## hadoop及其生态内技术部分

1. hadoop1.x中namenode保存的元数据有什么？

文件名、权限、大小、被分成哪些块，注意磁盘上保存的并没有块分布在哪些node上，而是在启动集群的时候datanode上报的，在内存中。

1. hadoop1.x中secondarynamenode是如何合并fsimage和edit log的？

①新建一个edits文件用来记录此时刻之后的用户所有操作日志记录

②把此时刻之前的edits文件和对用的fsimage文件拷贝到snn上

③snn进行合并

④把心的fsimage文件传给nn

⑤替换旧的fsimage

注意：内存中的元素局与fsimage中的同步问题，不要认为内存中的慢，因为在操作的时候记录edits的时候内存中就已经改变了，而fsimage就是一个持久化的机制而已，用于再次启动的时候加载。

合并时机：

①fs.checkpoint.period默认3600秒

②fs.checkpoint.size，默认edits文件64M

1. HDFS读写流程？

读流程：

①调用DistributedFileSystemAPI的open方法

②从NN中获取元数据——数据块的位置信息

③调用FSDataInputStreamAPI的read方法

④具体read读取数据，根据第二步返回的block信息，比如block1有三个副本，会找一个空闲的副本所在datanode所在机器读取，其他的Block可以同时进行。

⑤调用FSDataInputStreamAPI的close方法

写流程：

①调用DsitributedFileSystemAPI的create方法

②访问NN，告诉NN我要上传文件，并把文件名、权限、大小、用户给NN，此时，NN会根据文件的大小分blocks，以及blocks的第一个副本需要放在哪些机器上

③调用FSDataOutputStreamAPI的write方法

④根据第二步NN反馈的信息把block的第一个副本写入指定的DN上，写完第一个副本就结束

⑤调用FSDataOutputStreamAPI的close方法

⑥反馈给NN说success

注意：第四步client往DN上写完第一个副本就结束了，那么副本机制怎么实现？这里的第二个第三个副本的复制以及位置选择都由第一个副本所在的DN决定并完成！

1. 说一说你对MR的shuffle过程的理解？

从整理上来说shuffle过程主要分为三个机制的合作：分区、排序、合并、合并

①partition分区

把map的输出进行分区，默认分区是根据key的hash模 partition，可以自定义，目的是负载均衡

②sort排序

当缓冲区快满的时候对80M（总大小100M）的数据进行sort，默认的排序规则是字典排序，一定要注意字典排序中11在9个前面

③combiner合并（可有可无）

内存中有个环形缓冲区默认100M，在内存中进行partition和sort操作，当达到80%的时候将会发生溢写到磁盘，如果大量的小文件肯定不适合网络传输，所以在溢写之前进行combiner操作，将相同key的value加起来，减少溢写数据量。

④merge合并

小文件合并为大文件

用一个例子说一下比较明了

map读入的记录如下（两个reduce）：c: 2,a: 1,b: 1, b: 1,d: 1

partition操作：a和c分为一个区，b和d分为一个区

sort操作：环形缓冲区快满的时候，a和c位置调换，总的顺序是[a:1 c:1][b:1 b:1 d:1]

merge操作：这里合并多次溢写的临时文件{a,[1,1,1,1,1...],c,[1,1,1,1]}...

如果存在combiner的话：

sort之后溢写之前会进行combiner：[a:1 c:1][b:1 b:1 d:1]变成[a:1 c:1][b:2 d:1]

注意：从map端的merge可以看出为什么reduce接收到的value是一个迭代器了！

注意：combiner和reduce端的merge的区别。conbiner是我们可以操控的，合不合并，怎么合并，产生什么的keyvalue我们说了算，但是reduce端的merge和sort是不可控的，是同一个merge和sort，并且merge只是合并相同key，而不是可控的。

那么map端的merge和reduce端的merge有什么不同呢？也是有区别的，因为map端的输出和copy是同步进行的，所以map端的merge并不是merge所有map输出结果，但是reduce端的merge必须是所有的map输出copy过来的数据的总merge，不然数据肯定会出错。因为reduce的输入是按照key+iterator的，所有不允许出现相同key的多个结果。

1. MR的split大小？

max(min.split, min(max.split, block))

换成语言就是说，最大只能是max.split，最小只能是min.split，其他时候根据block大小决定，所以一般我们说split的大小是64M，不太准确

注意：可以优化，设置最小split和最大split的大小！

1. 请解释一下为什么hadoop2.x解决了hadoop1.x的NN的单点问题和内存受限问题？

单点问题：HA

内存受限：Federation

HA机制：

①两个NN，一个主一个备，DN要想所有的NN汇报块信息、心跳

②备用NN通过journalNode来同步元数据，所以当主宕机时，元数据还可以获取到

③FailoverController这个是主备切换的关键，用来检查NN的心跳和切换NN，具体操作就是Failovercontroller向zk汇报NN情况，如果主宕机了的话，zk就知道了，此时会通知备NN你们来竞争吧，竞争到锁的NN就切换为active。

注意：之前一直有一个疑问就是client是如何找到正在处于active的NN的IP的？纠结过一段时间，现在可以认为是zk的存在，因为zk可以做统一命名空间嘛，这个命名空间下有多个ip，这样client每次去找zk问我们的cluster现在活着的是哪个ip，所以....

Federation机制：

解决了NN可以水平扩展的问题，但是只有大公司才会用HA+Federation，直观上就是通过多个命名空间namespace将元数据的存储和管理分布在多态机器上。

注意：上面说的关于元数据高可用是有错误的！！！

关于两个NN和JN之间到底什么怎么实现元数据的高可用的？edits log和fsimage到底在哪合并的？

①客户端的操作请求发给Active NN，NN会在内存中修改，然后在edits log中追加一条操作记录，注意些edits log同样也会发生在JN上，所以他们是同步写入edits log的，所以并不是JN拉取的

②一般来说checkpoint合并操作并不会在Active NN上进行，所以会在StanBy NN上进行合并操作，具体就是StandBy根据dfs.namenode.checkpoint.preiod（默认3600秒）或者dfs.namenode.checkpoint.txns（默认100万次）时进行合并fsimage和edits，具体操作如下

③StandBy NN检查是否达到checkpoint条件（两个，上面说了）

④达到checkpoint条件后，将该namespace以fsimage.ckpt\_txid格式保存，并随机生成一个MD5文件，然后将fsimage.ckpt\_txid重命名为fsimage\_txid

⑤StandBy通过http与Active NN建立连接

⑥Active NN通过HTTP获取fsimage\_txid文件并保存为fsimage.ckpt\_txid文件，也生成一个MD5文件，ActiveNN与StandByNN上MD5文件比较，如果无误就将fsimage.xkpt\_txid重命名为fsimage\_txid。

所以关于第二个问题：到底在哪里进行的checkpoint，答案是StandBy NN上。

1. 简单说一说yarn，然后具体说一说任务申请执行的整体流程？

yarn是hadoop2.x引入，最重要的目的就是将资源分配和任务调度分离！！同时可以兼容第三方的计算框架。RM负责整个集群的资源分配，AM（applicationmaster）负责任务的分割、调度、监控、容错，RM不负责具体任务的容错，负责AM的容错，任务失败后AM向RM重新申请资源。

下面就说一下任务申请、调度、执行的流程：

step1：客户端向RM提交一个作业申请

step2：RM根据请求信息返回一些信息，包括application\_id、input路径对应的文件元数据、作业资源提交的hdfs路径

step3：客户端根据RM的返回信息生成资源文件（job.split、job.xml、app.jar）提交到指定hdfs路径

step4：向RM申请运行AM

step5：AM将用户的请求打包为task放置到任务队列中，等待RM的调度策略调度（从web上可以看出来，申请任务只有就会有AM出现但是是accepted状态而不是running状态）

step6：当RM调度到该task的时候会像一个NM分配该task，该NM从hdfs下载任务资源信息并运行AM

step7：！！AM是任务调度中心！AM根据任务资源信息想RM申请运行Map task的资源

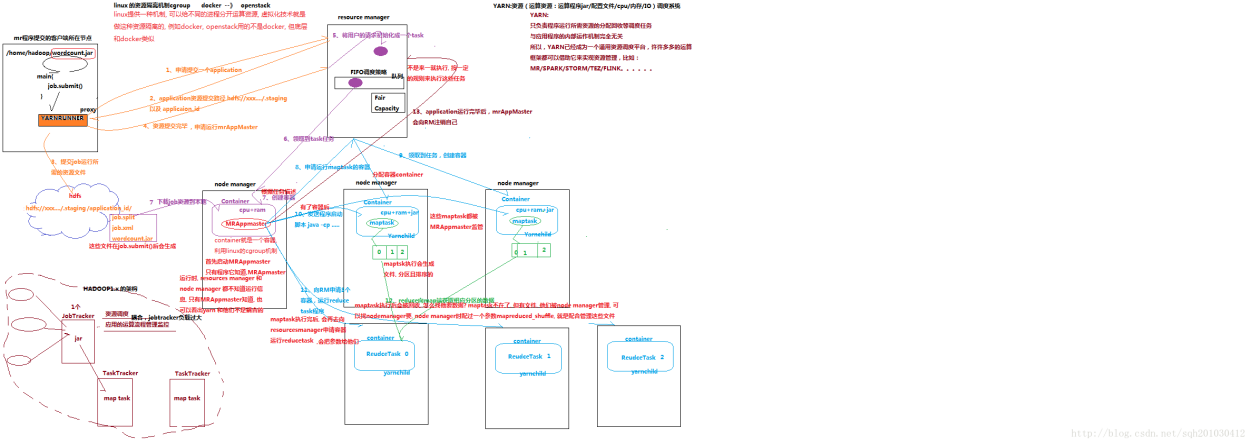
step8：注意：这里容易出错，AM申请的Map task仍然会放到RM的任务队列中等待，在map task运行之前AM显示的状态一直是accepted

step9：RM针对资源的调度是有自己的一套策略的，包括根据内存分配、资源阈值、动态分配等等，如果资源申请完毕，RM会指定哪些NM来运行containers，container下载task、资源文件信息

step10：AM启动NM上container的map task注意！！！container中的maptask不是RM启动的而是AM启动的！

step11：待运行到一定程度后会申请reduce task资源，重复上述过程

step12：AM向RM申请注销自己



注意：这里有一个优化就是container的重用，以及copy阶段的内存占用比例。

1. RM的三种调度策略对比以及适用条件？

①FIFO先进先出

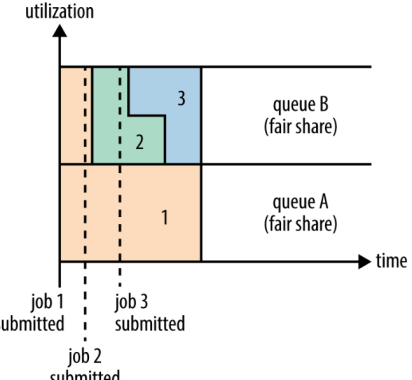
默认调度器，肯定不行

②capacity scheduler容器调度器

支持多队列，每个队列预先设定资源量，每个队列采用FIFO策略，存在资源抢占，但是仅限于配置中的最大设置

③fair schedule公平调度器

支持多队列，每个队列设置资源量，但是和capacity的区别在于队列中并不是固定的FIFO，而是可以设置的，默认是公平策略，说白了就是在分队列的情况下，内列内部的job是公平如下图



另外：我们要知道linux关于cpu调度的算法是CFS指的是complete fair scheduler绝对公平，从算法层面来看就是利用进程的等待时间来决定执行顺序，默认开始是0，选择一个等待时间最长的执行执行，其他进程等待时间增长，然后再选择等待时间最长的执行。关于具体执行多长时间是根据实际时间/进程数决定的。

进一步问：既然聊起了调度算法，那么问问你LVS负载均衡的调度算法你知道多少？

①radom随机：通过ip列表的随机int索引实现

②round robin轮询：通过ip列表轮询实现

③weight权重算法：通过权重比改变ip列表，权重大的占位多，然后用随机索引

④source hashing源地址哈希：计算源地址ip的hash然后取模作为列表索引

⑤LC（least connection）最少连接：发送到最少连接的ip上

⑥wlc（weight least connection）加权最少连接：....

你知道nginx的5中调度算法是什么吗？

①轮询

②加权

③ip\_hash

④url\_hash

⑤fair：比较智能，会根据相应快满决定，但是需要加载upstream\_fair模块

最后问你你个问题：怎么能够强制改变某个用户提交用的优先级和执行队列？

调整优先级：

hadoop1.0及以下版本：hadoop job -set-priority job\_201707060942\_6121418 VERY\_HIGH

hadoop2.0及以上版本：yarn application -appId application\_1478676388082\_963529 -updatePriority VERY\_HIGH

改变队列：

yarn application -movetoqueue application\_1478676388082\_963529 -queue root.etl

1. NN是如何用double-buffer双缓冲机制保证高并发访问的？

换句话问：每秒上千次请求的时候，NN多线程操作内存并进行edits log的磁盘写和网络传输给JN，多线程高并发存在安全隐患，也就是txid顺序的问题，那么为了保证安全就要加锁，那样的话，嘿嘿性能一落千丈，hadoop是如何解决的呢？

Double-Buffer：

hdfs采用一块缓冲内存，分成两部分，一部分专门用于edits log的写入，另一部分用于读出写入磁盘、通过网络写入journalnode

分段加锁+双缓冲机制：

第一阶段锁：各个操作线程依次获取锁，生成顺序的txid，把edits log写入第一块缓冲内存后，瞬间释放锁，这样操作元数据和写入edits log就比较快。可以感觉出并发的感觉。

第二阶段锁：之前的线程中的一个获取写磁盘和写网络的锁，如果当前没有线程在持久化edits，那么它就会把两块缓冲互换进行写磁盘和写网络操作，另一块可以进行并发写入，但是不读取。其他线程竞争这个锁发现有线程在写的话会阻塞（如果自己的txid已经被之前的线程写入磁盘的话就不阻塞，直接返回即可）

1. HDFS上传TB级大文件是如何优化上传速度的？

可以与hdfs写操作联系起来，当NN创建目录索引、返回第一个block副本需要上传到哪个DN上，之后client与DN打交道。这个过程无非就是本地输入输出流+拷贝流+网络传输+输入输出流的过程。但是如果基于简单的socket网络传输如用byte[]来作为缓冲效率肯定底下。

这就是为什么调用FSDataStreamInputAPI和FSDataStreamOuputAPI的write和close的原因。

Chunk缓冲+Packet缓冲+异步发送：

①chunk缓冲机制

inputstream读入的本地数据进入FSDataStreamoutput，数据会写入一个chunk缓冲数组，每个chunk有512字节大小，这样就可以不用因为等待网络传输而阻塞读入

②Packet数据包机制

chunk数组写满后就会进行切割，多个chunk数据段会写入Packet数据包，一个packet可以容纳127个chunk，默认64M，其实可以认为是另一层缓冲

③内存队列异步发送机制

当一个packet被塞满后，就会把packet放入队列，DataStreamer线程不断获取packet数据包并把它发送出去。

1. 谈一谈hadoop文件的续约检查机制是怎么实现的？

TreeSet：底层是TreeMap，TreeMap是排序的kv，底层是红黑树。

文件契约机制：

hadoop是不支持多客户端对同一文件的并发写入的，所以hadoop有一个文件契约机制保证同一时间只有一个客户端获取文件的契约。

但是有一个问题就是当长时间没有写入时，有一个时间阈值，hadoop认为放弃契约，解除契约好让其他客户端写入，但是低效的问题在这里，如何快速过滤过期且没有续约的契约。

Treeset可以按照续约时间来作为排序的key，这样每次只需要看最前头的契约是否过期即可，如果第一个都没过期其他的契约更不可能过期....

1. 你知道hadoop内部的数据限流吗？

①Balancer数据平衡数据流：如raplaceBlock限流，默认是1M。dfs.datanode.balance.bandwidthPerSec

②fdimage镜像文件的上传下载数据流传输，默认不开启dfs.image.transfer.bandwidthPerSec

③VolumeScanner磁盘扫描的数据读操作的数据传输，比如在探寻block是否损坏的时候，为了方式读取数据块影响业务的IO而设置的，默认1Mdfs.block.scanner.volume.bytes.per.second

注意：有一个优化点就是限流的period以及提高balance的带宽、balance的线程数等，因为限流的机制是period内流量是否用完，如果设置太小会有太多阻塞！

既然提到了数据平衡，那么就说一下数据平衡的过程吧？

第一种情况：集群数据不均

直接调用自带的start-balancer.sh -t 15%来平衡即可

第二种情况：节点内磁盘数据分布不均

①shutdown该节点的datanode，然后移动数据

②diskbalancer，生成计划、执行计划、查看执行进度

1. 如果hdfs数据节点挂载多个磁盘大小不一致会出现什么情况？

会出现该节点小磁盘写满后该服务器上datanode和nodemanager挂掉，如何解决？

hadoop写磁盘策略：

①循环选择策略

很简单，循环往多个挂载的磁盘上写block，很容易造成小磁盘写满

②可用空间策略

剩余空间减去磁盘剩余阈值（配置）如果可以大于block大小，那么进行循环选择，如果小于block大小，那就找可以撑得下的磁盘写入

注意：可优化的点：第一：可以设置写入策略为可用空间策略和阈值；第二：设置datanode.dfs.reserved，这个设置目的是防止磁盘写满之后无法运行mapreduce任务；第三：设置datanode运行损坏的磁盘数，比如说挂载的4块，可以设置允许损坏1块。

1. hadoop大量小文件解决办法？

其实可以分析一下这个问题，分两种情况：

第一种：这些小文件是任务产生的，并没有明显的区分，这种情况下可以通过改变任务的reduce数量、程序输出策略、自定义读取小文件生成大文件的方式解决，没有任何影响

第二种：这些小文件没有任何联系，各自有各自的文件名、不同的内容，说白了就是不是任务产出的，这些文件处理起来就比较麻烦了。

①hadoop archive——hadoop自带的归档命令

将小文件归档为一个大文件，可以指定块大小，缺陷是一旦创建不可改变

②sequence file

sequencefile是二进制的kv，文件名作为key，内容为value，缺陷是一旦创建不可改变

③计算时使用combinerFileInputFormat

可以将多个文件合并成单独的split，只是解决了map计算的资源问题但是没有解决存储问题

④hbase

文件名作为rowkey，内容作为value，但是成本略高，存储和获取都要经过hbaseAPI。

注意：有一个优化：可以设置最小的split输入大小，用于处理大量小文件很有意义，可以避免过多的申请map task资源。

1. 如何发现hadoop任务中某个map或者reduce task运行比别人慢很多怎么办？

如果是map阶段的话：

①hadoop job -kill-task taskid

②hadoop job -fail-task taskid

建议使用第②中，因为杀掉的话再次调度还是有可能在该服务器上，但是使其fail的话，会有黑名单的机制，所以不会在失败的那台上再次运行

如果是reduce阶段的话：

①使用上述的两种杀掉task方式

②很有可能出现的数据倾斜

1. 说一说hive元数据中各个表存的是什么吧？

①VERSION：版本信息

②DBS：hive中数据库与hdfs目录的映射

③SDS：hive中文件存储的基本信息如输入输出类型，hdfs目录

④TBLS：hive表、视图、索引表的基本信息

⑤COLUMNS\_V2：存储表对应字段信息

⑥PARTITIONS：存储表分区的基本信息

⑦PARTITION\_KEYS：储存分区字段信息

⑧ROLE:角色

1. hive的几种常见存储格式text、orc、parquet、sequence、lzo、rc、avro，最好能说说压缩算法？

通过网上已有的测试ORC最好，其次是parquet。

先说一说行式和列式存储：

①行式存储

优点：相关的数据保存一起，符合面向对象，一行一条记录，比较适合insertupdate操作

缺点：如果只查询几个列，会把整行数据读取出来，性能低；其次由于列数据不一致导致压缩比不高，空间利用率低

②列式存储

优点：只查询部分列时不会把整行查询出来，可以跳过不必要的查询；有高效压缩比

缺点：insertupdate比较麻烦

①Textfile

行式存储，不做压缩，磁盘开销大，解析开销大，可以结合gzip、bzip2使用，自动解压，但是hive是不可以进行split切分的，lzo+索引可以解决，但是snappy压缩比低，但是效率高，所以lzo是压缩比高、效率高的结合方式

②sequencefile

二进制文件，行式存储，kv格式

它有一些冗余的信息，所以比textfile大，但是支持压缩哦！

③RCFile

列式存储，压缩比高，保证同列数据尽可能在一个block中

存储结构是元信息+真实数据，真是数据lazy解压，提高效率

算法：元数据采用RLE(RunLength Encoding )压缩，真实数据采用gzip算法压缩

存储格式：以RowGroup为单位，包含（元信息+data）

④ORCFile

列式存储，是RCFile的优化版本

比RCFile多了索引信息，结构是：stripe（包含index+rowdata）

注意：有一个优化的点：hive默认不适用ORC索引，需要打开hive.optimize.index.filter= true

算法：基于数据类型的压缩，integer采用RLE，字符采用字典压缩算法

⑤parquet

列式存储，也有rowgroup

性能介于ORC和RC之间

⑥avro

二进制文件，主要为了解决跨语言的问题

下面就介绍一下RLE和字典压缩吧，因为用的比较多：

①RLE：行程长度压缩算法

以block块为基本单元来判断重复以及重复次数，为了提高压缩效率，block块一般取1Byte

两种处理方式：当连续的block是重复的block时，采用[2]block1[1]block方式来压缩，如果没有连续的block，一种方式放弃压缩，另一种则是采用第一种处理方式，只不过重复次数全是[1]

②DE字典压缩

之前遇到过，说白了就是让程序观察当前看到的数据是否和之前的有重复

第一：LZ77，采用滑动窗口的方式避免字典过大，如果发现有重复的，那就记录(距离,几次重复)，这个过程需要字典+带搜索缓冲（最长匹配），输出是（回溯几位，长度）+新字符

第二：LZ78，与LZ77相比，LZ78最大优点是在每个编码步骤中减少了比较的数目，输出是（出现重复的标记，新字符）

第三：LZW，比LZ78的优势在于，它预先把所有可能出现的重复情况全部列入字典，然后只需要在字符那里标记在字典中的索引顺序即可。

第四：huffman编码：它跟LZW的思想一模一样，就是把所有可能先列出来，然后只用编号来标记，但是区别再于LZW不需要把编码写入文件中

1. hive分桶是怎么回事，和分区有什么不一样，为什么数据量大时分桶比分区查询效率高？

分区：是根据列值的不同进行分区，在hdfs上以不同目录的形式展示出来，目的是提高查询命中性能。

分桶：是通过对列值的哈希计算来实现的。所以分桶是固定的桶数。

还有一点区别：分区是目录，而分桶是文件，粒度稍微小一些。

为什么当数据量足够大，分区数很多的时候分桶比分区的效率要高？

还没有一个很好的解释，暂且认为分桶的粒度更细，数据定位更好，而且还可以进行分区+分桶。

既然提到了分区表，那么请问你知道动态分区和静态分区吗，他们什么联系和区别？

①静态分区指的是，当load数据的时候需要指定所有分区值，如load data local inpath '/root/csdn01.txt' into table source\_table\_01 partition (event\_date = '2019-02-17');需要指定分区值

②动态分区指的是不需要指定分区字段值，查询结果自动分区，但是条件是动态分区的字段需要在select后的最后面，同时也跟设置有关hive.exec.dynamic.partition.mode=strict/nonstrict：

如果是strict，意思是虽然支持动态分区，但是必须要指定一个分区字段值，而且是第一位的如insert overwrite table target\_table\_01 partition (event\_date='2019-02-17',period)

select s.id as target\_id,s.name as target\_name,'day' as periodfrom source\_table\_01 s;

如果是nonstrict的话就没有任何限制如

insert overwrite table target\_table\_01 partition (event\_date,period)

select s.id as target\_id,s.name as target\_name,s.event\_date as event\_date,'day' as periodfrom source\_table\_01 s;

再问一个问题？msck命令用来干嘛的？

修复hive分区结构的，比如手动创建hive的分区目录之后hive中show partitions并没有该分区，此时需要执行msck rpair table ...;

1. hive支持的数据类型？

①数字类型

②日期时间类型

③字符串

④混杂类型boolean binary

⑤复合类型array、map、struct

1. mapreduce三次排序都出现在哪里，原理是什么？

在Map任务和Reduce任务的过程中，一共发生了3次排序

①当map函数产生输出时，会首先写入内存的环形缓冲区，当达到设定的阀值，在刷写磁盘之前，后台线程会将缓冲区的数据划分成相应的分区。在每个分区中，后台线程按键进行内排序

②在Map任务完成之前，磁盘上存在多个已经分好区，并排好序的，大小和缓冲区一样的溢写文件，这时溢写文件将被合并成一个已分区且已排序的输出文件。由于溢写文件已经经过第一次排序，所有合并文件只需要再做一次排序即可使输出文件整体有序。

③在reduce阶段，需要将多个Map任务的输出文件copy到ReduceTask中后合并，由于经过第二次排序，所以合并文件时只需再做一次排序即可使输出文件整体有序

注意：在这3次排序中第一次是内存缓冲区做的内排序，使用的算法使快速排序，第二次排序和第三次排序都是在文件合并阶段发生的，使用的是归并排序。

1. MapReduce与Tez的区别和联系？

MR是google提出的一种编程模型，适合海量数据并行批量计算，模型分为两个stage，map阶段进行input、process、sort、spill、combine、merge、output；reduce阶段进行input、sort&merge、process、output。

TEZ是apache开源的在yarn上支持DAG有向无环图作业的计算框架，是对MR的一种优化，比如impala、flink、spark、drill的作业都是根据有向无环图进行划分stage然后分发tasks的。

比如一个复杂的HQL可能通过Antlr对sql进行解析生成多个相互依赖的MR作业。然而TEZ可以将多个有依赖的作业转换为一个作业，减少大量的磁盘写操作。它是将MR整个过程拆分若干个子过程同时把多个MR任务组合一个较大的DAG任务，减少了MR之间的文件存储。

## spark部分

1. 说一说spark、flink的区别吧？

ss代表spark streaming

sss代表structured sparkstreaming

|  |  |  |
| --- | --- | --- |
|  | spark streaming | flink |
| 计算模型 | 微批  执行引擎层面：一批都处理完才发送，所以可能存在缓存不足写磁盘的过程 | 流，基于事件驱动  执行引擎层面：一条处理完立马通过缓存发往下一个operation  不过flink可以控制这个流程，到底缓存多少再发送，阈值为0时就是标准的流，一条一条，阈值无限大时就是批发送了，所以越小延时低但是吞吐小 |
| 编程模型 | 底层是RDD，基于内存还是磁盘可以设置。然后封装为dataframe和dataset，发展趋势就是要用sql统一 | 统一为有状态的streaming process，框架自己判断是否基于内存上层分为datastream和dataset，table+sql |
| 时间机制 | ss支持processingtime，sss支持processingtime和eventtime | flink支持eventtime，processingtime，ingestiontime |
| 延迟 | 较低 | 低 |
| 吞吐 | 较高 | 高 |
| 语义保证 | exactly once at least once | exactly once可选 |
| 背压机制 | 如果流量达到阈值就限流，流量是根据计算调度时间、处理时间、结束时间、消息条数计算的 | 从底向上，底层处理不过来就阻塞上层数据流入，主要是 jobmanager 针对每一个 task 每 50ms 触发 100 次 Thread.getStackTrace() 调用，求出阻塞的占比 |
| checkpoint机制 | 使用metadata+data checkpoint来达到exactly once语义，重量级  ，当然也可以自己写事务来实现exactly once。如果修改程序将无法从checkpoint中恢复，所以推荐自己实现事务提交达到exactlyonce | memory/file/db三种保存状态的statebackend。  开启事务，开始checkpoint，打入一个barrier，流程operation的时候写入将barrier汇报给jobmanager同时会记录算子和系统的state到statebackend中，当barrier流动到最后sink之后，通知checkpoint完成，提交事务。  程序改动可以从ckeckpoint恢复 |
| DAG的生成与执行 | 由diver完成构建DAG划分stage生成taskset调度task（注意：每个executor调度的task由于数据本地性导致不一样） | 客户端生成streamGraph、生成Jobgraph、由jobmanager生成并调度executiongraph（注意：每个executor运行的task一样） |
| 迭代 | 普通迭代 | 支持增量迭代 |
| lazy特性 | transformastion是lazy的action不是 | 所有操作都是lazy的，只有调用env.execute才加载 |

1. RDD是什么，它有哪5个特点？

RDD：resilient distribute dataset弹性分布式数据集

①可分区，一组分区 rdd可跨机器，partition不可以

②rdd中计算以partition为单位

③rdd之间存在依赖关系，所以有天然的容错性

④partition func有两个，一个是hash partition还有一个rangpartition，条件是kv数据

⑤rdd会有一个专门的列表存储每个partition对应hdfs上的数据块，这样就可以把task调度到对应的服务器上运行，移动计算不移动数据。

既然说到了移动计算不移动数据，还要说明一点：spark的stage的划分也有这方面的考虑，spark不像storm和flink一样，不同的task预先确定在哪里运行了，而spark为了加强数据本地化的考虑，降低网络开销，会动态调度task的执行服务器！

1. 既然提到了rdd的依赖关系，那么我想问问你宽依赖和窄依赖的区别，以及DAG的阶段划分又跟宽窄依赖有什么关系？

宽窄依赖的区分主要体现在父与子rdd之间的产出关系，形象的说就是独生子女的问题

如果父RDD的某一个分区数据进入子RDD的多个分区，超生了，那就是宽依赖。可以简单的任务shuffle过程就是典型的宽依赖。

DAG是Driver端产出的，driver也会把DAG划分不同的stage，依据就是依赖关系，窄依赖在一个stage内，出现宽依赖就划分不同的stage

1. spark standalone和on yarn任务提交流程、作业执行流程？

standalone模式：master+workers架构：

①client提交方式

--master spark://node01:7077 --deploy-mode client

这种方式，driver将会在client上运行

step1：driver启动sparkcontext，初始化DAGscheduler和taskscheduler，当调用到action操作的runjob时，向master注册app信息并申请executor资源（cpu mem）

step2：master分配worker并启动executor，里面有线程池供driver调度执行task，worker定时汇报心跳等信息给master用于监控，同时executors反向向driver注册自己

step3：driver遇到action操作时的DAGscheduler将任务生成DAG并划分stage，每个stage就是taskset，TaskScheduler将task分发给executor（根据数据本地化）运行，同时也将程序代码发送过去，同时也负责task的失败重试，他俩都有维护运行队列和等待队列，根据依赖关系来判断该等待还是该执行

step4：所有task运行完毕sparkcontext想master注销释放资源

②cluster提交方式

--master spark://node01:7077 --deploy-mode cluster

和上面唯一的区别在于driver是master选定的

on yarn模式：

①yarn-client提交方式

--master yarn-cluster --deploy-mode client

这种方式driver运行在提交的client上，appmaster在yarn中，他俩之间进行通信而不是包含关系

step1：client本地运行driver并向RM申请启动applicationmaster

step2：driver初始化sparkcontext、DAGScheduler、TaskScheduler，与appmaster通信，appmaster向RM申请运行资源，注意，这里跟standalone不同，这里是yarn，yarn的关键是资源和调度分离，RM只负责资源分配，调度由appmaster负责，所以例子由appmaster启动container作为executors，并且executors向driver汇报心跳而不是RM和appmaster

step3：接下来差不多一致了，DAGScheduler产出DAG和stage也就是taskset，TaskScheduler吧stask分发到executor上执行

step4：运行完毕后appmaster向RM申请注销释放资源

②yarn-cluster提交方式

--master yarn-cluster --deploy-mode cluster

和上面唯一的区别在于作为driver运行在appmaster中

1. driver端获取结果数据的大小有什么要求吗？

需要先了解一下executor端如何处理运算结果的：

task运行后的结果，executor会将结果序列化（字节数组）封装在DirectTaskResult里面，再把DirectTaskResult序列化（字节数组）——这个字节数组用于网络传输，那么为什么封装一下？因为不能直接返回结果，要先带有结果大小的对象给driver，让driver判断一下我要不要你才行，不然所有executor直接返回结果，driver一般来说会蹦。

①生成结果大于masResultSize（默认1G），直接在executor端丢弃，可以设置。driver只获得了InDirectTaskResult对象，只知道总结果的大小和blockid

②生成结果大于maxDrectResultSize（默认128M），小于maxResultSize，将结果存入BlockManager并返回其编号，通过Netty发给Driver，maxDirectResultSize 由 spark.task.maxDirectResultSiz 和 spark.rpc.message.maxSize 控制，取两个中的最小值。由driver综合所有结果后再做判断是否要拉取结果。

③生成结果小于maxDrictResultSize，直接返回给Driver所有结果

注意：这些设置是针对driver端的，假如有一个executor返回结果大于maxResultSize，那么所有的结果直接丢弃就好了就像①中所说的。

1. 什么是shuffle？有哪些操作可以触发shuffle？shuffle为什么耗性能？shuffle原理的实现发展历史？

其实还可以问一下调优的问题，不过这个问题会在最后总结。

shuffle从宏观上来说就是利用分区器对数据的重新分发；从执行的角度来看就是一个分区中的数据进入多个分区的过程，all to all的过程。通常涉及跨机器、跨程序的数据复制。

引起shuffle的操作分为三类：

①重分区（repartition、coalesce、repatitionandsortwithinpartition）

②ByKey操作（除count之外，groupbykey、reducebykey、sortbykey、aggregatereducebykey）

③join操作（join、cogroup）

为什么shuffle如此耗费性能？

关于shuffle在MR那里就学过hadoopshuffle，分为分区、排序、combine、合并，这个过程中会有大量的磁盘io、序列化、数据拉取的网络占用操作所以会耗费性能。

最理想的情况spark所有数据都在内存中，也要占用cpu序列化操作和大量的网络带宽，并且还要耗费内存，因为在传输数据之前要在内存中整理数据。

下面来说一说spark进行shuffle过程的发展史吧：

spark负责shuffle过程的执行、计算、处理的组件主要是ShuffleManager。shufflemanager实现shuffle主要有两种方式：

①hash base shuffle（已经过时了）

②sort base shuffle

在介绍shuffle之前还得说一点：shuffle分为shuffle wirte和shuffle read(fetch)两个阶段，而shuffle read肯定不用多说，就是数据copy sort merge，其实关键点就是在shuffle write

①最开始的hash shuffle

每个map task都有一组partitions文件，每个partition都对应一个文件，所以是maps\*partitions个shuffle write出的小文件

②优化后的hash shuffle——consolidation

使用一个buffer，不管map多少个，只要是相同key（partition）的数据都写入一个buffer，这样可以保证最终每台服务器上只有partitions个大文件（这句话不严谨，并行运行时得看使用多少个cpu core，如果1台服务器、3个core并行、4个分区，那么产生的文件是3\*4而不是4，因为并行同时写入一个buffer，所以服务器cores越多并行度越高，但是io小文件也增多了），多个map task对应一组partitions文件，每个partition都对应一个文件，所以是maps/并行cores \* partitions，说白了就是多个map向相同的partitions文件中写

③sort base shuffle

说白了：sort base shuffle比hash高明的一点就是把小文件最终合并成一个大文件了，对，是一个，同时会有一个索引文件。

第一种sort base shuffle：BypassMergeSortShuffleWriter

条件：shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值，默认200，而且不能是聚合类的shuffle算子如reducebykey

实现：跟未优化的hash base shuffle一样，只不过多了一步，把所有文件合并为一个大文件的过程。

就这就解释了那两个条件：如果是聚合类算子的话，map端没有预聚合的效果，性能大打折扣。如果分区数过多，那么合并大文件的时候并发打开小临时文件太多。

第二种：sort base shuffle：SortShuffleWriter——外部归并排序

没有条件限制，适用于聚合、排序的算子

其实最终目的仍然是一个大文件+索引文件，只是过程多了两步

在写入buffer之前，会先写入数据结构（map/array）进行预处理+排序

比如如果是聚合操作，会选择map，读入数据的时候会进行预聚合，如果是join操作会选择array数据结构，溢写之前要进行排序

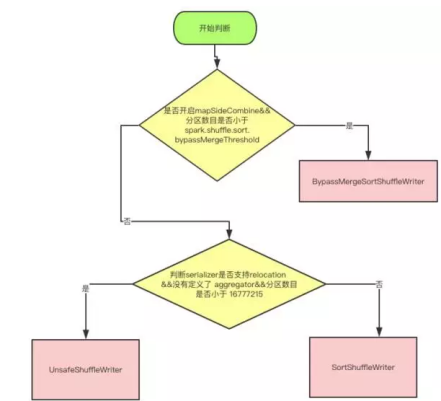
注意：bypassmergesortshufflewriter和sortshufflewriter的区别有三：其一是数据结构；其二是buffer；其三是全局排序（bypasssort只是简单的merge，而普通sort是全局sort+merge）

第三种：sort base shuffle：UnsafeShuffleWriter————RadixSort基数排序

条件：没有聚合要求、没有key排序要求；Serializer需要支持relocation；分区数目小于16777216

实现：UnsafeShuffleWriter维护一个外部排序器——基数排序，对全局进行排序，但是跟标准的SortShuffleWriter是有很大区别的，其一：它不是根据key来排序的而是根据partition id排序的，所以粒度比较大，所以不适合聚合、排序的算子；其二用于排序的数据结构内存不足时发生溢写，区别在于它溢写出来的数据要进行序列化压缩输出，并记录每个分区端的seek位置，这也就解释了为什么它的Seializer需要支持relocation，因为全局排序还需要再读取呢！

|  |  |  |  |
| --- | --- | --- | --- |
| 区别点 | BypassMergeSortShuffleWriter | SortShuffleWriter | UnsafeShuffleWriter |
| 排序 | 不排序 | 根据key排序 | 根据partition id排序 |
| 排序算法 | 无 | 快排+归并排序 | 基数 |
| 预聚合 | 无 | 支持 | 无 |
| 适用 | 无聚合、排序要求的shuffle，效率最高 | 有聚合、排序要求的shuffle | 无聚合、key排序的shuffle |



1. 请问spark的计算并行度是怎么确定的？

是由每个stage的最后一个rdd结果的分区数确定的，一般来说一个partition一个task，最后reduce出结果的时候可以手动改变reduce数量，也就是改变了最后一个RDD的分区数，也就是改变了并行度。

上面说的是窄依赖的并行度，那么shuffle时的并行度是什么？

如果是默认的hash分区，就会使用配置中默认的shuffle时分区，但是如果是自己定义的分区函数，那么就要自己制定分区时的并行度，如reduceByKey(...,num)

1. 请简述一下job的生成和执行过程？

一旦driver遇到action操作就会生成job，如果有多个action就会生成多个job，每个job包含一个或者多个stage，在生成DAG过程中会从后往前划分，遇到宽依赖就进行划分。提交过程是先提交没有父stage的stage，而且只有该stage运行完毕才会提交下一个stage。

1. Spark数据倾斜的现象、后果、原因、解决办法？

现象：多数task执行速度远块与少数task，等待时间很长时间可能会提示内存不足，比如从web页面中发现某个stage运行时间过长、shuffle read和shuffle write相差巨大，点进去发现某个task的read time和computetime过长等....

后果：很想然会延长计算时间，拖垮程序，可能还是导致计算失败

原因：

①数据本身分布极其不均（key=null/key选择不合理，比如选择性别作为key）

②计算的并发度过低，比如数据分布均匀，但是并发度只有1或者2，那么也可以认为是数据倾斜

解决方法：

①找出异常的key，可以通过采样的方式，对于null值或者其他无效的key，可以在计算之前过滤掉，如果是正常的key，确实出现了大量的key倾斜，解决办法就是加盐进行shuffle，然后去掉盐在进行shuffle一次，这样的意义在于经过第一次加盐shuffle时候先把那些大量重复的key进行了一次预聚合，跟map 端merge感觉差不多，但是机制不同。注意，这种方式仍然存在倾斜问题。

②如果加盐预聚合的方式不好，可以把这个大量重复的key单独处理，然后在union！！

③如果不是key的问题，而是程序的问题，那么可以加大并行度。

④尽量避免shuffle，使用广播。比如是join操作，可以选择map-side-join（适合有一个小表，这个方案不一定适合倾斜场景）

1. hive中数据倾斜.......如何解决？

现象上看跟spark差不多，spark是某个stage的tasks的某个或者某些计算过慢，从表现上来看就是reduce快速达到99%然后停留在99%然后停滞不前；从结果目录中发现大量小文件，一个大文件等....

原因：

①数据key分布极不均匀

②hsql语句本身就有数据倾斜，比如选择的key值分布很少而数据量很大

比如join时，groupby，count(distinct)这些涉及shuffle的都有可能出现

③join时，关联字段在两个表中的类型不同，比如int的id 和string的id，那么hive默认会把string的全部发到一个reduce上

解决方案：

针对key为null的解决方式：

②先把特殊key过滤掉然后在单独处理特殊值，最后union

③为null的key赋新值如字符串+随机数，这样就可以让key均匀，而结果不会影响，因为没有能跟字符串+随机数匹配上

针对不同数据类型的解决方式：

④对于不同数据类型关联的情况可以用类型强制转换解决原因就是上述所说的，hive默然把另一种类型的key方导一个reduce中

针对join的数据倾斜情况解决：

①开启map端聚合set hive.map.aggr=true（典型的是小表join大表），注意也要有限制条件，因为有的key没有分部不均，map端聚合就是浪费资源，如果是两个大表进行join呢？可以把其中一个大表划分小表再进行map join；如果不大不小的表进行join怎么办，利用子查询来做，也就是两层map join，第一层join先匹配+distinct出少量的值来，第二层在进行map join。如果是典型的group by，最好结合聚合函数sum avg 用，因为这样可以在map端就进行预聚合，减少数据分发量而且可以避免数据倾斜

⑤如果已经hive这个表有数据倾斜的话，我们可以告知hive，这个表数据倾斜了，set hive.groupby.skewindata=true，那么hive会以两个MR来进行，第一个MR将数据随机分布下去，进行预聚合，第二个MR进行真正的结果聚合。

1. 哎呀我突然想起来要问你个问题，刚才我们聊了shuffle的问题，join也会触发shuffle，spark的join的shuffle过程你了解吗？如何优化join操作？

终于等到你，join的实现机制？

在说spark的join实现之前先看一看hivesql的join是怎么实现的吧：

hive join：

hive sql中的join就分为两种情况，一种是common join（reduce-side-join），另一种是map join（map-side-join），hive0.11版本之后有配置可以进行map-side-join优化。

①reduce-side-join：很显然就是默认的join方式，根据关联key来进行分区、排序、合并、分发到reduce，在reduce端进行匹配

②map-side-join：一般是小表join大表，小表的大小限制默认是25M，默认是开启hive.auto.convert.join的：MR会在map之前产生一个local map task，将小表读成hashtable的数据结构，然后真正的map task会遍历大表与缓存中的小表记录进行匹配，输出匹配成功的进行shuffle，这样就减少了io和网络传输

spark join：

上面说过spark的shuffle，shuffle机制的选择（bypasssort、sort、unsafeshuffle）根据具体的操作算子和RDD来定，而join操作肯定会触发shuffle，那么join的shuffle有什么可选项吗？

①broadcast hash join：适合小表和大表

②shuffle hash join：适合中小表与中小表（或大表）

③sort merge join：适合大表与大表

broadcast hash join和shuffle hash join都是基于hash join的，而hashjoin也比较简单，就是将小表读取hash表，然后遍历另一个表与其匹配然后进行shuffle输出，区别在于broacast hash join是先匹配再shuffle出结果，而shuffle hash join是先shuffle，然后在进行匹配。当小表没那么小时，广播的方式给driver和executor都带来巨大压力，所以shuffle hash join的做法是先shuffle，把两个表进行重分区（shuffle），这样每个分区内的小表的数据就会很小了，此时再将每个分区的小表数据读入hash表，这样就跟上面一样了，一条一条遍历，匹配的数据就是结果。这种方式无疑比broadcast hash join要耗费网络带宽，但是没办法。

然而对于大表与大表的join应用上述的办法肯定不行，因为会耗费巨大的内存和网络带宽。那么sort merge join是怎么做的呢？它跟shuffle hash join一开始一样，都是先进行shuffle，这样虽然耗费带宽，但是减小了executor的压力，何况是大表与大表，然后就是sort merge join的区别了，它对各个分区的两个表的数据进行排序了而不是对表进行hash读取。接下来就简单了，采用归并merge匹配即可，因为是排好序的，所以不用全部加载到内存。

1. 请你总结一下hadoop的shuffle与spark的shuffle有什么不同？

①shuffle管理器

hadoop默认是sort-baseshuffle，过程中一定会发生内存sort和merge sort和reducesort过程，而spark分为hash-base shuffle和sort-based shuffle，1.2之后默认是sort-based，而具体实现中spark会自动选择bypasssortshuffle（无排序，很类似hash shuffle）、普通sortmergeshuffle（进行快排+归并排序）、unsafeshuffle（根据partition排序）三种中的一种进行shuffle

②shuffle的排序次数

hadoop——三次，map两次、reduce全局一次

spark，普通sortmergeshuffle才会进行排序（两次），reduce阶段不排序（因为spark是shuffle第二次排序是全局排序，并且是基于rdd，rdd是有分区的，并行度也跟分区有关，所以两个taskshufflewrite结果不会交叉，也就不存在reduce阶段的全局排序！，而hadoopreduce阶段拉取的数据在一起！）

③逻辑划分

hadoop：partition、sort、spill、merge、pull、sort+merge

spark：shuffle write +shuffle read（pull），注意spark管理shuffle数据是根据blockmanager的，task的结束会把blockid等信息给driver，driver进行下一个task时知道哪里pull数据

④shuffle map过程产生中间文件数量

hadoop虽然由于缓冲区会有很多磁盘小文件，但是有merge的机制，而且产生文件数量不确定，因为reduce拉数据是异步的

sparkshuffle中间结果文件数量根据shufflemanager有关。最初的hash shuffle产生文件是m\*r；优化后是cores(并行度)\*r，sort-base的是每个task只有一个大文件（所以是executors\*tasks）

⑤reduce操作（fetch操作）

hadoop的fetch是粗粒度的：拉取过来先进行sortmerge在进行聚合计算

spark不同，是细粒度的，边拉取边计算，上面说过原因，因为一个分区内的数据肯定已经排好序了，所以不用再排序和聚合

1. spark streaming对接kafka的两种方式你了解吗，说一说各自的实现和区别？

两种方式：reciever-base和direct的区别

③offset的保存

reciver：提交到zk上，所以才会出现程序与zk的消费不同步的危险

direct：offset在kafka中，它自己管理，跟zk没有关系

①简化并行(Simplified Parallelism)。

reciver方式程序需要指定多个reciver用来读取topic的所有分区，读取的最终形成RDD的时候需要合并reciver接收的数据，因为不知道哪些数据是哪个topic的，而且rdd的分区和topic的分区不是一一对应。

driest周期查询kafka中topic+partition的offset，不现需要创建以及union多输入源，Kafka topic的partition与RDD的partition一一对应

②高效(Efficiency)。

Receiver-based保证数据零丢失(zero-data loss)需要配置spark.streaming.receiver.writeAheadLog.enable,此种方式需要保存两份数据，因为reciver从kafka读取的数据缓存在bclokmanager中，浪费存储空间也影响效率。而Direct方式则不存在这个问题。

③强一致语义(Exactly-once semantics)。

High-level数据由Spark Streaming消费，但是Offsets则是由Zookeeper保存。通过参数配置，可以实现at-least once消费，此种情况有重复消费数据的可能。

④降低内存资源。

Direct不需要Receivers，其申请的Executors全部参与到计算任务中；而Receiver-based则需要专门的Receivers来读取Kafka数据且不参与计算。因此相同的资源申请，Direct 能够支持更大的业务。

⑤鲁棒性更好。

Receiver-based方法需要Receivers来异步持续不断的读取数据，因此遇到网络、存储负载等因素，导致实时任务出现堆积，但Receivers却还在持续读取数据，此种情况很容易导致计算崩溃。

Direct 则没有这种顾虑，其Driver在触发batch 计算任务时，才会读取数据并计算。队列出现堆积并不会引起程序的失败。

注意：官方推荐的保证exactly once消费语义的实现是自己管理offset，比如mysql/redis等，因为刚才说了一旦程序变了checkpoint将失效。

1. 是时候问问你RPC的问题了，请问spark的master与worker之间如何通信，hadoop的呢？

说spark和hadoop的RPC之前先说一下RPC是什么、工作机制等。

RPC，remote procedure call，远程过程调用，是一种通过网络从远程计算机上请求服务，而不需要了解底层网络技术的协议。

RPC工作机制：

RPC采用C/S模式，请求机作为客户机，提供服务的是服务器，可以认为是两个进程间的函数调用，比如A进程想要调用B进程中的函数或者方法，不在一个内存空间不能直接调用。

①通讯问题：client与server建立TCP连接，可以是短连接也可以是长连接，

②寻址问题：A需要告诉底层RPC框架B服务器的主机名/ip地址和端口以及需要调用的方法名称

③方法参数网络传递：方法参数需要通过底层的网络协议如TCP传递给B服务器，将内存中对象序列化为二进制进行传输

④B收到数据进行反序列化，然后找到对应的方法进行本地调用，返回

⑤返回值，同样经过网络传输给A。

JAVA中流行的RPC框架：

①RMI：java自带的框架，有局限性

②Hession：基于http的远程方法调用：基于http协议传输，性能方面不完美

③Dubbo：基于Netty的高性能RPC框架

HADOOP中的RPC的实现：

和其他RPC框架一样，分为四个部分：

①序列化层：client与server端通信信息采用了hadoop提供的序列化类Writable

②函数调用层：通过java的反射机制和动态代理实现

③网络传输层：采用基于TCP/IP的socket机制

④服务端框架层：server利用java NIO（非阻塞的异步）以及采用了事件驱动IO模型，提高并发

hadoopRPC构架那步骤：

①定义RPC协议：协议说白了就是定义接口，也就是client和server的通信接口

②实现RPC协议：就是实现接口

③构造和启动RPCserver：直接使用静态类Builder构造server并启动

④构造RPCclient，使用静态方法getProxy()构造代理对象，直接通过代理对象调用远程方法。

1. Dataset与Dataframe的区别？

首先说明一点，需要引入import spark.implicts.\_才可以使用dataset和dataframe

①DataFrame每一行是一个Row对象（类型），如果用case class来封装每一行的record也是可以的，所以访问数据有两种，一种是用Row来调用字段，dataframe不会解析里面的字段，不知道什么类型、哪些字段，只有当getAs调用他们的时候通过强制设定或者模式匹配来获得字段信息，比较复杂，另一种就调用class对象

②DateSet可以认为是DataFrame的特例，DataSet存储的每一行必须是强类型，在编译时就进行类型检查！

然而到底使用dataset还是dataframe要看具体情况，如果已经确定数据的格式，那么用case class来封装对象，然后用dataset即可，如果不能确定数据内容，那么最好用dataframe[Row]的方式，在程序中动态加载class来封装也可以

rdd、dataframe、dataset互相转化：

import spark.implicits.\_

testDF.rdd、testDS.rdd

rdd.toDF(“col1”, “col2”)

rdd.map(line => People(line.\_1),line.\_2)).toDS

testDS.toDF

testDF.as[People]

从DF转DS就可以看出，DS确实是DF的特例，因为DS可以随意转DF，但是DF转DS时就要指定强类型了

1. 刚才问你的spark与flink的区别，你说checkpoint的机制不同，现在请你详细说一说他俩都是怎么实现的？

flink——采用分布式快照的方式进行checkpoint：

①barrier

Flink的容错机制的核心部分是制作分布式数据流和操作算子状态的一致性快照，关键点在于barrier，flink将barrier注入数据流，始终不会超越数据流记录，barrier所在位置会不断汇报给jobmanager，所有操作算子都会受到barrier，当遇到sink的时候（DAG的末端）它就会向checkpoint确认快照完成！之后job将不会再向数据源请求本次barrier之前的记录。机制就是当一个算子受到barrier的时候它将不会操作之后的记录！而是把本次barrier和上次barrier之间的数据进行操作！它会把barrier之后的数据放入缓冲区！

②state

包含操作算子操作的状态、系统状态，当算子遇到barrier时，将操作缓冲区数据，在将barrier发往下一个输出流之前将对状态进行写快照！state很大！，默认存储在jonmanager的内存中，最好设置在hdfs中。

state包含：对于每个并行流数据源，创建快照时流中的偏移、对于每个运算符，存储在快照中状态指针。

③异步状态快照

从上面可以看出由于进行快照，如果同步进行的话会有记录处理的延时，所以RockDB这种backend方式可以异步进行快照，是通过创建一个快找对象来进行的，数据持续处理，ckeckpoint只有收到所有sinks的快照完成才算checkpoint完成。

④recovery

很简单，直接取出最新的ckeckpoint即可，重置算子状态，然后从数据源中从barrier隔离的offset消费数据即可。如果是增量快照，那么就从最新的完整快照开始，递增的应用一系列的增量快照

spark的checkpoint：

flink说白了就是保存了barrier的信息和state信息

spark保存两类信息：metadata和data checkpoint

spark的核心思想就是用metadata保存application的信息如配置、name、已完成batch和未完成batch，DAG、算子用来恢复driver，而另一方面则是依赖高容错的HDFS来保存真实的data，这个就比较重量级了，为了避免过长的的依赖，会定期将中间生成的RDD持久化到HDFS，这就是spark的checkpoint机制！

1. spark的数据本地性机制有几种？

|  |  |
| --- | --- |
| process\_local | 进程本地化，task需要的数据在同一个executor中 |
| node\_local | 同一台机器不同executor中或者同一台机器的本地磁盘或者hdfs的block块在该机器上 |
| no\_pref | 根本不存在本地性的数据，比如读取mysql |
| rack\_local | 机架，比如数据在同一个机架的另一台服务器的executor中或者另一台服务器的磁盘上 |
| any | 跨机架 |

在web UI中可以查看task的数据本地性机制的值，如果发现很多都是ANY的话，极有可能是hostname并没有用上，因为hadoop中一般是用hostname来区别机器的，所以最好在/etc/hosts把所有服务器的hostname和内网ip放进去！！！

1. spark的缓存机制，各自的适用情况？

①MEMORY\_ONLT：默认机制，只保存在JVM内存，所有对象都会反序列化到JVM中，如果保存不了，那么一部分partition数据就不放入内存，需要的时候recomput

②MEMORY\_AND\_DISK：内存+磁盘，与memory only的区别在于rdd缓存不下的数据会持久化到磁盘，需要的时候读取，所以这就需要权衡了，到底要CPU还是要IO。

③MEMORY\_ONLY\_SER：跟刚才俩者完全不同，它把对象序列化为字节数组！，这种方式占用内存小，但是需要用的时候需要反序列化占CPU，不过网络传输却很快，如果数据量还是非常大，内存放不下的话，需要的话还是要recomput

④MEMORY\_AND\_DISK\_SER：更刚才说的差不多，区别就是持久化到磁盘而不是recomput

⑤DISK\_ONLY：不用说肯定不行

⑥MEMORY\_ONLY\_2：两份，streaming方式默认级别

⑦MEMORY\_AND\_DISK\_2等等：两份

⑧OFF\_HEAP（experimental）

如果问有多少种缓存方式的话，一共12种（API中的），但是大致可以分为四类————只用内存、内存+磁盘、只用内存+序列化、内存+磁盘+序列化

建议是内存+磁盘+序列化（kryo）

注意：最好开启自动unpersit机制（配置），最好是程序中不用的时候进行unpersist！

1. rdd的创建方式，rdd有几种操作，如果你能说出第三种我就要你...？

①程序集合

②本地文件

③hdfs文件

④mysql等rdbs

⑤Nosql如hbase

⑥流入socket、mq

⑦hive等sql查询引擎

⑧基于s3

1. structured-streaming的三种sink（output）模式，你能说说到底有哪些sink和source吗？

①complete mode：完全模式，整个result table将被输出到sink，聚合查询支持

②append mode：默认，追加模式，从上一次trigger以来添加到result table中的新行会sink，并且要知道result table中已经存在的数据是不能更改的！

③update mode：更新模式，从上一次trigger以来result table中发生变化的行会sink。利用这一点可以使用watermark方式处理延时数据然后replace语法插入数据或者更新数据。

sinks:file sink、kafka sink、foreach sink、foreachBatch sink、console sink

source：file source、kafka source、socket source、rate source

既然说到了foreach sink和foreachBatch sink，他俩有什么区别吗？怎么用？

①foreachBath：其实名字已经告诉我们了foreacherBatch是将stream query的结果按照批来处理(DF)，这样可以将不能应用在stream上的DF操作应用起来然后再批量sink，当然了这个sink得接受批量写入，如果这个sink的下个阶段是srteam的话是不行的。

②foreach从而foreach sink可想而知就是一条一条的处理，这种方式完全符合流式处理的思想，需要实现一个ForeachWriter来处理record。而一般来说调用sink的时候说明已经到最后了，所以用batch效率更好一些。

上面提到了trigger，说白了就是触发嘛，那么有几种触发机制呢？

①default（不设置）：query会在上一个batch processing 完成之后立即执行（不太好）

②fixed interval micro-batches：固定间隔批处理，分三种情况，正常情况下上一个batch processing在间隔内完成，那么下一次proceesing会在固定时间执行，另一种情况是上一个processing在间隔内没有完成，那么下一个processing就等待其完成然后理解执行，第三种情况如果根本没有数据进来的话，是不会启动processing的！

③one-time micro-batch：全局只执行一次的query

④continuous with fixed checkpoint interval：实验阶段

1. structured-streaming的watermarking和flink中的watermark有什么不同？

首先要明确一点就是watermark是解决数据延时的问题，但是延时是相对event time而言的，如果程序使用processing time的话根本不存在数据延不延迟，所以只有当使用event time对数据处理的时候才会用到watermark。

sss的watermark目的是根据eventtime解决数据延时的问题，但是只适用于append和update mode，因为如果是complete那就完蛋了，一条数据延时一天，难道一天都不输出结果，就为了等着这个延时这么大的数据？？sss对watermark的应用很简单直接直接调用withWatermark(“时间字段”, “允许延时时间”)

flink的watermark是在给record assign event time的时候打上的水印。而flink的水印分为两类：with period watermark，基于时间间隔是水印，说白了就是给一个时间容忍度！另一类：with punctuated watermark，基于最新数据打水印的，如果是延时数据是不打水印的。

如果要说flink和sss的watermark的不同的话就是flink的更灵活。

1. spark的trigger和flink的trigger区别？window窗口的区别？

trigger区别：

sss的trigger刚才整理过了，官网给出能用的3中：default、固定间隔、全局执行一次，从我的角度来说sss的trigger没有什么意义，因为sss的trigger是在output的使用，而在output之前的由query产生的stream是每次的结果，这些结果是需要记录的，正常来说就是默认的有数据来就进行output就行，当然了具体情况具体分析，可能为了高效采用固定间隔的trigger。

flink的trigger则是专门针对window操作的，sss的trigger只要是output就可以应用。flink的trigger则跟sss的trigger不同，flink的trigger直接控制着什么时候计算！！比如我们使用了滑动窗口，但是真正的计算时是event time到达窗口时间的时候吗？？显然不是，是根据watermark来确定执行时间的！当watermark的时间由超过时间窗口的时候才计算！：

①onelement：只要有数据进来就计算当前窗口的数据，浪费

②oneventtime：当使用了eventtime作为时间戳的时候根据它和时间窗口的比较来决定

③onprocessingtime：跟oneventtime差不多，不过使用的是processlingtime

④onmerge：当时用session window的时候，这时就不是根据时间戳了，而是时间间隔

注意：：：：flink的trigger可不仅仅根据时间、时间间隔来讲进行计算，还可以自定义，flink内置有eventtimetrigger、processingtimetrigger、counttrigger、purgingtrigger，其中counttrigger是当窗口内数量达到一定数量的时候进行计算，超过的记录扔掉。另外flink还支持evictor，这个东西专门用来控制一个窗口内记录数、记录时间戳间隔等等，不满足evictor要求的记录丢掉！

window区别：

sss支持滑动和滚动窗口，flink支持滑动、滚动、session、global窗口

①tumbling window：滑动窗口

②sliding window：滚动窗口

③session window：是根据time gap来确定的窗口

④global window：这种情况下非常适合自定义trigger

以上是简单的做的对比，关于flink的内容下面会详细整理

1. 为什么出现了structured spark streaming，和streaming的区别有哪些，说白了就是为什么用sss而弃用ss？

ss的弊端：

①使用的processing time而不是event time

②只有low-level api，运行效率完全取决于程序员的编程能力

③end to end application，end to end指的是input到output，比如消费kafka数据然后写入mysql，ss只能保证DStream的exactly once语义却不能保证input和output的exactly once语义，如果想保证只消费一次语义就必须自己实现事务。

④批处理和流处理不统一

sss的特性：

①内置的很多connector来保证input和output的exactly once消费语义

②高级api如sql、dataset\dataframe api

③可以基于event time

1. 什么是output的幂等写入？

关于什么是幂等性，在介绍kafka的幂等性的时候已经介绍过了，目的是保证exactly once的语义，说白了就是一次请求和多次相同的请求实现的结果是一致的，比如mysql中的乐观锁机制，利用MVCC的版本控制达到如果version已经小于当前的version则不进行操作；比如mysql的悲观锁，利用唯一索引的方式只能有一个客户端操作当前订单，不管是成功还是失败都会删除这条数据，如果其他人想要操作必须等到上一个客户端完成，不管是成功还是失败；悲观锁可以用分布式锁来实现如redis、zk等；token令牌的方式：先申请tocken放在redis中，当发生真正支付的时候会根据判断你写到的token是不是我给你的，如果是就进行支付。这些不管是悲观还是乐观的实现都是为了保证幂等性。而这里说的output幂等性写入就是指相同的数据不能被写入多次。

## flink面试题部分

1. 为什么说flink统一了流和批处理？

因为flink无论是批处理还是流处理，底层都是有状态的流处理，flink执行批处理实际上是流处理的一种特例，只不过此时的流式有界的，而流处理的流式无界的，应用于流处理上的transformation完全可以应用在batch上并且table API和sql都可以用在批处理和流处理上只不过区别在于①容错并不是采用的流式处理的checkpoint，而是直接重新计算②dataset api处理的数据是很简单的数据结构，而stream处理的是key/value③流处理在应用transformation和table api和sql的时候不支持topN、limit、sort普通字段等操作

另外从计算模型上来说：批处理每个stage只有完全处理完才会把缓存中（缓存+磁盘）序列化的数据发往下一个stage，而流处理是一条一条，批处理吞吐量达，流处理时效性强，而flink则是采用了折中的方式，在内存中划分缓冲小块，当小块满了就发往下一个stage。如果缓存块无限大，那么就是批处理了。

1. flink的架构？

主从结构jobmanager+taskmanager两个进程（可以把client也加进去）。

集群模式：standalone，on yarn（在yarn上运行一个flink集群/提交到yarn上运行flink job）

jobmanager：

①registerTaskManager：在flink集群启动时，taskmanager会向jobmanager注册

②submitjob：flink程序内部通过client向jobmanager提交job，job是以jobgraph形式提交

③canceljob：请求取消一个flinkjob

④updateTaskExcutionStage：更新taskmanager中excution的状态信息

⑤requestnextinputsplit：运行在taskmanager上的task请求获取下一个要处理的split

⑥jobstatuschanged：executionGraph向jobmanager发送该消息，用来表示job的状态变化

taskmanager：

①注册：向jobmnager注册自己

②可操作阶段：该阶段taskmanager可以接受并处理与task有关的消息

client：

client对用户提交的代码进行预处理，client将程序组装成一个jobgraph，它是由多个jobvertex组成的DAG。

关于flink生成dag、提交job、分发task等细节再任务提交面试题会整理。

1. flink中应用在tableAPI中的UDF有几种？

①scalar function：针对一条record的一个字段的操作，返回一个字段

②table function：针对一条record的一个字段的操作，返回多个字段

③aggregate function：针对多条记录的一个字段操作，返回一条记录

1. 说一说flink的迭代与增量迭代吧？

迭代计算一般出现在机器学习和图计算的应用中，flink通过迭代的Operator中定义step函数来实现迭代算法，包括Iterate和Delta Iterate两种类型，实现他们就是反复在当前迭代状态上调用step函数，知道满足给定条件才会停止迭代。

①Ierate

是一种简单的迭代，每一轮迭代step函数的输入或者是输入的整个数据集或者是上一个迭代的结果，通过该轮迭代计算出下一轮计算所需要的输入（next partial solution），满足迭代终止条件后会输出迭代最终结果。说白了就是只要有不满足条件的元素，所有元素都一视同仁全部在进行迭代。

②Delta Iterate

增量迭代，它有2个输入，其中一个是初始workset，表示输入待处理的增量stream数据，另一个是初始solution set，他是经过stream方向上Operator处理过的结果。第一轮迭代会将step函数作用在初始workset上，得到的结果workset作为下一轮迭代的输入，同时还要增量更新初始solution set，如果反复迭代直到满足终止条件，最后会根据solution set的结果输出最终结果。说白了就是当前结果跟之前的结果和现在的数据是有关系的，是在原有结果上进行增量更改的。

需要注意的是如果是dataset的迭代需要设置终止条件，如果是stream的迭代就不需要给出终止条件。

1. flink的三种时间窗口？

①Processing time：根据task所在节点的本地时间来切分时间窗口

②event time：消息自带时间戳，但是这种时间是有延时的，也就是乱序的，为了防止同一个窗口的message被正确处理，所以需要其他方法如watermark，说白了就是给一个延时容忍度，然后根据watermark来判断窗口的划分，然后再根据trigger的类型判断什么时候进行计算

③ingestion time：有的消息本身不携带时间戳，但是用户依然希望按照消息而不是节点时钟划分窗口，在message进入flink的时候给他一个递增的时间，是event time的一种特例，用的很少

1. flink的容错机制详细讲一下？注意与spark的区别？

flink是通过checkpoint机制实现容错，它的原理是不断的生成分布式streaming数据流snapshot快照。在流处理失败时通过这些snapshot可以恢复数据流处理。而flink的快照需要记录两类东西：barrier相关和operation状态相关。

①barrier

barrier是checkpoint的核心，他会当做记录打入数据流，从而将数据流分组，并沿着数据流方向向前推荐，每个barrier会携带一个snapshotID，属于该snapshot的记录会被推向该barrier的前方。所以barrier之后的属于下一个ckeckpoint期间（snapshot中）的数据。然后当中间的operation接收到barrier后，会发送barrier到属于该barrier的snapshot的数据流中，等到sink operation接收到该barrier后会向checkpoint coordinator确认该snapshot，知道所有的sink operation都确认了该snapshot才会认为完成了本次checkpoint或者本次snapshot。

②operation 状态

operation状态分为两种：一种是系统状态：一个operator进行计算处理的时候需要对数据进行缓冲，所以数据缓冲区的状态时雨operator相关量的，以窗口操作的缓冲区为例，flink系统会收集和聚合记录数据并放到缓冲区中，知道该缓冲区中的数据被处理完成；另一种是用户自定义状态：它可以是函数中java对象这样简单的变量，也可以是与函数相关的keyvalue状态。（所以知道为什么我们在做长时间数据排序、去重等操作的时候不能用普通的数据结构存储数据了吧，因为checkpoint会记录这些变量。）

③stream aligning

这个情况出现的很少，用于解决同一个Operation处理多个输入流的情况（不是同一个数据源），这种情况下operation将先收到barrier k的数据缓存起来不进行处理，只有当另一个流的barrier k到达之后再进行处理同时opearion会向checkpoint coordinator上报snapshot。这就是barrier k对齐。

spark的checkpoint的方式没有这么复杂，直接通过记录metadata和data的方式来进行checkpoint。从checkpoint中恢复时ss是决不允许修改代码的，而sss是有些情况可以接受修改代码的。

①metadata checkpoint

将定义流式计算的信息保存到hdfs：配置、dstream操作、尚未完成的批次

②data checkpoint

这就比较直接了，直接持久化RDD到hdfs，因为我们知道spark的容错就是基于rdd的血缘关系的，而为了避免依赖关系链太长，spark会定期从最新的rdd中持久化数据到hdfs。

注意：：：如果spark程序中没有updateStateByKey或reduceByKeyAndWindow这种带有状态持续改变的算子操作的时候完全可以不用对rdd进行持久化，只需要利用metadata来恢复程序即可，因为数据的丢失时可以接受的，但是如果存在状态转换的算法就不行了。

1. flink的内存管理有什么特色？

①自带的序列化工具：序列化后的对象就是字节数组是连续存储，占用空间大大降低。又例如cpu多级缓存的命中，避免oom

使用定值的序列化工具前提是待处理的数据类型一样，这样可以再内存中存一份共享的schema，并且在操作对象时不用反序列化整个对象，而是根据字节数组的偏移量来反序列化一部分——访问对象的成员变量。

②显式内存管理：批量的申请内存和释放。避免了频繁申请释放导致的内存碎片和资源消耗，减少垃圾回收次数

③off-heap的使用：off-heap有三个特点：off-heap的数据可以与其他程序共享；off-heap的数据进行磁盘IO或者网络IO的时候支持zero-copy（零拷贝）技术，不需要至少一次的内存拷贝；off-heap可想而知可以延缓gc回收。

你既然提到了zero-copy(零拷贝)技术，你能详细说说什么是零拷贝技术吗？

首先明确两点：第一：零拷贝技术是针对内存数据的拷贝而言的；第二：正常的一次http请求-相应的过程需要四次内存拷贝的过程，服务端和客户端各两次，服务器的网线将数据写入内核缓存内存，此时cpu被迫中断执行中断进程，将数据拷贝到用户进程空间中，当处理完毕返回给客户端的时候还要讲用户空间的内存数据拷贝到内核缓冲中用于send。这样一次http请求响应就是4次拷贝。

而零拷贝zero-copy就是想不要内核内存数据向用户空间中拷贝的过程，实现：

通过找一块内存作为用户空间和内核的共享。（epoll中有应用），所以说白了零拷贝技术就是共享内存页！！！回想一下在解决fast-fail问题的时候有一个集合叫做copyonwritearraylist写时复制的技术就是应用了零拷贝技术。当发生修改list的操作的时候会fork子进程来操作，而此时并不是将整个list复制，而是只复制修改的内存页，其他内存页采用父子进程共享内存页来实现共享！

1. flink的backpressure背压机制，跟spark有什么区别？

①flink是通过自下而上的背压检测从而控制流量。如果下层的operation压力大那么上游的operation就会让它慢下来。Jobmanager会反复调用一个job的task运行所在线程的Thread.getStackTrace()，默认情况下，jobmanager会每个50ms触发对一个job的每个task依次进行100次堆栈跟踪调用，根据调用结果来确定backpressure，flink是通过计算得到一个比值radio来确定当前运行的job的backpressure状态。在web页面可以看到这个radio值，它表示在一个内部方法调用中阻塞的堆栈跟踪次数，例如radio=0.01表示100次中仅有一次方法调用阻塞。OK: 0 <= Ratio <= 0.10 LOW: 0.10 < Ratio <= 0.5 HIGH: 0.5 < Ratio <= 1。

②spark的背压检测和处理方式跟flink不同，在spark1.5之前需要自己配置接收速率来限速，所以这个值需要人为测试来决定，spark1.5版本之后只需要开启backpressure功能即可，spark会自己根据计算时间、延时时间等来确定是否限流。

1. flink的job在standalone和on yarn模式下的提交流程？

standalone：

①client会将flink代码转换成DAG jobgraph提交给jobmanager

②jobmanager会将JobGraph转换映射为一个ExecutionGrapg，区别在于JobGraph是用户层面的DAG，定点表示transformation的算子，箭头表示数据流动方向，从web页面可以看到；而executionGrapg则是并行执行这个job的DAG，其中每一个定点都代表这一个exector将要运行的task任务。

③jobmanager将task分发给taskmanager上启动的各个exector线程（这是跟spark不同的，spark是动态的，而flink是提前安排好，目的可以理解就是为了dataflow而设计的）

④在此期间各个executor会向jobmanager汇报进度、snapshot等信息

⑤任务执行完毕后jobmanager删除任务

on yarn：

如果是以session的方式预先在yarn中启动一个flink集群的话跟standalone类似，只不过运行致谢jobmanager和taskmanager是yarn中的container。

如果是直接提交job任务到yarn集群的话，yarn会启动applicationmaster用来承载jobmanager，然后动态分配taskmanager。