2021年超全超详细的最新大数据开发面试题，附答案解析

导语：此套面试题来自于各大厂的真实面试题及常问的知识点。

如果能理解吃透这些问题，你的大数据能力将会大大提升，进入大厂指日可待。

如果公司急招人，你回答出来面试官70%，甚至50%的问题他都会要你，如果这个公司不是真正缺人，或者只是作人才储备，那么你回答很好，他也可能不要你，只是因为没有眼缘；所以面试失败千万不要对自己的能力否定，不是你不行，而是伯乐未到。

Hadoop

hadoop中常问的有三块

第一：存储，问到存储，就把HDFS相关的知识点拿出来；

第二：计算框架(MapReduce)；

第三：资源调度框架(yarn)

1. 请说下HDFS读写流程

这个问题虽然见过无数次，面试官问过无数次，但是就是有人不能完整的说下来，所以请务必记住。并且很多问题都是从HDFS读写流程中引申出来的

HDFS写流程

1）client客户端发送上传请求，通过RPC与namenode建立通信，namenode检查该用户是否有上传权限，以及上传的文件是否在hdfs对应的目录下重名，如果这两者有任意一个不满足，则直接报错，如果两者都满足，则返回给客户端一个可以上传的信息

2）client根据文件的大小进行切分，默认128M一块，切分完成之后给namenode发送请求第一个block块上传到哪些服务器上

3）namenode收到请求之后，根据网络拓扑和机架感知以及副本机制进行文件分配，返回可用的DataNode的地址

注：Hadoop 在设计时考虑到数据的安全与高效, 数据文件默认在 HDFS 上存放三份,
存储策略为本地一份，同机架内其它某一节点上一份, 不同机架的某一节点上一份
4）客户端收到地址之后与服务器地址列表中的一个节点如A进行通信，本质上就是RPC调用，建立pipeline，A收到请求后会继续调用B，B在调用C，将整个pipeline建立完成，逐级返回client

5）client开始向A上发送第一个block（先从磁盘读取数据然后放到本地内存缓存），以packet（

数据包，64kb）为单位，A收到一个packet就会发送给B，然后B发送给C，A每传完一个packet就会放入一个应答队列等待应答

6）数据被分割成一个个的packet数据包在pipeline上依次传输，在pipeline反向传输中，逐个发送ack（命令正确应答），最终由pipeline 中第一个 DataNode 节点 A 将 pipelineack 发送给 Client

7）当一个 block 传输完成之后, Client 再次请求 NameNode 上传第二个 block ，namenode重新选择三台DataNode给client

HDFS读流程

1）client向namenode发送RPC请求。请求文件block的位置

2）namenode收到请求之后会检查用户权限以及是否有这个文件，如果都符合，则会视情况返回部分或全部的block列表，对于每个block，NameNode 都会返回含有该 block 副本的 DataNode 地址； 这些返回的 DN 地址，会按照集群拓扑结构得出 DataNode 与客户端的距离，然后进行排序，排序两个规则：网络拓扑结构中距离 Client 近的排靠前；心跳机制中超时汇报的 DN 状态为 STALE，这样的排靠后

3）Client 选取排序靠前的 DataNode 来读取 block，如果客户端本身就是DataNode,那么将从本地直接获取数据(短路读取特性)

4）底层上本质是建立 Socket Stream（FSDataInputStream），重复的调用父类 DataInputStream 的 read 方法，直到这个块上的数据读取完毕

5）当读完列表的 block 后，若文件读取还没有结束，客户端会继续向NameNode 获取下一批的 block 列表

6）读取完一个 block 都会进行 checksum 验证，如果读取 DataNode 时出现错误，客户端会通知 NameNode，然后再从下一个拥有该 block 副本的DataNode 继续读

7）read 方法是并行的读取 block 信息，不是一块一块的读取；NameNode 只是返回Client请求包含块的DataNode地址，并不是返回请求块的数据

8） 最终读取来所有的 block 会合并成一个完整的最终文件

2. HDFS在读取文件的时候,如果其中一个块突然损坏了怎么办

客户端读取完DataNode上的块之后会进行checksum 验证，也就是把客户端读取到本地的块与HDFS上的原始块进行校验，如果发现校验结果不一致，客户端会通知 NameNode，然后再从下一个拥有该 block 副本的DataNode 继续读

3. HDFS在上传文件的时候,如果其中一个DataNode突然挂掉了怎么办

客户端上传文件时与DataNode建立pipeline管道，管道正向是客户端向DataNode发送的数据包，管道反向是DataNode向客户端发送ack确认，也就是正确接收到数据包之后发送一个已确认接收到的应答，当DataNode突然挂掉了，客户端接收不到这个DataNode发送的ack确认，客户端会通知 NameNode，NameNode检查该块的副本与规定的不符，NameNode会通知DataNode去复制副本，并将挂掉的DataNode作下线处理，不再让它参与文件上传与下载。

4. NameNode在启动的时候会做哪些操作

NameNode数据存储在内存和本地磁盘，本地磁盘数据存储在fsimage镜像文件和edits编辑日志文件

首次启动NameNode
1、格式化文件系统，为了生成fsimage镜像文件
2、启动NameNode
（1）读取fsimage文件，将文件内容加载进内存
（2）等待DataNade注册与发送block report
3、启动DataNode
（1）向NameNode注册
（2）发送block report
（3）检查fsimage中记录的块的数量和block report中的块的总数是否相同
4、对文件系统进行操作（创建目录，上传文件，删除文件等）
（1）此时内存中已经有文件系统改变的信息，但是磁盘中没有文件系统改变的信息，此时会将这些改变信息写入edits文件中，edits文件中存储的是文件系统元数据改变的信息。
第二次启动NameNode
1、读取fsimage和edits文件
2、将fsimage和edits文件合并成新的fsimage文件
3、创建新的edits文件，内容为空
4、启动DataNode

5. Secondary NameNode了解吗，它的工作机制是怎样的

Secondary NameNode 是合并NameNode的edit logs到fsimage文件中；

它的具体工作机制：
（1）Secondary NameNode询问NameNode是否需要checkpoint。直接带回NameNode是否检查结果
（2）Secondary NameNode请求执行checkpoint
（3）NameNode滚动正在写的edits日志
（4）将滚动前的编辑日志和镜像文件拷贝到Secondary NameNode
（5）Secondary NameNode加载编辑日志和镜像文件到内存，并合并
（6）生成新的镜像文件fsimage.chkpoint
（7）拷贝fsimage.chkpoint到NameNode
（8）NameNode将fsimage.chkpoint重新命名成fsimage
所以如果NameNode中的元数据丢失，是可以从Secondary NameNode恢复一部分元数据信息的，但不是全部，因为NameNode正在写的edits日志还没有拷贝到Secondary

NameNode，这部分恢复不了

6. Secondary NameNode不能恢复NameNode的全部数据，那如何保证NameNode数据存储安全

这个问题就要说NameNode的高可用了，即 NameNode HA
一个NameNode有单点故障的问题，那就配置双NameNode，配置有两个关键点，一是必须要保证这两个NN的元数据信息必须要同步的，二是一个NN挂掉之后另一个要立马补上。

元数据信息同步在 HA 方案中采用的是“共享存储”。每次写文件时，需要将日志同步写入共享存储，这个步骤成功才能认定写文件成功。然后备份节点定期从共享存储同步日志，以便进行主备切换。
监控NN状态采用 zookeeper，两个NN节点的状态存放在ZK中，另外两个NN节点分别有一个进程监控程序，实施读取ZK中有NN的状态，来判断当前的NN是不是已经down机。如果standby的NN节点的ZKFC发现主节点已经挂掉，那么就会强制给原本的active
NN节点发送强制关闭请求，之后将备用的NN设置为active。
如果面试官再问HA中的 共享存储 是怎么实现的知道吗？
可以进行解释下：NameNode 共享存储方案有很多，比如 Linux HA, VMware FT, QJM等，目前社区已经把由 Clouderea 公司实现的基于 QJM（Quorum Journal Manager）的方案合并到 HDFS 的 trunk 之中并且作为默认的共享存储实现
基于 QJM 的共享存储系统主要用于保存 EditLog，并不保存 FSImage 文件。FSImage 文件还是在 NameNode 的本地磁盘上。QJM 共享存储的基本思想来自于 Paxos 算法，采用多个称为 JournalNode 的节点组成的 JournalNode 集群来存储 EditLog。每个 JournalNode 保存同样的 EditLog 副本。每次 NameNode 写 EditLog 的时候，除了向本地磁盘写入 EditLog 之外，也会并行地向 JournalNode 集群之中的每一个 JournalNode 发送写请求，只要大多数 (majority) 的 JournalNode 节点返回成功就认为向 JournalNode 集群写入 EditLog 成功。如果有 2N+1 台 JournalNode，那么根据大多数的原则，最多可以容忍有 N 台 JournalNode 节点挂掉

7. 在NameNode HA中，会出现脑裂问题吗？怎么解决脑裂

假设 NameNode1 当前为 Active 状态，NameNode2 当前为 Standby 状态。如果某一时刻 NameNode1 对应的 ZKFailoverController 进程发生了“假死”现象，那么 Zookeeper 服务端会认为 NameNode1 挂掉了，根据前面的主备切换逻辑，NameNode2 会替代 NameNode1 进入 Active 状态。但是此时 NameNode1 可能仍然处于 Active 状态正常运行，这样 NameNode1 和 NameNode2 都处于 Active 状态，都可以对外提供服务。这种情况称为脑裂

脑裂对于NameNode
这类对数据一致性要求非常高的系统来说是灾难性的，数据会发生错乱且无法恢复。Zookeeper 社区对这种问题的解决方法叫做 fencing，中文翻译为隔离，也就是想办法把旧的 Active NameNode 隔离起来，使它不能正常对外提供服务。

在进行 fencing 的时候，会执行以下的操作：

首先尝试调用这个旧 Active NameNode 的 HAServiceProtocol RPC 接口的 transitionToStandby 方法，看能不能把它转换为 Standby 状态。
如果 transitionToStandby 方法调用失败，那么就执行 Hadoop
配置文件之中预定义的隔离措施，Hadoop 目前主要提供两种隔离措施，通常会选择 sshfence：
(1) sshfence：通过 SSH 登录到目标机器上，执行命令 fuser 将对应的进程杀死
(2) shellfence：执行一个用户自定义的 shell 脚本来将对应的进程隔离

8. 小文件过多会有什么危害,如何避免

Hadoop上大量HDFS元数据信息存储在NameNode内存中,因此过多的小文件必定会压垮NameNode的内存

每个元数据对象约占150byte，所以如果有1千万个小文件，每个文件占用一个block，则NameNode大约需要2G空间。如果存储1亿个文件，则NameNode需要20G空间

显而易见的解决这个问题的方法就是合并小文件,可以选择在客户端上传时执行一定的策略先合并,或者是使用Hadoop的CombineFileInputFormat<K,V>实现小文件的合并

9. 请说下HDFS的组织架构

1）Client：客户端
（1）切分文件。文件上传HDFS的时候，Client将文件切分成一个一个的Block，然后进行存储
（2）与NameNode交互，获取文件的位置信息
（3）与DataNode交互，读取或者写入数据
（4）Client提供一些命令来管理HDFS，比如启动关闭HDFS、访问HDFS目录及内容等
2）NameNode：名称节点，也称主节点，存储数据的元数据信息，不存储具体的数据
（1）管理HDFS的名称空间
（2）管理数据块（Block）映射信息
（3）配置副本策略
（4）处理客户端读写请求
3）DataNode：数据节点，也称从节点。NameNode下达命令，DataNode执行实际的操作
（1）存储实际的数据块
（2）执行数据块的读/写操作
4）Secondary NameNode：并非NameNode的热备。当NameNode挂掉的时候，它并不能马上替换NameNode并提供服务
（1）辅助NameNode，分担其工作量
（2）定期合并Fsimage和Edits，并推送给NameNode
（3）在紧急情况下，可辅助恢复NameNode

10. 请说下MR中Map Task的工作机制

简单概述：
inputFile通过split被切割为多个split文件，通过Record按行读取内容给map（自己写的处理逻辑的方法）
，数据被map处理完之后交给OutputCollect收集器，对其结果key进行分区（默认使用的hashPartitioner），然后写入buffer，每个map task 都有一个内存缓冲区（环形缓冲区），存放着map的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式溢写到磁盘，当整个map task 结束后再对磁盘中这个maptask产生的所有临时文件做合并，生成最终的正式输出文件，然后等待reduce task的拉取

详细步骤：

读取数据组件 InputFormat (默认 TextInputFormat) 会通过 getSplits
方法对输入目录中的文件进行逻辑切片规划得到 block, 有多少个 block就对应启动多少个 MapTask.
将输入文件切分为 block 之后, 由 RecordReader 对象 (默认是LineRecordReader) 进行读取, 以 ₩n 作为分隔符, 读取一行数据, 返回 <key，value>. Key 表示每行首字符偏移值, Value 表示这一行文本内容
读取 block 返回 <key,value>, 进入用户自己继承的 Mapper 类中，执行用户重写的 map 函数, RecordReader 读取一行这里调用一次
Mapper 逻辑结束之后, 将 Mapper 的每条结果通过 context.write 进行collect数据收集. 在 collect 中, 会先对其进行分区处理，默认使用 HashPartitioner
接下来, 会将数据写入内存, 内存中这片区域叫做环形缓冲区(默认100M), 缓冲区的作用是批量收集 Mapper 结果, 减少磁盘 IO 的影响. 我们的 Key/Value 对以及 Partition 的结果都会被写入缓冲区. 当然, 写入之前，Key 与 Value 值都会被序列化成字节数组
当环形缓冲区的数据达到溢写比列(默认0.8)，也就是80M时，溢写线程启动, 需要对这 80MB 空间内的 Key 做排序 (Sort). 排序是 MapReduce 模型默认的行为,
这里的排序也是对序列化的字节做的排序
合并溢写文件, 每次溢写会在磁盘上生成一个临时文件 (写之前判断是否有 Combiner), 如果 Mapper 的输出结果真的很大, 有多次这样的溢写发生, 磁盘上相应的就会有多个临时文件存在.
当整个数据处理结束之后开始对磁盘中的临时文件进行 Merge 合并, 因为最终的文件只有一个, 写入磁盘, 并且为这个文件提供了一个索引文件, 以记录每个reduce对应数据的偏移量

11. 请说下MR中Reduce Task的工作机制

简单描述：
Reduce 大致分为 copy、sort、reduce 三个阶段，重点在前两个阶段。copy 阶段包含一个 eventFetcher 来获取已完成的 map 列表，由 Fetcher 线程去 copy 数据，在此过程中会启动两个 merge 线程，分别为 inMemoryMerger 和 onDiskMerger，分别将内存中的数据 merge 到磁盘和将磁盘中的数据进行 merge。待数据 copy 完成之后，copy 阶段就完成了，开始进行 sort 阶段，sort 阶段主要是执行 finalMerge 操作，纯粹的 sort 阶段，完成之后就是 reduce 阶段，调用用户定义的 reduce 函数进行处理
详细步骤：

Copy阶段：简单地拉取数据。Reduce进程启动一些数据copy线程(Fetcher)，通过HTTP方式请求maptask获取属于自己的文件（map task 的分区会标识每个map task属于哪个reduce task，默认reduce task的标识从0开始）。

Merge阶段：这里的merge如map端的merge动作，只是数组中存放的是不同map端copy来的数值。Copy过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比map端的更为灵活。merge有三种形式：内存到内存；内存到磁盘；磁盘到磁盘。默认情况下第一种形式不启用。当内存中的数据量到达一定阈值，就启动内存到磁盘的merge。与map 端类似，这也是溢写的过程，这个过程中如果你设置有Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种merge方式一直在运行，直到没有map端的数据时才结束，然后启动第三种磁盘到磁盘的merge方式生成最终的文件。

合并排序：把分散的数据合并成一个大的数据后，还会再对合并后的数据排序。

对排序后的键值对调用reduce方法，键相等的键值对调用一次reduce方法，每次调用会产生零个或者多个键值对，最后把这些输出的键值对写入到HDFS文件中。

12. 请说下MR中shuffle阶段

shuffle阶段分为四个步骤：依次为：分区，排序，规约，分组，其中前三个步骤在map阶段完成，最后一个步骤在reduce阶段完成
shuffle 是 Mapreduce 的核心，它分布在 Mapreduce 的 map 阶段和 reduce 阶段。一般把从 Map 产生输出开始到 Reduce 取得数据作为输入之前的过程称作 shuffle。

Collect阶段：将 MapTask 的结果输出到默认大小为 100M 的环形缓冲区，保存的是 key/value，Partition 分区信息等。
Spill阶段：当内存中的数据量达到一定的阀值的时候，就会将数据写入本地磁盘，在将数据写入磁盘之前需要对数据进行一次排序的操作，如果配置了 combiner，还会将有相同分区号和 key 的数据进行排序。
Merge阶段：把所有溢出的临时文件进行一次合并操作，以确保一个 MapTask 最终只产生一个中间数据文件
4.** Copy阶段**：ReduceTask 启动 Fetcher 线程到已经完成 MapTask 的节点上复制一份属于自己的数据，这些数据默认会保存在内存的缓冲区中，当内存的缓冲区达到一定的阀值的时候，就会将数据写到磁盘之上
Merge阶段：在 ReduceTask 远程复制数据的同时，会在后台开启两个线程对内存到本地的数据文件进行合并操作
Sort阶段：在对数据进行合并的同时，会进行排序操作，由于 MapTask 阶段已经对数据进行了局部的排序，ReduceTask 只需保证 Copy 的数据的最终整体有效性即可。
Shuffle 中的缓冲区大小会影响到 mapreduce 程序的执行效率，原则上说，缓冲区越大，磁盘io的次数越少，执行速度就越快
缓冲区的大小可以通过参数调整, 参数：mapreduce.task.io.sort.mb 默认100M

13. shuffle阶段的数据压缩机制了解吗

在shuffle阶段，可以看到数据通过大量的拷贝，从map阶段输出的数据，都要通过网络拷贝，发

送到reduce阶段，这一过程中，涉及到大量的网络IO，如果数据能够进行压缩，那么数据的发送量就会少得多。
hadoop当中支持的压缩算法：
gzip、bzip2、LZO、LZ4、Snappy，这几种压缩算法综合压缩和解压缩的速率，谷歌的Snappy是最优的，一般都选择Snappy压缩。谷歌出品，必属精品

14. 在写MR时，什么情况下可以使用规约

规约（combiner）是不能够影响任务的运行结果的，局部汇总，适用于求和类，不适用于求平均值，如果reduce的输入参数类型和输出参数的类型是一样的，则规约的类可以使用reduce类，只需要在驱动类中指明规约的类即可

15. yarn 集群的架构和工作原理知道多少

YARN的基本设计思想是将MapReduce V1中的JobTracker拆分为两个独立的服务：ResourceManager和ApplicationMaster。ResourceManager负责整个系统的资源管理和分配，ApplicationMaster负责单个应用程序的的管理。

ResourceManager：
RM是一个全局的资源管理器，负责整个系统的资源管理和分配，它主要由两个部分组成：调度器（Scheduler）和应用程序管理器（Application Manager）。
调度器根据容量、队列等限制条件，将系统中的资源分配给正在运行的应用程序，在保证容量、公平性和服务等级的前提下，优化集群资源利用率，让所有的资源都被充分利用应用程序管理器负责管理整个系统中的所有的应用程序，包括应用程序的提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重启它。

ApplicationMaster：
用户提交的一个应用程序会对应于一个ApplicationMaster，它的主要功能有：
a.与RM调度器协商以获得资源，资源以Container表示。
b.将得到的任务进一步分配给内部的任务。
c.与NM通信以启动/停止任务。
d.监控所有的内部任务状态，并在任务运行失败的时候重新为任务申请资源以重启任务。

nodeManager：
NodeManager是每个节点上的资源和任务管理器，一方面，它会定期地向RM汇报本节点上的资源使用情况和各个Container的运行状态；另一方面，他接收并处理来自AM的Container启动和停止请求。

container：
Container是YARN中的资源抽象，封装了各种资源。一个应用程序会分配一个Container，这个应用程序只能使用这个Container中描述的资源。

不同于MapReduceV1中槽位slot的资源封装，Container是一个动态资源的划分单位，更能充分利用资源。

16. yarn 的任务提交流程是怎样的

当jobclient向YARN提交一个应用程序后，YARN将分两个阶段运行这个应用程序：一是启动ApplicationMaster;第二个阶段是由ApplicationMaster创建应用程序，为它申请资源，监控运行直到结束。
具体步骤如下:

用户向YARN提交一个应用程序，并指定ApplicationMaster程序、启动ApplicationMaster的命令、用户程序。
RM为这个应用程序分配第一个Container，并与之对应的NM通讯，要求它在这个Container中启动应用程序ApplicationMaster。
ApplicationMaster向RM注册，然后拆分为内部各个子任务，为各个内部任务申请资源，并监控这些任务的运行，直到结束。
AM采用轮询的方式向RM申请和领取资源。
RM为AM分配资源，以Container形式返回
AM申请到资源后，便与之对应的NM通讯，要求NM启动任务。
NodeManager为任务设置好运行环境，将任务启动命令写到一个脚本中，并通过运行这个脚本启动任务
各个任务向AM汇报自己的状态和进度，以便当任务失败时可以重启任务。
应用程序完成后，ApplicationMaster向ResourceManager注销并关闭自己

17. yarn 的资源调度三种模型了解吗

在Yarn中有三种调度器可以选择：FIFO Scheduler ，Capacity Scheduler，Fair Scheduler

apache版本的hadoop默认使用的是capacity scheduler调度方式。CDH版本的默认使用的是fair scheduler调度方式

FIFO Scheduler（先来先服务）：

FIFO Scheduler把应用按提交的顺序排成一个队列，这是一个先进先出队列，在进行资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。

FIFO Scheduler是最简单也是最容易理解的调度器，也不需要任何配置，但它并不适用于共享集群。大的应用可能会占用所有集群资源，这就导致其它应用被阻塞，比如有个大任务在执行，占用了全部的资源，再提交一个小任务，则此小任务会一直被阻塞。

Capacity Scheduler（能力调度器）：

对于Capacity调度器，有一个专门的队列用来运行小任务，但是为小任务专门设置一个队列会预先占用一定的集群资源，这就导致大任务的执行时间会落后于使用FIFO调度器时的时间。

Fair Scheduler（公平调度器）：

在Fair调度器中，我们不需要预先占用一定的系统资源，Fair调度器会为所有运行的job动态的调整系统资源。

比如：当第一个大job提交时，只有这一个job在运行，此时它获得了所有集群资源；当第二个小任务提交后，Fair调度器会分配一半资源给这个小任务，让这两个任务公平的共享集群资源。

需要注意的是，在Fair调度器中，从第二个任务提交到获得资源会有一定的延迟，因为它需要等待第一个任务释放占用的Container。小任务执行完成之后也会释放自己占用的资源，大任务又获得了全部的系统资源。最终的效果就是Fair调度器即得到了高的资源利用率又能保证小任务及时完成。

Hive

1. hive 内部表和外部表的区别

未被external修饰的是内部表（managed table），被external修饰的为外部表（external table）

区别：

内部表数据由Hive自身管理，外部表数据由HDFS管理；
内部表数据存储的位置是hive.metastore.warehouse.dir（默认：/user/hive/warehouse），外部表数据的存储位置由自己制定（如果没有LOCATION，Hive将在HDFS上的/user/hive/warehouse文件夹下以外部表的表名创建一个文件夹，并将属于这个表的数据存放在这里）；
删除内部表会直接删除元数据（metadata）及存储数据；删除外部表仅仅会删除元数据，HDFS上的文件并不会被删除；

2. hive 有索引吗

Hive支持索引，但是Hive的索引与关系型数据库中的索引并不相同，比如，Hive不支持主键或者外键。

Hive索引可以建立在表中的某些列上，以提升一些操作的效率，例如减少MapReduce任务中需要读取的数据块的数量。

在可以预见到分区数据非常庞大的情况下，索引常常是优于分区的。

虽然Hive并不像事物数据库那样针对个别的行来执行查询、更新、删除等操作。它更多的用在多任务节点的场景下，快速地全表扫描大规模数据。但是在某些场景下，建立索引还是可以提高Hive表指定列的查询速度。（虽然效果差强人意）

索引适用的场景
适用于不更新的静态字段。以免总是重建索引数据。每次建立、更新数据后，都要重建索引以构建索引表。

Hive索引的机制如下：
hive在指定列上建立索引，会产生一张索引表（Hive的一张物理表），里面的字段包括，索引列的值、该值对应的HDFS文件路径、该值在文件中的偏移量;
v0.8后引入bitmap索引处理器，这个处理器适用于排重后，值较少的列（例如，某字段的取值只可能是几个枚举值）
因为索引是用空间换时间，索引列的取值过多会导致建立bitmap索引表过大。

但是，很少遇到hive用索引的。说明还是有缺陷or不合适的地方的。

3. 运维如何对hive进行调度

将hive的sql定义在脚本当中
使用azkaban或者oozie进行任务的调度
监控任务调度页面

4. ORC、Parquet等列式存储的优点

ORC和Parquet都是高性能的存储方式，这两种存储格式总会带来存储和性能上的提升

Parquet:
Parquet支持嵌套的数据模型，类似于Protocol Buffers，每一个数据模型的schema包含多个字段，每一个字段有三个属性：重复次数、数据类型和字段名。
重复次数可以是以下三种：required(只出现1次)，repeated(出现0次或多次)，optional(出现0次或1次)。每一个字段的数据类型可以分成两种：
group(复杂类型)和primitive(基本类型)。

Parquet中没有Map、Array这样的复杂数据结构，但是可以通过repeated和group组合来实现的。

由于Parquet支持的数据模型比较松散，可能一条记录中存在比较深的嵌套关系，如果为每一条记录都维护一个类似的树状结可能会占用较大的存储空间，因此Dremel论文中提出了一种高效的对于嵌套数据格式的压缩算法：Striping/Assembly算法。通过Striping/Assembly算法，parquet可以使用较少的存储空间表示复杂的嵌套格式，并且通常Repetition level和Definition level都是较小的整数值，可以通过RLE算法对其进行压缩，进一步降低存储空间。

Parquet文件是以二进制方式存储的，是不可以直接读取和修改的，Parquet文件是自解析的，文件
中包括该文件的数据和元数据。

ORC:
ORC文件是自描述的，它的元数据使用Protocol
Buffers序列化，并且文件中的数据尽可能的压缩以降低存储空间的消耗。

和Parquet类似，ORC文件也是以二进制方式存储的，所以是不可以直接读取，ORC文件也是自解析的，它包含许多的元数据，这些元数据都是同构ProtoBuffer进行序列化的。

ORC会尽可能合并多个离散的区间尽可能的减少I/O次数。

ORC中使用了更加精确的索引信息，使得在读取数据时可以指定从任意一行开始读取，更细粒度的统计信息使得读取ORC文件跳过整个row group，ORC默认会对任何一块数据和索引信息使用ZLIB压缩，因此ORC文件占用的存储空间也更小。

在新版本的ORC中也加入了对Bloom Filter的支持，它可以进一
步提升谓词下推的效率，在Hive 1.2.0版本以后也加入了对此的支
持。

5. 数据建模用的哪些模型？

星型模型
星形模式(Star Schema)是最常用的维度建模方式。星型模式是以事实表为中心，所有的维度表直接连接在事实表上，像星星一样。
星形模式的维度建模由一个事实表和一组维表成，且具有以下特点：
a. 维表只和事实表关联，维表之间没有关联；
b. 每个维表主键为单列，且该主键放置在事实表中，作为两边连接的外键；
c. 以事实表为核心，维表围绕核心呈星形分布；

雪花模型
雪花模式(Snowflake Schema)是对星形模式的扩展。雪花模式的维度表可以拥有其他维度表的，虽然这种模型相比星型更规范一些，但是由于这种模型不太容易理解，维护成本比较高，而且性能方面需要关联多层维表，性能也比星型模型要低。所以一般不是很常用。

星座模型
星座模式是星型模式延伸而来，星型模式是基于一张事实表的，而星座模式是基于多张事实表的，而且共享维度信息。前面介绍的两种维度建模方法都是多维表对应单事实表，但在很多时候维度空间内的事实表不止一个，而一个维表也可能被多个事实表用到。在业务发展后期，绝大部分维度建模都采用的是星座模式。

6. 为什么要对数据仓库分层？

用空间换时间，通过大量的预处理来提升应用系统的用户体验（效率），因此数据仓库会存在大量冗余的数据。

如果不分层的话，如果源业务系统的业务规则发生变化将会影响整个数据清洗过程，工作量巨大。

通过数据分层管理可以简化数据清洗的过程，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

7. 使用过Hive解析JSON串吗

hive 处理json数据总体来说有两个方向的路走
将json以字符串的方式整个入Hive表，然后通过使用UDF函数解析已经导入到hive中的数据，比如使用LATERAL VIEW json_tuple的方法，获取所需要的列名。

在导入之前将json拆成各个字段，导入Hive表的数据是已经解析过得。这将需要使用第三方的SerDe。

8. sort by 和 order by 的区别

order by 会对输入做全局排序，因此只有一个reducer（多个reducer无法保证全局有序）只有一个reducer，会导致当输入规模较大时，需要较长的计算时间。

sort by不是全局排序，其在数据进入reducer前完成排序.
因此，如果用sort by进行排序，并且设置mapred.reduce.tasks>1， 则sort by只保证每个reducer的输出有序，不保证全局有序。

9.hive优化有哪些？

数据存储及压缩。

针对hive中表的存储格式通常有orc和parquet，压缩格式一般使用snappy。相比与textfile格式表，orc占有更少的存储。因为hive底层使用MR计算架构，数据流是hdfs到磁盘再到hdfs，而且会有很多次，所以使用orc数据格式和snappy压缩策略可以降低IO读写，还能降低网络传输量，这样在一定程度上可以节省存储，还能提升hql任务执行效率；

通过调参优化。

并行执行，调节parallel参数；

调节jvm参数，重用jvm；

设置map、reduce的参数；开启strict mode模式；

关闭推测执行设置。

有效地减小数据集将大表拆分成子表；结合使用外部表和分区表。

SQL优化

大表对大表：尽量减少数据集，可以通过分区表，避免扫描全表或者全字段；

大表对小表：设置自动识别小表，将小表放入内存中去执行。

Spark

1. hadoop和spark使用场景？

Hadoop/MapReduce和Spark最适合的都是做离线型的数据分析，但Hadoop特别适合是单次分析的数据量“很大”的情景，而Spark则适用于数据量不是很大的情景。

一般情况下，对于中小互联网和企业级的大数据应用而言，单次分析的数量都不会“很大”，因此可以优先考虑使用Spark。

业务通常认为Spark更适用于机器学习之类的“迭代式”应用，80GB的压缩数据（解压后超过200GB），10个节点的集群规模，跑类似“sum+group-by”的应用，MapReduce花了5分钟，而spark只需要2分钟。

2. spark如何保证宕机迅速恢复?

适当增加spark standby master

编写shell脚本，定期检测master状态，出现宕机后对master进行重启操作

3. hadoop和spark的相同点和不同点？

Hadoop底层使用MapReduce计算架构，只有map和reduce两种操作，表达能力比较欠缺，而且在MR过程中会重复的读写hdfs，造成大量的磁盘io读写操作，所以适合高时延环境下批处理计算的应用；

Spark是基于内存的分布式计算架构，提供更加丰富的数据集操作类型，主要分成转化操作和行动操作，包括map、reduce、filter、flatmap、groupbykey、reducebykey、union和join等，数据分析更加快速，所以适合低时延环境下计算的应用；

spark与hadoop最大的区别在于迭代式计算模型。基于mapreduce框架的Hadoop主要分为map和reduce两个阶段，两个阶段完了就结束了，所以在一个job里面能做的处理很有限；spark计算模型是基于内存的迭代式计算模型，可以分为n个阶段，根据用户编写的RDD算子和程序，在处理完一个阶段后可以继续往下处理很多个阶段，而不只是两个阶段。所以spark相较于mapreduce，计算模型更加灵活，可以提供更强大的功能。

但是spark也有劣势，由于spark基于内存进行计算，虽然开发容易，但是真正面对大数据的时候，在没有进行调优的轻局昂下，可能会出现各种各样的问题，比如OOM内存溢出等情况，导致spark程序可能无法运行起来，而mapreduce虽然运行缓慢，但是至少可以慢慢运行完。

4. RDD持久化原理？

spark非常重要的一个功能特性就是可以将RDD持久化在内存中。

调用cache()和persist()方法即可。cache()和persist()的区别在于，cache()是persist()的一种简化方式，cache()的底层就是调用persist()的无参版本persist(MEMORY_ONLY)，将数据持久化到内存中。

如果需要从内存中清除缓存，可以使用unpersist()方法。RDD持久化是可以手动选择不同的策略的。在调用persist()时传入对应的StorageLevel即可。

5. checkpoint检查点机制？

应用场景：当spark应用程序特别复杂，从初始的RDD开始到最后整个应用程序完成有很多的步骤，而且整个应用运行时间特别长，这种情况下就比较适合使用checkpoint功能。

原因：对于特别复杂的Spark应用，会出现某个反复使用的RDD，即使之前持久化过但由于节点的

故障导致数据丢失了，没有容错机制，所以需要重新计算一次数据。

Checkpoint首先会调用SparkContext的setCheckPointDIR()方法，设置一个容错的文件系统的目录，比如说HDFS；然后对RDD调用checkpoint()方法。之后在RDD所处的job运行结束之后，会启动一个单独的job，来将checkpoint过的RDD数据写入之前设置的文件系统，进行高可用、容错的类持久化操作。

检查点机制是我们在spark streaming中用来保障容错性的主要机制，它可以使spark streaming阶段性的把应用数据存储到诸如HDFS等可靠存储系统中，以供恢复时使用。具体来说基于以下两个目的服务：

控制发生失败时需要重算的状态数。Spark streaming可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。

提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样spark
streaming就可以读取之前运行的程序处理数据的进度，并从那里继续。

6. checkpoint和持久化机制的区别？

最主要的区别在于持久化只是将数据保存在BlockManager中，但是RDD的lineage(血缘关系，依赖关系)是不变的。但是checkpoint执行完之后，rdd已经没有之前所谓的依赖rdd了，而只有一个强行为其设置的checkpointRDD，checkpoint之后rdd的lineage就改变了。

持久化的数据丢失的可能性更大，因为节点的故障会导致磁盘、内存的数据丢失。但是checkpoint的数据通常是保存在高可用的文件系统中，比如HDFS中，所以数据丢失可能性比较低

7. RDD机制理解吗？

rdd分布式弹性数据集，简单的理解成一种数据结构，是spark框架上的通用货币。所有算子都是基于rdd来执行的，不同的场景会有不同的rdd实现类，但是都可以进行互相转换。rdd执行过程中会形成dag图，然后形成lineage保证容错性等。从物理的角度来看rdd存储的是block和node之间的映射。

RDD是spark提供的核心抽象，全称为弹性分布式数据集。

RDD在逻辑上是一个hdfs文件，在抽象上是一种元素集合，包含了数据。它是被分区的，分为多

个分区，每个分区分布在集群中的不同结点上，从而让RDD中的数据可以被并行操作（分布式数据集）

比如有个RDD有90W数据，3个partition，则每个分区上有30W数据。RDD通常通过Hadoop上的文件，即HDFS或者HIVE表来创建，还可以通过应用程序中的集合来创建；RDD最重要的特性就是容错性，可以自动从节点失败中恢复过来。即如果某个结点上的RDD partition因为节点故障，导致数据丢失，那么RDD可以通过自己的数据来源重新计算该partition。这一切对使用者都是透明的。

RDD的数据默认存放在内存中，但是当内存资源不足时，spark会自动将RDD数据写入磁盘。比如某结点内存只能处理20W数据，那么这20W数据就会放入内存中计算，剩下10W放到磁盘中。RDD的弹性体现在于RDD上自动进行内存和磁盘之间权衡和切换的机制。

8. Spark streaming以及基本工作原理？

Spark streaming是spark core
API的一种扩展，可以用于进行大规模、高吞吐量、容错的实时数据流的处理。

它支持从多种数据源读取数据，比如Kafka、Flume、Twitter和TCP Socket，并且能够使用算子比如map、reduce、join和window等来处理数据，处理后的数据可以保存到文件系统、数据库等存储中。

Spark streaming内部的基本工作原理是：接受实时输入数据流，然后将数据拆分成batch，比如每收集一秒的数据封装成一个batch，然后将每个batch交给spark的计算引擎进行处理，最后会生产处一个结果数据流，其中的数据也是一个一个的batch组成的。

9. DStream以及基本工作原理？

DStream是spark streaming提供的一种高级抽象，代表了一个持续不断的数据流。

DStream可以通过输入数据源来创建，比如Kafka、flume等，也可以通过其他DStream的高阶函数来创建，比如map、reduce、join和window等。

DStream内部其实不断产生RDD，每个RDD包含了一个时间段的数据。

Spark streaming一定是有一个输入的DStream接收数据，按照时间划分成一个一个的batch，并转化为一个RDD，RDD的数据是分散在各个子节点的partition中。

10. spark有哪些组件？

master：管理集群和节点，不参与计算。

worker：计算节点，进程本身不参与计算，和master汇报。

Driver：运行程序的main方法，创建spark context对象。

spark context：控制整个application的生命周期，包括dagsheduler和task scheduler等组件。

client：用户提交程序的入口。

11. spark工作机制？

用户在client端提交作业后，会由Driver运行main方法并创建spark context上下文。执行add算子，形成dag图输入dagscheduler，按照add之间的依赖关系划分stage输入task scheduler。task scheduler会将stage划分为task set分发到各个节点的executor中执行。

12. 说下宽依赖和窄依赖

宽依赖：

本质就是shuffle。父RDD的每一个partition中的数据，都可能会传输一部分到下一个子RDD的每一个partition中，此时会出现父RDD和子RDD的partition之间具有交互错综复杂的关系，这种情况就叫做两个RDD之间是宽依赖。

窄依赖：

父RDD和子RDD的partition之间的对应关系是一对一的。

13. Spark主备切换机制原理知道吗？

Master实际上可以配置两个，Spark原生的standalone模式是支持Master主备切换的。当Active Master节点挂掉以后，我们可以将Standby Master切换为Active Master。

Spark Master主备切换可以基于两种机制，一种是基于文件系统的，一种是基于ZooKeeper的。

基于文件系统的主备切换机制，需要在Active Master挂掉之后手动切换到Standby Master上；

而基于Zookeeper的主备切换机制，可以实现自动切换Master。

14. spark解决了hadoop的哪些问题？

MR：抽象层次低，需要使用手工代码来完成程序编写，使用上难以上手；

Spark：Spark采用RDD计算模型，简单容易上手。

MR：只提供map和reduce两个操作，表达能力欠缺；

Spark：Spark采用更加丰富的算子模型，包括map、flatmap、groupbykey、reducebykey等；

MR：一个job只能包含map和reduce两个阶段，复杂的任务需要包含很多个job，这些job之间的管理以来需要开发者自己进行管理；

Spark：Spark中一个job可以包含多个转换操作，在调度时可以生成多个stage，而且如果多个map操作的分区不变，是可以放在同一个task里面去执行；

MR：中间结果存放在hdfs中；

Spark：Spark的中间结果一般存在内存中，只有当内存不够了，才会存入本地磁盘，而不是hdfs；

MR：只有等到所有的map task执行完毕后才能执行reduce task；

Spark：Spark中分区相同的转换构成流水线在一个task中执行，分区不同的需要进行shuffle操作，被划分成不同的stage需要等待前面的stage执行完才能执行。

MR：只适合batch批处理，时延高，对于交互式处理和实时处理支持不够；

Spark：Spark streaming可以将流拆成时间间隔的batch进行处理，实时计算。

15. 数据倾斜的产生和解决办法？

数据倾斜以为着某一个或者某几个partition的数据特别大，导致这几个partition上的计算需要耗费相当长的时间。

在spark中同一个应用程序划分成多个stage，这些stage之间是串行执行的，而一个stage里面的多个task是可以并行执行，task数目由partition数目决定，如果一个partition的数目特别大，那么导致这个task执行时间很长，导致接下来的stage无法执行，从而导致整个job执行变慢。

避免数据倾斜，一般是要选用合适的key，或者自己定义相关的partitioner，通过加盐或者哈希值来拆分这些key，从而将这些数据分散到不同的partition去执行。

如下算子会导致shuffle操作，是导致数据倾斜可能发生的关键点所在：groupByKey；reduceByKey；aggregaByKey；join；cogroup；

16. 你用sparksql处理的时候，处理过程中用的dataframe还是直接写的sql？为什么？

这个问题的宗旨是问你spark sql中dataframe和sql的区别，从执行原理、操作方便程度和自定义程度来分析这个问题。

17. 现场写一个笔试题

有hdfs文件，文件每行的格式为作品ID，用户id，用户性别。请用一个spark任务实现以下功能：统计每个作品对应的用户（去重后）的性别分布。输出格式如下：作品ID，男性用户数量，女性用户数量

答案：

```
sc.textfile() .flatmap(.split(","))//分割成作品ID，用户id，用户性别
.map(((_.1,_._2),1))//((作品id,用户性别),1)
.reduceByKey(_+_)//((作品id,用户性别),n)
```

.map(_._1._1,_._1._2,_._2)//(作品id,用户性别,n)
1
2
3
4

18. RDD中reduceBykey与groupByKey哪个性能好，为什么

reduceByKey：reduceByKey会在结果发送至reducer之前会对每个mapper在本地进行merge，有点类似于在MapReduce中的combiner。这样做的好处在于，在map端进行一次reduce之后，数据量会大幅度减小，从而减小传输，保证reduce端能够更快的进行结果计算。

groupByKey：groupByKey会对每一个RDD中的value值进行聚合形成一个序列(Iterator)，此操作发生在reduce端，所以势必会将所有的数据通过网络进行传输，造成不必要的浪费。同时如果数据量十分大，可能还会造成OutOfMemoryError。

所以在进行大量数据的reduce操作时候建议使用reduceByKey。不仅可以提高速度，还可以防止使用groupByKey造成的内存溢出问题。

19. Spark master HA主从切换过程不会影响到集群已有作业的运行，为什么？

不会的。

因为程序在运行之前，已经申请过资源了，driver和Executors通讯，不需要和master进行通讯的。

20. spark master使用zookeeper进行ha，有哪些源数据保存到Zookeeper里面

spark通过这个参数spark.deploy.zookeeper.dir指定master元数据在zookeeper中保存的位置，包括Worker，Driver和Application以及Executors。standby节点要从zk中，获得元数据信息，恢复集群运行状态，才能对外继续提供服务，作业提交资源申请等，在恢复前是不能接受请求的。

注：Master切换需要注意2点：
1、在Master切换的过程中，所有的已经在运行的程序皆正常运行！
因为Spark Application在运行前就已经通过Cluster

Manager获得了计算资源，所以在运行时Job本身的
调度和处理和Master是没有任何关系。
2、在Master的切换过程中唯一的影响是不能提交新的Job：一方面不能够提交新的应用程序给集群，
因为只有Active
Master才能接受新的程序的提交请求；另外一方面，已经运行的程序中也不能够因
Action操作触发新的Job的提交请求。

Kafka

1. 为什么要使用 kafka？

缓冲和削峰：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理。

解耦和扩展性：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。

冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。

健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。

异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

2. Kafka消费过的消息如何再消费？

kafka消费消息的offset是定义在zookeeper中的， 如果想重复消费kafka的消息，可以在redis中自己记录offset的checkpoint点（n个），当想重复消费消息时，通过读取redis中的checkpoint点进行zookeeper的offset重设，这样就可以达到重复消费消息的目的了

3. kafka的数据是放在磁盘上还是内存上，为什么速度会快？

kafka使用的是磁盘存储。

速度快是因为：

顺序写入：因为硬盘是机械结构，每次读写都会寻址->写入，其中寻址是一个“机械动作”，它是耗时的。所以硬盘 “讨厌”随机I/O，
喜欢顺序I/O。为了提高读写硬盘的速度，Kafka就是使用顺序I/O。
Memory Mapped Files（内存映射文件）：64位操作系统中一般可以表示20G的数据文件，它的工作原理是直接利用操作系统的Page来实现文件到物理内存的直接映射。完成映射之后你对物理内存的操作会被同步到硬盘上。
Kafka高效文件存储设计： Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。通过索引信息可以快速定位message和确定response的 大 小。通过index元数据全部映射到memory（内存映射文件），
可以避免segment
file的IO磁盘操作。通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。
注：

Kafka解决查询效率的手段之一是将数据文件分段，比如有100条Message，它们的offset是从0到99。假设将数据文件分成5段，第一段为0-19，第二段为20-39，以此类推，每段放在一个单独的数据文件里面，数据文件以该段中 小的offset命名。这样在查找指定offset的
Message的时候，用二分查找就可以定位到该Message在哪个段中。
为数据文件建 索引数据文件分段 使得可以在一个较小的数据文件中查找对应offset的Message了，但是这依然需要顺序扫描才能找到对应offset的Message。
为了进一步提高查找的效率，Kafka为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为.index。

4. Kafka数据怎么保障不丢失？

分三个点说，一个是生产者端，一个消费者端，一个broker端。

生产者数据的不丢失
kafka的ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到，其中状态有0，1，-1。

如果是同步模式：
ack设置为0，风险很大，一般不建议设置为0。即使设置为1，也会随着leader宕机丢失数据。所以如果要严格保证生产端数据不丢失，可设置为-1。

如果是异步模式：
也会考虑ack的状态，除此之外，异步模式下的有个buffer，通过buffer来进行控制数据的发送，有两个值来进行控制，时间阈值与消息的数量阈值，如果buffer满了数据还没有发送出去，有个选项是配置是否立即清空buffer。可以设置为-1，永久阻塞，也就数据不再生产。异步模式下，即使设置为-1。也可能因为程序员的不科学操作，操作数据丢失，比如kill
-9，但这是特别的例外情况。

注：
ack=0：producer不等待broker同步完成的确认，继续发送下一条(批)信息。
ack=1（默认）：producer要等待leader成功收到数据并得到确认，才发送下一条message。
ack=-1：producer得到follwer确认，才发送下一条数据。

消费者数据的不丢失
通过offset commit 来保证数据的不丢失，kafka自己记录了每次消费的offset数值，下次继续消费的时候，会接着上次的offset进行消费。

而offset的信息在kafka0.8版本之前保存在zookeeper中，在0.8版本之后保存到topic中，即使消费者在运行过程中挂掉了，再次启动的时候会找到offset的值，找到之前消费消息的位置，接着消费，由于 offset 的信息写入的时候并不是每条消息消费完成后都写入的，所以这种情况有可能会造成重复消费，但是不会丢失消息。

唯一例外的情况是，我们在程序中给原本做不同功能的两个consumer组设置KafkaSpoutConfig.bulider.setGroupid的时候设置成了一样的groupid，这种情况会导致这两个组共享同一份数据，就会产生组A消费partition1，partition2中的消息，组B消费partition3的消息，这样每个组消费的消息都会丢失，都是不完整的。
为了保证每个组都独享一份消息数据，groupid一定不要重复才行。

kafka集群中的broker的数据不丢失
每个broker中的partition我们一般都会设置有replication（副本）的个数，生产者写入的时候首先根据分发策略（有partition按partition，有key按key，都没有轮询）写入到leader中，follower（副本）再跟leader同步数据，这样有了备份，也可以保证消息数据的不丢失。

5. 采集数据为什么选择kafka？

采集层 主要可以使用Flume, Kafka等技术。

Flume：Flume 是管道流方式，提供了很多的默认实现，让用户通过参数部署，及扩展API.

Kafka：Kafka是一个可持久化的分布式的消息队列。 Kafka
是一个非常通用的系统。你可以有许多生产者和很多的消费者共享多个主题Topics。

相比之下,Flume是一个专用工具被设计为旨在往HDFS，HBase发送数据。它对HDFS有特殊的优

化，并且集成了Hadoop的安全特性。

所以，Cloudera
建议如果数据被多个系统消费的话，使用kafka；如果数据被设计给Hadoop使用，使用Flume。

6. kafka 重启是否会导致数据丢失？

kafka是将数据写到磁盘的，一般数据不会丢失。
但是在重启kafka过程中，如果有消费者消费消息，那么kafka如果来不及提交offset，可能会造成数据的不准确（丢失或者重复消费）。

7. kafka 宕机了如何解决？

先考虑业务是否受到影响
kafka 宕机了，首先我们考虑的问题应该是所提供的服务是否因为宕机的机器而受到影响，如果服务提供没问题，如果实现做好了集群的容灾机制，那么这块就不用担心了。

节点排错与恢复
想要恢复集群的节点，主要的步骤就是通过日志分析来查看节点宕机的原因，从而解决，重新恢复节点。

8. 为什么Kafka不支持读写分离？

在 Kafka 中，生产者写入消息、消费者读取消息的操作都是与 leader 副本进行交互的，从而实现的是一种主写主读的生产消费模型。
Kafka 并不支持主写从读，因为主写从读有 2 个很明显的缺点:

数据一致性问题：数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间窗口会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中 A 数据的值都为 X，之后将主节点中 A 的值修改为 Y，那么在这个变更通知到从节点之前，应用读取从节点中的 A 数据的值并不为最新的 Y，由此便产生了数据不一致的问题。

延时问题：类似 Redis 这种组件，数据从写入主节点到同步至从节点中的过程需要经历网络→主节点内存→网络→从节点内存 这几个阶段，整个过程会耗费一定的时间。而在 Kafka 中，主从同步会比 Redis 更加耗时，它需要经历 网络→主节点内存→主节点磁盘→网络→从节

点内存→从节点磁盘 这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

而kafka的主写主读的优点就很多了：

可以简化代码的实现逻辑，减少出错的可能;
将负载粒度细化均摊，与主写从读相比，不仅负载效能更好，而且对用户可控;
没有延时的影响;
在副本稳定的情况下，不会出现数据不一致的情况。

9. kafka数据分区和消费者的关系？

每个分区只能由同一个消费组内的一个消费者(consumer)来消费，可以由不同的消费组的消费者来消费，同组的消费者则起到并发的效果。

10. kafka的数据offset读取流程

连接ZK集群，从ZK中拿到对应topic的partition信息和partition的Leader的相关信息

连接到对应Leader对应的broker

consumer将□自□己保存的offset发送给Leader

Leader根据offset等信息定位到segment（索引□文件和□日志□文件）

根据索引□文件中的内容，定位到□日志□文件中该偏移量量对应的开始位置读取相应□长度的数据并返回给consumer

11. kafka内部如何保证顺序，结合外部组件如何保证消费者的顺序？

kafka只能保证partition内是有序的，但是partition间的有序是没办法的。爱奇艺的搜索架构，是从业务上把需要有序的打到同□个partition。

12. Kafka消息数据积压，Kafka消费能力不足怎么处理？

如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数=分区数。（两者缺一不可）

如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少（拉取数据/处理时间<生产速度），使处理的数据小于生产的数据，也会造成数据积压。

13. Kafka单条日志传输大小

kafka对于消息体的大小默认为单条最大值是1M但是在我们应用场景中, 常常会出现一条消息大于1M，如果不对kafka进行配置。则会出现生产者无法将消息推送到kafka或消费者无法去消费kafka里面的数据, 这时我们就要对kafka进行以下配置：server.properties

replica.fetch.max.bytes: 1048576 broker可复制的消息的最大字节数, 默认为1M
message.max.bytes: 1000012 kafka 会接收单个消息size的最大限制， 默认为1M左右
1
2
注意：message.max.bytes必须小于等于replica.fetch.max.bytes，否则就会导致replica之间数据同步失败。

Hbase

1. Hbase是怎么写数据的？

Client写入 -> 存入MemStore，一直到MemStore满 ->
Flush成一个StoreFile，直至增长到一定阈值 -> 触发Compact合并操作 ->
多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除 -> 当StoreFiles
Compact后，逐步形成越来越大的StoreFile ->
单个StoreFile大小超过一定阈值后（默认10G），触发Split操作，把当前Region Split成2个Region，Region会下线，新Split出的2个孩子Region会被HMaster分配到相应的HRegionServer
上，使得原先1个Region的压力得以分流到2个Region上

由此过程可知，HBase只是增加数据，没有更新和删除操作，用户的更新和删除都是逻辑层面的，在物理层面，更新只是追加操作，删除只是标记操作。

用户写操作只需要进入到内存即可立即返回，从而保证I/O高性能。

2. HDFS和HBase各自使用场景

首先一点需要明白：Hbase是基于HDFS来存储的。

HDFS：

一次性写入，多次读取。

保证数据的一致性。

主要是可以部署在许多廉价机器中，通过多副本提高可靠性，提供了容错和恢复机制。

HBase：

瞬间写入量很大，数据库不好支撑或需要很高成本支撑的场景。

数据需要长久保存，且量会持久增长到比较大的场景。

HBase不适用与有 join，多级索引，表关系复杂的数据模型。

大数据量（100s TB级数据）且有快速随机访问的需求。如：淘宝的交易历史记录。数据量巨大无容置疑，面向普通用户的请求必然要即时响应。

业务场景简单，不需要关系数据库中很多特性（例如交叉列、交叉表，事务，连接等等）。

3. Hbase的存储结构

Hbase 中的每张表都通过行键(rowkey)按照一定的范围被分割成多个子表（HRegion），默认一个HRegion 超过256M 就要被分割成两个，由HRegionServer管理，管理哪些 HRegion 由 Hmaster 分配。 HRegion 存取一个子表时，会创建一个 HRegion
对象，然后对表的每个列族（Column Family）创建一个 store 实例， 每个 store 都会有 0
个或多个 StoreFile 与之对应，每个 StoreFile 都会对应一个HFile，HFile
就是实际的存储文件，一个 HRegion 还拥有一个 MemStore实例。

4. 热点现象（数据倾斜）怎么产生的，以及解决方法有哪些

热点现象：

某个小的时段内，对HBase的读写请求集中到极少数的Region上，导致这些region所在的RegionServer处理请求量骤增，负载量明显偏大，而其他的RgionServer明显空闲。

热点现象出现的原因：

HBase中的行是按照rowkey的字典顺序排序的，这种设计优化了scan操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于scan。然而糟糕的rowkey设计是热点的源头。

热点发生在大量的client直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点region所在的单个机器超出自身承受能力，引起性能下降甚至region不可用，这也会影响同一个RegionServer上的其他region，由于主机无法服务其他region的请求。

热点现象解决办法：

为了避免写热点，设计rowkey使得不同行在同一个region，但是在更多数据情况下，数据应该被写入集群的多个region，而不是一个。常见的方法有以下这些：

加盐：在rowkey的前面增加随机数，使得它和之前的rowkey的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的region的数量一致。加盐之后的rowkey就会根据随机生成的前缀分散到各个region上，以避免热点。

哈希：哈希可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的rowkey，可以使用get操作准确获取某一个行数据

反转：第三种防止热点的方法时反转固定长度或者数字格式的rowkey。这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机rowkey，但是牺牲了rowkey的有序性。反转rowkey的例子以手机号为rowkey，可以将手机号反转后的字符串作为rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题

时间戳反转：一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为rowkey的一部分对这个问题十分有用，可以用 Long.Max_Value - timestamp
追加到key的末尾，例如[key][reverse_timestamp],[key]的最新值可以通过scan
[key]获得[key]的第一条记录，因为HBase中rowkey是有序的，第一条记录是最后录入的数据。

比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计rowkey的时候，可以这样设计[userId反转]
[Long.Max_Value - timestamp]，在查询用户的所有操作记录数据的时候，直接指定反转后的userId，startRow是[userId反转][000000000000],stopRow是[userId反转][Long.Max_Value - timestamp]

如果需要查询某段时间的操作记录，startRow是[user反转][Long.Max_Value - 起始时间]，stopRow是[userId反转][Long.Max_Value - 结束时间]

HBase建表预分区：创建HBase表时，就预先根据可能的RowKey划分出多个region而不是默认的一个，从而可以将后续的读写操作负载均衡到不同的region上，避免热点现象。

5. HBase的 rowkey 设计原则

长度原则：100字节以内，8的倍数最好，可能的情况下越短越好。因为HFile是按照 keyvalue 存储的，过长的rowkey会影响存储效率；其次，过长的rowkey在memstore中较大，影响缓冲效果，降低检索效率。最后，操作系统大多为64位，8的倍数，充分利用操作系统的最佳性能。

散列原则：高位散列，低位时间字段。避免热点问题。

唯一原则：分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

6. HBase的列簇设计

原则：在合理范围内能尽量少的减少列簇就尽量减少列簇，因为列簇是共享region的，每个列簇数据相差太大导致查询效率低下。

最优：将所有相关性很强的 key-value 都放在同一个列簇下，这样既能做到查询效率最高，也能保持尽可能少的访问不同的磁盘文件。以用户信息为例，可以将必须的基本信息存放在一个列族，而一些附加的额外信息可以放在另一列族。

7. HBase 中 compact 用途是什么，什么时候触发，分为哪两种，有什么区别

在 hbase 中每当有 memstore 数据 flush 到磁盘之后，就形成一个 storefile，当 storeFile的数量达到一定程度后，就需要将 storefile 文件来进行 compaction 操作。

Compact 的作用：

合并文件

清除过期，多余版本的数据

提高读写数据的效率
4
HBase 中实现了两种 compaction 的方式：minor and major. 这两种 compaction 方式的区别是：

Minor 操作只用来做部分文件的合并操作以及包括 minVersion=0 并且设置 ttl 的过期版本清理，不做任何删除数据、多版本数据的清理工作。

Major 操作是对 Region 下的 HStore 下的所有 StoreFile 执行合并操作，最终的结果是整理合并出一个文件。

Flink

1. Flink 的容错机制（checkpoint）

Flink可靠性的基石-checkpoint机制详细解析

2. Flink checkpoint与 Spark Flink 有什么区别或优势吗

spark streaming 的 checkpoint 仅仅是针对 driver 的故障恢复做了数据和元数据的 checkpoint。而 flink 的 checkpoint 机制 要复杂了很多，它采用的是轻量级的分布式快照，实现了每个算子的快照，及流动中的数据的快照。

3. Flink 中的 Time 有哪几种

在flink中被划分为事件时间，提取时间，处理时间三种。

如果以EventTime为基准来定义时间窗口那将形成EventTimeWindow,要求消息本身就应该携带EventTime。

如果以IngesingtTime为基准来定义时间窗口那将形成IngestingTimeWindow,以source的systemTime为准。

如果以ProcessingTime基准来定义时间窗口那将形成ProcessingTimeWindow，以operator的systemTime为准。

4. 对于迟到数据是怎么处理的

Flink中 WaterMark 和 Window 机制解决了流式数据的乱序问题，对于因为延迟而顺序有误的数据，可以根据eventTime进行业务处理，对于延迟的数据Flink也有自己的解决办法，主要的办法是给定一个允许延迟的时间，在该时间范围内仍可以接受处理延迟数据

设置允许延迟的时间是通过allowedLateness(lateness: Time)设置

保存延迟数据则是通过sideOutputLateData(outputTag: OutputTag[T])保存

获取延迟数据是通过DataStream.getSideOutput(tag: OutputTag[X])获取

5. Flink 的运行必须依赖 Hadoop组件吗

Flink可以完全独立于Hadoop，在不依赖Hadoop组件下运行。但是做为大数据的基础设施，Hadoop体系是任何大数据框架都绕不过去的。Flink可以集成众多Hadooop 组件，例如Yarn、Hbase、HDFS等等。例如，Flink可以和Yarn集成做资源调度，也可以读写HDFS，或者利用HDFS做检查点。

6. Flink集群有哪些角色？各自有什么作用

有以下三个角色：

JobManager处理器：

也称之为Master，用于协调分布式执行，它们用来调度task，协调检查点，协调失败时恢复等。Flink运行时至少存在一个master处理器，如果配置高可用模式则会存在多个master处理器，它们其中有一个是leader，而其他的都是standby。

TaskManager处理器：

也称之为Worker，用于执行一个dataflow的task(或者特殊的subtask)、数据缓冲和data stream的交换，Flink运行时至少会存在一个worker处理器。

Clint客户端：

Client是Flink程序提交的客户端，当用户提交一个Flink程序时，会首先创建一个Client，该Client首先会对用户提交的Flink程序进行预处理，并提交到Flink集群中处理，所以Client需要从用户提交的Flink程序配置中获取JobManager的地址，并建立到JobManager的连接，将Flink Job提交给JobManager

7. Flink 资源管理中 Task Slot 的概念

在Flink中每个TaskManager是一个JVM的进程, 可以在不同的线程中执行一个或多个子任务。为了控制一个worker能接收多少个task。worker通过task slot（任务槽）来进行控制（一个worker至少有一个task slot）。

8. Flink的重启策略了解吗

Flink支持不同的重启策略，这些重启策略控制着job失败后如何重启：

固定延迟重启策略
固定延迟重启策略会尝试一个给定的次数来重启Job，如果超过了最大的重启次数，Job最终将失败。在连续的两次重启尝试之间，重启策略会等待一个固定的时间。

失败率重启策略
失败率重启策略在Job失败后会重启，但是超过失败率后，Job会最终被认定失败。在两个连续的重启尝试之间，重启策略会等待一个固定的时间。

无重启策略

Job直接失败，不会尝试进行重启。

9. Flink是如何保证Exactly-once语义的

Flink通过实现两阶段提交和状态保存来实现端到端的一致性语义。分为以下几个步骤：

开始事务（beginTransaction）创建一个临时文件夹，来写把数据写入到这个文件夹里面

预提交（preCommit）将内存中缓存的数据写入文件并关闭

正式提交（commit）将之前写完的临时文件放入目标目录下。这代表着最终的数据会有一些延迟

丢弃（abort）丢弃临时文件

若失败发生在预提交成功后，正式提交前。可以根据状态来提交预提交的数据，也可删除预提交的数据。

10. 如果下级存储不支持事务，Flink 怎么保证 exactly-once

端到端的exactly-once对sink要求比较高，具体实现主要有幂等写入和事务性写入两种方式。

幂等写入的场景依赖于业务逻辑，更常见的是用事务性写入。而事务性写入又有预写日志（WAL）和两阶段提交（2PC）两种方式。

如果外部系统不支持事务，那么可以用预写日志的方式，把结果数据先当成状态保存，然后在收到 checkpoint 完成的通知时，一次性写入 sink 系统。

11. Flink是如何处理反压的

Flink 内部是基于 producer-consumer
模型来进行消息传递的，Flink的反压设计也是基于这个模型。Flink
使用了高效有界的分布式阻塞队列，就像 Java
通用的阻塞队列（BlockingQueue）一样。下游消费者消费变慢，上游就会受到阻塞。

12. Flink中的状态存储

Flink在做计算的过程中经常需要存储中间状态，来避免数据丢失和状态恢复。选择的状态存储策略不同，会影响状态持久化如何和 checkpoint 交互。Flink提供了三种状态存储方式：MemoryStateBackend、FsStateBackend、RocksDBStateBackend。

13. Flink是如何支持批流一体的

这道题问的比较开阔，如果知道Flink底层原理，可以详细说说，如果不是很了解，就直接简单一句话：Flink的开发者认为批处理是流处理的一种特殊情况。批处理是有限的流处理。Flink使用一个引擎支持了DataSet API 和 DataStream API。

14. Flink的内存管理是如何做的

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上。此外，Flink大量的使用了堆外内存。如果需要处理的数据超出了内存限制，则会将部分数据存储到硬盘上。Flink 为了直接操作二进制数据实现了自己的序列化框架。

15. Flink CEP 编程中当状态没有到达的时候会将数据保存在哪里

在流式处理中，CEP 当然是要支持 EventTime 的，那么相对应的也要支持数据的迟到现象，也就是watermark的处理逻辑。CEP对未匹配成功的事件序列的处理，和迟到数据是类似的。在 Flink CEP的处理逻辑中，状态没有满足的和迟到的数据，都会存储在一个Map数据结构中，也就是说，如果我们限定判断事件序列的时长为5分钟，那么内存中就会存储5分钟的数据，这在我看来，也是对内存的极大损伤之一。

业务方面

1. 在处理大数据过程中，如何保证得到期望值

保证在数据采集的时候不丢失数据，这个尤为重要，如果在数据采集的时候就已经不准确，后面很难达到期望值

在数据处理的时候不丢失数据，例如sparkstreaming处理kafka数据的时候，要保证数据不丢失，这个尤为重要

前两步中，如果无法保证数据的完整性，那么就要通过离线计算进行数据的校对，这样才能保证我们能够得到期望值

2. 你感觉数仓建设中最重要的是什么

数仓建设中，最重要的是数据准确性，数据的真正价值在于数据驱动决策，通过数据指导运营，在一个不准确的数据驱动下，得到的一定是错误的数据分析，影响的是公司的业务发展决策，最终导致公司的策略调控失败。

3. 数据仓库建模怎么做的

数仓建设中最常用模型–Kimball维度建模详解

4. 数据质量怎么监控

单表数据量监控

一张表的记录数在一个已知的范围内，或者上下浮动不会超过某个阈值

SQL结果：var 数据量 = select count (*) from 表 where 时间等过滤条件

报警触发条件设置：如果数据量不在[数值下限, 数值上限]， 则触发报警

同比增加：如果((本周的数据量 -上周的数据量)/上周的数据量*100)不在
[比例下线，比例上限]，则触发报警

环比增加：如果((今天的数据量 - 昨天的数据量)/昨天的数据量*100)不在
[比例下线，比例上限]，则触发报警

报警触发条件设置一定要有。如果没有配置的阈值，不能做监控
日活、周活、月活、留存（日周月）、转化率（日、周、月）GMV（日、周、月）
复购率（日周月）

单表空值检测

某个字段为空的记录数在一个范围内，或者占总量的百分比在某个阈值范围内

目标字段：选择要监控的字段，不能选“无”

SQL结果：var 异常数据量 = select count(*) from 表 where 目标字段 is null

单次检测：如果(异常数据量)不在[数值下限, 数值上限]，则触发报警

单表重复值检测

一个或多个字段是否满足某些规则

目标字段：第一步先正常统计条数；select count(*) form 表；

第二步，去重统计；select count(*) from 表 group by 某个字段

第一步的值和第二步不的值做减法，看是否在上下线阀值之内

单次检测：如果(异常数据量)不在[数值下限, 数值上限]， 则触发报警

跨表数据量对比

主要针对同步流程，监控两张表的数据量是否一致

SQL结果：count(本表) - count(关联表)

阈值配置与“空值检测”相同

5. 数据分析方法论了解过哪些？

数据商业分析的目标是利用大数据为所有职场人员做出迅捷，高质，高效的决策提供可规模化的解决方案。商业分析是创造价值的数据科学。

数据商业分析中会存在很多判断：

观察数据当前发生了什么？
比如想知道线上渠道A、B各自带来了多少流量，新上线的产品有多少用户喜欢，新注册流中注册的人数有多少。这些都需要通过数据来展示结果。

理解为什么发生？
我们需要知道渠道A为什么比渠道B好，这些是要通过数据去发现的。也许某个关键字带来的流量转化率比其他都要低，这时可以通过信息、知识、数据沉淀出发生的原因是什么。

预测未来会发生什么？
在对渠道A、B有了判断之后，根据以往的知识预测未来会发生什么。在投放渠道C、D的时候，猜测渠道C比渠道D好，当上线新的注册流、新的优化，可以知道哪一个节点比较容易出问题，这些都是通过数据进行预测的过程。

商业决策
所有工作中最有意义的还是商业决策，通过数据来判断应该做什么。这是商业分析最终的目的。

原文链接：https://blog.csdn.net/helloHbulie/article/details/113555820

本博客文章除特别声明，全部都是原创！
原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）