1. 什么是数据倾斜

数据倾斜是一种很常见的问题（依据二八定律），简单来说，比方WordCount中某个Key对应的数据量非常大的话，就会产生数据倾斜，导致两个后果：

1. OOM（out of memory，单或少数的节点）；
2. 拖慢整个Job执行时间（其他已经完成的节点都在等这个还在做的节点）。

2. 解决数据倾斜需要

1. 搞定 Shuffle；
2. 搞定业务场景；
3. 搞定 CPU core 的使用情况；
4. 搞定 OOM 的根本原因等：一般都因为数据倾斜（某task任务的数据量过大，GC压力大，和Kafka不同在于Kafka的内存不经过JVM，其基于Linux的Page）。

3. 导致Spark数据倾斜的本质

Shuffle时，需将各节点的相同key的数据拉取到某节点上的一个task来处理，若某个key对应的数据量很大就会发生数据倾斜。比方说大部分key对应10条数据，某key对应10万条，大部分task只会被分配10条数据，很快做完，个别task分配10万条数据，不仅运行时间长，且整个stage的作业时间由最慢的task决定。

数据倾斜只会发生在**Shuffle**过程，以下算法可能触发Shuffle操作： **distinct**、**groupByKey**、**reduceByKey**、**aggregateByKey**、**join**、**cogroup**、**repartition**等。

4. 定位最慢的Task所处的源码位置

步骤一： 看数据倾斜发生在哪个**stage**（也就是看以上算子出现在哪个阶段）。yarn-client模式下查看本地log或Spark Web UI中当前运行的是哪个stage；yarn-cluster模式下，通过Spark Web UI查看运行到了哪个Stage。 主要看最慢的Stage各task分配的数据量，来确定是否是数据倾斜。

步骤二：根据**Stage**划分，推算倾斜发生的代码（必然有**Shuffle**类算子）。简单实用方法：只要看到shuffle类算子或Spark SQL的SQL语句会有Shuffle类的算子的句子，就可以该地方划分为前后两个Stage。（之前用Python的PySpark接口，Spark Web UI会查看task在源码中的行数，Java或者Scala虽没用过，但我想应该有）

5. 解决方案

方案一：使用Hive ETL预处理

- 场景：若Hive表中数据不均匀，且业务中会频繁用Spark对Hive表分析；
- 思路：用Hive对数据预处理（对key聚合等操作），原本是Spark对Hive的原表操作，现在就是对Hive预处理后的表操作；

- 原理：从根源解决了数据倾斜，规避了了Spark进行Shuffle类算子操作。但Hive ETL中进行聚合等操作会发生数据倾斜，只是把慢转移给了Hive ETL；
- 优点：方便，效果好，规避了Spark数据倾斜；
- 缺点：治标不治本，Hive ETL会数据倾斜。

方案二：过滤导致倾斜的**key**

- 场景：发生倾斜的key很少且不重要；
- 思路：对发生倾斜的key过滤掉。比方在Spark SQL中用where子句或filter过滤，若每次作业执行，需要动态判定可使用sample算子对RDD采样后取数据量最多的key过滤；
- 原理：对倾斜的key过滤后，这些key便不会参与后面的计算，从本质上消除数据倾斜；
- 优点：简单，效果明显；
- 缺点：适用场景少，实际中导致倾斜的key很多。

方案三：提高**Shuffle**操作并行度

- 场景：任何场景都可以，优先选择的最简单方案；
- 思路：对RDD操作的Shuffle算子传入一个参数，也就是设置Shuffle算子执行时的Shuffle read task数量。对于Spark SQL的Shuffle类语句（如group by，join）即spark.sql.shuffle.partitions，代表shuffle read task的并行度，默认值是200可修改；
- 原理：增大shuffle read task参数值，让每个task处理比原来更少的数据；
- 优点：简单，有效；
- 缺点：缓解的效果很有限。

方案四：两阶段聚合（局部聚合**+**全局聚合）

- 场景：对RDD进行reduceByKey等聚合类shuffle算子，SparkSQL的groupBy做分组聚合这两种情况
- 思路：首先通过map给每个key打上n以内的随机数的前缀并进行局部聚合，即(hello, 1) (hello, 1) (hello, 1) (hello, 1)变为(1*hello, 1) (1*hello, 1) (2_hello, 1)，并进行reduceByKey的局部聚合，然后再次map将key的前缀随机数去掉再次进行全局聚合；
- 原理：对原本相同的key进行随机数附加，变成不同key，让原本一个task处理的数据分摊到多个task做局部聚合，规避单task数据过量。之后再去随机前缀进行全局聚合；
- 优点：效果非常好（对聚合类Shuffle操作的倾斜问题）；
- 缺点：范围窄（仅适用于聚合类的Shuffle操作，join类的Shuffle还需其它方案）。

方案五：将**reduce join**转为**map join**

- 场景：对RDD或Spark SQL使用join类操作或语句，且join操作的RDD或表比较小（百兆或1,2G）；
- 思路：使用broadcast和map类算子实现join的功能替代原本的join，彻底规避shuffle。对较小RDD直接collect到内存，并创建broadcast变量；并对另外一个RDD执行map类算子，在该算子的函数中，从broadcast变量（collect出的较小RDD）与当前RDD中的每条数据依次比对key，相同的key执行你需要方式的join；
- 原理：若RDD较小，可采用广播小的RDD，并对大的RDD进行map，来实现与join同样的效果。简而言之，用broadcast-map代替join，规避join带来的shuffle（无Shuffle无倾斜）；
- 优点：效果很好（对join操作导致的倾斜），根治；
- 缺点：适用场景小（大表+小表），广播（driver和executor节点都会驻留小表数据）小表也耗内存。

方案六：采样倾斜**key**并分拆**join**操作

- 场景：两个较大的（无法采用方案五）RDD/Hive表进行join时，且一个RDD/Hive表中少数key数据量过大，另一个RDD/Hive表的key分布较均匀（RDD中两者之一有一个更倾斜）；
- 思路：
  - 1. 对更倾斜rdd1进行采样（RDD.sample）并统计出数据量最大的几个key；
  - 2. 对这几个倾斜的key从原本rdd1中拆出形成一个单独的rdd1*1*，并打上*0~n*的随机数前缀，被拆分的原*rdd1*的另一部分（不包含倾斜*key*）又形成一个新*rdd1*2；
  - 3. 对rdd2过滤出rdd1倾斜的key，得到rdd2*1*，并将其中每条数据扩*n*倍，对每条数据按顺序附加*0~n*的前缀，被拆分出*key*的*rdd2*也独立形成另一个*rdd2*2； 【个人认为，这里扩了n倍，最后union完还需要将每个倾斜key对应的value减去(n-1)】
  - 4. 将加了随机前缀的rdd1*1*和*rdd2*1进行join（此时原本倾斜的key被打散n份并被分散到更多的task中进行join）； 【个人认为，这里应该做两次join，两次join中间有一个map去前缀】
  - 5. 另外两个普通的RDD（rdd12、*rdd2*2）照常join；
  - 6. 最后将两次join的结果用union结合得到最终的join结果。
- 原理：对join导致的倾斜是因为某几个key，可将原本RDD中的倾斜key拆分出原RDD得到新RDD，并以加随机前缀的方式打散n份做join，将倾斜key对应的大量数据分摊到更多task上来规避倾斜；
- 优点：前提是join导致的倾斜（某几个key倾斜），避免占用过多内存（只需对少数倾斜key扩容n倍）；
- 缺点：对过多倾斜key不适用。

方案七：用随机前缀和扩容**RDD**进行**join**

- 场景：RDD中有大量key导致倾斜；
- 思路：与方案六类似。 1. 查看RDD/Hive表中数据分布并找到造成倾斜的RDD/表； 2. 对倾斜RDD中的每条数据打上n以内的随机数前缀； 3. 对另外一个正常RDD的每条数据扩容n倍，扩容出的每条数据依次打上0到n的前缀； 4. 对处理后的两个RDD进行join。
- 原理：与方案六只有唯一不同在于这里对不倾斜RDD中所有数据进行扩大n倍，而不是找出倾斜key进行扩容（这是方案六）；
- 优点：对join类的数据倾斜都可处理，效果非常显著；
- 缺点：缓解，扩容需要大内存。 【个人认为，这里和方案六一样，也需要对扩容的key对应的value最后减去(n-1)，除非只需大小关系，对值没有要求】

方案八：多种方案组合

实际中，需综合着对业务全盘考虑，可先用方案一和二进行预处理，同时在需要Shuffle的操作提升Shuffle的并行度，最后针对数据分布选择后面方案中的一种或多种。实际中需要对数据和方案思路理解灵活应用。