



下载APP



01 | 性能调优的必要性：Spark本身就很快，为啥还需要我调优？

2021-03-15 吴磊

Spark性能调优实战

[进入课程 >](#)**讲述：吴磊**

时长 12:46 大小 11.70M



你好，我是吴磊。

在日常的开发工作中，我发现有个现象很普遍。很多开发者都认为 Spark 的执行性能已经非常强了，实际工作中只要按部就班地实现业务功能就可以了，没有必要进行性能调优。

你是不是也这么认为呢？确实，Spark 的核心竞争力就是它的执行性能，这主要得益于 Spark 基于内存计算的运行模式和钨丝计划的锦上添花，以及 Spark SQL 上的专注与发力。




但是，真如大家所说，**开发者只要把业务逻辑实现了就万事大吉了吗？**这样，咱们先不急于得出结论，你先跟着我一起看两个日常开发中常见的例子，最后我们再来回答这个问题。

在数据应用场景中, ETL (Extract Transform Load) 往往是打头阵的那个, 毕竟源数据经过抽取和转换才能用于探索和分析, 或者是供养给机器学习算法进行模型训练, 从而挖掘出数据深层次的价值。我们今天要举的两个例子, 都取自典型 ETL 端到端作业中常见的操作和计算任务。

开发案例 1: 数据抽取

第一个例子很简单: 给定数据条目, 从中抽取特定字段。这样的数据处理需求在平时的 ETL 作业中相当普遍。想要实现这个需求, 我们需要定义一个函数 `extractFields`: 它的输入参数是 `Seq[Row]` 类型, 也即数据条目序列; 输出结果的返回类型是 `Seq[(String, Int)]`, 也就是 `(String, Int)` 对儿的序列; 函数的计算逻辑是从数据条目中抽取索引为 2 的字符串和索引为 4 的整型。

应该说这个业务需求相当简单明了, 实现起来简直是小菜一碟。在实际开发中, 我观察到有不少同学一上来就迅速地用下面的方式去实现, 干脆利落, 代码写得挺快, 功能也没问题, UT、功能测试都能过。

 复制代码

```
1 //实现方案1 — 反例
2 val extractFields: Seq[Row] => Seq[(String, Int)] = {
3   (rows: Seq[Row]) => {
4     var fields = Seq[(String, Int)]()
5     rows.map(row => {
6       fields = fields :+ (row.getString(2), row.getInt(4))
7     })
8     fields
9   }
10 }
```

在上面这个函数体中, 是先定义一个类型是 `Seq[(String, Int)]` 的变量 `fields`, 变量类型和函数返回类型完全一致。然后, 函数逐个遍历输入参数中的数据条目, 抽取数据条目中索引是 2 和 4 的字段并且构建二元元组, 紧接着把元组追加到最初定义的变量 `fields` 中。最后, 函数返回类型是 `Seq[(String, Int)]` 的变量 `fields`。

乍看上去, 这个函数似乎没什么问题。特殊的地方在于, 尽管这个数据抽取函数很小, 在复杂的 ETL 应用里是非常微小的一环, 但在整个 ETL 作业中, 它会在不同地方被频繁地反

复调用。如果我基于这份代码把整个 ETL 应用推上线, 就会发现 ETL 作业端到端的执行效率非常差, 在分布式环境下完成作业需要两个小时, 这样的速度难免有点让人沮丧。

想要让 ETL 作业跑得更快, 我们自然需要做性能调优。可问题是我们该从哪儿入手呢? 既然 `extractFields` 这个小函数会被频繁地调用, 不如我们从它下手好了, 看看有没有可能给它“减个肥、瘦个身”。重新审视函数 `extractFields` 的类型之后, 我们不难发现, 这个函数从头到尾无非是从 `Seq[Row]` 到 `Seq[(String, Int)]` 的转换, 函数体的核心逻辑就是字段提取, 只要从 `Seq[Row]` 可以得到 `Seq[(String, Int)]`, 目的就达到了。

要达成这两种数据类型之间的转换, 除了利用上面这种开发者信手拈来的过程式编程, 我们还可以用函数式的编程范式。函数式编程的原则之一就是尽可能地在函数体中避免副作用 (Side effect), 副作用指的是函数对于状态的修改和变更, 比如上例中 `extractFields` 函数对于 `fields` 变量不停地执行追加操作就属于副作用。

基于这个想法, 我们就有了第二种实现方式, 如下所示。与第一种实现相比, 它最大的区别在于去掉了 `fields` 变量。之后, 为了达到同样的效果, 我们在输入参数 `Seq[Row]` 上直接调用 `map` 操作逐一地提取特定字段并构建元组, 最后通过 `toSeq` 将映射转换为序列, 干净利落, 一气呵成。


[复制代码](#)

```
1 //实现方案2 — 正例
2 val extractFields: Seq[Row] => Seq[(String, Int)] = {
3   (rows: Seq[Row]) =>
4     rows.map(row => (row.getString(2), row.getInt(4))).toSeq
5 }
6
```

你可能会问: “两份代码实现无非是差了个中间变量而已, 能有多大差别呢? 看上去不过是代码更简洁了而已。”事实上, 我基于第二份代码把 ETL 作业推上线后, 就惊奇地发现端到端执行性能提升了一倍! 从原来的两个小时缩短到一个小时。**两份功能完全一样的代码, 在分布式环境中的执行性能竟然有着成倍的差别。因此你看, 在日常的开发工作中, 仅仅专注于业务功能实现还是不够的, 任何一个可以进行调优的小环节咱们都不能放过。**

开发案例 2: 数据过滤与数据聚合

你也许会说: “你这个例子只是个例吧? 更何况, 这个例子中的优化, 仅仅是编程范式的调整, 看上去和 Spark 似乎也没什么关系啊!” 不要紧, 我们再来看第二个例子。第二个例子会稍微复杂一些, 我们先来把业务需求和数据关系交代清楚。

 复制代码


```
1  /**
2   (startDate, endDate)
3   e.g. ("2021-01-01", "2021-01-31")
4  */
5  val pairDF: DataFrame = _
6
7  /**
8   (dim1, dim2, dim3, eventDate, value)
9   e.g. ("X", "Y", "Z", "2021-01-15", 12)
10 */
11 val factDF: DataFrame = _
12
13 // Storage root path
14 val rootPath: String = _
15
```

在这个案例中, 我们两份数据, 分别是 pairDF 和 factDF, 数据类型都是 DataFrame。第一份数据 pairDF 的 Schema 包含两个字段, 分别是开始日期和结束日期。第二份数据的字段较多, 不过最主要的字段就两个, 一个是 Event date 事件日期, 另一个是业务关心的统计量, 取名为 Value。其他维度如 dim1、dim2、dim3 主要用于数据分组, 具体含义并不重要。从数据量来看, pairDF 的数据量很小, 大概几百条记录, factDF 数据量很大, 有上千万行。

对于这两份数据来说, 具体的业务需求可以拆成 3 步:

1. 对于 pairDF 中的每一组时间对, 从 factDF 中过滤出 Event date 落在其间的数据条目;
2. 从 dim1、dim2、dim3 和 Event date 4 个维度对 factDF 分组, 再对业务统计量 Value 进行汇总;
3. 将最终的统计结果落盘到 Amazon S3。

针对这样的业务需求, 不少同学按照上面的步骤按部就班地进行了如下的实现。接下来, 我就结合具体的代码来和你说说其中的计算逻辑。


 复制代码

```
1 //实现方案1 — 反例
2 def createInstance(factDF: DataFrame, startDate: String, endDate: String): Dat
3 val instanceDF = factDF
4 .filter(col("eventDate") > lit(startDate) && col("eventDate") <= lit(endDate))
5 .groupBy("dim1", "dim2", "dim3", "event_date")
6 .agg(sum("value") as "sum_value")
7 instanceDF
8 }
9
10 pairDF.collect.foreach{
11 case (startDate: String, endDate: String) =>
12 val instance = createInstance(factDF, startDate, endDate)
13 val outputPath = s"${rootPath}/endDate=${endDate}/startDate=${startDate}"
14 instance.write.parquet(outputPath)
15 }
```

首先, 他们是以 factDF、开始时间和结束时间为形参定义 createInstance 函数。在函数体中, 先根据 Event date 对 factDF 进行过滤, 然后从 4 个维度分组汇总统计量, 最后将汇总结果返回。定义完 createInstance 函数之后, 收集 pairDF 到 Driver 端并逐条遍历每一个时间对, 然后以 factDF、开始时间、结束时间为实参调用 createInstance 函数, 来获取满足过滤要求的汇总结果。最后, 以 Parquet 的形式将结果落盘。

同样地, 这段代码从功能的角度来说没有任何问题, 而且从线上的结果来看, 数据的处理逻辑也完全符合预期。不过, 端到端的执行性能可以说是惨不忍睹, 在 16 台机型为 C5.4xlarge AWS EC2 的分布式运行环境中, 基于上面这份代码的 ETL 作业花费了半个小时才执行完毕。

没有对比就没有伤害, 在同一份数据集之上, 采用下面的第二种实现方式, 仅用 2 台同样机型的 EC2 就能让 ETL 作业在 15 分钟以内完成端到端的计算任务。**两份代码的业务功能和计算逻辑完全一致, 执行性能却差了十万八千里。**

 复制代码

```
1 //实现方案2 — 正例
2 val instances = factDF
3 .join(pairDF, factDF("eventDate") > pairDF("startDate") && factDF("eventDate")
4 .groupBy("dim1", "dim2", "dim3", "eventDate", "startDate", "endDate")
5 .agg(sum("value") as "sum_value")
6
7 instances.write.partitionBy("endDate", "startDate").parquet(rootPath)
```


那么问题来了, 这两份代码到底差在哪里, 是什么导致它们的执行性能差别如此之大。我们不妨先来回顾第一种实现方式, 嗅一嗅这里面有哪些不好的代码味道。

我们都知道, 触发 Spark 延迟计算的 Actions 算子主要有两类: 一类是将分布式计算结果直接落盘的操作, 如 DataFrame 的 write、RDD 的 saveAsTextFile 等; 另一类是将分布式结果收集到 Driver 端的操作, 如 first、take、collect。

显然, 对于第二类算子来说, Driver 有可能形成单点瓶颈, 尤其是用 collect 算子去全量收集较大的结果集时, 更容易出现性能问题。因此, 在第一种实现方式中, 我们很容易就能嗅到 collect 这里的调用, 味道很差。

尽管 collect 这里味道不好, 但在我们的场景里, pairDF 毕竟是一份很小的数据集, 才几百条数据记录而已, 全量搜集到 Driver 端也不是什么大问题。

最要命的是 collect 后面的 foreach。要知道, factDF 是一份庞大的分布式数据集, 尽管 createInstance 的逻辑仅仅是对 factDF 进行过滤、汇总并落盘, 但是 createInstance 函数在 foreach 中会被调用几百次, pairDF 中有多少个时间对, createInstance 就会被调用多少次。对于 Spark 中的 DAG 来说, 在没有缓存的情况下, 每一次 Action 的触发都会导致整条 DAG 从头到尾重新执行。

明白了这一点之后, 我们再来仔细观察这份代码, 你品、你细品, 目不转睛地盯着 foreach 和 createInstance 中的 factDF, 你会惊讶地发现: 有着上千万行数据的 factDF 被反复扫描了几百次! 而且, 是全量扫描哟! 吓不吓人? 可不可怕? 这么分析下来, ETL 作业端到端执行效率低下的始作俑者, 是不是就暴露无遗了?

反观第二份代码, factDF 和 pairDF 用 pairDF.startDate < factDF.eventDate <= pairDF.endDate 的不等式条件进行数据关联。在 Spark 中, 不等式 Join 的实现方式是 Nested Loop Join。尽管 Nested Loop Join 是所有 Join 实现方式 (Merge Join, Hash Join, Broadcast Join 等) 中性能最差的一种, 而且这种 Join 方式没有任何优化空间, 但 factDF 与 pairDF 的数据关联只需要扫描一次全量数据, 仅这一项优势在执行效率上就可以吊打第一份代码实现。

小结

今天，我们分析了两个案例，这两个案例都来自数据应用的 ETL 场景。第一个案例讲的是，在函数被频繁调用的情况下，函数里面一个简单变量所引入的性能开销被成倍地放大。第二个例子讲的是，不恰当的实现方式导致海量数据被反复地扫描成百上千次。

通过对这两个案例进行分析和探讨，我们发现，对于 Spark 的应用开发，绝不仅仅是完成业务功能实现就高枕无忧了。**Spark 天生的执行效率再高，也需要你针对具体的应用场景和运行环境进行性能调优。**

而性能调优的收益显而易见：一来可以节约成本，尤其是按需付费的云上成本，更短的执行时间意味着更少的花销；二来可以提升开发的迭代效率，尤其是对于从事数据分析、数据科学、机器学习的同学来说，更高的执行效率可以更快地获取数据洞察，更快地找到模型收敛的最优解。因此你看，性能调优不是一件锦上添花的事情，而是开发者必须要掌握的一项傍身技能。

那么，对于 Spark 的性能调优，你准备好了吗？生活不止眼前的苟且，让我们来一场说走就走的性能调优之旅吧。来吧！快上车！扶稳坐好，系好安全带，咱们准备发车了！

每日一练

1. 日常工作中，你还遇到过哪些功能实现一致、但性能大相径庭的案例吗？
2. 我们今天讲的第二个案例中的正例代码，你觉得还有可能进一步优化吗？

期待在留言区看到你分享，也欢迎把你对开发案例的思考写下来，我们下节课见！

提建议

12.12 大促

每日一课 VIP 年卡

10分钟, 解决你的技术难题

¥159/年 ¥365/年

每日一课
VIP 年卡

仅3天, 【点击】图片, 立即抢购 >>>

© 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 开篇词 | Spark性能调优, 你该掌握这些“套路”

下一篇 02 | 性能调优的本质: 调优的手段五花八门, 该从哪里入手?

精选留言 (15)

写留言

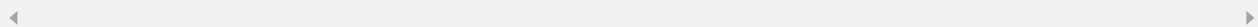


Will

2021-03-15

第二个例子, 可以利用map join, 让小数据分发到每个worker上, 这样不用shuffle数据

作者回复: 没错, Broadcast joins可以进一步提升性能。



2

8

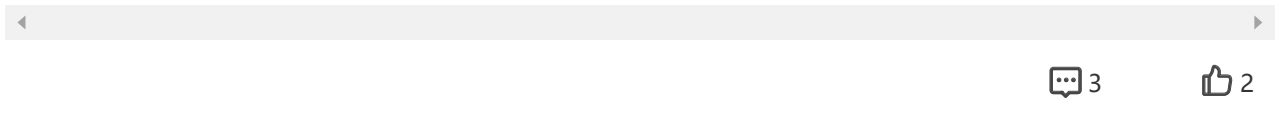


Taolnsight

2021-03-17

如果paimon的startDate和endDate范围有限, 可以把日期范围展开, 将非等值join转成等值join

作者回复: 这块能展开说说吗? 具体怎么转化为等值join? 可以举个例子哈~



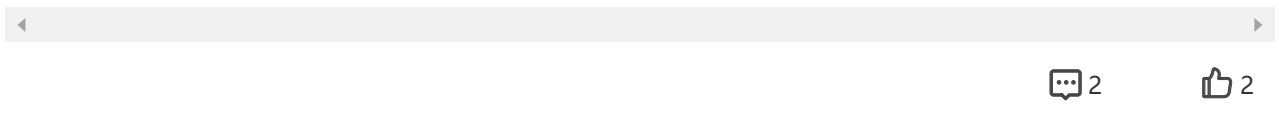
葛聂

2021-03-16

Case 1为什么性能差一倍呢

展开 ∨

作者回复: 好问题, 其实改动非常小, 开销相比正例也不大, 但这里的关键在于, 这个函数会被反反复复调用上百次, 累积下来, 开销就上去了。所以, 关键不在于点小不小, 而是这个点, 是不是瓶颈。



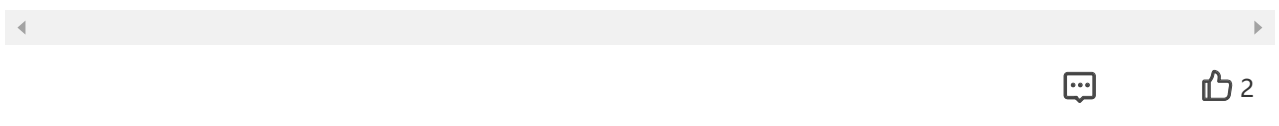
对方正在输入。。。

2021-03-15

可以先将pairedf collect到driver, 再将数组按照startdate排序, 然后再将其广播。然后在factdf.map里面实现一个方法来从广播的数组里面二分查找到eventdate所属的时间对子。最后就可以根据这个时间对子以及其他的维度属性进行分组聚合了

展开 ∨

作者回复: 广播的思路很赞。不过二分查找这里值得商榷哈, 咱们目的是过滤出满足条件的event date, 然后和其他维度一起、分组聚合。这里关键不在于过滤和查找效率, 关键在于大表的重复扫描, 只要解决这个核心痛点, 性能问题就迎刃而解。



刘吉超

2021-03-24

我们每天有9T数据, 用如下代码做ETL json平铺, 花很长时间

```
val adArr = ArrayBuffer[Map[String, String]]()
```

```
if (ads != null) {
```

```
  val adnum = ads.length
```

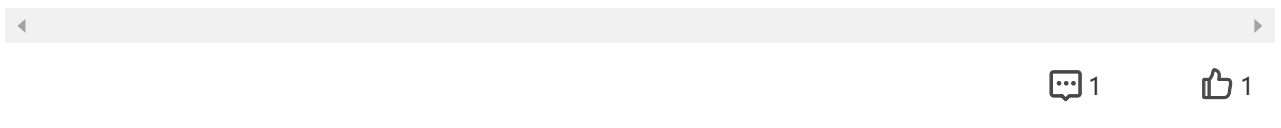
```
  for (i <- 0 until adnum) {...
```

展开 ∨

编辑回复: 兄弟我是作者哈, 第二份代码, 我有几个疑问:

1. 两个import语句的作用是什么?
2. ads具体是什么内容? 是RDD, 还是数组, 还是什么?
3. 没有看到哪里定义分布式数据集, 所有计算看上去是基于(0 until ads.size())这个List, 那么后续所有的计算, map, map里面的toMap, 都是在driver计算的, 如果你9T数据都在driver计算, 那结果。。。
4. toMap之后, 又加了个map, 我理解是为了把value中的null替换为空字符串, 如果是这样的话, map里面只处理value就好了, 不用带着key

不知道我理解的对不对哈, 期待老弟提供更多信息~



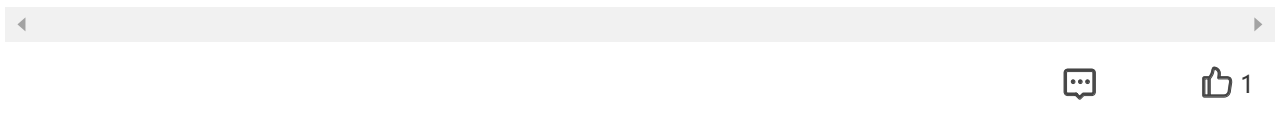
Elon

2021-03-22

函数式的副作用指的是不修改入参吧? 在函数内部是可以定义变量、修改变量的。因此fields变量在函数内部, 应该不算副作用吧?

展开 ∨

作者回复: 是的, 你说的没错。函数的副作用指的是对外部变量、外部环境的影响, 内部状态的改变和转换不算。文中这块表述的不严谨哈, 这里主要是想强调可变变量fields带来的计算开销。



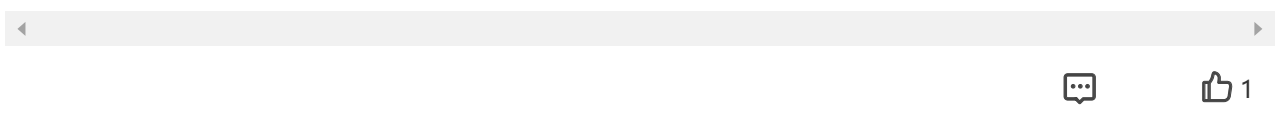
fsc2016

2021-03-18

请问老师, 这个课程需要哪些基础, 我平时使用过pysaprk 做过一些机器学习相关数据处理练习, 对于我这种使用spark不多的, 可以消化吸收嘛

展开 ∨

作者回复: 可以, 没问题, 接触过Spark就行。放心吧, 原理部分会有大量的生活化类比和故事, 尽可能地让你“边玩边学”。另外, 咱们有微信群, 有问题可以随时探讨~

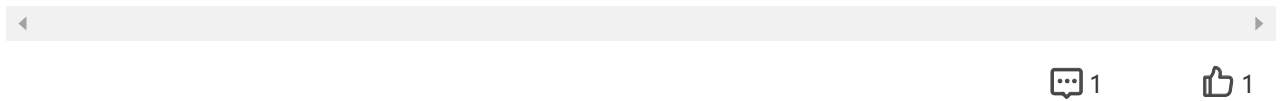


Fendora范东_

2021-03-31

请问磊哥, spark里面nested loop join和cartesian product jion有什么区别?

作者回复: nlj是一种join实现方式哈, 和hash join、sort merge join一样, 是一种join实现机制。cartesian join是一种join形式, 和inner、left、right对等。每一种join形式, 都可以用多种实现机制来做、来实现 ~

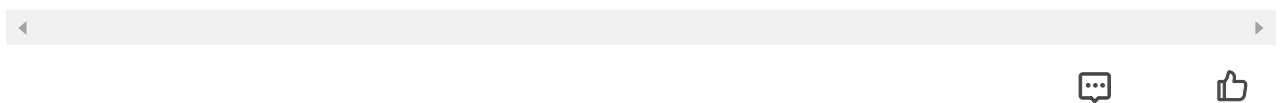


方得始终

2021-03-21

错过了老师上星期的直播, 请问在哪里可以看到回放吗?

编辑回复: 上传到了B站: <https://www.bilibili.com/video/BV1AX4y1G7Ks/>



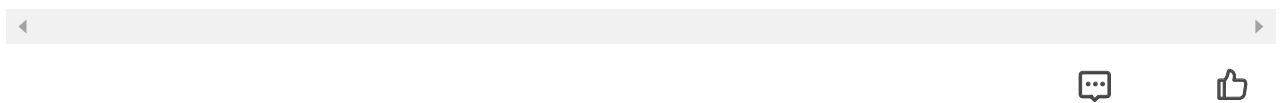
方得始终

2021-03-20

代码会集中放到GitHub上面吗? 这样方便于以后学习查找。我主要用 pyspark, 这个课程只用Scala吗? 请问有关于Scala spark的入门教程吗?

展开 ∨

作者回复: 可以的, 后面会统一放到git上, 方便大家查阅。对, 目前专栏里的代码示例都是Scala, 主要是简洁, 用很短的代码就能示意。如果习惯用pyspark开发, 其实不用刻意去学Scala, API其实大同小异, 区别不大。pyspark比较熟悉的话, Scala的代码也很容易看懂 ~ 另外, 执行性能pyspark也不差, 和Scala, Java相当。除了udf需要跨进程、有一些开销之外, 整体还好。

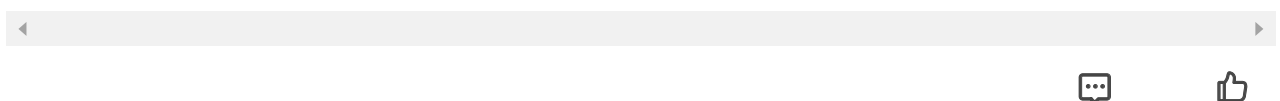


天渡

2021-03-18

可否将小表进行broadcast, 将reduce端join变为map端join。

作者回复: 广播的思路没问题 ~

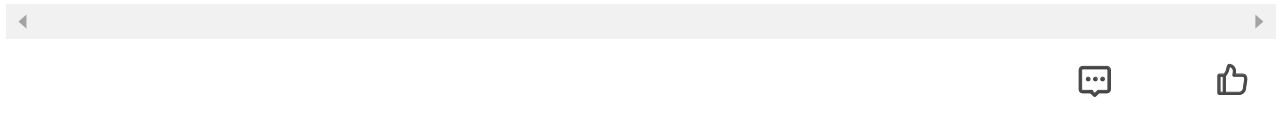


LJK

2021-03-17

同一个application如果action多的话一定会影响效率吗?

作者回复: 不一定的, action个数不是关键, 关键是数据的访问效率。关于提升数据访问效率, 咱们专栏后面的内容会讲哈~



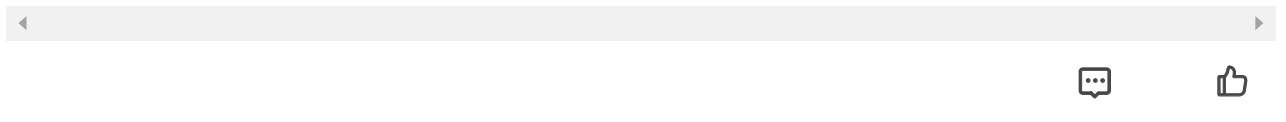
Shockang

2021-03-16

Case1里面除了老师讲的副作用外, 我认为Scala在处理闭包时也会存在一定的性能损耗, Case2里面把大变量广播出去是一种常见的操作, 另外, filter之后加上coalesce 也是比较常见的优化手段

作者回复: case1的副作用本身其实还好, 主要是它引入的开销, 就是你说的性能损耗。case2的广播思路没问题~

coalesce这里有待商榷哈, 正例里面已经没有filter, 反例里面加coalesce也于事无补。

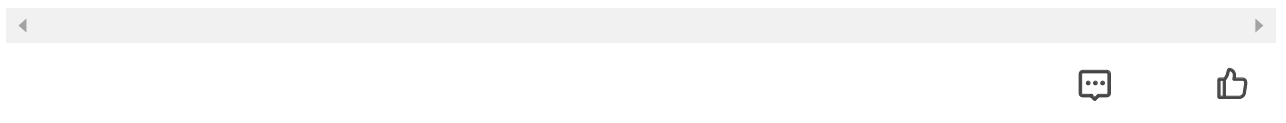


rb@31

2021-03-16

另外, 不知道老师的更新频率。大概多久能全部更新好?

编辑回复: 你好, 课程详情页处有更新安排哦, 每周一, 三, 五更新。预计更新到5月份



Geek_92df49

2021-03-15

四个维度分组为什么要加上开始时间和结束时间?

```
.groupBy("dim1", "dim2", "dim3", "event_date", "startDate", "endDate")
```

展开 ∨

编辑回复: 兄弟我是作者哈, 你说的没错, 分组只需要前4个字段, 但是你看最后, instances.write.partitionBy("endDate", "startDate").parquet(rootPath), 需要用开始和结束时间这两个字段去做分区存储, 因此, 在前一步分组的时候, 把这两个字段保留了下来。

