2022最新大数据必备面试题

大数据面试宝典

第3版

五分钟学大数据 出品
微信扫码关注

五分钟学大数据

目录

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

复习大数据面试题，看这一套就够了！

Hadoop

Hadoop 中常问的就三块，第一：分布式存储(HDFS)；第二：分布式计算框架(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，还是有不少面试者不能完整的说出来，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DataNode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 DataNode 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，管道的正方向是客户端向 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：
   - 读取 fsimage 文件，将文件内容加载进内存
   - 等待 DataNade 注册与发送 block report
3. 启动 DataNode：
   - 向 NameNode 注册
   - 发送 block report
   - 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）：

- 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，内容开始为空；
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 NameNode 的元数据信息必须要同步的，二是一个 NameNode 挂掉之后另一个要立马补上。

1. **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。
2. 监控 NameNode 状态采用 zookeeper，两个 NameNode 节点的状态存放在 zookeeper 中，另外两个 NameNode 节点分别有一个进程监控程序，实施读取 zookeeper 中有 NameNode 的状态，来判断当前的 NameNode 是不是已经 down 机。如果 Standby 的 NameNode 节点的 ZKFC 发现主节点已经挂掉，那么就会强制给原本的 Active NameNode 节点发送强制关闭请求，之后将备用的 NameNode 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：

1. 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。
2. 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：
    - 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；
    - 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
    - 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
    - 与 NameNode 交互，获取文件的位置信息
    - 与 DataNode 交互，读取或者写入数据

- 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
    - 管理 HDFS 的名称空间
    - 管理数据块（Block）映射信息
    - 配置副本策略
    - 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
    - 存储实际的数据块
    - 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
    - 辅助 NameNode，分担其工作量
    - 定期合并 Fsimage 和 Edits，并推送给 NameNode
    - 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：

1. 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。

2. 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。
3. 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。
4. 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。
5. **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**。
6. 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。
7. 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：

1. 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）。
2. Merge 阶段：在远程拷贝数据的同时，ReduceTask 启动了两个后台线程对内存和磁盘上的文件进行合并，以防止内存使用过多或磁盘上文件过多。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，就直接启动内存到磁盘的 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。内存到磁盘的 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。
3. 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。
4. 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。

1. 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。
2. 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。
3. MapTask 阶段的 Merge：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。
5. ReduceTask 阶段的 Merge：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。
6. 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。

1. 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。

2. 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：
   - 与 RM 调度器协商以获得资源，资源以 Container 表示。
   - 将得到的任务进一步分配给内部的任务。
   - 与 NM 通信以启动/停止任务。
   - 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。
3. 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。
4. 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:

1. 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。
2. 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。

3. 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。
4. AM 采用轮询的方式向 RM 申请和领取资源。
5. RM 为 AM 分配资源，以 Container 形式返回。
6. AM 申请到资源后，便与之对应的 NM 通讯，要求 NM 启动任务。
7. 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。
8. 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。
9. 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，被 external 修饰的为外部表。

区别：

1. 内部表数据由 Hive 自身管理，外部表数据由 HDFS 管理；
2. 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）；

3. 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（3.0 版本之前），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。并且 Hive 索引提供的功能很有限，效率也并不高，因此 Hive 索引很少使用。

- 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。Hive 0.8 版本后引入 bitmap 索引处理器，这个处理器适用于去重后，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。

注意：Hive 中每次有数据时需要及时更新索引，相当于重建一个新表，否则会影响数据查询的效率和准确性，Hive 官方文档已经明确表示 Hive 的索引不推荐被使用，在新版本的 Hive 中已经被废弃了。

扩展：Hive 是在 0.7 版本之后支持索引的，在 0.8 版本后引入 bitmap 索引处理器，在 3.0 版本开始移除索引的功能，取而代之的是 2.3 版本开始的物化视图，自动重写的物化视图替代了索引的功能。

3. 运维如何对 Hive 进行调度

1. 将 hive 的 sql 定义在脚本当中；
2. 使用 azkaban 或者 oozie 进行任务的调度；
3. 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:

1. 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)。
2. 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。
3. 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。
4. 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。

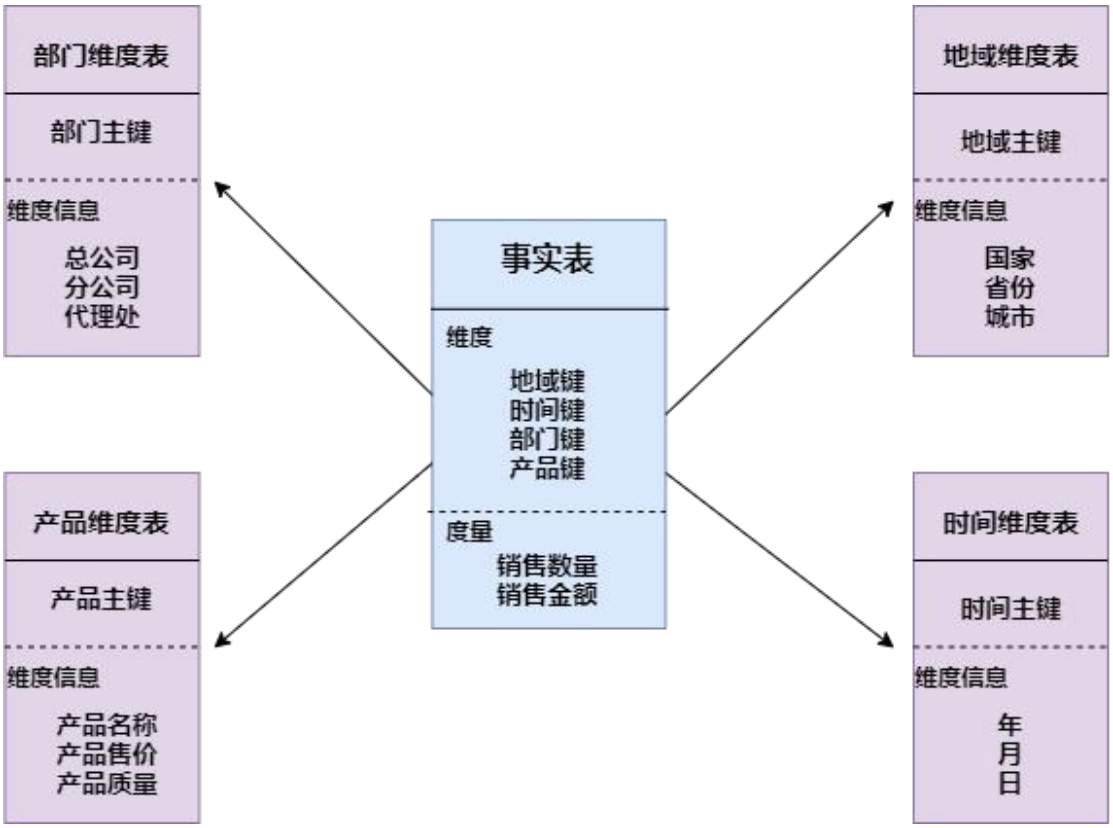
ORC:

1. 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。
2. 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。
3. ORC 会尽可能合并多个离散的区间尽可能的减少 I/O 次数。
4. 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。

5. 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？

1. 星型模型



星形模式

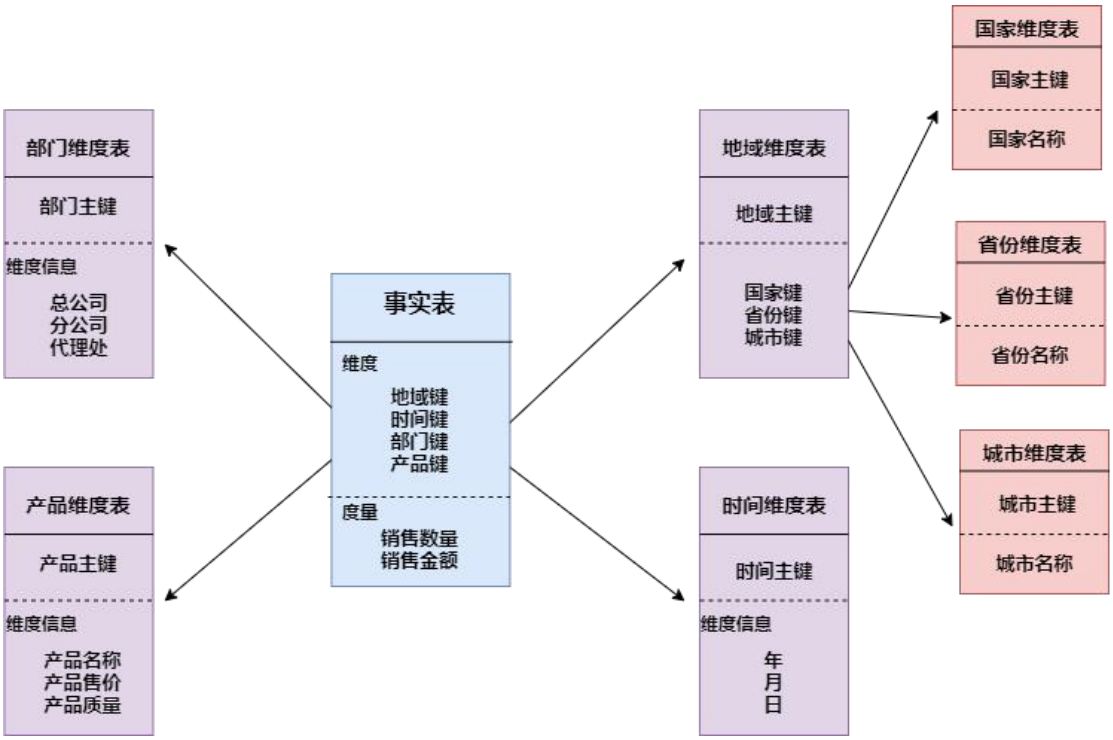
星形模式(Star Schema)是最常用的维度建模方式。星型模式是以事实表为中心，所有的维度表直接连接在事实表上，像星星一样。 星形模式的维度建模由一个事实表和一组维表成，且具有以下特点：

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。

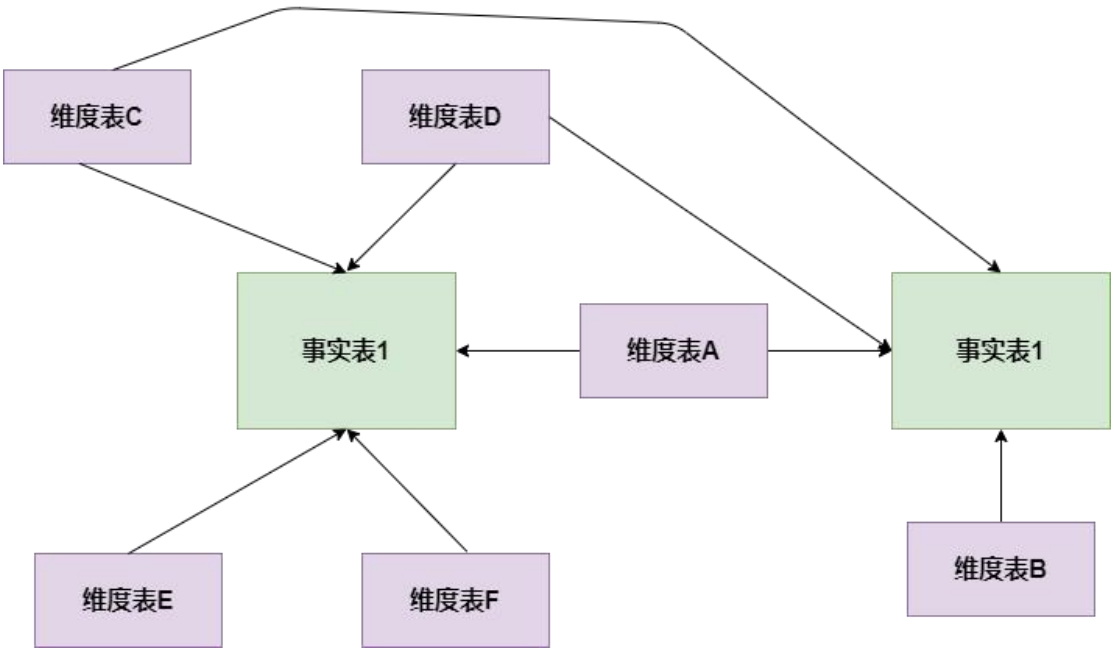
2. 雪花模型



雪花模式

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，性能比星型模型要低。

3. 星座模型



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。

数仓建模详细介绍可查看：**通俗易懂数仓建模**

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。

数据仓库详细介绍可查看：**万字详解整个数据仓库建设体系**

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：

1. 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。
2. 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的。这将需要使用第三方的 SerDe。

详细介绍可查看：**Hive 解析 Json 数组超全讲解**

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. 数据倾斜怎么解决

数据倾斜问题主要有以下几种：

1. 空值引发的数据倾斜
2. 不同数据类型引发的数据倾斜
3. 不可拆分大文件引发的数据倾斜
4. 数据膨胀引发的数据倾斜
5. 表连接时引发的数据倾斜
6. 确实无法减少数据量引发的数据倾斜

以上倾斜问题的具体解决方案可查看：Hive 千亿级数据倾斜解决方案

注意：对于 left join 或者 right join 来说，不会对关联的字段自动去除 null 值，对于 inner join 来说，会对关联的字段自动去除 null 值。

小伙伴们在阅读时注意下，在上面的文章（Hive 千亿级数据倾斜解决方案）中，有一处 sql 出现了上述问题（举例的时候原本是想使用 left join 的，结果手误写成了 join）。此问题由公众号读者发现，感谢这位读者指正。

10. Hive 小文件过多怎么解决

1. 使用 hive 自带的 concatenate 命令，自动合并小文件

使用方法：

```
#对于非分区表
alter table A concatenate;

#对于分区表
alter table B partition(day=20201224) concatenate;
```

注意：

1、concatenate 命令只支持 RCFILE 和 ORC 文件类型。

2、使用 concatenate 命令合并小文件时不能指定合并后的文件数量，但可以多次执行该命令。
3、当多次使用 concatenate 后文件数量不在变化，这个跟参数 mapreduce.input.fileinputformat.split.minsize=256mb 的设置有关，可设定每个文件的最小 size。

2. 调整参数减少 Map 数量

设置 map 输入合并小文件的相关参数（执行 Map 前进行小文件合并）：
在 mapper 中将多个文件合成一个 split 作为输入（CombineHiveInputFormat 底层是 Hadoop 的 CombineFileInputFormat 方法）：

```
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat; -- 默认
```

每个 Map 最大输入大小（这个值决定了合并后文件的数量）：

```
set mapred.max.split.size=256000000;    -- 256M
```

一个节点上 split 的至少大小（这个值决定了多个 DataNode 上的文件是否需要合并）：

```
set mapred.min.split.size.per.node=100000000;  -- 100M
```

一个交换机下 split 的至少大小(这个值决定了多个交换机上的文件是否需要合并)：

```
set mapred.min.split.size.per.rack=100000000;  -- 100M
```

3. 减少 Reduce 的数量

reduce 的个数决定了输出的文件的个数,所以可以调整 reduce 的个数控制 hive 表的文件数量。
hive 中的分区函数 distribute by 正好是控制 MR 中 partition 分区的，可以通过设置 reduce 的数量，结合分区函数让数据均衡的进入每个 reduce 即可：

```
#设置 reduce 的数量有两种方式，第一种是直接设置 reduce 个数
set mapreduce.job.reduces=10;

#第二种是设置每个 reduce 的大小，Hive 会根据数据总大小猜测确定一个 reduce 个数
set hive.exec.reducers.bytes.per.reducer=5120000000; -- 默认是 1G，设置为 5G

#执行以下语句，将数据均衡的分配到 reduce 中
set mapreduce.job.reduces=10;
insert overwrite table A partition(dt)
```

```
select * from B
distribute by rand();
```

对于上述语句解释：如设置 reduce 数量为 10，使用 rand()， 随机生成一个数 x % 10 ， 这样数据就会随机进入 reduce 中，防止出现有的文件过大或过小。

4. 使用 hadoop 的 archive 将小文件归档

Hadoop Archive 简称 HAR，是一个高效地将小文件放入 HDFS 块中的文件存档工具，它能够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，仍然允许对文件进行透明的访问。

```
#用来控制归档是否可用
set hive.archive.enabled=true;
#通知Hive 在创建归档时是否可以设置父目录
set hive.archive.har.parentdir.settable=true;
#控制需要归档文件的大小
set har.partfile.size=1099511627776;
```

使用以下命令进行归档：

```
ALTER TABLE A ARCHIVE PARTITION(dt='2021-05-07', hr='12');
```

对已归档的分区恢复为原文件：

```
ALTER TABLE A UNARCHIVE PARTITION(dt='2021-05-07', hr='12');
```

注意：

归档的分区可以查看不能 insert overwrite，必须先 unarchive

Hive 小文件问题具体可查看：**解决 hive 小文件过多问题**

11. Hive 优化有哪些

1. 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；

2. 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。

3. 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。

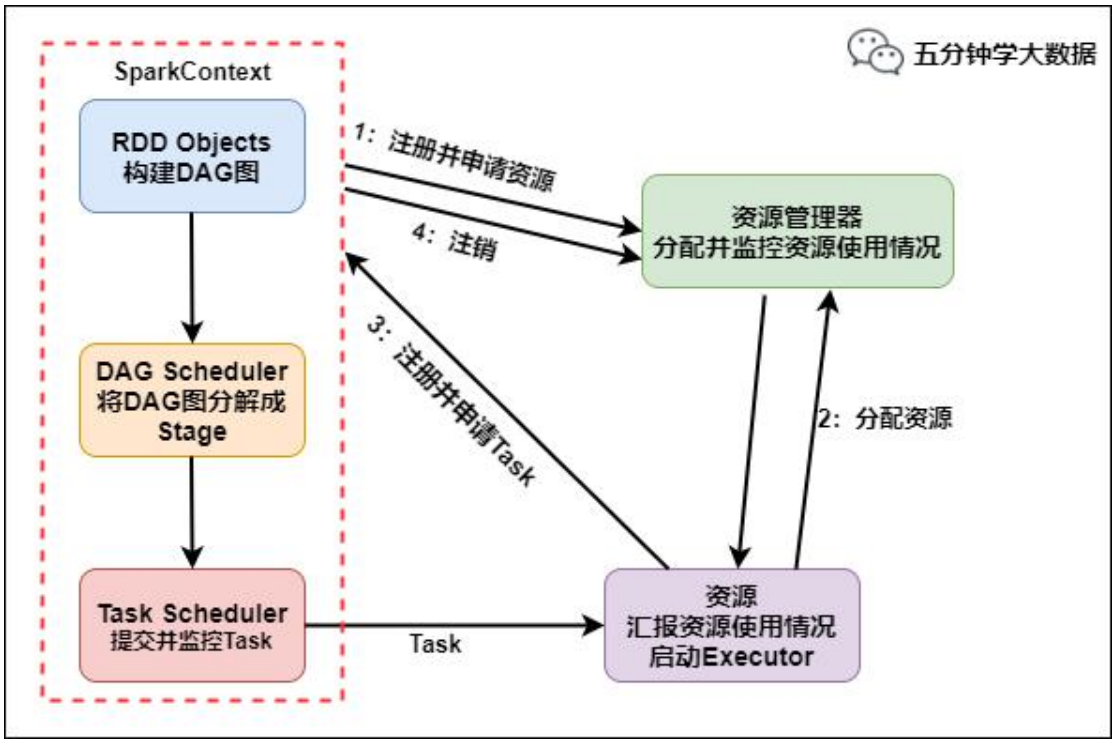
4. SQL 优化

- 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。

Hive 优化详细剖析可查看：Hive 企业级性能优化

Spark

1. Spark 的运行流程？



Spark 运行流程

具体运行流程如下：

1. SparkContext 向资源管理器注册并向资源管理器申请运行 Executor
2. 资源管理器分配 Executor，然后资源管理器启动 Executor
3. Executor 发送心跳至资源管理器
4. SparkContext 构建 DAG 有向无环图
5. 将 DAG 分解成 Stage（TaskSet）
6. 把 Stage 发送给 TaskScheduler
7. Executor 向 SparkContext 申请 Task
8. TaskScheduler 将 Task 发送给 Executor 运行
9. 同时 SparkContext 将应用程序代码发放给 Executor
10. Task 在 Executor 上运行，运行完毕释放所有资源

2. Spark 有哪些组件？

1. master：管理集群和节点，不参与计算。
2. worker：计算节点，进程本身不参与计算，和 master 汇报。
3. Driver：运行程序的 main 方法，创建 spark context 对象。

4. spark context：控制整个 application 的生命周期，包括 dagsheduler 和 task scheduler 等组件。
5. client：用户提交程序的入口。

3. Spark 中的 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。

4. 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。

5. 介绍一下 cogroup rdd 实现原理，你在什么场景下用过这个 rdd？

cogroup：对多个（2~4）RDD 中的 KV 元素，每个 RDD 中相同 key 中的元素分别聚合成一个集合。
与 reduceByKey 不同的是:reduceByKey 针对一个 RDD 中相同的 key 进行合并。而 cogroup 针对多个 RDD 中相同的 key 的元素进行合并。
cogroup 的函数实现：这个实现根据要进行合并的两个 RDD 操作，生成一个 CoGroupedRDD 的实例，这个 RDD 的返回结果是把相同的 key 中两个 RDD 分别进行合并操作，最后返回的 RDD 的 value 是一个 Pair 的实例，这个实例包含两个 Iterable 的值，第一个值表示的是 RDD1 中相同 KEY 的值，第二个值表示的是 RDD2 中相同 key 的值。
由于做 cogroup 的操作，需要通过 partitioner 进行重新分区的操作，因此，执行这个流程时，需要执行一次 shuffle 的操作(如果要进行合并的两个 RDD 的都已经是 shuffle 后的 rdd，同时他们对应的 partitioner 相同时，就不需要执行 shuffle)。
场景：表关联查询或者处理重复的 key。

6. 如何区分 RDD 的宽窄依赖？

窄依赖:父 RDD 的一个分区只会被子 RDD 的一个分区依赖；
宽依赖:父 RDD 的一个分区会被子 RDD 的多个分区依赖(涉及到 shuffle)。

7. 为什么要设计宽窄依赖？

1. 对于窄依赖：
   窄依赖的多个分区可以并行计算；
   窄依赖的一个分区的数据如果丢失只需要重新计算对应的分区的数据就可以了。
2. 对于宽依赖：
   划分 Stage(阶段)的依据:对于宽依赖,必须等到上一阶段计算完成才能计算下一阶段。

8. DAG 是什么？

DAG(Directed Acyclic Graph 有向无环图)指的是数据转换执行的过程,有方向,无闭环(其实就是 RDD 执行的流程)；
原始的 RDD 通过一系列的转换操作就形成了 DAG 有向无环图，任务执行时，可以按照 DAG 的描述，执行真正的计算(数据被操作的一个过程)。

9. DAG 中为什么要划分 Stage？

并行计算。
一个复杂的业务逻辑如果有 shuffle，那么就意味着前面阶段产生结果后，才能执行下一个阶段，即下一个阶段的计算要依赖上一个阶段的数据。那么我们按照 shuffle 进行划分(也就是按照宽依赖就行划分)，就可以将一个 DAG 划分成多个 Stage/阶段，在同一个 Stage 中，会有多个算子操作，可以形成一个 pipeline 流水线，流水线内的多个平行的分区可以并行执行。

10. 如何划分 DAG 的 stage？

对于窄依赖，partition 的转换处理在 stage 中完成计算，不划分(将窄依赖尽量放在在同一个 stage 中，可以实现流水线计算)。
对于宽依赖，由于有 shuffle 的存在，只能在父 RDD 处理完成后，才能开始接下来的计算，也就是说需要要划分 stage。

11. DAG 划分为 Stage 的算法了解吗？

核心算法：回溯算法

从后往前回溯/反向解析，遇到窄依赖加入本 Stage，遇见宽依赖进行 Stage 切分。

Spark 内核会从触发 Action 操作的那个 RDD 开始**从后往前推**，首先会为最后一个 RDD 创建一个 Stage，然后继续倒推，如果发现对某个 RDD 是宽依赖，那么就会将宽依赖的那个 RDD 创建一个新的 Stage，那个 RDD 就是新的 Stage 的最后一个 RDD。 然后依次类推，继续倒推，根据窄依赖或者宽依赖进行 Stage 的划分，直到所有的 RDD 全部遍历完成为止。

具体划分算法请参考：AMP 实验室发表的论文

《Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing》

http://xueshu.baidu.com/usercenter/paper/show?paperid=b33564e60f0a7e7a1889a9da10963461&site=xueshu_se

12. 对于 Spark 中的数据倾斜问题你有什么好的方案？

1. 前提是定位数据倾斜，是 OOM 了，还是任务执行缓慢，看日志，看 WebUI
2. 解决方法，有多个方面：

- 避免不必要的 shuffle，如使用广播小表的方式，将 reduce-side-join 提升为 map-side-join
- 分拆发生数据倾斜的记录，分成几个部分进行，然后合并 join 后的结果
- 改变并行度，可能并行度太少了，导致个别 task 数据压力大
- 两阶段聚合，先局部聚合，再全局聚合
- 自定义 paritioner，分散 key 的分布，使其更加均匀

13. Spark 中的 OOM 问题？

1. map 类型的算子执行中内存溢出如 flatMap，mapPatitions

- 原因：map 端过程产生大量对象导致内存溢出：这种溢出的原因是在单个 map 中产生了大量的对象导致的针对这种问题。

2. 解决方案：

- 增加堆内内存。
- 在不增加内存的情况下，可以减少每个 Task 处理数据量，使每个 Task 产生大量的对象时，Executor 的内存也能够装得下。具体做法可以在会

产生大量对象的 map 操作之前调用 repartition 方法，分区成更小的块传入 map。

2. shuffle 后内存溢出如 join，reduceByKey，repartition。

- shuffle 内存溢出的情况可以说都是 shuffle 后，单个文件过大导致的。在 shuffle 的使用，需要传入一个 partitioner，大部分 Spark 中的 shuffle 操作，默认的 partitioner 都是 HashPatitioner，默认值是父 RDD 中最大的分区数. 这个参数 spark.default.parallelism 只对 HashPartitioner 有效.如果是别的 partitioner 导致的 shuffle 内存溢出就需要重写 partitioner 代码了.

3. driver 内存溢出

- 用户在 Dirver 端口生成大对象，比如创建了一个大的集合数据结构。解决方案：将大对象转换成 Executor 端加载，比如调用 sc.textfile 或者评估大对象占用的内存，增加 dirver 端的内存
- 从 Executor 端收集数据（collect）回 Dirver 端，建议将 driver 端对 collect 回来的数据所作的操作，转换成 executor 端 rdd 操作。

14. Spark 中数据的位置是被谁管理的？

每个数据分片都对应具体物理位置，数据的位置是被 blockManager 管理，无论数据是在磁盘，内存还是 tacyan，都是由 blockManager 管理。

15. Spaek 程序执行，有时候默认为什么会产生很多 task，怎么修改默认 task 执行个数？

1. 输入数据有很多 task,尤其是有很多小文件的时候,有多少个输入 block 就会有多少个 task 启动;
2. spark 中有 partition 的概念，每个 partition 都会对应一个 task，task 越多，在处理大规模数据的时候，就会越有效率。不过 task 并不是越多越好,如果平时测试,或者数据量没有那么大,则没有必要 task 数量太多。
3. 参数可以通过 spark_home/conf/spark-default.conf 配置文件设置:

针对 spark sql 的 task 数量：spark.sql.shuffle.partitions=50

非 spark sql 程序设置生效：spark.default.parallelism=10

16. 介绍一下 join 操作优化经验？

这道题常考，这里只是给大家一个思路，简单说下！面试之前还需做更多准备。

join 其实常见的就分为两类： map-side join 和 reduce-side join。

当大表和小表 join 时，用 map-side join 能显著提高效率。

将多份数据进行关联是数据处理过程中非常普遍的用法，不过在分布式计算系统中，这个问题往往会变的非常麻烦，因为框架提供的 join 操作一般会将所有数据根据 key 发送到所有的 reduce 分区中去，也就是 shuffle 的过程。造成大量的网络以及磁盘 IO 消耗，运行效率极其低下，这个过程一般被称为 reduce-side-join。

如果其中有张表较小的话，我们则可以自己实现在 map 端实现数据关联，跳过大量数据进行 shuffle 的过程，运行时间得到大量缩短，根据不同数据可能会有几倍到数十倍的性能提升。

在大数据量的情况下，join 是一中非常昂贵的操作，需要在 join 之前应尽可能的先缩小数据量。

对于缩小数据量，有以下几条建议：

1. 若两个 RDD 都有重复的 key，join 操作会使得数据量会急剧的扩大。所有，最好先使用 distinct 或者 combineByKey 操作来减少 key 空间或者用 cogroup 来处理重复的 key，而不是产生所有的交叉结果。在 combine 时，进行机智的分区，可以避免第二次 shuffle。
2. 如果只在一个 RDD 出现，那你将在无意中丢失你的数据。所以使用外连接会更加安全，这样你就能确保左边的 RDD 或者右边的 RDD 的数据完整性，在 join 之后再过滤数据。
3. 如果我们容易得到 RDD 的可以的有用的子集合，那么我们可以先用 filter 或者 reduce，如何在再用 join。

17. Spark 与 MapReduce 的 Shuffle 的区别？

1. 相同点：都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）

2. 不同点：

- MapReduce 默认是排序的，spark 默认不排序，除非使用 sortByKey 算子。
- MapReduce 可以划分成 split，map()、spill、merge、shuffle、sort、reduce()等阶段，spark 没有明显的阶段划分，只有不同的 stage 和算子操作。
- MR 落盘，Spark 不落盘，spark 可以解决 mr 落盘导致效率低下的问题。

18. Spark SQL 执行的流程？

这个问题如果深挖还挺复杂的，这里简单介绍下总体流程：

1. parser：基于 antlr 框架对 sql 解析，生成抽象语法树。
2. 变量替换：通过正则表达式找出符合规则的字符串，替换成系统缓存环境的变量

SQLConf 中的 spark.sql.variable.substitute，默认是可用的；参考 SparkSqlParser

3. parser：将 antlr 的 tree 转成 spark catalyst 的 LogicPlan，也就是 未解析的逻辑计划；详细参考 AstBuild，ParseDriver
4. analyzer：通过分析器，结合 catalog，把 logical plan 和实际的数据绑定起来，将 未解析的逻辑计划 生成 逻辑计划；详细参考 QureyExecution
5. 缓存替换：通过 CacheManager，替换有相同结果的 logical plan（逻辑计划）
6. logical plan 优化，基于规则的优化；优化规则参考 Optimizer，优化执行器 RuleExecutor
7. 生成 spark plan，也就是物理计划；参考 QueryPlanner 和 SparkStrategies
8. spark plan 准备阶段
9. 构造 RDD 执行，涉及 spark 的 wholeStageCodegenExec 机制，基于 janino 框架生成 java 代码并编译

19. Spark SQL 是如何将数据写到 Hive 表的？

- 方式一：是利用 Spark RDD 的 API 将数据写入 hdfs 形成 hdfs 文件，之后再将 hdfs 文件和 hive 表做加载映射。
- 方式二：利用 Spark SQL 将获取的数据 RDD 转换成 DataFrame，再将 DataFrame 写成缓存表，最后利用 Spark SQL 直接插入 hive 表中。而对于利用 Spark SQL 写 hive 表官方有两种常见的 API，第一种是利用 JavaBean 做映射，第二种是利用 StructType 创建 Schema 做映射。

20. 通常来说，Spark 与 MapReduce 相比，Spark 运行效率更高。请说明效率更高来源于 Spark 内置的哪些机制？

1. 基于内存计算，减少低效的磁盘交互；
2. 高效的调度算法，基于 DAG；
3. 容错机制 Linage。

重点部分就是 DAG 和 Lingae

21. 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，在没有进行调优的情况下，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。

22. 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。

1. 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。
2. 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。

23. Spark 如何保证宕机迅速恢复？

1. 适当增加 spark standby master
2. 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

24. 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。

25. 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：

1. 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。
2. 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。

26. 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

27. 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。

28. 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。

29. Spark Streaming 整合 Kafka 的两种模式？

1. receiver 方式：将数据拉取到 executor 中做操作，若数据量大，内存存储不下，可以通过 WAL，设置了本地存储，保证数据不丢失，然后使用 Kafka 高级 API 通过 zk 来维护偏移量，保证消费数据。receiver 消费的数据偏移量是在 zk 获取的，此方式效率低，容易出现数据丢失。

- receiver 方式的容错性：在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用 Spark Streaming 的预写日志机制（Write Ahead Log，WAL）。该机制会同步地将接收到的 Kafka 数据写入分布式文件系统（比如 HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

- Kafka 中的 topic 的 partition，与 Spark 中的 RDD 的 partition 是没有关系的。在 1、KafkaUtils.createStream()中，提高 partition 的数量，只会增加 Receiver 方式中读取 partition 的线程的数量。不会增加 Spark 处理数据的并行度。 可以创建多个 Kafka 输入 DStream，使用不同的 consumer group 和 topic，来通过多个 receiver 并行接收数据。

2. 基于 Direct 方式：使用 Kafka 底层 Api，其消费者直接连接 kafka 的分区上，因为 createDirectStream 创建的 DirectKafkaInputDStream 每个 batch 所对应的 RDD 的分区与 kafka 分区一一对应，但是需要自己维护偏移量，即用即取，不会给内存造成太大的压力，效率高。

- 优点：简化并行读取：如果要读取多个 partition，不需要创建多个输入 DStream 然后对它们进行 union 操作。Spark 会创建跟 Kafka partition 一样多的 RDD partition，并且会并行从 Kafka 中读取数据。所以在 Kafka partition 和 RDD partition 之间，有一个一对一的映射关系。
- 高性能：如果要保证零数据丢失，在基于 receiver 的方式中，需要开启 WAL 机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka 自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到 WAL 中。而基于 direct 的方式，不依赖 Receiver，不需要开启 WAL 机制，只要 Kafka 中作了数据的复制，那么就可以通过 Kafka 的副本进行恢复。

3. receiver 与和 direct 的比较：

- 基于 receiver 的方式，是使用 Kafka 的高阶 API 来在 ZooKeeper 中保存消费过的 offset 的。这是消费 Kafka 数据的传统方式。这种方式配合着 WAL 机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为 Spark 和 ZooKeeper 之间可能是不同步的。
- 基于 direct 的方式，使用 Kafka 的低阶 API，Spark Streaming 自己就负责追踪消费的 offset，并保存在 checkpoint 中。Spark 自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

- Receiver 方式是通过 zookeeper 来连接 kafka 队列，Direct 方式是直接连接到 kafka 的节点上获取数据。

30. 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。

31. Spark 解决了 Hadoop 的哪些问题？

1. 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。
2. 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；
3. 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；
4. 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；
5. 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。

6. 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。

32. 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；

33. 你用 Spark Sql 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。

34. 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。

35. 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。

36. 如何实现 Spark Streaming 读取 Flume 中的数据？

可以这样说：

- 前期经过技术调研，查看官网相关资料，发现 sparkStreaming 整合 flume 有 2 种模式，一种是拉模式，一种是推模式，然后在简单的聊聊这 2 种模式的特点，以及如何部署实现，需要做哪些事情，最后对比两种模式的特点，选择那种模式更好。
- 推模式：Flume 将数据 Push 推给 Spark Streaming
- 拉模式：Spark Streaming 从 flume 中 Poll 拉取数据

37. 在实际开发的时候是如何保证数据不丢失的？

可以这样说：

- flume 那边采用的 channel 是将数据落地到磁盘中，保证数据源端安全性（可以在补充一下，flume 在这里的 channel 可以设置为 memory 内存中，提高数据接收处理的效率，但是由于数据在内存中，安全机制保证不了，故选择 channel 为磁盘存储。整个流程运行有一点的延迟性）
- sparkStreaming 通过拉模式整合的时候，使用了 FlumeUtils 这样一个类，该类是需要依赖一个额外的 jar 包（spark-streaming-flume_2.10）
- 要想保证数据不丢失，数据的准确性，可以在构建 StreamingConext 的时候，利用 StreamingContext.getOrCreate（checkpoint，creatingFunc: () => StreamingContext）来创建一个 StreamingContext，使用 StreamingContext.getOrCreate 来创建 StreamingContext 对象，传入的第一个参数是 checkpoint 的存放目录，第二参数是生成 StreamingContext 对象的用户自定义函数。如果 checkpoint 的存放目录存在，则从这个目录中生成 StreamingContext 对象；如果不存在，才会调用第二个函数来生成新的 StreamingContext 对象。在 creatingFunc 函数中，除了生成一个新的 StreamingContext 操作，还需要完成各种操作，然后调用 ssc.checkpoint(checkpointDirectory)来初始化 checkpoint 功能，最后再返回 StreamingContext 对象。
  这样，在 StreamingContext.getOrCreate 之后，就可以直接调用 start()函数来启动（或者是从中断点继续运行）流式应用了。如果有其他在启动或继续运行都要做的工作，可以在 start()调用前执行。

38. RDD 有哪些缺陷？

1. **不支持细粒度的写和更新操作**，Spark 写数据是粗粒度的，所谓粗粒度，就是批量写入数据，目的是为了提高效率。但是 Spark 读数据是细粒度的，也就是说可以一条条的读。
2. **不支持增量迭代计算**，如果对 Flink 熟悉，可以说下 Flink 支持增量迭代计算。

1. 为什么要使用 kafka？

1. 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。
2. 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。
3. 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。
4. 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。
5. 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：

1. 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。
2. 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。
3. 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：

1. 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。
2. 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。

1. 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。

2. 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。

3. 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？

1. kafka 是将数据写到磁盘的，一般数据不会丢失。
2. 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？

1. 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。

2. 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:

1. 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。
2. 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：

1. 可以简化代码的实现逻辑，减少出错的可能;
2. 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;
3. 没有延时的影响;
4. 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

1. 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
2. 连接到对应 Leader 对应的 broker
3. consumer 将自自己己保存的 offset 发送给 Leader
4. Leader 根据 offset 等信息定位到 segment（索引文文件和日日志文文件）
5. 根据索引文文件中的内容，定位到日日志文文件中该偏移量量对应的开始位置读取相应长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一个 partition。

12. Kafka 消息数据积压，Kafka 消费能力不足怎么处理？

1. 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）
2. 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
```

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：

1. 一次性写入，多次读取。
2. 保证数据的一致性。
3. 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：

1. 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。
2. 数据需要长久保存，且量会持久增长到比较大的场景。
3. HBase 不适用与有 join，多级索引，表关系复杂的数据模型。

4. 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。
5. 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：

1. 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。
2. 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
3. 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
4. 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]
5. 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：

1. 合并文件
2. 清除过期，多余版本的数据
3. 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：
4. 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。

5. 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. 简单介绍一下 Flink

Flink 是一个面向流处理和批处理的分布式数据计算引擎，能够基于同一个 Flink 运行，可以提供流处理和批处理两种类型的功能。 在 Flink 的世界观中，一切都是由流组成的，离线数据是有界的流；实时数据是一个没有界限的流：这就是所谓的有界流和无界流。

2. 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。

3. Flink 集群运行时角色

Flink 运行时由两种类型的进程组成：一个 JobManager 和一个或者多个 TaskManager。

Client 不是运行时和程序执行的一部分，而是用于准备数据流并将其发送给 JobManager。之后，客户端可以断开连接（分离模式），或保持连接来接收进程报告（附加模式）。客户端可以作为触发执行 Java/Scala 程序的一部分运行，也可以在命令行进程 `./bin/flink run ...` 中运行。

可以通过多种方式启动 JobManager 和 TaskManager：直接在机器上作为 standalone 集群启动、在容器中启动、或者通过 YARN 等资源框架管理并启动。TaskManager 连接到 JobManagers，宣布自己可用，并被分配工作。

JobManager：

JobManager 具有许多与协调 Flink 应用程序的分布式执行有关的职责：它决定何时调度下一个 task（或一组 task）、对完成的 task 或执行失败做出反应、协调 checkpoint、并且协调从失败中恢复等等。这个进程由三个不同的组件组成：

- ResourceManager

ResourceManager 负责 Flink 集群中的资源提供、回收、分配，管理 task slots。

- Dispatcher

Dispatcher 提供了一个 REST 接口，用来提交 Flink 应用程序执行，并为每个提交的作业启动一个新的 JobMaster。它还运行 Flink WebUI 用来提供作业执行信息。

- JobMaster

JobMaster 负责管理单个 JobGraph 的执行。Flink 集群中可以同时运行多个作业，每个作业都有自己的 JobMaster。

TaskManagers：

TaskManager（也称为 worker）执行作业流的 task，并且缓存和交换数据流。

必须始终至少有一个 TaskManager。在 TaskManager 中资源调度的最小单位是 task slot。TaskManager 中 task slot 的数量表示并发处理 task 的数量。请注意一个 task slot 中可以执行多个算子。

4. Flink 相比 Spark Streaming 有什么区别

1. 架构模型

Spark Streaming 在运行时的主要角色包括：Master、Worker、Driver、Executor，Flink 在运行时主要包含：Jobmanager、Taskmanager 和 Slot。

2. 任务调度

Spark Streaming 连续不断的生成微小的数据批次，构建有向无环图 DAG，Spark Streaming 会依次创建 DStreamGraph、JobGenerator、JobScheduler。

Flink 根据用户提交的代码生成 StreamGraph，经过优化生成 JobGraph，然后提交给 JobManager 进行处理，JobManager 会根据 JobGraph 生成 ExecutionGraph，ExecutionGraph 是 Flink 调度最核心的数据结构，JobManager 根据 ExecutionGraph 对 Job 进行调度。

3. 时间机制

Spark Streaming 支持的时间机制有限，只支持处理时间。Flink 支持了流处理程序在时间上的三个定义：处理时间、事件时间、注入时间。同时也支持 watermark 机制来处理滞后数据。

4. 容错机制

对于 Spark Streaming 任务，我们可以设置 checkpoint，然后假如发生故障并重启，我们可以从上次 checkpoint 之处恢复，但是这个行为只能使得数据不丢失，可能会重复处理，不能做到恰一次处理语义。

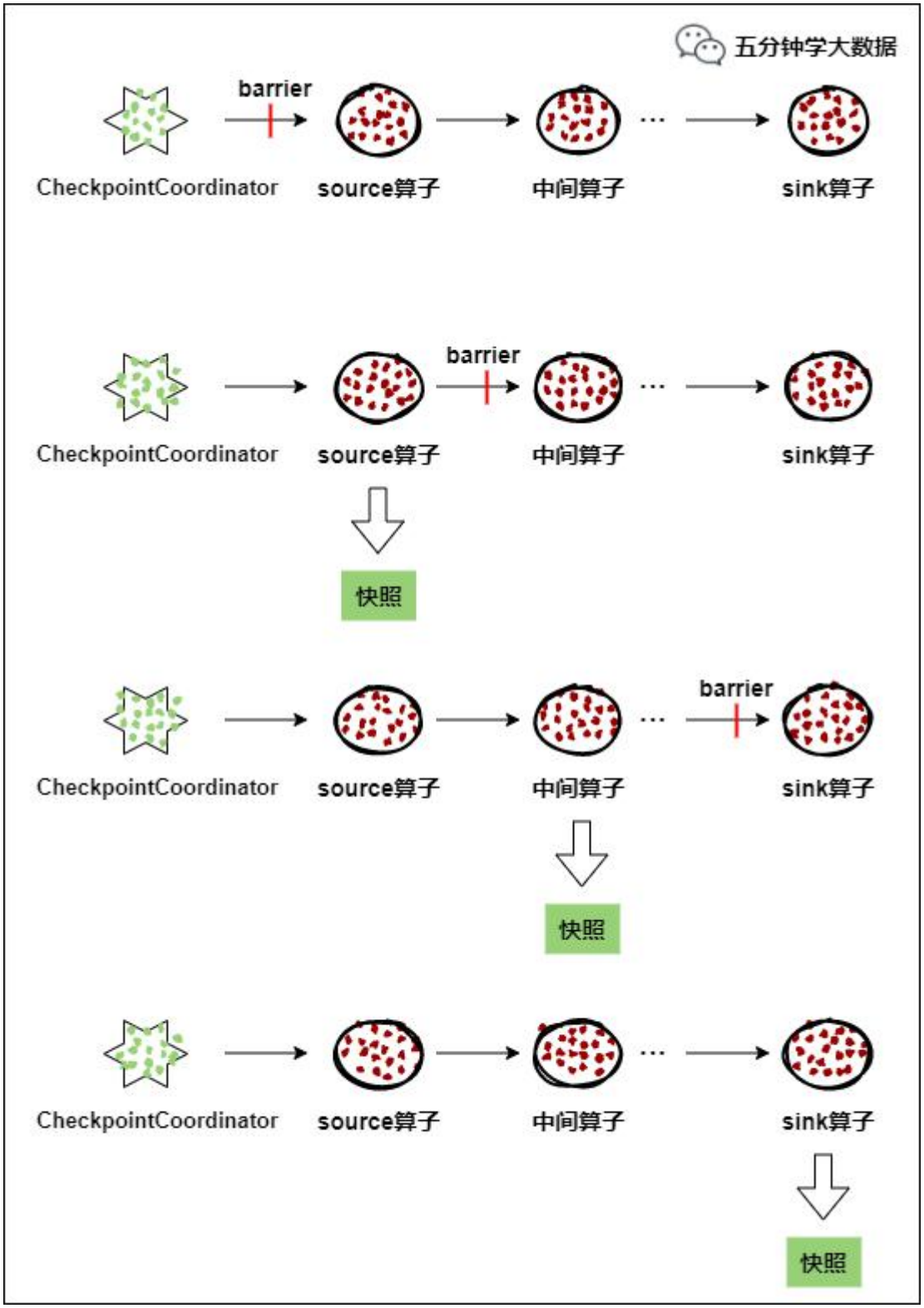
Flink 则使用两阶段提交协议来解决这个问题。

5. 介绍下 Flink 的容错机制（checkpoint）

Checkpoint 机制是 Flink 可靠性的基石，可以保证 Flink 集群在某个算子因为某些原因(如 异常退出)出现故障时，能够将整个应用流图的状态恢复到故障之前的某一状态，保证应用流图状态的一致性。Flink 的 Checkpoint 机制原理来自“Chandy-Lamport algorithm”算法。

每个需要 Checkpoint 的应用在启动时，Flink 的 JobManager 为其创建一个 CheckpointCoordinator(检查点协调器)，CheckpointCoordinator 全权负责本应用的快照制作。

CheckpointCoordinator(检查点协调器)，CheckpointCoordinator 全权负责本应用的快照制作。



1. CheckpointCoordinator(检查点协调器) 周期性的向该流应用的所有 source 算子发送 barrier(屏障)。
2. 当某个 source 算子收到一个 barrier 时，便暂停数据处理过程，然后将自己的当前状态制作成快照，并保存到指定的持久化存储中，最后向 CheckpointCoordinator 报告自己快照制作情况，同时向自身所有下游算子广播该 barrier，恢复数据处理
3. 下游算子收到 barrier 之后，会暂停自己的数据处理过程，然后将自身的相关状态制作成快照，并保存到指定的持久化存储中，最后向 CheckpointCoordinator 报告自身快照情况，同时向自身所有下游算子广播该 barrier，恢复数据处理。
4. 每个算子按照步骤 3 不断制作快照并向下游广播，直到最后 barrier 传递到 sink 算子，快照制作完成。
5. 当 CheckpointCoordinator 收到所有算子的报告之后，认为该周期的快照制作成功；否则，如果在规定的时间内没有收到所有算子的报告，则认为本周期快照制作失败。

文章推荐：

Flink 可靠性的基石-checkpoint 机制详细解析

6. Flink checkpoint 与 Spark Streaming 的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。

7. 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。

两阶段提交协议详解：八张图搞懂 Flink 的 Exactly-once

8. 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。

9. Flink 常用的算子有哪些

分两部分：

1. 数据读取，这是 Flink 流计算应用的起点，常用算子有：

- 从内存读：fromElements
- 从文件读：readTextFile
- Socket 接入 ：socketTextStream
- 自定义读取：createInput

2. 处理数据的算子，常用的算子包括：Map（单输入单输出）、FlatMap（单输入、多输出）、Filter（过滤）、KeyBy（分组）、Reduce（聚合）、Window（窗口）、Connect（连接）、Split（分割）等。

推荐阅读：一文学完 Flink 流计算常用算子（Flink 算子大全）

10. Flink 任务延时高，如何入手

在 Flink 的后台任务管理中，我们可以看到 Flink 的哪个算子和 task 出现了反压。最主要的手段是资源调优和算子调优。资源调优即是对作业中的 Operator 的并发数（parallelism）、CPU（core）、堆内存（heap_memory）等参数进行调优。作业参数调优包括：并行度的设置，State 的设置，checkpoint 的设置。

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. 如何排查生产环境中的反压问题

1. 反压出现的场景

反压经常出现在促销、热门活动等场景。短时间内流量陡增造成数据的堆积或者消费速度变慢。

它们有一个共同的特点：数据的消费速度小于数据的生产速度。

2. 反压监控方法

通过 Flink Web UI 发现反压问题。

Flink 的 TaskManager 会每隔 50 ms 触发一次反压状态监测，共监测 100 次，并将计算结果反馈给 JobManager，最后由 JobManager 进行计算反压的比例，然后进行展示。

这个比例展示逻辑如下：

OK: 0 <= Ratio <= 0.10，表示状态良好正；

LOW: 0.10 < Ratio <= 0.5，表示有待观察；

HIGH: 0.5 < Ratio <= 1，表示要处理了（增加并行度/subTask/检查是否有数据倾斜/增加内存）。

0.01，代表 100 次中有一次阻塞在内部调用。

3. flink 反压的实现方式

Flink 任务的组成由基本的“流”和“算子”构成，“流”中的数据在“算子”间进行计算和转换时，会被放入分布式的阻塞队列中。当消费者的阻塞队列满时，则会降低生产者的数据生产速度

4. 反压问题定位和处理

Flink 会因为数据堆积和处理速度变慢导致 checkpoint 超时，而 checkpoint 是 Flink 保证数据一致性的关键所在，最终会导致数据的不一致发生。

数据倾斜：可以在 Flink 的后台管理页面看到每个 Task 处理数据的大小。当数据倾斜出现时，通常是简单地使用类似 KeyBy 等分组聚合函数导致的，需要用户将热点 Key 进行预处理，降低或者消除热点 Key 的影。

GC：不合理的设置 TaskManager 的垃圾回收参数会导致严重的 GC 问题，我们可以通过 `-XX:+PrintGCDetails` 参数查看 GC 的日志。

代码本身：开发者错误地使用 Flink 算子，没有深入了解算子的实现机制导致性能问题。我们可以通过查看运行机器节点的 CPU 和内存情况定位问题。

13. 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。

14. Operator Chains（算子链）这个概念你了解吗

为了更高效地分布式执行，Flink 会尽可能地将 operator 的 subtask 链接（chain）在一起形成 task。每个 task 在一个线程中执行。将 operators 链接成 task 是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量。这就是我们所说的算子链。

15. 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。

16. 如何处理生产环境中的数据倾斜问题

1. flink 数据倾斜的表现：

任务节点频繁出现反压，增加并行度也不能解决问题；

部分节点出现 OOM 异常，是因为大量的数据集中在某个节点上，导致该节点内存被爆，任务失败重启。

2. 数据倾斜产生的原因：

业务上有严重的数据热点，比如滴滴打车的订单数据中北京、上海等几个城市的订单量远远超过其他地区；

技术上大量使用了 KeyBy、GroupBy 等操作，错误的使用了分组 Key，人为产生数据热点。

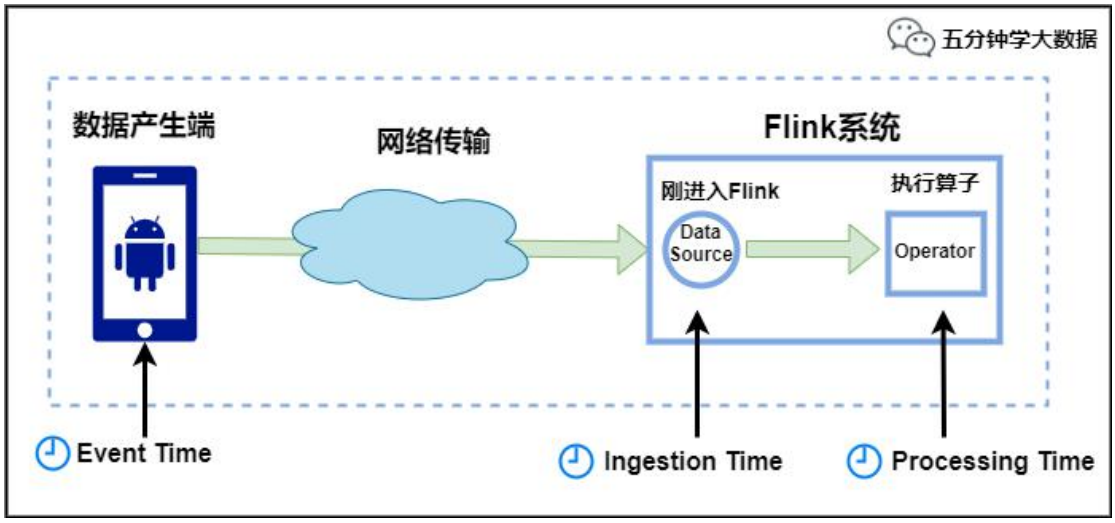
3. 解决问题的思路：

业务上要尽量避免热点 key 的设计，例如我们可以把北京、上海等热点城市分成不同的区域，并进行单独处理；

技术上出现热点时，要调整方案打散原来的 key，避免直接聚合；此外 Flink 还提供了大量的功能可以避免数据倾斜。

17. Flink 中的 Time 有哪几种

Flink 中的时间有三种类型，如下图所示：



- Event Time：是事件创建的时间。它通常由事件中的时间戳描述，例如采集的日志数据中，每一条日志都会记录自己的生成时间，Flink 通过时间戳分配器访问事件时间戳。
- Ingestion Time：是数据进入 Flink 的时间。
- Processing Time：是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是 Processing Time。

例如，一条日志进入 Flink 的时间为 2021-01-22 10:00:00.123，到达 Window 的系统时间为 2021-01-22 10:00:01.234，日志的内容如下：

```
2021-01-06 18:37:15.624 INFO Fail over to rm2
```

对于业务来说，要统计 1min 内的故障日志个数，哪个时间是最有意义的？——eventTime，因为我们要根据日志的生成时间进行统计。

18. Flink 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

文章推荐：

Flink 中极其重要的 Time 与 Window 详细解析

19. Flink 中 window 出现数据倾斜怎么解决

window 产生数据倾斜指的是数据在不同的窗口内堆积的数据量相差过多。本质上产生这种情况的原因是数据源头发送的数据量速度不同导致的。出现这种情况一般通过两种方式来解决：

- 在数据进入窗口前做预聚合
- 重新设计窗口聚合的 key

20. 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。
推荐阅读：一文学会 Flink CEP

21. Flink 设置并行度的方式

们在实际生产环境中可以从四个不同层面设置并行度：

1. 操作算子层面(Operator Level)

```
.map(new RollingAdditionMapper()).setParallelism(10) //将操作算子设置并行度
```

2. 执行环境层面(Execution Environment Level)

```
$FLINK_HOME/bin/flink 的-p 参数修改并行度
```

3. 客户端层面(Client Level)

```
env.setParallelism(10)
```

4. 系统层面(System Level)

全局配置在 flink-conf.yaml 文件中，parallelism.default，默认是 1：可以设置默认值大一点
需要注意的优先级：算子层面>环境层面>客户端层面>系统层面。

22. Flink 中 Task 如何做到数据交换

在一个 Flink Job 中，数据需要在不同的 task 中进行交换，整个数据交换是有 TaskManager 负责的，TaskManager 的网络组件首先从缓冲 buffer 中收集 records，然后再发送。Records 并不是一个一个被发送的，是积累一个批次再发送，batch 技术可以更加高效的利用网络资源。

23. 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。

24. 介绍下 Flink 的序列化

Flink 摒弃了 Java 原生的序列化方法，以独特的方式处理数据类型和序列化，包含自己的类型描述符，泛型类型提取和类型序列化框架。
TypeInformation 是所有类型描述符的基类。它揭示了该类型的一些基本属性，并且可以生成序列化器。
TypeInformation 支持以下几种类型：

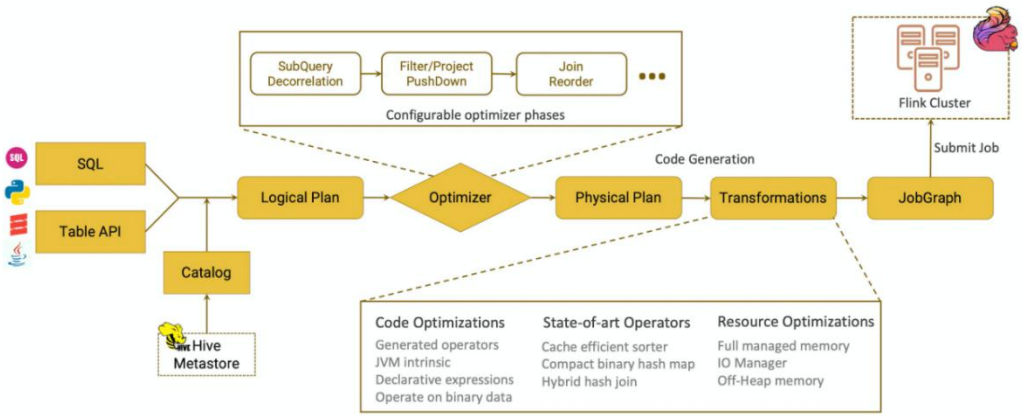
- BasicTypeInfo: 任意 Java 基本类型或 String 类型
- BasicArrayTypeInfo: 任意 Java 基本类型数组或 String 数组
- WritableTypeInfo: 任意 Hadoop Writable 接口的实现类
- TupleTypeInfo: 任意的 Flink Tuple 类型(支持 Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的 Java Tuple 实现
- CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)

- PojoTypeInfo: 任意的 POJO (Java or Scala)，例如，Java 对象的所有成员变量，要么是 public 修饰符定义，要么有 getter/setter 方法
- GenericTypeInfo: 任意无法匹配之前几种类型的类

25. Flink 海量数据高效去重

1. 基于状态后端。
2. 基于 HyperLogLog：不是精准的去重。
3. 基于布隆过滤器（BloomFilter）；快速判断一个 key 是否存在于某容器，不存在就直接返回。
4. 基于 BitMap；用一个 bit 位来标记某个元素对应的 Value，而 Key 即是该元素。由于采用了 Bit 为单位来存储数据，因此可以大大节省存储空间。
5. 基于外部数据库；选择使用 Redis 或者 HBase 存储数据，我们只需要设计好存储的 Key 即可，不需要关心 Flink 任务重启造成的状态丢失问题。

26. Flink SQL 的是如何实现的



构建抽象语法树的事情交给了 Calcite 去做。SQL query 会经过 Calcite 解析器转变成 SQL 节点树，通过验证后构建成 Calcite 的抽象语法树（也就是图中的 Logical Plan）。另一边，Table API 上的调用会构建成 Table API 的抽象语法树，并通过 Calcite 提供的 RelBuilder 转变成 Calcite 的抽象语法树。然后依次被转换成逻辑执行计划和物理执行计划。

在提交任务后会分发到各个 TaskManager 中运行，在运行时会使用 Janino 编译器编译代码后运行。

业务方面

1. ODS 层采用什么压缩方式和存储格式？

压缩采用 Snappy，存储采用 orc，压缩比是 100g 数据压缩完 10g 左右。

2. DWD 层做了哪些事？

1. 数据清洗

- 空值去除
- 过滤核心字段无意义的数据，比如订单表中订单 id 为 null，支付表中支付 id 为空
- 对手机号、身份证号等敏感数据脱敏
- 对业务数据传过来的表进行维度退化和降维。
- 将用户行为宽表和业务表进行数据一致性处理

2. 清洗的手段

- Sql、mr、rdd、kettle、Python（项目中采用 sql 进行清除）

3. DWS 层做了哪些事？

1. DWS 层有 3-5 张宽表（处理 100-200 个指标 70%以上的需求）

具体宽表名称：用户行为宽表，用户购买商品明细行为宽表，商品宽表，购物车宽表，物流宽表、登录注册、售后等。

2. 哪个宽表最宽？大概有多少个字段？ 最宽的是用户行为宽表。大概有60-100 个字段

1. 在处理大数据过程中，如何保证得到期望值

1. 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
2. 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
3. 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

1. SQL 结果：var 数据量 = select count（*）from 表 where 时间等过滤条件
2. 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
3. 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

4. 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
5. 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

1. 目标字段：选择要监控的字段，不能选“无”
2. SQL 结果：var 异常数据量 = select count(*) from 表 where 目标字段 is null
3. 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

1. 目标字段：第一步先正常统计条数；select count(*) form 表；
2. 第二步，去重统计；select count(*) from 表 group by 某个字段
3. 第一步的值和第二步的值做减法，看是否在上下线阀值之内
4. 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

1. SQL 结果：count(本表) - count(关联表)
2. 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：

1. 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。

2. 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。

3. 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。

4. 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。

算法

大数据面试中考察的算法相对容易一些，常考的有排序算法，查找算法，二叉树等，下面讲解一些最容易考的算法。

1. 排序算法

十种常见排序算法可以分为两大类：

- **比较类排序**：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破 O(nlogn)，因此也称为非线性时间比较类排序。
- **非比较类排序**：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排序。

算法复杂度：

相关概念：

- 稳定：如果 a 原本在 b 前面，而 a=b，排序之后 a 仍然在 b 的前面。
- 不稳定：如果 a 原本在 b 的前面，而 a=b，排序之后 a 可能会出现在 b 的后面。

- 时间复杂度：对排序数据的总的操作次数。反映当 n 变化时，操作次数呈现什么规律。
- 空间复杂度：是指算法在计算机内执行时所需存储空间的度量，它也是数据规模 n 的函数。

下面讲解大数据中最常考的两种：**快排和归并**

1）快速排序

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法描述

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

- 从数列中挑出一个元素，称为 “基准”（pivot）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

代码实现：

```
function quickSort(arr, left, right) {
    var len = arr.length,
        partitionIndex,
        left = typeof left != 'number' ? 0 : left,
        right = typeof right != 'number' ? len - 1 : right;

    if (left < right) {
        partitionIndex = partition(arr, left, right);
        quickSort(arr, left, partitionIndex-1);
        quickSort(arr, partitionIndex+1, right);
    }
    return arr;
}

function partition(arr, left ,right) {     // 分区操作
    var pivot = left,                      // 设定基准值（pivot）
```

```
        index = pivot + 1;
    for (var i = index; i <= right; i++) {
        if (arr[i] < arr[pivot]) {
            swap(arr, i, index);
            index++;
        }
    }
    swap(arr, pivot, index - 1);
    return index-1;
}

function swap(arr, i, j) {
    var temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

2）归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为 2-路归并。

算法描述

- 把长度为 n 的输入序列分成两个长度为 n/2 的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。

代码实现：

```
function mergeSort(arr) {
    var len = arr.length;
    if (len < 2) {
        return arr;
    }
    var middle = Math.floor(len / 2),
        left = arr.slice(0, middle),
        right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
    var result = [];
```

```
    while (left.length>0 && right.length>0) {
        if (left[0] <= right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }

    while (left.length)
        result.push(left.shift());

    while (right.length)
        result.push(right.shift());

    return result;
}
```

2. 查找算法

七大查找算法：1. 顺序查找、2. 二分查找、3. 插值查找、4. 斐波那契查找、5. 树表查找、6. 分块查找、7. 哈希查找

这些查找算法中二分查找是最容易考察的，下面讲解二分查找算法。

1）二分查找

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列，注意必须要是有序排列。

代码实现：

1. 使用递归

```
/**
 * 使用递归的二分查找
 *title:recursionBinarySearch
 *@param arr 有序数组
 *@param key 待查找关键字
 *@return 找到的位置
 */
public static int recursionBinarySearch(int[] arr,int key,int low,int high){
```

```
  if(key < arr[low] || key > arr[high] || low > high){
   return -1;
  }

  int middle = (low + high) / 2;   //初始中间位置
  if(arr[middle] > key){
   //比关键字大则关键字在左区域
   return recursionBinarySearch(arr, key, low, middle - 1);
  }else if(arr[middle] < key){
   //比关键字小则关键字在右区域
   return recursionBinarySearch(arr, key, middle + 1, high);
  }else {
   return middle;
  }

 }
```

2. 不使用递归实现（while 循环）

```
/**
  * 不使用递归的二分查找
  *title:commonBinarySearch
  *@param arr
  *@param key
  *@return 关键字位置
  */
 public static int commonBinarySearch(int[] arr,int key){
  int low = 0;
  int high = arr.length - 1;
  int middle = 0;   //定义middle

  if(key < arr[low] || key > arr[high] || low > high){
   return -1;
  }

  while(low <= high){
   middle = (low + high) / 2;
   if(arr[middle] > key){
    //比关键字大则关键字在左区域
    high = middle - 1;
   }else if(arr[middle] < key){
    //比关键字小则关键字在右区域
    low = middle + 1;
```

```
    }else{
      return middle;
    }
   }

  return -1;  //最后仍然没有找到，则返回-1
 }
```

3. 二叉树实现及遍历

定义：二叉树，是一种特殊的树，二叉树的任意一个节点的度都不大于 2，不包含度的节点称之为叶子。

遍历方式：二叉树的遍历方式有三种，中序遍历，先序遍历，后序遍历。

将一个数组中的数以二叉树的存储结构存储，并遍历打印：

代码实现：

```
import java.util.ArrayList;
import java.util.List;

public class bintree {
    public bintree left;
    public bintree right;
    public bintree root;
//    数据域
    private Object data;
    //    存节点
    public List<bintree> datas;

    public bintree(bintree left, bintree right, Object data){
        this.left=left;
        this.right=right;
        this.data=data;
    }
//    将初始的左右孩子值为空
    public bintree(Object data){
        this(null,null,data);
    }

    public bintree() {

    }
```

```
    public void creat(Object[] objs){
        datas=new ArrayList<bintree>();
        //        将一个数组的值依次转换为Node 节点
        for(Object o:objs){
            datas.add(new bintree(o));
        }
//        第一个数为根节点
        root=datas.get(0);
//        建立二叉树
        for (int i = 0; i <objs.length/2; i++) {
//            左孩子
            datas.get(i).left=datas.get(i*2+1);
//            右孩子
            if(i*2+2<datas.size()){//避免偶数的时候 下标越界
                datas.get(i).right=datas.get(i*2+2);
            }

        }

    }
//先序遍历
public void preorder(bintree root){
    if(root!=null){
        System.out.println(root.data);
        preorder(root.left);
        preorder(root.right);
    }

}
//中序遍历
    public void inorder(bintree root){
        if(root!=null){
            inorder(root.left);
            System.out.println(root.data);
            inorder(root.right);
        }

    }
//    后序遍历
    public void afterorder(bintree root){
        if(root!=null){
            System.out.println(root.data);
            afterorder(root.left);
            afterorder(root.right);
```

```
        }

    }

    public static void main(String[] args) {
        bintree bintree=new bintree();
        Object []a={2,4,5,7,1,6,12,32,51,22};
        bintree.creat(a);
        bintree.preorder(bintree.root);
    }
}
```

最后

第一时间获取最新大数据技术，尽在公众号：五分钟学大数据

搜索公众号：五分钟学大数据，学更多大数据技术！

其他大数据技术文档可下方扫码关注获取：