2024年最新大数据必备面试题

公众号@五分钟学大数据

大数据面试宝典

第五版

微信扫码关注公众号

来自公众号@五分钟学大数据

本文档来自公众号@五分钟学大数据　盗版必究

目录

本文档来自公众号@五分钟学大数据　盗版必究

本文档来自公众号@五分钟学大数据　盗版必究

本文档来自公众号@五分钟学大数据　盗版必究

本文档来自公众号@五分钟学大数据　盗版必究

本文档来自公众号@五分钟学大数据　盗版必究

本文档来自公众号@五分钟学大数据　盗版必究

前言

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

版本更新如下：

版本	时间	描述
V1.0	2020-12-18	创建
V1.2	2021-01-17	新增：spark 面试题
V1.3	2021-01-18	新增：kafka 面试题
V1.4	2021-01-20	新增：hbase 面试题
V1.5	2021-01-30	新增：flink 面试题
V3.0	2022-01-10	新增：数据仓库，算法等面试题 修复：部分答案不完整或有误
V4.0	2023-02-12	更新：数据仓库及算法； 新增：数据湖，必备 SQL 题，Clickhouse，Doris，大数据算法设计题
V5.0（此版本）	2024-05-05	更新：数据仓库及 Hive； 新增：大厂真实 SQL 题，Linux，Java，数据治理等

本文档来自公众号@五分钟学大数据　盗版必究

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。

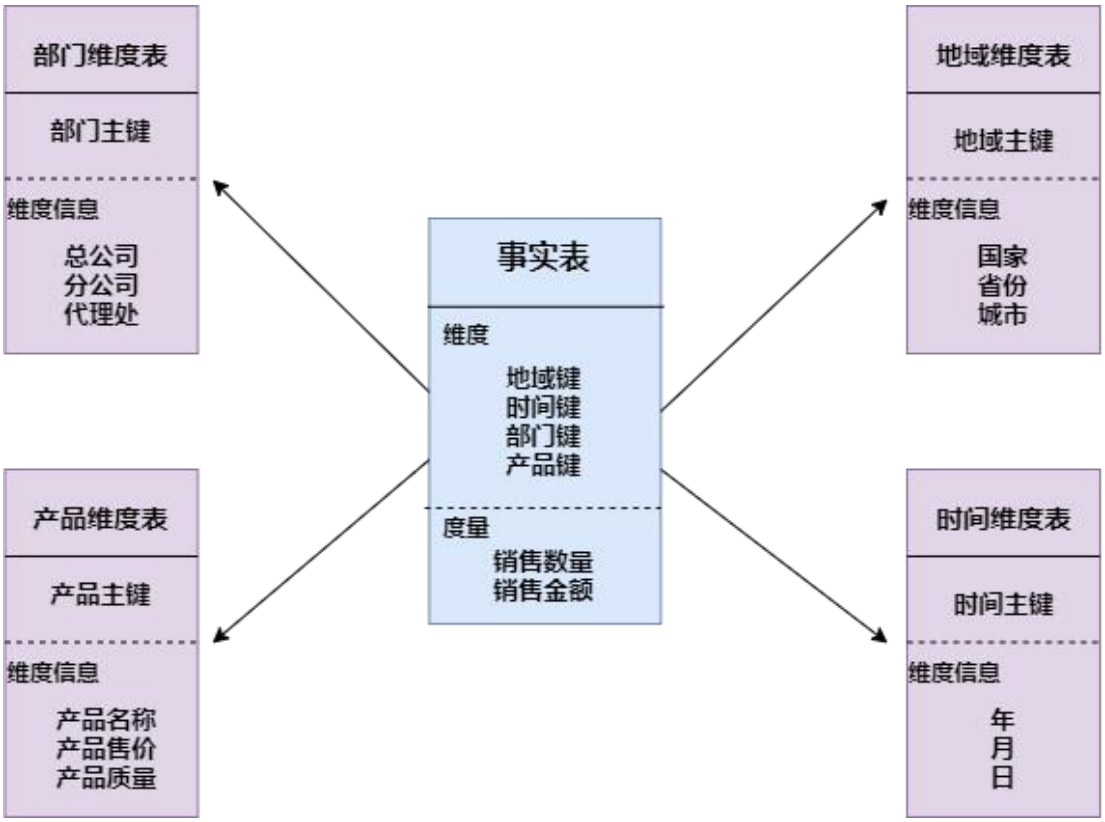
ORC:

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

本文档来自公众号@五分钟学大数据　盗版必究



星形模式

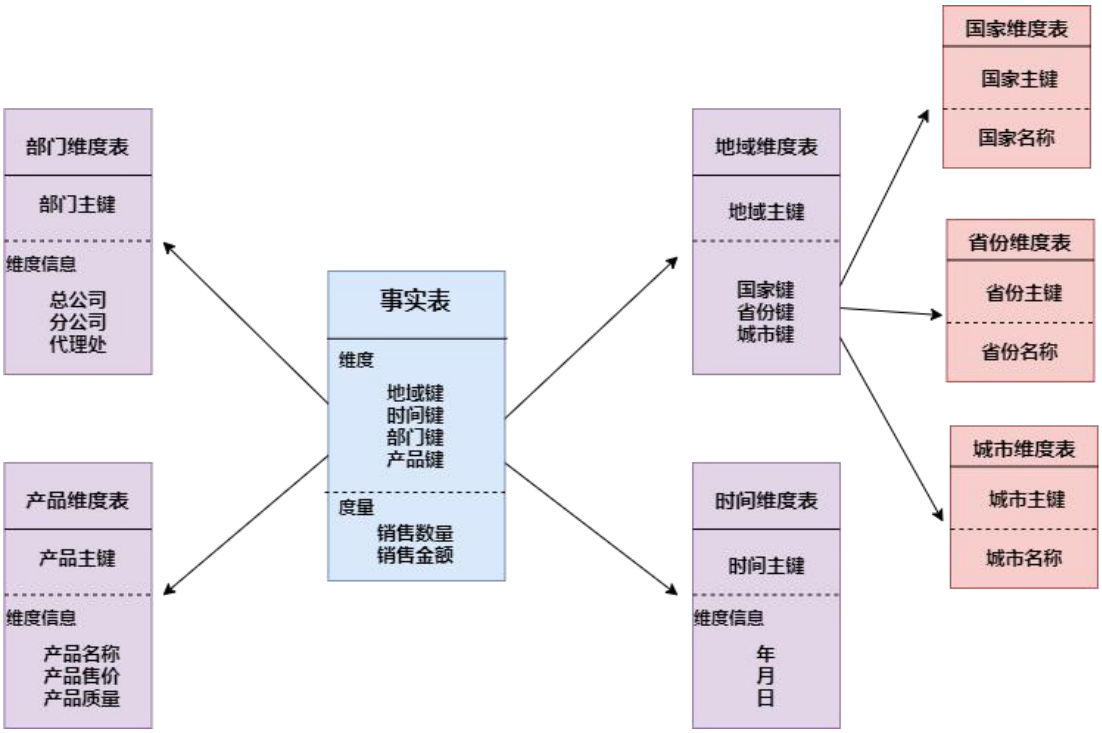
星形模式(Star Schema)是最常用的维度建模方式。星型模式是以事实表为中心，所有的维度表直接连接在事实表上，像星星一样。 星形模式的维度建模由一个事实表和一组维表成，且具有以下特点：

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。

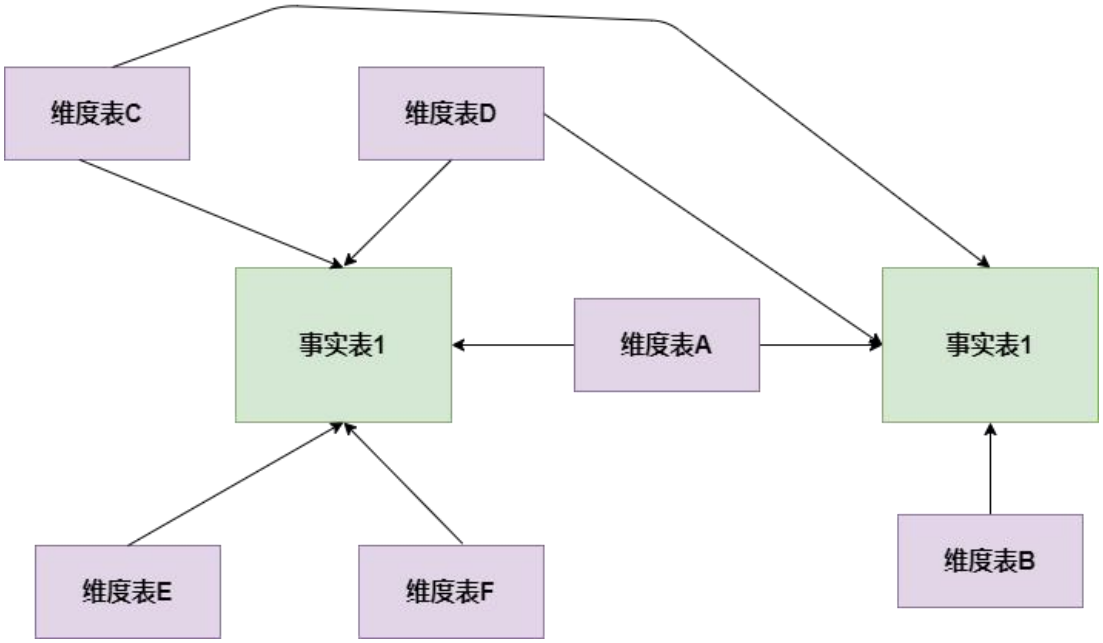
2. 雪花模型



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。

本文档来自公众号@五分钟学大数据　盗版必究

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

本文档来自公众号@五分钟学大数据　盗版必究

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

12. Tez 引擎优点？

Tez 可以将多个有依赖的作业转换为一个作业，这样只需写一次 HDFS，且中间节点较少，从而大大提升作业的计算性能。
Mr/tez/spark 区别：
Mr 引擎：多 job 串联，基于磁盘，落盘的地方比较多。虽然慢，但一定能跑出结果。一般处理，周、月、年指标。

Spark 引擎:虽然在 Shuffle 过程中也落盘,但是并不是所有算子都需要 Shuffle,尤其是多算子过程，中间过程不落盘 DAG 有向无环图。 兼顾了可靠性和效率。一般处理天指标。

Tez 引擎：完全基于内存。 注意：如果数据量特别大，慎重使用。容易 OOM。一般用于快速出结果，数据量比较小的场景。

13. Hive SQL 的执行流程

（1）解析器（SQLParser）：将 SQL 字符串转换成抽象语法树（AST）
（2）语义分析器（Semantic Analyzer）：将 AST 进一步抽象为 QueryBlock（可以理解为一个子查询划分成一个 QueryBlock）
（2）逻辑计划生成器（Logical Plan Gen）：由 QueryBlock 生成逻辑计划
（3）逻辑优化器（Logical Optimizer）：对逻辑计划进行优化
（4）物理计划生成器（Physical Plan Gen）：根据优化后的逻辑计划生成物理计划
（5）物理优化器（Physical Optimizer）：对物理计划进行优化
（6）执行器（Execution）：执行该计划，得到查询结果并返回给客户端

14. 几十张表 join 如何优化

（1）减少 join 的表数量：不影响业务前提，可以考虑将一些表进行预处理和合并，从而减少 join 操作。
（2）使用 Map Join：将小表加载到内存中，从而避免了 Reduce 操作，提高了性能。通过设置 hive.auto.convert.join 为 true 来启用自动 Map Join。
（3)使用 Bucketed Map Join:通过设置 hive.optimize.bucketmapjoin 为 true 来启用 Bucketed Map Join。
（4)使用 Sort Merge Join:这种方式在 Map 阶段完成排序,从而减少了 Reduce 阶段的计算量。通过设置 hive.auto.convert.sortmerge.join 为 true 来启用。
（5）控制 Reduce 任务数量：通过合理设置 hive.exec.reducers.bytes.per.reducer 和 mapreduce.job.reduces 参数来控制 Reduce 任务的数量。
（6）过滤不需要的数据：join 操作之前，尽量过滤掉不需要的数据，从而提高性能。
（7）选择合适的 join 顺序：将小表放在前面可以减少中间结果的数据量，提高性能。

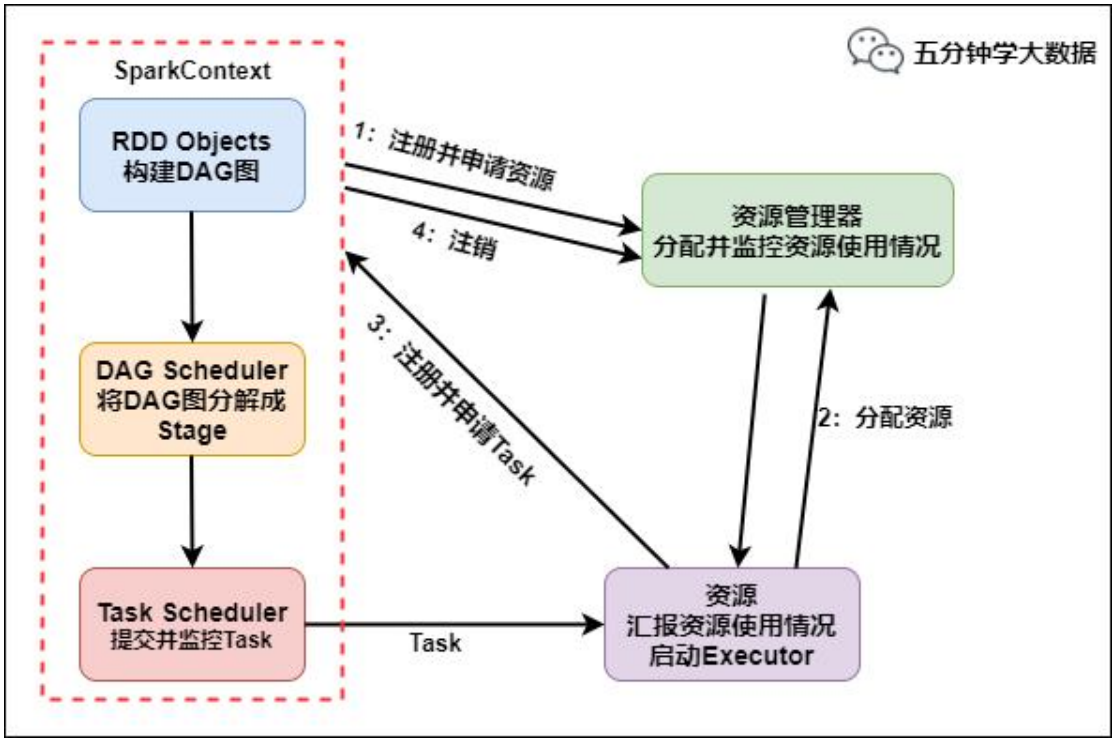
（8）使用分区：可以考虑使用分区技术。只需要读取与查询条件匹配的分区数据，从而减少数据量和计算量。

（9）使用压缩：通过对数据进行压缩，可以减少磁盘和网络 IO，提高性能。注意选择合适的压缩格式和压缩级别。

（10）调整 Hive 配置参数：根据集群的硬件资源和实际需求，合理调整 Hive 的配置参数，如内存、CPU、IO 等，以提高性能。

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。

本文档来自公众号@五分钟学大数据　盗版必究

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。

Kafka

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

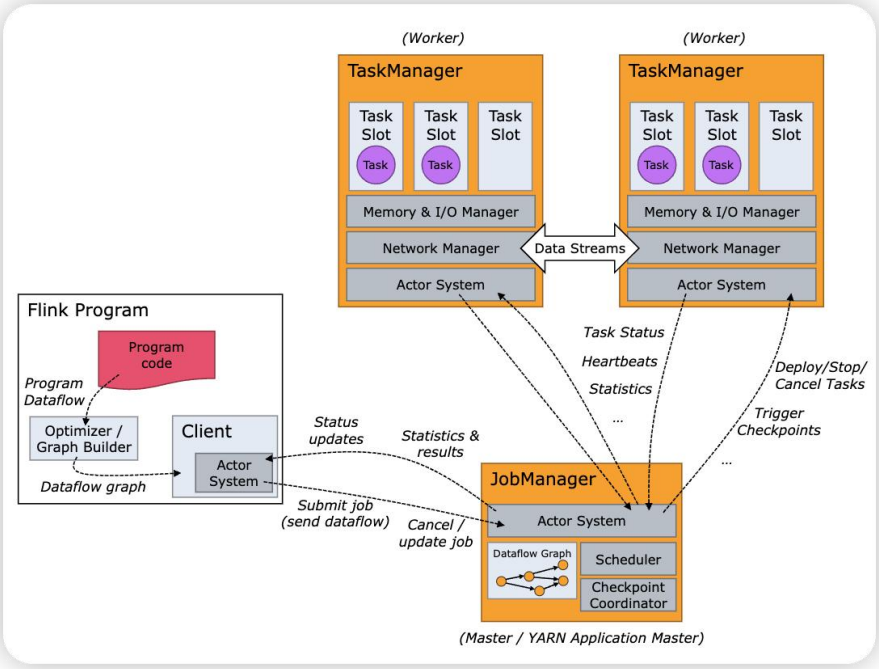
Flink 是一个面向流处理和批处理的分布式数据计算引擎，能够基于同一个 Flink 运行，可以提供流处理和批处理两种类型的功能。 在 Flink 的世界观中，一切都是由流组成的，离线数据是有界的流；实时数据是一个没有界限的流：这就是所谓的有界流和无界流。

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。

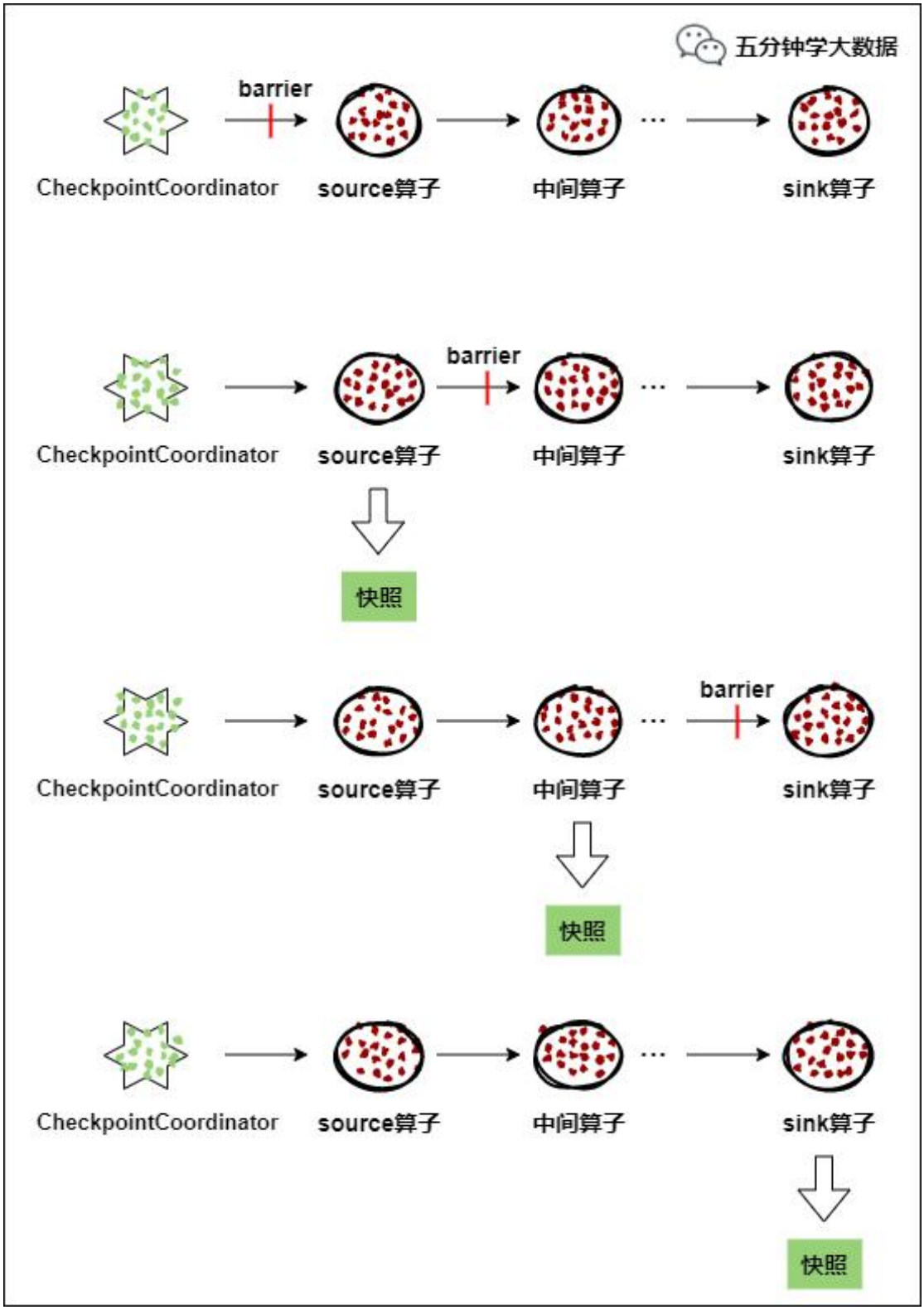
5. 介绍下 Flink 的容错机制（checkpoint）

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。

本文档来自公众号@五分钟学大数据　盗版必究

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。

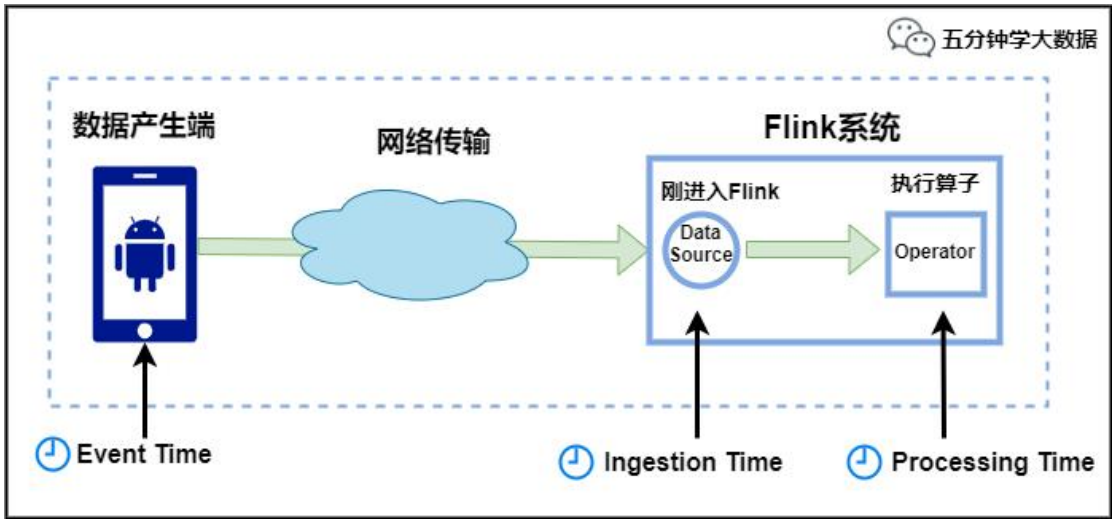
3. 解决问题的思路：

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：

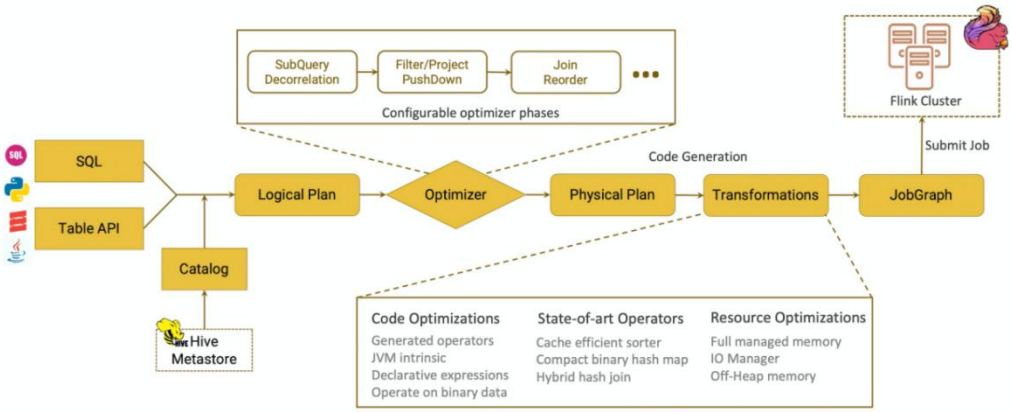
- BasicTypeInfo: 任意 Java 基本类型或 String 类型
- BasicArrayTypeInfo: 任意 Java 基本类型数组或 String 数组
- WritableTypeInfo: 任意 Hadoop Writable 接口的实现类
- TupleTypeInfo: 任意的 Flink Tuple 类型(支持 Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的 Java Tuple 实现
- CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)
- PojoTypeInfo: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

ClickHouse

本文档来自公众号@五分钟学大数据　盗版必究

1. ClickHouse 的应用场景有哪些？

1. 绝大多数请求都是用于读访问的；
2. 数据需要以大批次（大于 1000行）进行更新，而不是单行更新；
3. 数据只是添加到数据库，没有必要修改；
4. 读取数据时，会从数据库中提取出大量的行，但只用到一小部分列；
5. 表很“宽”，即表中包含大量的列；
6. 查询频率相对较低（通常每台服务器每秒查询数百次或更少）；
7. 对于简单查询，允许大约 50 毫秒的延迟；
8. 列的值是比较小的数值和短字符串（例如，每个 URL 只有 60 个字节）；
9. 在处理单个查询时需要高吞吐量（每台服务器每秒高达数十亿行）；
10. 不需要事务；
11. 数据一致性要求较低；
12. 每次查询中只会查询一个大表。除了一个大表，其余都是小表；
13. 查询结果显著小于数据源。即数据有过滤或聚合。返回结果不超过单个服务器内存。

2. ClickHouse 的优缺点

优点：

1. 为了高效的使用 CPU，数据不仅仅按列存储，同时还按向量进行处理；
2. 数据压缩空间大，减少 IO；处理单查询高吞吐量每台服务器每秒最多数十亿行；
3. 索引非 B 树结构，不需要满足最左原则；只要过滤条件在索引列中包含即可；即使在使用的数据不在索引中，由于各种并行处理机制 ClickHouse 全表扫描的速度也很快；
4. 写入速度非常快，50-200M/s，对于大量的数据更新非常适用。

缺点：

1. 不支持事务，不支持真正的删除/更新；
2. 不支持高并发，官方建议 qps 为 100，可以通过修改配置文件增加连接数，但是在服务器足够好的情况下；
3. SQL 满足日常使用 80%以上的语法，join 写法比较特殊；最新版已支持类似 SQL 的 join，但性能不好；
4. 尽量做 1000 条以上批量的写入，避免逐行 insert 或小批量的 insert，update，delete 操作，因为 ClickHouse 底层会不断的做异步的数据合并，会影响查询性能，这个在做实时数据写入的时候要尽量避开；
5. Clickhouse 快是因为采用了并行处理机制，即使一个查询，也会用服务器一半的 CPU 去执行，所以 ClickHouse 不能支持高并发的使用场景，默认单查询使用 CPU 核数为服务器核数的一半，安装时会自动识别服务器核数，可以通过配置文件修改该参数。

3. ClickHouse 的核心特性？

- 列存储：列存储是指仅从存储系统中读取必要的列数据，无用列不读取，速度非常快。ClickHouse 采用列存储，这对于分析型请求非常高效。一个典型且真实的情况是，如果我们需要分析的数据有 50 列，而每次分析仅读取其中的 5 列，那么通过列存储，我们仅需读取必要的列数据，相比于普通行存，可减少 10 倍左右的读取、解压、处理等开销，对性能会有质的影响。
- 向量化执行：在支持列存的基础上，ClickHouse 实现了一套面向 向量化处理 的计算引擎，大量的处理操作都是向量化执行的。相比于传统火山模型中的逐行处理模式，向量化执行引擎采用批量处理模式，可以大幅减少函数调用开销，降低指令、数据的 Cache Miss，提升 CPU 利用效率。并且 ClickHouse 可利用 SIMD 指令进一步加速执行效率。这部分是 ClickHouse 优于大量同类 OLAP 产品的重要因素。
- 编码压缩：由于 ClickHouse 采用列存储，相同列的数据连续存储，且底层数据在存储时是经过排序的，这样数据的局部规律性非常强，有利于获得更高的数据压缩比。此外，ClickHouse 除了支持 LZ4、ZSTD 等通用压缩算法外，还支持 Delta、DoubleDelta、Gorilla 等专用编码算法，用于进一步提高数据压缩比。

- 多索引：列存用于裁剪不必要的字段读取，而索引则用于裁剪不必要的记录读取。ClickHouse支持丰富的索引，从而在查询时尽可能的裁剪不必要的记录读取，提高查询性能。

4. 使用ClickHouse 时有哪些注意点？

分区和索引

分区粒度根据业务特点决定，不宜过粗或过细。一般选择按天分区，也可指定为 tuple()；以单表 1 亿数据为例，分区大小控制在 10-30 个为最佳。

必须指定索引列，clickhouse 中的索引列即排序列，通过 order by 指定，一般在查询条件中经常被用来充当筛选条件的属性被纳入进来；可以是单一维度，也可以是组合维度的索引；通常需要满足高级列在前、查询频率大的在前原则；还有基数特别大的不适合做索引列，如用户表的 userid 字段；通常筛选后的数据满足在百万以内为最佳。

数据采样策略

通过采用运算可极大提升数据分析的性能。

数据量太大时应避免使用select * 操作，查询的性能会与查询的字段大小和数量成线性变换；字段越少，消耗的 IO 资源就越少，性能就会越高。

千万以上数据集用order by 查询时需要搭配 where 条件和 limit 语句一起使用。

如非必须不要在结果集上构建虚拟列，虚拟列非常消耗资源浪费性能，可以考虑在前端进行处理，或者在表中构造实际字段进行额外存储。

不建议在高基列上执行distinct 去重查询，改为近似去重 uniqCombined。

多表 Join 时要满足小表在右的原则，右表关联时被加载到内存中与左表进行比较。

存储

ClickHouse 不支持设置多数据目录，为了提升数据 io 性能，可以挂载虚拟券组，一个券组绑定多块物理磁盘提升读写性能；多数查询场景 SSD 盘会比普通机械硬盘快 2-3 倍。

5. ClickHouse 的引擎有哪些？

ClickHouse 提供了大量的数据引擎，分为数据库引擎、表引擎，根据数据特点及使用场景选择合适的引擎至关重 要。

ClickHouse 引擎分类：

在以下几种情况下，ClickHouse 使用自己的数据库引擎：

1. MergeTree 系列引擎
2. Log 系列引擎
3. 与其他存储/处理系统集成的引擎
4. 特定功能的引擎

引擎分类	引擎名称	
数据库引擎	Ordinary/Dictionary/Memory/Lazy/MySQL	
数据表引擎	MergeTree系列	MergeTree 、ReplacingMergeTree 、SummingMergeTree 、AggregatingMergeTree CollapsingMergeTree 、VersionedCollapsingMergeTree 、GraphiteMergeTree
	Log系列	TinyLog 、StripeLog 、Log
	Integration Engines	Kafka 、MySQL、ODBC 、JDBC、HDFS
	Special Engines	Distributed 、MaterializedView、Dictionary 、Merge 、File、Null 、Set 、Join 、URL View、Memory 、Buffer

- 决定表存储在哪里以及以何种方式存储；
- 支持哪些查询以及如何支持；
- 并发数据访问；
- 索引的使用；
- 是否可以执行多线程请求；
- 数据复制参数。

在所有的表引擎中，**最为核心的当属 MergeTree 系列表引擎**，这些表引擎拥有最为强大的性能和最广泛的使用场合。对于非MergeTree 系列的其他引擎而言，主要用于特殊用途，场景相对有限。而MergeTree 系列表引擎是官方主推的存储引擎，支持几乎所有 ClickHouse 核心功能。

MergeTree 作为家族系列最基础的表引擎，主要有以下特点：

本文档来自公众号@五分钟学大数据　盗版必究

- 存储的数据按照主键排序：允许创建稀疏索引，从而加快数据查询速度；
- 支持分区，可以通过 PRIMARY KEY 语句指定分区字段；
- 支持数据副本；
- 支持数据采样。

6. 建表引擎参数有哪些？

1. ENGINE：ENGINE = MergeTree()，MergeTree 引擎没有参数。
2. ORDER BY：order by 设定了分区内的数据按照哪些字段顺序进行有序保存。

order by 是 MergeTree 中唯一一个必填项，甚至比primary key 还重要，因为当用户不设置主键的情况，很多处理会依照 order by 的字段进行处理。

要求：主键必须是 order by 字段的前缀字段。

如果 ORDER BY 与 PRIMARY KEY 不同，PRIMARY KEY 必须是 ORDER BY 的前缀(为了保证分区内数据和主键的有序性)。

ORDER BY 决定了每个分区中数据的排序规则;

PRIMARY KEY 决定了一级索引(primary.idx);

ORDER BY 可以指代 PRIMARY KEY，通常只用声明 ORDER BY 即可。

3. PARTITION BY：分区字段，可选。如果不填：只会使用一个分区。

分区目录：MergeTree 是以列文件+索引文件+表定义文件组成的，但是如果设定了分区那么这些文件就会保存到不同的分区目录中。

4. PRIMARY KEY：指定主键，如果排序字段与主键不一致，可以单独指定主键字段。否则默认主键是排序字段。可选。
5. SAMPLE BY：采样字段，如果指定了该字段，那么主键中也必须包含该字段。比如 SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))。可选。
6. TTL：数据的存活时间。在 MergeTree 中，可以为某个列字段或整张表设置 TTL。当时间到达时，如果是列字段级别的 TTL，则会删除这一列的数据；如果是表级别的 TTL，则会删除整张表的数据。可选。
7. SETTINGS：额外的参数配置。可选。

本文档来自公众号@五分钟学大数据　盗版必究

Doris

1. Doris 的应用场景有哪些？

首先 Doris 是一个有着 MPP 架构的分析型数据库产品。对于 PB 数量级、结构化数据可以做到亚秒级查询响应。使用上兼容 MySQL 协议，语法是标准的 SQL。Doris 本身不依赖任何其他系统，相比 Hadoop 生态产品更易于运维。
应用场景包括：固定历史报表分析、实时数据分析、交互式数据分析等。
一般情况下，用户的原始数据，比如日志或者在事务型数据库中的数据，经过流式系统或离线处理后，导入到 Doris 中以供上层的报表工具或者数据分析师查询使用。

2. Doris 的架构介绍下

Doris 的架构很简洁，只设 FE（Frontend）、BE（Backend）两种角色、两个进程。

- 以数据存储的角度观看，FE 存储、维护集群元数据；BE 存储物理数据，数据的可靠性由 BE 保证，BE 会对整个数据存储多副本。
- 以查询处理的角度观看，FE 节点接收、解析查询请求，规划查询计划，调度查询执行，返回查询结果；BE 节点依据 FE 生成的物理计划，分布式地执行查询。

FE 主要有有三个角色，一个是 Leader，一个是 Follower，还有一个 Observer。Leader 跟 Follower，主要是用来达到元数据的高可用，保证单节点宕机的情况下，元数据能够实时地在线恢复，而不影响整个服务。Observer 只是用来扩展查询节点，就是说如果在发现集群压力非常大的情况下，需要去扩展整个查询的能力，那么可以加 Observer 的节点。Observer 不参与任何的写入，只参与读取。

在使用接口方面,Doris 采用 MySQL 协议,高度兼容 MySQL 语法,支持标准 SQL,用户可以通过各类客户端工具来访问 Doris，并支持与 BI 工具的无缝对接。

3. Doris 的数据模型

Doris 的数据模型主要分为 3 类:

- Aggregate 聚合模型
- Uniq 唯一主键模型
- Duplicate 模型

Aggregate 聚合模型:

聚合模型需要用户在建表时显式的将列分为 Key 列和 Value 列。该模型会自动的对 Key 相同的行，在 Value 列上进行聚合操作。

当我们导入数据时，对于 Key 列相同的行会聚合成一行，而 Value 列会按照设置的 AggregationType 进行聚合。AggregationType 目前有以下四种聚合方式:

- SUM：求和，多行的 Value 进行累加。
- REPLACE：替代，下一批数据中的 Value 会替换之前导入过的行中的 Value。
- MAX：保留最大值。
- MIN：保留最小值。

例如：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户 id",
    `date` DATE NOT NULL COMMENT "数据灌入日期时间",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最
```

```
后一次访问时间",
    `cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
    `max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
    `min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
AGGREGATE KEY(`user_id`, `date`, `city`, `age`, `sex`)
... /* 省略 Partition 和 Distribution 信息 */
;
```

Unique 唯一主键模型：

在某些多维分析场景下，用户更关注的是如何保证 Key 的唯一性，即如何获得 Primary Key 唯一性约束。因此，我们引入了 Unique 的数据模型。该模型本质上是聚合模型的一个特例，也是一种简化的表结构表示方式。我们举例说明。这是一个典型的用户基础信息表。这类数据没有聚合需求，只需保证主键唯一性。（这里的主键为 user_id + username）。那么我们的建表语句如下：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户 id",
    `username` VARCHAR(50) NOT NULL COMMENT "用户昵称",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `phone` LARGEINT COMMENT "用户电话",
    `address` VARCHAR(500) COMMENT "用户地址",
    `register_time` DATETIME COMMENT "用户注册时间"
)
UNIQUE KEY(`user_id`, `user_name`)
... /* 省略 Partition 和 Distribution 信息 */
;
```

而这个表结构，完全同等于以下使用聚合模型描述的表结构：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户 id",
    `username` VARCHAR(50) NOT NULL COMMENT "用户昵称",
    `city` VARCHAR(20) REPLACE COMMENT "用户所在城市",
    `age` SMALLINT REPLACE COMMENT "用户年龄",
    `sex` TINYINT REPLACE COMMENT "用户性别",
    `phone` LARGEINT REPLACE COMMENT "用户电话",
    `address` VARCHAR(500) REPLACE COMMENT "用户地址",
    `register_time` DATETIME REPLACE COMMENT "用户注册时间"
)
AGGREGATE KEY(`user_id`, `user_name`)
```

```
... /* 省略 Partition 和 Distribution 信息 */
;
```

即 Unique 模型完全可以用聚合模型中的 REPLACE 方式替代。其内部的实现方式和数据存储方式也完全一样。这里不再继续举例说明。

Duplicate 模型:

在某些多维分析场景下，数据既没有主键，也没有聚合需求。因此，我们引入 Duplicate 数据模型来满足这类需求。

例如：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `timestamp` DATETIME NOT NULL COMMENT "日志时间",
    `type` INT NOT NULL COMMENT "日志类型",
    `error_code` INT COMMENT "错误码",
    `error_msg` VARCHAR(1024) COMMENT "错误详细信息",
    `op_id` BIGINT COMMENT "负责人 id",
    `op_time` DATETIME COMMENT "处理时间"
)
DUPLICATE KEY(`timestamp`, `type`)
... /* 省略 Partition 和 Distribution 信息 */
;
```

这种数据模型区别于 Aggregate 和 Uniq 模型。数据完全按照导入文件中的数据进行存储，不会有任何聚合。即使两行数据完全相同，也都会保留。 而在建表语句中指定的 DUPLICATE KEY，只是用来指明底层数据按照那些列进行排序。

4. 介绍下 Doris 的 ROLLUP

ROLLUP

ROLLUP 在多维分析中是“上卷”的意思，即将数据按**某种指定的粒度进行进一步聚合**。

在 Doris 中，我们将用户通过建表语句创建出来的表称为 Base 表（Base Table）。在 Base 表之上，我们可以创建任意多个 ROLLUP 表。这些 ROLLUP 的数据是基于 Base 表产生的，并且在物理上是**独立存储**的。

ROLLUP 表的基本作用，在于在 Base 表的基础上，获得更粗粒度的聚合数据。

Duplicate 模型中的 ROLLUP

因为 Duplicate 模型没有**聚合的语意**。所以该模型中的 ROLLUP，已经失去了“上卷”这一层含义。而仅仅是作为调整列顺序，以命中前缀索引的作用。

5. Doris 的前缀索引了解吗？

不同于传统的数据库设计，Doris 不支持在任意列上创建索引。Doris 这类 MPP 架构的 OLAP 数据库，通常都是通过提高并发，来处理大量数据的。

本质上，Doris 的数据存储在类似 SSTable（Sorted String Table）的数据结构中。该结构是一种有序的数据结构，可以按照指定的列进行排序存储。在这种数据结构上，以排序列作为条件进行查找，会非常的高效。

在 Aggregate、Uniq 和 Duplicate 三种数据模型中。底层的数据存储，是按照各自建表语句中，AGGREGATE KEY、UNIQ KEY 和 DUPLICATE KEY 中指定的列进行排序存储的。

而前缀索引，即在排序的基础上，实现的一种根据给定前缀列，快速查询数据的索引方式。

在建表时，正确的选择列顺序，能够极大地提高查询效率。

ROLLUP 调整前缀索引

因为建表时已经指定了列顺序，所以一个表只有一种前缀索引。这对于使用其他不能命中前缀索引的列作为条件进行的查询来说，效率上可能无法满足需求。因此，我们可以通过创建 ROLLUP 来人为的调整列顺序，以获得更好的查询效率。

6. 讲下 Doris 的物化视图

物化视图是将预先计算(根据定义好的 SELECT 语句)好的数据集，存储在 Doris 中的一个特殊的表。

物化视图的出现主要是为了满足用户，**既能对原始明细数据的任意维度分析，也能快速的对固定维度进行分析查询。**

使用场景（物化视图主要针对 Duplicate 明细模型做聚合操作）

- 分析需求覆盖明细数据查询以及固定维度查询两方面。
- 查询仅涉及表中的很小一部分列或行。
- 查询包含一些耗时处理操作，比如：时间很久的聚合操作等。
- 查询需要匹配不同前缀索引。

优势

- 对于那些经常重复的使用相同的子查询结果的查询性能大幅提升。

本文档来自公众号@五分钟学大数据　盗版必究

- Doris 自动维护物化视图的数据，无论是新的导入，还是删除操作都能保证 base 表和物化视图表的数据一致性。无需任何额外的人工维护成本。
- 查询时，会自动匹配到最优物化视图，并直接从物化视图中读取数据。

7. 物化视图和 Rollup 的区别是什么

在没有物化视图功能之前，用户一般都是使用 Rollup 功能通过预聚合方式提升查询效率的。但是 Rollup 具有一定的局限性，他不能基于明细模型做预聚合。物化视图则在覆盖了 Rollup 的功能的同时，还能支持更丰富的聚合函数。所以物化视图其实是 Rollup 的一个超集。

物化视图的局限性

1. 物化视图的聚合函数的参数不支持表达式仅支持单列，比如： sum(a+b) 不支持。
2. 如果删除语句的条件列，在物化视图中不存在，则不能进行删除操作。如果一定要删除数据，则需要先将物化视图删除，然后方可删除数据。
3. 单表上过多的物化视图会影响导入的效率：导入数据时，物化视图和 base 表数据是同步更新的，如果一张表的物化视图表超过 10 张，则有可能导致导入速度很慢。这就像单次导入需要同时导入 10 张表数据是一样的。
4. 相同列，不同聚合函数，不能同时出现在一张物化视图中，比如：select sum(a), min(a) from table 不支持。
5. 物化视图针对 Unique Key 数据模型，只能改变列顺序，不能起到聚合的作用，所以在 Unique Key 模型上不能通过创建物化视图的方式对数据进行粗粒度聚合操作

数据仓库

本文档来自公众号@五分钟学大数据　盗版必究

推荐数仓建设好文，建议读一读：万字详解整个数据仓库建设体系

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

本文档来自公众号@五分钟学大数据　盗版必究

4. 事实表的类型？

事实表有：事务事实表、周期快照事实表、累积快照事实表、非事实事实表。

1）事务事实表

事务事实表记录的是事务层面的事实，保存的是最原子的数据，也称“原子事实表”。事务事实表中的数据在事务事件发生后产生，数据的粒度通常是每个事务记录一条记录。

2）周期快照事实表

以具有规律性的、可预见的时间间隔来记录事实。它统计的是间隔周期内的度量统计，每个时间段一条记录，是在事务事实表之上建立的聚集表。

3）累积快照事实表

累积快照表记录的不确定的周期的数据。代表的是完全覆盖一个事务或产品的生命周期的时间跨度，通常具有多个日期字段，用来记录整个生命周期中的关键时间点。

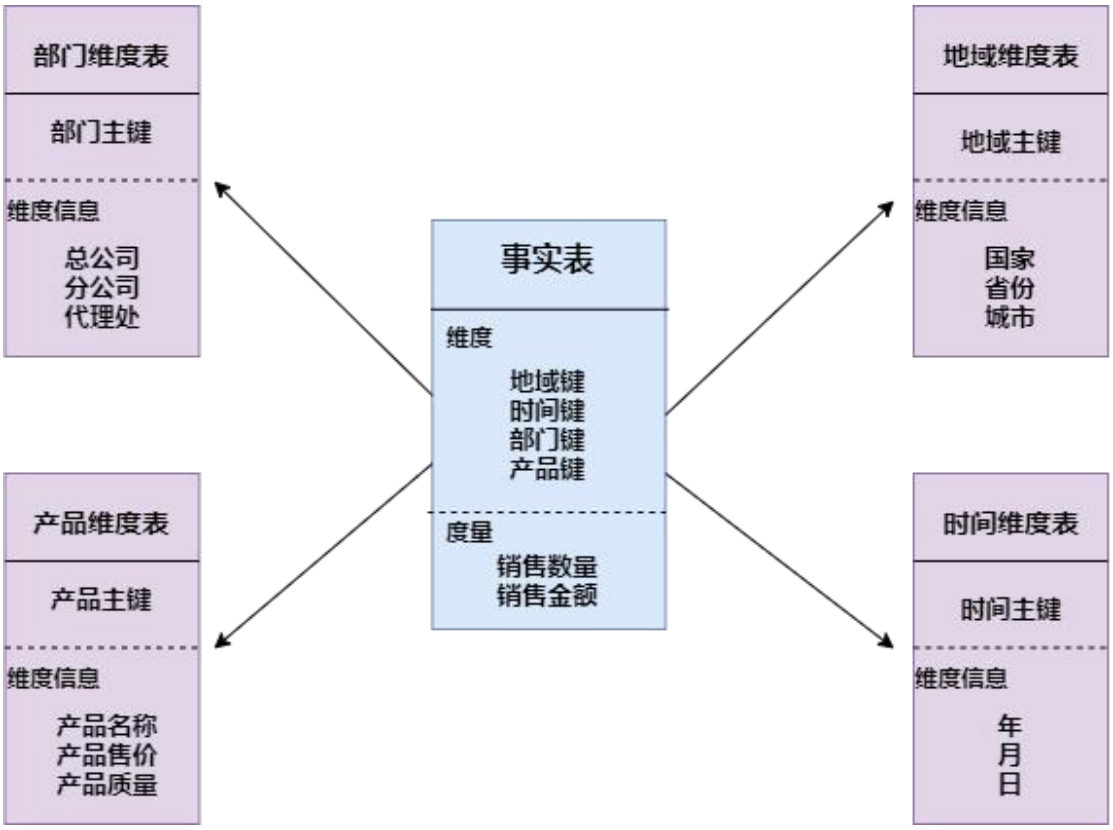
4）非事实型事实表

这个与上面三个有所不同。事实表中通常要保留度量事实和多个维度外键，度量事实是事实表的关键所在。
非事实表中没有这些度量事实，只有多个维度外键。非事实型事实表通常用来跟踪一些事件或说明某些活动的范围。
第一类非事实型事实表是用来跟踪事件的事实表。例如：学生注册事件
第二类非事实型事实表是用来说明某些活动范围的事实表。例如：促销范围事实表。

5. 星型模型和雪花模型的区别

1）星型模式

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；

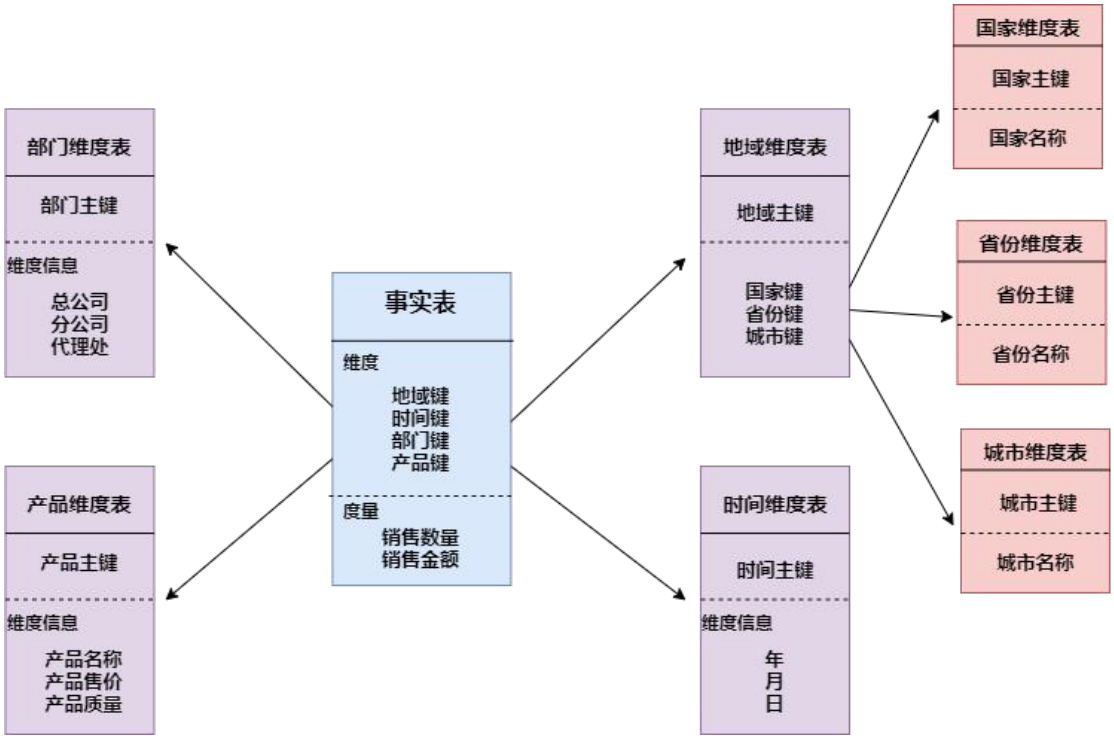


星型模式

2. 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，性能也比星型模型要低。所以一般不是很常用

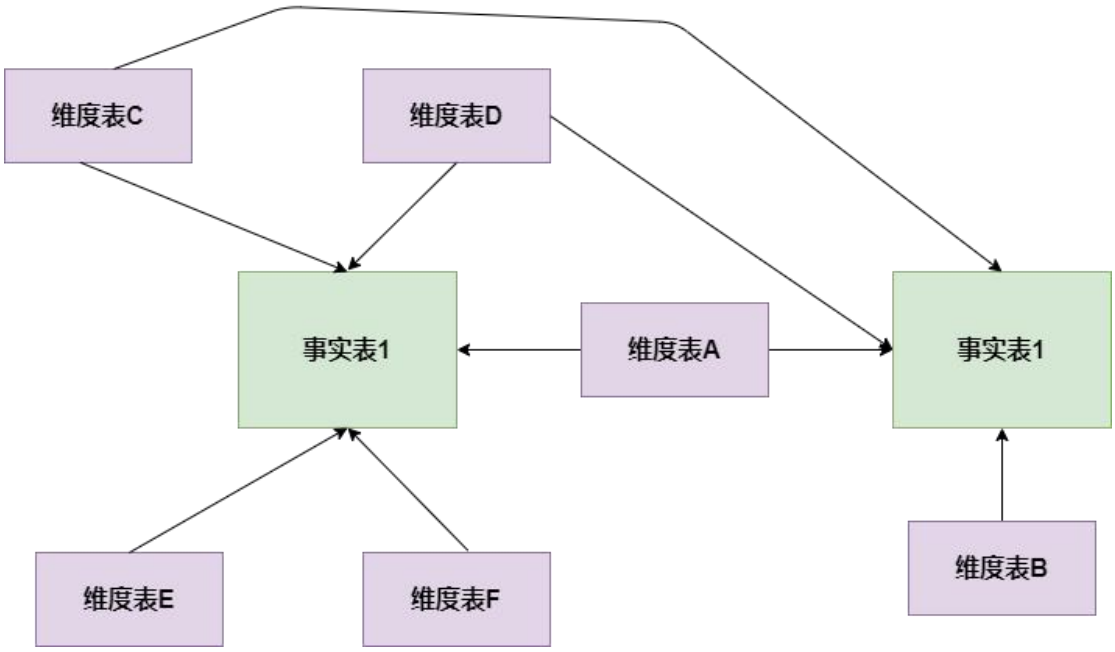
本文档来自公众号@五分钟学大数据　盗版必究



雪花模式

3．星座模式

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。



星座模型

本文档来自公众号@五分钟学大数据　盗版必究

6. 数据漂移如何解决?

1）什么是数据漂移？

通常是指 ods 表的同一个业务日期数据中包含了前一天或后一天凌晨附近的数据或者丢失当天变更的数据，这种现象就叫做漂移，且在大部分公司中都会遇到的场景。

2）如何解决数据漂移问题？

通常有两种解决方案：

1. 多获取后一天的数据，保障数据只多不少
2. 通过多个时间戳字段来限制时间获取相对准确的数据

第一种方案比较暴力，这里不做过多解释，主要来讲解一下第二种解决方案。（这种解决方案在大数据之路这本书有体现）。

第一种方案里，时间戳字段分为四类：

1. 数据库表中用来标识数据记录更新时间的时间戳字段（假设这类字段叫 modified time ）。
2. 数据库日志中用来标识数据记录更新时间的时间戳字段 •（假设这类字段叫 log_time）。
3. 数据库表中用来记录具体业务过程发生时间的时间戳字段 （假设这类字段叫 proc_time）。
4. 标识数据记录被抽取到时间的时间戳字段（假设这类字段 extract time）。

理论上这几个时间应该是一致的，但往往会出现差异，造成的原因可能为：

1. 数据抽取需要一定的时间，extract_time 往往晚于前三个时间。
2. 业务系统手动改动数据并未更新 modfied_time。
3. 网络或系统压力问题，log_time 或 modified_time 晚于 proc_time。

通常都是根据以上的某几个字段来切分 ODS 表，这就产生了数据漂移。具体场景如下：

1. 根据 extract_time 进行同步。
2. 根据 modified_time 进行限制同步， 在实际生产中这种情况最常见，但是往往会发生不更新 modified time而导致的数据遗漏，或者凌晨时间产生的数据记录漂移到后天 。由于网络或者系统压力问题， log_time 会晚 proc_time ，从而导致凌晨时间产生的数据记录漂移到后一天。

3. 根据 proc_time 来限制，会违背 ods 和业务库保持一致的原则，因为仅仅根据 proc_time 来限制，会遗漏很多其他过程的变化。

第二种解决方案：

1. 首先通过 log_time 多同步前一天最后 15 分钟和后一天凌晨开始 15 分钟的数据，然后用modified_time 过滤非当天的数据，这样确保数据不会因为系统问题被遗漏。
2. 然后根据 log_time 获取后一天 15 分钟的数据，基于这部分数据，按照主键根据 log_time 做升序排序，那么第一条数据也就是最接近当天记录变化的。
3. 最后将前两步的数据做全外连接，通过限制业务时间 proc_time 来获取想要的数据。

7. 维度建模和范式建模的区别

通常数据建模有以下几个流程：

1. 概念建模：即通常先将业务划分多个主题。
2. 逻辑建模：即定义各种实体、属性和关系。
3. 物理建模：设计数据对象的物理实现，比如表字段类型、命名等。

那么范式建模，即 3NF 模型具有以下特点：

1. 原子性，即**数据不可分割**。
2. 基于第一个条件，实体属性完全依赖于主键，不能存在仅依赖主关键字一部分属性。即**不能存在部分依赖**。
3. 基于第二个条件，任何非主属性不依赖于其他非主属性。即**消除传递依赖**。

基于以上三个特点，3NF 的最终目的就是为了降低数据冗余，保障数据一致性；同时也有了数据关联逻辑复杂的缺点。

而维度建模是面向分析场景的，主要关注点在于快速、灵活，能够提供大规模的数据响应。

常用的维度模型类型主要有：

1. 星型模型：即由一个事实表和一组维度表组成，每个维表都有一个维度作为主键。事实表居中，多个维表呈辐射状分布在四周，并与事实表关联，形成一个星型结构。
2. 雪花模型：在星型模型的基础上，基于范式理论进一步层次化，将某些维表扩展成事实表，最终形成雪花状结构。

3. 星系模型：基于多个事实表，共享一些维度表。

8. 谈谈元数据的理解？

狭义来讲就是用来描述数据的数据。

广义来看，除了业务逻辑直接读写处理的业务数据，所有其他用来维护整个系统运转所需要的数据，都可以较为元数据。

定义：元数据 metadata 是关于数据的数据。在数仓系统中，元数据可以帮助数据仓库管理员和数据仓库开发人员方便的找到他们所关心的数据；元数据是描述数据仓库内部数据的结构和建立方法的数据。按照用途可分为：技术元数据、业务元数据。

技术元数据

存储关于数据仓库技术细节的数据，用于开发和管理数据仓库使用的数据。

1. 数据仓库结构的描述，包括数据模式、视图、维、层次结构和导出数据的定义，以及数据集市的位置和内容。
2. 业务系统、数据仓库和数据集市的体系结构和模式。
3. 由操作环境到数据仓库环境的映射，包括元数据和他们的内容、数据提取、转换规则和数据刷新规则、权限等。

业务元数据

从业务角度描述了数据仓库中的数据，他提供了介于使用者和实际系统之间的语义层，使不懂计算机技术的业务人员也能读懂数仓中的数据。

1. 企业概念模型：表示企业数据模型的高层信息。整个企业业务概念和相互关系。以这个企业模型为基础，不懂 sql 的人也能做到心中有数
2. 多维数据模型。告诉业务分析人员在数据集市中有哪些维、维的类别、数据立方体以及数据集市中的聚合规则。
3. 业务概念模型和物理数据之间的依赖。业务视图和实际数仓的表、字段、维的对应关系也应该在元数据知识库中有所体现。

9. 数仓如何确定主题域？

主题

主题是在较高层次上将数据进行综合、归类和分析利用的一个抽象概念，每一个主题基本对应一个宏观的分析领域。在逻辑意义上，它是对企业中某一宏观分析领域所涉及的分析对象。

面向主题的数据组织方式，就是在较高层次上对分析对象数据的一个完整并且一致的描述，能刻画各个分析对象所涉及的企业各项数据，以及数据之间的联系。

主题是根据分析的要求来确定的。

主题域

1. **从数据角度看（集合论）**

主题语通常是联系较为紧密的数据主题的集合。可以根据业务的关注点，将这些数据主题划分到不同的主题域。主题域的确定由最终用户和数仓设计人员共同完成。

2. **从需要建设的数仓主题看（边界论）**

主题域是对某个主题进行分析后确定的主题的边界。

数仓建设过程中，需要对主题进行分析，确定主题所涉及到的表、字段、维度等界限。

3. **确定主题内容**

数仓主题定义好以后，数仓中的逻辑模型也就基本成形了，需要在主题的逻辑关系中列出属性和系统相关行为。此阶段需要定义好数据仓库的存储结构，向主题模型中添加所需要的信息和能充分代表主题的属性组。

10. 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

11. 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。

12. 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

13. 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

14. 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

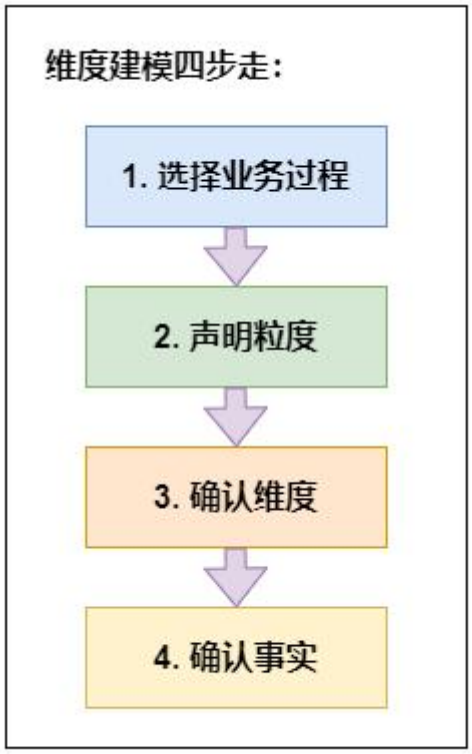
所有工作中最有意义的还是商业决策，通过数据来判断应该做什么。这是商业分析最终的目的。

本文档来自公众号@五分钟学大数据　盗版必究

15. 维度建模的过程

我们知道维度建模的表类型有事实表，维度表；模式有星形模型，雪花模型，星座模型这些概念了，但是实际业务中，给了我们一堆数据，我们怎么拿这些数据进行数仓建设呢，数仓工具箱作者根据自身 60 多年的实际业务经验，给我们总结了如下四步，请务必记住！

数仓工具箱中的维度建模四步走：



请**牢记**以上四步，不管什么业务，就按照这个步骤来，顺序不要搞乱，因为这四步是环环相扣，步步相连。下面详细拆解下每个步骤怎么做

1、选择业务过程 维度建模是紧贴业务的，所以必须以业务为根基进行建模，那么选择业务过程，顾名思义就是在整个业务流程中选取我们需要建模的业务，根据运营提供的需求及日后的易扩展性等进行选择业务。比如商城，整个商城流程分为商家端，用户端，平台端，运营需求是总订单量，订单人数，及用户的购买情况等，我们选择业务过程就选择用户端的数据，商家及平台端暂不考虑。业务选择非常重要，因为后面所有的步骤都是基于此业务数据展开的。

2、**声明粒度** 先举个例子：对于用户来说，一个用户有一个身份证号，一个户籍地址，多个手机号，多张银行卡，那么与用户粒度相同的粒度属性有身份证粒度，户籍地址粒度，比用户粒度更细的粒度有手机号粒度，银行卡粒度，存在一对一的关系就是相同粒度。为什么要提相同粒度呢，因为维度建模中要求我们，在**同一事实表**中，必须具有**相同的粒度**，同一事实表中不要混用多种不同的粒度，不同的粒度数据建立不同的事实表。并且从给定的业务过程获取数据时，强烈建议从关注原子粒度开始设计，也就是从最细粒度开始，因为原子粒度能够承受无法预期的用户查询。但是上卷汇总粒度对查询性能的提升很重要的，所以对于有明确需求的数据，我们建立针对需求的上卷汇总粒度，对需求不明朗的数据我们建立原子粒度。

3、确认维度 维度表是作为业务分析的入口和描述性标识，所以也被称为数据仓库的“灵魂”。在一堆的数据中怎么确认哪些是维度属性呢，如果该列是对具体值的描述，是一个文本或常量，某一约束和行标识的参与者，此时该属性往往是维度属性，数仓工具箱中告诉我们**牢牢掌握事实表的粒度，就能将所有可能存在的维度区分开**，并且要**确保维度表中不能出现重复数据，应使维度主键唯一**

4、确认事实 事实表是用来度量的，基本上都以数量值表示，事实表中的每行对应一个度量，每行中的数据是一个特定级别的细节数据，称为粒度。维度建模的核心原则之一**是同一事实表中的所有度量必须具有相同的粒度**。这样能确保不会出现重复计算度量的问题。有时候往往不能确定该列数据是事实属性还是维度属性。记住**最实用的事实就是数值类型和可加类事实**。所以可以通过分析该列是否是一种包含多个值并作为计算的参与者的度量，这种情况下该列往往是事实。

本文档来自公众号@五分钟学大数据　盗版必究

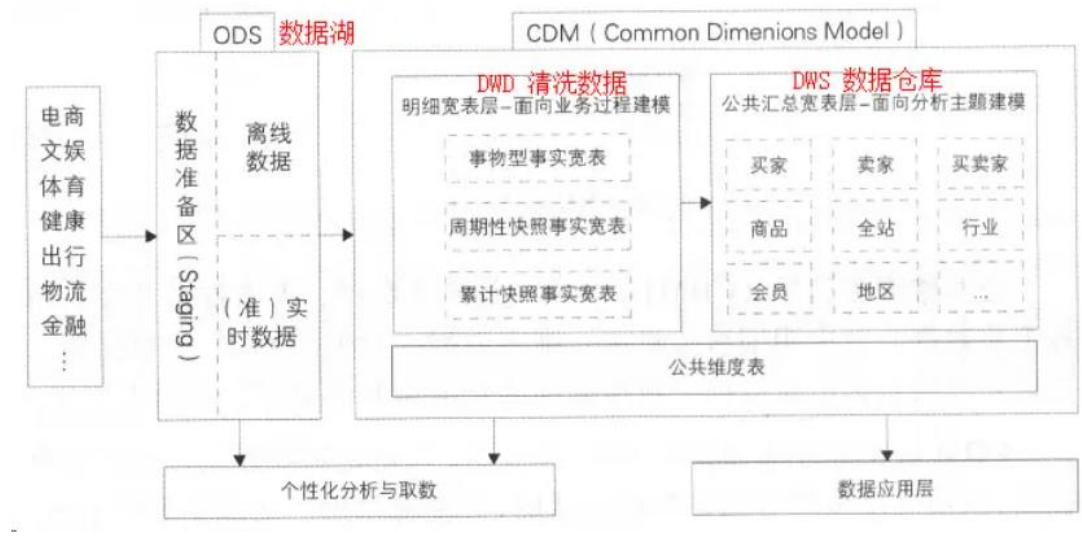
数据湖知识了解推荐阅读：**万字详解数据仓库、数据湖、数据中台和湖仓一体**

1. 什么是数据湖

本文首发于公众号【五分钟学大数据】，点击获取：**数仓建设保姆级教程**

数据湖是一种不断演进中、可扩展的大数据存储、处理、分析的基础设施；以数据为导向，实现任意来源、任意速度、任意规模、任意类型数据的全量获取、全量存储、多模式处理与全生命周期管理；并通过与各类外部异构数据源的交互集成，支持各类企业级应用。

用架构图能很快说明白，用阿里的数据架构图来说：



- ODS（operational data store，staging area）存储来自各业务系统（生产系统）的原始数据，即为数据湖。
- CDM 为经过整合、清洗的数据。其中的 DWS 汇总层，为面向主题的数据仓库（狭义），用于 BI 报表出数。

简单来说，数据湖的定义就是原始数据保存区. 虽然这个概念国内谈的少，但绝大部分互联网公司都已经有了。国内一般把整个 HDFS 叫做数仓（广义），即存放所有数据的地方。

2. 数据湖的发展

数据湖最早是 2011 年由 Pentaho 的首席技术官 James Dixon 提出的一个概念，他认为诸如数据集市，数据仓库由于其有序性的特点，势必会带来数据孤岛效应，而数据湖可以由于其开放性的特点可以解决数据孤岛问题。
为什么不是数据河？
因为，数据要能存，而不是一江春水向东流。
为什么不是数据池？
因为，要足够大，大数据太大，一池存不下。
为什么不是数据海？
因为，企业的数据要有边界，可以流通和交换，但更注重隐私和安全，“海到无边天作岸”，那可不行。
所以数据要能“存”，数据要够“存”，数据要有边界地“存”。企业级的数据是需要长期积淀的，因此是“数据湖”。
同时湖水天然会进行分层，满足不同的生态系统要求，这与企业建设统一数据中心，存放管理数据的需求是一致的。热数据在上层方便流通应用，温数据、冷数据位于数据中心的不同存储介质之中，达到数据存储容量与成本的平衡。
但随着数据湖在各类企业的应用，大家都觉得：嗯，这个数据有用，我要放进去；那个数据也有用，我也要放进去；于是把所有的数据不假思索地扔进基于数据湖的相关技术或工具中，没有规则不成方圆，当我们认为所有数据都有用时，那么所有的数据都是垃圾，数据湖也变成了造成企业成本高企的数据沼泽。

3. 数据湖有哪些优势

- **轻松地收集数据**：数据湖与数据仓库的一大区别就是，Schema On Read，即在使用数据时才需要 Schema 信息；而数据仓库是 Schema On Write，即在存储数据时就需要设计好 Schema。这样，由于对数据写入没有限制，数据湖可以更容易的收集数据。

- **从数据中发掘更多价值**：数据仓库和数据市场由于只使用数据中的部分属性，所以只能回答一些事先定义好的问题；而数据湖存储所有最原始、最细节的数据，所以可以回答更多的问题。并且数据湖允许组织中的各种角色通过自助分析工具，对数据进行分析，以及利用 AI、机器学习的技术，从数据中发掘更多的价值。
- **消除数据孤岛**：数据湖中汇集了来自各个系统中的数据，这就消除了数据孤岛问题。
- **具有更好的扩展性和敏捷性**：数据湖可以利用分布式文件系统来存储数据，因此具有很高的扩展能力。开源技术的使用还降低了存储成本。数据湖的结构没那么严格，因此天生具有更高的灵活性，从而提高了敏捷性。

4. 数据湖应该具备哪些能力

1. **数据集成能力**：

需要具备把各种数据源接入集成到数据湖中的能力。数据湖的存储也应该是多样的，比如 HDFS、HIVE、HBASE 等等。

2. **数据治理能力**：

治理能力的核心是维护好数据的元数据（metadata）。强制要求所有进入数据湖的数据必须提供相关元数据，应该作为最低限度的治理管控。没有元数据，数据湖就面临成为数据沼泽的风险。更丰富的功能还包括：

- 自动提取元元数据，并根据元数据对数据进行分类，形成数据目录。

- 自动对数据目录进行分析，可以基于 AI 和机器学习的方法，发现数据之间的关系。
- 自动建立数据之间血缘关系图。
- 跟踪数据的使用情况，以便将数据作为产品，形成数据资产。

3. **数据搜索和发现能力**：

如果把整个互联网想象成一个巨大的数据湖。那么，之所以人们可以这么有效的利用这个湖中的数据，就是因为有了 Google 这样的搜索引擎。人们可以通过搜索，方便地找到他们想要的数据，进而进行分析。搜索能力是数据湖的十分重要的能力。

4. **数据安全管控能力**：

对数据的使用权限进行管控，对敏感数据进行脱敏或加密处理，也是数据湖能商用所必须具备的能力。

5. **数据质量检验能力**：

数据质量是分析正确的关键。因此必须对进入数据湖中的数据的质量情况进行检验。及时发现数据湖中数据质量的问题。为有效的数据探索提供保障。

6. **自助数据探索能力**：

应该具备一系列好用的数据分析工具，以便各类用户可以对数据湖中的数据进行自助探索。包括：

- 支持对流、NoSQL、图等多种存储库的联合分析能力
- 支持交互式的大数据 SQL 分析
- 支持 AI、机器学习分析
- 支持类似 OLAP 的 BI 分析
- 支持报表的生成

5. 数据湖的实现遇到了哪些问题

数据湖刚提出来时，只是一个朴素的理念。而从理念变成一个可以落地的系统，就面临着许多不得不考虑的现实问题：
首先，把所有原始数据都存储下来的想法，要基于一个前提，就是存储成本很低。而今数据产生的速度越来越快、产生的量越来越大的情况下，把所有原始数据，

不分价值大小，都存储下来，这个成本在经济上能不能接受，可能需要打一个问号。

其次，数据湖中存放这各类最原始的明细数据，包括交易数据、用户数据等敏感数据，这些数据的安全怎么保证？用户访问的权限如何控制？

再次，湖中的数据怎么治理？谁对数据的质量、数据的定义、数据的变更负责？如何确保数据的定义、业务规则的一致性？

数据湖的理念很好，但是它现在还缺乏像数据仓库那样，有一整套方法论为基础，有一系列具有可操作性的工具和生态为支撑。正因如此，目前把 Hadoop 用来对特定的、高价值的数据进行处理，构建数据仓库的模式，取得了较多的成功；而用来落实数据湖理念的模式，遭遇了一系列的失败。这里，总结一些典型的数据湖失败的原因：

1. **数据沼泽**：当越来越多的数据接入到数据湖中，但是却没有有效的方法跟踪这些数据，数据沼泽就发生了。在这种失败中，人们把所有东西都放在 HDFS 中，期望以后可以发掘些什么，可没多久他们就忘那里有什么。
2. **数据泥团**：各种各样的新数据接入进数据湖中，它们的组织形式、质量都不一样。 由于缺乏用于检查，清理和重组数据的自助服务工具，使得这些数据很难创造价值。
3. **缺乏自助分析工具**：由于缺乏好用的自助分析工具，直接对数据湖中的数据分析很困难。一般都是数据工程师或开发人员创建一个整理后的小部分数据集，把这些数据集交付给更广泛的用户，以便他们使用熟悉的工具进行数据分析。这限制了更广泛的人参与到探索大数据中，降低了数据湖的价值。
4. **缺乏建模的方法论和工具**：在数据湖中，似乎每一项工作都得从头开始，因为以前的项目产生的数据几乎没有办法重用。 其实，我们骂数据仓库很难变化以适应新需求，这其中有个原因就是它花很多时间来对数据进行建模，而正是有了这些建模，使得数据可以共享和重用。数据湖也需要为数据建模，不然每次分析师都得从头开始。
5. **缺少数据安全管理**：通常的想法是每个人都可以访问所有数据，但这是行不通的。企业对自己的数据是有保护本能的，最终一定是需要数据安全管理的。
6. **一个数据湖搞定一切**：大家都对能在一个库中存储所有数据的想法很兴奋。然而，数据湖之外总会有新的存储库，很难把他们全都消灭掉。 其实，

大多数公司所需的，是可以对多种存储库联合访问功能。是不是在一个地方存储，并不是那么重要。

6. 数据湖与数据仓库的区别

数据仓库，准确说，是面向**历史数据沉淀和分析使用的**，有三大特点：

- 其一是**集成性**，由于数据来源众多，因而需要技术和规范来统一存储方式；
- 其二是**非易失和随时间变化**，数据仓库存储了过去每一天的快照且通常不更新，使用者可以在任一天向前或者向后对比数据的变化；
- 其三是**面向主题**，根据业务对数据进行有效的编码，让理论最佳值在应用中落地。

数据湖，准确说，其出发点是**补全数据仓库实时处理能力、交互式分析能力**等新技术缺失的情况。其最重要的特点，就是丰富的计算引擎：批处理、流式、交互式、机器学习，该有的，应有尽有，企业需要什么，就用什么。数据湖也有三个特征：

- 其一是**灵活性**，默认业务的不确定性是常态的，在无法预期未来变化时，技术设施基础，就要具备“按需”贴合业务的能力；
- 其二是**管理性**，数据湖需要保存原始信息和处理后的信息，在数据源、数据格式、数据周期等维度上，能够追溯数据的接入、存储、分析和使用等流动过程；
- 其三是**多态性**，本身的引擎需要进可能的丰富，因为业务场景不固定，而多态的引擎支持、扩展能力，能够较好的适应业务的快速变化。

7. 为什么要做数据湖？区别在于？

数据湖和数仓，就是原始数据和数仓模型的区别。因为数仓（狭义）中的表，主要是事实表-维度表，主要用于 BI、出报表，和原始数据是不一样的。

为什么要强调数据湖呢？

真正的原因在于，data science 和 machine learning 进入主流了，需要用原始数据做分析，而数仓的维度模型则通常用于聚合。

另一方面，机器学习用到的数据，也不止于结构化数据。用户的评论、图像这些非结构化数据，也都可以应用到机器学习中。

本文档来自公众号@五分钟学大数据　盗版必究

特性	数据仓库	数据湖
数据	来自事务系统、运营数据库和业务线应用程序的关系数据	来自 IoT 设备、网站、移动应用程序、社交媒体和企业应用程序的非关系和关系数据
Schema	设计在数据仓库实施之前（写入型 Schema）	写入在分析时（读取型 Schema）
性价比	更快查询结果会带来较高存储成本	更快查询结果只需较低存储成本
数据质量	可作为重要事实依据的高度监管数据	任何可以或无法进行监管的数据（例如原始数据）
用户	业务分析师	数据科学家、数据开发人员和业务分析师（使用监管数据）
分析	批处理报告、BI 和可视化	机器学习、预测分析、数据发现和分析

但数据湖背后其实还有更大的区别：

- 传统数仓的工作方式是集中式的：业务人员给需求到数据团队，数据团队根据要求加工、开发成维度表，供业务团队通过 BI 报表工具查询。
- 数据湖是开放、自助式的（self-service）：开放数据给所有人使用，数据团队更多是提供工具、环境供各业务团队使用（不过集中式的维度表建设还是需要的），业务团队进行开发、分析。

也就是组织架构和分工的差别 —— 传统企业的数据团队可能被当做 IT，整天要求提数，而在新型的互联网/科技团队，数据团队负责提供简单易用的工具，业务部门直接进行数据的使用。

8. 数据湖挑战

从传统集中式的数仓转为开放式的数据湖，并不简单，会碰到许多问题

- 数据发现：如何帮助用户发现数据、了解有哪些数据？
- 数据安全：如果管理数据的权限和安全？因为一些数据是敏感的、或者不应直接开放给所有人的（比如电话号码、地址等）
- 数据管理：多个团队使用数据，如何共享数据成果（比如画像、特征、指标），避免重复开发

这也是目前各大互联网公司都在改进的方向！

9. 湖仓一体

2020 年，大数据 DataBricks 公司首次提出了湖仓一体（Data Lakehouse）概念，希望将数据湖和数据仓库技术合而为一，此概念一出各路云厂商纷纷跟进。

Data Lakehouse（湖仓一体）是新出现的一种数据架构，它同时吸收了数据仓库和数据湖的优势，数据分析师和数据科学家可以在同一个数据存储中对数据进行操作，同时它也能为公司进行数据治理带来更多的便利性。

1）目前数据存储的方案

一直以来，我们都在使用两种数据存储方式来架构数据：

- **数据仓库**：主要存储的是以关系型数据库组织起来的结构化数据。数据通过转换、整合以及清理，并导入到目标表中。在数仓中，数据存储的结构与其定义的 schema 是强匹配的。
- **数据湖**：存储任何类型的数据，包括像图片、文档这样的非结构化数据。数据湖通常更大，其存储成本也更为廉价。存储其中的数据不需要满足特定的 schema，数据湖也不会尝试去将特定的 schema 施行其上。相反的是，数据的拥有者通常会在读取数据的时候解析 schema（schema-on-read），当处理相应的数据时，将转换施加其上。

现在许多的公司往往同时会搭建数仓、数据湖这两种存储架构，一个大的数仓和多个小的数据湖。这样，数据在这两种存储中就会有一定的冗余。

2）Data Lakehouse（湖仓一体）

Data Lakehouse 的出现试图去融合数仓和数据湖这两者之间的差异，通过将数仓构建在数据湖上，使得存储变得更为廉价和弹性，同时 lakehouse 能够有效地提升数据质量，减小数据冗余。在 lakehouse 的构建中，ETL 起了非常重要的作用，它能够将未经规整的数据湖层数据转换成数仓层结构化的数据。

下面详细解释下：

湖仓一体（Data Lakehouse）：

依据 DataBricks 公司对 Lakehouse 的定义：一种结合了数据湖和数据仓库优势的新范式，解决了数据湖的局限性。Lakehouse 使用新的系统设计：直接在用于数据湖的低成本存储上实现与数据仓库中类似的数据结构和数据管理功能。

解释拓展：

湖仓一体，简单理解就是把面向企业的数据仓库技术与数据湖存储技术相结合，为企业提供一个统一的、可共享的数据底座。

避免传统的数据湖、数据仓库之间的数据移动，将原始数据、加工清洗数据、模型化数据，共同存储于一体化的“湖仓”中，既能面向业务实现高并发、精准化、高性能的历史数据、实时数据的查询服务，又能承载分析报表、批处理、数据挖掘等分析型业务。

湖仓一体方案的出现，帮助企业构建起全新的、融合的数据平台。通过对机器学习和 AI 算法的支持，实现数据湖+数据仓库的闭环，提升业务的效率。数据湖和数据仓库的能力充分结合，形成互补，同时对接上层多样化的计算生态。

11. 目前有哪些开源数据湖组件

目前开源的数据湖有江湖人称“数据湖三剑客”的 Hudi、Delta Lake 和 IceBerg。

1）Hudi

Apache Hudi 是一种数据湖的存储格式，在 Hadoop 文件系统之上提供了更新数据和删除数据的能力以及消费变化数据的能力。

Hudi 支持如下两种表类型：

- Copy On Write

使用 Parquet 格式存储数据。Copy On Write 表的更新操作需要通过重写实现。

- Merge On Read

使用列式文件格式（Parquet）和行式文件格式（Avro）混合的方式来存储数据。Merge On Read 使用列式格式存放 Base 数据，同时使用行式格式存放增量数据。最新写入的增量数据存放至行式文件中，根据可配置的策略执行 COMPACTION 操作合并增量数据至列式文件中。

应用场景

- 近实时数据摄取

Hudi 支持插入、更新和删除数据的能力。可以实时摄取消息队列（Kafka）和日志服务 SLS 等日志数据至 Hudi 中，同时也支持实时同步数据库 Binlog 产生的变更数据。

Hudi 优化了数据写入过程中产生的小文件。因此，相比其他传统的文件格式，Hudi 对 HDFS 文件系统更加的友好。

- 近实时数据分析

Hudi 支持多种数据分析引擎，包括 Hive、Spark、Presto 和 Impala。Hudi 作为一种文件格式，不需要依赖额外的服务进程，在使用上也更加的轻量化。

- 增量数据处理

Hudi 支持 Incremental Query 查询类型，可以通过 Spark Streaming 查询给定 COMMIT 后发生变更的数据。Hudi 提供了一种消费 HDFS 变化数据的能力，可以用来优化现有的系统架构。

2) Delta Lake

Delta Lake 是 Spark 计算框架和存储系统之间带有 Schema 信息数据的存储中间层。它给 Spark 带来了三个最主要的功能：
第一，Delta Lake 使得 Spark 能支持数据更新和删除功能；
第二，Delta Lake 使得 Spark 能支持事务；
第三，支持数据版本管理，运行用户查询历史数据快照。

核心特性

- ACID 事务：为数据湖提供 ACID 事务，确保在多个数据管道并发读写数据时，数据能保持完整性。
- 数据版本管理和时间旅行：提供了数据快照，使开发人员能够访问和还原早期版本的数据以进行审核、回滚或重现实验
- 可伸缩的元数据管理：存储表或者文件的元数据信息，并且把元数据也作为数据处理，元数据与数据的对应关系存放在事务日志中；
- 流和批统一处理：Delta 中的表既有批量的，也有流式和 sink 的；
- 数据操作审计：事务日志记录对数据所做的每个更改的详细信息，提供对更改的完整审计跟踪；
- Schema 管理功能：提供自动验证写入数据的 Schema 与表的 Schema 是否兼容的能力，并提供显示增加列和自动更新 Schema 的能力；
- 数据表操作(类似于传统数据库的 SQL)：合并、更新和删除等，提供完全兼容 Spark 的 Java/scala API；

- 统一格式：Delta 中所有的数据和元数据都存储为 Apache Parquet。

3）IceBerg

Iceberg 官网定义：Iceberg 是一个通用的表格式（数据组织格式），它可以适配 Presto，Spark 等引擎提供高性能的读写和元数据管理功能。
数据湖相比传统数仓而言，最明显的便是优秀的 T+0 能力，这个解决了 Hadoop 时代数据分析的顽疾。传统的数据处理流程从数据入库到数据处理通常需要一个较长的环节、涉及许多复杂的逻辑来保证数据的一致性，由于架构的复杂性使得整个流水线具有明显的延迟。
Iceberg 的 ACID 能力可以简化整个流水线的设计，降低整个流水线的延迟。降低数据修正的成本。传统 Hive/Spark 在修正数据时需要将数据读取出来，修改后再写入，有极大的修正成本。Iceberg 所具有的修改、删除能力能够有效地降低开销，提升效率。

1. **ACID 能力，无缝贴合流批一体数据存储最后一块版图**

随着 flink 等技术的不断发展，流批一体生态不断完善，但在流批一体数据存储方面一直是个空白，直到 Iceberg 等数据湖技术的出现，这片空白被慢慢填补。
Iceberg 提供 ACID 事务能力，上游数据写入即可见，不影响当前数据处理任务，这大大简化了 ETL；
Iceberg 提供了 upsert、merge into 能力，可以极大地缩小数据入库延迟；

2. **统一数据存储，无缝衔接计算引擎和数据存储**

Iceberg 提供了基于流式的增量计算模型和基于批处理的全量表计算模型。批处理和流任务可以使用相同的存储模型，数据不再孤立；Iceberg 支持隐藏分区和分区进化，方便业务进行数据分区策略更新。
Iceberg 屏蔽了底层数据存储格式的差异，提供对于 Parquet，ORC 和 Avro 格式的支持。Iceberg 起到了中间桥梁的能力，将上层引擎的能力传导到下层的存储格式。

3. **开放架构设计，开发维护成本相对可控**

Iceberg 的架构和实现并未绑定于某一特定引擎，它实现了通用的数据组织格式，利用此格式可以方便地与不同引擎对接，目前 Iceberg 支持的计算引擎有 Spark、Flink、Presto 以及 Hive。

相比于 Hudi、Delta Lake，Iceberg 的架构实现更为优雅，同时对于数据格式、类型系统有完备的定义和可进化的设计；面向对象存储的优化。Iceberg 在数据组织方式上充分考虑了对象存储的特性，避免耗时的 listing 和 rename 操作，使其在基于对象存储的数据湖架构适配上更有优势。

4. **增量数据读取，实时计算的一把利剑**

Iceberg 支持通过流式方式读取增量数据，支持 Structed Streaming 以及 Flink table Source。

11. 三大数据湖组件对比

1）概览

Delta lake

由于 Apache Spark 在商业化上取得巨大成功，所以由其背后商业公司 Databricks 推出的 Delta lake 也显得格外亮眼。在没有 delta 数据湖之前，Databricks 的客户一般会采用经典的 lambda 架构来构建他们的流批处理场景。

Hudi

Apache Hudi 是由 Uber 的工程师为满足其内部数据分析的需求而设计的数据湖项目，它提供的 fast upsert/delete 以及 compaction 等功能可以说是精准命中广大人民群众的痛点，加上项目各成员积极地社区建设，包括技术细节分享、国内社区推广等等，也在逐步地吸引潜在用户的目光。

Iceberg

Netflix 的数据湖原先是借助 Hive 来构建，但发现 Hive 在设计上的诸多缺陷之后，开始转为自研 Iceberg，并最终演化成 Apache 下一个高度抽象通用的开源数据湖方案。

Apache Iceberg目前看则会显得相对平庸一些，简单说社区关注度暂时比不上 delta，功能也不如 Hudi 丰富，但却是一个野心勃勃的项目，因为它具有高度抽象和非常优雅的设计，为成为一个通用的数据湖方案奠定了良好基础。

本文档来自公众号@五分钟学大数据　盗版必究

2）共同点

三者均为Data Lake的数据存储中间层，其数据管理的功能均是基于一系列的meta文件。Meta文件的角色类似于数据库的catalog\wal，起到schema管理、事务管理和数据管理的功能。与数据库不同的是，这些meta文件是与数据文件一起存放在存储引擎中的，用户可以直接看到。这个做法直接继承了大数据分析中数据对用户可见的传统，但是无形中也增加了数据被不小心破坏的风险。一旦删了meta目录，表就被破坏了，恢复难度很大。

Meta包含有表的schema信息。因此系统可以自己掌握schema的变动，提供schema演化的支持。Meta文件也有transaction log的功能（需要文件系统有原子性和一致性的支持）。所有对表的变更都会生成一份新的meta文件，于是系统就有了ACID和多版本的支持，同时可以提供访问历史的功能。在这些方面，三者是相同的。

3）关于Hudi

Hudi 的设计目标正如其名，Hadoop Upserts Deletes and Incrementals（原为Hadoop Upserts anD Incrementals），强调了其主要支持Upserts、Deletes 和Incremental 数据处理，其主要提供的写入工具是 Spark HudiDataSource API和自身提供的 HoodieDeltaStreamer,均支持三种数据写入方式:UPSERT,INSERT和 BULK_INSERT。其对 Delete 的支持也是通过写入时指定一定的选项支持的，并不支持纯粹的 delete 接口。

在查询方面，Hudi 支持 Hive、Spark、Presto。

在性能方面,Hudi 设计了 HoodieKey ，一个类似于主键的东西。对于查询性能，一般需求是根据查询谓词生成过滤条件下推至datasource。Hudi 这方面没怎么做工作，其性能完全基于引擎自带的谓词下推和 partition prune 功能。

Hudi 的另一大特色是支持 Copy On Write 和 Merge On Read。前者在写入时做数据的 merge，写入性能略差，但是读性能更高一些。后者读的时候做 merge，读性能差,但是写入数据会比较及时,因而后者可以提供近实时的数据分析能力。

最后，Hudi 提供了一个名为run_sync_tool 的脚本同步数据的 schema 到Hive 表。Hudi 还提供了一个命令行工具用于管理 Hudi 表。

4）关于Iceberg

本文档来自公众号@五分钟学大数据　盗版必究

Iceberg 没有类似的 HoodieKey 设计，其不强调主键。没有主键，做 update/delete/merge 等操作就要通过 Join 来实现，而 Join 需要有一个类似 SQL 的执行引擎。

Iceberg 在查询性能方面做了大量的工作。值得一提的是它的 hidden partition 功能。Hidden partition 意思是说，对于用户输入的数据，用户可以选取其中某些列做适当的变换（Transform）形成一个新的列作为 partition 列。这个 partition 列仅仅为了将数据进行分区，并不直接体现在表的 schema 中。

5）关于 Delta

Delta 的定位是流批一体的 Data Lake 存储层，支持 update/delete/merge。由于出自 Databricks，spark 的所有数据写入方式，包括基于 dataframe 的批式、流式，以及 SQL 的 Insert、Insert Overwrite 等都是支持的（开源的 SQL 写暂不支持，EMR 做了支持）。不强调主键，因此其 update/delete/merge 的实现均是基于 spark 的 join 功能。在数据写入方面，Delta 与 Spark 是强绑定的，这一点 Hudi 是不同的：Hudi 的数据写入不绑定 Spark（可以用 Spark，也可以使用 Hudi 自己的写入工具写入）。

在查询方面，开源 Delta 目前支持 Spark 与 Presto，但是，Spark 是不可或缺的，因为 delta log 的处理需要用到 Spark。这意味着如果要用 Presto 查询 Delta，查询时还要跑一个 Spark 作业。更为难受的是，Presto 查询是基于 SymlinkTextInputFormat 。在查询之前，要运行 Spark 作业生成这么个 Symlink 文件。如果表数据是实时更新的，意味着每次在查询之前先要跑一个 SparkSQL，再跑 Presto。为此，EMR 在这方面做了改进可以不必事先启动一个 Spark 任务。

在查询性能方面，开源的 Delta 几乎没有任何优化。

Delta 在数据 merge 方面性能不如 Hudi，在查询方面性能不如 Iceberg，是不是意味着 Delta 一无是处了呢？其实不然。Delta 的一大优点就是与 Spark 的整合能力，尤其是其流批一体的设计，配合 multi-hop 的 data pipeline，可以支持分析、Machine learning、CDC 等多种场景。使用灵活、场景支持完善是它相比 Hudi 和 Iceberg 的最大优点。另外，Delta 号称是 Lambda 架构、Kappa 架构的改进版，无需关心流批，无需关心架构。这一点上 Hudi 和 Iceberg 是力所不及的。

本文档来自公众号@五分钟学大数据　盗版必究

6）总结

三个引擎的初衷场景并不完全相同，Hudi 为了 incremental 的 upserts，Iceberg 定位于高性能的分析与可靠的数据管理，Delta 定位于流批一体的数据处理。这种场景的不同也造成了三者在设计上的差别。尤其是 Hudi，其设计与另外两个相比差别更为明显。因此后面是趋同还筑起各自专长优势壁垒未可知。Delta、Hudi、Iceberg 三个开源项目中，Delta 和 Hudi 跟 Spark 的代码深度绑定，尤其是写入路径。这两个项目设计之初，都基本上把 Spark 作为他们的默认计算引擎了。而Apache Iceberg 的方向非常坚定，宗旨就是要做一个通用化设计的 Table Format。

它完美的解耦了计算引擎和底下的存储系统，便于多样化计算引擎和文件格式，很好的完成了数据湖架构中的 Table Format 这一层的实现，因此也更容易成为 Table Format 层的开源事实标准。

另一方面，Apache Iceberg 也在朝着流批一体的数据存储层发展，manifest 和 snapshot 的设计，有效地隔离不同 transaction 的变更，非常方便批处理和增量计算。并且，Apache Flink 已经是一个流批一体的计算引擎，二者都可以完美匹配，合力打造流批一体的数据湖架构。

Apache Iceberg 这个项目背后的社区资源非常丰富。在国外，Netflix、Apple、Linkedin、Adobe 等公司都有 PB 级别的生产数据运行在 Apache Iceberg 上；在国内，腾讯这样的巨头也有非常庞大的数据跑在 Apache Iceberg 之上，最大的业务每天有几十T 的增量数据写入。

必备 SQL 题

微信搜一搜

五分钟学大数据

本文档来自公众号@五分钟学大数据　盗版必究

1. 第二高的薪水

编写一个 SQL 查询，获取 Employee 表中第二高的薪水（Salary） 。

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

例如上述 Employee 表，SQL 查询应该返回 200 作为第二高的薪水。如果不存在第二高的薪水，那么查询应返回 null。

```
+---------------------+
| SecondHighestSalary |
+---------------------+
| 200                 |
+---------------------+
```

```sql
SELECT
    IFNULL(
      (SELECT DISTINCT Salary
       FROM Employee
       ORDER BY Salary DESC
        LIMIT 1 OFFSET 1),
    NULL) AS SecondHighestSalary
```

2. 分数排名

编写一个 SQL 查询来实现分数排名。
如果两个分数相同，则两个分数排名（Rank）相同。请注意，平分后的下一个名次应该是下一个连续的整数值。换句话说，名次之间不应该有“间隔”。

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
```

```
| 6  | 3.65  |
+----+-------+
```

例如，根据上述给定的 Scores 表，你的查询应该返回（按分数从高到低排列）：

```
+-------+------+
| Score | Rank |
+-------+------+
| 4.00  | 1    |
| 4.00  | 1    |
| 3.85  | 2    |
| 3.65  | 3    |
| 3.65  | 3    |
| 3.50  | 4    |
+-------+------+
```

```
select Score,
dense_rank() over(order by Score desc) `rank`
from Scores
```

3. 连续出现的数字

编写一个 SQL 查询，查找所有至少连续出现三次的数字。

```
+----+-----+
| Id | Num |
+----+-----+
| 1  |  1  |
| 2  |  1  |
| 3  |  1  |
| 4  |  2  |
| 5  |  1  |
| 6  |  2  |
| 7  |  2  |
+----+-----+
```

例如，给定上面的 Logs 表， 1 是唯一连续出现至少三次的数字。

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

```
select distinct Num ConsecutiveNums
from
(
```

```
select
Num,
lead(Num,1,null) over(order by id) n2,
lead(Num,2,null) over(order by id) n3
from Logs
)t1
where Num = n2 and Num = n3
```

4. 员工薪水中位数

Employee 表包含所有员工。Employee 表有三列：员工 Id，公司名和薪水。

```
+-----+------------+--------+
|Id   | Company    | Salary |
+-----+------------+--------+
|1    | A          | 2341   |
|2    | A          | 341    |
|3    | A          | 15     |
|4    | A          | 15314  |
|5    | A          | 451    |
|6    | A          | 513    |
|7    | B          | 15     |
|8    | B          | 13     |
|9    | B          | 1154   |
|10   | B          | 1345   |
|11   | B          | 1221   |
|12   | B          | 234    |
|13   | C          | 2345   |
|14   | C          | 2645   |
|15   | C          | 2645   |
|16   | C          | 2652   |
|17   | C          | 65     |
+-----+------------+--------+
```

请编写 SQL 查询来查找每个公司的薪水中位数。挑战点：你是否可以在不使用任何内置的 SQL 函数的情况下解决此问题。

```
+-----+------------+--------+
|Id   | Company    | Salary |
+-----+------------+--------+
|5    | A          | 451    |
|6    | A          | 513    |
|12   | B          | 234    |
```

```
|9    | B          | 1154   |
|14   | C          | 2645   |
+-----+------------+--------+
```

```
select Id,Company,Salary
from (
select Id,Company,Salary,
ROW_NUMBER() over(partition by Company order by Salary) rk,
count(*) over(partition by Company) cnt
from Employee
)t1
where rk IN (FLOOR((cnt + 1)/2), FLOOR((cnt + 2)/2))
```

5. 游戏玩法分析

Table: Activity

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
```

（player_id，event_date）是此表的主键。
这张表显示了某些游戏的玩家的活动情况。
每一行是一个玩家的记录，他在某一天使用某个设备注销之前登录并玩了很多游戏（可能是 0）。

编写一个 SQL 查询，报告在首次登录的第二天再次登录的玩家的分数，四舍五入到小数点后两位。换句话说，您需要计算从首次登录日期开始至少连续两天登录的玩家的数量，然后除以玩家总数。

查询结果格式如下所示：

```
Activity table:
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-03-02 | 6            |
| 2         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+
```

```
Result table:
+-----------+
| fraction  |
+-----------+
| 0.33      |
+-----------+
```

只有 ID 为 1 的玩家在第一天登录后才重新登录，所以答案是 1/3 = 0.33

```
select round(avg(a.event_date is not null), 2) fraction
from
    (select player_id, min(event_date) as login
    from activity
    group by player_id) p
left join activity a
on p.player_id=a.player_id and datediff(a.event_date, p.login)=1
```

解释：is not null 判断后，有 eventdate 值的返回 1，null 的返回 0，avg 相当于求和后(即符合条件的 id 个数)除以总 id 数即所求比例

6. 2016 年的投资

写一个查询语句，将 2016 年（TIV_2016）所有成功投资的金额加起来，保留 2 位小数。

对于一个投保人，他在 2016 年成功投资的条件是：

1. 他在 2015 年的投保额（TIV_2015）至少跟一个其他投保人在 2015 年的投保额相同。
2. 他所在的城市必须与其他投保人都不同（也就是说维度和经度不能跟其他任何一个投保人完全相同）。

输入格式：表 insurance 格式如下：

Column Name	Type
PID	INTEGER(11)
TIV_2015	NUMERIC(15,2)
TIV_2016	NUMERIC(15,2)
LAT	NUMERIC(5,2)
LON	NUMERIC(5,2)

PID 字段是投保人的投保编号， TIV_2015 是该投保人在 2015 年的总投保金额，TIV_2016 是该投保人在 2016 年的投保金额， LAT 是投保人所在城市的维度，LON 是投保人所在城市的经度。

样例输入

PID	TIV_2015	TIV_2016	LAT	LON
1	10	5	10	10
2	20	20	20	20
3	10	30	20	20
4	10	40	40	40

样例输出

TIV_2016
45.00

解释

就如最后一个投保人，第一个投保人同时满足两个条件：
1. 他在 2015 年的投保金额 TIV_2015 为 '10' ，与第三个和第四个投保人在 2015 年的投保金额相同。
2. 他所在城市的经纬度是独一无二的。

第二个投保人两个条件都不满足。他在 2015 年的投资 TIV_2015 与其他任何投保人都不相同。且他所在城市的经纬度与第三个投保人相同。基于同样的原因，第三个投保人投资失败。

所以返回的结果是第一个投保人和最后一个投保人的 TIV_2016 之和，结果是 45 。

```
select sum(TIV_2016) TIV_2016
from (
    select PID,TIV_2016,cnt,
    count(*) over(partition by loc ) lcnt
    from (
        select PID,TIV_2016,
        count(TIV_2015) over(partition by TIV_2015 ) cnt,
        concat_ws(",",LAT,LON) loc
        from insurance
    )t1
)t2
where lcnt=1 and cnt!=1
```

注意去重顺序 不要先对 TIV_2015 去重 不然 local 去重时会丢失数据

优化 窗口

```
SELECT
    ROUND(SUM(TIV_2016), 2) as TIV_2016
FROM(
    SELECT
        *,
        count(*) over(partition by TIV_2015) as cnt_1,
        count(*) over(partition by LAT, LON) as cnt_2
    FROM
        insurance
) a
WHERE a.cnt_1 > 1 AND a.cnt_2 < 2
```

大厂面试中出现频率最高的 SQL 面试题

1. 求留存率

背景

留 存 率 ：是用户分析的核心指标之一。它也是经典的 AARRR 模型（海盗模型）中就有一个重要节点——留存（Acquisition）。留存率的计算也是用户分析模型的计算基础

常见的留存率有次日留存、三日留存、7 日留存、14 日留存、30 日留存、90 日留存等等，不同产品用户行为的频率是有差别的，留存率的设定也应该视不同产品而定，有些低频的产品用周或月的颗粒度就够了。

留存率计算逻辑：假如某日新增了 100 个用户，第二天登录了 50 个，则次日留存率为 50/100=50%，第三天登录了 30 个，则第二日留存率为 30/100=30%，以此类推，第 7 天登录了 10 个用户，则 7 日留存率就是 10/100=10%。

数据说明

计算留存率只需要 2 个字段：用户 ID (user_id) 和 登录日期 (login_time)

t_user_login：表名

user_id: 用户 id，也可用设备 ID 等

login_time：登录日期时间，例如：2024-05-01 16:03:05

请使用 SQL 计算 2023-10-01 新增的用户的七日留存和 14 日留存。

答案：

本文档来自公众号@五分钟学大数据　盗版必究

```
select
'2023-10-01' as curr_date
,count(if(curr_date_next7=1 and curr_date=1,1,null))/count(if(curr_date=1,1,null))
as retention_7d
,count(if(curr_date_next14=1 and curr_date=1,1,null))/count(if(curr_date=1,1,null))
 as retention_14d
from (
select
user_id
,max(if(substr(login_time,1,10)='2023-10-01',1,0)) curr_date
,max(if(substr(login_time,1,10)=date_add('2023-10-01',7),1,0)) curr_date_next7
,max(if(substr(login_time,1,10)=date_add('2023-10-01',14),1,0)) curr_date_next14
from
t_user_login
group by user_id
) t1
```

2. 求最大连续登陆天数

背景：求最大连续登陆天数的意义在于考察面试者对于 SQL 的使用和逻辑思维能力。这道题目可以评估面试者对于数据分析和处理的能力，以及对于时间序列数据的理解和处理能力。通过解决这个问题，面试官可以了解面试者是否熟悉 SQL 的基本语法和函数，是否能够正确运用 SQL 语句来处理数据，并且是否能够思考和实现解决问题的算法和逻辑。此外，这个问题也可以考察面试者对于连续性数据处理的思路和方法，以及对于复杂问题的解决能力。

数据说明

计算最大连续登陆天数只需要 2 个字段：用户 ID（user_id）和 登录日期（login_time）

t_user_login：表名

user_id：用户 id，也可用设备 ID 等

login_time：登录日期时间，例如：2023-10-01 16:03:05

请使用 SQL 计算每个用户的最大连续登录天数。

答案：

```
select user_id,max(days) maxday  -- 计算每个用户最大的连续登录天数
from
(
select
user_id, groupday, count(*) as days -- 计算每个用户不同时间的连续登录天数
from
(
select
```

```
user_id,login_date,date_sub(login_date,row_number() over (partition by user_id orde
r by login_date)) groupday -- SQL 核心：如果是连续登陆的日期，当前日期减去 row_number 生成
的连续序号，那么 groupday 应该是同一个值
from
(
select
user_id,substr(login_time,1,10) as login_date
from t_user_login
group by user_id,substr(login_time,1,10)
) t1
) tt1
group by user_id, groupday
) ttt1 group by user_id
```

3. 最大连续登陆天数进阶版

最大连续登陆天数进阶版：求最大连续登录天数，间隔 n 天内都可以算做连续登录

数据说明

表名：t_user_login，表中只有每个用户每天首次登陆的数据

```
id          dt
1        2023-05-01
1        2023-05-02
1        2023-05-04
1        2023-05-07
```

请使用 SQL 计算用户的最大连续登陆天数，间隔 1 天内的都算做连续登陆

如上述示例，得到 id=1 的连续登录天数为 4 天，2023-05-01 到 2023-05-04

答案：

```
1、获得每行数据的前一行数据（注意：这里没有重复值，如果有重复值的时候，要去重的）
select id
,dt
,lag(dt,1,'1970-00-00')
over(partition by id order by dt) lagdt
from t_user_login

2、获得和前一行数据的时间差
select id
,dt
,datediff(dt, lagdt) flag
from t1
```

本文档来自公众号@五分钟学大数据　盗版必究

3、开窗，如果时间差大于 2 则为 1，求和

```
select id
,dt
,sum(if(flag>2,1,0)）
over(partition by id order by dt) flag
from t2
```

4、按照求和结果分组，得到最大值

```
select id
,flag
,datediff(max(dt),min(dt))+1 days
from t3
group by id,flag
```

5、取得最大值

```
select id
,max(days) max_days
from t4
group by id
```

总代码：

```
with r1 as (
select id
,dt
,datediff(dt, lagdt) flag
from (
select id
,dt
,lag(dt,1,'1970-00-00')
over(partition by id order by dt) lagdt
from t_user_login
) t1
),
r2 as (
select id
,dt
,sum(if(flag>2,1,0))
over(partition by id order by dt) flag
from r1
),
r3 as (
select id
,flag
,datediff(max(dt),min(dt)) days
```

```
from r2
group by id,flag
)
select id
,max(days)+1 max_days
from r3
group by id
```

4. 求互相关注好友

求互相关注好友，互相关注的用户最后只保留一条
背景：此题重点在后面的互相关注的用户最后只保留一条，怎么对互相关注的用户去重
数据说明
计算互相关注好友只需要 2 个字段：关注者（from_user）和 被关注者（to_user）
user_follow：表名
from_user：关注者的用户 id
to_user：被关注者的用户 id
user_follow 表的数据示例如下：

```
from_user     to_user
  A           B
  B           A
  A           C
  C           B
```

请使用 SQL 计算互相关注的好友，互相关注的用户只保留一条数据。
结果：

```
user0           user1
  A               B
```

答案：

```
select
split(concat_users,'-')[0] as user0
,split(concat_users,'-')[1] as user1
from (
select
if(from_user>to_user,concat(to_user,'-',from_user),concat(from_user,'-',to_user)) a
s concat_users
from
```

```
user_follow
) t1 group by concat_users having count(*)=2
```

Linux

1. Linux 常用高级命令

序号	命令	命令解释
1	top	实时显示系统中各个进程的资源占用状况（CPU、内存和执行时间）
2	jmap -heap	进程号 查看某个进程内存
3	free -m	查看系统内存使用情况
4	ps -ef	查看进程
5	netstat -tunlp | grep 端口号	查看端口占用情况
6	du -sh 路径*	查看路径下的磁盘使用情况，例如：$ du -sh /opt/*
7	df -h	查看磁盘存储情况

2. Shell 常用脚本

（1）集群启动，分发脚本

```
#!/bin/bash
case $1 in
"start")
for i in hadoop102 hadoop103 hadoop104
do
ssh $i "绝对路径" done
;;"stop")
;;
esac
```

（2）数仓层级内部的导入：ods->dwd->dws ->ads

本文档来自公众号@五分钟学大数据　盗版必究

①#!/bin/bash
②定义变量 APP=gmall
③获取时间
传入 按照传入时间
不传 T+1
④sql=" 先按照当前天 写 sql => 遇到时间 $do_date 遇到表 {$APP}. 自定义函数 UDF UDTF {$APP}. "
⑤执行 sql

Shell 中单引号和双引号区别

1） 创建一个 test.sh 文件 vim test.sh
在文件中添加如下内容

```
#!/bin/bash
do_date=$1
echo '$do_date'
echo "$do_date"
echo "'$do_date'"
echo '"$do_date"'
echo `date`
```

2）查看执行结果
test.sh 2022-02-10

```
$do_date
2022-02-10
'2022-02-10'
"$do_date"
2022 年 05 月 02 日 星期四 21:02:08 CST
```

3）总结：

1. 单引号不取变量值
2. 双引号取变量值
3. 反引号`，执行引号中命令
4. 双引号内部嵌套单引号，取出变量值
5. 单引号内部嵌套双引号，不取出变量值

本文档来自公众号@五分钟学大数据　盗版必究

Java

1. 什么是多线程&多线程的优点

多线程是指程序中包含多个执行流，即一个程序中可以同时运行多个不同的线程来执行不同的任务。
优点：可以提高 cpu 的利用率。多线程中，一个线程必须等待的时候，cpu 可以运行其它的线程而不是等待，这样大大提高了程序的效率。

2. 如何创建多线程

Java 3 种常见创建多线程的方式
（1）继承 Thread 类，重 run()方法
（2）实现 Runnable 接口，重写 run()方法
（3）通过创建线程池实现

3. 如何创建线程池

Executors 提供了线程工厂方法用于创建线程池，返回的线程池都实现了 ExecutorServer 接口。

- newSingleThreadExecutor
- newFixedThreadExecutor
- newCachedThreadPool
- newScheduledThreadPool

虽然 Java 自带的工厂方法很便捷，但都有弊端，《阿里巴巴 Java 开发手册》中强制线程池不允许使用以上方法创建，而是通过 ThreadPoolExecutor 的方式，这样处理可以更加明确线程池运行规则，规避资源耗尽的风险。

4. ThreadPoolExecutor 构造函数参数解析

（1）corePoolSize 创建线程池的线程数量
（2）maximumPoolSize 线程池的最大线程数
（3）keepAliveTime 当线程数量大于 corePoolSize ，空闲的线程当空闲时间超过 keepAliveTime 时就会回收

本文档来自公众号@五分钟学大数据　盗版必究

（4）unit ｛ keepAliveTime｝ 时间单位
（5）workQueue 保留任务的队列

5. 列举线程安全的 Map 集合

SynchronizedMap、ConcurrentHashMap

6. StringBuffer 和 StringBuilder 的区别

（1）StringBuffer 中的方法大都采用 synchronized 关键字进行修饰，是线程安全的，效率低。
（2）StringBuilder 是线程不安全的，效率高。

7. ArrayList 和 LinkedList 的区别

（1）ArrayList 基于动态数据实现，LinkedList 基于链表实现，两者都是线程不安全的
（2）ArrayList 基于数组，查询快；linkedList 基于链表，新增和删除更快
（3）LinkedList 不支持高效的随机访问

8. HashMap 和 HashTable 的区别

1）继承的父类不同
HashMap 继承 AbstractMap 类
HashTable 继承 Dictionary 类（已经废弃的类），用比较少
2）是否线程安全
HashMap 是线程不安全的效率高，HashTable 是线程安全的，效率低。
3）key 和 value 是否允许 null 值
Hashtable 中，key 和 value 都不允许出现 null 值。HashMap 中，都可出现 null。

9. HashMap 的底层原理

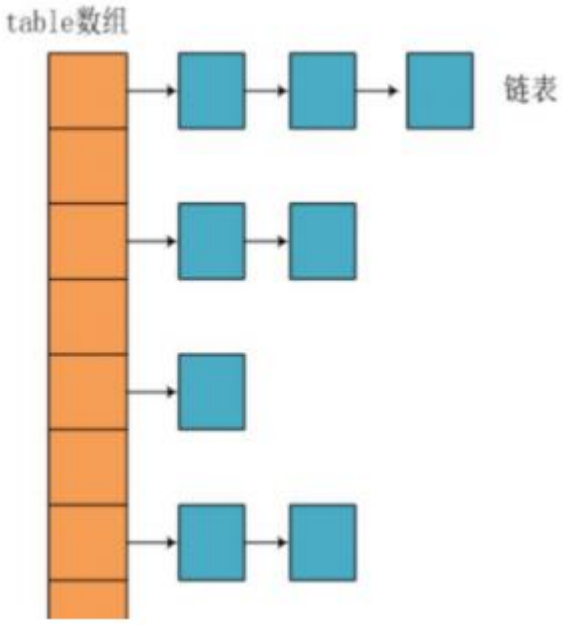
1）HashMap 的实现原理
HashMap 实际上是一个数组和链表的结合体，HashMap 基于 Hash 算法实现的；
（1）当我们向 HashMap 中 Put 元素时，利用 key 的 hashCode 重新计算出当前对象的元素在数组中的下标

（2）写入时，如果出现 Hash 值相同的 key，此时分类，如果 key 相同，则覆盖原始值；如果 key 不同，value 则放入链表中

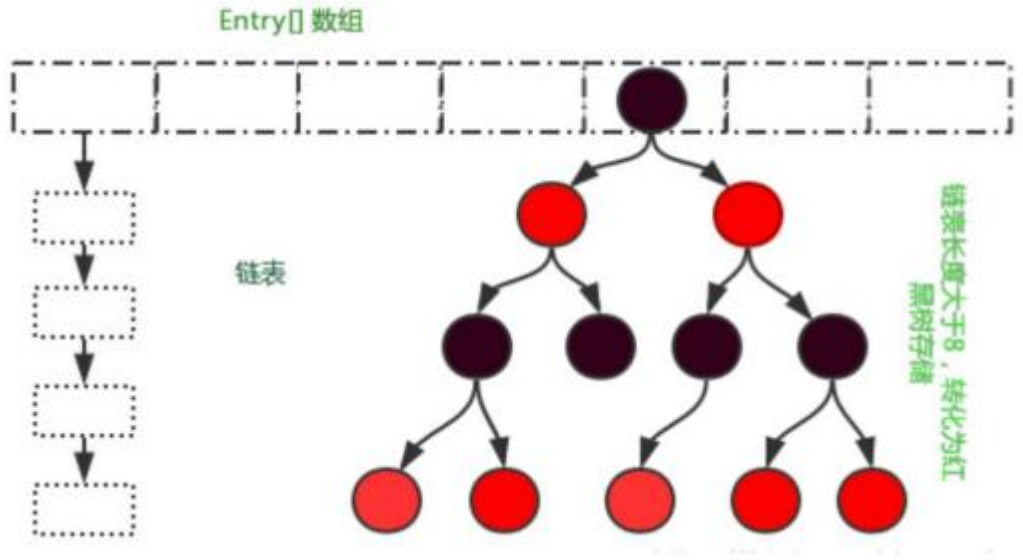
（3）读取时，直接找到 hash 值对应的下标，在进一步判断 key 是否相同，进而找到对应值

2）HashMap 在 JDK1.7 和 JDK1.8 中有哪些区别

JDK1.7：数组 + 链表



JDK1.8：数组+红黑树



10. HashMap 里面放 100 条数据，初始化应该是多少

扩容因子 0.75

100/0.75≈133.3

本文档来自公众号@五分钟学大数据　盗版必究

初始化 134

数据治理

数仓建设真正的难点不在于数仓设计，而在于后续业务发展起来，业务线变的庞大之后的数据治理，包括资产治理、数据质量监控、数据指标体系的建设等。
其实数据治理的范围很广，包含数据本身的管理、数据安全、数据质量、数据成本等。在_DAMA 数据管理知识体系指南_中，数据治理位于数据管理“车轮图”的正中央，是数据架构、数据建模、数据存储、数据安全、数据质量、元数据管理、主数据管理等 10 大数据管理领域的总纲，为各项数据管理活动提供总体指导策略。

1. 数据治理之道是什么

1. 数据治理需要体系建设

为发挥数据价值需要满足三个要素：**合理的平台架构、完善的治理服务、体系化的运营手段**。
根据企业的规模、所属行业、数据量等情况选择合适的平台架构；治理服务需要贯穿数据全生命周期，保证数据在采集、加工、共享、存储、应用整个过程中的完整性、准确性、一致性和实效性；运营手段则应当包括规范的优化、组织的优化、平台的优化以及流程的优化等等方面。

2. 数据治理需要夯实基础

数据治理需要循序渐进，但在建设初期至少需要关注三个方面：**数据规范、数据质量、数据安全**。规范化的模型管理是保障数据可以被治理的前提条件，高质量的数据是数据可用的前提条件，数据的安全管控是数据可以共享交换的前提条件。

3. 数据治理需要 IT 赋能

本文档来自公众号@五分钟学大数据　盗版必究

数据治理不是一堆规范文档的堆砌，而是需要将治理过程中所产生的的规范、流程、标准落地到 IT 平台上，在数据生产过程中通过“以终为始”前向的方式进行数据治理，避免事后稽核带来各种被动和运维成本的增加。

4. 数据治理需要聚焦数据

数据治理的本质是管理数据，因此需要加强元数据管理和主数据管理，从源头治理数据，补齐数据的相关属性和信息，比如：元数据、质量、安全、业务逻辑、血缘等，通过元数据驱动的方式管理数据生产、加工和使用。

5. 数据治理需要建管一体化

数据模型血缘与任务调度的一致性是建管一体化的关键，有助于解决数据管理与数据生产口径不一致的问题，避免出现两张皮的低效管理模式。

2. 数据治理方式有哪些

数据治理的范围非常广，其中最重要的是数据质量治理，而数据质量涉及的范围也很广，贯穿数仓的整个生命周期，从**数据产生->数据接入->数据存储->数据处理->数据输出->数据展示**，每个阶段都需要质量治理，评价维度包括**完整性、规范性、一致性、准确性、唯一性、关联性**等。
在系统建设的各个阶段都应该根据标准进行数据质量检测和规范，及时进行治理，避免事后的清洗工作。
质量检测可参考以下维度：

维度	衡量标准
完整性	业务指定必须的数据是否缺失，不允许为空字符或者空值等。例如，数据源是否完整、维度取值是否完整、数据取值是否完整等
时效性	当需要使用时，数据能否反映当前事实。即数据必须及时，能够满足系统对数据时间的要求。例如处理（获取、整理、清洗、加载等）的及时性
唯一性	在指定的数据集中数据值是否唯一
参照完整性	数据项是否在父表中有定义
依赖一致性	数据项取值是否满足与其他数据项之间的依赖关系
正确性	数据内容和定义是否一致

本文档来自公众号@五分钟学大数据　盗版必究

维度	衡量标准
精确性	数据精度是否达到业务规则要求的位数
技术有效性	数据项是否按已定义的格式标准组织
业务有效性	数据项是否符合已定义的
可信度	根据客户调查或客户主动提供获得
可用性	数据可用的时间和数据需要被访问时间的比例
可访问性	数据是否便于自动化读取

你从哪些方面进行过数据治理：

1. 规范治理

规范是数仓建设的保障。为了避免出现指标重复建设和数据质量差的情况，统一按照最详细、可落地的方法进行规范建设。

(1) 词根

词根是维度和指标管理的基础，划分为普通词根与专有词根，提高词根的易用性和关联性。

- 普通词根：描述事物的最小单元体，如：交易-trade。
- 专有词根：具备约定成俗或行业专属的描述体，如：美元-USD。

(2) 表命名规范

通用规范

- 表名、字段名采用一个下划线分隔词根（示例：clienttype->client_type）。
- 每部分使用小写英文单词，属于通用字段的必须满足通用字段信息的定义。
- 表名、字段名需以字母为开头。
- 表名、字段名最长不超过 64 个英文字符。
- 优先使用词根中已有关键字（数仓标准配置中的词根管理），定期 Review 新增命名的不合理性。
- 在表名自定义部分禁止采用非标准的缩写。

表命名规则

- **表名称** = 类型 + 业务主题 + 子主题 + 表含义 + 存储格式 + 更新频率 +结尾，如下图所示：

(3) **指标命名规范**

结合指标的特性以及词根管理规范，将指标进行结构化处理。

1. 基础指标词根，即所有指标必须包含以下基础词根：

基础指标词根	英文全称	Hive数据类型	MySQL数据类型	长度	精度	词根	样例
数量	count	Bigint	Bigint	10	0	cnt	
金额类	amout	Decimal	Decimal	20	4	amt	
比率/占比	ratio	Decimal	Decimal	10	4	ratio	0.9818
.....	.....	.....	.....			.....	

1. 业务修饰词，用于描述业务场景的词汇，例如 trade-交易。

3. 日期修饰词，用于修饰业务发生的时间区间。

4. 聚合修饰词，对结果进行聚集操作。

5. 基础指标，单一的业务修饰词+基础指标词根构建基础指标 ，例如：交易金额-trade_amt。
6. 派生指标，多修饰词+基础指标词根构建派生指标。派生指标继承基础指标的特性，例如：安装门店数量-install_poi_cnt。

7. 普通指标命名规范，与字段命名规范一致，由词汇转换即可以。

2. 架构治理

(1) **数据分层**

优秀可靠的数仓体系，往往需要清晰的数据分层结构，即要保证数据层的稳定又要屏蔽对下游的影响，并且要避免链路过长，一般的分层架构如下：

(2) **数据流向**

稳定业务按照标准的数据流向进行开发，即 ODS-->DWD-->DWA-->APP。非稳定业务或探索性需求，可以遵循 ODS->DWD->APP 或者 ODS->DWD->DWT->APP 两个模型数据流。在保障了数据链路的合理性之后，又在此基础上确认了模型分层引用原则：

- 正常流向：ODS>DWD->DWT->DWA->APP，当出现 ODS >DWD->DWA->APP 这种关系时，说明主题域未覆盖全。应将 DWD 数据落到 DWT 中，对于使用频度非常低的表允许 DWD->DWA。
- 尽量避免出现 DWA 宽表中使用 DWD 又使用（该 DWD 所归属主题域）DWT 的表。
- 同一主题域内对于 DWT 生成 DWT 的表，原则上要尽量避免，否则会影响 ETL 的效率。
- DWT、DWA 和 APP 中禁止直接使用 ODS 的表， ODS 的表只能被 DWD 引用。
- 禁止出现反向依赖，例如 DWT 的表依赖 DWA 的表。

3. 元数据治理

元数据可分为技术元数据和业务元数据：

技术元数据为开发和管理数据仓库的 IT 人员使用，它描述了与数据仓库开发、管理和维护相关的数据，包括数据源信息、数据转换描述、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等。

常见的技术元数据有：

- 存储元数据：如表、字段、分区等信息。
- 运行元数据：如大数据平台上所有作业运行等信息：类似于 Hive Job 日志，包括作业类型、实例名称、输入输出、 SQL 、运行参数、执行时间，执行引擎等。
- 数据开发平台中数据同步、计算任务、任务调度等信息：包括数据同步的输入输出表和字段，以及同步任务本身的节点信息：计算任务主要有输入输出、任务本身的节点信息 任务调度主要有任务的依赖类型、依赖关系等，以及不同类型调度任务的运行日志等。

- 数据质量和运维相关元数据：如任务监控、运维报警、数据质量、故障等信息，包括任务监控运行日志、告警配置及运行日志、故障信息等。

业务元数据为管理层和业务分析人员服务，从业务角度描述数据，包括商务术语、数据仓库中有什么数据、数据的位置和数据的可用性等，帮助业务人员更好地理解数据仓库中哪些数据是可用的以及如何使用。

- 常见的业务元数据有维度及属性(包括维度编码，字段类型，创建人，创建时间，状态等)、业务过程、指标(包含指标名称,指标编码，业务口径，指标类型，责任人，创建时间，状态，sql 等)，安全等级，计算逻辑等的规范化定义，用于更好地管理和使用数据。数据应用元数据，如数据报表、数据产品等的配置和运行元数据。

元数据不仅定义了数据仓库中数据的模式、来源、抽取和转换规则等，而且是整个数据仓库系统运行的基础，_元数据把数据仓库系统中各个松散的组件联系起来，组成了一个有机的整体_。

元数据治理主要解决三个问题：

1. 通过建立相应的组织、流程和工具，推动业务标准的落地实施，实现指标的规范定义，消除指标认知的歧义；
2. 基于业务现状和未来的演进方式，对业务模型进行抽象，制定清晰的主题、业务过程和分析方向，构建完备的技术元数据，对物理模型进行准确完善的描述，并打通技术元数据与业务元数据的关系，对物理模型进行完备的刻画；
3. 通过元数据建设，为使用数据提效，解决“找数、理解数、评估”难题以及“取数、数据可视化”等难题。

4. 安全治理

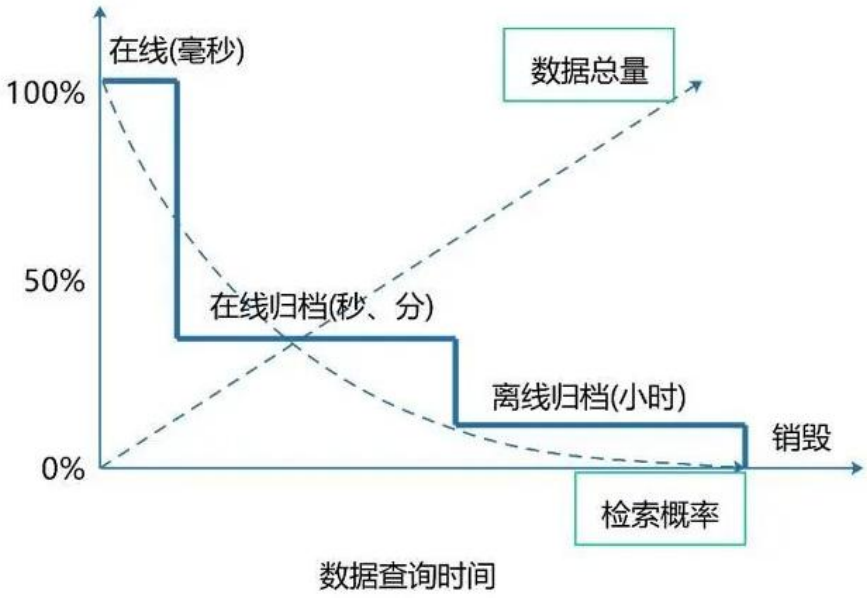
围绕数据安全标准，首先要有数据的分级、分类标准，确保数据在上线前有着准确的密级。第二，针对数据使用方，要有明确的角色授权标准，通过分级分类和角色授权，来保障重要数据拿不走。第三，针对敏感数据，要有隐私管理标准，保障敏感数据的安全存储，即使未授权用户绕过权限管理拿到敏感数据，也要确保其看不懂。第四，通过制定审计标准，为后续的审计提供审计依据，确保数据走不脱。

5. 数据生命周期治理

任何事物都具有一定的生命周期，数据也不例外。从数据的产生、加工、使用乃至消亡都应该有一个科学的管理办法，将极少或者不再使用的数据从系统中剥离

本文档来自公众号@五分钟学大数据　盗版必究

出来，并通过核实的存储设备进行保留，不仅能够提高系统的运行效率，更好的服务客户，还能大幅度减少因为数据长期保存带来的储存成本。数据生命周期一般包含在线阶段、归档阶段（有时还会进一步划分为在线归档阶段和离线归档阶段）、销毁阶段三大阶段，管理内容包括建立合理的数据类别，针对不同类别的数据制定各个阶段的保留时间、存储介质、清理规则和方式、注意事项等。



从上图数据生命周期中各参数间的关系中我们可以了解到，数据生命周期管理可以使得高价值数据的查询效率大幅提升，而且高价格的存储介质的采购量也可以减少很多；但是随着数据的使用程度的下降，数据被逐渐归档，查询时间也慢慢的变长；最后随着数据的使用频率和价值基本没有了之后，就可以逐渐销毁了。

猜你喜欢：

1. 美团数据平台及数仓建设实践，超十万字总结
2. 上百本优质大数据书籍，附必读清单(大数据宝藏)

必备算法

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。

本文档来自公众号@五分钟学大数据　盗版必究

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;
```

本文档来自公众号@五分钟学大数据　盗版必究

```
  }

 }
```

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。

本文档来自公众号@五分钟学大数据　盗版必究

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
```

本文档来自公众号@五分钟学大数据　盗版必究

```
                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);
    }
}
```

4. 约瑟夫环

本文档来自公众号@五分钟学大数据　盗版必究

15 个基督教徒和 15 个非教徒在海上遇险，必须将其中一半的人投入海中，其余的人才能幸免于难，于是 30 个人围成 一圈，从某一个人开始从 1 报数，报到 9 的人就扔进大海，他后面的人继续从 1 开始报数，重复上面的规则，直到剩下 15 个人为止。结果由于上帝的保佑，15 个基督教徒最后都幸免于难，问原来这些人是怎么排列的，哪些位置是基督 教徒，哪些位置是非教徒。

代码实现：

```
public class Josephu {
    private static final int DEAD_NUM = 9;

    public static void main(String[] args) {
        boolean[] persons = new boolean[30];
        for (int i = 0; i < persons.length; i++) {
            persons[i] = true;
        }
        int counter = 0;
        int claimNumber = 0;
        int index = 0;
        while (counter < 15) {
            if (persons[index]) {
                claimNumber++;
                if (claimNumber == DEAD_NUM) {
                    counter++;
                    claimNumber = 0;
                    persons[index] = false;
                }
            }
            index++;
            if (index >= persons.length) {
                index = 0;
            }
        }
        for (boolean p : persons) {
            if (p) {
                System.out.print("基督教");
            } else {
                System.out.print("非基督教");
            }
        }
    }
}
```

本文档来自公众号@五分钟学大数据　盗版必究

5. 回文素数

回文数就是顺着读和倒着读一样的数(例如：11，121，1991…)，回文素数就是既是回文数又是素数(只能被 1 和自 身整除的数)的数。编程找出 11～9999 之间的回文素数。

代码实现：

```
public class PalindromicPrimeNumber {
    public static void main(String[] args) {
        for (int i = 11; i <= 9999; i++) {
            if (isPrime(i) && isPalindromic(i)) {
                System.out.println(i);
            }
        }
    }

    public static boolean isPrime(int n) {
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }

    public static boolean isPalindromic(int n) {
        int temp = n;
        int sum = 0;
        while (temp > 0) {
            sum = sum * 10 + temp % 10;
            temp /= 10;
        }
        return sum == n;
    }
}
```

6. 最大子数组

对于一个有 N 个整数元素的一维数组，找出它的子数组（数组中下标连续的元素组成的数组）之和最大值。

下面给出几个例子（最大子数组用粗体表示）：

1）数组：{ 1, -2, **3, 5**, -3, 2 }，结果是：8

2）数组：{ 0, -2, **3, 5, -1, 2** }，结果是：9

3）数组：{ -9, **-2**, -3, -5, -3 }，结果是：-2

可以使用动态规划的思想求解：

代码实现：

```
public class MaxSum {
    private static int max(int x, int y) {
        return x > y ? x : y;
    }

    public static int maxSum(int[] array) {
        int n = array.length;
        int[] start = new int[n];
        int[] all = new int[n];
        all[n - 1] = start[n - 1] = array[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            start[i] = max(array[i], array[i] + start[i + 1]);
            all[i] = max(start[i], all[i + 1]);
        }
        return all[0];
    }

    public static void main(String[] args) {
        int[] x1 = {1, -2, 3, 5, -3, 2};
        int[] x2 = {0, -2, 3, 5, -1, 2};
        int[] x3 = {-9, -2, -3, -5, -3};
        System.out.println(maxSum(x1)); // 8
        System.out.println(maxSum(x2)); // 9
        System.out.println(maxSum(x3)); //-2
    }
}
```

7. 用递归实现字符串倒转

如，输入：hello ；输出：olleh

代码实现：

```
public class StringReverse {
    public static String reverse(String originStr) {
        if (originStr == null || originStr.length() == 1) {
```

```
            return originStr;
        }
        return reverse(originStr.substring(1)) + originStr.charAt(0);
    }

    public static void main(String[] args) {
        System.out.println(reverse("hello"));
    }
}
```

大数据算法设计题

1. TOP K 算法

有 10 个文件，每个文件 1G，每个文件的每一行存放的都是用户的 query，每个文件的 query 都可能重复。要求你按照 query 的频度排序。

解答：

1）方案 1：

顺序读取 10 个文件，按照 hash(query)%10 的结果将 query 写入到另外 10 个文件（记为）中。这样新生成的文件每个的大小大约也 1G（假设 hash 函数是随机的）。 找一台内存在 2G 左右的机器，依次对用hash_map(query, query_count)来统计每个 query 出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的 query 和对应的 query_cout 输出到文件中。这样得到了 10 个排好序的文件（记为）。 对这 10 个文件进行归并排序（内排序与外排序相结合）。

2）方案 2：

一般 query 的总量是有限的，只是重复的次数比较多而已，可能对于所有的 query，一次性就可以加入到内存了。这样，我们就可以采用 trie 树/hash_map 等直接来统计每个 query 出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

3）方案 3：

与方案 1 类似，但在做完 hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如 MapReduce），最后再进行合并。

2. 不重复的数据

在 2.5 亿个整数中找出不重复的整数，注，内存不足以容纳这 2.5 亿个整数。

解答：

1）方案 1：采用 2-Bitmap（每个数分配 2bit，00 表示不存在，01 表示出现一次，10 表示多次，11 无意义）进行，共需内存 2^32 * 2 bit=1 GB 内存，还可以接受。然后扫描这 2.5 亿个整数，查看 Bitmap 中相对应位，如果是 00 变 01，01 变 10，10 保持不变。所描完事后，查看 bitmap，把对应位是 01 的整数输出即可。

2）方案 2：也可采用与第 1 题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

3. 判断数据是否存在

给 40 亿个不重复的 unsigned int 的整数，没排过序的，然后再给一 个数，如何快速判断这个数是否在那 40 亿个数当中？

1）方案 1：oo，申请 512M 的内存，一个 bit 位代表一个 unsigned int 值。读入 40 亿个数，设置相应的 bit 位，读入要查询的数，查看相应 bit 位是否为 1，为 1 表示存在，为 0 表示不存在。

2）方案 2：这个问题在《编程珠玑》里有很好的描述，大家可以参考下面的思路，探讨一下： 又因为 2^32 为 40 亿多，所以给定一个数可能在，也可能不在其中； 这里我们把 40 亿个数中的每一个用 32 位的二进制来表示 ，假设这 40 亿个数开始放在一个文件中。 然后将这 40 亿个数分成两类: 1.最高位为 0 ； 2.最高位为 1 ；并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20 亿，而另一个>=20 亿（这相当于折半了）； 与要查找的数的最高位

比较并接着进入相应的文件再查找 再然后把这个文件为又分成两类: 1.次最高位为 0 ；2.次最高位为 1 ；并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10 亿，而另一个>=10 亿（这相当于折半了）； 与要查找的数的次最高位比较并接着进入相应的文件再查找。..... 以此类推，就可以找到了，而且时间复杂度为 O(logn)，方案 2 完。

4. 重复最多的数据

有一千万条短信，有重复，以文本文件的形式保存，一行一条，有重复。请用 5 分钟时间，找出重复出现最多的前 10 条。

解答：

1）分析： 常规方法是先排序，在遍历一次，找出重复最多的前 10 条。但是排序的算法复杂度最低为 nlgn。

2）可以设计一个 hash_table, hash_map<string, int>，依次读取一千万条短信，加载到 hash_table 表 中，并且统计重复的次数，与此同时维护一张最多 10 条的短信表。 这样遍历一次就能找出最多的前 10 条，算法复 杂度为 O(n)。

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：