大数据面试题

目录

hadoop 面试题

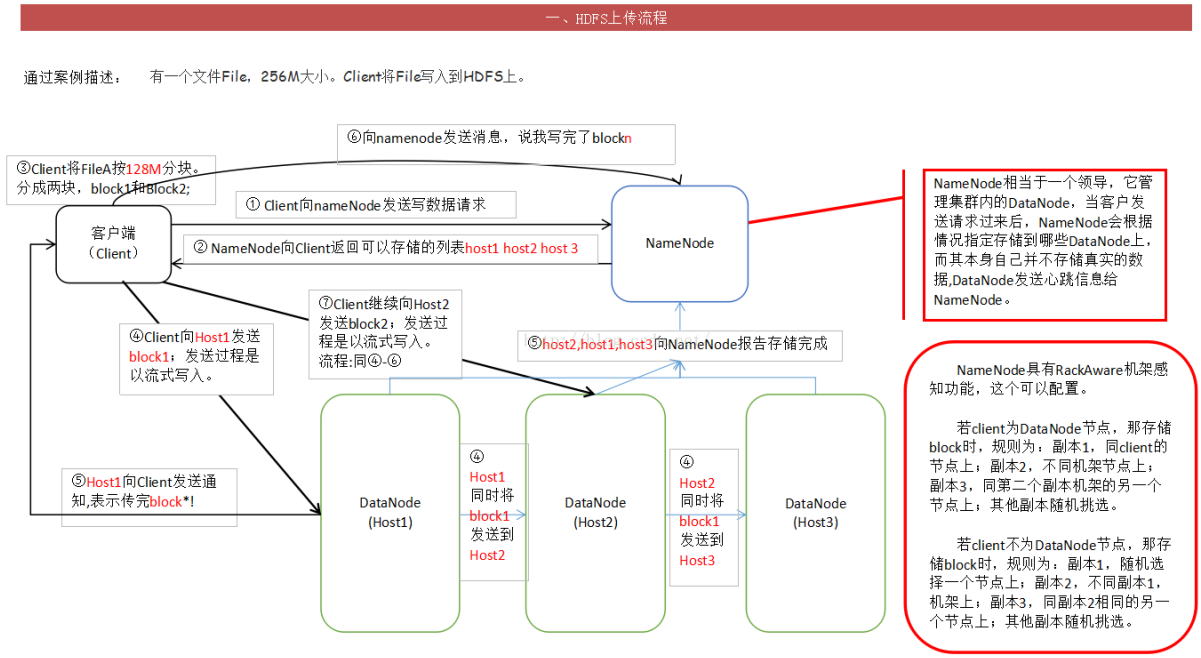
Hive 面试题

Spark 面试题

Hadoop 面试题

1.讲述 HDFS 上传文件和读文件的流程

HDFS 上传流程，举例说明一个 256M 的文件上传过程
(1)由客户端 Client 向 NameNode 节点发出请求;
(2)NameNode 向 Client 返回可以存数据的 DataNode 列表，这里遵循机架感应原则(把副本分别放在不同的机架，甚至不同的数据中心);
(3)客户端首先根据返回的信息先将文件分块(Hadoop2.X 版本每一个 block 为 128M，而之前的版本为 64M);
(4)通过 NameNode 返回的 DataNode 信息，将文件块以写入方式直接发送给 DataNode，同时复制到其他两台机器(默认一份数据，有两个副本);
(5)数据块传送完成以后，dataNode 向 Client 通信，同时向 NameNode 报告;
(6)依照上面(4)到(5)的原理将所有的数据块都上传，结束后向 NameNode 报告表明已经传完所有的数据块。
HDFS 上传流程如下图所示：



HDFS 上传流程

2.HDFS 在上传文件的时候，如果其中一个块突然损坏了怎么办？

其中一个块坏了，只要有其它块存在，会自动检测还原。

3.NameNode 的作用

namenode 总体来说是管理和记录恢复功能。比如管理 datanode，保持心跳，如果超时则排除。对于上传文件都有镜像 images 和 edits,这些可以用来恢复。

4.NameNode 在启动的时候会做哪些操作

NameNode 启动的时候，会加载 fsimage

Fsimage 加载过程完成的操作主要是为了：

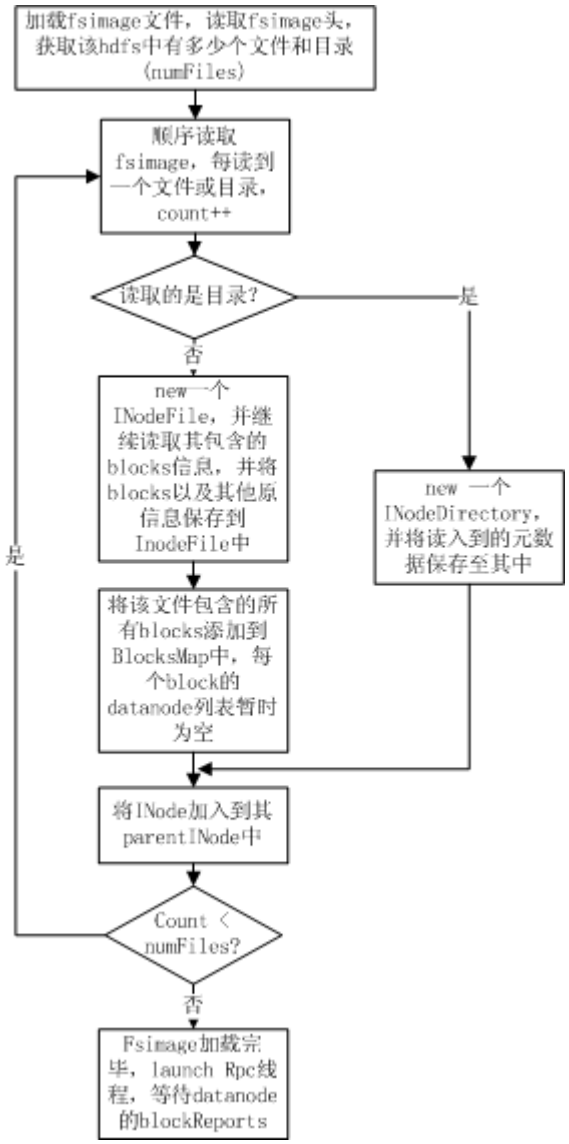
(1)从 fsimage 中读取该 HDFS 中保存的每一个目录和每一个文件

(2)初始化每个目录和文件的元数据信息

(3)根据目录和文件的路径，构造出整个 namespace 在内存中的镜像

(4)如果是文件，则读取出该文件包含的所有 blockid，并插入到 BlocksMap 中。

整个加载流程如下图所示：



Fsimage 加载过程

如上图所示，namenode 在加载 fsimage 过程其实非常简单，就是从 fsimage 中不停的顺序读取文件和目录的元数据信息，并在内存中构建整个 namespace，同时将每个文件对应的 blockid 保存入 BlocksMap 中，此时 BlocksMap 中每个 block 对应的 datanodes 列表暂时为空。当 fsimage 加载完毕后，整个 HDFS 的目录结构在内存中就已经初始化完毕，所缺的就是每个文件对应的 block 对应的 datanode 列表信息。这些信息需要从 datanode 的 blockReport 中获取，所以加载 fsimage 完毕后，namenode 进程进入 rpc 等待状态，等待所有的 datanodes 发送 blockReports。

5.NameNode 的 HA

NameNode 的 HA 一个备用，一个工作，且一个失败后，另一个被激活。他们通过 journal node 来实现共享数据。

6.Hadoop 的作业提交流程

Hadoop2.x Yarn 作业提交（客户端）
http://www.aboutyun.com/forum.php?mod=viewthread&tid=9498
Hadoop2.x Yarn 作业提交（服务端）
http://www.aboutyun.com/forum.php?mod=viewthread&tid=9496

7.Hadoop 怎么分片

HDFS 存储系统中，引入了文件系统的分块概念（block），块是存储的最小单位，HDFS 定义其大小为 64MB。与单磁盘文件系统相似，存储在 HDFS 上的文件均存储为多个块，不同的是，如果某文件大小没有到达 64MB，该文件也不会占据整个块空间。在分布式的 HDFS 集群上，Hadoop 系统 保证一个块存储在一个 datanode 上。
HDFS 的 namenode 只存储整个文件系统的元数据镜像，这个镜像由配置 dfs.name.dir 指定，datanode 则存有文件的 metainfo 和具体的分块，存储路径由 dfs.data.dir 指定。
分析完毕分块，下面讨论一下分片：
hadoop 的作业在提交过程中，需要把具体的输入进行分片。具体的分片细节由 InputSplitFormat 指定。分片的规则为 FileInputFormat.class 中的 getSplits()方法指定：
long splitSize = computeSplitSize(goalSize, minSize, blockSize);
computeSplitSize:
Math.max(minSize, Math.min(goalSize, blockSize));
其中 goalSize 为“InputFile 大小”/“我们在配置文件中定义的 mapred.map.tasks”值，minsize 为 mapred.min.split.size，blockSize 为 64，所以，这个算式为取分片大小不大于 block，并且不小于在 mapred.min.split.size 配置中定义的最小 Size。
当某个分块分成均等的若干分片时，会有最后一个分片大小小于定义的分片大小，则该分片独立成为一个分片。

8.如何减少 Hadoop Map 端到 Reduce 端的数据传输量

减少传输量，可以让 map 处理完，让同台的 reduce 直接处理，理想情况下，没有数据传输。

9.Hadoop 的 Shuffle?

hadoop：map 端保存分片数据，通过网络收集到 reduce 端
Shuffle 产生的意义是什么？

完整地从 map task 端拉取数据到 reduce 端；在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗；减少磁盘 IO 对 task 执行的影响；每个 map task 都有一个内存缓冲区，存储着 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存放到磁盘，当整个 map task 结束后再对磁盘中这个 map task 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 来拉数据。

10.哪些场景才能使用 Combiner 呢？

Combiner 的输出是 Reducer 的输入，Combiner 绝不能改变最终的计算结果。所以从我的想法来看，Combiner 只应该用于那种 Reduce 的输入 key/value 与输出 key/value 类型完全一致，且不影响最终结果的场景。比如累加，最大值等。Combiner 的使用一定得慎重，如果用好，它对 job 执行效率有帮助，反之会影响 reduce 的最终结果。

combiner 最基本是实现本地 key 的聚合，对 map 输出的 key 排序，value 进行迭代。combiner 的目的是减少 map 网络流量。combiner 的对象是对于 map。combiner 具有和 reduce 相似的功能。只不过 combiner 合并对象，是对于一个 map。reduce 合并对象，是对于多个 map。

11.HMaster 的作用

为 region server 分配 region.
负责 region server 的负载均衡。
发现失效的 region server 并重新分配其上的 region.
Gfs 上的垃圾文件回收。
处理 schema 更新请求。

12.如何实现 hadoop 的安全机制

(1)共享 hadoop 集群:

a: 管理人员把开发人员分成了若干个队列，每个队列有一定的资源，每个用户及用户组只能使用某个队列中指定资源。

b: HDFS 上有各种数据，公用的，私有的，加密的。不用的用户可以访问不同的数据。

(2) HDFS 安全机制

client 获取 namenode 的初始访问认证(使用 kerberos)后，会获取一个 delegation token，这个 token 可以作为接下来访问 HDFS 或提交作业的认证。同样，读取 block 也是一样的。

(3) mapreduce 安全机制

所有关于作业的提交或者作业运行状态的追踪均是采用带有 Kerberos 认证的 RPC 实现的。授权用户提交作业时，JobTracker 会为之生成一个 delegation token，该 token 将被作为 job 的一部分存储到 HDFS 上并通过 RPC 分发给各个 TaskTracker，一旦 job 运行结束，该 token 失效。

(4) DistributedCache 是安全的。

DistribuedCache 分别两种，一种是 shared，可以被所有作业共享，而 private 的只能被该用户的作业共享。

(5) RPC 安全机制

在 Hadoop RP 中添加了权限认证授权机制。当用户调用 RPC 时，用户的 login name 会通过 RPC 头部传递给 RPC，之后 RPC 使用 Simple Authentication and Security Layer（SASL）确定一个权限协议（支持 Kerberos 和 DIGEST-MD5 两种），完成 RPC 授权。

13.hadoop 的调度策略的实现，你们使用的是那种策略，为什么。

(1)默认情况下 hadoop 使用的 FIFO, 先进先出的调度策略。按照作业的优先级来处理。

(2)计算能力调度器(Capacity Scheduler) 支持多个队列，每个队列可配置一定的资源量，每个队列采用 FIFO, 为了防止同一个用户的作业独占资源，那么调度器会对同一个用户提交的作业所占资源进行限定，首先按以下策略选择一个合适队列：计算每个队列中正在运行的任务数与其应该分得的计算资源之间的比值，选择一个该比值最小的队列；然后按以下策略选择该队列中一个作业：按照作业优先级和提交时间顺序选择，同时考虑用户资源量限制和内存限制。

(3)公平调度器(Fair Scheduler) 支持多队列多用户，每个队列中的资源量可以配置，同一队列中的作业公平共享队列中所有资源。

(4)异构集群的调度器 LATE

(5)实时作业的调度器 Deadline Scheduler 和 Constraint-based Scheduler

14.数据倾斜怎么处理？

数据倾斜有很多解决方案，本例子简要介绍一种实现方式，假设表 A 和表 B 连接，表 A 数据倾斜，只有一个 key 倾斜，首先对 A 进行采样，统计出最倾斜的 key，将 A 表分隔为 A1 只有倾斜 key， A2 不包含倾斜 key， 然后分别与 表 B 连接。最后将结果合并， union

15.评述 hadoop 运行原理

有 hdfs 负责数据存放，是 Hadoop 的分布式文件存储系统

将大文件分解为多个 Block，每个 Block 保存多个副本。提供容错机制，副本丢失或者宕机时自动恢复。默认每个 Block 保存 3 个副本，64M 为 1 个 Block。由 mapreduce 负责计算，Map（映射）和 Reduce（归约）

16.简答说一下 hadoop 的 map-reduce 编程模型

(1)map task 会从本地文件系统读取数据，转换成 key-value 形式的键值对集合。使用的是 hadoop 内置的数据类型，比如 longwritable、text 等。
(2)将键值对集合输入 mapper 进行业务处理过程，将其转换成需要的 key-value 在输出之后会进行一个 partition 分区操作，默认使用的是 hashpartitioner，可以通过重写 hashpartitioner 的 getpartition 方法来自定义分区规则。
(3)会对 key 进行进行 sort 排序，grouping 分组操作将相同 key 的 value 合并分组输出，在这里可以使用自定义的数据类型，重写 WritableComparator 的 Comparator 方法来自定义排序规则，重写 RawComparator 的 compara 方法来自定义分组规则
(4)进行一个 combiner 归约操作，其实就是一个本地段的 reduce 预处理，以减小后面 shufle 和 reducer 的工作量
reduce task 会通过网络将各个数据收集进行 reduce 处理，最后将数据保存或者显示，结束整个 job。

举例说明：将一副牌的分成四种花色。

17.hadoop 的 TextInputFormat 作用是什么，如何自定义实现

InputFormat 会在 map 操作之前对数据进行两方面的预处理
(1)是 getSplits，返回的是 InputSplit 数组，对数据进行 split 分片，每片交给 map 操作一次
(2)是 getRecordReader，返回的是 RecordReader 对象，对每个 split 分片进行转换为 key-value 键值对格式传递给 map

常用的 InputFormat 是 TextInputFormat，使用的是 LineRecordReader 对每个分片进行键值对的转换，以行偏移量作为键，行内容作为值

自定义类继承 InputFormat 接口，重写 createRecordReader 和 isSplitable 方法 在 createRecordReader 中可以自定义分隔符

18.map-reduce 程序运行的时候会有什么比较常见的问题

比如说作业中大部分都完成了，但是总有几个 reduce 一直在运行。这是因为这几个 reduce 中的处理的数据要远远大于其他的 reduce，可能是因为对键值对任务划分的不均匀造成的数据倾斜。解决的方法可以在分区的时候重新定义分区规则对于 value 数据很多的 key 可以进行拆分、均匀打散等处理，或者是在 map 端的 combiner 中进行数据预处理的操作

19.Hadoop 平台集群配置、环境变量设置？

zookeeper：修改 zoo.cfg 文件，配置 dataDir，和各个 zk 节点的 server 地址端口，tickTime 心跳时间默认是 2000ms，其他超时的时间都是以这个为基础的整数倍，之后再 dataDir 对应

目录下写入 myid 文件和 zoo.cfg 中的 server 相对应。
hadoop：修改
hadoop-env.sh 配置 java 环境变量
core-site.xml 配置 zk 地址，临时目录等
hdfs-site.xml 配置 nn 信息，rpc 和 http 通信地址，nn 自动切换、zk 连接超时时间等
yarn-site.xml 配置 resourcemanager 地址
mapred-site.xml 配置使用 yarn
slaves 配置节点信息
格式化 nn 和 zk。
hbase：修改
hbase-env.sh 配置 java 环境变量和是否使用自带的 zk
hbase-site.xml 配置 hdfs 上数据存放路径，zk 地址和通讯超时时间、master 节点
regionservers 配置各个 region 节点
zoo.cfg 拷贝到 conf 目录下
spark：
安装 Scala
修改 spark-env.sh 配置环境变量和 master 和 worker 节点配置信息
环境变量的设置：直接在/etc/profile 中配置安装的路径即可，或者在当前用户的宿主目录下，配置在.bashrc 文件中，该文件不用 source 重新打开 shell 窗口即可，配置在.bash_profile 的话只对当前用户有效。

20.Hadoop 性能调优？

调优可以通过系统配置、程序编写和作业调度算法来进行。 hdfs 的 block.size 可以调到 128/256（网络很好的情况下，默认为 64）
调优的大头：mapred.map.tasks、mapred.reduce.tasks 设置 mr 任务数（默认都是 1）
mapred.tasktracker.map.tasks.maximum 每台机器上的最大 map 任务数
mapred.tasktracker.reduce.tasks.maximum 每台机器上的最大 reduce 任务数
mapred.reduce.slowstart.completed.maps 配置 reduce 任务在 map 任务完成到百分之几的时候开始进入
这个几个参数要看实际节点的情况进行配置，reduce 任务是在 33%的时候完成 copy，要在这之前完成 map 任务，（map 可以提前完成）
mapred.compress.map.output,mapred.output.compress 配置压缩项，消耗 cpu 提升网络和磁盘 io 合理利用 combiner 。注意重用 writable 对象

21.Hadoop 高并发？

首先肯定要保证集群的高可靠性，在高并发的情况下不会挂掉，支撑不住可以通过横向扩展。 datanode 挂掉了使用 hadoop 脚本重新启动。

22. 请简述 MapReduce 中 combiner、partition 的作用。

有时一个 map 可能会产生大量的输出，combiner 的作用是在 map 端对输出先做一次合并，以减少网络传输到 reducer 的数量。
combine 函数把一个 map 函数产生的<key,value>对（多个 key,value）合并成一个新的<key2,value2>.将新的<key2,value2>作为输入到 reduce 函数中
这个 value2 亦可称之为 values，因为有多个。这个合并的目的是为了减少网络传输。
注意：mapper 的输出为 combiner 的输入，reducer 的输入为 combiner 的输出。

partition：
把 map 任务输出的中间结果按照 key 的范围划分成 R 份(R 是预先定义的 reduce 任务的个数)，划分时通常使用 hash 函数，如：hash(key) mod R

这样可以保证一段范围内的 key，一定会由一个 reduce 任务来处理。

23 简述 hadoop 实现 Join 的几种方法。

1) reduce side join

reduce side join 是一种最简单的 join 方式，其主要思想如下：

在 map 阶段，map 函数同时读取两个文件 File1 和 File2，为了区分两种来源的 key/value 数据对，对每条数据打一个标签（tag）,比如：tag=0 表示来自文件 File1，tag=2 表示来自文件 File2。即：map 阶段的主要任务是对不同文件中的数据打标签。

在 reduce 阶段，reduce 函数获取 key 相同的来自 File1 和 File2 文件的 value list， 然后对于同一个 key，对 File1 和 File2 中的数据进行 join（笛卡尔乘积）。即：reduce 阶段进行实际的连接操作。

2) map side join

之所以存在 reduce side join，是因为在 map 阶段不能获取所有需要的 join 字段，即：同一个 key 对应的字段可能位于不同 map 中。Reduce side join 是非常低效的，因为 shuffle 阶段要进行大量的数据传输。

Map side join 是针对以下场景进行的优化：两个待连接表中，有一个表非常大，而另一个表非常小，以至于小表可以直接存放到内存中。这样，我们可以将小表复制多份，让每个 map task 内存中存在一份（比如存放到 hash table 中），然后只扫描大表：对于大表中的每一条记录 key/value，在 hash table 中查找是否有相同的 key 的记录，如果有，则连接后输出即可。

为了支持文件的复制，Hadoop 提供了一个类 DistributedCache，使用该类的方法如下：

（1）用户使用静态方法 DistributedCache.addCacheFile()指定要复制的文件，它的参数是文件的 URI（如果是 HDFS 上的文件，可以这样：hdfs://namenode:9000/home/XXX/file，其中 9000 是自己配置的 NameNode 端口号）。JobTracker 在作业启动之前会获取这个 URI 列表，并将相应的文件拷贝到各个 TaskTracker 的本地磁盘上。（2）用户使用 DistributedCache.getLocalCacheFiles()方法获取文件目录，并使用标准的文件读写 API 读取相应的文件。

3) SemiJoin

SemiJoin，也叫半连接，是从分布式数据库中借鉴过来的方法。它的产生动机是：对于 reduce side join，跨机器的数据传输量非常大，这成了 join 操作的一个瓶颈，如果能够在 map 端过滤掉不会参加 join 操作的数据，则可以大大节省网络 IO。

实现方法很简单：选取一个小表，假设是 File1，将其参与 join 的 key 抽取出来，保存到文件 File3 中，File3 文件一般很小，可以放到内存中。在 map 阶段，使用 DistributedCache 将 File3 复制到各个 TaskTracker 上，然后将 File2 中不在 File3 中的 key 对应的记录过滤掉，剩下的 reduce 阶段的工作与 reduce side join 相同。

更多关于半连接的介绍，可参考：半连接介绍：
http://wenku.baidu.com/view/ae7442db7f1922791688e877.html

4) reduce side join + BloomFilter

在某些情况下，SemiJoin 抽取出来的小表的 key 集合在内存中仍然存放不下，这时候可以使用 BloomFiler 以节省空间。

BloomFilter 最常见的作用是：判断某个元素是否在一个集合里面。它最重要的两个方法是：add() 和 contains()。最大的特点是不会存在 false negative，即：如果 contains()返回 false，则该元素一定不在集合中，但会存在一定的 true negative，即：如果 contains()返回 true，则该元素可能在集合中。

因而可将小表中的 key 保存到 BloomFilter 中，在 map 阶段过滤大表，可能有一些不在小表中的记录没有过滤掉（但是在小表中的记录一定不会过滤掉），这没关系，只不过增加了少量的网络 IO 而已。

更多关于 BloomFilter 的介绍，可参考：http://blog.csdn.net/jiaomeng/article/details/1495500

24.请简述 hadoop 怎样实现二级排序。

在 Hadoop 中，默认情况下是按照 key 进行排序，如果要按照 value 进行排序怎么办？

有两种方法进行二次排序，分别为：buffer and in memory sort 和 value-to-key conversion。

buffer and in memory sort

主要思想是：在 reduce()函数中，将某个 key 对应的所有 value 保存下来，然后进行排序。 这种方法最大的缺点是：可能会造成 out of memory。

value-to-key conversion

主要思想是：将 key 和部分 value 拼接成一个组合 key（实现 WritableComparable 接口或者调 setSortComparatorClass 函数），这样 reduce 获取的结果便是先按 key 排序，后按 value 排序的结果，需要注意的是，用户需要自己实现 Paritioner，以便只按照 key 进行数据划分。Hadoop 显式的支持二次排序，在 Configuration 类中有个 setGroupingComparatorClass()方法，可用于设置排序 group 的 key 值

Hive 面试题

1.hadoop 中两个大表实现 join 的操作，简单描述。

Hive 中可以通过分区来减少数据量；
还可以通过优化 HQL 语句，比如只查询需要的字段，尽量避免全表、全字段查询；

2.Hive 中存放是什么？

表。存的是和 hdfs 的映射关系，hive 是逻辑上的数据仓库，实际操作的都是 hdfs 上的文件，HQL 就是用 sql 语法来写的 mr 程序。

3.Hive 与关系型数据库的关系？

没有关系，hive 是数据仓库，不能和数据库一样进行实时的 CURD 操作。 是一次写入多次

读取的操作，可以看成是 ETL 工具。

4.讲一下数据库，SQl ，左外连接， 原理，实现？

5.大表和小表 join

Map side join。将小表存入内存中，将小表复制多份，让每个 map task 内存中保留一份(比如存放到 hash table 中)，这样只需要扫描大表。对于大表中的每一条记录 key/value，在 hash table 中查找是否有相同的 key，如果有，则连接后输出即可。

6.数据清洗怎么做的？怎么用 spark 做数据清洗

数据清洗的目的是为了保证数据质量，包括数据的完整性、唯一性、一致性、合法性和权威性。数据清洗的结果是对各种脏数据进行对应的处理方式，从而得到标准的、干净的、连续的数据，提供给数据统计和数据挖掘使用。

解决数据的完整性问题：

(1) 通过其他信息不全；

(2) 通过前后数据不全；

(3) 如果实在无法不全，虽然可惜，但是还是要剔除掉进行统计。但是没必要删除，后续其他分析可能还需要。

解决数据的唯一性问题：

(1) 根据主键进行去除，去除重复数据；

(2) 制定一系列规则，保证根据某种规则下只保存一条数据。

解决数据权威性的问题：

选择最权威的数据作为统计和挖掘。

解决合法性的问题：

设定判定规则，通过特定的规则来判断字段或者值来确定数据是否需要被清洗。

7.Hadoop 中二次排序怎么做？

在 hadoop 中一般都是按照 key 进行排序的，但是有时候还需要按照 value 进行排序。

有两种办法进行二次排序：buffer and int memory sort 和 value-to-key conversion。

Buffer and in memory sort 主要是在 reduce()函数中,将每个 key 对应的 value 值保存下来，进行排序。但是缺点在于可能会出现 out of memory。

Value-to-key conversion 主要思想是将 key 和 value 拼接成一个组合 key，然后进行排序，这样 reduce()函数获取的结果就实现了先按照 key 排序，然后按照 value 进行排序。需要注意的是，用户需要自己实现 paritioner，以便只按照 key 进行数据划分。

8.hadoop 常见的 join 操作？

(1) reduce side join：是最简单的 join 操作，主要是在 reduce 端进行 join 操作；
(2) Map side join：之所以存在 reduce side join，是因为在 map 端不能获得需要连接的全部的字段。Reduce side join 比较低效，因为 shuffle 传输数据需要消耗大量的性能。
(3) Semijoin：半连接，对于 reduce side join，跨机器的数据传输量特别大，成为 join 的一个瓶颈。如果能在 map 端过滤掉不会参加 join 的数据，那么可以大大节省网络 IO。

9.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。

10.分析函数？

row_number() over(partition by regionX order by nameX desc) as tn
1 93;2 90;3 90　排名是连续的，相同的分数会有排名先后，前 100 名只有 100 个
rank() over(partiition by regionX order by nameX desc) as tn
1 93;2 90;2 90;4 89 排名不是连续的，相同的分数是同名次，前 100 名只有 100 个
dense_rank() over()
1 93;2 90;2 90;3 89 排名是连续的，相同的分数是同名次，前 100 名可能多于 100 个

Spark 面试题

1.Spark 的 Shuffle 原理及调优

https://www.zhihu.com/question/27643595

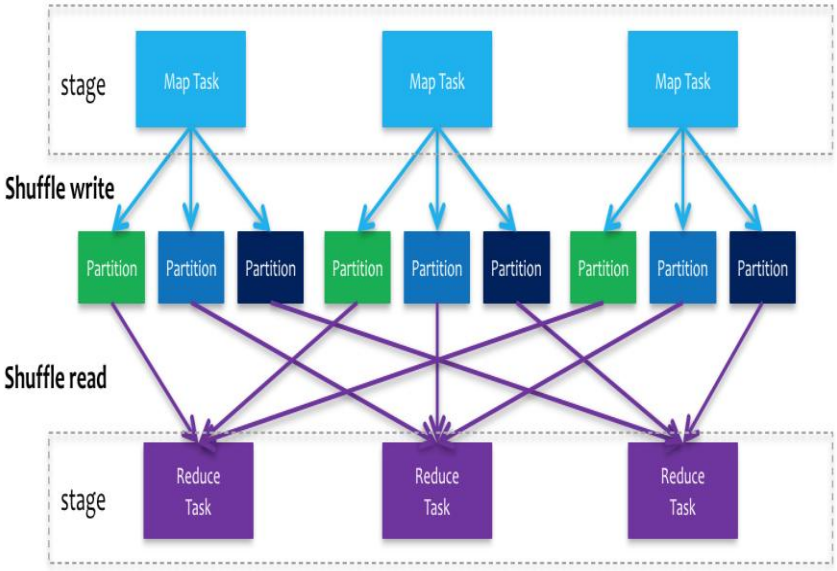
(1)**shuffle 原理**

当使用 reduceByKey、groupByKey、sortByKey、countByKey、join、cogroup 等操作的时候，会发生 shuffle 操作。

Spark 在 DAG 调度阶段将 job 划分成多个 stage，上游 stage 做 map 操作，下游 stage 做 reduce 操作，其本质还是 MR 计算架构。Shuffle 是连接 map 和 reduce 之间的桥梁，它将 map 的输出对应到 reduce 的输入，这期间涉及到序列化和反序列化、跨节点网络 IO 和磁盘读写 IO 等，所以说 shuffle 是整个应用过程特别昂贵的阶段。

与 MapReduce 计算框架一样，spark 的 shuffle 实现大致如下图所示，在 DAG 阶段以 shuffle 为界，划分 stage，上游 stage 做 map task，每个 map task 将计算结果数据分成多份，每一份对应到下游 stage 的每个 partition 中，并将其临时写到磁盘，该过程就叫做 shuffle write；下游 stage 叫做 reduce task，每个 reduce task 通过网络拉取指定分区结果数据，该过程叫做 shuffle read，最后完成 reduce 的业务逻辑。举例：上游 stage 有 100 个 map task，下游有 1000 个 reduce task，那么这 100 个 map task 中每个 map task 都会得到 1000 份数据，而这 1000 个 reduce task 中的每个 reduce task 都会拉取上游 100 个 map task 对应的那份数据，即第一个 reduce task 会拉取所有 map task 结果数据的第一份，以此类推。

在 map 阶段，除了 map 的业务逻辑外，还有 shuffle write 的过程，这个过程涉及序列化、磁盘 IO 等耗时操作；在 reduce 阶段，除了 reduce 的业务逻辑外，还有 shuffle read 过程，这个过程涉及到网络 IO、反序列化等耗时操作。所以整个 shuffle 过程是极其昂贵的。



因为 shuffle 是一个涉及 CPU(序列化反序列化)、网络 IO(跨节点数据传输)

以及磁盘 IO(shuffle 中间结果落地)的操作，所以应当考虑 shuffle 相关的调优，提升 spark 应用程序的性能。

(2)shuffle 调优

(2-1)程序调优；

首先，尽量减少 shuffle 次数；

//两次 shuffle

rdd.map().repartition(1000).reduceByKey(_+_,3000)

//一次 shuffle

Rdd.map().repartition(3000).reduceByKey(_+_)

然后必要时主动 shuffle，通常用于改变并行度，提高后续分布式运行速度；

rdd.repartition(largerNumPartition).map()

最后，使用 treeReduce&treeAggregate 替换 reduce&aggregate。数据量较大时，reduce&aggregate 一次性聚合，shuffle 量太大，而 treeReduce&treeAggregate 是分批聚合，更为保险。

(2-2)参数调优；

spark.shuffle.file.buffer:map task 到 buffer 到磁盘

默认值：32K

参数说明：该参数用于设置 shuffle write task 的 BufferedOutputStream 的 buffer 缓冲大小。将数据写到磁盘文件之前，会先写入 buffer 缓冲中，待缓冲写满之后，才会溢写到磁盘；

调优建议：如果作业可用的内存资源较为充足的话，可以适当**增加这个参数的大小(比如 64k)，从而减少 shuffle write 过程中溢写磁盘文件的次数，也就可以减少磁盘 IO 次数，进而提升性能**。在实践中发现，合理调节该参数，性能会有 1 到 5%的提升。

spark.reducer.maxSizeFlight:reduce task 去磁盘拉取数据

默认值：48m

参数说明：该参数用于设置 shuffle read task 的 buffer 缓冲大小，而这个 buffer 缓冲决定了每次能够拉取多少数据。

调优建议：如果作业可用的内存资源较为充足的话，可以增加这个参数的大小(比如 96M)，从而**减少拉取数据的次数，也就可以减少网络传输的次数**，进而提升性能。在实践中发现，合理调节该参数，性能会有 1 到 5%的提升。

Spark.shuffle.io.maxRetries

默认值：3

参数说明：shuffle read task 从 shuffle write task 所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，时会自动进行重试的。该参数就代表了可以重试的最大次数，如果在指定次数内拉取属于还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些**包含了特别耗时的 shuffle 操作的作业，建议增加重试最大次数(比如 6 次)，可以避免由于 JVM 的 full gc 或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于超大数据量(数十亿到上百亿)的 shuffle 过程，调节该参数可以大幅度提升稳定性。**

Spark.shuffle.io.retryWait

默认值：5s

参数说明：shuffle read task 从 shuffle write task 所在节点拉取属于自己的数据

时，如果拉取失败了每次重试拉取数据的等待时间间隔，默认是 5s；

调优建议：**建议加大时间间隔时长，比如 60s，以增加 shuffle 操作的稳定性。**

spark.shuffle.memoryFraction

默认值：0.2

参数说明：该参数代表了 executor 内存中，分配给 shuffle read task 进行聚合操作的内存比例，默认是 20%；

调优建议：如果内存充足，而且很少使用持久化操作，建议调高和这个比例，**给 shuffle read 的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘**。在实践中发现，合理调节该参数可以将性能提升 10%。

Spark.shuffle.manager

默认值：sort

参数说明：该参数用于设置 shuffleManager 的类型。Spark1.5 以后有三个可选项：hash、sort 和 tungsten-sort。Tungsten-sort 与 sort 类似，但是使用了 tungsten 计划中的堆外内存管理机制，内存使用效率提高。

调优建议：由于 sort shuffleManager 默认会对数据进行排序，因此**如果你的业务逻辑中需要该排序机制的话，则使用默认的 sort ShuffleManager 就可以；但是如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过 bypass 机制或优化的 hash ShuffleManager 来避免排序操作，同时提供较好的磁盘读写性能。**这里要注意的是，tungsten-sort 要慎用，因为之前发现了一些相应的 bug。

Spark.shuffle.sort.bypassMergeThreshold

默认值：200

参数说明：当 shuffleManager 为 sortshuffleManager 时，如果 shuffle read task 的数量小于这个阈值，则 shuffle write 过程中不会进行排序操作，而是直接按照未经优化的 hashShuffleManager 的方式去写数据，但是最后会将每个 task 产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

调优建议：当你使用 sortShuffleManager 时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于 shuffle read task 的数量，那么此时就会自动启用 bupass 机制，map-side 就不会进行排序，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此 shuffle write 性能有待提高。

Spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用 hashShuffleManager，该参数有效。如果设置为 true，那么就会开启 consilidate 机制，会大幅度合并 shuflle write 的输出文件，对于 shuffle read task 数量特别多的情况下，这种方法可以极大地减少磁盘 IO 开销，提升性能。

调优建议：如果的确不需要 sortHashShuffle 的排序机制，那么除了使用 bypass 机制，还可以尝试将 spark.shuffle.manager 参数手动调节为 hash，使用 hashShuffleManager，同时开启 consolidate 机制。在实践中尝试过，发现其性能比开启了 bypass 机制的 sortshuffleManager 要高出 10%到 30%。

2.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。

3.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

4.hadoop 和 spark 的相同点和不同点？

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

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

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

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

5.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。

(1)MEMORY_ONLY:以非序列化的 Java 对象的方式持久化在 JVM 内存中。如果内存无法完全存储 RDD 所有的 partition，那么那些没有持久化的 partition 就会在下一次需要使用它的时候，被重新计算。

(2)MEMORY_AND_DISK:同上，但是当某些 partition 无法存储在内存中时，会持久化到磁盘中。下次需要使用这些 partition 时，需要从磁盘上读取。

(3)MEMORY_ONLY_SER:同 MEMORY_ONLY，但是会使用 Java 序列化方式，将 Java 对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大 CPU 开销。

(4)MEMORY_AND_DSK_SER:同 MEMORY_AND_DSK，但是使用序列化方式持久化 Java 对象。

(5)DISK_ONLY:使用非序列化 Java 对象的方式持久化，完全存储到磁盘上。

(6)MEMORY_ONLY_2/MEMERY_AND_DISK_2：如果是尾部加了 2 的持久化级别，表示会将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可。

6.checkpoint 检查点机制？

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

原因：对于特别复杂的 Spark 应用，会出现某个反复使用的 RDD，即使之前持久化过但由于节点的故障导致数据丢失了，没有容错机制，所以需要重新计算一次数据。

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

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

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

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

http://www.cnblogs.com/dt-zhw/p/5664663.html

7.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。

8.Spark Streaming 和 Storm 有何区别？

Spark Streaming 与 Storm 都可以用于进行实时流计算，但是他们两者的区别是非常大的。

Storm 的优势在于以下两个方面：一方面，Spark Streaming 和 Storm 的计算模型完全不一样。Spark Streaming 是基于 RDD 的，因此需要将一小段时间内的，比如 1 秒内的数据，收集起来，作为一个 RDD，然后再针对这个 batch 的数据进行处理。而 Storm 却可以做到每来一条数据，都可以立即进行处理和计算。因此，Spark Streaming 只能称作准实时的流计算框架，而 Storm 是真正意义上的实时计算框架；另一方面，Storm 支持在分布式流式计算程序运行过程中，可以动态地调整并行度，从而动态提高并发处理能力。而 Spark Streaming 是无法动态调整并行度的；

Spark Streaming 的优势在于：一方面，由于 Spark Streaming 是基于 batch 进行处理的，因此相较于 Storm 基于单条数据进行处理，具有数倍甚至数十倍的吞吐量；另一方面，Spark Streaming 由于身处于 Spark 生态圈内，因此可以和 Spark Core、Spark SQL、Spark MLlib、Spark GraphX 进行无缝整合。流式处理完的数据，可以立即进行各种 map、reduce 转换操作，可以立即使用 sql 进行查询，甚至可以立即使用 machine learning 或者图计算算法进行处理。这种一站式的大数据处理功能和优势，是 Storm 无法匹敌的。

综合上述来看，通常在对实时性要求特别高，而且实时数据量不稳定，比如在白天有高峰期的情况下，可以选择使用 Storm。但是如果是对实时性要求一般，允许 1 秒的准实时处理，而且不要求动态调整并行度的话，选择 Spark Streaming 是更好的选择。

9.RDD 机制？

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。

10.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。

11.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。

12.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。

13.spark 工作机制？

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

14.Spark 工作的一个流程？

提交任务：
用户提交一个任务。 入口是从 sc 开始的。 sc 会去创建一个 taskScheduler。根据不同的提交模式， 会根据相应的 taskchedulerImpl 进行任务调度。同时会去创建 Scheduler 和 DAGScheduler。DAGScheduler 会根据 RDD 的宽依赖或者窄依赖，进行阶段的划分。划分好后放入 taskset 中，交给 taskscheduler 。appclient 会到 master 上注册。首先会去判断数据本地化，尽量选最好的本地化模式去执行。打散 Executor 选择相应的 Executor 去执行。ExecutorRunner 会去创建 CoarseGrainerExecutorBackend 进程。 通过线程池的方式去执行任务。
反向：
Executor 向 SchedulerBackend 反向注册
Spark On Yarn 模式下。 driver 负责计算调度。appmaster 负责资源的申请。

15.spark 核心编程原理？

(1)定义初始的 RDD，即第一个 RDD 是从哪里来，读取数据，包括 hdfs、linux 本地文件、程序中的集合；
(2)定义对 RDD 的计算操作，这在 spark 中称为算子，转换操作和行动操作，包括 map、reduce、flatmap、reducebykey 等，比 mapreduce 提供的 map 和 reduce 强大的太多；
(3)就是循环往复迭代的过程。第一个计算完了以后，数据可能到了新的一批结点上，
变成了一个新的 RDD。然后再次反复，针对新的 RDD 定义计算操作；
(4)获得最终的数据，将保存起来。

16.spark 基本工作原理？

客户端:我们在本地编写了 spark 程序，必须在某台能够连接 spark 集群的机器上提交该 spark 程序；
spark 集群：
将 spark 程序提交到 spark 集群上进行运行。

17.spark 性能优化有哪些？

(1)**使用 Kryo 进行序列化。**

在 spark 中主要有三个地方涉及到序列化：**第一**，在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输；**第二**，将自定义的类型作为 RDD 的泛型数据时(JavaRDD，Student 是自定义类型)，所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现 serializable 借口；**第三**，使用可序列化的持久化策略时，spark 会将 RDD 中的每个 partition 都序列化成为

一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过 Kryo 序列化类库，来优化序列化和反序列化的性能。Spark 默认采用的是 Java 的序列化机制。但是 Spark 同时支持使用 Kryo 序列化库，而且 Kryo 序列化类库的性能比 Java 的序列化类库要高。官方介绍，Kryo 序列化比 Java 序列化性能高出 10 倍。Spark 之所以默认没有使用 Kryo 作为序列化类库，是因为 Kryo 要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说这种方式比较麻烦。

(2)**优化数据结构。**

Java 中有三种类型比较耗费内存：**对象**，每个 Java 对象都有对象头、引用等额外的信息，因此比较占用内存空间；**字符串**，每个字符串内部都有一个字符数组以及长度等额外信息；**集合类型**，比如 HashMap、LinkedList 等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如 Map.Entry。

因此 Spark 官方建议，在 spark 编码实现中，特别对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型(比如 int、long)替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低 GC 频率，提升性能。

使用数组替代集合类型：

举例：有个 List<Integer> list=new ArrayList<Integer>(),可以将其替换为 int[] arr=new int[].这样 array 既比 list 少了额外信息的存储开销，还能使用原始数据类型（int）来存储数据，比 list 中用 Integer 这种包装类型存储数据，要节省内存的多。

使用字符串替代对象：

还比如，通常企业级应用中的做法是，对于 HashMap、List 这种数据，统一用 String 拼接成特殊格式的字符串，比如 Map<Integer,Person> persons = new HashMap<Integer,Person>(),可以优化为特殊的字符串格式：id：name，address|id：name，address。

再比如，**避免使用多层嵌套的对象结构**。比如说，public class Teacher{private List<Student> students =new ArrayList<Student>()}.就是非常不好的例子。因为 Teacher 类的内部又嵌套了大量的小 Student 对象。解决措施是，完全可以使用特殊的字符串来进行数据的存储，比如用 json 字符串来存储数据就是一个好的选择。{"teacherId":1,"teacherName":"leo",students:[{},{}]}

使用原始类型(比如 int、long)替代字符串：

对于有些能够避免的场景，尽量使用 int 代替 String。因为 String 虽然比 ArrayList、HashMap 等数据结构高效多了，占用内存上少多了，但是还是有额外信息的消耗。比如之前用 String 表示 id，那么现在完全可以用数字类型的 int，来进行替代。这里提醒，在 spark 应用中，id 就不要使用常用的 uuid，因为无法转成 int，就用自增的 int 类型的 id 即可。

但是在编码实践中要做到上述原则其实并不容易。因为要同时考虑到代码的可维护性，如果一个代码中，完全没有任何对象抽象，全部是字符串拼接的方式，那么对于后续的代码维护和修改，无疑是一场巨大的灾难。同理，如果所有操作都基于数组实现，而不是用 HashMap、LinkedList 等集合类型，那么对于我们编码的难度以及代码的可维护性，也是一个极大的挑战。因此建议是在保证代码可维护性的前提下，使用占用内存较少的数据结构。

(4)**对多次使用的 RDD 进行持久化并序列化**

原因：Spark 中对于一个 RDD 执行多次算子的默认原理是这样的：每次对一个 RDD 执行一个算子操作时，都会重新从源头出计算一遍，计算出那个 RDD 来，然后再对这个 RDD 执行你的算子操作。这种方式的性能是很差的。

解决办法：因此对于这种情况，建议是对多次使用的 RDD 进行持久化。此时 spark 就会根据你的持久化策略，将 RDD 中的数据保存到内存或者磁盘中。以后每次对这个 RDD 进行算子操作时，都会直接从内存或磁盘中提取持久化的 RDD 数据，然后执行算子，而不会从源头出重新计算一遍这个 RDD。

(5)**垃圾回收调优**

首先使用更高效的数据结构，比如 array 和 string；

其次是在持久化 rdd 时，使用序列化的持久化级别，而且使用 Kryo 序列化类库；这样每个 partition 就只是一个对象(一个字节数组)

然后是监测垃圾回收，可以通过在 spark-submit 脚本中，增加一个配置即可 --conf"spark.executor.extraJavaOptions=-verbose:gc-XX:+PrintGCDetails -XX:+PrintGCTimeStamps"

注意，这里打印出 java 虚拟机的垃圾回收的相关信息，但是输出到了 worker 上的日志，而不是 driver 日志上。还可以通过 sparkUI（4040 端口）来观察每个 stage 的垃圾回收的情况；

然后，**优化 executor 内存比例。对于垃圾回收来说，最重要的是调节 RDD 缓存占用的内存空间，与算子执行时创建对象占用的内存空间的比例。默认是 60%存放缓存 RDD，40%存放 task 执行期间创建的对象。**出现的问题是，task 创建的对象过大，一旦发现 40%内存不够用了，就会频繁触发 GC 操作，从而频繁导致 task 工作线程停止，降低 spark 程序的性能。**解决措施是调优这个比例**，使用 new SparkConf().set("spark.storage.memoryFraction", "0.5")即可，给年轻代更多的空间来存放短时间存活的对象。

最后，如果发现 task 执行期间大量的 Full GC 发生，那么说明年轻代的 Eden 区域给的空间不够大，可以执行以下操作来优化垃圾回收行为：给 Eden 区域分配更大的空间，使用-Xmn 即可，通常建议给 Eden 区域预计大小的 4/3；

如果使用 hdfs 文件，那么很好估计 Eden 区域大小。如果每个 executor 有 4 个 task，然后每个 hdfs 压缩块解压后大小是 3 倍，此外每个 hdfs 块的大小是 64M，那么 Eden 区域的预计代销就是 4*3*64MB，通过-Xmn 参数，将 Eden 区域大小设置为 4*3*64*4/3。

(6)**提高并行度**

减少批处理所消耗时间的常见方式还有提高并行度。首先可以增加接收器数目，当记录太多导致但台机器来不及读入并分发的话，接收器会成为系统瓶颈，这时需要创建多个输入 DStream 来增加接收器数目，然后使用 union 来把数据合并为一个数据源；然后可以将接收到的数据显式的重新分区，如果接收器数目无法在增加，可以通过使用 DStream.repartition 来显式重新分区输入流来重新分配收到的数据；最后可以提高聚合计算的并行度，对于像 reduceByKey()这样的操作，可以在第二个参数中制定并行度。

(7)**广播大数据集**

有时会遇到在算子函数中使用外部变量的场景，建议使用 spark 的广播功能来提升性能。默认情况下，算子函数使用外部变量时会将该变量复制多个副本，通过网络传输到 task 中，此时每个 task 都有一个变量副本。如果变量本身比较大，那么大量的变量副本在网络中传输的性能开销以及在各个节点的 executor 中占用过多的内存导致频繁 GC，都会极大影响性能。所以**建议使用 spark 的广**

播性能，对该变量进行广播。广播的好处在于，会保证每个 executor 的内存中，只驻留一份变量副本，而 executor 中的 task 执行时会共享该 executor 中的那份变量副本。这样的话，可以大大降低变量副本的数量，从而减少网络传输的性能开销，并减少对 executor 内存的占用开销，降低 GC 的频率。

(8)**数据本地化**

数据本地化对于 spark job 性能有着巨大的影响。如果数据以及要计算它的代码是在一起的，那么性能自然会高。但是如果数据和计算它的代码是分开的，那么其中之一必须要另外一方的机器上。通常来说，移动代码到其他节点，会比移动数据到所在节点上去，速度要快的多，因为代码比较小。Spark 也正是基于整个数据本地化的原则来构建 task 调度算法的。数据本地化，指的是数据距离它的代码有多近。基于数据距离代码的距离，有几种数据本地化级别：

(a)PROCESS_LOCAL:数据和计算它的代码在同一个 JVM 进程里面；

(b)NODE_LOCAL:数据和计算它的代码在一个节点上，但是不在一个进程中，比如不在同一个 executor 进程中，或者是数据在 hdfs 文件的 block 中；

(c)NO_PREF:数据从哪里过来，性能都是一样的；

(d)RACK_LOCAL:数据和计算它的代码在一个机架上；

(e)ANY：数据可能在任意地方，比如其他网络环境内，或者其他机架上。

Spark 倾向于使用最好的本地化级别来调度 task，但是这是不可能的。如果没有任何未处理的数据在空闲的 executor 上，那么 Spark 就会放低本地化级别。这时有两个选择：第一，等待，直到 executor 上的 cpu 释放出来，那么就分配 task 过去；第二，立即在任意一个 executor 上启动一个 task。Spark 默认会等待一会，来期望 task 要处理的数据所在的节点上的 executor 空闲出一个 cpu，从而将 task 分配过去。只要超过了时间，那么 spark 就会将 task 分配到其他任意一个空闲的 executor 上。**可以设置参数，spark.locality 系列参数，来调节 spark 等待 task 可以进行数据本地化的时间。**

saprk.locality.wait(3000ms) 、 spark.locality.wait.node 、 spark.locality.wait.process 、 spark.locality.wait.rack。

(9)**尽量使用高性能的算子**

使用 reduceBykey/aggregateBykey 替代 groupByKey。原因是：如果因为业务需要，一定要使用 shuffle 操作，无法用 map 类算子来代替，那么尽量使用可以 map-side 预聚合的算子。所谓的 **map-side 预聚合**，说的是**在每个节点本地对相同的 key 进行一次聚合操作，类似于 MR 的本地 combiner。map-side 预聚合之后，每个节点本地就只会有一条相同的 key**，因为多条相同的 key 都被聚合起来了。其他节点在拉取所有节点上的相同 key 时，就会大大减少需要拉取的数据量，从而也就减少了磁盘 IO 以及网络传输开销。通过来说，在可能的情况下，建议尽量使用 reduceByKey 或者 aggregateByKey 算子来替代 groupBykey 算子。因为 reduceBykey 和 aggregateBykey 算子都会使用用户自定义的函数对每个节点本地相同的 key 进行预聚合。但是 groupbykey 算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

使用 mapPartitions 替代普通 map；

使用 foreachPartitions 替代 foreach；

使用 filter 之后进行 coalesce 操作：通常对一个 RDD 执行 filter 算子过滤掉 RDD 中以后比较多的数据后，建议使用 coalesce 算子，手动减少 RDD 的 partitioning 数量，将 RDD 中的数据压缩到更少的 partition 中去，只要使用更少的 task 即可处理完所有的 partition，在某些场景下对性能有提升。

使用 repartitionAndSortWithinPartitions 替代 repartition 与 sort 类操作：repartitionAndSortWithinPartitions 是 spark 官网推荐的一个算子。官方建议，如果需要在 repartition 重分区之后，还要进行排序，建议直接使用是这个算子。因为该算子可以一边进行重分区的 shuffle 操作，一边进行排序。Shuffle 和 sort 两个操作同时进行，比先 shuffle 再 sort 来说，性能更高。

(10)**shuffle 性能调优**

(11)**批次和窗口大小的设置(针对 spark streaming 中的特殊优化)**

最常见的问题是 Spark Streaming 可以使用的最小批次间隔是多少。寻找最小批次大小的最佳实践是从一个比较大的批次开始，不断使用更小的批次大小。如果 streaming 用户界面中显示的处理时间保持不变，那么就可以进一步减小批次大小。对于窗口操作，计算结果的间隔对于性能也有巨大的影响。

18.updateStateByKey

对整个实时计算的所有时间间隔内产生的相关数据进行统计。

spark streaming 的解决方案是累加器，工作原理是定义一个类似全局的可更新的变量，每个时间窗口内得到的统计值都累加到上个时间窗口得到的值，这样整个累加值就是跨越多个时间间隔。

updateStateByKey 操作可以让我们为每个 key 维护一份 state，并持续不断的更新该 state。

首先，要定义一个 state，可以是任意的数据类型；

其次，要定义 state 更新函数(指定一个函数如何使用之前的 state 和新值来更新 state)。

对于每个 batch，spark 都会为每个之前已经存在的 key 去应用一次 state 更新函数，无论这个 key 在 batch 中是否有新的数据。如果 state 更新函数返回 none，那么 key 对应的 state 就会被删除。

当然对于每个新出现的 key 也会执行 state 更新函数。

注意 updatestateBykey 要求必须开启 checkpoint 机制。

updateStateByKey 返回的都是 DStream 类型。根据 updateFunc 这个函数来更新状态。其中参数：Seq[V]是本次的数据类型，Option[S]是前次计算结果类型，本次计算结果类型也是 Option[S]。计算肯定需要 Partitioner。因为 Hash 高效率且不做排序，默认 Partitioner 是 HashPartitoner。

由于 cogroup 会对所有数据进行扫描，再按 key 进行分组，所以性能上会有问题。特别是随着时间的推移，这样的计算到后面会越算越慢。

所以数据量大的计算、复杂的计算，都不建议使用 updateStateByKey。

mapWithState

19.宽依赖和窄依赖

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

窄依赖：narrow dependency，父 RDD 和子 RDD 的 partition 之间的对应关系是一对一的。

20.spark streaming 中有状态转化操作？

DStream 的有状态转化操作是跨时间区间跟踪数据的操作，即一些先前批次的数据也被用来在新的批次中计算结果。主要包括滑动窗口和 updateStateByKey()，前者以一个时间阶段为滑动窗口进行操作，后者则用来跟踪每个键的状态变化。(例如构建一个代表用户会话的对象)。

有状态转化操作需要在你的 Streaming Context 中打开检查点机制来确保容错性。

滑动窗口：基于窗口的操作会在一个比 Streaming Context 的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。基于窗口的操作需要两个参数，分别为窗口时长以及滑动时长，两者都必须是 Streaming Context 的批次间隔的整数倍。窗口时长控制每次计算最近的多少个批次的数据，滑动步长控制对新的 DStream 进行计算的间隔。

最简单的窗口操作是 window()，它返回的 DStream 中的每个 RDD 会包含多个批次中的户数，可以分别进行其他 transform()操作。在轨迹异常项目中，duration 设置为 15s，窗口函数设置为 kafkadstream.windo(Durations.seconds(600),Durations.seconds(600));

updateStateByKey 转化操作：需要在 DStream 中跨批次维护状态(例如跟踪用户访问网站的会话)。针对这种情况，用于键值对形式的 DStream。给定一个由(键、事件)对构成的 DStream，并传递一个指定根据新的事件更新每个键对应状态的函数。

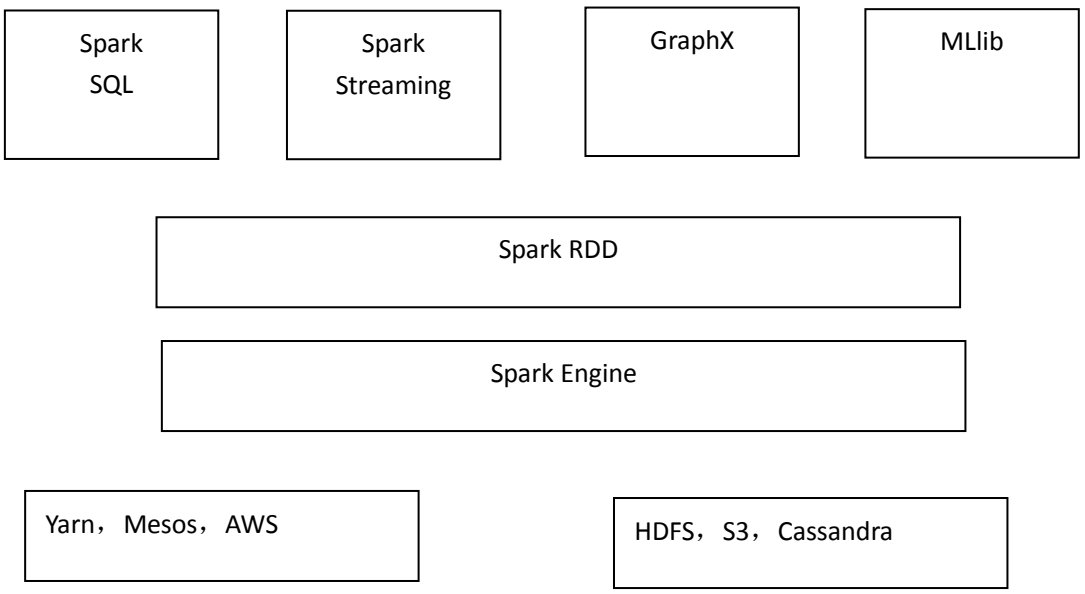
举例：在网络服务器日志中，事件可能是对网站的访问，此时键是用户的 ID。使用 UpdateStateByKey()可以跟踪每个用户最近访问的 10 个页面。这个列表就是“状态”对象，我们会在每个事件到来时更新这个状态。

21.spark 常用的计算框架？

Spark Core 用于离线计算，Spark SQL 用于交互式查询，Spark Streaming 用于实时流式计算，Spark MLlib 用于机器学习，Spark GraphX 用于图计算。

Spark 主要用于大数据的计算，而 hadoop 主要用于大数据的存储(比如 hdfs、hive 和 hbase 等)，以及资源调度 yarn。Spark+hadoop 的组合是未来大数据领域的热门组合。

22.spark 整体架构？



23.Spark 的特点是什么？

(1)速度快：Spark 基于内存进行计算（当然也有部分计算基于磁盘，比如 shuffle）。
(2)容易上手开发：Spark 的基于 RDD 的计算模型，比 Hadoop 的基于 Map-Reduce 的计算模型要更加易于理解，更加易于上手开发，实现各种复杂功能，比如二次排序、topn 等复杂操作时，更加便捷。
(3)超强的通用性：Spark 提供了 Spark RDD、Spark SQL、Spark Streaming、Spark MLlib、Spark GraphX 等技术组件，可以一站式地完成大数据领域的离线批处理、交互式查询、流式计算、机器学习、图计算等常见的任务。
(4)集成 Hadoop：Spark 并不是要成为一个大数据领域的“独裁者”，一个人霸占大数据领域所有的“地盘”，而是与 Hadoop 进行了高度的集成，两者可以完美的配合使用。Hadoop 的 HDFS、Hive、HBase 负责存储，YARN 负责资源调度；Spark 复杂大数据计算。实际上，Hadoop+Spark 的组合，是一种“double win”的组合。
(5)极高的活跃度：Spark 目前是 Apache 基金会的顶级项目，全世界有大量的优秀工程师是 Spark 的 committer。并且世界上很多顶级的 IT 公司都在大规模地使用 Spark。

24.搭建 spark 集群步骤？

(1)安装 spark 包
(2)修改 spark-env.sh
(3)修改 slaves 文件
(4)安装 spark 集群
(5)启动 spark 集群

25.Spark 的三种提交模式是什么？

(1)Spark 内核架构，即 standalone 模式，基于 Spark 自己的 Master-Worker 集群；
(2)基于 Yarn 的 yarn-cluster 模式；
(3)基于 Yarn 的 yarn-client 模式。

如果你要切换到第二种和第三种模式，将之前提交 spark 应用程序的 spark-submit 脚本，加上--master 参数，设置为 yarn-cluster，或 yarn-client 即可。如果没设置就是 standalone 模式。

26..spark 内核架构原理

Master 进程：主要负责资源的调度和分配，还有集群的监控等职责。

Driver 进程：我们编写的 spark 程序就在 Driver 上由 Driver 进程执行。

Worker 进程：主要负责两个，第一个是用自己的内存去存储 RDD 的某个或者某些 partition，第二个是启动其他进程和线程，对 RDD 上的 partition 进行并行的处理和计算。

Executor 和 Task：负责执行对 RDD 的 partition 进行并行的计算，也就是执行我们对 RDD 定义的各种算子。

(1) 将 spark 程序通过 spark-submit(shell)命令提交到结点上执行，其实会通过反射的方式，创建和构造一个 DriverActor 进程出来，通过 Driver 进程执行我们的 Application 应用程序。

(2) 应用程序的第一行一般是先构造 SparkConf，再构造 SparkContext；
(3)SparkContext 在初始化的时候，做的最重要的两件事就是构造 DAGScheduler 和 TaskScheduler；
(4)TaskScheduler 会通过对应的一个后台进程去连接 Master，向 Master 注册 Application；
(5)Master 通知 Worker 启动 executor；
(6)executor 启动之后会自己反向注册到 TaskSchduler，所有的 executor 都反向注册到 Driver 之后，Driver 结束 SparkContext 初始化，然后继续执行自己编写的代码；
(7)每执行一个 action 就会创建一个 job；
(8)DAGScheduler 会根据 Stage 划分算法，将 job 划分为多个 stage，并且为每个 stage 创建 TaskSet(里面有多个 task)；
(9)TaskScheduler 会根据 task 分配算法，把 TaskSet 里面每一个 task 提交到 executor 上执行；
(10)executor 每接收到一个 task，都会用 taskRunner 来封装 task，然后从线程池取出一个线

程，执行这个 task；
(11)taskRunner 将我们编写的代码(算子及函数)进行拷贝，反序列化，然后执行 task；
(12)task 有两种，shuffleMapTask 和 resultTask，只有最后一个 stage 是 resultTask，之前的都是 shuffleMapTask；
(13)所以最后整个 spark 应用程序的执行，就是 job 划分为 stage，然后 stage 分批次作为 taskset 提交到 executor 执行，每个 task 针对 RDD 的一个 partition 执行我们定义的算子和函数，以此类推，直到所有的操作执行完止。

27.Spark yarn-cluster 架构？

Yarn-cluster 用于生产环境，优点在于 driver 运行在 NM，没有网卡流量激增的问题。缺点在于调试不方便，本地用 spark-submit 提交后，看不到 log，只能通过 yarm application-logs application_id 这种命令来查看，很麻烦。
(1)将 spark 程序通过 spark-submit 命令提交，会发送请求到 RM(相当于 Master)，请求启动 AM；
(2)在 yarn 集群上，RM 会分配一个 container，在某个 NM 上启动 AM；
(3)在 NM 上会启动 AM(相当于 Driver)，AM 会找 RM 请求 container，启动 executor；
(4)RM 会分配一批 container 用于启动 executor；
(5)AM 会连接其他 NM(相当于 worker)，来启动 executor；
(6)executor 启动后，会反向注册到 AM。

28.Spark yarn-client 架构？

Yarn-client 用于测试，因为 driver 运行在本地客户端，负责调度 application，会与 yarn 集群产生大量的网络通信，从而导致网卡流量激增，可能会被公司的 SA 警告。好处在于，直接执行时本地可以看到所有的 log，方便调试。
(1)将 spark 程序通过 spark-submit 命令提交，会发送请求到 RM，请求启动 AM；
(2)在 yarn 集群上，RM 会分配一个 container 在某个 NM 上启动 application；
(3)在 NM 上会启动 application master，但是这里的 AM 其实只是一个 ExecutorLauncher，功能很有限，只会去申请资源。AM 会找 RM 申请 container，启动 executor；
(4)RM 会分配一批 container 用于启动 executor；
(5)AM 会连接其他 NM(相当于 worker)，用 container 的资源来启动 executor；
(6)executor 启动后，会反向注册到本地的 Driver 进程。通过本地的 Driver 去执行 DAGsheduler 和 Taskscheduler 等资源调度。

和 Spark yarn-cluster 的区别在于，cluster 模式会在某一个 NM 上启动 AM 作为 Driver。

29.SparkContext 初始化原理？

(1) TaskScheduler 如何注册 application，executor 如何反向注册到 TaskScheduler；
(2) DAGScheduler；
(3) SparkUI。

30.Spark 主备切换机制原理剖析？

Master 实际上可以配置两个，Spark 原声的 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 支持故障恢复的方式？

主要包括两种方式：一种是通过血缘关系 lineage，当发生故障的时候通过血缘关系，再执行一遍来一层一层恢复数据；另一种方式是通过 checkpoint()机制，将数据存储到持久化存储中来恢复数据。

32.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。

(7)MR:对于迭代式计算处理较差；

Spark:Spark 将中间数据存放在内存中，提高迭代式计算性能。

总结：Spark 替代 hadoop，其实应该是 spark 替代 mapreduce 计算模型，因为 spark 本身并不提供存储，所以现在一般比较常用的大数据架构是基于 spark+hadoop 这样的模型。

33.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；

http://blog.csdn.net/u012102306/article/details/51322209

Spark streaming　空 RDD 判断和处理？http://www.aboutyun.com/thread-19303-1-1.html

34.spark 实现高可用性：High Availability

如果有些数据丢失，或者节点挂掉；那么不能让你的实时计算程序挂了；必须做一些数据上的冗余副本，保证你的实时计算程序可以 7 * 24 小时的运转。

(1).updateStateByKey、window 等有状态的操作，自动进行 checkpoint，必须设置 checkpoint 目录：容错的文件系统的目录，比如说，常用的是 HDFS

SparkStreaming.checkpoint("hdfs://192.168.1.105:9090/checkpoint")

设置完这个基本的 checkpoint 目录之后，有些会自动进行 checkpoint 操作的 DStream，就实现了 HA 高可用性；checkpoint，相当于是会把数据保留一份在容错的文件系统中，一旦内存中的数据丢失掉；那么就可以直接从文件系统中读取数据；不需要重新进行计算

(2).Driver 高可用性

第一次在创建和启动 StreamingContext 的时候，那么将持续不断地将实时计算程序的元数据（比如说，有些 dstream 或者 job 执行到了哪个步骤），如果后面，不幸，因为某些原因导致 driver 节点挂掉了；那么可以让 spark 集群帮助我们自动重启 driver，然后继续运行实时计算程序，并且是接着之前的作业继续执行；没有中断，没有数据丢失

第一次在创建和启动 StreamingContext 的时候，将元数据写入容错的文件系统（比如 hdfs）；spark-submit 脚本中加一些参数；保证在 driver 挂掉之后，spark 集群可以自己将 driver 重新启动起来；而且 driver 在启动的时候，不会重新创建一个 streaming context，而是从容错文件系统（比如 hdfs）中读取之前的元数据信息，包括 job 的执行进度，继续接着之前的进度，继续执行。

使用这种机制，就必须使用 cluster 模式提交，确保 driver 运行在某个 worker 上面；但是这种模式不方便我们调试程序，一会儿还要最终测试整个程序的运行，打印不出 log；我们这里仅仅是用我们的代码给大家示范一下：

```
JavaStreamingContextFactory contextFactory = new JavaStreamingContextFactory() {
  @Override
  public JavaStreamingContext create() {
    JavaStreamingContext jssc = new JavaStreamingContext(...);
    JavaDStream<String> lines = jssc.socketTextStream(...);
    jssc.checkpoint(checkpointDirectory);
    return jssc;
  }
};

JavaStreamingContext context = JavaStreamingContext.getOrCreate(checkpointDirectory,
contextFactory);
context.start();
context.awaitTermination();

spark-submit
--deploy-mode cluster
--supervise
```

(3).实现 RDD 高可用性：启动 WAL 预写日志机制

spark streaming，从原理上来说，是通过 receiver 来进行数据接收的；接收到的数据，会被划分成一个一个的 block；block 会被组合成一个 batch；针对一个 batch，会创建一个 rdd；启动一个 job 来执行我们定义的算子操作。

receiver 主要接收到数据，那么就会立即将数据写入一份到容错文件系统（比如 hdfs）上的 checkpoint 目录中的，一份磁盘文件中去；作为数据的冗余副本。无论你的程序怎么挂掉，或者是数据丢失，那么数据都不肯能会永久性的丢失；因为肯定有副本。

WAL（Write-Ahead Log）预写日志机制

spark.streaming.receiver.writeAheadLog.enable true

35.spark 实际工作中，是怎么来根据任务量，判定需要多少资源的？

```
/mydata/spark-1.6.3-bin-spark-1.6.3-2.11-withhive/bin/spark-submit \
--master yarn \
--deploy-mode client \
--num-executors 1 \
--executor-memory 7G \
--executor-cores 6 \
--conf spark.ui.port=5052 \
--conf spark.yarn.executor.memoryOverhead=1024 \
--conf spark.storage.memoryFraction=0.2 \
--class UserAnalytics \
/mydata/dapeng/comecarsparkstreamingproject.jar
```

--num-executors 表示启动多少个 executor 来运行该作业
--executor-memory 表示每一个 executor 进程允许使用的内存空间
--executor-cores 在同一个 executor 里，最多允许多少个 task 可同时并发运行
--executor.memoryOverhead 用于存储已被加载的类信息、常量、静态变量等数据
--shuffle.memoryFraction 用于 shuffle 阶段缓存拉取到的数据所使用的内存空间
--storage.memoryFraction 用于 Java 堆栈的空间

```
/mydata/spark-1.6.3-bin-spark-1.6.3-2.11-withhive/bin/spark-submit \
--master yarn \
--deploy-mode client \
--num-executors 6 \
--executor-memory 7G \
--executor-cores 5 \
--conf spark.ui.port=5051 \
--conf spark.yarn.executor.memoryOverhead=1024 \
--conf spark.memory.storageFraction=0.2 \
--conf spark.executor.extraJavaOptions=-XX:+UseG1GC \
```

配置了 6 个 executor，每个 executor 有 5 个 core，每个 executor 有 7G 内存

36.Spark 提交的 job 的工作流程？

用户提交一个任务。 入口是从 sc 开始的。 sc 会去创建一个 taskScheduler。根据不同的提交模式， 会根据相应的 taskchedulerImpl 进行任务调度。
同时会去创建 Scheduler 和 DAGScheduler。DAGScheduler 会根据 RDD 的宽依赖或者窄依赖，进行阶段的划分。划分好后放入 taskset 中，交给 taskscheduler 。
appclient 会到 master 上注册。首先会去判断数据本地化，尽量选最好的本地化模式去执行。打散 Executor 选择相应的 Executor 去执行。ExecutorRunner 会去创建 CoarseGrainerExecutorBackend 进程。 通过线程池的方式去执行任务。

反向：
Executor 向 SchedulerBackend 反向注册

Spark On Yarn 模式下。 driver 负责计算调度。appmaster 负责资源的申请。

海量数据

海量日志数据 top1 问题

海量日志中，提取出某日访问百度次数最多的那个 IP？
首先是这一天，并且是访问百度的日志中的 IP 取出来，逐个写入到一个大文件中。注意到

IP 是 32 位的，最多有个 2^32 个 IP。同样可以采用映射的方法， 比如模 1000，把整个大文件映射为 1000 个小文件，再找出每个小文中出现频率最大的 IP（可以采用 hash_map 进行频率统计，然后再找出频率最大 的几个）及相应的频率。然后再在这 1000 个最大的 IP 中，找出那个频率最大的 IP，即为所求。

或者如下阐述（雪域之鹰）：

算法思想：分而治之+Hash

（1）.IP 地址最多有 2^32=4G 种取值情况，所以不能完全加载到内存中处理；

（2）.可以考虑采用“分而治之”的思想，按照 IP 地址的 Hash(IP)%1024 值，把海量 IP 日志分别存储到 1024 个小文件中。这样，每个小文件最多包含 4MB 个 IP 地址；

（3）.对于每一个小文件，可以构建一个 IP 为 key，出现次数为 value 的 Hash map，同时记录当前出现次数最多的那个 IP 地址；

（4）.可以得到 1024 个小文件中的出现次数最多的 IP，再依据常规的排序算法得到总体上出现次数最多的 IP；

有限内存的 topN 统计

有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 字节，内存限制大小是 1M。返回频数最高的 100 个词。

方案：顺序读文件中，对于每个词 x，取 hash(x)%5000，然后按照该值存到 5000 个小文件（记为 x0,x1,…x4999）中。这样每个文件大概是 200k 左右。

如果其中的有的文件超过了 1M 大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过 1M。

对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用 trie 树/hash_map 等），并取出出现频率最大的 100 个词（可以用含 100 个结 点的最小堆），并把 100 个词及相

应

的频率存入文件，这样又得到了 5000 个文件。下一步就是把这 5000 个文件进行归并（类

似与归并排序）的过程了。

搜索引擎热点词汇 topN 统计

问题：
搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。 假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

答案：
典型的 Top K 算法，还是在这篇文章里头有所阐述，详情请参见：十一、从头到尾彻底解析 Hash 表算法。文中，给出的最终算法是：

第一步、先对这批海量数据预处理，在 O（N）的时间内用 Hash 表完成统计（之前写成了排序，特此订正。July、2011.04.27）；

第二步、借助堆这个数据结构，找出 Top K，时间复杂度为 N‘logK。
即，借助堆结构，我们可以在 log 量级的时间内查找和调整/移动。因此，维护一个 K(该题目中是 10)大小的小根堆，然后遍历 300 万的 Query，分别 和根元素进行对比所以，我们

最终的时间复杂度是：O（N） + N’*O（logK），（N 为 1000 万，N’为 300 万）。ok，

更多，详情，请参考原文。

或者：采用 trie 树，关键字域存该查询串出现的次数，没有出现为 0。最后用 10 个元素的

最小推来对出现频率进行排序。

多文件排序

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。

还是典型的 TOP K 算法，解决方案如下：

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）。

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。

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。