



下载APP



19 | 网络视角：如何有效降低网络开销？

2021-04-26 吴磊

Spark性能调优实战

[进入课程 >](#)



讲述：吴磊

时长 15:40 大小 14.36M



你好，我是吴磊。

在平衡不同硬件资源的时候，相比 CPU、内存、磁盘，网络开销无疑是最拖后腿的那一个，这一点在处理延迟上表现得非常明显。

下图就是不同硬件资源的处理延迟对比结果，我们可以看到最小的处理单位是纳秒。你可能对纳秒没什么概念，所以为了方便对比，我把纳秒等比放大到秒。这样，其他硬件资源的处理延迟也会跟着放大。最后一对比我们会发现，网络延迟是以天为单位的！



硬件资源	处理延迟	时间单位换算
CPU L1 缓存	0.5纳秒	0.5秒
CPU L2 缓存	7纳秒	7秒
内存	100纳秒	100秒
同DC网络传输	0.5毫秒	5.8天



不同硬件资源处理延迟对比

因此，要想维持硬件资源之间的平衡，尽可能地降低网络开销是我们在性能调优中必须要做的。今天这一讲，我就按照数据进入系统的时间顺序，也就是数据读取、数据处理和数据传输的顺序，带你去分析和总结数据生命周期的不同阶段有效降低网络开销的方法。

数据读写

对于绝大多数应用来说，第一步操作都是从分布式文件系统读取数据源。Spark 支持的数据源种类非常丰富，涉及的存储格式和存储系统可以说是五花八门。

存储格式	存储系统
Avro	Amazon
CSV	Redshift
Hive表	Amazon S3
JSON	Azure Data
LZO	Lake
Parquet	Cassandra
ORC	HDFS
XML	MongoDB
...	Redis
	Snowflake
	...



存储格式与存储系统

这么多存储格式和外部存储系统交叉在一起又会有无数种组合，并且每一种组合都有它的应用场景。那么，我们该怎么判断网络开销会出现在哪些场景下呢？其实，**不管是什么文件格式，也不管是哪种存储系统，访问数据源是否会引入网络开销，取决于任务与数据的本地性关系，也就是任务的本地性级别，它一共有 4 种：**

- PROCESS_LOCAL：任务与数据同在一个 JVM 进程中
- NODE_LOCAL：任务与数据同在一个计算节点，数据可能在磁盘上或是另一个 JVM 进程中
- RACK_LOCAL：任务与数据不在同一节点，但在同一个物理机架上
- ANY：任务与数据是跨机架、甚至是跨 DC（Data Center，数据中心）的关系

根据定义我们很容易判断出，不同本地性级别下的计算任务是否会引入磁盘或网络开销，结果如下表所示。从表格中我们不难发现，从 PROCESS_LOCAL 到 ANY，数据访问效率是逐级变差的。在读取数据源阶段，数据还未加载到内存，任务没有办法调度到 PROCESS_LOCAL 级别。因此，这个阶段我们能够调度的最佳级别是 NODE_LOCAL。

本地性级别	内存计算	磁盘开销	同机架网络开销	跨机架网络开销
PROCESS_LOCAL	R			
NODE_LOCAL		R		
RACK_LOCAL			R	
ANY				R



不同本地性级别与磁盘、网络开销的关系

根据 NODE_LOCAL 的定义，在这个级别下，调度的目标节点至少在磁盘上存有 Spark 计算任务所需的数据分片。这也就意味着，在集群部署上，Spark 集群与外部存储系统在物理上是紧紧耦合在一起的。相反，如果 Spark 集群与存储集群在物理上是分开的，那么任务的本地性级别只能退化到 RACK_LOCAL，甚至是 ANY，来通过网络获取所需的数据分片。

因此，对于 Spark 加 HDFS 和 Spark 加 MongoDB 来说，是否会引入网络开销完全取决于它们的部署模式。物理上紧耦合，在 NODE_LOCAL 级别下，Spark 用磁盘 I/O 替代网络开销获取数据；物理上分离，网络开销就无法避免。

除此之外，物理上的隔离与否同样会影响数据的写入效率。当数据处理完毕，需要将处理结果落盘到外部存储的时候，紧耦合模式下的数据写入会把数据分片落盘到本地节点，避免网络开销。

值得一提的是，在企业的私有化 DC 中更容易定制化集群的部署方式，大家通常采用紧耦合的方式来提升数据访问效率。但是在公有云环境中，计算集群在物理上往往和存储系统隔离，因此数据源的读取只能走网络。

通过上面的分析，对于数据读写占比较高的业务场景，我们就可以通过在集群的部署模式上做规划，从而在最开始部署 Spark 集群的时候就提前做好准备。

数据处理

数据读取完成后，就进入数据处理环节了。那在数据处理的过程中，都有哪些技巧能够帮助减少网络开销呢？

能省则省

说起数据处理中的网络开销，我猜你最先想到的操作就是 Shuffle。Shuffle 作为大多数计算场景的“性能瓶颈担当”，确实是网络开销的罪魁祸首。根据“能省则省”的开发原则，我们自然要想尽办法去避免 Shuffle。在数据关联的场景中，省去 Shuffle 最好的办法，就是把 Shuffle Joins 转化为 Broadcast Joins。关于这方面的调优技巧，我们在广播变量那几讲有过详细的讲解，你可以翻回去看一看。尽管广播变量的创建过程也会引入网络传输，但是，两害相权取其轻，相比 Shuffle 的网络开销，广播变量的开销算是小巫见大巫了。

遵循“能省则省”的原则，把 Shuffle 消除掉自然是最好的。如果实在没法避免 Shuffle，我们要尽可能地在计算中多使用 Map 端聚合，去减少需要在网络中分发的数据量。这方面的典型做法就是用 reduceByKey、aggregateByKey 替换 groupByKey，不过在 RDD API 使用频率越来越低的当下，这个调优技巧实际上早就名存实亡了。但是，Map 端聚合的思想并不过时。为什么这么说呢？下面，我通过一个小例子来你详细讲一讲。

在绝大多数 2C (To Consumer) 的业务场景中，我们都需要刻画用户画像。我们的小例子就是“用户画像”中的一环，：给定用户表，按照用户群组统计兴趣列表，要求兴趣列表内容唯一，也就是不存在重复的兴趣项，用户表的 Schema 如下表所示。

字段	groupId	userId	age	gender	...	interestList
含义	用户群组ID	用户ID	年龄	性别	其他用户属性	兴趣列表
类型	Int	Int	Int	String	...	Array[String]
示例	123	27	32	Male	...	["足球", "围棋", ... "摄影"]



用户表Schema示例

要获取群组兴趣列表，我们应该先按照 groupId 分组，收集群组内所有用户的兴趣列表，然后再把列表中的兴趣项展平，最后去重得到内容唯一的兴趣列表。应该说思路还是蛮简单的，我们先来看第一版实现代码。

复制代码

```
1 val filePath: String = _
2 val df = spark.read.parquet(filePath)
3 df.groupBy("groupId")
4   .agg(array_distinct(flatten(collect_list(col("interestList")))))
```

这版实现分别用 collect_list、flatten 和 array_distinct，来做兴趣列表的收集、展平和去重操作，它完全符合业务逻辑。不过，见到“收集”类的操作，比如 groupByKey，以及这里的 collect_list，我们应该本能地提高警惕。因为这类操作会把最细粒度的全量数据在全网分发。相比其他算子，这类算子引入的网络开销最大。

那我们是不是可以把它们提前到 Map 端，从而减少 Shuffle 中需要分发的数据量呢？当然可以。比如，对于案例中的收集操作，我们可以在刚开始收集兴趣列表的时候就在 Map 端做一次去重，然后去查找 DataFrame 开发 API，看看有没有与 collect_list 对应的 Map 端聚合算子。

因此，在数据处理环节，我们要遵循“能省则省”的开发原则，主动削减计算过程中的网络开销。对于数据关联场景，我们要尽可能地把 Shuffle Joins 转化为 Broadcast Joins 来消除 Shuffle。如果确实没法避免 Shuffle，我们可以在计算中多使用 Map 端聚合，减少需要在网络中分发的数据量。

除了 Shuffle 之外，还有个操作也会让数据在网络中分发，这个操作很隐蔽，我们经常注意不到它，它就是多副本的 RDD 缓存。


比如说，在实时流处理这样的场景下，对于系统的高可用性，应用的要求比较高，因此你可能会用 “_2” 甚至是 “_3” 的存储模式，在内存和磁盘中缓存多份数据拷贝。当数据副本数大于 1 的时候，本地数据分片就会通过网络被拷贝到其他节点，从而产生网络开销。虽然这看上去只是存储模式字符串的一个微小改动，但在运行时，它会带来很多意想不到的开销。因此，如果你的应用对高可用特性没有严格要求，我建议你尽量不要滥用多副本的 RDD 缓存，

数据传输

最后就到了数据传输的环节。我们知道，在落盘或是在网络传输之前，数据都是需要先进行序列化的。在 Spark 中，有两种序列化器供开发者选择，分别是 Java serializer 和 Kryo Serializer。Spark 官方和网上的技术博客都会推荐你使用 Kryo Serializer 来提高效率，通常来说，Kryo Serializer 相比 Java serializer，在处理效率和存储效率两个方面都会胜出数倍。因此，在数据分发之前，使用 Kryo Serializer 对其序列化会进一步降低网络开销。

不过，经常有同学向我抱怨：“为什么我用了 Kryo Serializer，序列化之后的数据尺寸反而比 Java serializer 的更大呢？” 注意啦，这里我要提醒你：**对于一些自定义的数据结构来说，如果你没有明确把这些类型向 Kryo Serializer 注册的话，虽然它依然会帮你做序列化的工作，但它序列化的每一条数据记录都会带一个类名字，这个类名字是通过反射机制得到的，会非常长。在上亿的样本中，存储开销自然相当可观。**

那该怎么向 Kryo Serializer 注册自定义类型呢？其实非常简单，**我们只需要在 SparkConf 之上调用 registerKryoClasses 方法就好了**，代码示例如下所示。

 复制代码

```
1 //向Kryo Serializer注册类型
2 val conf = new SparkConf().setMaster("").setAppName("")
3 conf.registerKryoClasses(Array(
4   classOf[Array[String]],
5   classOf[HashMap[String, String]],
6   classOf[MyClass]
7 ))
```

另外，与 Kryo Serializer 有关的配置项，我也把它们汇总到了下面的表格中，方便你随时查找。其中，spark.serializer 可以明确指定 Spark 采用 Kryo Serializer 序列化器。而 spark.kryo.registrationRequired 就比较有意思了，如果我们把它设置为 True，当 Kryo Serializer 遇到未曾注册过的自定义类型的时候，它就不会再帮你做序列化的工作，而是抛出异常，并且中断任务执行。这么做的好处在于，在开发和调试阶段，它能帮我们捕捉那些忘记注册的类型。

配置项	含义	默认值	推荐设置
spark.serializer	指定使用哪一种序列化器	JavaSerializer	KryoSerializer
spark.kryo.registrationRequired	是否强制注册自定义类型	FALSE	TRUE



与Kryo Serializer有关的配置项

为了方便你理解，我们不妨把 Java serializer 和 Kryo Serializer 比作是两个不同的搬家公司。

Java Serializer 是老牌企业，市场占有率高，而且因为用户体验很好，所以非常受欢迎。只要你出具家庭住址，Java Serializer 会派专人到家里帮你打包，你并不需要告诉他们家里都有哪些物件，他们对不同种类的物品有一套自己的打包标准，可以帮你省去很多麻烦。不过，Java Serializer 那套打包标准过于刻板，不仅打包速度慢，封装出来的包裹往往个头超大、占地儿，你必须租用最大号的货车才能把家里所有的物品都装下。

Kryo Serializer 属于市场新贵，在打包速度和包裹尺寸方面都远胜 Java Serializer。Kryo Serializer 会以最紧凑的方式打包，一寸空间也不浪费，因此，所有包裹用一辆小货车就能装下。但是，订阅 Kryo Serializer 的托管服务之前，用户需要提供详尽的物品明细表，因此，很多用户都嫌麻烦，Kryo Serializer 市场占有率也就一直上不去。

好啦，现在你是不是对 Java Serializer 和 Kryo Serializer 有了更深的理解了？由此可见，如果你想要不遗余力去削减数据传输过程中的网络开销，就可以尝试使用 Kryo Serializer 来做数据的序列化。相反，要是你觉得开发成本才是核心痛点，那采用默认的 Java Serializer 也未尝不可。

小结

这一讲，面对数据处理不同阶段出现的网络开销，我带你总结出了有效降低它的办法。

首先，在数据读取阶段，要想获得 `NODE_LOCAL` 的本地级别，我们得让 Spark 集群与外部存储系统在物理上紧紧耦合在一起。这样，Spark 就可以用磁盘 I/O 替代网络开销获取数据，否则，本地级别就会退化到 `RACK_LOCAL` 或者 `ANY`，那网络开销就无法避免。

其次，在数据处理阶段，我们应当遵循“能省则省”的开发原则，在适当的场景用 Broadcast Joins 来避免 Shuffle 引入的网络开销。如果确实没法避免 Shuffle，我们可以在计算中多使用 Map 端聚合，减少需要在网络中分发的数据量。另外，如果应用对于高可用的要求不高，那我们应该尽量避免副本数量大于 1 的存储模式，避免副本跨节点拷贝带来的额外开销。

最后，在数据通过网络分发之前，我们可以利用 Kryo Serializer 序列化器，提升序列化字节的存储效率，从而有效降低在网络中分发的数据量，整体上减少网络开销。需要注意的，为了充分利用 Kryo Serializer 序列化器的优势，开发者需要明确注册自定义的数据类型，否则效果可能适得其反。

每日一练

1. 对于文中 Map 端聚合的示例，你知道和 `collect_list` 对应的 Map 端聚合算子是什么吗？
2. 你还能想到哪些 Map 端聚合的计算场景？
3. 对于不同的数据处理阶段，你还知道哪些降低网络开销的办法吗？

期待在留言区看到你的思考和答案，也欢迎你把这一讲转发出去，我们下一讲见！

提建议

更多课程推荐

Kafka 核心技术与实战

全面提升你的 Kafka 实战能力

胡夕

Apache Kafka Committer
老虎证券技术总监

涨价倒计时 🕒

今日订阅 **¥89**，4月29日涨价至 **¥199**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 磁盘视角：如果内存无限大，磁盘还有用武之地吗？

下一篇 20 | RDD和DataFrame：既生瑜、何生亮

精选留言 (5)

💬 写留言



aof

2021-05-05

1. org.apache.spark.sql.functions中用来collect就两个函数，一个collect_list可以有重复元素，一个collect_set元素唯一
2. 好像没怎么理解问题的意思哈哈，聚合的话无非就是计数、就和、平均值、最大值最小值这些吗
3. cache的时候，数据以序列化的形式进行缓存（比如，StorageLevel.MEMORY_ONL...
展开 ✓



Fendora范东_

2021-04-27

有点疑问

1.没有RDD缓存的情况下，是不是最好的任务级别是node_level，而且是最底下包含scan操作的tasks是这个级别？

因为:最开始数据全在磁盘，第一个stage生成的task本地性级别最好为node_level，后面的stage生成的task都是需要进行shuffle，最起码也是rack node。 ...

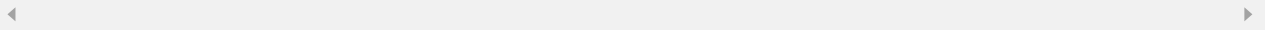
展开 ∨

作者回复: 好问题。

先来说疑问1：没错，从磁盘文件系统读数据，最好的本地性级别就是NODE_LOCAL。不过，现在有一些内存文件系统，比如Alluxio。我们之前做过Alluxio和Spark集成，这个时候数据读取，最好的时候，就是PROCESS_LOCAL。其实，Alluxio的文件存储，本质上相当于OFF HEAP的cache，因此可以做到PROCESS_LOCAL。

因此，很显然，Cache是可以保证PROCESS_LOCAL的。但是，要做到PROCESS_LOCAL，却不是一定非要做cache。举个例子，Broadcast Joins机制下，小表的广播，读取的时候也是PROCESS_LOCAL。当然了，你也可以说，广播是一种特殊的缓存，这么说也没毛病。再举个栗子，Collocated joins，你不妨搜搜这个数据关联的计算过程，它的计算，也是PROCESS_LOCAL的。总之一句话，Cache可以保证PROCESS_LOCAL，但是要做到PROCESS_LOCAL，不一定非要做Cache。

疑问2：其实上面说过了，没错，Cache可以保证后面的Task就是PROCESS_LOCAL级别。



Jay

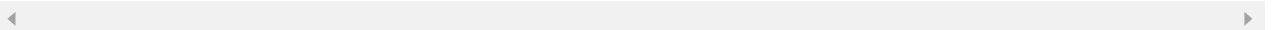
2021-04-27

文中提到 “RDD API使用频率越来越低”。

公司一直还用的是Rdd, 我也一直没学过spark sql。是否有必要切到spark sql呢？

展开 ∨

作者回复: 推荐用Spark SQL哈~ Spark SQL内置了很多优化机制（Catalyst、Tungsten），这个我们在20-23这4讲会详细展开。



kingcall

2021-04-27

DataFrame 要想实现map 端预聚合只能靠优化器自己了吧比较抽象层次比较高了，灵活度就降低了，但是RDD 的话还是可以自己实现实现的，虽然reduceByKey可以预聚合，但是在这个例子中不合适，不能替换collect_list，对于RDD 我们呢可以使用aggregateByKey

ey

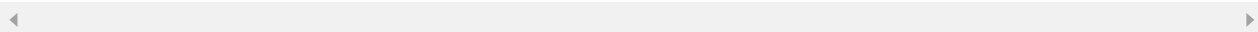
```
def localDistinct(set: Set[String], b: String): mutable.Set[String] = {...
```

展开 ∨

作者回复: Nice~ aggregateByKey玩儿的很溜啊~ 不过咱们这个例子其实可以用collect_set哈~

不过你说的对，确实RDD更灵活，DataFrame抽象更高，那也就意味着灵活性变差。驴和熊猫不可兼得么，low level API比如RDD，高阶算子，开发灵活性非常高，但是优化空间有限；high level API比如DataFrame，Spark SQL优化空间大，但是就没有RDD开发起来那么灵活了。

RDD和DataFrame的恩怨情仇，第20讲可以了解一下~

**zyk**

2021-04-26

问题一: reduceByKey 可以在 map 端预聚合

问题二: 在 map 端聚合的场景，比如求某个 key 的数量，求和等，业务方面来看凡是预聚合不影响正确性的都可以先在 map 端做聚合

问题三: 在读取数据源阶段，可以尽可能将 executor 落在数据同节点上，实现node local，再次就是同个机架下，实现 rack local。数据Shuffle时，可以考虑使用Broadcast代...

展开 ∨

作者回复: 答得挺好~ 不过有个细节有疑问。

问题二: “预聚合不影响正确性的”，我能想到的是“依赖排序的聚合计算”，比如分位数（不同百分位、中位数等等）的计算，你指的是这种操作吗？

