1. 史诗级最详细10招Spark数据倾斜调优
2. 预习分布式计算应用执行机制
   2. 1. MapReduce执行流程
   2. 2. Spark应用程序执行流程
3. 什么是数据倾斜
   3. 1. 数据倾斜发生的现象
   3. 2. 数据倾斜发生的原理
   3. 3. 数据倾斜的危害
   3. 4. 数据倾斜是如何造成的
   3. 5. 如何消除或缓解数据倾斜
      3. 5. 1. 避免数据源倾斜-HDFS
      3. 5. 2. 避免数据源倾斜-Kafka
      3. 5. 3. 定位处理逻辑 - Stage 和 Task
      3. 5. 4. 查看导致倾斜的key的数据分布情况
4. 数据倾斜解决方案
   4. 1. 方案一：使用Hive ETL预处理数据
      4. 1. 1. 适用场景
      4. 1. 2. 实现思路
      4. 1. 3. 实现原理
      4. 1. 4. 方案优缺点
      4. 1. 5. 企业最佳实践
   4. 2. 方案二：调整shuffle操作的并行度
      4. 2. 1. 适用场景
      4. 2. 2. 实现思路
      4. 2. 3. 实现原理
      4. 2. 4. 方案优缺点
      4. 2. 5. 企业最佳实践
   4. 3. 方案三：过滤少数导致倾斜的key
      4. 3. 1. 适用场景
      4. 3. 2. 实现思路
      4. 3. 3. 实现原理
      4. 3. 4. 方案优缺点
      4. 3. 5. 企业最佳实践
   4. 4. 方案四：将reduce join转为map join
      4. 4. 1. 适用场景
      4. 4. 2. 实现思路
      4. 4. 3. 实现原理
      4. 4. 4. 方案优缺点
   4. 5. 方案五：采样倾斜 key 并分拆 join 操作
      4. 5. 1. 适用场景
      4. 5. 2. 实现思路
      4. 5. 3. 实现原理
      4. 5. 4. 方案优缺点
   4. 6. 方案六：两阶段聚合（局部聚合+全局聚合）
      4. 6. 1. 适用场景
      4. 6. 2. 实现思路
      4. 6. 3. 实现原理
      4. 6. 4. 方案优缺点
   4. 7. 方案七：使用随机前缀和扩容 RDD 进行 join

4. 7. 1. 适用场景
4. 7. 2. 实现思路
4. 7. 3. 实现原理
4. 7. 4. 方案优缺点
4. 7. 5. 企业最佳实践
4. 8. 方案八：任务横切，一分为二，单独处理
4. 8. 1. 适用场景
4. 8. 2. 实现思路
4. 8. 3. 实现原理
4. 8. 4. 方案优缺点
4. 9. 方案九：多种方案组合使用
4. 10. 方案十：自定义 Partitioner
4. 10. 1. 适用场景
4. 10. 2. 实现思路
4. 10. 3. 实现原理
4. 10. 4. 常见分区解释
4. 10. 5. 方案优缺点
5. Spark倾斜调优案例
5. 1. 方案十一：Spark 整合 BitMap 求 Join
5. 1. 1. 适用场景
5. 1. 2. 实现思路
5. 1. 3. 实现原理
5. 1. 4. 方案优缺点

1. 史诗级最详细10招Spark数据倾斜调优

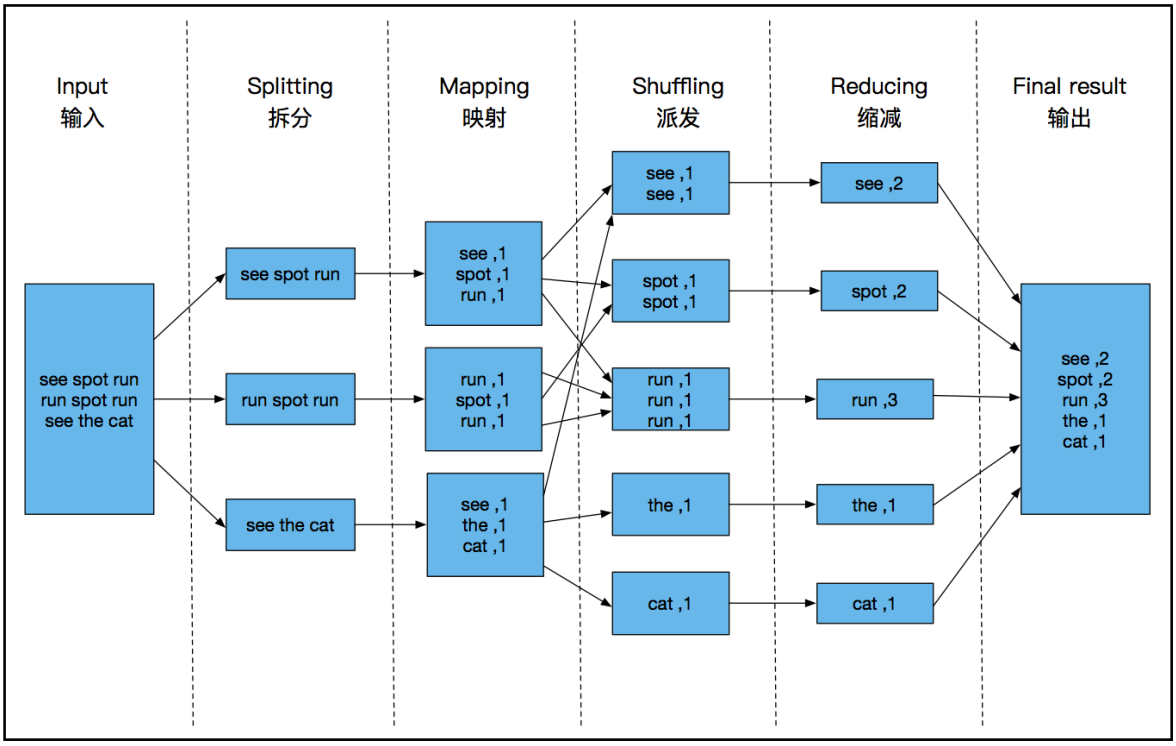
数据倾斜是大数据计算中一个最棘手的问题，出现数据倾斜后，Spark 作业的性能会比期望值差很多，两大直接后果：Spark 任务 OOM 异常退出，数据倾斜拖慢整个任务的执行。数据倾斜的调优，就是利用各种技术方案解决不同类型的数据倾斜问题，保证 Spark 作业的性能。掌握 Spark 的数据倾斜不仅能在提升工作效率，还能在面试中脱颖而出。

2. 预习分布式计算应用执行机制

2.1. MapReduce执行流程

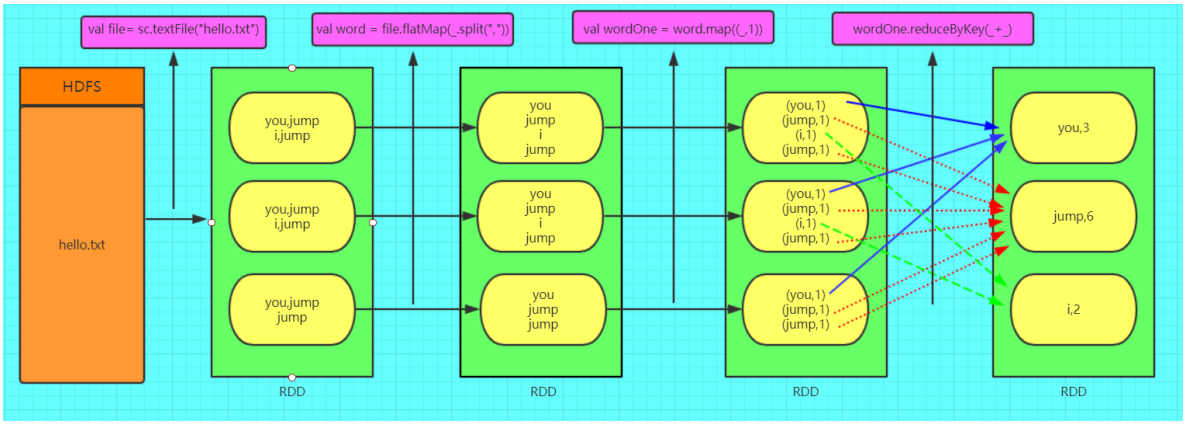
核心思想：大问题拆分成多个小问题，然后分布式的并行执行

```
两个阶段：
1、mapper阶段：  提取数据，赋予特征   映射     value  ====>  key, value
    mapreducce框架是怎么把相同特征的数据组合到一起来，然后交给reduceTask执行一次聚合操作的呢
2、reducer阶段：  把相同特征的数据进行聚合操作     key, (value, value, ...)
```



2.2. Spark应用程序执行流程

sparkContext .rextFile() .flatMap() .map() .reduceByKey()



首先让你确认，导致分布式计算应用改程序出现数据倾斜的原因就是 Shuffle 数据倾斜的调优，都是围绕着：

```
1、要么就不要使用shuffle
2、要么就让shuffle在执行过程中均匀分发数据
```

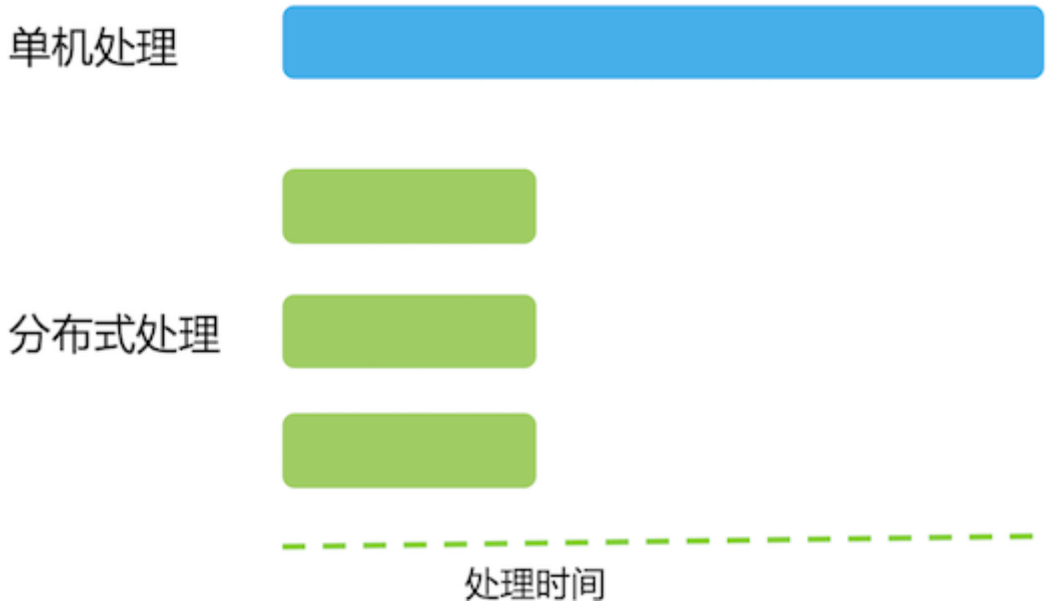
最终的目的：Spark 中的同一个 stage 中的多个 Task 处理的数据量大小几乎是一致的。

3. 什么是数据倾斜

对 Hadoop Spark Flink 这样的大数据系统来讲，**数据量大并不可怕，可怕的是数据倾斜。**

何谓数据倾斜？数据倾斜指的是，并行处理的数据集中，某一部分（如Spark 或 Kafka 的一个 Partition）的数据显著多于其它部分，从而使得该部分的处理速度成为整个数据集处理的瓶颈。

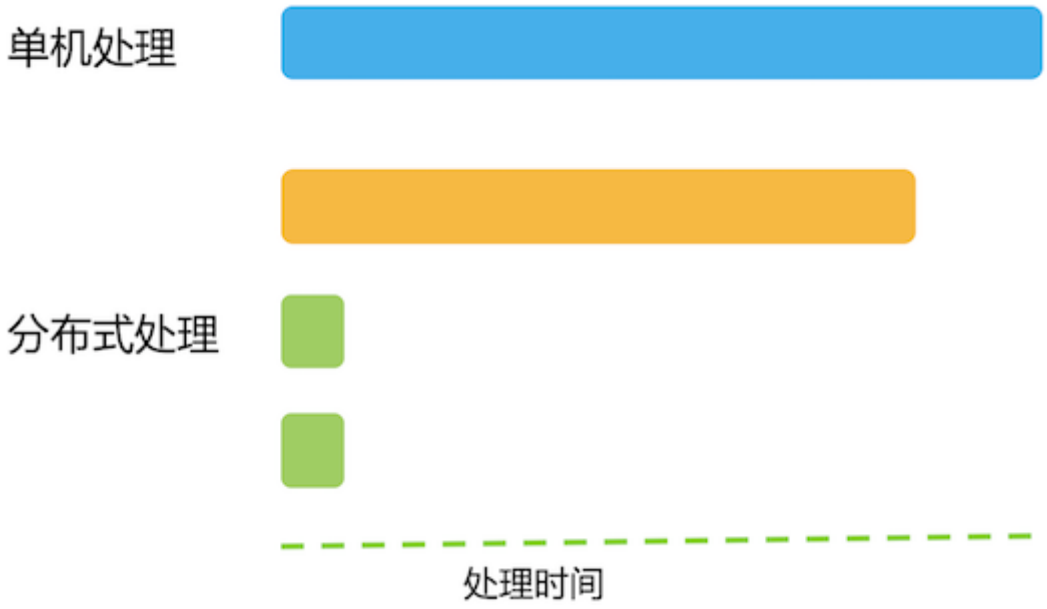
对于分布式系统而言，理想情况下，随着系统规模（节点数量）的增加，应用整体耗时线性下降。如果一台机器处理一批大量数据需要120 分钟，当机器数量增加到三时，理想的耗时为 120 / 3 = 40 分钟，如下图所示



原来是均匀的分成3份：
现在按照年龄阶段来分： 18 17 19 80%=18 10%=17 10% 19 100份

但是，上述情况只是理想情况，实际上将单机任务转换成分布式任务后，会有 overhead，使得总的任务量较之单机时有所增加，所以每台机器的执行时间加起来比单台机器时更大。这里暂不考虑这些 overhead，假设单机任务转换成分布式任务后，总任务量不变。

但即使如此，想做到分布式情况下每台机器执行时间是单机时的 `1/N`，就必须保证每台机器的任务量相等。不幸的是，很多时候，任务的分配是不均匀的，甚至不均匀到大部分任务被分配到个别机器上，其它大部分机器所分配的任务量只占总得的小部分。比如一台机器负责处理 80% 的任务，另外两台机器各处理 10% 的任务，如下图所示



在上图中，机器数据增加为 3 倍，预期执行时间应该为原来的 1/3，但执行时间只降为原来的 80%，远低于理想值。

3.1. 数据倾斜发生的现象

- 绝大多数 task 执行得都非常快，但个别 task 执行极慢。比如，总共有 1000 个 task，997 个 task 都在 1 分钟之内执行完了，但是剩余两三个 task 却要一两个小时。这种情况很常见。
- 原本能够正常执行的 Spark 作业，某天突然报出 OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

Tasks (31)

Index	ID	Attempt	Status	Locality Level	Executor ID / Host		Launch Time	Duration	GC Time	Input Size / Records
6	6	0	SUCCESS	NODE_LOCAL	1 / ambari4	stdout stderr	2017/10/16 17:36:04	4.4 min	1 s	25.3 MB / 570232449
14	9	0	SUCCESS	NODE_LOCAL	10 / ambari7	stdout stderr	2017/10/16 17:36:04	13 s	0.3 s	128.1 MB / 8388608
8	4	0	SUCCESS	NODE_LOCAL	12 / ambari8	stdout stderr	2017/10/16 17:36:04	11 s	0.3 s	128.1 MB / 8388608
23	16	0	SUCCESS	NODE_LOCAL	12 / ambari8	stdout stderr	2017/10/16 17:36:04	10 s	0.3 s	128.1 MB / 8388608
0	2	0	SUCCESS	NODE_LOCAL	2 / ambari6	stdout stderr	2017/10/16 17:36:04	10 s	0.3 s	128.1 MB / 8388609
17	22	0	SUCCESS	NODE_LOCAL	11 / ambari5	stdout stderr	2017/10/16 17:36:04	7 s	0.3 s	128.1 MB / 8388608
7	8	0	SUCCESS	NODE_LOCAL	6 / ambari12	stdout stderr	2017/10/16 17:36:04	7 s	0.2 s	128.1 MB / 8388609
1	0	0	SUCCESS	NODE_LOCAL	4 / ambari1	stdout stderr	2017/10/16 17:36:03	7 s	0.3 s	128.1 MB / 8388608
25	24	0	SUCCESS	NODE_LOCAL	4 / ambari1	stdout stderr	2017/10/16 17:36:04	7 s	0.3 s	128.1 MB / 8388609
19	17	0	SUCCESS	NODE_LOCAL	9 / ambari9	stdout stderr	2017/10/16 17:36:04	7 s	0.3 s	128.1 MB / 8388609

总结：

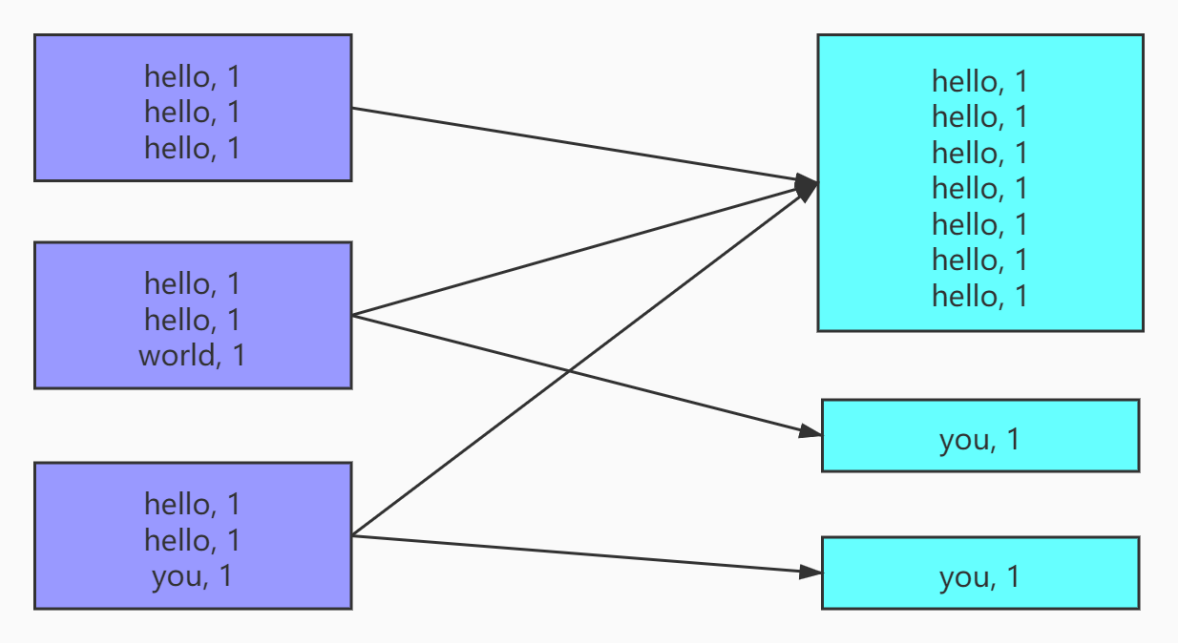
> 1、大部分任务都很快执行完，用时也相差无几，但个别Task执行耗时很长，整个应用程序一直处于99%左右的状态。
> 2、一直运行正常的Spark Application昨晚突然OOM了。

3.2. 数据倾斜发生的原理

数据倾斜的原理很简单：在进行 shuffle 的时候，必须将各个节点上相同的 key 的数据拉取到某个节点上的一个 task 来进行处理，比如按照 key 进行聚合或 join 等操作。此时如果某个 key 对应的数据量特别大的话，就会发生数据倾斜。比如大部分 key 对应 10 条数据，但是个别 key 却对应了 100 万条数据，那么大部分 task 可能就只会分配到 10 条数据，然后 1 秒钟就运行完了；但是个别 task 可能分配到了 100 万数据，要运行一两个小时。因此，整个 Spark 作业的运行进度是由运行时间最长的那个 task 决定的。

因此出现数据倾斜的时候，Spark 作业看起来会运行得非常缓慢，甚至可能因为某个 task 处理的数据量过大导致内存溢出。

下图就是一个很清晰的例子：hello 这个 key，在三个节点上对应了总共 7 条数据，这些数据都会被拉取到同一个task中进行处理；而world 和 you 这两个 key 分别才对应 1 条数据，所以另外两个 task 只要分别处理 1 条数据即可。此时第一个 task 的运行时间可能是另外两个 task 的 7 倍，而整个 stage 的运行速度也由运行最慢的那个 task 所决定。



总结：数据倾斜发生的本质，就是在执行多阶段的计算的时候，中间的shuffle策略可能导致分发到下游Task的数据量不均匀，进而导致下游Task执行时长的不一致。不完全均匀是正常的，但是如果相差太大，那么就产生性能问题了。

3.3. 数据倾斜的危害

从上图可见，当出现数据倾斜时，小量任务耗时远高于其它任务，从而使得整体耗时过大，未能充分发挥分布式系统的并行计算优势。另外，当发生数据倾斜时，少量部分任务处理的数据量过大，可能造成内存不足使得任务失败，并进而引进整个应用失败。如果应用并没有因此失败，但是大量正常任务都早早完成处于等待状态，资源得不到充分利用。

总结：

```
1、整体耗时过大（整个任务的完成由执行时间最长的那个Task决定）
2、应用程序可能异常退出（某个Task执行时处理的数据量远远大于正常节点，则需要的资源容易出现瓶颈，当资源不足，则应用程序退出）
3、资源闲置（处理等待状态的Task资源得不到及时的释放，处于闲置浪费状态）
```

3.4. 数据倾斜是如何造成的

在 Spark 中，同一个 Stage 的不同 Partition 可以并行处理，而具有依赖关系的不同 Stage 之间是串行处理的。假设某个 Spark Job 分为Stage0 和 Stage1 两个 Stage，且 Stage1 依赖于 Stage0，那 Stage0 完全处理结束之前不会处理 Stage1。而 Stage0 可能包含 N 个Task，这 N 个 Task 可以并行进行。如果其中 N-1 个 Task 都在 10 秒内完成，而另外一个 Task 却耗时 1 分钟，那该 Stage 的总时间至少为 1 分钟。换句话说，一个 Stage 所耗费的时间，主要由最慢的那个 Task 决定。

由于同一个 Stage 内的所有 Task 执行相同的计算，在排除不同计算节点计算能力差异的前提下，不同 Task 之间耗时的差异主要由该 Task 所处理的数据量决定。

Stage 的数据来源主要分为如下两类：

```
1、数据源本身分布有问题：从数据源直接读取。如读取HDFS，Kafka，有可能出现，大概率不会
2、自己指定的分区规则：读取上一个 Stage 的 Shuffle 数据
```

朴素的分布式计算的核心思想：

```
1、大问题拆分成小问题：分而治之

2、既然要分开算，那最后就一定要把分开计算的那么多的小 Task 的结果执行汇总

3、所以必然分布式计算引擎的设计中，应用程序的执行一定是分阶段

4、分布计算引擎的而核心：一个复杂的分布式计算应用程序的执行肯定要分成多个阶段，每个阶段分布式并行运行多个Task

5、DAG引擎：
Spark:   stage1 ==> stage2 ===> stage3
mapreduce:  就只有两个阶段：mapper  reducer
阶段与阶段之间需要进行 shuffle，只要进行了数据混洗，就存在着数据分发不均匀的情况。如果情况严重，就是数据倾斜。
```

分布式计算引擎的设计，免不了有shuffle，既然有shuffle操作，就一定有产生数据倾斜的可能。如果你是做大数据处理的，就一定会遇到 数据倾斜！

3.5. 如何消除或缓解数据倾斜

大数据场景最经典的数据来源： HDFS Kafka HBase

3.5.1. 避免数据源倾斜-HDFS

Spark 以通过 `textFile(path, minPartitions)` 方法读取文件时，使用 TextInputFormat。

对于不可切分的文件，每个文件对应一个 Split 从而对应一个 Partition。此时各文件大小是否一致，很大程度上决定了是否存在数据源侧的数据倾斜。另外，对于不可切分的压缩文件，即使压缩后的文件大小一致，它所包含的实际数据量也可能差别很多，因为源文件数据重复度越高，压缩比越高。反过来，即使压缩文件大小接近，但由于压缩比可能差距很大，所需处理的数据量差距也可能很大。

此时可通过在数据生成端将不可切分文件存储为可切分文件，或者保证各文件包含数据量相同的方式避免数据倾斜。

总结：

```
对于不可切分文件可能出现数据倾斜，对于可切分文件，一般来说，不存在数据倾斜问题。
1、可切分：  基本上不会！  默认数据块大小：128M
2、不可切分：  源文件不均匀，最终导致  分布式引用程序计算产生数据倾斜
      日志：每一个小时生成一个日志文件
```

3.5.2. 避免数据源倾斜-Kafka

```
Topic 主题：  分布式的组织形式：  分区，  既然要进行数据分区，那就有可能产生数据分布不均匀
```

以 Spark Stream 通过 DirectStream 方式读取 Kafka 数据为例。由于 Kafka 的每一个 Partition 对应 Spark 的一个 Task（Partition），所以 Kafka 内相关 Topic 的各 Partition 之间数据是否平衡，直接决定 Spark 处理该数据时是否会产生数据倾斜。

Kafka 某一 Topic 内消息在不同 Partition 之间的分布，主要由 Producer 端所使用的 Partitioner 实现类决定。如果使用随机 Partitioner，则每条消息会随机发送到一个 Partition 中，从而从概率上来讲，各 Partition 间的数据会达到平衡。此时源 Stage（直接读取 Kafka 数据的 Stage）不会产生数据倾斜。

但很多时候，业务场景可能会要求将具备同一特征的数据顺序消费，此时就需要将具有相同特征的数据放于同一个 Partition 中。一个典型的场景是，需要将同一个用户相关的PV信息置于同一个 Partition 中。此时，如果产生了数据倾斜，则需要通过其它方式处理。

总结：

```
1、Kafka不是计算引擎，只是一个用来在流式项目架构中起削峰填谷作用的消息中转平台，所以为保证一个Topic的分布式平衡，尽量不要使用Hash散列或者是跟业务有关的自定义分区规则等方式来进行数据分区，否则会造成下游消费者一开始就产生了数据倾斜。
2、Kafka尽量使用随机，轮询等不会造成数据倾斜的数据分区规则
```

3.5.3. 定位处理逻辑 - Stage 和 Task

归根结底，数据倾斜产生的原因，就是两个 stage 中的 shuffle 过程导致的。所以我们只需要研究 Shuffle 算子即可。

我们知道了导致数据倾斜的问题就是 shuffle 算子，所以我们先去找到代码中的 shuffle 的算子，比如 distinct、groupByKey、reduceByKey、aggergateByKey、join、cogroup、repartition 等，那么问题一定就出现在这里。

spark的执行，按照hsuffle算子分成多个stage来执行。

总结：

```
如果 Spark Application 运行过程中，出现数据倾斜，可以通过 web 管理监控界面，查看 各stage 的运行情况，如果某一个 stage 的运行很长，并且这个 stage 的大部分Task都运行很快，则
```

3.5.4. 查看导致倾斜的key的数据分布情况

知道了数据倾斜发生在哪里之后，通常需要分析一下那个执行了shuffle操作并且导致了数据倾斜的 RDD/Hive表，查看一下其中key的分布情况。这主要是为之后选择哪一种技术方案提供依据。针对不同的key分布与不同的shuffle算子组合起来的各种情况，可能需要选择不同的技术方案来解决。

此时根据你执行操作的情况不同，可以有很多种查看key分布的方式：

```
1、如果是Spark SQL中的group by、join语句导致的数据倾斜，那么就查询一下 SQL 中使用的表的key分布情况。
2、如果是对 Spark RDD执行shuffle算子导致的数据倾斜，那么可以在Spark作业中加入查看 key 分布的代码，比如 RDD.countByKey()。然后对统计出来的各个key出现的次数，collect/take到客户端打印一下，就可以看到key的分布情况。
```

举例来说，对于上面所说的单词计数程序，如果确定了是 stage1 的 reduceByKey 算子导致了数据倾斜，那么就应该看看进行 reduceByKey 操作的 RDD 中的 key 分布情况，在这个例子中指的就是 pairs RDD。如下示例，我们可以先对 pairs 采样 10% 的样本数据，然后使用 countByKey 算子统计出每个 key 出现的次数，最后在客户端遍历和打印样本数据中各个 key 的出现次数。

```
val sampledPairs = pairs.sample(false, 0.1)
val sampledWordCounts = sampledPairs.countByKey()
sampledWordCounts.foreach(println(_))
```

采样！(离线处理：无放回采样， 流式处理：鱼塘采样)

4. 数据倾斜解决方案

4.1. 方案一：使用Hive ETL预处理数据

4.1.1. 适用场景

导致数据倾斜的是 Hive 表。如果该 Hive 表中的数据本身很不均匀（比如某个 key 对应了 100 万数据，其他 key 才对应了 10 条数据），而且业务场景需要频繁使用 Spark 对 Hive 表执行某个分析操作，那么比较适合使用这种技术方案。

4.1.2. 实现思路

此时可以评估一下，是否可以通过Hive来进行数据预处理（即通过 Hive ETL 预先对数据按照 key 进行聚合，或者是预先和其他表进行join），然后在 Spark 作业中针对的数据源就不是原来的 Hive 表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在 Spark 作业中也就不需要使用原先的 shuffle 类算子执行这类操作了。

4.1.3. 实现原理

这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

4.1.4. 方案优缺点

优点：

> 实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

缺点：

> 治标不治本，Hive ETL中还是会发生数据倾斜。

4.1.5. 企业最佳实践

在一些 Java 系统与 Spark 结合使用的项目中，会出现 Java 代码频繁调用 Spark 作业的场景，而且对 Spark 作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的 Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次 Java 调用 Spark作业时，执行速度都会很快，能够提供更好的用户体验。

在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过 Java Web 系统提交数据分析统计任务，后端通过Java 提交 Spark作业进行数据分析统计。要求 Spark 作业速度必须要快，尽量在10 分钟以内，否则速度太慢，用户体验会很差。所以我们将有些 Spark 作业的shuffle操作提前到了Hive ETL中，从而让 Spark 直接使用预处理的 Hive 中间表，尽可能地减少 Spark 的 shuffle操作，大幅度提升了性能，将部分作业的性能提升了6倍以上。

4.2. 方案二：调整shuffle操作的并行度

4.2.1. 适用场景

大量不同的Key被分配到了相同的Task造成该Task数据量过大。

如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。但是也是一种属于碰运气的方案。因为这种方案，并不能让你一定解决数据倾斜，甚至有可能加重。那当然，总归，你会调整到一个合适的并行度是能解决的。前提是这种方案适用于 Hash散列的分区方式。凑巧的是，各种分布式计算引擎，比如MapReduce，Spark 等默认都是使用 Hash散列的方式来进行数据分区。

Spark 在做 Shuffle 时，默认使用 HashPartitioner（非Hash Shuffle）对数据进行分区。如果并行度设置的不合适，可能造成大量不相同的 Key 对应的数据被分配到了同一个 Task 上，造成该 Task 所处理的数据远大于其它 Task，从而造成数据倾斜。

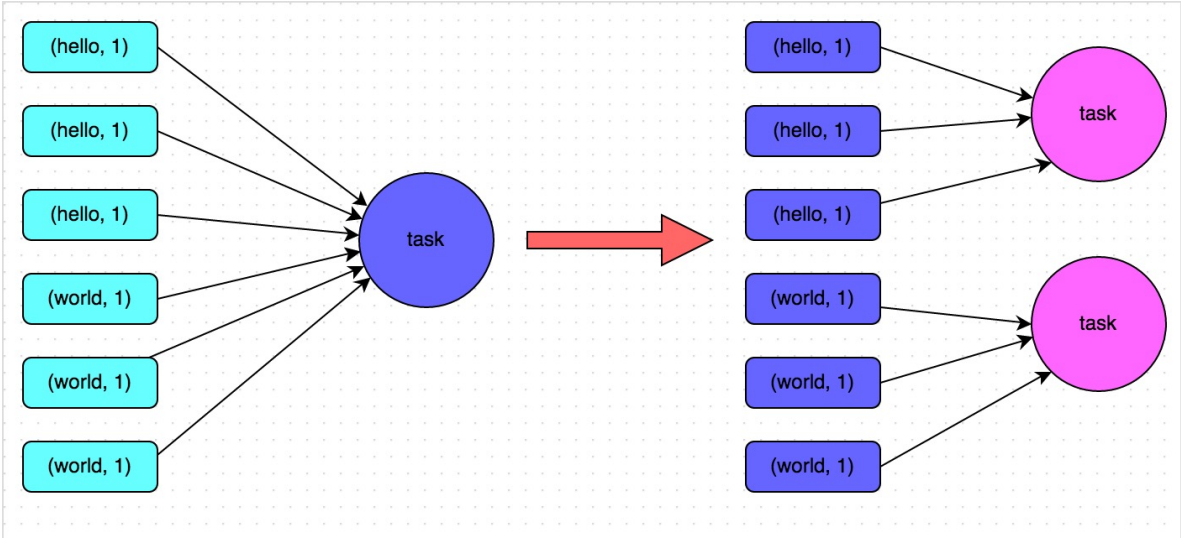
如果调整 Shuffle 时的并行度，使得原本被分配到同一 Task 的不同 Key 发配到不同 Task 上处理，则可降低原 Task 所需处理的数据量，从而缓解数据倾斜问题造成的短板效应。

4.2.2. 实现思路

在对 RDD 执行 Shuffle 算子时，给 Shuffle 算子传入一个参数，比如 reduceByKey(1000)，该参数就设置了这个 shuffle 算子执行时shuffle read task 的数量。对于 Spark SQL 中的 Shuffle 类语句，比如 group by、join 等，需要设置一个参数，即 spark.sql.shuffle.partitions，该参数代表了 shuffle readTask 的并行度，该值默认是 200，对于很多场景来说都有点过小。

4.2.3. 实现原理

增加 shuffle read task 的数量，可以让原本分配给一个 task 的多个 key 分配给多个 task，从而让每个 task 处理比原来更少的数据。举例来说，如果原本有 5 个key，每个 key 对应 10 条数据，这 5 个 key 都是分配给一个 task 的，那么这个 task 就要处理 50 条数据。而增加了 shuffle read task 以后，每个 task 就分配到一个 key，即每个 task 就处理 10 条数据，那么自然每个 task 的执行时间都会变短了。具体原理如下图所示。



一句话总结：调整并行度分散同一个 Task的不同 Key，之前由于运气比较差，多个数据比较多的 key 都分布式在同一个 Task 上，如果调整了并行度，极大可能会让这些 key 分布式到不同的 Task，有效缓解数据倾斜。

4.2.4. 方案优缺点

优点：

> 实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。实现简单，可在需要Shuffle的操作算子上直接设置并行度或者使用spark.default.parallelism设置。如果是Spark SQL，还可通过SET spark.sql.shuffle.partitions=[num_tasks]设置并行度。可用最小的代价解决问题。一般如果出现数据倾斜，都可以通过这种方法先试验几次，如果问题未解决，再尝试其它方法。

缺点：

> 只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。适用场景少，只能将分配到同一Task的不同Key分散开，但对于同一Key倾斜严重的情况该方法并不适用。并且该方法一般只能缓解数据倾斜，没有彻底消除问题。从实践经验来看，其效果一般。

4.2.5. 企业最佳实践

该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用嘴简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

如果之前的并行度是12，现在调整成为 18 有用么？并没有多大的改善　10个 11个 13

4.3. 方案三：过滤少数导致倾斜的key

4.3.1. 适用场景

如果发现导致倾斜的 key 就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如 99% 的 key 就对应 10 条数据，但是只有一个 key 对应了 100 万数据，从而导致了数据倾斜。

4.3.2. 实现思路

如果我们判断那少数几个数据量特别多的 key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个 key。比如，在 Spark SQL 中可以使用 where 子句过滤掉这些 key 或者在 SparkCore 中对 RDD 执行 filter 算子过滤掉这些 key。如果需要每次作业执行时，动态判定哪些 key 的数据量最多然后再进行过滤，那么可以使用 sample 算子对 RDD 进行采样，然后计算出每个 key 的数量，取数据量最多的 key 过滤掉即可。

4.3.3. 实现原理

将导致数据倾斜的 key 给过滤掉之后，这些 key 就不会参与计算了，自然不可能产生数据倾斜。

4.3.4. 方案优缺点

优点：

```
实现简单，而且效果也很好，可以完全规避掉数据倾斜。
```

缺点：

```
适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。
```

4.3.5. 企业最佳实践

在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天 Spark 作业在运行的时候突然 OOM 了，追查之后发现，是 Hive 表中的某一个 key 在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个 key 之后，直接在程序中将那些key给过滤掉。

4.4. 方案四：将reduce join转为map join

4.4.1. 适用场景

在对 RDD 使用 join 类操作，或者是在 Spark SQL 中使用 join 语句时，而且 join 操作中的一个 RDD 或表的数据量比较小（比如几百M或者一两G），比较适用此方案。

在分布式计算引擎中，实现Join的思路有两种：

1、MapJoin，顾名思义，Join逻辑的完成是在 Mapper 阶段就完成了，这是假定执行的是 MapReduce 任务，如果是 Spark任务，表示只用一个 Stage 就执行完了 Join 操作。

```
优点：避免了两阶段之间的shuffle，效率高，没有shuffle也就没有了倾斜。
缺点：多使用内存资源，只适合大小表做join的场景
```

2、ReduceJoin，顾名思义，Join逻辑的完成是在 Reducer 阶段完成的。那么如果是MapReduce任务，则表示 Maper阶段执行完之后把数据 Shuffle到 Reducer阶段来执行 Join 逻辑，那么就可能导致数据倾斜。如果是 Spark任务，意味着，上一个stage的执行结果数据shuffle到 下一个stage中来完成 Join 操作，同样也可能产生数据倾斜。

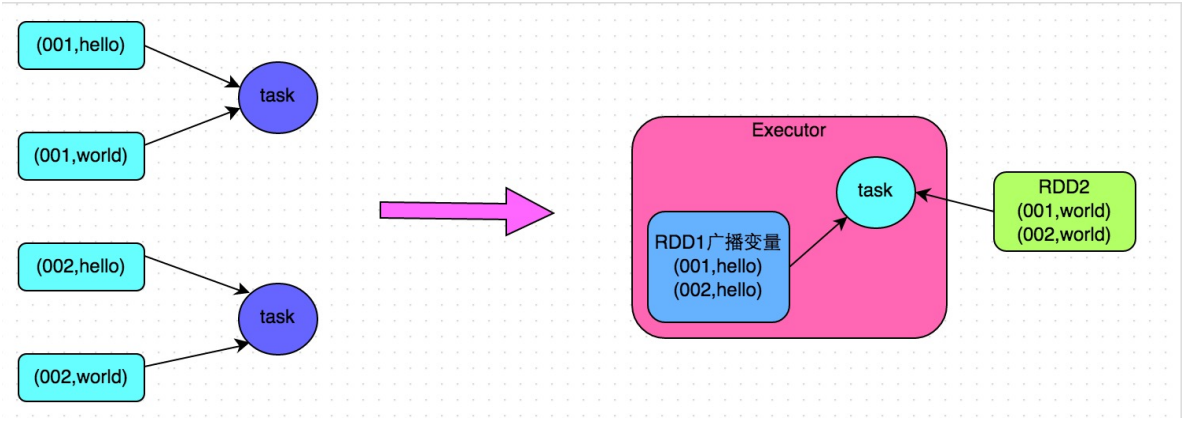
```
优点：这是一种通用的join，在不产生数据倾斜的情况下，能完成各种类型的join
缺点：会发生数据倾斜的情况
```

4.4.2. 实现思路

不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

4.4.3. 实现原理

普通的 join 是会走 shuffle 过程的，而一旦 shuffle，就相当于会将相同 key 的数据拉取到一个 shuffle read task 中再进行 join，此时就是 reduce join。但是如果一个 RDD 是比较小的，则可以采用广播小 RDD 全量数据 +map 算子来实现与 join 同样的效果，也就是 map join，此时就不会发生 shuffle 操作，也就不会发生数据倾斜。具体原理如下图所示。



4.4.4. 方案优缺点

优点：

> 对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜。

缺点：

> 适用场景较少，因为这个方案只适用于一个大表和一个小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver 和每个Executor 内存中都会驻留一份小 RDD 的全量数据。如果我们广播出去的 RDD 数据比较大，比如 10G 以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。

4.5. 方案五：采样倾斜 key 并分拆 join 操作

4.5.1. 适用场景

两个 RDD/Hive 表进行 join 的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个 RDD/Hive 表中的 key 分布情况。如果出现数据倾斜，是因为其中某一个 RDD/Hive 表中的少数几个 key 的数据量过大，而另一个 RDD/Hive 表中的所有 key 都分布比较均匀，那么采用这个解决方案是比较合适的。

4.5.2. 实现思路

思路：

1、对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key。
2、然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀，而不会导致倾斜的大部分key形成另外一个RDD。
3、接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个单独的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀，不会导致倾斜的大部分key也形成另外一个RDD。
4、再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了。
5、而另外两个普通的RDD就照常join即可。
6、最后将两次join的结果使用union算子合并起来即可，就是最终的join结果。

4.5.3. 实现原理

对于 join 导致的数据倾斜，如果只是某几个 key 导致了倾斜，可以将少数几个 key 分拆成独立 RDD，并附加随机前缀打散成 n 份去进行join，此时这几个 key 对应的数据就不会集中在少数几个 task 上，而是分散到多个 task 进行 join 了。具体原理见下图。

4.5.4. 方案优缺点

优点：

对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

缺点：

如果导致倾斜的key特别多的话，比如成千上万个key都导致数据倾斜，那么这种方式也不适合。

4.6. 方案六：两阶段聚合（局部聚合+全局聚合）

4.6.1. 适用场景

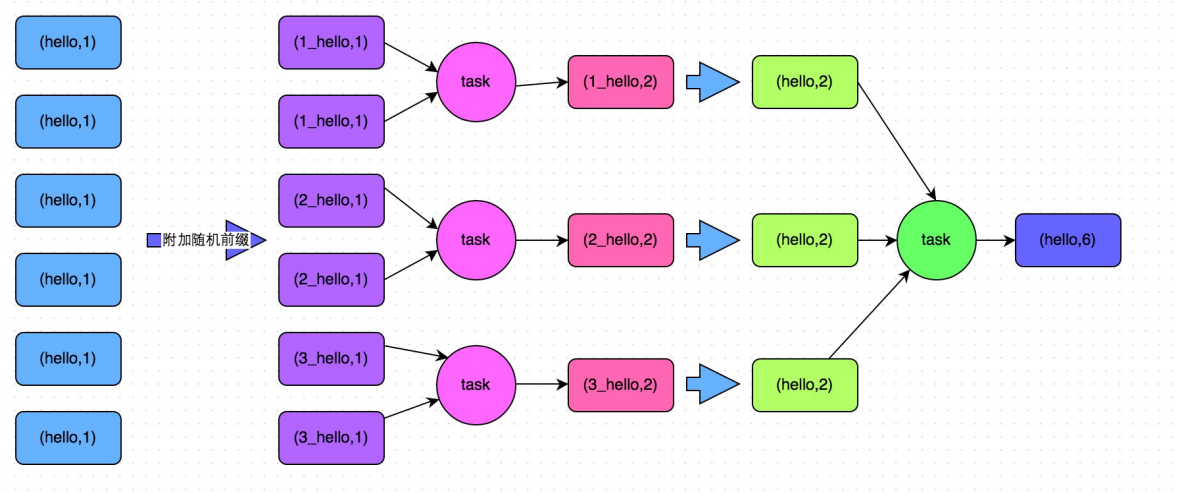
对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案。

4.6.2. 实现思路

这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1_hello, 1) (1_hello, 1) (2_hello, 1) (2_hello, 1)。接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1_hello, 2) (2_hello, 2)。然后将各个key的前缀给去掉，就会变成(hello,2)(hello,2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

4.6.3. 实现原理

将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。



4.6.4. 方案优缺点

优点：

> 对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上。

缺点：

> 仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案。

4.7. 方案七：使用随机前缀和扩容 RDD 进行 join

4.7.1. 适用场景

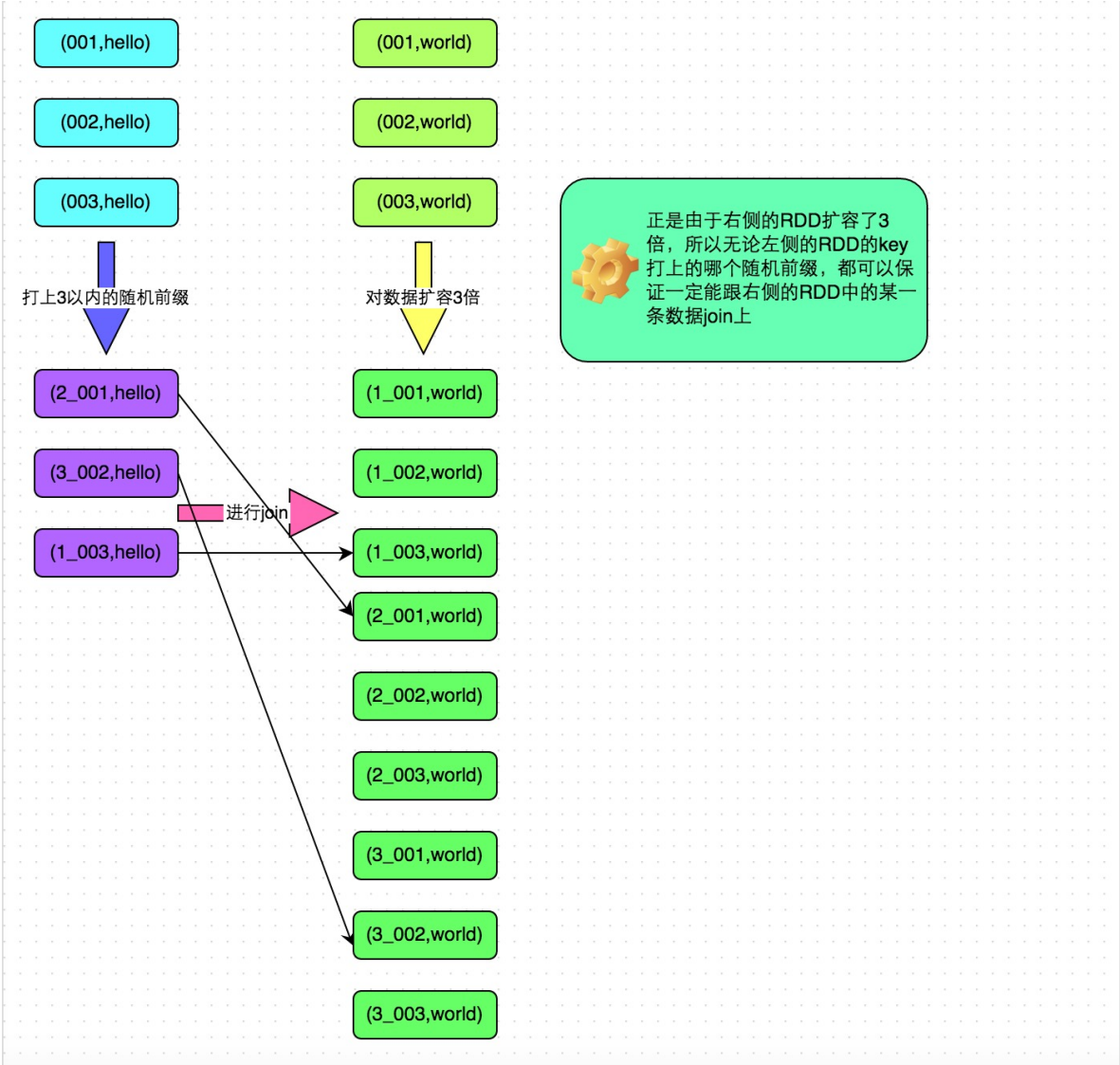
如果在进行 join 操作时，RDD 中有大量的 key 导致数据倾斜，那么进行分拆 key 也没什么意义。

4.7.2. 实现思路

> 1、该方案的实现思路基本和“解决方案五”类似，首先查看 RDD/Hive 表中的数据分布情况，找到那个造成数据倾斜的 RDD/Hive 表，比如有多个key 都对应了超过1万条数据。
> 2、然后将该RDD的每条数据都打上一个n以内的随机前缀。
> 3、同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。
> 4、最后将两个处理后的RDD进行join即可。

4.7.3. 实现原理

将原先一样的 key 通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。



4.7.4. 方案优缺点

优点：

> 对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

缺点：

> 该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。

4.7.5. 企业最佳实践

曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。

4.8. 方案八：任务横切，一分为二，单独处理

有时候，一个Spark应用程序中，导致倾斜的因素不是一个单一的，比如有一部分倾斜的因素是null，有一部分倾斜的因素是某些个key分布特别多。那么拆分出来也得使用不同的手段来处理

4.8.1. 适用场景

导致数据倾斜的因素比较多，比较复杂的场景中。

4.8.2. 实现思路

在了解清楚数据的分布规律，以及确定了数据倾斜是由何种原因导致的，那么按照这些原因，进行数据的拆分，然后单独处理每个部分的数据，最后把结果合起来。

4.8.3. 实现原理

方案六其实是一种纵切，方案八就是一种横切。原理同思路。

4.8.4. 方案优缺点

优点：

> 将多种简单的方案综合起来，解决一个复杂的问题。可以算上一种万能的方案。

缺点：

> 确定数据倾斜的因素比较复杂，导致解决该数据倾斜的方案比较难实现落地。代码复杂度也较高。

4.9. 方案九：多种方案组合使用

在实践中发现，很多情况下，如果只是处理较为简单的数据倾斜场景，那么使用上述方案中的某一种基本就可以解决。但是如果要处理一个较为复杂的数据倾斜场景，那么可能需要将多种方案组合起来使用。比如说，我们针对出现了多个数据倾斜环节的Spark作业，可以先运用解决方案一和二，预处理一部分数据，并过滤一部分数据来缓解；其次可以对某些shuffle操作提升并行度，优化其性能；最后还可以针对不同的聚合或join操作，选择一种方案来优化其性能。大家需要对这些方案的思路和原理都透彻理解之后，在实践中根据各种不同的情况，灵活运用多种方案，来解决自己的数据倾斜问题。

如果这多种方案，组合使用也不行，请看最后一招：自定义分区规则

4.10. 方案十：自定义 Partitioner

使用自定义的 Partitioner 实现类代替默认的 HashPartitioner，尽量将所有不同的 Key 均匀分配到不同的 Task中。

4.10.1. 适用场景

大量不同的Key被分配到了相同的Task造成该Task数据量过大。

4.10.2. 实现思路

先通过抽样，了解数据的 key 的分布规律，然后根据规律，去定制自己的数据分区规则，尽量保证所有的 Task 的数据量相差无几。

4.10.3. 实现原理

使用自定义的 Partitioner（默认为HashPartitioner），将原本被分配到同一个 Task 的不同 Key 分配到不同 Task。

4.10.4. 常见分区解释

总结几种常见的数据分区方式：

第一种：随机分区

```
优点：数据分布均匀
缺点：具有相同特点的数据不会保证被分配到相同的分区
```

第二种：轮询分区

```
优点：确保一定不会出现数据倾斜
缺点：无法根据存储/计算能力分配存储/计算压力
```

第三种：Hash散列

```
优点：具有相同特点的数据保证被分配到相同的分区
缺点：极容易产生数据倾斜
```

第四种：范围分区

```
缺点：相邻的数据都在相同的分区
缺点：部分分区的数据量会超出其他的分区，需要进行裂变以保持所有分区的数据量是均匀的，如果每个分区不排序，那么裂变就会非常困难
```

4.10.5. 方案优缺点

优点：

```
灵活，通用。
```

缺点：

```
必须根据对应的场景设计合理的分区方案。没有现成的方案可用，需临时实现。
```

5. Spark倾斜调优案例

面试用这个，基本上能征服面试官！

5.1. 方案十一：Spark 整合 BitMap 求 Join

解决方案：位图！

如果是两张特大宽表做 Join 怎么办？前面有讲过，如果是大小表做 Join，使用 MapJoin 轻松快速高效解决，如果是大大表呢？

```
1、把其中一个较小的表拆分成多个小表，执行多次大小表链接
2、SortMergeJoin
2、位图法
```

5.1.1. 适用场景

两张大款表做 Join，典型需求：最近7天都登陆过的用户有哪些?

神奇的需求： 假设微信想统计过去7天理，每天都发了朋友圈的用户有哪些?

```
微信的总用户量：10E  每天活跃用户： 3E
过去第一天：3E用户发了 3E条朋友圈   A
过去第二天：3E用户发了 3E条朋友圈   B
过去第三天：4E用户发了 4E条朋友圈   C
过去第四天：2E用户发了 2E条朋友圈   D
过去第五天：3E用户发了 3E条朋友圈   E
过去第六天：3E用户发了 3E条朋友圈   F
过去第七天：3E用户发了 3E条朋友圈   G
```

SQL实现：

```
select a.id from A a join B b on a.id = b.id
                                  join C c on a.id = c.id
                                 ....
                                 join G g on a.id = g.id;
```

位图!

5.1.2. 实现思路

不是有 10E 用户么： 维护一个位数组：10E个位 每个位 要么是1 要么 0

5.1.3. 实现原理

5.1.4. 方案优缺点

优点：

> 占用内存少，处理速度高，

缺点：

> 维护 BitMap 要求高。