Spark 数据读取冷启动优化分析

breeze_lsw 浪尖聊大数据 3/22

> 有时候会发现即使是读取少量的数据，启动延时可能也非常大，针对该现象进行分析，并提供一些解决思路。

背景

Spark 一次查询过程可以简单抽象为 planning 阶段和 execution 阶段，在一个新的 Spark Session 中第一次查询某数据的过程称为冷启动，在这种情况下 planning 的耗时可能会比 execution 更长。
Spark 读取数据冷启动时，会从文件系统中获取文件的一些元数据信息（location,size,etc.）用于优化，如果一个目录下的文件过多，就会比较耗时(可能达到数十分钟)，该逻辑在 InMemoryFieIndex 中实现。
后续再次多次查询则会在 FileStatusCache 中进行查询，planning 阶段性能也就大幅提升了，下文将探讨 planning 阶段如何加载元数据以及可能的一些优化点。

InMemoryFileIndex

before spark 2.1

spark 2.1 版本前，spark 直接从文件系统中查询数据的元数据并将其缓存到内存中，元数据包括一个 partition 的列表和文件的一些统计信息（路径，文件大小，是否为目录，备份数，块大小，定义时间，访问时间，数据块位置信息）。一旦数据缓存后，在后续的查询中，表的 partition 就可以在内存中进行下推，得以快速的查询。
将元数据缓存在内存中虽然提供了很好的性能，但也存在2个缺点：在 spark 加载所有表分区的元数据之前，会阻塞查询。对于大型分区表，递归的扫描文件系统以发现初始查询文件的元数据可能会花费数分钟，特别是当数据存储在云端。其次，表的所有元数据都需要放入内存中，增加了内存压力。

after spark 2.1

spark 2.1 针对上述缺点进行了优化，可参考 SPARK-17861
- 将表分区元数据信息缓存到 catalog 中，例如 (hive metastore)，因此可以在 PruneFileSourcePartitions 规则中提前进行分区发现，catalyse optimeizer 会在逻辑计划中对分区进行修剪，避免读取到不需要的分区文件信息。

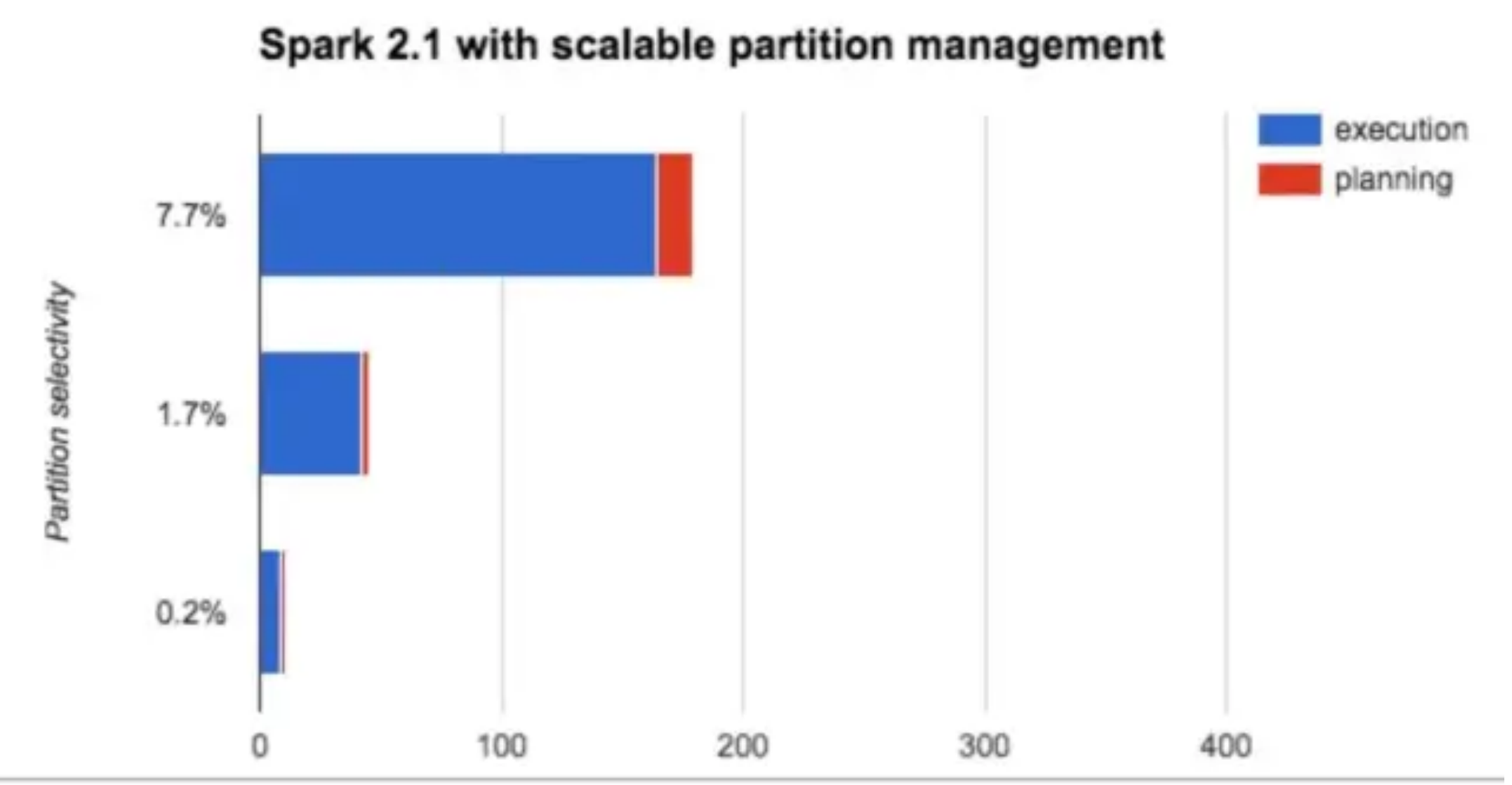
- 文件统计现在可以在计划期间内增量的，部分的缓存，而不是全部预先加载。Spark需要知道文件的大小以便在执行物理计划时将它们划分为读取任务。通过共享一个固定大小的250MB缓存(可配置)，而不是将所有表文件统计信息缓存到内存中，在减少内存错误风险的情况下显著加快重复查询的速度。

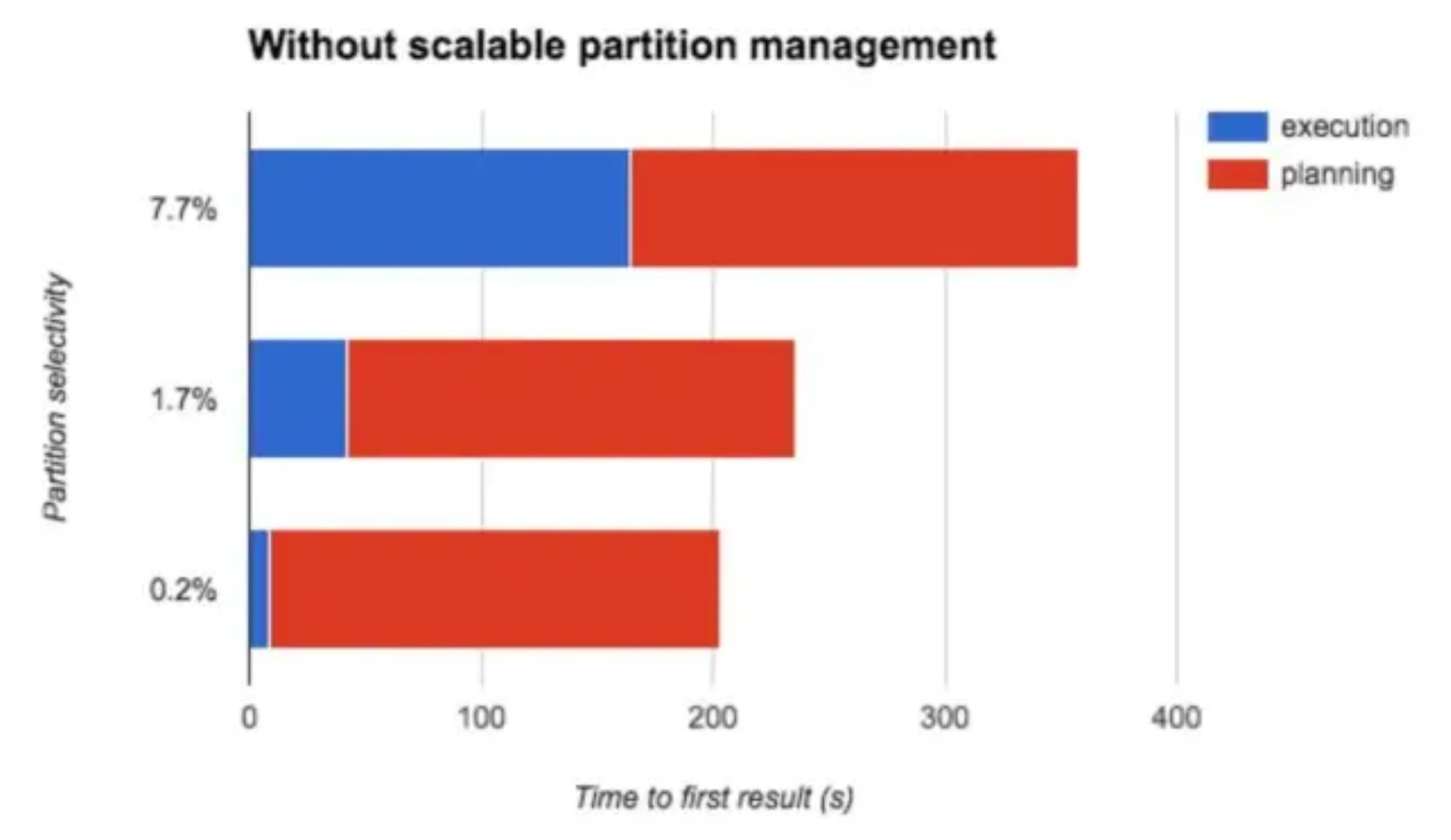
旧表可以使用 `MSCK REPAIR TABLE` 命令进行转化，查看是否生效，如果 `Partition Provider` 为 `Catalog` 则表示会从 catalog 中获取分区信息

```
sql("describe formatted test_table")
.filter("col_name like '%Partition Provider%'").show
+-------------------+---------+-------+
|           col_name|data_type|comment|
+-------------------+---------+-------+
|Partition Provider:|  Catalog|       |
+-------------------+---------+-------+
```

性能对比

出自官方blog，通过读取一张表不同的分区数，观察任务 execution time 和 planning time，在spark2.1之前 planning 阶段的耗时是相同的，意味着读取一个分区也需要扫描全表的 file status。





优化 HDFS 获取 File 元数据性能

虽然优化了避免加载过多元数据的问题，但是单个分区下文件过多导致读取文件元数据缓慢的问题并没有解决。

在 SPARK-27801 中(将在 spark3.0 release)，对一个目录下多文件的场景进行了优化，性能有大幅度的提升。

使用 `DistributedFileSystem.listLocatedStatus` 代替了 `fs.listStatus + getFileBlockLocations`的方式

- `listLocatedStatus` 向 namenode 发起一次请求获得 `file status` 和 `file block location` 信息

- `listStatus` 获取一系列的 file status 后，还要根据 file status 循环向 namenode 发起请求获得 `file block location`信息

listLocatedStatus

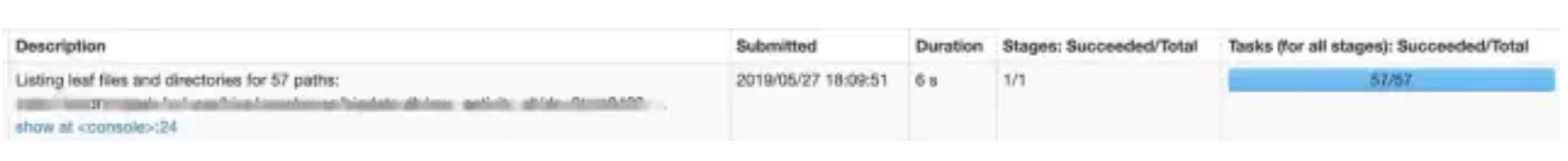
```
// 对 namenode 只发起一次 listLocatedStatus 请求，在方法内部获得每个文件 block
val statuses = fs.listLocatedStatus(path)
new Iterator[LocatedFileStatus]() {
  def next(): LocatedFileStatus = remoteIter.next
  def hasNext(): Boolean = remoteIter.hasNext
}.toArray
statuses.flatMap{
  Some(f)
}
```

fs.listStatus + getFileBlockLocations (只展示核心代码)

```
val statuses = fs.listStatus(path)
statuses.flatMap{
  val locations = fs.getFileBlockLocations(f, 0, f.getLen).map { loc =>
    if (loc.getClass == classOf[BlockLocation]) {
        loc
    } else {
        new BlockLocation(loc.getNames, loc.getHosts, loc.getOffset, lo
    }
    }
  val lfs = new LocatedFileStatus(f.getLen, f.isDirectory, f.getReplica
             f.getModificationTime, 0, null, null, null, null, f.getPath
  if (f.isSymlink) {
    lfs.setSymlink(f.getSymlink)
  }
  Some(lfs)
}
```

性能对比

实测一个57个分区，每个分区1445个文件的任务，性能提升6倍左右



打入 SPARK-27801 前

打入 SPARK-27801 后

文件元数据读取方式及元数据缓存管理
1. 读取数据时会先判断分区的数量，如果分区数量小于等于`spark.sql.sources.parallelPartitionDiscovery.threshold`（默认32），则使用 driver 循环读取文件元数据，如果分区数量大于该值，则会启动一个 spark job，分布式的处理元数据信息(每个分区下的文件使用一个task进行处理)
2. 分区数量很多意味着 Listing leaf files task 的任务会很多，分区里的文件数量多意味着每个 task 的负载高，使用 FileStatusCache 缓存文件状态，默认的缓存 `spark.sql.hive.filesourcePartitionFileCacheSize` 为 250MB

Tip
Listing leaf files task 的数量计算公式为

```
val numParallelism = Math.min(paths.size, parallelPartitionDiscoveryPara
```

其中，`paths.size` 为需要读取的分区数量，`parallelPartitionDiscoveryParallelism` 由参数 `spark.sql.sources.parallelPartitionDiscovery.parallelism` 控制，默认为10000，目的是防止 task 过多，但从生产任务上观察发现大多数 get status task 完成的时间都是毫秒级，可以考虑把这个值调低，减少任务启动关闭的开销，或者直接修改源码将 `paths.size` 按一定比例调低，例如 `paths.size/2`

控制 task 数量之前

控制 task 数量之后

结语

spark 查询冷启动（获取文件元数据性能）对比前几个版本已经有非常大提升，降低了查询的延时
- SPARK-17861 在物理计划中进行了优化，通过将分区信息存入 catalog，避免了读取时加载全量表的文件信息
- SPARK-27801 优化读取 hdfs 文件元数据的方式，之前 getFileBlockLocations 的方式是串行的，在文件数量很多的情况下速度会很慢，同时用 listLocatedStatus 的方式减少了客户端对 namenode 的直接调用，例如需要读取的数据为3个分区，每个分区 10k 个文件，之前客户端需要访问 namenode 的次数为30k，现在为3次
- 打入最新的 patch 和 优化 task 数量后，随机找的一个生产任务 Listing Leaf files job 时间从数十秒减少到1S以内，不过有时候依旧存在毛刺，这与 namenode 和机器的负载程度有关

一些思考，是否可以考虑用 `Redis` 替换 `FileStatusCache`，在数据写入的时候更新 Redis 中的 file status 信息，这样就相当于所有的 spark 应用共享了 `FileStatusCache`，减少了内存使用的同时也不再有读数据冷启动的问题了。

参考

scalable-partition-handling-for-cloud-native-architecture-in-apache-spark-2-1