别。使用 kryo 序列化需要修改 spark 的序列化模式, 并且需要进程注册类操作。

打成 jar 包在 yarn 上运行。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-
executors 3 --executor-cores 2 --executor-memory 6g --class
com.atguigu.sparktuning.cache.RddCacheKryoDemo spark-tuning-1.0-SNAPSHOT-
jar-with-dependencies.jar
```

查看 storage 所占内存, 内存占用减少了 1083.6mb 并且缓存了 100%。使用序列化缓存配合 kryo 序列化, 可以优化存储内存占用。

#### Storage

##### ▼ RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
5	MapPartitionsRDD	Memory Serialized 1x Replicated	7	100%	1083.6 MiB	0.0 B

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as <i>serialized</i> Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a <i>fast serializer</i> but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in <i>off-heap memory</i> . This requires off-heap memory to be enabled.

## Which Storage Level to Choose? [🔗](#)

Spark's storage levels are meant to provide different trade-offs between memory usage and CPU efficiency. We recommend going through the following process to select one:

- If your RDDs fit comfortably with the default storage level (MEMORY\_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
- If not, try using MEMORY\_ONLY\_SER and [selecting a fast serialization library](#) to make the objects much more space-efficient, but still reasonably fast to access. (Java and Scala)
- Don't spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.
- Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). All the storage levels provide full fault tolerance by recomputing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.

根据官网的描述，那么可以推断出，如果 yarn 内存资源充足情况下，使用默认级别 MEMORY\_ONLY 是对 CPU 的支持最好的。但是序列化缓存可以让体积更小，那么当 yarn 内存资源不充足情况下可以考虑使用 MEMORY\_ONLY\_SER 配合 kryo 使用序列化缓存。

## 2.1.2 DataFrame、DataSet

### 1、cache

提交任务，在 yarn 上查看 spark ui，查看 storage 内存占用。内存使用 612.3mb。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.cache.DatasetCacheDemo spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Spark 3.0.0 Jobs Stages Storage Environment Executors SQL test application UI					
Storage					
▼ RDDs					
ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory
3	*[1] ColumnarToRow ~ FileScan parquet dwd.dwd_course_pay[orderId#0,discount#1,paymoney#2,createtime#3,did#4,dn#5] Batched: true, DataFilters: [], Format: Parquet, Location: CatalogFileIndex[hdfs://mycluster/user/hive/warehouse/dwd.db/dwd_course_pay], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<orderid:string,discount:decimal(2,1),paymoney:decimal(10,4),createtime:timestamp>	Disk Memory Deserialized 1x Replicated	6	100%	612.3 MiB
					0.0 B

DataSet 的 cache 默认缓存级别与 RDD 不一样，是 MEMORY\_AND\_DISK。

源码：Dataset.cache() -> Dataset.persist() -> CacheManager.cacheQuery()

```
CacheManager.scala x
69
70 /**
71  * Caches the data produced by the logical representation of the given [[Dataset]].
72  * Unlike RDD.cache(), the default storage level is set to be MEMORY_AND_DISK because
73  * recomputing the in-memory columnar representation of the underlying table is expensive.
74  */
75 def cacheQuery(
76   query: Dataset[_],
77   tableName: Option[String] = None,
78   storageLevel: StorageLevel = MEMORY_AND_DISK): Unit = {
```

## 2、序列化缓存

DataSet 类似 RDD，但是并不使用 JAVA 序列化也不使用 Kryo 序列化，而是使用一种特有的编码器进行序列化对象。

## Creating Datasets [🔗](#)

Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized Encoder to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

打成 jar 包，提交 yarn。查看 spark ui, storage 占用内存 646.2mb。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.cache.DatasetCacheSerDemo spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Spark 3.0.0 Jobs Stages Storage Environment Executors SQL test application UI					
Storage					
▼ RDDs					
ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory
3	*(1) ColumnarToRow +- FileScan parquet dwd.dwd_course_pay[orderId#0,discount#1,paymoney#2,createTime#3,dn#4,dn#5] Batched: true, DataFilters: [], Format: Parquet, Location: CatalogFileIndex[hdfs://mycluster/user/hive/warehouse/dwd.db/dwd_course_pay], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<orderId:string,discount:decimal(2,1),paymoney:decimal(10,4),createTime:timestamp>	Disk Memory Serialized 1x Replicated	6	100%	646.2 MB
					0.0 B

和默认 cache 缓存级别差别不大。所以 DataSet 可以直接使用 cache。

从性能上来讲，DataSet, DataFrame 大于 RDD，建议开发中使用 DataSet、DataFrame。

## 2.2 CPU 优化

### 2.2.1 CPU 低效原因

#### 1、概念理解

##### 1) 并行度

➤ spark.default.parallelism

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网



spark.default.parallelism	For distributed shuffle operations like reduceByKey and join, the	Default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by user.	0.5.0
---------------------------	---	--	-------

设置 RDD 的默认并行度，没有设置时，由 join、reduceByKey 和 parallelize 等转换决定。

## ➤ spark.sql.shuffle.partitions

适用 SparkSQL 时，Shuffle Reduce 阶段默认的并行度，默认 200。此参数只能控制 Spark sql、DataFrame、DataSet 分区个数。不能控制 RDD 分区个数

spark.sql.shuffle.partitions	200	The default number of partitions to use when shuffling data for joins or aggregations. Note: For structured streaming, this configuration cannot be changed between query restarts from the same checkpoint location.	1.1.0
------------------------------	-----	---	-------

## 2) 并发度：同时执行的 task 数

### 2、CPU 低效原因

假设并行度为2，  
一个任务处理一个分片要占用更多的内存，  
导致处理其他的分片时，内存就不够了，  
就使得，只有这一个任务在执行，另一个任务无法执行，它占用的核也无法使用，就挂起

1) 并行度较低、数据分片较大容易导致 CPU 线程挂起

2) 并行度过高、数据过于分散会让调度开销更多

Executor 接收到 TaskDescription 之后，首先需要对 TaskDescription 反序列化才能读取任务信息，然后将任务代码再反序列化得到可执行代码，最后再结合其他任务信息创建 TaskRunner。当数据过于分散，分布式任务数量会大幅增加，但每个任务需要处理的数据量却少之又少，就 CPU 消耗来说，**相比花在数据处理上的比例，任务调度上的开销几乎与之分庭抗礼**。显然，在这种情况下，CPU 的有效利用率也是极低的。

## 2.2.2 合理利用 CPU 资源

每个并行度的数据量（总数据量/并行度） 在（Executor 内存/core 数/2, Executor 内存/core 数）区间 **0.75-1.5GB**

提交执行：

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 6g --class com.atguigu.sparktuning.partition.PartitionDemo spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

去向 yarn 申请的 executor vcore 资源个数为 12 个（num-executors\*executor-cores），如

就是“并行度过高、数据过于分散”情况

果不修改 spark sql 分区个数，那么就会像上图所展示存在 cpu 空转的情况。这个时候需要合理控制 shuffle 分区个数。**默认200**。如果想要让任务运行的最快当然是一个 task 对应一个 vcore,但是一般不会这样设置，为了合理利用资源，一般会**将并行度（task 数）设置成并发度（vcore 数）的 2 倍到 3 倍。**



修改参数 `spark.sql.shuffle.partitions`（默认 200），根据我们当前任务的提交参数有 12 个 vcore，将此参数设置为 24 或 36 为最优效果：

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 6g --class com.atguigu.sparktuning.partition.PartitionTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 第 3 章 SparkSQL 语法优化

SparkSQL 在整个执行计划处理的过程中，使用了 Catalyst 优化器。

### 3.1 基于 RBO 的优化

在 Spark 3.0 版本中，Catalyst 总共有 81 条优化规则（Rules），分成 27 组（Batches），其中有些规则会被归类到多个分组里。因此，如果不考虑规则的重复性，27 组算下来总共会有 129 个优化规则。

如果从优化效果的角度出发，这些规则可以归纳到以下 3 个范畴：

#### 3.1.1 谓词下推（Predicate Pushdown）

将过滤条件的谓词逻辑都尽可能提前执行，减少下游处理的数据量。对应 `PushDownPredicate` 优化规则，对于 Parquet、ORC 这类存储格式，结合文件注脚（Footer）中的统计信息，下推的谓词能够大幅减少数据扫描量，降低磁盘 I/O 开销。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 6g --class com.atguigu.sparktuning.PredicateTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

左外关联下推规则：左表 left join 右表

	左表	右表
Join 中条件（on）	只下推右表	只下推右表
Join 后条件（where）	两表都下推	两表都下推

**注意：外关联时，过滤条件写在 on 与 where，结果是不一样的！**

#### 3.1.2 列剪裁（Column Pruning）

列剪裁就是扫描数据源的时候，只读取那些与查询相关的字段。`Project`

#### 3.1.3 常量替换（Constant Folding）

假设我们在年龄上加的过滤条件是 “age < 12 + 18”，Catalyst 会使用 `ConstantFolding` 规则，自动帮我们吧条件变成 “age < 30”。再比如，我们在 select 语句中，掺杂了一些 **常量表达式**，Catalyst 也会自动地用表达式的结果进行替换。

更多 [Java](#) -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

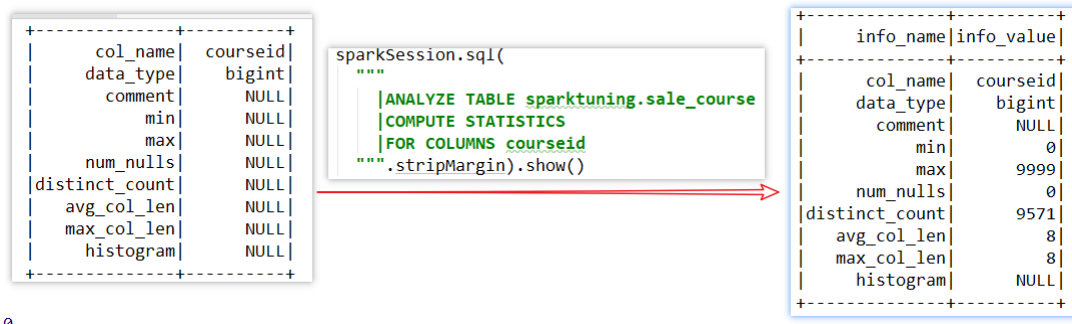


sizeInBytes 和原来的大小一样，则保留 rowCount（若存在），否则清除 rowCount。

### ➤ 生成列级别统计信息

ANALYZE TABLE 表名 COMPUTE STATISTICS FOR COLUMNS 列1,列2,列3

生成列统计信息，为保证一致性，会同步更新表统计信息。目前不支持复杂数据类型（如 Seq, Map 等）和 HiveStringType 的统计信息生成。



### ➤ 显示统计信息

DESC FORMATTED 表名

在 Statistics 中会显示“xxx bytes, xxx rows”分别表示表级别的统计信息。

dt	string	null
dn	string	null
# Partition Infor...		
# col_name	data_type	comment
dt	string	null
dn	string	null
# Detailed Table ...		
Database	sparktuning	
Table	sale_course	
Owner	root	
Created Time	Sat Nov 06 16:42:...	
Last Access	UNKNOWN	
Created By	Spark 3.0.0	
Type	MANAGED	
Provider	parquet	
Statistics	140628 bytes, 100...	
Location	hdfs://hadoop1:80...	
Serde Library	org.apache.hadoop...	
InputFormat	org.apache.hadoop...	
OutputFormat	org.apache.hadoop...	
Partition Provider	Catalog	

也可以通过如下命令显示列统计信息：

DESC FORMATTED 表名 列名

执行：

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 6g --class com.atguigu.sparktuning.cbo.StaticsCollect spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 3.2.2 使用 CBO

通过 "spark.sql.cbo.enabled" 来开启，默认是 false。配置开启 CBO 后，CBO 优化器可以基于表和列的统计信息，进行一系列的估算，最终选择出最优的查询计划。比如：Build 侧选择、优化 Join 类型、优化多表 Join 顺序等。

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

参数	描述	默认值
spark.sql.cbo.enabled	CBO 总开关。 true 表示打开，false 表示关闭。 要使用该功能，需确保相关表和列的统计信息已经生成。	false
spark.sql.cbo.joinReorder.enabled  不用等待前面的join完成， 再执行后面的join， 可以同时执行后面的join	使用 CBO 来自动调整连续的 inner join 的顺序。 true: 表示打开，false: 表示关闭 要使用该功能，需确保相关表和列的统计信息已经生成，且 CBO 总开关打开。	false
spark.sql.cbo.joinReorder.dp.threshold	使用 CBO 来自动调整连续 inner join 的表的个数阈值。 如果超出该阈值，则不会调整 join 顺序。	12

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 4g --class com.atguigu.sparktuning.cbo.CBOTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

### 3.3 广播 Join 大表和小表join

Spark join 策略中，如果当一张小表足够小并且可以先缓存到内存中，那么可以使用 Broadcast Hash Join,其原理就是先将小表聚合到 driver 端，再广播到各个大表分区中，那么再次进行 join 的时候，就相当于大表的各自分区的数据与小表进行本地 join，从而规避了 shuffle。

#### 1) 通过参数指定自动广播

广播 join 默认值为 10MB，由 spark.sql.autoBroadcastJoinThreshold 参数控制。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 4g --class com.atguigu.sparktuning.join.AutoBroadcastJoinTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

#### 2) 强行广播

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 4g --class com.atguigu.sparktuning.join.ForceBroadcastJoinTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

### 3.4 SMB Join 大表间join

SMB JOIN 是 sort merge bucket 操作，需要进行分桶，首先会进行排序，然后根据 key 值合并，把相同 key 的数据放到同一个 bucket 中（按照 key 进行 hash）。分桶的目的其实就是把大表化成小表。相同 key 的数据都在同一个桶中之后，再进行 join 操作，那么在联合的时候就会大幅度的减小无关项的扫描。

使用条件：

更多 [Java](#) -[大数据](#) -[前端](#) -[python](#) 人工智能资料下载，可百度访问：尚硅谷官网

- (1) 两表进行分桶，桶的个数必须相等
- (2) 两边进行 join 时，join 列=排序列=分桶列

不使用 SMB Join:

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.join.BigJoinDemo spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

使用 SMB Join:

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.join.SMBJoinTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 第 4 章 数据倾斜

### 4.1 数据倾斜现象

#### 1、现象

绝大多数 task 任务运行速度很快，但是就是有那么几个 task 任务运行极其缓慢，慢慢的可能就接着报内存溢出的问题。



#### 2、原因

数据倾斜一般是发生在 shuffle 类的算子，比如 distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup 等，涉及到数据重分区，如果其中某一个 key 数量特别大，就发生了数据倾斜。

### 4.2 数据倾斜大 key 定位

从所有 key 中，把其中每一个 key 随机取出来一部分，然后进行一个百分比的推算，

这是用局部取推算整体，虽然有点不准确，但是在整体概率上来说，我们只需要大概就可以定位那个最多的 key 了

执行：

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.join.SampleKeyDemo spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 4.3 单表数据倾斜优化

为了减少 shuffle 数据量以及 reduce 端的压力，通常 Spark SQL 在 map 端会做一个 partial aggregate（通常叫做预聚合或者偏聚合），即在 shuffle 前将同一分区内所属同 key 的记录先进行一个预结算，再将结果进行 shuffle，发送到 reduce 端做一个汇总，类似 MR 的提前 Combiner，所以执行计划中 HashAggregate 通常成对出现。

### 1、适用场景

聚合类的 shuffle 操作，部分 key 数据量较大，且大 key 的数据分布在很多不同的切片。

### 2、解决逻辑

随机数

两阶段聚合（加盐局部聚合+去盐全局聚合）

### 3、案例演示

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.skew.SkewAggregationTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 4.4 Join 数据倾斜优化

### 4.4.1 广播 Join

#### 1、适用场景

适用于小表 join 大表。小表足够小，可被加载进 Driver 并通过 Broadcast 方法广播到各个 Executor 中。

#### 2、解决逻辑

在小表 join 大表时如果产生数据倾斜，那么广播 join 可以直接规避掉此 shuffle 阶段。直接优化掉 stage。并且广播 join 也是 Spark Sql 中最常用的优化方案。

#### 3、案例演示

2.2.2 中的 PartitionTuning 案例关闭了广播 join，可以看到数据倾斜

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.skew.SkewMapJoinTuning spark-tuning-1.0-SNAPSHOT-
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
jar-with-dependencies.jar
```

## 4.4.2 拆分大 key 打散大表 扩容小表

### 1、适用场景

适用于 join 时出现数据倾斜。 **其中一张表存在倾斜**

### 2、解决逻辑

1) 将存在倾斜的表，根据抽样结果，拆分为倾斜 key (skew 表) 和没有倾斜 key (common) 的两个数据集。

2) 将 skew 表的 key 全部加上随机前缀，然后对另外一个不存在严重数据倾斜的数据集 (old 表) 整体与随机前缀集作笛卡尔乘积 (即将数据量扩大 N 倍，得到 new 表)。

3) 打散的 skew 表 join 扩容的 new 表

union

Common 表 join old 表

以下为打散大 key 和扩容小表的实现思路

1) 打散大表：实际就是数据一进一出进行处理，对大 key 前拼上随机前缀实现打散

2) 扩容小表：实际就是将 DataFrame 中每一条数据，转成一个集合，并往这个集合里循环添加 10 条数据，最后使用 flatmap 压平此集合，达到扩容的效果。

### 3、案例演示

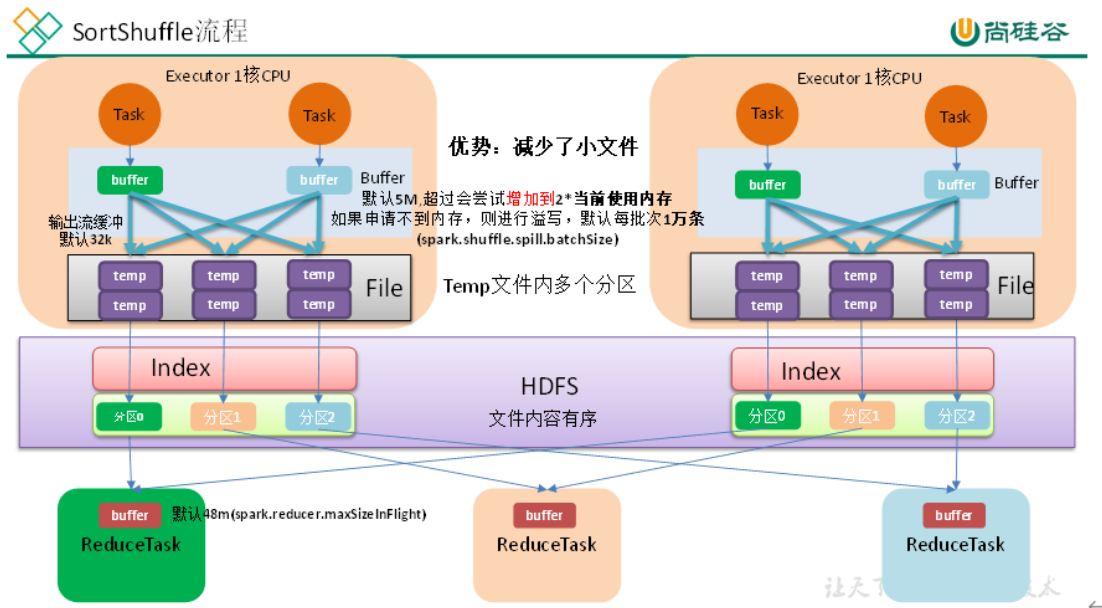
```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.skew.SkewJoinTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 4.4.3 参设开启 AQE

详见 6.3



## 第 5 章 Job 优化



## 5.1 Map 端优化

### 5.1.1 Map 端聚合

map-side 预聚合，就是在每个节点本地对相同的 key 进行一次聚合操作，类似于 MapReduce 中的本地 combiner。map-side 预聚合之后，每个节点本地就只会有一条相同的 key，因为多条相同的 key 都被聚合起来了。其他节点在拉取所有节点上的相同 key 时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘 IO 以及网络传输开销。

RDD 的话建议使用 `reduceByKey` 或者 `aggregateByKey` 算子来替代掉 `groupByKey` 算子。

因为 `reduceByKey` 和 `aggregateByKey` 算子都会使用用户自定义的函数对每个节点本地的相同 `key` 进行预聚合。而 `groupByKey` 算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

SparkSQL 本身的 HashAggregate 就会实现本地预聚合+全局聚合。

### 5.1.2 读取小文件优化

读取的数据源有很多小文件，会造成查询性能的损耗，大量的数据分片信息以及对应产生的 Task 元信息也会给 Spark Driver 的内存造成压力，带来单点问题。

设置参数:

spark.sql.files.maxPartitionBytes=128MB 默认 128m

```
spark.files.openCostInBytes=4194304
```

默认 4m 设成小文件的大小

更多 Java -大数据 -前端 -python 人工智能资料下载,可百度访问: 尚硅谷官网

参数（单位都是 bytes）：

- `maxPartitionBytes`：一个分区最大字节数。
- `openCostInBytes`：打开一个文件的开销。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.map.MapSmallFileTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

源码理解： `DataSourceScanExec.createNonBucketedReadRDD()`

```
def maxSplitBytes(
  sparkSession: SparkSession,
  selectedPartitions: Seq[PartitionDirectory]): Long = {
  val defaultMaxSplitBytes = sparkSession.sessionState.conf.filesMaxPartitionBytes
  val openCostInBytes = sparkSession.sessionState.conf.filesOpenCostInBytes
  val defaultParallelism = sparkSession.sparkContext.defaultParallelism
  val totalBytes = selectedPartitions.flatMap(_.files.map(_.getLen + openCostInBytes)).sum
  val bytesPerCore = totalBytes / defaultParallelism
  Math.min(defaultMaxSplitBytes, Math.max(openCostInBytes, bytesPerCore))
}
```

`FilePartition.getFilePartitions()`

```
def getFilePartitions(
  sparkSession: SparkSession,
  partitionedFiles: Seq[PartitionedFile],
  maxSplitBytes: Long): Seq[FilePartition] = {
  val partitions = new ArrayBuffer[FilePartition]
  val currentFiles = new ArrayBuffer[PartitionedFile]
  var currentSize = 0L

  /** Close the current partition and move to the next. */
  def closePartition(): Unit = {
    if (currentFiles.nonEmpty) {
      // Copy to a new Array.
      val newPartition = FilePartition(partitions.size, currentFiles.toArray)
      partitions += newPartition
    }
    currentFiles.clear()
    currentSize = 0
  }

  val openCostInBytes = sparkSession.sessionState.conf.filesOpenCostInBytes
  // Assign files to partitions using "Next Fit Decreasing"
  partitionedFiles.foreach { file =>
    if (currentSize + file.length > maxSplitBytes) {
      closePartition()
    }
    // Add the given file to the current partition.
    currentSize += file.length + openCostInBytes
    currentFiles += file
  }
}
```

加了打开文件的开销

1) 切片大小 = `Math.min(defaultMaxSplitBytes, Math.max(openCostInBytes, bytesPerCore))`

计算 `totalBytes` 的时候，每个文件都要加上一个 `open` 开销

`defaultParallelism` 就是 RDD 的并行度

2) 当（文件 1 大小 + `openCostInBytes`）+（文件 2 大小 + `openCostInBytes`）+...+（文件 n-1 大小 + `openCostInBytes`）+ 文件 n <= `maxPartitionBytes` 时，n 个文件可以读入同一个分区，即满足：`N 个小文件总大小 + (N-1) * openCostInBytes <= maxPartitionBytes` 的话。

更多 Java - 大数据 - 前端 - python 人工智能资料下载，可百度访问：尚硅谷官网  
这样能倒推出来大致的 N

### 5.1.3 增大 map 溢写时输出流 buffer 调整 spark.shuffle.file.buffer 参数

1) map 端 Shuffle Write 有一个缓冲区，初始阈值 5m，超过会尝试增加到 2\*当前使用内存。如果申请不到内存，则进行溢写。这个参数是 **internal**，指定无效（见下方源码）。也就是说资源足够会自动扩容，所以不需要我们去设置。

2) 溢写时使用输出流缓冲区默认 32k，这些缓冲区减少了磁盘搜索和系统调用次数，适当提高可以提升溢写效率。

3) Shuffle 文件涉及到序列化，是采取批的方式读写，默认按照每批次 1 万条去读写。设置得太低会导致在序列化时过度复制，因为一些序列化器通过增长和复制的方式来翻倍内部数据结构。这个参数是 **internal**，指定无效（见下方源码）。

综合以上分析，我们可以调整的就是输出缓冲区的大小。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.map.MapFileBufferTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

源码理解：

```
* Spills the current in-memory collection to disk if needed. Attempts to acquire more
* memory before spilling.
* 
* @param collection collection to spill to disk
* @param currentMemory estimated size of the collection in bytes
* @return true if collection was spilled to disk; false otherwise
*/
protected def maybeSpill(collection: C, currentMemory: Long): Boolean = {
  var shouldSpill = false
  if (elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold) {
    // Claim up to double our current memory from the shuffle memory pool
    val amountToRequest = 2 * currentMemory - myMemoryThreshold
    val granted = acquireMemory(amountToRequest)
    myMemoryThreshold += granted
    // If we were granted too little memory to grow further (either tryToAcquire returned 0,
    // or we already had more memory than myMemoryThreshold), spill the current collection
    shouldSpill = currentMemory >= myMemoryThreshold
  }
  shouldSpill = shouldSpill || _elementsRead > numElementsForceSpillThreshold
  // Actually spill
  if (shouldSpill) {
    _spillCount += 1
  }
}
```

```

private[this] def spillMemoryIteratorToDisk(inMemoryIterator: Iterator[(K, C)])
    : DiskMapIterator = {
    val (blockId, file) = diskBlockManager.createTempLocalBlock()
    val writer = blockManager.getDiskWriter(blockId, file, ser, fileBufferSize, writeMetrics)
    var objectsWritten = 0

    // List of batch sizes (bytes) in the order they are written to disk
    val batchSizes = new ArrayBuffer[Long]

    // Flush the disk writer's contents to disk, and update relevant variables
    def flush(): Unit = {
        val segment = writer.commitAndGet()
        batchSizes += segment.length
        diskBytesSpilled += segment.length
        objectsWritten = 0
    }

    var success = false
    try {
        while (inMemoryIterator.hasNext) {
            val kv = inMemoryIterator.next()
            writer.write(kv._1, kv._2)
            objectsWritten += 1

            if (objectsWritten == serializerBatchSize) {
                flush()
            }
        }
    }
}

```

默认32k

序列化器批次, 默认10000条

```

private[spark] val SHUFFLE_SPILL_INITIAL_MEM_THRESHOLD =
    ConfigBuilder("spark.shuffle.spill.initialMemoryThreshold")
        .internal()
        .doc("Initial threshold for the size of a collection before we start tracking its " +
            "memory usage.")
        .version("1.1.1")
        .bytesConf(ByteUnit.BYTE)
        .createWithDefault(default = 5 * 1024 * 1024)

def internal(): ConfigBuilder = {
    _public = false
    this
}

```

```

private[spark] val SHUFFLE_SPILL_BATCH_SIZE =
    ConfigBuilder("spark.shuffle.spill.batchSize")
        .internal()
        .doc("Size of object batches when reading/writing from serializers.")
        .version("0.9.0")
        .longConf
        .createWithDefault(default = 10000)

def internal(): ConfigBuilder = {
    _public = false
    this
}

```

## 5.2 Reduce 端优化

### 5.2.1 合理设置 Reduce 数

过多的 cpu 资源出现空转浪费, 过少影响任务性能。关于并行度、并发度的相关参数介绍, 在 2.2.1 中已经介绍过。

## 5.2.2 输出产生小文件优化

### 1、Join 后的结果插入新表

join 结果插入新表，生成的文件数等于 shuffle 并行度，默认就是 200 份文件插入到 hdfs 上。

解决方式：**shuffle后**

1) 可以在插入表数据前进行缩小分区操作来解决小文件过多问题，如 coalesce、repartition 算子。

2) 调整 shuffle 并行度。根据 2.2.2 的原则来设置。

### 2、动态分区插入数据

1) 没有 Shuffle 的情况下。最差的情况下，每个 Task 中都有表各个分区的记录，那文件数最终文件数将达到 Task 数量 \* 表分区数。这种情况下是极易产生小文件的。

```
INSERT overwrite table A partition ( aa )  
SELECT * FROM B;
```

2) 有 Shuffle 的情况下，上面的 Task 数量 就变成了 spark.sql.shuffle.partitions（默认值 200）。那么最差情况就会有 spark.sql.shuffle.partitions \* 表分区数。

当 spark.sql.shuffle.partitions 设置过大时，小文件问题就产生了；当 spark.sql.shuffle.partitions 设置过小时，任务的并行度就下降了，性能随之受到影响。

最理想的情况是根据分区字段进行 shuffle，在上面的 sql 中加上 distribute by aa。把同一分区的记录都哈希到同一个分区中去，由一个 Spark 的 Task 进行写入，这样的话只会产生 N 个文件,但是这种情况下也容易出现数据倾斜的问题。

**解决思路：**

结合第 4 章解决倾斜的思路，在确定哪个分区键倾斜的情况下，将倾斜的分区键单独拎出来：

将入库的 SQL 拆成（where 分区 != 倾斜分区键）和（where 分区 = 倾斜分区键）几个部分，非倾斜分区键的部分正常 distribute by 分区字段，倾斜分区键的部分 distribute by 随机数，sql 如下：

```
//1.非倾斜键部分  
INSERT overwrite table A partition ( aa )  
SELECT *  
FROM B where aa != 大key  
distribute by aa;  
  
//2.倾斜键部分  
INSERT overwrite table A partition ( aa )
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
SELECT *
FROM B where aa = 大key
distribute by cast(rand() * 5 as int);
```

案例实操:

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-
executors 3 --executor-cores 2 --executor-memory 6g --class
com.atguigu.sparktuning.reduce.DynamicPartitionSmallFileTuning spark-
tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

### 5.2.3 增大 reduce 缓冲区，减少拉取次数

Spark Shuffle 过程中，shuffle reduce task 的 buffer 缓冲区大小决定了 reduce task 每次能够缓冲的数据量，也就是每次能够拉取的数据量，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 端数据拉取缓冲区的大小可以通过 `spark.reducer.maxSizeInFlight` 参数进行设置，默认为 48MB。

源码: BlockStoreShuffleReader.read()

```
/** Read the combined key-values for this reduce task */
override def read(): Iterator[Product2[K, C]] = {
  val wrappedStreams = new ShuffleBlockFetcherIterator(
    context,
    blockManager.blockStoreClient,
    blockManager,
    blocksByAddress,
    serializerManager.wrapStream,
    // Note: we use getSizeAsMb when no suffix is provided for backwards compatibility
    SparkEnv.get.conf.get(config.REDUCER_MAX_SIZE_IN_FLIGHT) * 1024 * 1024,
    SparkEnv.get.conf.get(config.REDUCER_MAX_REQS_IN_FLIGHT),
    SparkEnv.get.conf.get(config.REDUCER_MAX_BLOCKS_IN_FLIGHT_PER_ADDRESS),
    SparkEnv.get.conf.get(config.MAX_REMOTE_BLOCK_SIZE_FETCH_TO_MEM),
    SparkEnv.get.conf.get(config.SHUFFLE_DETECT_CORRUPT),
    SparkEnv.get.conf.get(config.SHUFFLE_DETECT_CORRUPT_MEMORY),
    readMetrics,
    fetchContinuousBlocksInBatch.toCompletionIterator
  )

  private[spark] val REDUCER_MAX_SIZE_IN_FLIGHT = ConfigBuilder("spark.reducer.maxSizeInFlight")
    .doc("Maximum size of map outputs to fetch simultaneously from each reduce task, " +
      "in MiB unless otherwise specified. Since each output requires us to create a " +
      "buffer to receive it, this represents a fixed memory overhead per reduce task, " +
      "so keep it small unless you have a large amount of memory")
    .version("1.4.0")
    .bytesConf(ByteUnit.MiB)
    .createWithDefaultString( default = "48m")
}
```

### 5.2.4 调节 reduce 端拉取数据重试次数

Spark Shuffle 过程中，reduce task 拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试。对于那些包含了特别耗时的 shuffle 操作的作业，建议增加重试最大次数（比如 60 次），以避免由于 JVM 的 full gc 或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的 shuffle 过程，调节该参数

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网



可以大幅度提升稳定性。 6次

reduce 端拉取数据重试次数可以通过 `spark.shuffle.io.maxRetries` 参数进行设置，该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败，默认为 3：

### 5.2.5 调节 reduce 端拉取数据等待间隔

Spark Shuffle 过程中，reduce task 拉取属于自己的数据时，如果因为网络异常等原因导致失败会自动进行重试，在一次失败后，会等待一定的时间间隔再进行重试，可以通过加大间隔时长（比如 60s），以增加 shuffle 操作的稳定性。

reduce 端拉取数据等待间隔可以通过 `spark.shuffle.io.retryWait` 参数进行设置，默认值为 5s。

综合 5.2.3、5.2.4、5.2.5，案例实操：

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 6g --class com.atguigu.sparktuning.reduce.ReduceShuffleTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

### 5.2.6 合理利用 bypass

当 ShuffleManager 为 SortShuffleManager 时，如果 shuffle read task 的数量小于这个阈值（默认是 200）且不需要 map 端进行合并操作，则 shuffle write 过程中不会进行排序操作，使用 BypassMergeSortShuffleWriter 去写数据，但是最后会将每个 task 产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

当你使用 SortShuffleManager 时，如果确实不需要排序操作，那么建议将这个参数调大一些，大于 shuffle read task 的数量。那么此时就会自动启用 bypass 机制，map-side 就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此 shuffle write 性能有待提高。

源码分析：`SortShuffleManager.registerShuffle()`



```

/**
 * Obtains a [[ShuffleHandle]] to pass to tasks.
 */
override def registerShuffle[K, V, C](
  shuffleId: Int,
  dependency: ShuffleDependency[K, V, C]): ShuffleHandle = {
  if (SortShuffleWriter.shouldBypassMergeSort(conf, dependency)) {
    // If there are fewer than spark.shuffle.sort.bypassMergeThreshold partitions and we don't
    // need map-side aggregation, then write numPartitions files directly and just concatenate
    // them at the end. This avoids doing serialization and deserialization twice to merge
    // together the spilled files, which would happen with the normal code path. The downside is
    // having multiple files open at a time and thus more memory allocated to buffers.
    new BypassMergeSortShuffleHandle[K, V](
      shuffleId, dependency.asInstanceOf[ShuffleDependency[K, V, V]])
  } else if (SortShuffleManager.canUseSerializedShuffle(dependency)) {
    // Otherwise, try to buffer map outputs in a serialized form, since this is more efficient:
    new SerializedShuffleHandle[K, V](
      shuffleId, dependency.asInstanceOf[ShuffleDependency[K, V, V]])
  } else {
    // Otherwise, buffer map outputs in a deserialized form:
    new BaseShuffleHandle(shuffleId, dependency)
  }
}

private[spark] object SortShuffleWriter {
  def shouldBypassMergeSort(conf: SparkConf, dep: ShuffleDependency[_ , _ , _]): Boolean = {
    // We cannot bypass sorting if we need to do map-side aggregation.
    if (dep.mapSideCombine) {
      false
    } else {
      val bypassMergeThreshold: Int = conf.get(config.SHUFFLE_SORT_BYPASS_MERGE_THRESHOLD)
      dep.partitioner.numPartitions <= bypassMergeThreshold
    }
  }
}

private[spark] val SHUFFLE_SORT_BYPASS_MERGE_THRESHOLD =
  ConfigBuilder("spark.shuffle.sort.bypassMergeThreshold")
    .doc("In the sort-based shuffle manager, avoid merge-sorting data if there is no " +
      "map-side aggregation and there are at most this many reduce partitions")
    .version("1.1.1")
    .intConf
    .createWithDefault(default = 200)

```

SortShuffleManager.getWriter()

```

case bypassMergeSortHandle: BypassMergeSortShuffleHandle[K @unchecked, V @unchecked] =>
  new BypassMergeSortShuffleWriter(
    env.blockManager,
    bypassMergeSortHandle,
    mapId,
    env.conf,
    metrics,
    shuffleExecutorComponents)

```

```

spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-
executors 3 --executor-cores 2 --executor-memory 6g --class
com.atguigu.sparktuning.reduce.BypassTuning spark-tuning-1.0-SNAPSHOT-
jar-with-dependencies.jar

```

## 5.3 整体优化

### 5.3.1 调节数据本地化等待时长

也可以在 web ui

在 Spark 项目开发阶段，可以使用 client 模式对程序进行测试，此时，可以在本地看到

比较全的日志信息，日志信息中有明确的 Task 数据本地化的级别，如果大部分都是

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

PROCESS\_LOCAL、NODE\_LOCAL，那么就无需进行调节，但是如果发现很多的级别都是 RACK\_LOCAL、ANY，那么需要对本地化的等待时长进行调节，应该是反复调节，每次调节完以后，再来运行观察日志，看看大部分的 task 的本地化级别有没有提升；看看，整个 spark 作业的运行时间有没有缩短。

注意过犹不及，不要将本地化等待时长延长地过长，导致因为大量的等待时长，使得 Spark 作业的运行时间反而增加了。

下面几个参数，默认都是 3s，可以改成如下：

```
spark.locality.wait           //建议 6s、10s 全局
spark.locality.wait.process   //建议 60s
spark.locality.wait.node      //建议 30s
spark.locality.wait.rack      //建议 20s
```

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-
executors 3 --executor-cores 2 --executor-memory 6g --class
com.atguigu.sparktuning.job.LocalityEngine spark-tuning-1.0-SNAPSHOT-
jar-with-dependencies.jar
```

### 5.3.2 使用堆外内存

#### 1、堆外内存参数

讲到堆外内存，就必须去提一个东西，那就是去 yarn 申请资源的单位，容器。Spark on yarn 模式，一个容器到底申请多少内存资源。

一个容器最多可以申请多大资源，是由 yarn 参数 yarn.scheduler.maximum-allocation-mb 决定，需要满足：

$$\text{spark.executor.memoryOverhead} + \text{spark.executor.memory} + \text{spark.memory.offHeap.size} \leq \text{yarn.scheduler.maximum-allocation-mb}$$

参数解释：

- spark.executor.memory：提交任务时指定的堆内内存。
- spark.executor.memoryOverhead：堆外内存参数，内存额外开销。

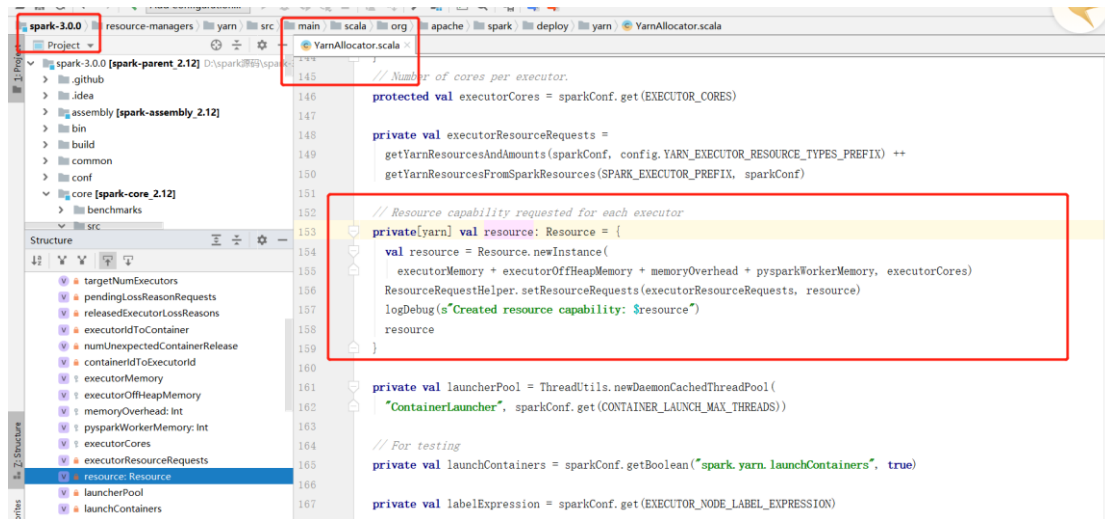
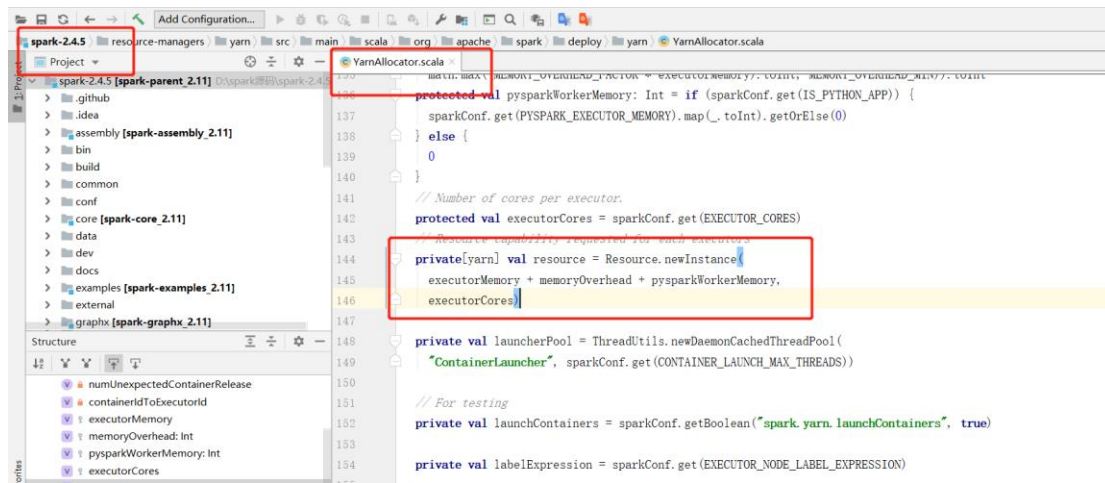
默认开启，默认值为  $\text{spark.executor.memory} * 0.1$  并且会与最小值 384mb 做对比，取最大值。所以 spark on yarn 任务堆内内存申请 1 个 g，而实际去 yarn 申请的内存大于 1 个 g 的原因。

- spark.memory.offHeap.size：堆外内存参数，spark 中默认关闭，需要将 spark.memory.enable.offheap.enable 参数设置为 true。

注意：很多网上资料说 spark.executor.memoryOverhead 包含 spark.memory.offHeap.size，

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](http://www.shangguigu.com)

这是由版本区别的，仅限于 spark3.0 之前的版本。3.0 之后就发生改变，实际去 yarn 申请的内存资源由三个参数相加。



测试申请容器上限：

yarn.scheduler.maximum-allocation-mb 修改为 7G，将三个参数设为如下，大于 7G，会报错：

```

spark-submit --master yarn --deploy-mode client --driver-memory 1g --
num-executors 3 --executor-cores 4 --conf
spark.memory.offHeap.enabled=true --conf spark.memory.offHeap.size=2g --
executor-memory 5g --class com.atguigu.sparktuning.join.SMBJoinTuning
spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
  
```

将 spark.memory.offHeap.size 修改为 1g 后再次提交：

```

spark-submit --master yarn --deploy-mode client --driver-memory 1g --
num-executors 3 --executor-cores 4 --conf
spark.memory.offHeap.enabled=true --conf spark.memory.offHeap.size=1g --
executor-memory 5g --class com.atguigu.sparktuning.join.SMBJoinTuning
spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
  
```

## 2、使用堆外缓存

使用堆外内存可以减轻垃圾回收的工作，也加快了复制的速度。

当需要缓存非常大的数据量时，虚拟机将承受非常大的 GC 压力，因为虚拟机必须检

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

查每个对象是否可以收集并必须访问所有内存页。本地缓存是最快的，但会给虚拟机带来 GC 压力，所以，当你需要处理非常多 GB 的数据量时可以考虑使用堆外内存来进行优化，因为这不会给 Java 垃圾收集器带来任何压力。让 JAVA GC 为应用程序完成工作，缓存操作交给堆外。

**堆外存大量缓存数据，堆内存用来运行程序**

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --conf spark.memory.offHeap.enabled=true --conf spark.memory.offHeap.size=1g --executor-memory 5g --class com.atguigu.sparktuning.job.OFFHeapCache spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

### 5.3.3 调节连接等待时长

在 Spark 作业运行过程中，Executor 优先从自己本地关联的 BlockManager 中获取某份数据，如果本地 BlockManager 没有的话，会通过 TransferService 远程连接其他节点上 Executor 的 BlockManager 来获取数据。

如果 task 在运行过程中创建大量对象或者创建的对象较大，会占用大量的内存，这会导致频繁的垃圾回收，但是垃圾回收会导致工作现场全部停止，也就是说，垃圾回收一旦执行，Spark 的 Executor 进程就会停止工作，无法提供相应，此时，由于没有响应，无法建立网络连接，会导致网络连接超时。

在生产环境下，有时会遇到 file not found、file lost 这类错误，在这种情况下，很有可能是 Executor 的 BlockManager 在拉取数据的时候，无法建立连接，然后超过默认的连接等待时长 120s 后，宣告数据拉取失败，如果反复尝试都拉取不到数据，可能会导致 Spark 作业的崩溃。这种情况也可能导致 DAGScheduler 反复提交几次 stage，TaskScheduler 反复提交几次 task，大大延长了我们的 Spark 作业的运行时间。

为了避免长时间暂停(如 GC)导致的超时，可以考虑调节连接的超时时长，连接等待时长需要在 spark-submit 脚本中进行设置，设置方式可以在提交时指定：

```
--conf spark.core.connection.ack.wait.timeout=300s
```

调节连接等待时长后，通常可以避免部分的 XX 文件拉取失败、XX 文件 lost 等报错。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 1g --conf spark.core.connection.ack.wait.timeout=300s --class com.atguigu.sparktuning.job.AckWaitTuning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 第 6 章 Spark3.0 AQE

Spark 在 3.0 版本推出了 AQE (Adaptive Query Execution)，即自适应查询执行。AQE 是 Spark SQL 的一种动态优化机制，在运行时，每当 Shuffle Map 阶段执行完毕，AQE 都会结

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

合这个阶段的统计信息，基于既定的规则动态地调整、修正尚未执行的逻辑计划和物理计划，来完成对原始查询语句的运行优化。

## 6.1 动态合并分区

在 Spark 中运行查询处理非常大的数据时，shuffle 通常会对查询性能产生非常重要的影响。shuffle 是非常昂贵的操作，因为它需要进行网络传输移动数据，以便下游进行计算。

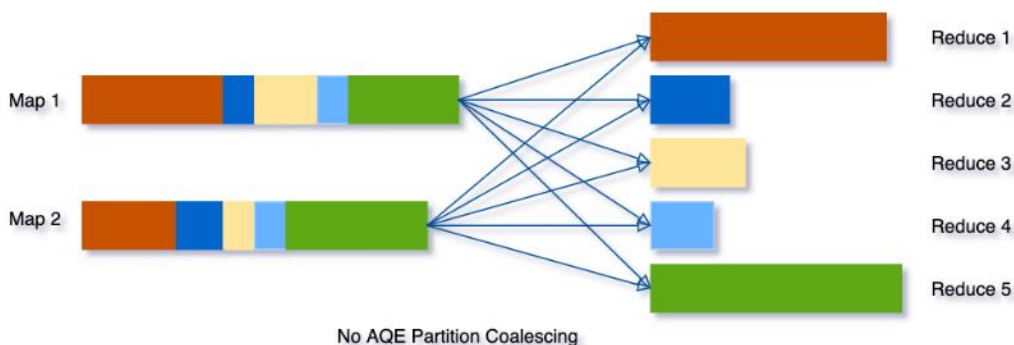
最好的分区取决于数据，但是每个查询的阶段之间的数据大小可能相差很大，这使得该数字难以调整：

（1）如果分区太少，则每个分区的数据量可能会很大，处理这些数据量非常大的分区，可能需要将数据溢写到磁盘（例如，排序和聚合），降低了查询。

（2）如果分区太多，则每个分区的数据量大小可能很小，读取大量小的网络数据块，这也会导致 I/O 效率低而降低了查询速度。拥有大量的 task（一个分区一个 task）也会给 Spark 任务计划程序带来更多负担。

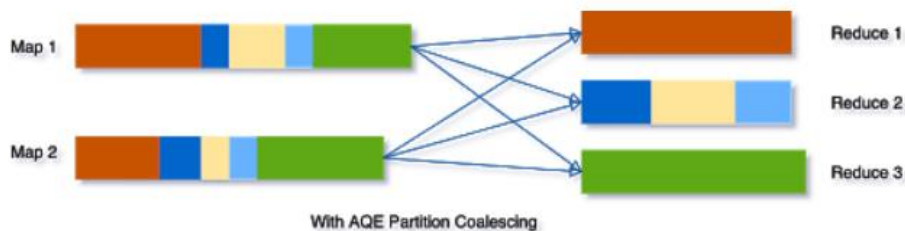
为了解决这个问题，我们可以先在任务开始时先设置较多的 shuffle 分区个数，然后在运行时通过查看 shuffle 文件统计信息将相邻的小分区合并成更大的分区。

例如，假设正在运行 `select max(i) from tbl group by j`。输入 tbl 很小，在分组前只有 2 个分区。那么任务刚初始化时，我们将分区数设置为 5，如果没有 AQE，Spark 将启动五个任务来进行最终聚合，但是其中会有三个非常小的分区，为每个分区启动单独的任务这样就浪费。



取而代之的是，AQE 将这三个小分区合并为一个，因此最终聚只需三个 task 而不是五个





```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 2g --class com.atguigu.sparktuning.aqe.AQEPartitionTunning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

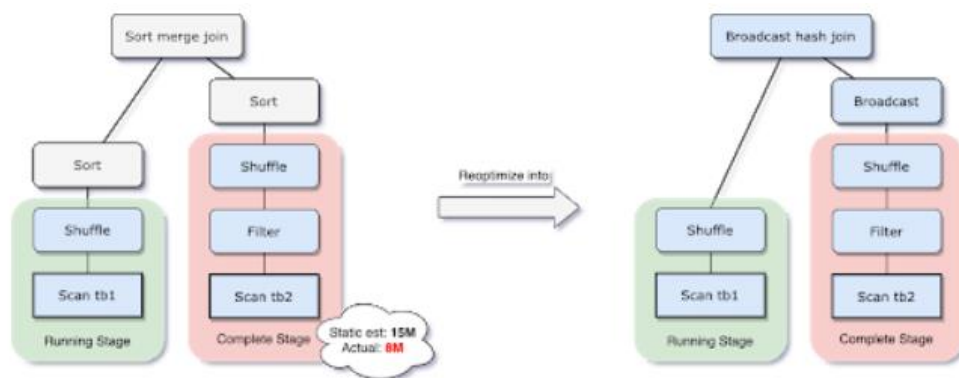
结合动态申请资源：

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 2 --executor-memory 2g --class com.atguigu.sparktuning.aqe.DynamicAllocationTunning spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 6.2 动态切换 Join 策略

Spark 支持多种 join 策略，其中如果 join 的一张表可以很好的插入内存，那么 broadcast shah join 通常性能最高。因此，spark join 中，如果小表小于广播大小阈值（默认 10mb），Spark 将计划进行 broadcast hash join。但是，很多事情都会使这种大小估计出错（例如，存在选择性很高的过滤器），或者 join 关系是一系列的运算符而不是简单的扫描表操作。

为了解决此问题，AQE 现在根据最准确的 join 大小运行时重新计划 join 策略。从下图实例中可以看出，发现连接的右侧表比左侧表小的多，并且足够小可以进行广播，那么 AQE 会重新优化，将 sort merge join 转换成为 broadcast hash join。



对于运行是的 broadcast hash join,可以将 shuffle 优化成本地 shuffle,优化掉 stage 减少网络传输。Broadcast hash join 可以规避 shuffle 阶段，相当于本地 join。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 2g --class
```

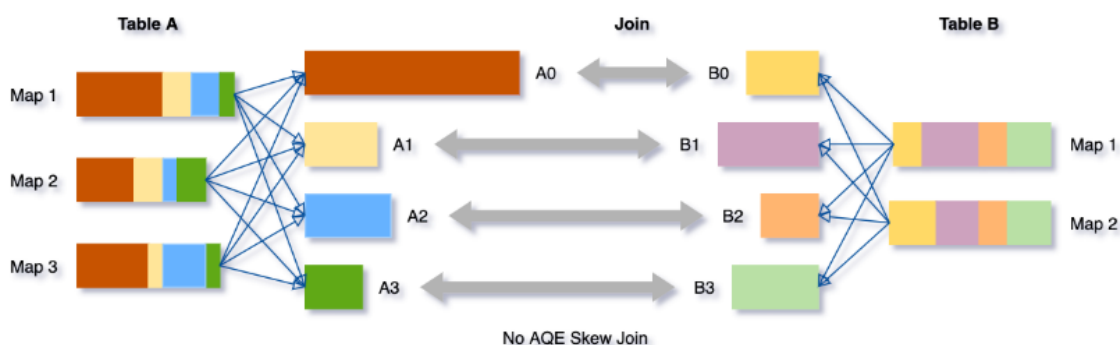
更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网

```
com.atguigu.sparktuning.aqe.AqeDynamicSwitchJoin spark-tuning-1.0-  
SNAPSHOT-jar-with-dependencies.jar
```

### 6.3 动态优化 Join 倾斜

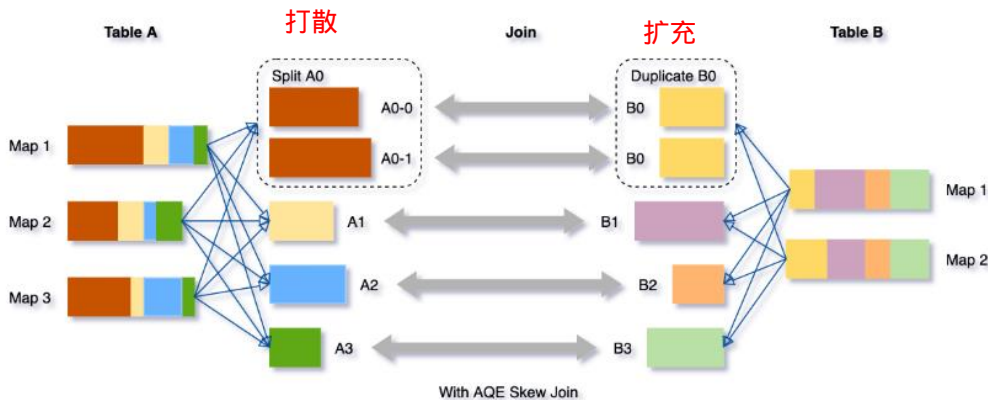
当数据在群集中的分区之间分布不均匀时，就会发生数据倾斜。严重的倾斜会大大降低查询性能，尤其对于 join。AQE skew join 优化会从随机 shuffle 文件统计信息自动检测到这种倾斜。然后它将倾斜分区拆分成较小的子分区。

例如,下图 A join B,A 表中分区 A0 明细大于其他分区



对应的分区数统计会增加，超过默认的200

因此，skew join 会将 A0 分区拆分成两个子分区，并且对应连接 B0 分区



没有这种优化，会导致其中一个分区特别耗时拖慢整个 stage,有了这个优化之后每个 task 耗时都会大致相同，从而总体上获得更好的性能。

可以采取第 4 章提到的解决方式，3.0 有了 AQE 机制就可以交给 Spark 自行解决。Spark3.0 增加了以下参数。

- 1) `spark.sql.adaptive.skewJoin.enabled` :是否开启倾斜 join 检测，如果开启了，那么会将倾斜的分区数据拆成多个分区,默认是开启的，但是得打开 aqe。
- 2) `spark.sql.adaptive.skewJoin.skewedPartitionFactor` :默认值 5，此参数用来判断分区数



据量是否数据倾斜，当任务中最大数据量分区对应的数据量大于的分区中位数乘以参数，并且也大于 spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes 参数，那么此任务是数据倾斜。

3) spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes :默认值 256mb，用于判断是否数据倾斜

4) spark.sql.adaptive.advisoryPartitionSizeInBytes :此参数用来告诉 spark 进行拆分后推荐分区大小是多少。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 2g --class com.atguigu.sparktuning.age.AgeOptimizingSkewJoin spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

如果同时开启了 spark.sql.adaptive.coalescePartitions.enabled 动态合并分区功能，那么会先合并分区，再去判断倾斜，将动态合并分区打开后，重新执行：

分区数就会小于200

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 2g --class com.atguigu.sparktuning.age.AgeOptimizingSkewJoin spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

修改中位数的倍数为 2，重新执行：

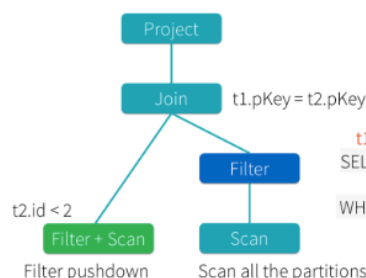
```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 2g --class com.atguigu.sparktuning.age.AgeOptimizingSkewJoin spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 第 7 章 Spark3.0 DPP

Spark3.0 支持动态分区裁剪 Dynamic Partition Pruning，简称 DPP，核心思路就是先将 join 一侧作为子查询计算出来，再将其所有分区用到 join 另一侧作为表过滤条件，从而实现对分区的动态修剪。如下图所示

### Dynamic Partition Pruning

```
SELECT t1.id, t2.pKey
FROM t1
JOIN t2
ON t1.pKey = t2.pKey
AND t2.id < 2
```



```
t1.pKey IN (
SELECT t2.pKey
FROM t2
WHERE t2.id < 2)
```

过滤出来t1的key 在过滤和的t2里的数据  
SubqueryBroadcast dynamici pruning

将 select t1.id,t2.pkey from t1 join t2 on t1.pkey =t2.pkey and t2.id<2 优化成了 select t1.id,t2.pkey from t1 join t2 on t1.pkey=t2.pkey and t1.pkey in(select t2.pkey from t2 where

t2.id<2)

触发条件:

- (1) 待裁剪的表 join 的时候, join 条件里必须有分区字段
- (2) 如果是需要修剪左表, 那么 join 必须是 inner join ,left semi join 或 right join,反之亦然。但如果是 left out join,无论右边有没有这个分区, 左边的值都存在, 就不需要被裁剪
- (3) 另一张表需要存在至少一个过滤条件, 比如 a join b on a.key=b.key and a.id<2

参数 spark.sql.optimizer.dynamicPartitionPruning.enabled 默认开启。

```
spark-submit --master yarn --deploy-mode client --driver-memory 1g --num-executors 3 --executor-cores 4 --executor-memory 2g --class com.atguigu.sparktuning.dpp.DPPTest spark-tuning-1.0-SNAPSHOT-jar-with-dependencies.jar
```

## 第 8 章 Spark3.0 Hint 增强

优先级为: Broadcast Hash Join > Sort Merge Join > Shuffle Hash Join > cartesian Join > Broadcast Nested Loop Join.

在 spark2.4 的时候就有了 hint 功能, 不过只有 broadcasthash join 的 hint,这次 3.0 又增加了 sort merge join,shuffle\_hash join,shuffle\_replicate nested loop join。

Spark 的 5 种 Join 策略: <https://www.cnblogs.com/jmx-bigdata/p/14021183.html>

### 8.1 broadcasthash join

```
sparkSession.sql("select /*+ BROADCAST(school) */ * from test_student student left join test_school school on student.id=school.id").show()
sparkSession.sql("select /*+ BROADCASTJOIN(school) */ * from test_student student left join test_school school on student.id=school.id").show()
sparkSession.sql("select /*+ MAPJOIN(school) */ * from test_student student left join test_school school on student.id=school.id").show()
```

### 8.2 sort merge join

```
sparkSession.sql("select /*+ SHUFFLE_MERGE(school) */ * from test_student student left join test_school school on student.id=school.id").show()
sparkSession.sql("select /*+ MERGEJOIN(school) */ * from test_student student left join test_school school on student.id=school.id").show()
sparkSession.sql("select /*+ MERGE(school) */ * from test_student student left join test_school school on student.id=school.id").show()
```

### 8.3 shuffle\_hash join

```
sparkSession.sql("select /*+ SHUFFLE_HASH(school) */ * from test_student student left join test_school school on student.id=school.id").show()
```

### 8.4 shuffle\_replicate\_nl join

使用条件非常苛刻, 驱动表 (school 表) 必须小,且很容易被 spark 执行成 sort merge join。

```
sparkSession.sql("select /*+ SHUFFLE_REPLICATE_NL(school) */ * from
```

更多 Java -大数据 -前端 -python 人工智能资料下载, 可百度访问: 尚硅谷官网

```
test_student student inner join test_school school on
student.id=school.id").show()
```

## 第 9 章 故障排除

### 9.1 故障排除一：控制 reduce 端缓冲大小以避免 OOM

在 Shuffle 过程，reduce 端 task 并不是等到 map 端 task 将其数据全部写入磁盘后再去拉取，而是 map 端写一点数据，reduce 端 task 就会拉取一小部分数据，然后立即进行后面的聚合、算子函数的使用等操作。

reduce 端 task 能够拉取多少数据，由 reduce 拉取数据的缓冲区 buffer 来决定，因为拉取过来的数据都是先放在 buffer 中，然后再进行后续的处理，buffer 的默认大小为 48MB。

reduce 端 task 会一边拉取一边计算，不一定每次都会拉满 48MB 的数据，可能大多数时候拉取一部分数据就处理掉了。

虽然说增大 reduce 端缓冲区大小可以减少拉取次数，提升 Shuffle 性能，但是有时 map 端的数据量非常大，写出的速度非常快，此时 reduce 端的所有 task 在拉取的时候，有可能全部达到自己缓冲的最大极限值，即 48MB，此时，再加上 reduce 端执行的聚合函数的代码，可能会创建大量的对象，这可难会导致内存溢出，即 OOM。

如果一旦出现 reduce 端内存溢出的问题，我们可以考虑减小 reduce 端拉取数据缓冲区的大小，例如减少为 12MB。

在实际生产环境中是出现过这种问题的，这是典型的以性能换执行的原理。reduce 端拉取数据的缓冲区减小，不容易导致 OOM，但是相应的，reduce 端的拉取次数增加，造成更多的网络传输开销，造成性能的下降。

注意，要保证任务能够运行，再考虑性能的优化。

### 9.2 故障排除二：JVM GC 导致的 shuffle 文件拉取失败

在 Spark 作业中，有时会出现 shuffle file not found 的错误，这是非常常见的一个报错，有时出现这种错误以后，选择重新执行一遍，就不再报出这种错误。

出现上述问题可能的原因是 Shuffle 操作中，后面 stage 的 task 想要去上一个 stage 的 task 所在的 Executor 拉取数据，结果对方正在执行 GC，执行 GC 会导致 Executor 内所有的工作现场全部停止，比如 BlockManager、基于 netty 的网络通信等，这就会导致后面的 task 拉取数据拉取了半天都没有拉取到，就会报出 shuffle file not found 的错误，而第二次再次执行就不会再出现这种错误。

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：[尚硅谷官网](#)

可以通过调整 `reduce` 端拉取数据重试次数和 `reduce` 端拉取数据时间间隔这两个参数来对 `Shuffle` 性能进行调整，增大参数值，使得 `reduce` 端拉取数据的重试次数增加，并且每次失败后等待的时间间隔加长。

```
val conf = new SparkConf()
    .set("spark.shuffle.io.maxRetries", "60")
    .set("spark.shuffle.io.retryWait", "60s")
```

### 9.3 故障排除三：解决各种序列化导致的报错

当 `Spark` 作业在运行过程中报错，而且报错信息中含有 `Serializable` 等类似词汇，那么可能是序列化问题导致的报错。

序列化问题要注意以下三点：

- 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`。

### 9.4 故障排除四：解决算子函数返回 `NULL` 导致的问题

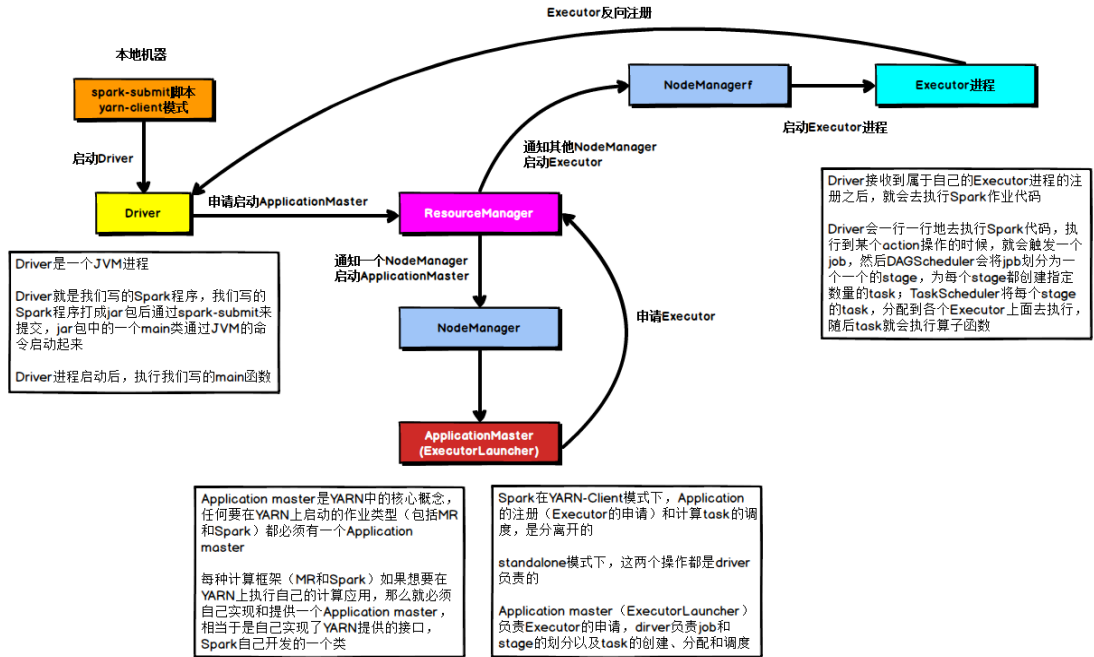
在一些算子函数里，需要我们有一个返回值，但是在一些情况下我们不希望有返回值，此时我们如果直接返回 `NULL`，会报错，例如 `Scala.Math(NULL)` 异常。

如果你遇到某些情况，不希望有返回值，那么可以通过下述方式解决：

- 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。

### 9.5 故障排除五：解决 `YARN-CLIENT` 模式导致的网卡流量激增问题

`YARN-client` 模式的运行原理如下图所示：



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。

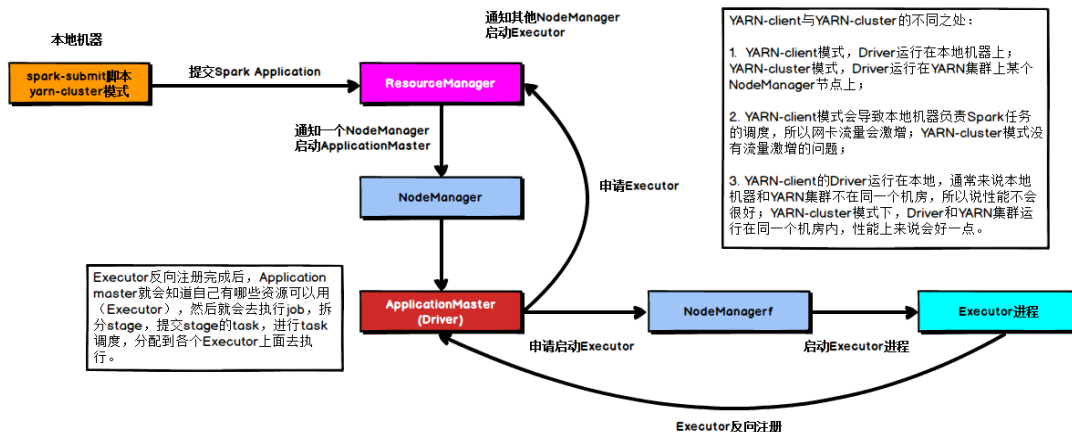
假设有 100 个 Executor，1000 个 task，那么每个 Executor 分配到 10 个 task，之后，Driver 要频繁地跟 Executor 上运行的 1000 个 task 进行通信，通信数据非常多，并且通信品类特别高。这就导致有可能在 Spark 任务运行过程中，由于频繁大量的网络通讯，本地机器的网卡流量会激增。

注意，YARN-client 模式只会在测试环境中使用，而之所以使用 YARN-client 模式，是由于可以看到详细全面的 log 信息，通过查看 log，可以锁定程序中存在的问题，避免在生产环境下发生故障。

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，如果 YARN-cluster 模式下存在网络通信的问题，需要运维团队进行解决。

## 9.6 故障排除六：解决 YARN-CLOUD 模式的 JVM 栈内存溢出无法执行问题

YARN-cluster 模式的运行原理如下图所示：



当 Spark 作业中包含 SparkSQL 的内容时，可能会碰到 YARN-client 模式下可以运行，但是 YARN-cluster 模式下无法提交运行（报出 OOM 错误）的情况。

YARN-client 模式下，Driver 是运行在本地机器上的，Spark 使用的 JVM 的 PermGen 的配置，是本地机器上的 `spark-class` 文件，JVM 永久代的大小是 128MB，这个是没有问题的，但是在 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。

SparkSQL 的内部要进行很复杂的 SQL 的语义解析、语法树转换等等，非常复杂，如果 sql 语句本身就非常复杂，那么很有可能会导致性能的损耗和内存的占用，特别是对 PermGen 的占用会比较大。

所以，此时如果 PermGen 的占用好过了 82MB，但是又小于 128MB，就会出现 YARN-client 模式下可以运行，YARN-cluster 模式下无法运行的情况。

解决上述问题的方法时增加 PermGen 的容量，需要在 `spark-submit` 脚本中对相关参数进行设置，设置方法如代码清单所示。

```
--conf spark.driver.extraJavaOptions="-XX:PermSize=128M -XX:MaxPermSize=256M"
```

通过上述方法就设置了 Driver 永久代的大小，默认为 128MB，最大 256MB，这样就可以避免上面所说的问题。

## 9.7 故障排除七：解决 SparkSQL 导致的 JVM 栈内存溢出

当 SparkSQL 的 sql 语句有成百上千的 `or` 关键字时，就可能会出现 Driver 端的 JVM 栈内存溢出。  
**或者子查询数量过多时**

JVM 栈内存溢出基本上就是由于调用的方法层级过多，产生了大量的，非常深的，超出了 JVM 栈深度限制的递归。（我们猜测 SparkSQL 有大量 `or` 语句的时候，在解析 SQL 时，例如转换为语法树或者进行执行计划的生成的时候，对于 `or` 的处理是递归，`or` 非常多时，



会发生大量的递归)

此时, 建议将一条 sql 语句拆分为多条 sql 语句来执行, 每条 sql 语句尽量保证 100 个以内的子句。根据实际的生产环境试验, 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。

## 9.8 故障排除八：持久化与 checkpoint 的使用

Spark 持久化在大部分情况下是没有问题的, 但是有时数据可能会丢失, 如果数据一旦丢失, 就需要对丢失的数据重新进行计算, 计算完后再缓存和使用, 为了避免数据的丢失, 可以选择对这个 RDD 进行 checkpoint, 也就是将数据持久化一份到容错的文件系统上 (比如 HDFS)。

一个 RDD 缓存并 checkpoint 后, 如果一旦发现缓存丢失, 就会优先查看 checkpoint 数据存不存在, 如果有, 就会使用 checkpoint 数据, 而不用重新计算。也即是说, checkpoint 可以视为 cache 的保障机制, 如果 cache 失败, 就使用 checkpoint 的数据。

使用 checkpoint 的优点在于提高了 Spark 作业的可靠性, 一旦缓存出现问题, 不需要重新计算数据, 缺点在于, checkpoint 时需要将数据写入 HDFS 等文件系统, 对性能消耗较大。

## 9.9 故障排除九：内存泄漏排查

内存泄露是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放, 造成系统内存的浪费, 导致程序运行速度减慢, 甚至系统崩溃等严重后果。

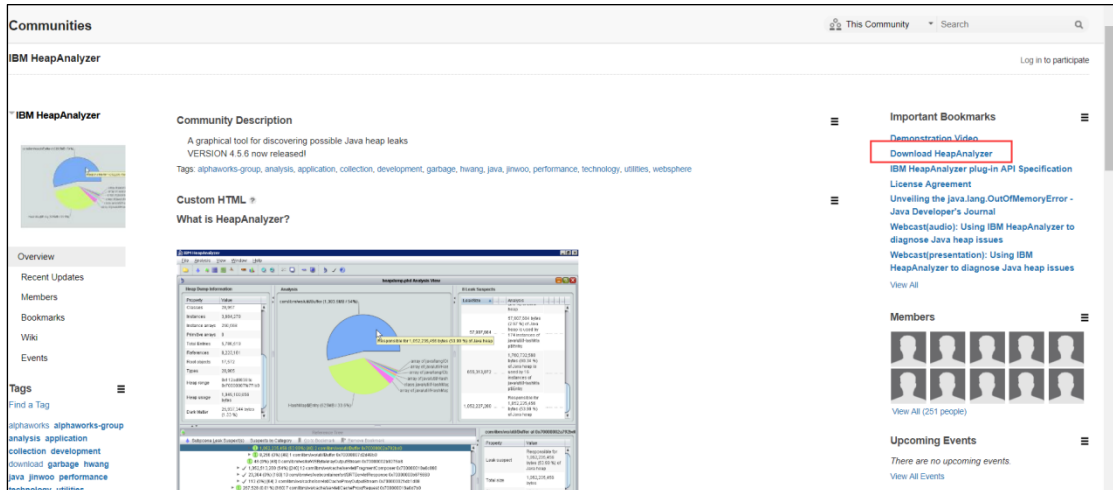
在 Spark Streaming 中往往会因为开发者代码未正确编写导致无法回收或释放对象, 造成 Spark Streaming 内存泄露越跑越慢甚至崩溃的结果。那么排查内存泄露需要一些第三方的工具。

### 3.9.1 IBM HeapAnalyzer

官网地址


<https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=4544bafec7a2-455f-9d43-eb866ea60091>






点击下载 内存泄露分析工具

下载下来是一个 jar 包

 ha456.jar	2019/6/6 9:57	Executable Jar File	5,626 KB
---	---------------	---------------------	----------

那么需要编写 bat 批处理来运行

创建 run.bat

 run.bat	2019/6/6 9:57	Windows 批处理...	1 KB
---	---------------	----------------	------

编辑

```
title ibm-heap-analyzer

path=%PATH%;%C:\JAVA\jdk1.8.0_51\bin

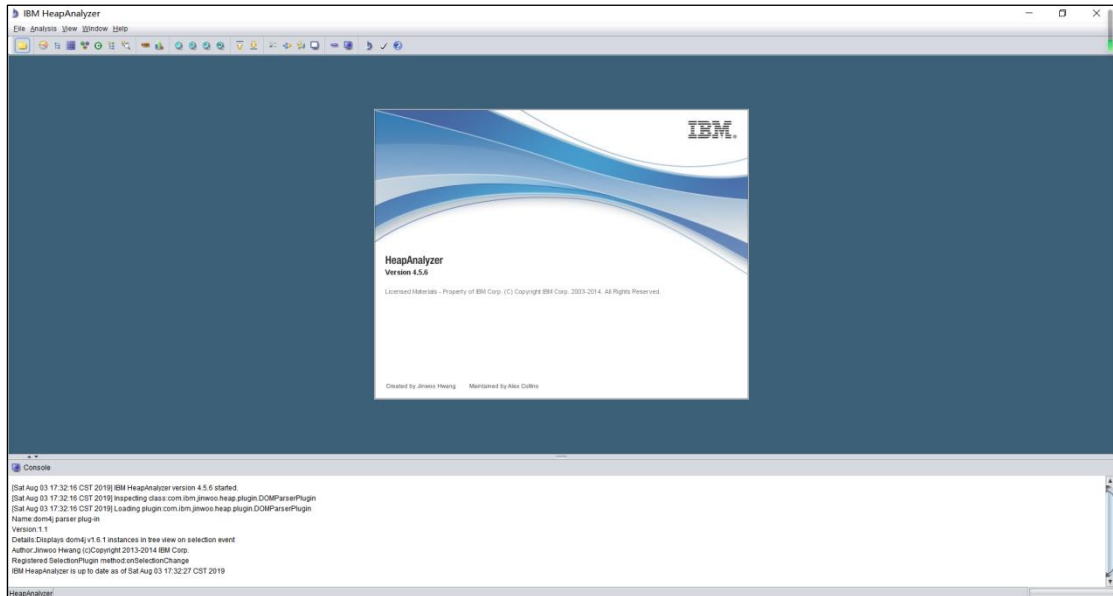
E:

cd E:\IBM heapAnalyzer\IBM_DUMP_wjfx

java.exe -Xms1048M -Xmx4096M -jar ha456.jar
```

路径需要改成自己当前路径

点击 run.bat 运行



运行成功

### 3.9.2 模拟内存泄露场景

内存泄露的原因往往是因为对象无法释放或被回收造成，那么在本项目中就模拟此场景。

```
22  * 知识点掌握度实时统计
23  */
24  object QzPointStreaming {
25
26      private val groupid = "qz_point_group"
27
28      val map = new mutable.HashMap[String, LearnModel]()
29
30      def main(args: Array[String]): Unit = {
31          val conf = new SparkConf().setAppName(this.getClass.getSimpleName)
32              .setMaster("local[*]")
33              .set("spark.streaming.kafka.maxRatePerPartition", "10")
34              .set("spark.streaming.stopGracefullyOnShutdown", "true")
35
36          val ssc = new StreamingContext(conf, Seconds(3))
37          val topics = Array("qz_log")
38          val kafkaMap: Map[String, Object] = Map[String, Object]({
```

```

})
//处理完 业务逻辑后 手动提交offset维护到本地 mysql中
stream.foreachRDD(rdd => {
    val sqlProxy = new SqlProxy()
    val client = DataSourceUtil.getConnection
    try {
        val offsetRanges: Array[OffsetRange] = rdd.asInstanceOf[HasOffsetRanges].offsetRanges
        for (or <- offsetRanges) {
            sqlProxy.executeUpdate(client, sql = "replace into `offset_manager` (groupid,topic,`partition`,untilOffset) values(?,?,?,?)",
                Array(groupid, or.topic, or.partition.toString, or.untilOffset))
        }
        for (i <- 0 until 100000) {
            new LearnModel( userid = 1, cwareId = 1, videoid = 1, chapterId = 1, edutypeId = 1, subjectId = 1, sourceType = "", speed = 2, ts = 1)
        }
    } catch {
        case e: Exception => e.printStackTrace()
    } finally {
        sqlProxy.shutdown(client)
    }
})
ssc.start()

```

如上图所示，在计算学员知识点正确率与掌握度代码中，在最后提交 offset 提交偏移量后，循环往 map 里添加 LearnMode 对象，使每处理一批数据就往 map 里添加 100000 个 LearnMode 对象，使堆内存撑满。

## 3.9.3 查找 driver 进程

在集群上提交 spark streaming 任务

```
ps -ef |grep com.atguigu.qzpoint.streaming.QzPointStreaming
```

通过此命令查找到 driver 进程号

```

[atguigu@hadoop104 ~]$ ps -ef |grep com.atguigu.qzpoint.streaming.QzPointStreaming
atguigu 6860 2868 96 10:11 pts/0 00:03:59 /opt/module/jdk1.8.0_144/bin/java -cp /opt/module/spark-2.1.1-bin-hadoop2.7/conf:/opt/module/spark-2.1.1-bin-hadoop2.7/jars/*:/opt/module/hadoop-2.7.2/etc/hadoop:/opt/module/hadoop-2.7.2/etc/hadoop/ -Xmx1g org.apache.spark.deploy.SparkSubmit --master yarn --deploy-mode client --conf spark.driver.memory=1g --class com.atguigu.qzpoint.streaming.QzPointStreaming --num-executors 2 --executor-cores 2 --executor-memory 2g com.atguigu.sparkstreaming-1.0-SNAPSHOT-jar-with-dependencies.jar
atguigu 6889 4336 0 10:17 pts/1 00:00:00 grep com.atguigu.qzpoint.streaming.QzPointStreaming

```

进程号为 6860

Event Timeline

Active Jobs (1)

Job ID	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	Streaming job from [output operation 0, batch time 14:59:57] foreachPartition at QzPointStreaming.scala:90 (kill)	2019/08/05 15:03:31	32 s	1/2	10/20

Completed Jobs (39)

Job ID	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8	Streaming job from [output operation 0, batch time 14:59:54] foreachPartition at QzPointStreaming.scala:90	2019/08/05 15:03:19	3 s	2/2	20/20
7	Streaming job from [output operation 0, batch time 14:59:51] foreachPartition at QzPointStreaming.scala:90	2019/08/05 15:03:14	3 s	2/2	20/20
6	Streaming job from [output operation 0, batch time 14:59:48] foreachPartition at QzPointStreaming.scala:90	2019/08/05 15:03:11	0.8 s	2/2	20/20
5	Streaming job from [output operation 0, batch time 14:59:45] foreachPartition at QzPointStreaming.scala:90	2019/08/05 15:03:09	0.3 s	2/2	20/20
4	Streaming job from [output operation 0, batch time 14:59:42] foreachPartition at QzPointStreaming.scala:90	2019/08/05 15:03:07	0.3 s	2/2	20/20

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input
Active(1)	2	6.2 KB / 384.1 MB	0.0 B	8	0	0	540	540	13 min (1.4 min)	0.0 B
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B
Total(1)	2	6.2 KB / 384.1 MB	0.0 B	8	0	0	540	540	13 min (1.4 min)	0.0 B

Executors

Show

20

 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input
driver	192.168.1.104:51790	Active	2	6.2 KB / 384.1 MB	0.0 B	8	0	0	540	540	13 min (1.4 min)	0.0 B

Showing 1 to 1 of 1 entries

通过 Spark Ui 发现该 Spark Streaming task 任务发生长时间卡住现象，GC 出现异常。疑似发生内存泄露

### 3.9.4 JMAP 命令

使用 jmap -heap pid 命令查看 6860 进程，内存使用情况。

```
jmap -heap 6860
```

## 3.10 故障排除十：频繁 GC 问题

#### 1、打印 GC 详情

统计一下 GC 启动的频率和 GC 使用的总时间，在 spark-submit 提交的时候设置参数

```
--conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps"
```

如果出现了多次 Full GC，首先考虑的是可能配置的 Executor 内存较低，这个时候需要增加 Executor Memory 来调节。

2、如果一个任务结束前，Full GC 执行多次，说明老年代空间被占满了，那么有可能是没有分配足够的内存。

- 1.调整 executor 的内存，配置参数 executor-memory
- 2.调整老年代所占比例：配置-XX:NewRatio 的比例值
- 3.降低 spark.memory.storageFraction 减少用于缓存的空间

3、如果有太多 Minor GC，但是 Full GC 不多，可以给 Eden 分配更多的内存。

- 1.比如 Eden 代的内存需求量为 E，可以设置 Young 代的内存为-Xmn=4/3\*E，设置该值也会导致 Survivor 区域扩张
- 2.调整 Eden 在年轻代所占的比例，配置-XX:SurvivorRatio 的比例值

4、调整垃圾回收器，通常使用 G1GC，即配置-XX:+UseG1GC。当 Executor 的堆空间比较大时，可以提升 G1 region size(-XX:G1HeapRegionSize)，在提交参数指定：

```
--conf "spark.executor.extraJavaOptions=-XX:+UseG1GC -XX:G1HeapRegionSize=16M -XX:+PrintGCDetails -XX:+PrintGCTimeStamps"
```

更多 Java -大数据 -前端 -python 人工智能资料下载，可百度访问：尚硅谷官网