1. **JRE、JDK、JVM 及 JIT 之间有什么不同？**

JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开发工具，如 Java 编译器，它也包含 JRE。JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。JIT 代表即时编译（Just In Time compilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。

1. **a.hashCode() 有什么用？与 a.equals(b) 有什么关系？**

hashCode() 方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap等等。它与 equals() 方法关系特别紧密。根据 Java 规范，两个使用 equal() 方法来判断相等的对象，必须具有相同的 hash code。

1. **Java 中，抽象类与接口之间有什么不同？**

Java 中，抽象类和接口有很多不同之处，但是最重要的一个是 Java 中限制一个类只能继承一个类，但是可以实现多个接口。抽象类可以很好的定义一个家族类的默认行为，而接口能更好的定义类型，有助于后面实现多态机制。

1. **final、finalize 和 finally 的不同之处？**

final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。finalize 方法是在对象被回收之前调用的方法，给对象自己最后一个复活的机会，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

1. **用哪两种方式来实现集合的排序？**

你可以使用有序集合，如 TreeSet 或 TreeMap，你也可以使用有顺序的的集合，如 list，然后通过 Collections.sort() 来排序。

1. **解释 Java 堆空间及 GC**

当通过 Java 命令启动 Java 进程的时候，会为它分配内存。内存的一部分用于创建堆空间，当程序中创建引用类型对象的时候，就从对空间中分配内存。GC 是 JVM 内部的一个进程，回收无效对象的内存用于将来的分配。

JVM内存模型  
1、栈  
Java栈是与每一个线程关联的，JVM在创建每一个线程的时候，会分配一定的栈空间给线程。存储局部变量、引用、方法、返回值等。  
StackOverflowError：如果在线程执行的过程中，栈空间不够用，那么JVM就会抛出此异常，这种情况一般是死递归造成的。  
2、堆

Java中堆是由所有的线程共享的一块内存区域，堆用来保存各种JAVA对象，比如数组，线程对象等。

分代垃圾回收  
不同的对象生命周期不同。与业务信息有关的对象生命周期较长，如Http请求中的session对象、线程、socket连接。程序运行过程中的临时变量生命周期较短，如String对象。  
把不同生命周期的对象放在不同代上，不同代上采用最合适它的垃圾回收方式进行回收。JVM中共划分为三个代：年轻代、年老代和持久代，其中持久代主要存放Java类的类信息，与垃圾收集要收集的Java对象关系不大，年轻代和年老代的划分是对垃圾收集影响较大的。  
年轻代：存放所有新生成的对象；  
年老代：在年轻代中经历了N次垃圾回收仍然存活的对象，将被放到年老代中，故都是一些生命周期较长的对象；  
持久代：用于存放静态文件，如Java类、方法等。（持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。）

1. **String、StringBuffer与StringBuilder的区别**

第一点：可变和适用范围。String对象是不可变的，而StringBuffer和StringBuilder是可变字符序列。每次对String的操作相当于生成一个新的String对象，而对StringBuffer和StringBuilder的操作是对对象本身的操作，而不会生成新的对象，所以对于频繁改变内容的字符串避免使用String，因为频繁的生成对象将会对系统性能产生影响。

第二点：线程安全。String由于有final修饰，是immutable的，安全性是简单而纯粹的。StringBuilder和StringBuffer的区别在于StringBuilder不保证同步，也就是说如果需要线程安全需要使用StringBuffer，不需要同步的StringBuilder效率更高。

1. **ArrayList、LinkedList、Vector的区别**

首先它们均是List接口的实现。ArrayList、LinkedList的区别

1.随机存取：ArrayList是基于可变大小的**数组**实现，LinkedList是**链接列表**的实现。这也就决定了对于随机访问的get和set的操作，ArrayList要优于LinkedList，因为LinkedList要移动指针。

2.插入和删除：LinkedList要好一些，因为ArrayList要移动数据，更新索引。

3.内存消耗：LinkedList需要更多的内存，因为需要维护指向后继结点的指针。

Vector从Java 1.0起就存在，在1.2时改为实现List接口，功能与ArrayList类似，但是Vector具备线程安全。

1. **线程和进程有什么区别**

从概念上： 进程：一个程序对一个数据集的动态执行过程，是**分配资源的基本单位**。 线程：存在于进程内，是进程内的**基本调度单位**。共享进程的资源。

从执行过程中来看： 进程：拥有独立的内存单元，而多个线程共享内存，从而提高了应用程序的运行效率。 线程：每一个独立的线程，都有一个程序运行的入口、顺序执行序列、和程序的出口。但是线程不能够独立的执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看：（重要区别） 多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但是，操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理及资源分配。

简言之，一个程序至少有一个进程,一个进程至少有一个线程。进程是资源分配的基本单位，线程共享进程的资源。

1. **OOM解决办法**

内存溢出的空间：Permanent Generation和Heap Space，也就是永久代和堆区

第一种情况永久代的溢出：出现这种问题的原因可能是应用程序加载了大量的jar或class，使虚拟机装载类的空间不够，与Permanent Generation Space的大小有关。

解决办法有2种：1、通过虚拟机参数-XX：PermSize和-XX：MaxPermSize调整永久代大小 2、清理程序中的重复的Jar文件，减少类的重复加载

第二种堆区的溢出：发生这种问题的原因是java虚拟机创建的对象太多，在进行垃圾回收之间，虚拟机分配的到堆内存空间已经用满了，与Heap Space的size有关。解决这类问题有两种思路：

1、检查程序，看是否存在死循环或不必要地重复创建大量对象，定位原因，修改程序和算法。

1. 通过虚拟机参数-Xms和-Xmx设置初始堆和最大堆的大小

**运行时异常与一般异常有何异同？**异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种**常见运行错误**。java编译器要求方法必须声明抛出可能发生的非运行时异常，但是并不要求必须声明抛出未被捕获的运行时异常。

**请写出你最常见到的5个runtime exception。**这道题主要考你的代码量到底多大，如果你长期写代码的，应该经常都看到过一些系统方面的异常，你不一定真要回答出5个具体的系统异常，但你要能够说出什么是系统异常，以及几个系统异常就可以了，当然，这些异常完全用其英文名称来写是最好的，如果实在写不出，那就用中文吧，有总比没有强！  
  
所谓系统异常，就是…..，它们都是RuntimeException的子类，在jdk doc中查RuntimeException类，就可以看到其所有的子类列表，也就是看到了所有的系统异常。我比较有印象的系统异常有：NullPointerException、ArrayIndexOutOfBoundsException、ClassCastException。ArrayStoreException，NumberFormatException

**同步和异步有何异同，在什么情况下分别使用他们？举例说明。**  
如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须进行同步存取。  
  
当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。

**多线程有几种实现方法?同步有几种实现方法?**  
多线程有两种实现方法，分别是继承Thread类与实现Runnable接口  
同步的实现方面有两种，分别是synchronized,wait与notify  
wait():使一个线程处于等待状态，并且释放所持有的对象的lock。  
sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉InterruptedException异常。  
notify():唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且不是按优先级。  
Allnotity():唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。

1. **Java 中 sleep 方法和 wait 方法的区别？**

两个方法都是用来暂停当前线程的执行的，其中sleep方法是本线程的一个停顿，在停顿期间不释放锁资源，停顿完设置的时间段之后继续执行本线程的下一条指令，而wait在停顿本线程之后释放该线程的锁，并一直等待其他线程的notify之后才继续接收到锁，然后继续往下执行。

一、继承**Thread**类

二、实现**Runnable**接口

（1）用synchronized关键字**修饰的方法**。

（2）有synchronized关键字**修饰的语句块**。

（3）使用特殊域变量(volatile)实现线程同步

（4）**使用局部变量**实现线程同步

**阐述静态变量和实例变量的区别。**静态变量是被static修饰符修饰的变量，也称为类变量，它**属于类**，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。**补充：**在Java开发中，上下文类Context和工具类Util中通常会有大量的静态成员。

**当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？**答：是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码，但是在Java中却做不到。

说明：Java中没有传引用实在是非常的不方便，这一点在Java 8中仍然没有得到改进，正是如此在Java编写的代码中才会出现大量的Wrapper类（将需要通过方法调用修改的引用置于一个Wrapper类中，再将Wrapper对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从C和C++转型为Java程序员的开发者无法容忍。

**抽象类（abstract class）和接口（interface）有什么异同？**答：抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义**构造器**，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是private、默认、protected、public的，而接口中的成员全都是public的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

**构造器（constructor）是否可被重写（override）**

答：构造器不能被继承，因此不能被重写，但可以被重载。

**HADOOP**

1. **简要描述如何安装配置一个apache开源版Hadoop，只描述即可，无需列出完整步骤，能列出步骤更好。**

(1) 使用root账户登录

(2) 修改IP

(3) 修改host主机名

(4) 配置SSH免密码登录

(5) 关闭防火墙

(6) 安装JDK

(7) 解压hadoop安装包

(8) 配置hadoop的核心文件 hadoop-env.sh，core-site.xml , mapred-site.xml ， hdfs-site.xml

(9) 配置hadoop环境变量

(10) 格式化 hadoop namenode-format

(11) 启动节点start-all.sh

**列出Hadoop集群中hadoop都分别需要启动哪些进程，他们的作用是什么。**

namenode：管理集群，并记录datanode文件信息。

Secondname:可以做冷备，对一定范围内数据做快照性备份。

Datanode:存储数据

yarn:管理任务，并将任务分配给tasktracker。

ResourceManager:

NodeManager

Tasktracker:任务执行方。

**请列出你知道的hadoop调度器，并简要说明工作方法。**

Fifo schedular :默认，先进先出的原则

Capacity schedular :计算能力调度器，选择占用最小、优先级高的先执行，依此类推。

Fair schedular: 公平调度，所有的job具有相同的资源。

1. **请列举曾经修改过的/etc 下的配置文件，并说明修改要解决的问题？**

/etc/profile这个文件，主要是用来配置环境变量。让hadoop命令可以在任意目录下面执行。

/ect/sudoers 修改系统范围环境变量时经常用；

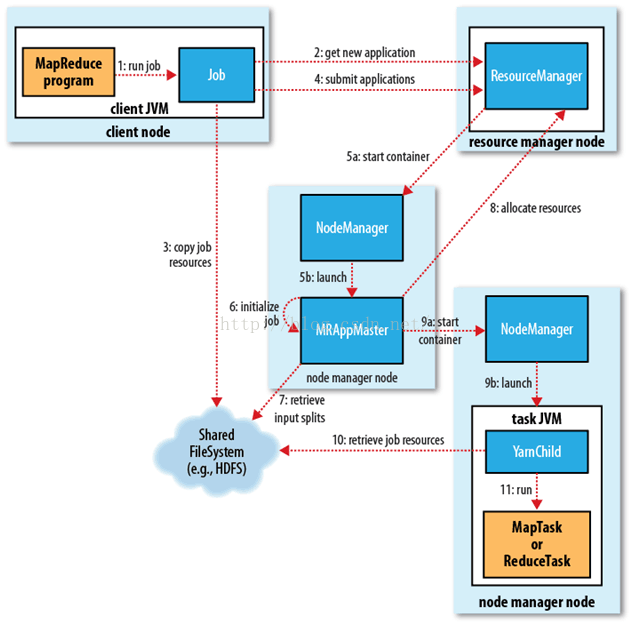
/etc/hosts增加局域网主机名和ip对应关系，省得再记住ip;

/etc/sysconfig/network 配置静态IP时需要。

/etc/inittab

**YARN**

1. **[YARN原理及其工作流程](http://blog.csdn.net/xiaoshunzi111/article/details/48810239)**



一 YARN组成元素及其功能

RM:YARN的管理者

NM:YARN的执行者

二 YARN工作原理

首先数据上传到集群中.然后将写好的程序打成架包通过命令提交MR作业.提交到集群后由集群管理者MR开始调度分配资源.到HDFS读取数据执行MapReduce相关进程对数据进行计算

三 具体流程

第一步: Client执行main()函数中runjob();开启作业

第二步:client向RM发送作业请求同时RM将作业id以及jar包存放路径返回给Client

第三步:Client会把Jar路径为前缀作业id为后缀作为唯一存放路径,将jar包写入到HDFS集群中,默认情况下jar包写10份,而其他数据只写3份 当改程序运行完后删除这些数据

第四步:Client再次将Jar存放地址(更为详细的描述)提交给RM.

第五步:RM将其放入调度器,向NM发送命令,NM开启MRAPPMaster进程,MR根据HDFS中jar包数据量为NM分配任务.

第六步:NM通过心跳机制接受调度器分配的任务

第七步:NM会开启内部YARNChild

第八步:YARNChild根据命令到HDFS检索作业资源,

第九步:YARNChild开启MapTask 或者Reduce Task

第十步:map计算YARNChild调度的数据.

1. **为什么会产生 yarn,它解决了什么问题，有什么优势？**

1、Hadoop YARN的目的是使得Hadoop数据处理能力超越MapReduce。

2、随着 YARN 的出现，您不再受到更简单的 MapReduce 开发模式约束，而是可以创建更复杂的分布式应用程序。实际上，您可以将 MapReduce 模型视为 YARN 架构可运行的一些应用程序中的其中一个，只是为自定义开发公开了基础框架的更多功能。这种能力非常强大，因为 YARN 的使用模型几乎没有限制，不再需要与一个集群上可能存在的其他更复杂的分布式应用程序框架相隔离，就像 MRv1 一样。甚至可以说，随着 YARN 变得更加健全，它有能力取代其他一些分布式处理框架，从而完全消除了专用于其他框架的资源开销，同时还简化了整个系统。

3、

（1、）YARN的设计减小了JobTracker的资源消耗，并且让监测每一个Job子任务（tasks)状态的程序分布式化了，更安全、更优美。

（2、）在新的Yarn中，ApplicationMaster是一个可变更的部分，用户可以对不同的编程模型写自己的AppMst，让更多类型的编程模型能够跑在Hadoop集群中。

（3、）对于资源的表示以内存为单位，比之前以剩余slot数目更加合理。

（4、）MRv1中JobTracker一个很大的负担就是监控job下的tasks的运行状况，现在这个部分就扔给ApplicationMaster做了， 而ResourceManager中有一个模块叫做ApplicationManager，它是监测ApplicationMaster的运行状况，如果出问题，会在其他机器上重启。

（5、）Container用来作为YARN的一个资源隔离组件，可以用来对资源进行调度和控制。

1. **列举几个配置文件优化**

fs.inmemory.size.mb 默认值： 100

说明： reduce阶段用户合并map输出的内存限制。这里设置200，可根据自身硬件设备进行更改测试。

io.file.buffer.size

默认值：4096

说明：SequenceFiles在读写中可以使用的缓存大小，这里设置为131072。貌似这个参数在新版本里变为了：file.stream-buffer-size，单位bytes 。

**MAPREDUCE**

1. **谈谈数据倾斜，如何发生的，并给出优化方案。**

原因：

（1）key 分布不均匀

（2）业务数据本身的特性

（3）建表时考虑不周

（4）某些 SQL 语句本身就有数据倾斜

map 处理数据量的差异取决于上一个 stage 的 reduce 输出，所以如何将数据均

匀的分配到各个 reduce 中，就是解决数据倾斜的根本所在。

优化：参数调节；

1. **用 mapreduce 怎么处理数据倾斜问题？**

本质：让各分区的数据分布均匀

可以根据业务特点，设置合适的**partition**策略

如果事先根本不知道数据的分布规律，**利用随机抽样器抽样后生成partition策略再处理**

1. **用mapreduce怎么处理数据倾斜问题？**

Mapreduce过程有可能某一个key的条数比其他key多很多（有时是百倍或者千倍之多），这条key所在的reduce节点所处理的数据量比其他节点就大很多，从而导致某几个节点迟迟运行不完，此称之为数据倾斜

1. 设置一个hash份数N，用来对条数众多的**key进行打散**。
2. 对有多条重复key的那份数据进行处理：从1到N将数字加在key后面作为新key，如果需要和另一份数据关联的话，则要重写比较类和分发类，如此实现多条key的平均分发。

（3）上一步之后，key被平均分散到很多不同的reduce节点。如果需要和其他数据关联，为了保证每个reduce节点上都有关联的key，对另一份单一key的数据进行处理：循环的从1到N将数字加在key后面作为新key

1. **你在项目中遇到了哪些难题，是怎么解决的？**

某些任务执行时间过长，且失败率过高，检查日志后发现没有执行完就失败，原因出在

hadoop 的 job 的 **timeout** 过短（相对于集群的能力来说），设置长一点即可

1. **MapReduce 中排序发生在哪几个阶段？这些排序是否可以避免？为什么？**

一个 MapReduce 作业由 Map 阶段和 Reduce 阶段两部分组成，这两阶段会对数

据排序，从这个意义上说，MapReduce 框架本质就是一个 Distributed Sort。

在 Map 阶段，Map Task 会在本地磁盘输出一个按照 key 排序（采用的是快速

排序）的文件（中间可能产生多个文件，但最终会合并成一个），在 Reduce 阶段，每

个 Reduce Task 会对收到的数据排序，这样，数据便按照 Key 分成了若干组，之后以

组为单位交给 reduce（）处理。很多人的误解在 Map 阶段，如果不使用 Combiner

便不会排序，这是错误的，不管你用不用 Combiner，Map Task 均会对产生的数据排

序（如果没有 Reduce Task，则不会排序，实际上 Map 阶段的排序就是为了减轻 Reduce

端排序负载）。由于这些排序是 MapReduce 自动完成的，用户无法控制，因此，在

hadoop 1.x 中无法避免，也不可以关闭，但 hadoop2.x 是可以关闭的。

1. **请简述Hadoop怎样实现二级排序。**

(1). 自定义一个类，实现 WritableComparable 这个接口，实现compareTo 方法

(2). Partitioner 自定义分区方法

(3). WritableComparator 自定义 ，实现compare方法。

1. **请简述hadoop怎样实现二级排序**

第一种方法是，Reducer将给定key的所有值都缓存起来，然后对它们再做一个Reducer内排序。但是，由于Reducer需要保存给定key的所有值，可能会导致出现内存耗尽的错误。

第二种方法是，将值的一部分或整个值加入原始key，生成一个合成key。这两种方法各有优势，第一种方法可能会更快一些(但有内存耗尽的危险)，第二种方法则是将排序的任务交给MapReduce框架，更符合Hadoop/Reduce的设计思想。这篇文章里选择的是第二种。我们将编写一个Partitioner，确保拥有相同key(原始key，不包括添加的部分)的所有数据被发往同一个Reducer，还将编写一个Comparator，以便数据到达Reducer后即按原始key分组。

1. **简述Hadoop实现join的几种方法。**

Map side join----大小表join的场景

Reduce join----适用于两张表的数据都非常大

Semi join----已知key值，根据key值查询

1. **请简述MapReduce中combiner, partition作用**

combiner是发生在map的最后一个阶段，其原理也是一个小型的reducer，主要作用是减少输出到reduce的数据量，缓解网络传输瓶颈，提高reducer的执行效率。

partition的主要作用将map阶段产生的所有kv对分配给不同的reducer task处理，可以将reduce阶段的处理负载进行分摊。

1. **hdfs 的数据压缩算法**

Lzo

Gzip

Default

Snapyy

如果要对数据进行压缩，最好是将原始数据转为SequenceFile 或者 Parquet File（Spark）

1. **NameNode 与** **SecondaryNameNode 的区别与联系？**

SecondaryNameNode ：分担 namenode 的工作量；是 NameNode 的冷备份；合并 fsimage和 fsedits 然后再发给 namenode。

合并fsimage和fsedits文件，然后发送并替换NameNode的fsimage文件，

同时自己留下一个副本。这个副本可供 NameNode 毁灭之后的部分文件恢复。

**如果 NameNode 意外终止，SecondaryNameNode 会接替它使集群继续工作。（错误 ）**

分析：

SecondaryNameNode 是帮助恢复，而不是替代，如何恢复，可以查看

1. **对 Hadoop 有没有调优经验，没有什么使用心得？（调优从参数调优讲起）**

1.规划合理的任务数目

2.增加输入文件的副本数

3.启动推测执行机制

4.设置失败容忍度

5.适当打开JVM重用功能

6.设置任务超时时间

7.合理使用DistributedCache

8.跳过坏记录

9.提高作业优先级

10.合理控制Reduce Task的启动时机

1. **MapReduce 出现单点负载多大，怎么负载平衡？（可以用 Partitioner）**

可以用 partioner 进行分区操作，将大数据分成多个小数据进行逐一处理。

**mapreduce的原理?**

Hadoop 中的MapReduce 是一个使用简易的**软件框架**，基于它写出来的应用程序能够运行在由上千个商用

机器组成的大型集群上，并以一种**可靠容错的式并行处理上T 级别的数据集**。

一个MapReduce 作业（job）通常会把输入的数据集**切分为若干独立的数据块**，由map 任务（task）以完

全并行的方式处理它们。框架会对map 的输出先进行排序，然后把结果输入给reduce 任务。通常作业的

输入和输出都会被存储在文件系统中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

通常，MapReduce 框架和分布式文件系统是运行在一组相同的节点上的，也就是说，计算节点和存储节点

通常在一起。这种配置允许框架在那些已经存好数据的节点上高效地调度任务，这可以使整个集群的网络

带宽被非常高效地利用。

MapReduce 框架由一个单独的master JobTracker 和每个集群节点一个slave TaskTracker 共同组成。master

负责调度构成一个作业的所有任务，这些任务分布在不同的slave 上，master 监控它们的执行，重新执行

已经失败的任务。而slave 仅负责执行由master 指派的任务

1. **MapReduce 程序如何发布？如果 MapReduce 中涉及到了第三方的 jar 包，该如何**

**处理？**

**hadoop jar** /jar包路径 包名+类名

可以放到 hadoop的 **lib目录**下

在classpath下建立 lib目录，将jar拷到lib目录下。

将jar包**打入目标jar包**中

1. **Hadoop 生态圈中各种框架的运用场景？**

1、HDFS（Hadoop分布式文件系统）

HDFS是Hadoop体系中数据存储管理的基础。它是一个**高度容错的系统**，能检测和应对硬件故障，用于在低成本的通用硬件上运行。

HDFS简化了文件的一致性模型，通过流式数据访问，提供高吞吐量应用程序数据访问功能，适合带有大型数据集的应用程序。它提供了一次写入多次读取的机制，数据以块的形式，同时分布在集群不同物理机器上。

2、Mapreduce（分布式计算框架）

MapReduce是一种**分布式计算模型**，用以进行大数据量的计算。

3、4. HBASE（分布式列存数据库）

是一个建立在HDFS之上，面向列的针对结构化数据的**可伸缩**、**高可靠**、**高性能**、**分布式和面向列**的**动态模式数据库**。提供了对大规模数据的**随机、实时**读写访问。

4、 Zookeeper（分布式协作服务）

解决分布式环境下的**数据管理问题**：**统一命名**，**状态同步**，**集群管理**，**配置同步**等。

5、 HIVE（数据仓库）

Hive定义了一种类似SQL的查询语言(HQL),将SQL转化为MapReduce任务在Hadoop上执行。通常用于离线分析。

6、Yarn(**分布式资源管理器**）

7、Phoenix（hbase sql接口）

8、. Kafka（分布式消息队列）

9、 Streaming（流计算模型） 实时处理分析

10、Spark(基于内存的计算模型)

12、Sqoop(数据ETL/同步工具）

13、.Flume（日志收集工具）

14、Mahout（数据挖掘算法库）

15、Oozie(工作流调度器）

1. **Hadoop调优**

**从应用程序角度进行优化**

**（1） 避免不必要的reduce任务**

如果要处理的数据是排序且已经分区的，或者对于一份数据, 需要多次处理, 可以先排序分区；然后自定义InputSplit, 将单个分区作为单个mapred的输入；在map中处理数据, Reducer设置为空。

这样, 既重用了已有的 “排序”, 也避免了多余的reduce任务。

**（2）外部文件引入**

有些应用程序要使用外部文件，如**字典，配置文件**等，这些文件需要在所有task之间共享，可以放到分布式缓存DistributedCache中（或直接采用-files选项，机制相同）。

更多的这方面的优化方法，还需要在实践中不断积累。

**（3） 为job添加一个Combiner**

为job添加一个combiner可以大大减少shuffle阶段从map task拷贝给远程reduce task的数据量。一般而言，combiner与reducer相同。

**（4） 根据处理数据特征使用最适合和简洁的Writable类型**

Text对象使用起来很方便，但它在由数值转换到文本或是由UTF8字符串转换到文本时都是低效的，且会消耗大量的CPU时间。当处理那些非文本的数据时，可以使用二进制的Writable类型，如IntWritable， FloatWritable等。二进制writable好处：避免文件转换的消耗；使map task中间结果占用更少的空间。

**（5） 重用Writable类型**

很多MapReduce用户常犯的一个错误是，在一个map/reduce方法中为每个输出都创建Writable对象。例如，你的Wordcout mapper方法可能这样写：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public void map(...) {      …      for (String word : words) {        output.collect(new Text(word), new IntWritable(1));      }    } |

这样会导致程序分配出成千上万个短周期的对象。Java垃圾收集器就要为此做很多的工作。更有效的写法是：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class MyMapper … {      Text wordText = new Text();      IntWritable one = new IntWritable(1);      public void map(...) {        for (String word: words) {          wordText.set(word);          output.collect(wordText, one);        }      }    } |

**（6） 使用StringBuffer而不是String**

当需要对字符串进行操作时，使用StringBuffer而不是String，String是read-only的，如果对它进行修改，会产生临时对象，而StringBuffer是可修改的，不会产生临时对象。

**（7）调试**

最重要，也是最基本的，是要掌握MapReduce程序调试方法，跟踪程序的瓶颈。具体可参考：

<http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/>

**Hadoop通用参数调整**

**（1） dfs.namenode.handler.count或mapred.job.tracker.handler.count**

namenode或者jobtracker中用于处理RPC的线程数，默认是10，较大集群，可调大些，比如64。

**（2） dfs.datanode.handler.count**

datanode上用于处理RPC的线程数。默认为3，较大集群，可适当调大些，比如8。需要注意的是，每添加一个线程，需要的内存增加。

**（3） tasktracker.http.threads**

HTTP server上的线程数。运行在每个TaskTracker上，用于处理map task输出。大集群，可以将其设为40~50。

**3.2.3 HDFS相关配置**

**（1） dfs.replication**

文件副本数，通常设为3，不推荐修改。

**（2） dfs.block.size**

HDFS中数据block大小，默认为64M，对于较大集群，可设为128MB或者256MB。（也可以通过参数mapred.min.split.size配置）

**（3） mapred.local.dir和dfs.data.dir**

这两个参数mapred.local.dir和dfs.data.dir 配置的值应当是分布在各个磁盘上目录，这样可以充分利用节点的IO读写能力。运行 Linux sysstat包下的iostat -dx 5命令可以让每个磁盘都显示它的利用率。

**3.2.4 map/reduce 相关配置**

**（1） {map/reduce}.tasks.maximum**

同时运行在TaskTracker上的最大map/reduce task数，一般设为(core\_per\_node)/2~2\*（cores\_per\_node）。

**（2） io.sort.factor**

当一个map task执行完之后，本地磁盘上(mapred.local.dir)有若干个spill文件，map task最后做的一件事就是执行merge sort，把这些spill文件合成一个文件（partition）。执行merge sort的时候，每次同时打开多少个spill文件由该参数决定。打开的文件越多，不一定merge sort就越快，所以要根据数据情况适当的调整。

**（3） mapred.child.java.opts**

设置JVM堆的最大可用内存，需从应用程序角度进行配置。

**3.2.5 map task相关配置**

**（1） io.sort.mb**

Map task的输出结果和元数据在内存中所占的buffer总大小。默认为100M，对于大集群，可设为200M。当buffer达到一定阈值，会启动一个后台线程来对buffer的内容进行排序，然后写入本地磁盘(一个spill文件)。

**（2） io.sort.spill.percent**

这个值就是上述buffer的阈值，默认是0.8，即80%，当buffer中的数据达到这个阈值，后台线程会起来对buffer中已有的数据进行排序，然后写入磁盘。

**（3） io.sort.record**

Io.sort.mb中分配给元数据的内存百分比，默认是0.05。这个需要根据应用程序进行调整。

**（4） mapred.compress.map.output/ Mapred.output.compress**

中间结果和最终结果是否要进行压缩，如果是，指定压缩方式（Mapred.compress.map.output.codec/ Mapred.output.compress.codec）。推荐使用LZO压缩。Intel内部测试表明，相比未压缩，使用LZO压缩的TeraSort作业运行时间减少60%，且明显快于Zlib压缩。

1. **列举几个配置文件优化？**

fs.inmemory.size.mb 默认值： 100

说明： reduce阶段用户合并map输出的内存限制。这里设置200，可根据自身硬件设备进行更改测试。

io.file.buffer.size

默认值：4096

说明：SequenceFiles在读写中可以使用的缓存大小，这里设置为131072。貌似这个参数在新版本里变为了：file.stream-buffer-size，单位bytes 。

1. **在mr环节中，那些环节需要优化，如何优化，请详细说明。**

    1、**setNumReduceTasks**  适当的设置reduce的数量，如果数据量比较大，那么可以增加reduce的数量

    2、适当的时候使用 combine 函数，减少网络传输数据量

    3、压缩map和reduce的输出数据

   4、使用SequenceFile二进制文件。

   5、通过application 的ui页面观察job的运行参数

  6、太多小文件，造成map任务过多的问题，应该可以先合并小文件，或者有一个特定的map作为处理小文件的输入

   7、map端效率低原因分析

* 源文件的大小远**小于**HDFS的块的大小。这意味着任务的开启和停止要耗费更多的时间，就没有足够的时间来读取并处理输入数据。
* 源文件无法分块。这导致需要通过网络IO从其他节点读取文件块。
* 一个节点的本地磁盘或磁盘控制器运行在降级模式中，读取写入性能都很差。这会影响某个节点，而不是全部节点。
* 源文件不来自于HDFS。则可能是Hadoop节点和数据源之间的延迟导致了性能低下。
* Map任务从其他数据节点读取数据。可以从JobTracker的map任务细节信息和任务运行尝试中找到输入块的位置。如果输入块的位置不是任务执行的节点，那就不是本地数据了。

**HDFS**

**HDFS存储的机制?**

HDFS 的三个实体

**数据块**

每个磁盘都有默认的数据块大小,这是磁盘进行读写的基本单位.构建于单个磁盘之上的文件系统通过磁盘

块来管理该文件系统中的块.该文件系统中的块一般为磁盘块的整数倍.磁盘块一般为512 字节.HDFS 也有

块的概念,默认为64MB(一个map 处理的数据大小).HDFS 上的文件也被划分为块大小的多个分块,与其他文件系统不同的是,HDFS 中小于一个块大小的文件不会占据整个块的空间.

HDFS 用块存储带来的第一个明显的好处一个文件的大小可以大于网络中任意一个磁盘的容量,数据块可以利用磁盘中任意一个磁盘进行存储.第二个简化了系统的设计,将控制单元设置为块,可简化存储管理,计算单个磁盘能存储多少块就相对容易.同时也消除了对元数据的顾虑,如权限信息,可以由其他系统单独管理.

**DataNode 节点**

DataNode 是HDFS 文件系统的工作节点,它们根据需要存储并检索数据块,受NameNode 节点调度.并且定

期向NameNode 发送它们所存储的块的列表

**NameNode 节点**

NameNode 管理HDFS 文件系统的命名空间,它维护着文件系统树及整棵树的所有的文件及目录.这些文件

以两个文件形式永久保存在本地磁盘上(命名空间镜像文件和编辑日志文件).NameNode 记录着每个文件中

各个块所在的数据节点信息但并不永久保存这些块的位置信息,因为这些信息在系统启动时由数据节点重

建.

没有NameNode,文件系统将无法使用.如提供NameNode 服务的机器损坏,文件系统上的所有文件丢失,我

们就不能根据DataNode 的块来重建文件.因此,对NameNode 的容错非常重要.第一种机制,备份那些组成文件系统元数据持久状态的文件.通过配置使NameNode 在多个文件系统上保存元数据的持久状态或将数据写入本地磁盘的同时,写入一个远程挂载的网络文件系统.当然这些操作都是原子操作.第二种机制是运行一个辅助的NameNode,它会保存合并后的命名空间镜像的副本,并在Name/Node 发生故障时启用.但是辅助NameNode 保存.态总是滞后于主力节点,所以在主节点全部失效后难免丢失数据.在这种情况下,一般把存储在远程挂载的网络文件系统-的数据复制到辅助NameNode 并作为新的主NameNode 运行

**HIVE**

**Hive有哪些方式保存元数据的，各有哪些特点。**

三种：内存数据库derby，挺小，不常用。

本地mysql。。常用

远程端mysql。。不常用

1. **hive中order by,sort by, distribute by, cluster by作用以及用法**
2. order by

Hive中的order by跟传统的sql语言中的order by作用是一样的，会对查询的结果做一次全局排序，所以说，只有hive的sql中制定了order by所有的数据都会到同一个reducer进行处理（不管有多少map，也不管文件有多少的block只会启动一个reducer）。但是对于大量数据这将会消耗很长的时间去执行。

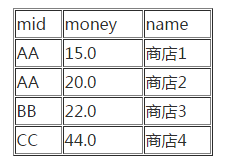
1. sort by

Hive中指定了sort by，那么在每个reducer端都会做排序，也就是说保证了局部有序（每个reducer出来的数据是有序的，但是不能保证所有的数据是有序的，除非只有一个reducer），好处是：执行了局部排序之后可以为接下去的全局排序提高不少的效率（其实就是做一次归并排序就可以做到全局排序了）。

1. distribute by和sort by一起使用

ditribute by是控制map的输出在reducer是如何划分的，举个例子，我们有一张表，mid是指这个store所属的商户，money是这个商户的盈利，name是这个store的名字

store:



select mid, money, name from store distribute by mid sort by mid asc, money asc  我们所有的mid相同的数据会被送到同一个reducer去处理，这就是因为指定了distribute by mid，这样的话就可以统计出每个商户中各个商店盈利的排序了（这个肯定是全局有序的，因为相同的商户会放到同一个reducer去处理）。这里需要注意的是distribute by必须要写在sort by之前。

1. cluster by

cluster by的功能就是distribute by和sort by相结合，如下2个语句是等价的：

select mid, money, name from store cluster by mid

select mid, money, name from store distribute by mid sort by mid

select mid, money, name from store cluster by mid sort by money

注意被cluster by指定的列只能是降序，不能指定asc和desc。

1. **Hive优化介绍**

　优化时，把hive sql当做mapreduce程序来读，会有意想不到的惊喜。理解hadoop的核心能力，是hive优化的根本。这是这一年来，项目组所有成员宝贵的经验总结。

* 长期观察hadoop处理数据的过程，有几个显著的特征:

1. 不怕数据多，就怕数据倾斜。
2. 对jobs数比较多的作业运行效率相对比较低，比如即使有几百行的表，如果多次关联多次汇总，产生十几个jobs，没半小时是跑不完的。map reduce作业初始化的时间是比较长的。
3. 对sum，count来说，不存在数据倾斜问题。
4. 对count(distinct ),效率较低，数据量一多，准出问题，如果是多count(distinct )效率更低。

* 优化可以从几个方面着手：

1. 好的模型设计事半功倍。
2. 解决数据倾斜问题。
3. 减少job数。
4. 设置合理的map reduce的task数，能有效提升性能。(比如，10w+级别的计算，用160个reduce，那是相当的浪费，1个足够)。
5. 自己动手写sql解决数据倾斜问题是个不错的选择。set hive.groupby.skewindata=true;这是通用的算法优化，但算法优化总是漠视业务，习惯性提供通用的解决方法。 Etl开发人员更了解业务，更了解数据，所以通过业务逻辑解决倾斜的方法往往更精确，更有效。
6. 对count(distinct)采取漠视的方法，尤其数据大的时候很容易产生倾斜问题，不抱侥幸心理。自己动手，丰衣足食。
7. 对小文件进行合并，是行至有效的提高调度效率的方法，假如我们的作业设置合理的文件数，对云梯的整体调度效率也会产生积极的影响。

优化时把握整体，单个作业最优不如整体最优。

**主要由三个属性来决定：**

* hive.exec.reducers.bytes.per.reducer   ＃这个参数控制一个job会有多少个reducer来处理，依据的是输入文件的总大小。默认1GB。
* hive.exec.reducers.max    ＃这个参数控制最大的reducer的数量， 如果 input / bytes per reduce > max  则会启动这个参数所指定的reduce个数。  这个并不会影响mapre.reduce.tasks参数的设置。默认的max是999。
* mapred.reduce.tasks ＃这个参数如果指定了，hive就不会用它的estimation函数来自动计算reduce的个数，而是用这个参数来启动reducer。默认是-1。

如果reduce太少：如果数据量很大，会导致这个reduce异常的慢，从而导致这个任务不能结束，也有可能会OOM 2、如果reduce太多：  产生的小文件太多，合并起来代价太高，namenode的内存占用也会增大。如果我们不指定mapred.reduce.tasks， hive会自动计算需要多少个reducer。

具体优化：

**1列裁剪**

　　Hive 在读数据的时候，可以只读取查询中所需要用到的列，而忽略其它列。 例如，若有以下查询：

SELECT a,b FROM q WHERE e<**10**;

　　在实施此项查询中，Q 表有 5 列（a，b，c，d，e），Hive 只读取查询逻辑中真实需要 的 3 列 a、b、e，而忽略列 c，d；这样做节省了读取开销，中间表存储开销和数据整合开销。

　　裁剪所对应的参数项为：hive.optimize.cp=true（默认值为真）

**2分区裁剪**

　　可以在查询的过程中减少不必要的分区。 例如，若有以下查询：

SELECT \* FROM (SELECTT a1,COUNT(**1**) FROM T GROUP BY a1) subq WHERE subq.prtn=**100**; #（多余分区）

SELECT \* FROM T1 JOIN (SELECT \* FROM T2) subq ON (T1.a1=subq.a2) WHERE subq.prtn=**100**;

　　查询语句若将“subq.prtn=100”条件放入子查询中更为高效，可以减少读入的分区 数目。 Hive 自动执行这种裁剪优化。

　　分区参数为：hive.optimize.pruner=true（默认值为真）

## 3JOIN操作

　　在编写带有 join 操作的代码语句时，应该将条目少的表/子查询放在 Join 操作符的左边。 因为在 Reduce 阶段，位于 Join 操作符左边的表的内容会被加载进内存，载入条目较少的表 可以有效减少 OOM（out of memory）即内存溢出。所以对于同一个 key 来说，对应的 value 值小的放前，大的放后，这便是“小表放前”原则。 若一条语句中有多个 Join，依据 Join 的条件相同与否，有不同的处理方法。

### 3.1JOIN原则

　　在使用写有 Join 操作的查询语句时有一条原则：应该将条目少的表/子查询放在 Join 操作符的左边。原因是在 Join 操作的 Reduce 阶段，位于 Join 操作符左边的表的内容会被加载进内存，将条目少的表放在左边，可以有效减少发生 OOM 错误的几率。对于一条语句中有多个 Join 的情况，如果 Join 的条件相同，比如查询：

INSERT OVERWRITE TABLE pv\_users

SELECT pv.pageid, u.age FROM page\_view p

JOIN user u ON (pv.userid = u.userid)

JOIN newuser x ON (u.userid = x.userid);

* 如果 Join 的 key 相同，不管有多少个表，都会则会合并为一个 Map-Reduce
* 一个 Map-Reduce 任务，而不是 ‘n’ 个
* 在做 OUTER JOIN 的时候也是一样

　　如果 Join 的条件不相同，比如：

INSERT OVERWRITE TABLE pv\_users

SELECT pv.pageid, u.age FROM page\_view p

JOIN user u ON (pv.userid = u.userid)

JOIN newuser x on (u.age = x.age);

　　Map-Reduce 的任务数目和 Join 操作的数目是对应的，上述查询和以下查询是等价的：

INSERT OVERWRITE TABLE tmptable

SELECT \* FROM page\_view p JOIN user u

ON (pv.userid = u.userid);

INSERT OVERWRITE TABLE pv\_users

SELECT x.pageid, x.age FROM tmptable x

JOIN newuser y ON (x.age = y.age);

## 4MAP JOIN操作

　　Join 操作在 Map 阶段完成，不再需要Reduce，前提条件是需要的数据在 Map 的过程中可以访问到。比如查询：

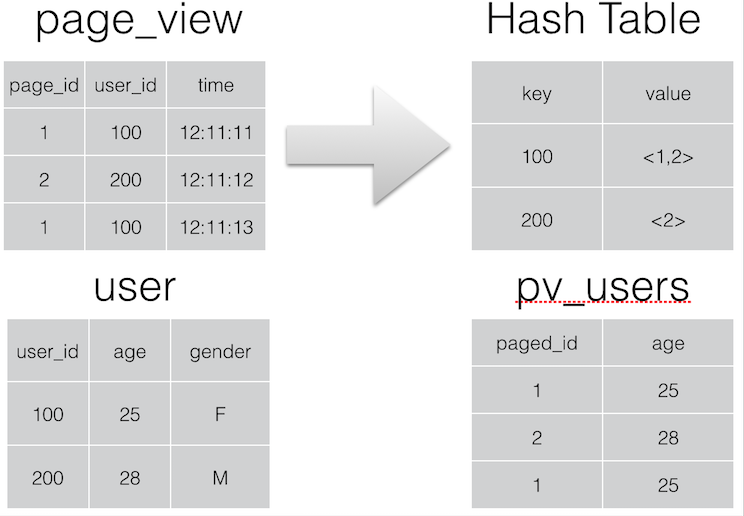
INSERT OVERWRITE TABLE pv\_users

SELECT /\*+ MAPJOIN(pv) \*/ pv.pageid, u.age

FROM page\_view pv

JOIN user u ON (pv.userid = u.userid);

　　可以在 Map 阶段完成 Join，如图所示：



　　相关的参数为：

* **hive.join.emit.interval = 1000**
* **hive.mapjoin.size.key = 10000**
* **hive.mapjoin.cache.numrows = 10000**

## ****5GROUP BY操作****

　　进行GROUP BY操作时需要注意一下几点：

* Map端部分聚合

　　事实上并不是所有的聚合操作都需要在reduce部分进行，很多聚合操作都可以先在Map端进行部分聚合，然后reduce端得出最终结果。

　　这里需要修改的参数为：

　　hive.map.aggr=true（用于设定是否在 map 端进行聚合，默认值为真） hive.groupby.mapaggr.checkinterval=100000（用于设定 map 端进行聚合操作的条目数）

* 有数据倾斜时进行负载均衡

　　此处需要设定 hive.groupby.skewindata，当选项设定为 true 是，生成的查询计划有两 个 MapReduce 任务。在第一个 MapReduce 中，map 的输出结果集合会随机分布到 reduce 中， 每个 reduce 做部分聚合操作，并输出结果。这样处理的结果是，相同的 Group By Key 有可 能分发到不同的 reduce 中，从而达到负载均衡的目的；第二个 MapReduce 任务再根据预处 理的数据结果按照 Group By Key 分布到 reduce 中（这个过程可以保证相同的 Group By Key 分布到同一个 reduce 中），最后完成最终的聚合操作。

## 6合并小文件

　　我们知道文件数目小，容易在文件存储端造成瓶颈，给 HDFS 带来压力，影响处理效率。对此，可以通过合并Map和Reduce的结果文件来消除这样的影响。

　　用于设置合并属性的参数有：

* 是否合并Map输出文件：hive.merge.mapfiles=true（默认值为真）
* 是否合并Reduce 端输出文件：hive.merge.mapredfiles=false（默认值为假）
* 合并文件的大小：hive.merge.size.per.task=256\*1000\*1000（默认值为 256000000）

1. **Sql优化**

**1**.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。  
  
 **2**.对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。  
  
 **3**.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：  
     select id from t where num is null  
     可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：  
     select id from t where num=**0**  
 **4**.应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：  
     select id from t where num=**10** or num=**20**  
     可以这样查询：  
     select id from t where num=**10**  
     union all  
     select id from t where num=**20**  
 **5**.下面的查询也将导致全表扫描：(不能前置百分号)  
     select id from t where name like ‘%abc%’  
    若要提高效率，可以考虑全文检索。  
  
 **6**.in 和 not in 也要慎用，否则会导致全表扫描，如：  
     select id from t where num in(**1**,**2**,**3**)  
     对于连续的数值，能用 between 就不要用 in 了：  
     select id from t where num between **1** and **3**

select xx,phone FROM send  a JOIN (  
 select '13891030091' phone  union select '13992085916' …………  UNION  SELECT '13619100234' ) b   
  on  a.Phone=b.phone  
--替代下面  很多数据隔开的时候  
in('13891030091','13992085916','13619100234'…………)

1. 如果在 where 子句中**使用参数**，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：  
        select id from t where num=@num     可以改为强制查询使用索引：  
        select id from t with(index(索引名)) where [num=@num](mailto:num=@num)  
     
    **8**.应尽量避免在 where 子句中对字段进行**表达式操作**，这将导致引擎放弃使用索引而进行全表扫描。如：  
        select id from t where num/**2**=**100**  
        应改为:  
        select id from t where num=**100**\***2**  
    **9**.应尽量避免在where子句中对字段进行**函数操作**，这将导致引擎放弃使用索引而进行全表扫描。如：  
        select id from t where substring(name,**1**,**3**)=’abc’–name以abc开头的id  
        select id from t where datediff(day,createdate,’**2005**-**11**-**30**′)=**0**–’**2005**-**11**-**30**′生成的id  
        应改为:  
        select id from t where name like ‘abc%’  
        select id from t where createdate>=’**2005**-**11**-**30**′ and createdate<’**2005**-**12**-**1**′  
     
    **10**.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。  
     
    **11**.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使 用，并且应尽可能的让字段顺序与索引顺序相一致。  
     
    **12**.不要写一些没有意义的查询，如需要生成一个空表结构：  
        select col1,col2 into #t from t where **1**=**0**  
        这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：  
        create table #t(…)  
     
    **13**.很多时候用 exists 代替 in 是一个好的选择：  
        select num from a where num in(select num from b)  
        用下面的语句替换：  
        select num from a where exists(select **1** from b where num=a.num)  
     
    **14**.并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。  
     
    **15**.**索引并不是越多越好**，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。  
     
    **16**.应尽可能的**避免更新 clustered** 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。  
     
    **17**.尽量使用**数字型字段**，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会 逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。  
     
    **18**.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。  
     
    **19**.任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。  
     
    **20**.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。  
     
    **21**.避免频繁创建和删除临时表，以减少系统表资源的消耗。  
     
    **22**.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使 用导出表。  
     
    **23**.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。  
     
    **24**.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。  
     
    **25**.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。  
     
    **26**.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。  
     
    **27**.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。  
     
    **28**.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。  
     
    **29**.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。  
     
    **30**.尽量避免大事务操作，提高系统并发能力。
2. **Hive有哪些方式保存元数据的，各有哪些特点。**

三种：内存数据库 derby，挺小，不常用 。

本地 mysql。。常用

远程端 mysql。。不常用

上网上找了下专业名称：single user mode..multi user mode...remote user mode

1. **Hive 内部表和外部表的区别？**

Hive 向内部表导入数据时，会将数据移动到数据仓库指向的路径；若是外部表，数据的具体存放目录由用户建表时指定。

在删除表的时候，内部表的元数据和数据会被一起删除； 而外部表只删除元数据，不删除数据。这样外部表相对来说更加安全些，数据组织也更加灵活，方便共享源数据。

**29. hive 跟hbase 的区别是？**

1. 两者分别是什么？

 Apache Hive是一个构建在Hadoop基础设施之上的数据仓库。通过Hive可以使用HQL语言查询存放在HDFS上的数据。HQL是一种类SQL语言，这种语言最终被转化为Map/Reduce. 虽然Hive提供了SQL查询功能，但是Hive不能够进行交互查询--因为它只能够在Haoop上批量的执行Hadoop。

    Apache HBase是一种Key/Value系统，它运行在HDFS之上。和Hive不一样，Hbase的能够在它的数据库上实时运行，而不是运行MapReduce任务。Hive被分区为表格，表格又被进一步分割为列簇。列簇必须使用schema定义，列簇将某一类型列集合起来（列不要求schema定义）。例如，“message”列簇可能包含：“to”, ”from” “date”, “subject”, 和”body”. 每一个 key/value对在Hbase中被定义为一个cell，每一个key由row-key，列簇、列和时间戳。在Hbase中，行是key/value映射的集合，这个映射通过row-key来唯一标识。Hbase利用Hadoop的基础设施，可以利用通用的设备进行水平的扩展。

2. 两者的特点

  Hive帮助熟悉SQL的人运行MapReduce任务。因为它是JDBC兼容的，同时，它也能够和现存的SQL工具整合在一起。运行Hive查询会花费很长时间，因为它会默认遍历表中所有的数据。虽然有这样的缺点，一次遍历的数据量可以通过Hive的分区机制来控制。分区允许在数据集上运行过滤查询，这些数据集存储在不同的文件夹内，查询的时候只遍历指定文件夹（分区）中的数据。这种机制可以用来，例如，只处理在某一个时间范围内的文件，只要这些文件名中包括了时间格式。

    HBase通过存储key/value来工作。它支持四种主要的操作：增加或者更新行，查看一个范围内的cell，获取指定的行，删除指定的行、列或者是列的版本。版本信息用来获取历史数据（每一行的历史数据可以被删除，然后通过Hbase compactions就可以释放出空间）。虽然HBase包括表格，但是schema仅仅被表格和列簇所要求，列不需要schema。Hbase的表格包括增加/计数功能。

3. 限制

  Hive目前不支持更新操作。另外，由于hive在hadoop上运行批量操作，它需要花费很长的时间，通常是几分钟到几个小时才可以获取到查询的结果。Hive必须提供预先定义好的schema将文件和目录映射到列，并且Hive与ACID不兼容。

    HBase查询是通过特定的语言来编写的，这种语言需要重新学习。类SQL的功能可以通过Apache Phonenix实现，但这是以必须提供schema为代价的。另外，Hbase也并不是兼容所有的ACID特性，虽然它支持某些特性。最后但不是最重要的--为了运行Hbase，Zookeeper是必须的，zookeeper是一个用来进行分布式协调的服务，这些服务包括配置服务，维护元信息和命名空间服务。

4. 应用场景

    Hive适合用来对一段时间内的数据进行分析查询，例如，用来计算趋势或者网站的日志。Hive不应该用来进行实时的查询。因为它需要很长时间才可以返回结果。

    Hbase非常适合用来进行大数据的实时查询。Facebook用Hbase进行消息和实时的分析。它也可以用来统计Facebook的连接数。

5. 总结

    Hive和Hbase是两种基于Hadoop的不同技术--Hive是一种类SQL的引擎，并且运行MapReduce任务，Hbase是一种在Hadoop之上的NoSQL 的Key/vale数据库。当然，这两种工具是可以同时使用的。就像用Google来搜索，用FaceBook进行社交一样，Hive可以用来进行统计查询，HBase可以用来进行实时查询，数据也可以从Hive写到Hbase，设置再从Hbase写回Hive。

1. **hive 跟 hbase 的区别是？**

1.Hive是建立在Hadoop之上为了减少**MapReduce jobs编写**工作的批处理系统，HBase是为了支持弥补Hadoop对**实时操作**的缺陷的项目 。

2.想象你在操作RMDB数据库，如果是全表扫描，就用Hive+Hadoop,如果是索引访问，就用HBase+Hadoop 。

3.Hive query就是MapReduce jobs可以从5分钟到数小时不止，**HBase是非常高效**的，肯定比Hive高效的多。

4.Hive本身不存储和计算数据，它完全依赖于HDFS和MapReduce，**Hive中的表纯逻辑。**

5.hive借用hadoop的MapReduce来完成一些hive中的命令的执行

6.**hbase是物理表**，不是逻辑表，提供一个超大的内存hash表，搜索引擎通过它来存储索引，方便查询操作。

7.**hbase是列存储**。

8.hdfs作为底层存储，hdfs是存放文件的系统，而Hbase负责组织文件。

9.hive需要用到hdfs存储文件，需要用到MapReduce计算框架。

**HBASE**

1. **介绍一下 HBase 过滤器**

Hbase提供的专用过滤器直接继承自FilterBase,其中一些过滤器只能做行筛选，因此只适合于扫描操作，对get（），这些过滤器限制的更苛刻：要么包含整行，要么什么都不包括

过滤器也被称为下**推判断器**（push-down predicates），支持你把数据过滤标准从客户端下推到服务器这些过滤逻辑在读操作时使用，对返回客户端的数据有影响。这样通过减少网络传输的数据来节省网络IO

1. **Hbase 的 rowkey 怎么创建比较好？列族怎么创建比较好？**

rowKey最好要创建有规则的rowKey，即最好是**有序**的。

经常需要**批量读取的数据**应该让他们的rowkey**连续**；

将经常需要作为**条件查询的关键词**组织到rowkey中；

列族的创建：按照**业务特点**，把**数据归类**，不同类别的放在不同列族

1. **Hbase 内部是什么机制？**

Hbase是一个能适应联机业务的数据库系统

**物理存储**：hbase的持久化数据是存放在hdfs上

**存储管理**：一个表是划分为很多region的，这些region分布式地存放在很多regionserver上

Region内部还可以划分为store，store内部有memstore和storefile

**版本管理**：hbase中的数据更新本质上是不断追加新的版本，通过compact操作来做版本间的文件合并Region的split

**集群管理**：zookeeper + hmaster（职责） + hregionserver（职责）

1. **hbase 过滤器实现原则**

可以说一下过滤器的父类（**比较过滤器**，**专用过滤器**）

过滤器有什么用途：

**增强**hbase**查询数据**的功能

减少服务端返回给客户端的数据量

1. **Hbase 中的 metastore 用来做什么的？**

MemStore是 一个有序的内存缓存区，用户写入的数据首先放

入MemStore，当MemStore满了以后Flush成一个StoreFile

（存储时对应为HFile）， 当StoreFile数量增到一定阈值，触发

Compact合并，将多个StoreFiles合并成一个StoreFile。

StoreFiles 合并后逐步形成越来越大的StoreFile，当Region内

所有StoreFiles（Hfile）的总大小超过阈值

（hbase.hregion.max.filesize）即触发分裂Split，把当前的

Region Split成2个Region，父Region下线，新Split出的2个孩

子Region被HMaster分配到合适的HRegionServer 上，使得

原先1个Region的压力得以分流到2个Region上

1. **hbase 怎么预分区？**

1.取样，先随机生成一定数量的rowkey,将取样数据按升序排序放到一个集合里

2.根据预分区的region个数，对整个集合平均分割，即是相关的splitKeys.

3.指定region间的rowkey临界值.

1. **hbase 的并发问题？**

1、大幅度**减少**了访问hbase的**进程数**以减少对hbase的并发连接，为了不降低处理能力，在进程内部使用**更多的处理线程**,很大程度缓解了hbase region server的压力。

2.**避免**HBase**访问热点**

3.**对于日志优化**（有些不必要的操作，可以关闭日志）

1. **hdfs 和hbase 各自使用场景。**

整理总结：

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

1. **Hbase中的META和ROOT用来做什么的？**

ROOT-表包含.META.表所在的region列表，该表只会有一个Region；

Zookeeper中记录了-ROOT-表的location。

META.表包含所有的用户空间region列表，以及RegionServer的服务器地址。

1. **HBase性能优化**

### 查询缓存

Scan的caching属性默认值是1，意味着扫描器每次从region服务器抓取1条记录进行匹配。我们可以设置caching为比1大得多的值。例如，设置为500，则一次可以抓取500条，需要注意的是该值设得越大服务器的内存开销会越多。

HTableInterface hTable=getHTable(tableName);

 Scan scan=new Scan();

/\*设置缓存\*/

scan.setCaching(StaticConfig.getIControl\_hbase\_cache());

ResultScanner scanner= hTable.getScanner(scan);

### 多线程配置

Hbase.regionser.handler.count

RegionServer中RPC监听器实例的数量。对于master来说，这个属性是master受理的处理线程（handler）数量。默认值是10。

根据调控层的业务场景，1条运价的匹配查询就会产生4条hbase并发查询。如果有20条，就可能有80条并发，这个并发量是相当的。除了将该参数适当调大可以增加并发处理能力外，还跟集群的数量和服务器的配置有直接的关系，预计集群数量越多，服务器CPU核数越高，并发处理能力越强。

### 预分区

HRegion是Hbase中分布式存储和负载均衡的最小单元。最小单元就表示不同的Hregion可以分布在不同的HRegion server上。但一个Hregion是不会拆分到多个server上的。

Hbase.hregion.max.filesize

HstoreFile的最大值。Region中任何一个列族的存储文件如果超过了这个上限，就会被拆分成两个region。默认：268435456（256x1024x1024），即256M。

我们的调控文件比较小，要达到分区最大上限256M需要较多的调控文件。为了提高并发量，我们需要在没有达到分区上限的情况下，产生多个hregion来保存和处理数据，这里就用hbase的预分区功能。

示例：

Configuration conf = HBaseConfiguration.create()

HBaseAdmin admin = new HBaseAdmin(conf);

HTableDescriptor desc = new HTableDescriptor(

Bytes.toBytes(tablename));

HColumnDescriptor coldef = new HColumnDescriptor(

Bytes.toBytes(colfamily));

admin.createTable(desc, Bytes.toBytes(1L), Bytes.toBytes(10L), 10);

//以第一位字符不同划分区

desc.setValue(HTableDescriptor.SPLIT\_POLICY,

KeyPrefixRegionSplitPolicy.class.getName());

desc.setValue("prefix\_split\_key\_policy.prefix\_length", "1");

1. **HBase的RowKey设计**

Rowkey长度原则

Rowkey是一个二进制码流，Rowkey的长度被很多开发者建议说设计在10~100个字节，不过建议是越短越好，**不要超过16个字节。**

原因如下：

（1）数据的持久化文件HFile中是按照KeyValue存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用100\*1000万=10亿个字节，将近1G数据，这会极大影响HFile的存储效率；

（2）MemStore将缓存部分数据到内存，如果Rowkey字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好。

（3）目前操作系统是都是64位系统，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

Rowkey散列原则

如果Rowkey是按时间戳的方式递增，不要将时间放在二进制码的前面，建议将Rowkey的高位作为散列字段，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个Regionserver实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息将产生所有新数据都在一个 RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别RegionServer，降低查询效率。

Rowkey唯一原则

必须在设计上保证其唯一性。

**ZOOKEPEER**

1. **zookeeper 优点，用在什么场合**

ZooKeeper是一个分布式的，开放源码的**分布式应用程序协调服务**，是Google的Chubby一个开源的实现，是**Hadoop**和**Hbase**的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。

1. **写出你对 zookeeper 的理解**

提示：大部分分布式应用需要一个主控、协调器或控制器来管理物理分布的子

进程（如资源、任务分配等）。目前，大部分应用需要开发私有的协调程序，缺乏一个

通用的机制协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器。

ZooKeeper：提供通用的**分布式锁服务**，用以协调分布式应用。

1. **spark使用zookeeper配置高可用的原理。**

原理是spark的master节点的运行状态数据和备master节点的信息保存在zookeeper集群的服务器上，当主master节点宕机以后zookeeper会讲原主master节点的状态数据恢复到备master节点上，同时备master节点的状态变成主master节点并开始执行主master节点的工作任务。

**SPARK**

1. **在使用hadoop 或者是spark 中遇到过哪些问题，是如何处理解决的。**

数据倾斜。

出现这种情况：多数是由于代码的质量写的不够健壮。查看日志：发现问题。

spark-出现OOM

小数据量的情况可以**cache**，数据量大的情况必须考虑内存使用。

1. **Spark的RDD操作有哪两类，有什么区别，分别举出些例子来**

一种是转化操作（transformation）

一种是行为操作（action）

转化操作是**返回一个新的RDD的操作**，如filter（），map（）

行为操作是**返回结果或者将结果写入外部系统的操作**，如count（），first（）

1. **Spark Streaming的工作原理**

Spark Streaming 是基于spark的流式批处理引擎，其基本原理是把输入数据以某一时间间隔批量的处理，当批处理间隔缩短到秒级时，便可以用于处理实时数据流。

具体流程：

1. 客户端提交作业后启动Driver，Driver是spark作业的Master。
2. 每个作业包含多个Executor，每个Executor以线程的方式运行task，Spark Streaming至少包含一个receiver task。
3. Receiver接收数据后生成Block，并把BlockId汇报给Driver，然后备份到另外一个Executor上。
4. ReceiverTracker维护Reciver汇报的BlockId。
5. Driver定时启动JobGenerator，根据Dstream的关系生成逻辑RDD，然后创建Jobset，交给JobScheduler。
6. JobScheduler负责调度Jobset，交给DAGScheduler，DAGScheduler根据逻辑RDD，生成相应的Stages，每个stage包含一到多个task。
7. TaskScheduler负责把task调度到Executor上，并维护task的运行状态。
8. 当tasks，stages，jobset完成后，单个batch才算完成。
9. **Spark性能优化主要有哪些手段？**

将默认调用的java序列化器改为kyro序列化器(减少序列化数据80%的空间占用(问：为何序列化可以减少存储空间占用(回答：对同类型的数据对象的头进行压缩合并：我感觉这个我回答错误了)))；

由于Spark1.6.0的统一内存管理模型，若算法的数据量大，而计算逻辑较为简单，可以增大内存管理中**cache**块的比例(默认70%(我也一下子想不起来，感觉这个附近))，如果是数据量小而算法逻辑复杂，可以适当减少cache快的比例；

​如果因对是集群CPU资源过分盈余，可以采用增加core的数目，但是core的数目增加到一定程度后，依旧无法完全利用CPU的计算资源，可以选择增加Executor的数目，通常环境下，一个Executor推荐设置5个Core的个数，超过5个推荐增加Executor的个数

​面试官没有询问关于OOM情况的优化(后面了解，他们目前还是处于测试环境，集群性能相对优越，这方面应该还没有碰到情况)

1. **flume如何和sparkStreaming结合使用**

Flume通过配置sink将event以avro的方式发送到sparkStreaming运行的端口，sparkStreaming接收到数据以后在对DStream的操作过程中将数据**反序列化并解析**。

38. Phoenix的作用和用处

Phoenix为hbase提供了一种sql的查询语法，它把对hbase的操作封装成jdbc的方式，让我们可以通过java代码或者phoenix客户端上使用sql来操作hbase

**FLUME**

1. **flume-ng的工作原理**

Flume NG是Cloudera提供的一个**分布式、可靠、可用的**系统，它能够将不同数据源的海量日志数据进行高效收集、聚合、移动，最后存储到一个中心化数据存储系统中。

Flume的架构主要有一下几个核心概念：

* Event：一个数据单元，带有一个可选的消息头
* Flow：Event从源点到达目的点的迁移的抽象
* Client：操作位于源点处的Event，将其发送到Flume Agent
* Agent：一个独立的Flume进程，包含组件Source、Channel、Sink
* Source：用来消费传递到该组件的Event
* Channel：中转Event的一个临时存储，保存有Source组件传递过来的Event
* Sink：从Channel中读取并移除Event，将Event传递到Flow Pipeline中的下一个Agent（如果有的话）

1. **kafka和flume-ng的集成方式和原因。**

1．Kafka可以作为消息的收集者，flume-ng的source作为kafka的消息消费者，这种集成方式，是当收集的信息需要存放入hdfs中时flume可以作为一种与hdfs交互方便的桥梁，只需配置sink就可以把数据轻松的存入hdfs或者hbase中

2.flume-ng作为消息的采集端，把采集到的消息通过sink发送到kafka，作为kafka的生产者，kafka作为一种高吞吐量的消息收集工具，可以作为flume-ng采集到的信息的一种缓存，然后在将消息存放入目标存储系统，这种情况一般适用于信息的采集和消费速度不一致时，使用kafka作为一种缓存机制的解决方案。

**kafka**

1. **kafka的工作原理**

Kafka是一个信息收集平台，它的特性是**高吞吐量**，它对信息的**收集和消费**都是被动的，生产者讲消息发送到**broker**，消费者从broker中获取并消费消息。在kafka中通过建立Topic来接收保存同一类型的消息，topic是可以分区的，每个分区保存在不同的broker上面，并且可以实现消息的发布/订阅机制。

**sqoop**

1. **qoop的基本原理和使用方法**

Sqoop是一个**数据传递的工具**，主要用于在Hadoop(Hive)与传统的数据库(mysql、postgresql...)间进行数据的传递，可以将一个关系型数据库（例如 ： MySQL ,Oracle ,Postgres等）中的数据导进到Hadoop的HDFS中，也可以将HDFS的数据导进到关系型数据库中。

1. **列举你知道的常用的Hadoop管理和监控的命令、比如hdfs dfs -mkdir /usr**

**-ls -cat -text -cp -put -chmod -chown**

**-du -get -copyFromLocal -copyToLocal**

**-mv -rm - tail -chgrp**