Interviews Colleciton

# Hadoop

## 简述 Mapreduce 架构组成？

### 概念：

一个基于海量数据处理的分布式离线计算框架

### 核心思想：

对海量的数据集 分而治之（map） 归约处理（reduce）

Mapreduce 计算一个 job 的时候 首先讲这个job 拆分成若干个map task 然后将这些task 分配到不同的机器节点上去执行

map 函数处理数据 当 map 函数完成后 会生成一些中间文件 这些中间文件会作为reduce task 的输入数据 reduce 主要将前面不同机器节点计算 的 map输出 汇总到一起进行处理并输出

### 特点

1. 易于编程

相比传统的分布式城市 不需要关注一些偏底层的操作比如 数据的切割 数据的传输 节点之间的通信等 只需要关注于自己的应用程序的逻辑的实现

简化了程序设计 提高了你的开发效率

2） 良好的扩展性

假设数据量或者数据规模越来越大的时候 现有的集群就无法支撑或者满足现有的计算能力这种情况下管理员可以通过添加计算来达到线性扩展集群能力（HDFS）

3） 并行处理

4） 较高的容错性

### 架构

和Hdfs 一样 Mapreduce 采用master/slave 架构

1） Client 客户端

每一个 Job 都会在用户端 通过Client 类将应用程序以及相关的配置的参数Configuration 打包成Jar包文件存储在HDFS 并把路径提交到 JobTracker 的master 节点服务，然后由 master 去创建每一个 Task（即Map Task 和Reduce Task） 将这些任务 分发到各个 TaskTracker 服务中去执行

2） JobTracker

JboTracker 负责资源监控与作业调度 ,JobTracker 监控所有 TaskTracker 与 Job 的健康情况， 会跟踪任务的执行进度 资源使用情况 并将这些信息告诉 Task Scheduler ，Task Scheduler会在资源出现空闲的时候 选择合适的任务使用这些资源

3） TaskTracker

会周期性地会通过Heartbeat 将本节点上的资源的使用情况和任务的运行进度 回报给JobTracker 同时接收JobTracker 发送过的来的命令并去执行相应的 操作（如“启动新任务 杀死任务等等”）Task 使用“slot”（计算资源（cpu 内存等）） 等量划分本节点上的资源量（类似于Yarn 中的 Container）

一个Task 获取到 一个 slot 后才有机会运行

4） Task

Task 分为 Map Task 和 Reduce Task 均由 TaskTracker 启动

Hdfs 以固定大小的block 为基本存储单位

而对于 Mapreduce 处理的基本单位是split

## mapreduec的任务流程是什么？

input ---> split--->(key,value) --->map()--->shuffle--->(key,(value1,valu2,valuen))---reduce()--->output

（1） 对输入进行 split 分隔 生成（key,value）

（2） map 函数处理上一步生成（key,value）生成 新的（key,value）

（3） shuffle 对 map 函数输出的新的（key,value） 进行分区、排序合并、拷贝 转换为 （key,（value1,valu2,valuen） 传递给 reduce 端

（4）reduce对于传递过来的接受到（key,（value1,valu2,valuen））进行处理产生最终的 （key,value） 结果 输出到 Hdfs

## hadoop的作业运行流程？

（1）Client 向JobTracker 提交作业 申请JobID 将作业资源复制到Hdfs上

（2）初始化作业 根据输入创建map task（reduce task 的个数由相关配置控制）

（3） Jobt Tacker 分发 task 给 Task Tracker

（4） 开始执行任务

（5） 更新任务执行进度和状态

（6） 完成任务 清理一些状态数据和临时的输出文件

## 有不同的数据库，比如 oracle、mysql 等，它们存储着不同格式的数据，如果想对不同来源的数据进行清洗和分析，请写出一个设计方案

RDBMS--->Hdfs

方案一、Sqoop

方案二、Spark SQL

val dataFrame = sqlContext.read.jdbc(url,tableName,properties)

DataFrame 是对RDD进行的上层的封装 RDD 提供了丰富的算子操作

比传统 Sqoop 使用起来更加方便 更加灵活

## hadoop hdfs ----> mapreduce ---> hive ---> spark

## 简要描述如何安装配置Hadoop集群，简单描述即可，无需列出完整步骤，能列出完整步骤更好？

（1） 节点规划

通常我会根据提供的机器资源情况 做节点划分

HDFS HA:

machine1 machine2 machine3

Namenode(Active) NameNode(Standy)??

ZKFC ZKFC

Jouralnode Jouralnode Jouralnode

DataNode DataNode DataNode

zookeeper zookeeper zookeeper

（去修改相关的配置文件

slaves core-site.xml hdfs-site.xml yarn-site.xml mapred-site.xml）

（2）集群时间同步

（3）ssh 无密钥登录

## 请列出正常工作的 Hadoop 集群中，Hadoop 分别需要启动哪些进程，它们的作用分别是什么，尽可能写的全面些？

Hdfs:

Namenode

Datanode

Secondarynamenoe

Mapreduce:

JobTracker

TaskTracker

Yarn:

ResourceManager

NodeManger

Jobhistory

## mapreduce 常见的join方法的几种方式及原理？

假设要进行join的数据分别来自File1和File2.

### reduce 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。

### map 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。

### 2SemiJoin

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

### 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

## 请列出你所知道的 Hadoop 调度器，并简要说明其工作方法？

YARN

a.默认调度器 FIFO 按照作业的提交的顺序进行调度

b.公平调度器 Fair Scheduler Clouder

版本默认的调度方式

同计算能力调度器类似 支持多队列 多用户 每个对垒中的资源量可以配置 同一个队列中的作业 公平共享队列的所有资源

c.能力调度器 Capacity Scheduler

Apache 版本默认的调度

根据作业计算的能力划分

## 请简述mapreduce中，combiner，partition作用？

（1） combiner（迷你版本的reduce） ：

在map端和reduce端具有相同key 进行合并

每一个 map 都可能会产生大量的本地输出 Combiner 作用就是对 map 输出先做一次合并，以减少在map和reduce之间的数据量传输 以提供网络IO性能 是Mapreduce shuffle 优化一种手段

（2） partition:

对map 输出结果进行分区 发送给不同的reducer 进行处理

## mapreduce的优化？

mapreduce 的优化核心点就是shuffle 。shuffle 从map端输入从到reduce 输入之间的过程 。 完成 数据的合并 排序 分块 合并的过程

1） 任务调度 将 map任务 分配数据所在节点上（本地化计算）

2） 数据预处理和 input split 大小

剔除无用数据和脏数据 将小文件合并成大文件 减少map 任务数量

3） 合理设置集群map和reduce的数量

4） combiner 函数 对map 输出的中间结果进行合并 减少网路IO传输

5） 压缩 对map的的输出进行压缩 map的输出首先存储在缓存中 达到一定量以后写入磁盘 map结束后会将结果复制给reduce ,对reduce 的最终结果进行压缩 ，这样可以能减少写HDFS的时间

6）自定Comparator 可以自定义数据类型实现复杂的目的 定义自已的comparator对数据进行比较，有些可以省去数据序列化与反序列化的时间

# JVM 相关

## jvm 内存调优

首先需要注意的是在对JVM内存调优的时候不能只看操作系统级别Java进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况，因为GC过后这个值是不会变化的，因此内存调优的时候要更多地使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。

对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

1）旧生代空间不足

调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象

2）Pemanet Generation空间不足

增大Perm Gen空间，避免太多静态对象

统计得到的GC后晋升到旧生代的平均大小大于旧生代剩余空间

控制好新生代和旧生代的比例

3）System.gc()被显示调用

垃圾回收不要手动触发，尽量依靠JVM自身的机制

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，

## 看看各部分比例不良设置会导致什么后果

1）新生代设置过小

一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

1. 新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；

二是新生代GC耗时大幅度增加 一般说来新生代占整个堆1/3比较合适

3）Survivor设置过小

导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4）Survivor设置过大

导致eden过小，增加了GC频率

另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

由内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM提供两种较为简单的GC策略的设置方式

5）吞吐量优先

JVM以吞吐量为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，来达到吞吐量指标。这个值可由-XX:GCTimeRatio=n来设置

6）暂停时间优先

JVM以暂停时间为指标，自行选择相应的GC策略及控制新生代与旧生代的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。这个值可由-XX:MaxGCPauseRatio=n来设置

## 通常参数设置：

1）堆设置：

-Xms:初始堆大小 -Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n:设置持久代大小

## 垃圾回收

GC(Garbage Collection)垃圾回收器机制是由垃圾回收器Garbage Collection实现的GC 是后台的守护进程，它的特别之处是一个低优先级进程可以根据内存的使用情况动态调整他的优先级因此它在内存中低到一定限度时才会自动运行 从而实现对内存的回收--垃圾回收的运行时间不确定的原因。

### 垃圾回收器常用算法：

#### 引用计数（Reference Counting）:

比较古老的回收算法 原理此对象有一个引用，即增加一个计数，删除一个引用就是减少一个计数，垃圾回收的时候 只回收计数为0的对象

问题：无法处理循环引用 A--->B B--->A

#### 标记-清除（Mark-Sweep）

算法分为两个阶段 第一阶段是从引用根几点开始标记所有被引用的对象，

第二阶段遍历整个heap 把未标记的对象清除

这个算法需要暂停整个应用 同时会产生内存的碎片

#### 复制（Coppying）

此算法把内存空间划分为两个相等的区域 每次只使用其中一个区域，垃圾回收的时候 遍历当前使用的区域 吧正在使用的对象复制到另一个区域中 算法每次只处理正在使用的对象 因此复制成本较小 同时复制过去以后能进行相应的调整 不会出“碎片”问题

缺点：需要两倍内存空间

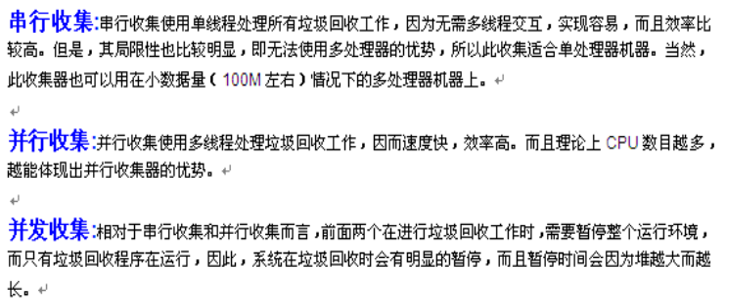
#### 标记-整理（Mark-Compact）:

结合上面“标记-清除”和“复制”两个算法的优点 也是分为两个解读那

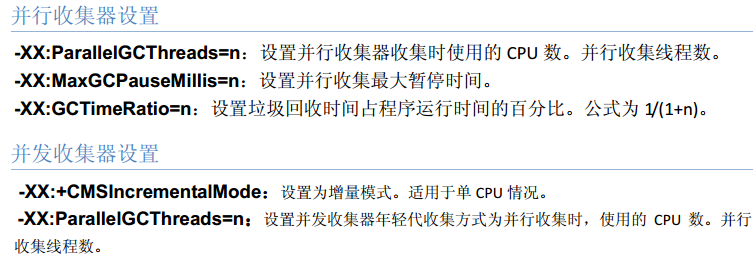
第一阶段是从引用根几点开始标记所有被引用的对象，

第二阶段遍历整个heap 把清除未标记对象并且把存活对象压缩到堆的其中某一块 按顺序排放 此算法 避免了“标记-清除”算法的碎片问题 和 “复制”算法空间问题

## 系统线程划分



## 典型配置和调优举例

## GC 种类：

Sun GC 主要方式有CMS 和 G1

CMS(Concurrent Mark-Sweep)是JDK 1.4 后期引用的新的GC算法

应对场景：响应时间的重要性比较高的引用

并且预期这部分应用能够承受垃圾回收线程和应用线程共享处理器资源

是以牺牲吞吐量为代价来获取最短回收停顿时间的垃圾回收器

针对年老代的回收

JVM 参数： -XX:+UseConcMarkSweepGC