High Performance Spark - June 3'rd, 2018

Joins(SQL and Core)

https://github.com/rh01

Agenda

本部分主要讲解了在 Spark Core 和 Spark SQL 中核心的操作之一：Join 操作，Join 数据是我们许多 pipeline 的重要组成部分，Spark Core 和 SQL 都支持基本类型的 Join. 虽然 Join 操作非常普遍且强大，但它们在使用的过程中往往受性能要求，因为 Join 操作可能需要大量的网络传输，甚至 Join 后的产生的数据集可能会超出我们 Spark（集群）处理能力. 在 Spark Core 中，与 SQL 优化器不同，DAG 的操作顺序可能更为重要，优化器无法重新排序或过滤下推.

Core Spark Joins

在这节将会介绍 RDD 的连接操作. 连接操作通常很昂贵，因为它们要求每个 RDD 对应的键位于同一分区，以便它们可以在本地组合. 如果 RDD 没有已知的分区器，则需要对它们进行重新“洗牌”，以便两个 RDD 在同一个分区，具有相同键的数据存在于相同的分区中，如图 1 所示. 如果它们具有相同的分区，则可以对数据进行合并，如图 2 所示，以避免网络传输. 无论分区器是否相同，如果一个（或两个）RDD 具有已知分区器，则只创建一个窄依赖关系，如图 3 所示. 与大多数键/值操作一样，连接的成本随着键的数量和记录必须经过的距离而增加，以便到达正确的分区.

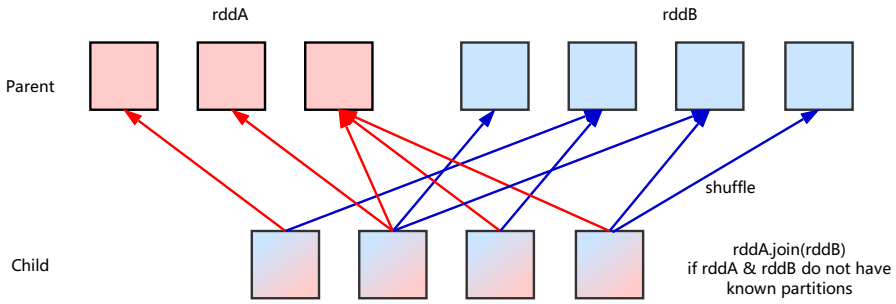

图 1：Shuffle join

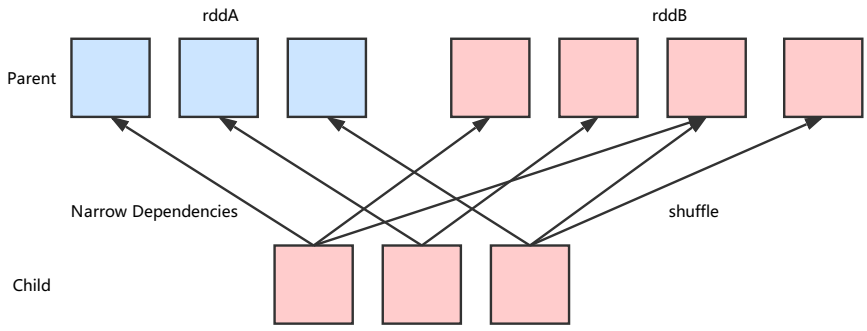

图 2：Both known partitioner join

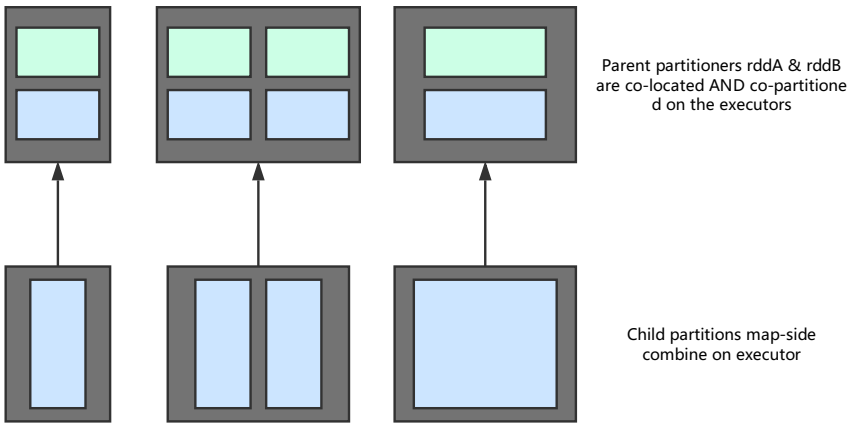

图 3：Colocated join

如何选择 Join 操作的类型

以下是选择 Join 操作的建议：

- 当全部 RDDs 有相同的键时，此时全部 RDD 之间可以进行 join 操作，如果相同的键比较少时，那么 join 产生的数据将会指数增加，这时可以利用 `distinct` 和 `combineByKey` 等操作来减少数据键空间的大小，或者利用 `cogroup` 操作冗余的键，而不是像笛卡尔积那样，这样做的好处就是可以避免 shuffle 过程的发生,
- 当全部的 RDDs 没有相同的键时，这样立即做 join 操作会有很大的风险丢失数据，这时可以使用外连接操作，外连接操作可以保证数据在左边或者右边，然后在 join 之后，再执行 `filter` 操作，这样做比较好.
- 如果一个 RDD 有一些易于定义的键的子集，此时可以采取 `filter` 操作或者在 join 之前，减少键的大小，从而可以避免数据的 `shuffle` 过程.

> join 操作在 Spark 中代价是非常大的，因此在执行 join 之前，最好将你的数据适当的收缩一下.

比如，假设你的 RDD 格式是 `(Panda id, score)`, 另一个 RDD 的格式是 `(Panda id, address)`, 这时你想要把每个 `Panda` 的 `email` 和它的最好 `socore` 关联起来，这时可以利用在 `id` 上执行连接操作，然后对于每个 `address` 计算出最好的成绩，代码如 Listing 1 所示.

```
1  def joinScoresWithAddress1( scoreRDD : RDD[(Long, Double)],
2      addressRDD : RDD[(Long, String )]) : RDD[(Long, (Double, String))]= {
3      val joinedRDD = scoreRDD.join(addressRDD)
4      joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y )
5  }
```

Listing 1: Basic RDD join

这样做在 reduceByKey 阶段时，并不是很快，因为 join 操作并不是最优. 那么有一个问题就是你的第一个数据集存在着一个 id 可能有很多 socre，那么我们可以在 join 之前先算出最佳的成绩，然后再执行 join 操

作. 修改版的 join 见 Listing 2.

```
def joinScoresWithAddress2(scoreRDD : RDD[(Long, Double)],
    addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {
    val bestScoreData = scoreRDD.reduceByKey((x, y) => if(x > y) x else y)
    bestScoreData.join(addressRDD)
}
```

Listing 2: Pre-filter before join

如果每个 Panda 有 1,000 个成绩，那么第一种做法再 shuffle 的次数将是第二种做法的 1,000 倍. 通过使用 leftOuterJoin 代替 join 来保留所有处理过程中右边 RDD 缺失的键. Spark 还有 fullOuterJoin 和 rightOuterJoin 等操作，具体取决于我们希望保留的记录.

```
def outerJoinScoresWithAddress(scoreRDD : RDD[(Long, Double)],
    addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, Option[String]))]= {
    val joinedRDD = scoreRDD.leftOuterJoin(addressRDD)
    joinedRDD.reduceByKey( (x, y) => if(x._1 > y._1) x else y )
}
```

Listing 3: Basic RDD left outer join

选择合适的执行方案

为了连接数据，Spark 需要将要连接的数据（即基于每个键的数据）存放在同一分区上. Spark 中的连接的默认实现是 *shuffle hash join*. 它通过使用与第一个数据集相同的默认分区器对第二个数据集进行分区来确保每个分区上的数据将包含相同的键，以便来自两个数据集的具有相同散列值的键位于同一分区中. 虽然这种方法总是有效，但它的代价可能更高，因为它需要 shuffle 过程. 如果出现以下情况，可以避免洗牌：

- 这两个 RDD 都有一个已知的分区器.
- 其中一个数据集足够小以装载到内存，在这种情况下我们可以进行广播散列连接（*broadcast hash join*）.

通过分配已知分区来加速连接

如果必须在需要 shuffle 的连接之前执行操作，例如 `aggregateByKey` 或 `reduceByKey` 操作，则可以通过添加具有与显式参数相同数量的分区的哈希分区程序来防止 shuffle.

那么我们可以再改进一下上面的程序.

```
  def joinScoresWithAddress3(scoreRDD: RDD[(Long, Double)],
   addressRDD: RDD[(Long, String)]) : RDD[(Long, (Double, String))]= {
    // If addressRDD has a known partitioner we should use that,
    // otherwise it has a default hash parttioner, which we can reconstruct by
    // getting the number of partitions.
    val addressDataPartitioner = addressRDD.partitioner match {
      case (Some(p)) => p
      case (None) => new HashPartitioner(addressRDD.partitions.length)
    }
```

```
10      val bestScoreData = scoreRDD.reduceByKey(addressDataPartitioner,
11        (x, y) => if(x > y) x else y)
12      bestScoreData.join(addressRDD)
13    }
```

Listing 4: Known partitioner join

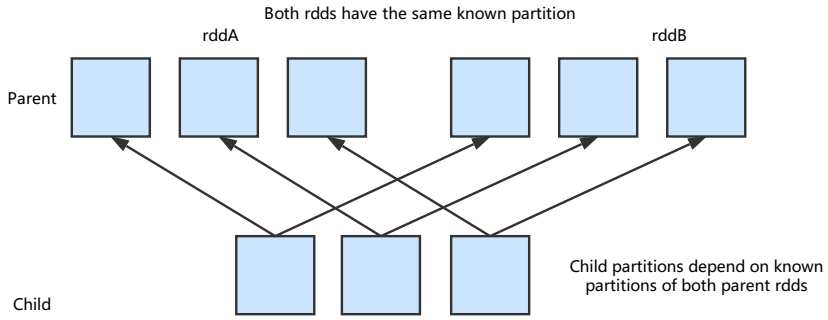

图 4：Both known partitioner join

Always persist after repartitioning.

Speeding up joins using a broadcast hash join

广播散列连接将其中一个 RDD（较小的一个）推送到每个工作节点. 然后它与较大 RDD 的每个分区进行映射组合. 如果您的某个 RDD 可以装载到内存或者可以使其装载到内存，那么进行广播散列连接总是有帮助的，因为它不需要 shuffle 操作. 有时（但不总是）Spark SQL 可以智能地配置广播连接本身; 在 Spark SQL 中，它由 `spark.sql.autoBroadcastJoinThreshold` 和 `spark.sql.broadcastTimeout` 控制. 如图 5 所示.

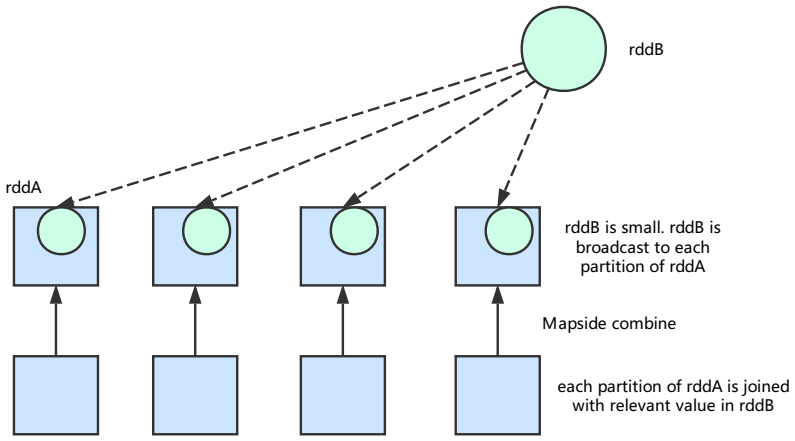

图 5：Broadcast hash join

Spark Core 没有广播散列连接的实现. 但是，我们可以手动实现广播散列连接，方法是将较小的 RDD 作为映射收集到驱动程序，然后广播结果，并使用 `mapPartitions` 组合元素.

Listing 5 是一个用于连接更大的 RDD 的通用函数. 它的行为反映了 Spark 中的默认“join”操作. 我们排除其键不出现在两个 RDD 中的元素.

```
def manualBroadCastHashJoin[K : Ordering : ClassTag, V1 : ClassTag,
 V2 : ClassTag](bigRDD : RDD[(K, V1)],
  smallRDD : RDD[(K, V2)])= {
  val smallRDDLocal: Map[K, V2] = smallRDD.collectAsMap()
  bigRDD.sparkContext.broadcast(smallRDDLocal)
  bigRDD.mapPartitions(iter => {
   iter.flatMap{
    case (k,v1 ) =>
     smallRDDLocal.get(k) match {
      case None => Seq.empty[(K, (V1, V2))]
      case Some(v2) => Seq((k, (v1, v2)))
     }
   }
  }, preservesPartitioning = true)
 }
 //end:coreBroadCast[]
}
```

Listing 5: Manual broadcast hash join

有时并非所有较小的 RDD 都可以装载到内存中，但是在大型数据集中，某些键过多，而你只想广播最常用的键. 如果当一个键太大而无法放在单个分区上，这尤其有用. 在这种情况下，可以在 RDD 上使用 `countByKeyApprox` 来大致了解哪些键最有利于广播.

Spark SQL Joins

Spark SQL 支持与 Spark core 相同的基本连接类型，但优化器能够完成更多繁重工作. 例如，Spark SQL 有时可以 *push down* 或重新排序操作，以使连接操作效率更高. 另一方面，您不能操纵 DataFrames 或 Datasets 的分区程序，因此我们无法像使用 Spark core 连接那样手动避免 shuffle.

DataFrame Joins

```
// Inner join implicit
df1.join(df2, df1("name") === df2("name"))

// Inner join explicit
df1.join(df2, df1("name") === df2("name"), "inner")

// Left outer join explicit
df1.join(df2, df1("name") === df2("name"), "left_outer")

// Right outer join explicit
df1.join(df2, df1("name") === df2("name"), "right_outer")

// Left semi join explicit
```

```
14  df1.join(df2, df1("name") === df2("name"), "left_semi")
```

Listing 6: Simple join

Self joins

DataFrames 支持自连接,但会出现重复的列名. 在访问结果时,可以将 `DataFrame` 设置为不同的名称,否则会因为名称冲突,而无法选择产生的列. 在为每个 DataFrame 设置别名后,可以在结果中使用 `dfName.colName` 访问每个 `DataFrame` 的各个列.

```
val joined = df.as("a").join(df.as("b")).where($"a.name" === $"b.name")
```

Broadcast hash joins

在 Spark SQL 中，可以通过调用 `queryExecution.executedPlan` 来查看正在执行的连接类型. 与 Spark core 一样，如果其中一个表比另一个表小得多，则可能需要广播散列连接 (*broadcast hash join*).

在 `join` 之前通过在 DataFrame 上调用广播来向 Spark SQL 提示. （例如，`df1.join(broadcast(df2), "key")`). Spark 还会自动使用 `spark.sql.conf.autoBroadcastJoinThreshold` 来确定是否应该广播表.

Dataset Joins

数据集的连接是使用 `joinWith` 完成的，其行为类似于常规关系数据库的连接，但结果是不同记录类型的元组. 它可以使自连接变得更容易，因为不需要对列进行别名处理.

```
1  // Joining two databases
2  val result: Dataset[(RawPanda, CoffeeShop)] = pandas.joinWith(coffeeShops,
3      $"zip" === $"zip")
4
5  // self join a Database
6  val result: Dataset[(RawPanda, RawPanda)] = pandas.joinWith(pandas,
7      $"zip" === $"zip")
```

Listing 7: Join two database & self join a Database

Using a self join and a lit(true), you can produce the cartesian product of your Dataset, which can be useful but also illustrates how joins (especially self joins) can easily result in unworkable data sizes.

与 DataFrames 一样, 可以指定所需的连接类型（例如，`inner`，`left_outer`，`right_outer`，`left_semi`）. 缺少的记录由空值表示，因此请小心.