总结

1、数仓分层的好处

对数据进行分层的一个主要原因就是希望管理数据的时候，对数据有一个更加清晰的掌控
1、清晰的数据结构：每一个数据分层都有他的作用，我们在使用表的时候可以根据表名来定位表所在的分层，然后根据表的位置来理解表的作用
2、数据血缘追踪：分层之后的表存在一定的依赖关系，当有表出现问题的时候，我们可以根据这张表的依赖关系向前溯源，查找问题
3、减少重复开发：规范数据分层，开发一些通用的中层数据，能够减少极大的数据计算
4、把复杂问题简单化：把复杂问题拆分成多个步骤，每一层只需要解决一个的步骤，易于开发和维护
5、屏蔽原始数据的影响：不必改一次业务就重新从源接入数据

2、什么样的数仓是一个好的数仓

具有清晰的分层结构，既要保证数据层的稳定又要屏蔽对下游的影响，而且要避免链路过长，最重要的是能够为业务发展提供稳定、准确的数据支撑。

3、缓慢变化维

定义：缓慢变化维就是变化相对缓慢的维度（相对于事实表）
1、保留原始值
　　如果某一个属性值绝对不会发生变化，直接将该维度属性代替维度键写入事实表中
2、改变属性值
　　如果用户不关心这个属性的变化，那么直接覆盖维度表中的数据，只保留最新的值
3、增加新行
　　增加行生效时间、行失效时间、以及行标识（标识行数据是否有效，0无效、1有效），除此之外还需要分配一个新的代理键作为事实表的外键
4、增加新维度属性
　　原属性存放上一个版本的属性值，然后在新属性列存放当前新的属性值
5、每日快照
　　每天保留全量维度数据，以空间换时间
6、创建拉链表

4、map join

1、定义：map join是hive的一种优化操作，适用于小表join大表的场景，由于表join的操作是在map端且在内存中进行的，所以其并不需要启动reduce任务，也就不需要经过shuffle，从而可以提高join效率
2、原理：map join会把小表读入内存，在map阶段直接拿另一个表的数据和内存中的数据做匹配，而普通的join在map阶段只做分组，然后shuffle落盘排序，最后在reduce阶段进行聚合。shuffle阶段由于落盘会导致非常耗时，而map join中没有落盘操作和reduce阶段，因此效率会高很多。
3、如何使用map join
    map join通常用于一个很小的表和一个大表进行join的场景
    通过设置参数hive.mapjoin.smalltable.filesize来决定多大的表会被定义为小表读入内存，默认为25000000字节（25M）
    通过设置参数hive.auto.convert.join来开启map join
    在使用mapjoin时，引用小表或子查询时，需要引用别名
    mapjoin支持小表为自查询
    left join的左表必须为大表（左连接从右加载表，先读取右表）
    right join的右表必须为大表
    inner join左右都可为大表
    full join不能使用mapjoin
    mapjoin可以使用不等值连接或or连接多个条件（不写on或on 1=1）

5、谓词下推（一篇搞懂谓词下推-CSDN博客）

1、原理：尽可能的使过滤条件提前执行，减少逻辑计算的数据量（所谓下推，即过滤条件在map端执行，不下推，过滤条件在reduce端执行）
2、对于inner join和full join，过滤条件在on或在where，性能没有区别
3、对于left join，左表的过滤条件写在where，右表的过滤条件写在on，性能会提高
4、对于right join，右表的过滤条件写在where，左边的过滤条件写在on，性能会提高
5、如果表达式中含有不确定函数，整个表达式的谓词将不会下推

6、数据倾斜

1、定义：在并行计算的时候，数据分布不均匀，导致大量的数据集中分布到一台或某几台计算节点上，使得该节点的处理速度远低于平均计算速度，成为整个数据集处理的瓶颈，从而影响整体的计算性能
2、表象：
    2.1、MapReduce任务（reduce阶段卡在99.99%，一直不能结束）
        ① 有一个或多个reduce卡住
        ② 各种container报错OOM
        ③ 读写数据量极大，远超其他reduce
    2.2、spark任务
        ① 绝大多数task执行的都非常快，但是个别task执行很慢
        ② 整个Executor执行时间特别长，整体任务卡在某个stage不能结束
        ③ 原本能够正常执行的spark作业，报内存溢出
3、原因：
    3.1、key-value值分布不均（存在大量相同key的数据，存在大量key为null的数据）
    3.2、机器配置不合理（分配给机器的数据量与机器处理能力不匹配的时候，会造成reduce进程卡顿）
    3.3、业务数据自身特性（例如某个分公司或某个城市订单量大幅提升几十倍甚至几百倍，对该城市的订单统计聚合时，容易发生数据倾斜）
    3.4、某些sql语句会触发数据倾斜
        join：其中一张表的key比较集中，导致分发到某个reduce中的数据远高于平均值
        group by：group by字段中的某些值的数据量较多，同样会导致发到某个reduce中的数据远高于平均值
4、解决方案
    4.1、空值引发数据倾斜
        ① 如果key为null的数据不重要，直接过滤掉这部分数据

② 对null值加盐操作，赋一个随机值（使用concat拼接随机值与一个固定的字符串，防止与其他真实数据产生冲突），只要key的hash值不一样就不会被分发到同一个reduce中（concat('hive_',rand())）

4.2、表连接时引发的数据倾斜

大表join小表：小表中的key比较集中，导致大量的数据在reduce阶段集中到了少量reduce中。可以装换为mapjoin，在map阶段完成join，这样就避免了shuffle和reduce，进而就不存在数据倾斜（spark可以将小表进行广播，然后大表在map广播流从而实现mapjoin）

大表join大表（倾斜部分单独处理）：大表中的某些key过于集中，例如某月销量过万的卖家只有10%，那么在reduce阶段这部分数据就会集中发送到少量的reduce中，导致某几台节点压力倍增，从而降低集群的效率。那么我们可以把这部分key比较集中的数据单独提取出来join，然后union all剩余数据join的结果。

4.3、group by分组的时候key分布不均

采用两阶段聚合，先局部聚合再整体聚合。可以给每个key加盐操作，拼接随机值从而使分发到一个节点的数据打散至多台机器上，实现局部聚合之后，再将拼接的随机值去除，再次进行聚合得到最终结果。

7、MapReduce与spark的shuffle区别（Spark的两种核心Shuffle详解（建议收藏）-腾讯云开发者社区-腾讯云 (tencent.com)）

MapReduce的map阶段会对数据进行切片，然后将每一个切片中的数据转换为key-value键值对。shuffle阶段就发生在map阶段处理结束之后，map阶段处理的数据会写入环形缓冲区，环形缓冲区的大小默认为100M，当环形缓冲区中的数据达到80%，就会溢写到磁盘中。在溢写过程中，会调用分区器对数据进行分区并且会根据key进行排序，如果设置了combine操作，那么也会按照key对数据进行合并。

reduce阶段，会根据分区号去maptask机器上取相应的分区数据，当从不同maptask机器上抓取到同一个分区的数据时，reducetask会将这些文件进行合并（归并排序），文件合并之后shuffle过程就结束了。reduce阶段会按key进行分组，然后执行reduce方法逻辑。

spark引入了两种shuffle，hash shuffle和sort shuffle（hash shuffle在2.0已经不在使用）

hash shuffle有两个阶段，shuffle write阶段发生在一个stage执行结束之后，下一个stage开始执行之前，每个task会将key的hash值相同的数据划分至相同的缓冲队列中，当缓存写满之后会将数据溢写到磁盘中。下一个stage中有多少个task那么我们上一个stage的每个task就得创建多少个磁盘文件，这样的话就会产生非常多的磁盘文件。在shuffle read阶段，stage中的每个task会从上一个stage中所有的task节点中拉取key相同的文件，一边拉取一边聚合。

为了优化shuffle write阶段落盘文件数量过多的问题，我们可以开启spark.shuffle.consolidateFiles参数，设置为true，这样的话在每个executor中只会为下游同一个task创建一个磁盘文件，从而大幅度减少落地的磁盘文件，进而提升shuffle的性能。

sort shuffle有三种运行机制：普通运行机制、bypass运行机制、tungsten sort运行机制

普通运行机制：前一个stage的输出结果会根据不同的shuffle算子写入到不同的数据结构中，如果是像reduceByKey这种聚合类shuffle算子，就会选用map数据结构，一边通过map进行聚合，一边写入内存；如果是join这种普通的shuffle算子，就会选用array数据结构，直接写入内存。当内存中的数据到达了设定的阈值，就对内存中的数据按照key进行排序，然后分批溢写到磁盘，最后将task产生的所有磁盘文件进行合并，并且生成一份索引文件，这个索引文件标识了下游各个task数据在文件中的start offset和end offset

bypass运行机制（shuffle map task 数量小于spark.shuffle.sort.bypassMergeThreshold=200参数的值。并且不是聚合类的shuffle算子）：reduce端的任务比较少的情况下，基于hash shuffle实现机制明显比基于sort shuffle机制的效率高，bypass运行机制将hash shuffle机制的shuffle write写入磁盘的临时文件进行了聚合并创建了索引，使最终落盘文件数量减少，并且不需要排序，这使得shuffle write和shuffle read效率都有所提升。

tungsten sort shuffle运行机制：是对普通sort shuffle的优化，会对数据进行排序，但是排序的不是数据本身，而是直接基于二进制数据进行排序，大大降低了内存的消耗。

8、小文件（大数据Hadoop之——HDFS小文件问题与处理实战操作 - 大数据老司机 - 博客园 (cnblogs.com)）

1、小文件的危害

① namenode会将hdfs上存储的文件的元数据读取到内存中，如果小文件数量过多就会给namenode和内存带来巨大的压力。并且每次读取文件在磁盘上寻址所耗费的时间要大于读取文件数据本身，给DataNode也会带来很大的压力。

② reduce阶段会对输入文件进行切片，然后为每个切片启动一个maptask，如果小文件过多就会开启很多maptask，这会给集群带来很大的任务开销，造成集群资源的浪费，降低集群的吞吐量。

2、如何解决小文件问题

2.1、hadoop archive：这是一个高效地将小文件放入hdfs块中的文件存档工具，他能将多个小文件打包成一个har文件，这样在减少namenode内存使用的同时，任然允许对文件进行透明的访问。eg：将目录/foo/bar下的所有小文件存档成/outputdir/zoo.har(hadoop archive -archiveName foo.har -p /foo/bar /outputdir)。解开归档文件只需要复制归档文件到另一个路径即可hadoop distcp har:///outputdir/user.har /outputdir/newdir2

对小文件存档后，原文件并不会被删除，需要用户自己删除；创建har文件实际上是在运行一个mapreduce作业，需要有一个hadoop集群运行此命令；

归档文件一旦创建便不可改变，要增加或移除里面的文件，必须重新创建归档文件

2.2、sequence file：由一系列的二进制key-value组成，如果key为小文件名， value为文件内容，则可以将大批小文件合并成一个大文件。sequence file支持压缩，但是不支持文件追加写和修改，适用于一次性写入大量小文件的操作。

实现方式：在mapper中的setup方法中获取文件名，然后在map方法中获取文件内容，然后将文件名作key，文件内容作value。最后再驱动类中绑定SequenceFileOutputFormat为输出格式。

2.3、开启JVM重用：开启JVM重用可以使mapreduce中的同一个job下的task共用同一个资源容器，这样可以避免频繁的申请和关闭资源容器，使同一个资源容器可以运行多个task，从而达到提高作业运行效率，减轻集群压力的作用

2.4、向hdfs上传文件的时候对文件进行合并（hdfs dfs -appendToFile file1 file2 /hdfsdir/mergefile）

2.5、针对hive表小文件进行设置：将hive的输入文件格式hive.input.format更改为org.apache.hadoop.hive.ql.io.CombineHiveInputFormat，控制输入阶段合并；设置mapred.min.split.size.per.node和mapred.min.split.size.per.rack控制单节点或单机架上最小切片的大小，如果有切片大小小于这两个值（默认都是100M），则进行合并；设置hive.merge.mapfiles和hive.merge.mapredfiles为true，控制mapreduce任务输出合并，hive会启动一个新的mr任务对输出的小文件进行合并；设置hive.merge.size.per.task和hive.merge.size.smallfiles.avgsize指定每个task输出合并文件的大小期望值和指定所有输出文件大小的平均值，都为1G，若不满足则会启动新的mr任务对文件进行合并

2.6、如果使用spark，可以对rdd进行重分区（使用coalesce(新分区数,true)或repartition()对数据进行重分区）

9、HDFS读写流程

9.1、写流程：

1、客户端向namenode发送写文件请求

2、namenode会去检查请求用户是否有写权限，以及现在的文件系统中是否已经存在该文件。如果该请求通过了namenode的请求，那么就将该写操作先写入编辑日志中（editlog），并且给客户端返回应答

3、客户端在接收到namenode的应答信息后，将需要写的文件进行分块（128MB），再次向namenode请求将文件块上传到哪些datanode上

4、namenode会向客户端返回可分配的datanode列表

5、客户端根据namenode返回的datanode列表向最近的datanode发送请求，然后datanode列表中的datanode就会建立起一个数据管道

6、客户端会按packet（数据包64KB）的形式向datanode管道发送数据

7、一个数据块传输完之后客户端会继续向namenode发送写请求，直到数据传输完毕

9.2、读请求：

1、客户端向namenode发送读文件请求

2、namenode会去检查请求用户是否有读文件权限，以及元数据中是否存在该文件信息。如果通过请求就向客户端返回文件块所在datanode地址

3、客户端会挑选最近的datanode服务器请求读数据

4、datanode以packet的形式向客户端返回数据，客户端会将数据写入缓存，最后写入目标文件

10、namenode和secondarynamenode工作机制

namenode用来管理元数据，secondarynamenode用来辅助namenode管理元数据

如果是第一次启动namenode，系统会自动创建镜像文件和编辑日志，如果不是，namenode会将镜像文件和编辑日志直接读取到内存中。客户端对数据的增删改都会被namenode追加写到编辑日志中。

一旦编辑日志文件大小达到一定的阈值或者距离上次镜像文件和编辑日志的合并时间达到一定的间隔时间，secondarynamenode就会向namenode请求合并操作，然后拷贝namenode的镜像文件和编辑日志，然后将这两个文件加载到内存中进行合并，并且写入到新的镜像文件中，并且拷贝到namenode替换原来的镜像文件

11、yarn工作机制

客户端向resourcemanager提交作业，resourcemanager会为该作业分配一个nodemanage，nodemanager会为改作业启动一个applicationmaster用于管理该作业；applicationmaster会根据作业的需求向resourcemanager请求资源，resourcemanager会将有足够资源的nodemanager返回给applicationmaster，然后applicationmaster会通知nodemanager创建资源容器去启动task。在task执行结束后，applicationmaster会向resourcemanager申请注销

12、资源调度器

1、先进先出（FIFO）

优点：调度算法简单

缺点：忽略了不同作业的差异，例如对海量数据进行统计分析的作业需要长期占用计算资源，而在其后提交的交互型作业不能及时得到相应，影响用户使用

2、容量调度器（默认的调度器）

支持多队列，每个队列都使用FIFO策略

如果一个队列中有资源剩余，可以暂时共享给需要资源的队列，当该队列有新的应用程序提交，其他队列需要归还这些资源

支持多用户共享集群资源，但是为了防止同一用户的作业独占队列资源，调度器会对同一用户提交的作业所占资源进行限定

队列资源分配：优先选择资源占用率最低的队列分配资源

作业资源分配：默认按照提交作业的优先级和提交时间顺序分配资源

容器资源分配：按照容器的优先级分配资源，如果优先级相同，按照数据本地性原则：同一节点->同一机架->不同机架

3、公平调度器

支持多队列

如果一个队列中有资源剩余，可以暂时共享给需要资源的队列，当该队列有新的应用程序提交，其他队列需要归还这些资源

支持多用户共享集群资源，但是为了防止同一用户的作业独占队列资源，调度器会对同一用户提交的作业所占资源进行限定

优先选择对资源缺额比例大的，每个队列可以单独设置资源分配方式

13、shuffle优化

13.1、map阶段：

1、增大环形缓冲区

2、增大环形缓冲区溢写比例，由80%扩大到90%

3、不影响实际业务的前提下，采用combiner提前进行合并

13.2、reduce阶段：

1、合理设置map、reduce数：太少会导致任务等待，延迟处理时间；太多会导致任务间资源竞争，造成处理超时等错误

2、规避使用reduce，使用mapjoin

3、增加每个reduce去map拿数据的并行数

4、增大reduce端存储数据的内存大小

13.3、IO传输

数据采用压缩方式，减少IO时间

map端主要考虑数据量大小和切片，支持切片的优BZip2、LZO

map端输出主要考虑速度，速度快的由snappy、LZO

reduce端主要考虑需求，如果作为下一个mr输入需要考虑切片，永久保存考虑压缩率大的gzip

14、zookeeper选举机制

14.1、zookeeper集群中只有超过半数以上的服务器启动，集群才能正常工作

14.2、全局数据一致，每个server保存一份相同的数据副本，客户端无论连接到那个server数据都是一致的

14.3、第一次启动选举：

假设集群共有5台机器。

服务器1启动，发起一次选举，投自己一票，此时服务器票数不够半数，选举无法完成

服务器2启动，再次发起选举，服务器1和2各自投自己一票，此时服务器1发现服务器2的myid比自己大，改投服务器2一票；此时服务器2有两票，但是票数还没有过半，所以选举无法完成

服务器3启动，再次发起选举，服务器1和2都会改投服务器3，这时服务器3的票数过半，所以服务器3状态改为leader，服务器1和2改为follower，

服务器4和5启动，发现已经存在leader，改自己状态为follower

14.4、无法与集群中的leader建立通信

如果集群中本来就已经存在一个leader，不能建立通信的follower尝试重新选举时，会被告知集群此时的leader信息，follower只需与leader建立通信，重新进行状态同步即可

如果集群中不存在leader，会根据epoch（选举代号），事务id，服务器id依次比较选出leader

15、zookeeper监听原理

在main（）方法中创建zookeeper客户端，启动两个线程，一个负责通信，一个负责监听；通过通信的线程将需要监听的事件发送给zookeeper，zookeeper会将这些事件添加到监听器列表中，当负责监听的线程监听到列表中的事件发生变化，就会调用process方法对发生变化的事件做出相应的处理

16、cap法则

cap理论：一个分布式系统不能同时满足一致性、可用性、分区容错性
一致性：指数据在多个副本间能够保持数据一致的特性
可用性：指系统提供的服务必须一直处于可用状态，对用户的每一个操作总是能够在有限时间内返回结果，如果超出了这个时间范围，那么系统被认为是不可以的
分区容错性：分布式系统在遇到任务网络分区故障时，仍然需要能够保证对外提供满足一致性和可用性的服务

zookeeper满足强一致性和高可用性

17、flume事务

17.1、put事务（应用在source-channel）
source会采集一批数据，封装为event，缓存达到设置的最大容量后开启put事务
doPut()方法将这批数据写入临时缓冲区putLIst中
doCommit()方法会检查channel内存队列是否足够合并，如果够，那么就提交event
如果检查channel内存队列空间不够，doRollback()方法会回滚，清空putList，然后给source返回异常，source会重新采集这批数据，然后开启新的事物
17.2、take事务（应用在channel-sink）
doTake()方法：sink将数据剪切到临时缓冲区takeList中，同时也拷贝一份放入写往hdfs的io流中
doCommit():如果event全部发生成功，就清除takeList
doRollback():如果发送过程中发生了异常，进行回滚，将takeList中的数据归还给channel。这个操作可能会导致数据重复，如果已经写入一半的event到了hdfs，但是回滚时会向channel归还整个takeList中的event，后续在开启事务想hdfs写入这批event的时候就出现了数据重复
17.3、flume事务仅能保证两个传输阶段的数据不丢，如果channel采用memory channel，数据存储在内存中，一旦发生异常数据仍然可能丢失；如果采用file channel那么数据不会丢失，但是在take事务阶段数据可能会发生重复

18、flume source

18.1、spooling directory source
监控指定目录下的文件，它会监视这个目录中产生的新文件，并在新文件出现时从新文件中解析数据出来，数据解析逻辑可配置。在新文件全部读入channel之后默认会重命名该文件以示完成（也可立即删除）
spooling directory source是可靠的，即使flume重新启动或被kill，也不会丢失数据。但是作为这个可靠传输的代价，指定目录下的文件必须是不可变的，唯一命名的。
如果文件在写入完成后又被再次写入新内容，flume将向其日志文件打印错误并停止处理；如果以后重新使用以前的文件名，flume将向其日志文件打印错误并停止处理
18.2、taildir source
taildir source监控指定的一些文件，并且在文件产生一行新数据的时候读取它，如果新一行数据还没写完，taildir source会等这行数据完成在读取
taildir source是可靠的，他会定期的以JSON的格式记录每个文件的最后读取位置，如果flume挂掉，它可以从文件标记的位置重新读取
taildir source可以从文件任意位置读取文件，默认从文件第一行开始读取
文件按照修改时间的顺序读取（FIFO）
taildir source不重名、删除或修改它监控的文件，不支持读取二进制文件，只能逐行读取文本文件
18.3、taildir source重点
1、支持断点续传、多目录
2、重复数据 通过下游处理，groupby 窗口

19、flume channel

19.1、memory channel
把event存储到内存上，适用于吞吐量有较高要求的场景，但是发生故障的时候会丢失内存中的所有event
19.2、kafka channel
1、将event存储到kafka集群，kafka提供了高可用性和复制机制，因此如果flume实例或者kafka实例挂掉，能保证event数据随时可用。
2、适用场景
① 与source和sink一起：给多有event提供一个可靠、高可用的channel
② 与source、interceptor一起，没有sink：可以把所有的event发送给kafka的topic中，来给其他应用使用
③ 与sink一起，没有source：提供一种低延迟、容错高的方式将event发送给各种sink上
19.3、file channel
数据基于磁盘存储，可靠性高，但是传输速度低。file channel可以配置datadirs指向多个路径，每个路径对应不同的硬盘，增大flume的吞吐量
19.4、生产环境如何选择
如果下一级是kafka，优先选择kafkachannel
如果是金融、对钱要求准确的公司，选择file channel
如果是普通日志，选择memory channel

20、flume sink

20.1、hdfs sink
将event数据写入hdfs，目前支持创建文本和序列文件。支持两种文件类型的压缩，可以根据写入事件、文件大小或event数量定期滚动文件（关闭当前文件创建新文件）。还可以根据event自带的时戳或系统时间对数据进行分区

21、zookeeper中保存着kafka的什么信息

保存着broker id和消费者offsets信息，没有生产者信息

22、kafka优点

削峰：上游数据有突发流量，下游可能扛不住，此时可以把消息暂存到kafka，起到缓冲作用
解耦：kafka可以作为一个接口层，解耦重要的业务流程，只需要针对数据进行编程
冗余：采用一对多的方式，一个生产者发布消息，可以被多个topic订阅，供给多个业务使用
异步通信：应用间可以并发处理消息，相比串行处理，减少处理时间

23、kafka消费过的数据如何再消费

可以在redis中存储offset信息，当想重复读取数据是，可以读取redis中存储的offset信息达到重复消费的效果

24、kafka数据存储在磁盘还是内存上，为什么速度会快（高效读写）

数据存储在磁盘上，
速度快是因为：
1、kafka采用顺序写入，减少了中间的寻址时间
2、kafka采用分片索引机制，将每个分区文件分为多个segment进行存储，每个segment包括index文件、log文件、timeindex文件，这些文件位于同一个文件夹下，这个文件夹以分区名+分区序号命名。根据offset可以定位segment文件，然后根据index文件确定log文件的位置，然后遍历log文件找到目标记录
3、采用分区技术，并行度高
4、采用页缓存和零拷贝技术，kafka的数据加工都由生产者和消费者处理，kafka broker应用层不需要关系存储的数据，因此数据不需要走应用层，传输效率高。当上层有写操作时，操作系统只是将数据写入页缓存中。读数据时，先从页缓存中查找数据，如果找不到再去磁盘中查找。

25、kafka数据怎么保证不丢失（不重复）

1、生产者端
kafka的ack机制有三种状态：如果设置为0，则生产者不需要接收broker的应答；如果设置为1，则生产者需要接收leader的应答才可以继续发送消息；如果设置为-1，则生产者需要接收leader和follower的应答才可以继续发生消息
如果ack设置为1，可能在leader宕机的时候丢失数据，想要严格保证生产端数据不丢失，必须设置为-1
2、broker
需要设置分区副本，生产者和消费者都只会和leader进行交互，follower会主动和leader同步数据；当leader宕机时，可以从follower中重新选举leader与生产者和消费者进行交互。如果leader宕机，那么就可能造成数据重复，开启幂等性和事务可以保证单分区数据不重复。
3、消费者端
消费者通过记录每次消费数据的offset值来保证数据不丢失，，每次消费数据时都会接着上次的offset进行消费。但是offset信息并不是每次消费后都会保存的，因此可能会造成重复数据。可以将数据的唯一主键保存在redis中，每次消费数据时先去redis判断是否已经存在该数据的主键

26、kafka数据积压

1、如果是kafka消费能力不足，可以考虑增加topic的分区数，并且增加消费者组中消费者数量
2、如果是下游数据处理不及时，可以提高每批次拉取的数量。如果每批次拉取数据过少，使处理数据小于生产数据，就会造成数据积压

27、为什么kafka不支持读写分离

在kafka中，生产者和消费者都是与leader进行交互，并不支持主写从读
假设某一时刻leader刚更新一条数据，follower此时还在同步中，因为同步过程具有一定的时延，数据还未完全同步，如果直接从follower中获取数据，那么就会产生数据不一致问题

28、HBase读写流程

28.1、写流程：
    1、从zookeeper中找到存储元数据表的节点位置
    2、访问该节点，获取hbase:mate表，将其缓存到内存中（matecache）
    3、根据表空间、表名、rowkey对照元数据缓冲（matecache），需要将数据写入到哪个regionserver
    4、把数据追加到预写日志（wal）中，然后写入到写缓存中（memstory）并排序
    在memstory达到阈值时，将缓存中的数据刷写到story中
28.2、读流程
    1、首先去zookeeper中获取存储元数据表的节点信息
    2、去存储元数据表的节点上访问元数据表，并读入缓存中
    3、根据表空间、表名、rowkey找到对应的regionserver
    4、首先去写缓存中查看是否更新过这条数据，如果更新过就直接读取，如果没有就查看读缓存（block cache），如果读缓存中也没有，就到story上取数据加载到读缓存中，最后再返回给客户端

29、spark作业提交流程

以spark yarn cluster为例
    cluster模式下，用于监控和调度的driver模块在yarn集群资源中执行，一般用于生产环境
    1、任务提交后会和resourcemanager申请启动applicationmaster
    2、随后rm会分配资源容器（container），在合适的nodemanager上启动applicationmaster，此时的applicationmaster就是driver
    3、applicationmaster启动后会向resourcemanager申请启动executor
    4、resourcemanager会为executor分配资源容器（container），然后再合适的nodemanager上启动executor
    5、executor启动之后会向driver进行注册，然后driver开始执行main函数
    6、在执行到action算子时，触发一个job，并根据宽依赖划分stage，每个stage生成对应的task，然后分发到executor上执行

30、