## 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

②一把来说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

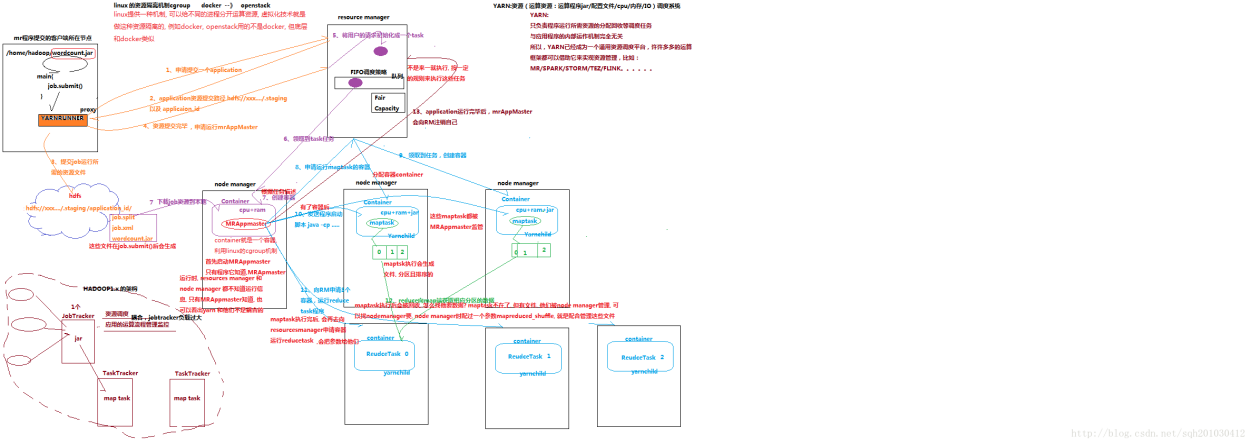
step8：注意：这里容易出错，AM申请的Map task仍然会放到RM的任务队列中等待，在map task运行之前AM显示的状态一直是accepted

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

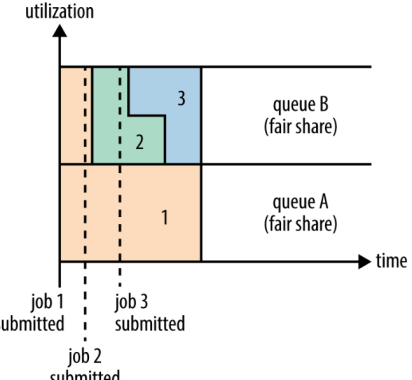
默认调度器，肯定不行

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，如果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？

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、师徒、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读次数，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：是根据列值的不通进行分区，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。

## spark部分

1. 说一说spark、flink的区别吧？

ss代表spark streaming

sss代表structured sparkstreaming

|  |  |  |
| --- | --- | --- |
|  | spark | 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 | file  spark streaming支持atleast once语义，与kafka的directconnect支持exactly once语义  ，当然也可以自己写事务来实现exactly once | 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statebackend，当barrier流动到最后sink之后，通知checkpoint完成，提交事务。 |
| 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

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

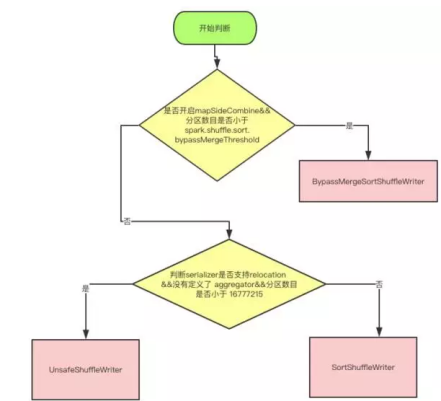
注意：bypassmergesortshufflewriter和sortshufflewriter的区别有三：其一是数据结构；其二是buffer；其三是全局排序（bypasssort只是简单的merge，而普通sort是全局sort+merge）

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+归并排序 |
| 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 |



7.