**大数据面试相关知识**

大数据面试面经

<https://www.nowcoder.com/discuss/13857?type=2&order=3&pos=160&page=1>

**一、Hadoop知识**

**1、常见的mapreduce join方法介绍**

<http://www.cnblogs.com/chengyeliang/p/4512207.html>

<http://www.cnblogs.com/raymoc/p/5323824.html>

假设要进行join的数据分别来自File1和File2.

**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（分布式缓存）**，使用该类的方法如下：

* 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**。
* 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。但这种方法的前提是已知所有需要参与join的key。

**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（过滤操作）**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>

**5）join优化手段之一---二次排序**

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？即：对于同一个key，reduce函数接收到的 value list是按照value排序的。这种应用需求在join操作中很常见，比如，希望相同的key中，小表对应的value排在前面。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接口或者调用 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**。

<http://www.cnblogs.com/xuxm2007/archive/2011/09/03/2165805.html>

**6）总结**

join的核心就是小表内存能不能放下，能放下的map join放不下的，reduce join。

reduce join 大量的网络和磁盘io，性能很差，想办法优化：

* 办法一：半连接，map过滤join需求的列和key
* 办法二：BloomFilter过滤**肯定不在驱动表**中的key记录
* 办法三：二次排序。便于区分两表及merge join

**2、DistributedCache原理**

<http://dongxicheng.org/mapreduce-nextgen/hadoop-distributedcache-details/>

**1）概念**

DistributedCache是Hadoop提供的文件缓存工具，它能够**自动将指定的文件分发到各个节点上**，缓存到本地，供用户程序读取使用。它具有以下几个**特点**：缓存的文件是只读的，修改这些文件内容没有意义；用户可以**调整文件可见范围**（比如只能用户自己使用，所有用户都可以使用等），进而防止重复拷贝现象；按需拷贝，文件是通过HDFS作为共享数据中心分发到各节点的，且只发给任务被调度到的节点。

**2）应用场景**

Hadoop DistributedCache有以下几种典型的应用场景：

* **分发字典文件**，一些情况下Mapper或者Reducer需要用到一些外部字典，比如黑白名单、词表等；
* **map-side join**：当多表连接时，一种场景是一个表很大，一个表很小，小到足以加载到内存中，这时可以使用DistributedCache将小表分发到各个节点上，以供Mapper加载使用；
* **自动化软件部署**：有些情况下，MapReduce需依赖于特定版本的库，比如依赖于某个版本的PHP解释器，一种做法是让集群管理员把这个版本的PHP装到各个机器上，这通常比较麻烦，另一种方法是使用DistributedCache分发到各个节点上，程序运行完后，Hadoop自动将其删除。

**3）使用方式**

Hadoop提供了两种DistributedCache使用方式，一种是通过API，在程序中设置文件路径，另外一种是通过命令行（-files，-archives或-libjars）参数告诉Hadoop，个人建议使用第二种方式，该方式可使用以下**三个参数设置文件**：

* **-files：**将指定的本地/hdfs文件分发到各个Task的工作目录下，不对文件进行任何处理；
* **-archives：**将指定文件分发到各个Task的工作目录下，并对名称后缀为“.jar”、“.zip”，“.tar.gz”、“.tgz”的**文件自动解压**，默认情况下，解压后的内容存放到工作目录下名称为解压前文件名的目录中，比如压缩包为dict.zip，则解压后内容存放到目录dict.zip中。为此，你可以给文件起**个别名/软链接**，比如**dict.zip#dict**，这样，压缩包会被解压到目录dict中。
* **-libjars：指定待分发的jar包**，Hadoop将这些jar包分发到各个节点上后，会将其自动添加到任务的CLASSPATH环境变量中。

**4）可见性分析**

DistributedCache分发的文件是有**可见范围**的，有的文件可以只对当前程序可见，程序运行完后，直接删除；有的文件只对当前用户可见（该用户所有程序都可以访问）；有的文件对所有用户可见。DistributedCache会为每种资源（文件）计算一个唯一ID，以识别每个资源，从而**防止资源重复下载**，举个例子，如果文件可见范围是所有用户，则在每个节点上，第一个使用该文件的用户负责缓存该文件，之后的用户直接使用即可，无需重复下载。那么，Hadoop是怎样区分文件可见范围的呢？

**（1）Hadoop 1.0**

在Hadoop 1.0版本中，Hadoop是以HDFS文件的属性作为标识判断文件可见性的，需要注意的是，**待缓存的文件即使是在Hadoop提交作业的客户端上，也会首先上传到HDFS的某一目录下，再分发到各个节点上的**，因此，HDFS是缓存文件的必经之路。对于经常使用的文件或者字典，建议放到HDFS上，这样可以防止每次重复下载，做法如下：

比如将数据保存在HDFS的/dict/public目录下，并将/dict和/dict/public两层目录的可执行权限全部打开（在Hadoop中，可执行权限的含义与linux中的不同，**该权限只对目录有意义，表示可以查看该目录中的子目录**），这样，里面所有的资源（文件）便是所有用户可用的，并且第一个用到的应用程序会将之缓存到各个节点上，之后所有的应用程序无需重复下载，可以在提交作业时通过以下命令指定：

-files hdfs:///dict/public/blacklist.txt, hdfs:///dict/public/whilelist.txt

如果有多个HDFS集群可以指定namenode的对外rpc地址：

-files hdfs://host:port/dict/public/blacklist.txt, hdfs://host:port/dict/public/whilelist.txt

DistributedCache会将blacklist.txt和whilelist.txt两个文件缓存到各个节点的一个公共目录下，并在需要时，在任务的工作目录下建立一个指向这两个文件的软连接。如果可执行权限没有打开，则默认只对该应用程序的拥有者可见，该用户所有应用程序可共享这些文件。一旦你对/dict/public下的某个文件进行了**修改**，则下次有作业用到对应文件时，会发现文件被修改过了，进而自动重新缓存文件。

对于一些频繁使用的字典，不建议存放在客户端，每次通过-files指定，这样的文件，每次都要经历以下流程：**上传到HDFS上—》缓存到各个节点上—》之后不再使用这些文件，直到被清除**，也就是说，这样的文件，只会被这次运行的应用程序使用，如果再次运行同样的应用程序，即使文件没有被修改，也会重新经历以上流程，非常耗费时间，尤其是字典非常多，非常大时。

DistributedCache内置缓存置换算法，一旦缓存（文件数目达到一定上限或者文件总大小超过某一上限）满了之后，会**踢除最久没有使用的文件**。

**（2）Hadoop 2.0**

在Hadopo 2.0中，自带的MapReduce框架仍支持1.0的这种DistributedCache使用方式，但DistributedCache本身是由YARN实现的，不再集成到MapReduce中。YARN还提供了很多相关编程接口供用户调用，有兴趣的可以阅读源代码。

下面介绍Hadoop 2.0中，DistributedCache通过命令行分发文件的基本使用方式：

* （1）运行Hadoop自带的example例子， dict.txt会被缓存到各个Task的工作目录下，因此，直接像读取本地文件一样，在Mapper和Reducer中，读取dict.txt即可：

**3、Hadoop RPC机制**

<http://blog.csdn.net/u010010428/article/details/51345693>

<http://www.cnblogs.com/sh425/p/6893491.html>

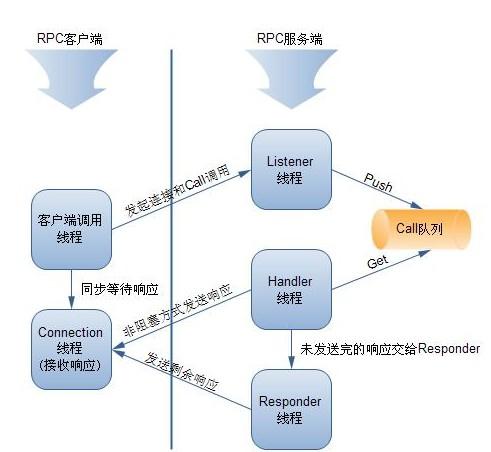
<http://www.cnblogs.com/shijianlvzhe/p/6026062.html>

<http://blog.csdn.net/haihongazar/article/details/52572980>

RPC（Remote Procedure Call Protocol，远程过程调用协议）是Hadoop的基础组件，提供分布式环境下的对象调用功能。概括的说，RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

Hadoop的RPC机制

* **序列化层**：Clent与Server端通信传递的信息采用了Hadoop里提供的序列化类或自定义的Writable类型；
* **函数调用层**：Hadoop RPC通过动态代理以及java反射实现函数调用；
* **网络传输层**：Hadoop RPC采用了基于TCP/IP的socket机制；
* **服务器端框架层**：RPC Server利用java NIO以及采用了事件驱动的I/O模型，提高RPC Server的并发处理能力；



使用Hadoop RPC的四大步骤：

* **定义RPC协议**：RPC协议是客户端和服务器端之间的通信接口，它定义了服务器端对外提供的服务接口。
* **实现RPC协议**：Hadoop RPC协议通常是一个Java接口，用户需要实现该接口。
* **构造和启动RPC SERVER**：直接使用静态类Builder构造一个RPC Server，并调用函数start()启动该Server。
* **构造RPC Client并发送请求**：使用静态方法getProxy构造客户端代理对象，直接通过代理对象调用远程端的方法。

**4、Hadoop心跳机制**

<http://blog.csdn.net/paicmis/article/details/52675634>

<http://blog.csdn.net/fly542/article/details/6797139>

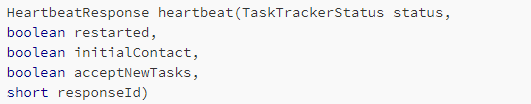
<http://www.cnblogs.com/sh425/p/6893528.html>

心跳是Jobtracker和Tasktracker的桥梁，它实际上是一个**RPC函数**，Tasktracker周期性的调用该函数汇报节点和任务状态信息，从而形成心跳。在hadoop中，心跳主要有三个作用：

* 判断Tasktracker是否活着；
* 及时让Jobtracker获取各个节点上的资源使用情况和任务运行状态；
* 为Tasktracker分配任务
* **注意：**Jobtracker与Tasktracker之间采用了**Pull而不是Push模型**，是Jobtracker不会主动向Tasktracker发送任何信息，而是由Tasktracker主动通过心跳领取属于自己的信息，Jobtracker只能通过心跳应答的形式为各个Tasktracker分配任务。

**1）RPC函数heartbeat**

Tasktracker周期性的调用RPC函数heartbeat向Jobtracker汇报信息和领取任务，这个函数定义是：



这些参数含义是：

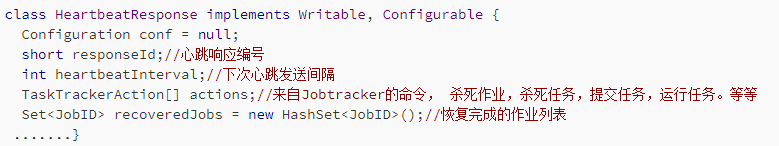
* **Status**：该类参数封装了Tasktracker上的各种状态信息，包括 ：
* String **trackerName**：//TaskTracker名称，形式如 tracker\_mymachine:localhost.

localdomain/127.0.0.1:34196

* String **host**：//TaskTracker主机名
* int **httpPort**：//TaskTracker对外的HTTP端口号
* int **failures**：//该TaskTracker上已经失败的任务数
* List **taskReports**：//正在运行的各个任务状态
* volatile long **lastSeen**：//上次汇报心跳的时间
* private int **maxMapTasks**：//Map slot总数，即允许同时运行的Map Task总数，有参数mapred.tasktracker.map.tasks.maximum设定
* private int **maxReduceTasks**：//Reduce slot 总数
* private TaskTrackerHealthStatus **healthStatus**：//Tasktracker健康状态
* private ResourceStatus **resStatus**：//Tasktracker 资源(内存 CPU等)信息
* **Restarted**：表示Tasktracker是否刚刚重新启动
* **initialContact**：表示TaskTracker是否初次连接Jobtracker
* **acceptNewTasks**：表示Tasktracker是否可以接收新任务，这通常取决于slot是否有剩余和节点健康状态等
* **responseId**：表示心跳响应编号，用于防止重复发送心跳，每接收一次心跳后，该值加1

**2）HeartbeatResponse 对象**

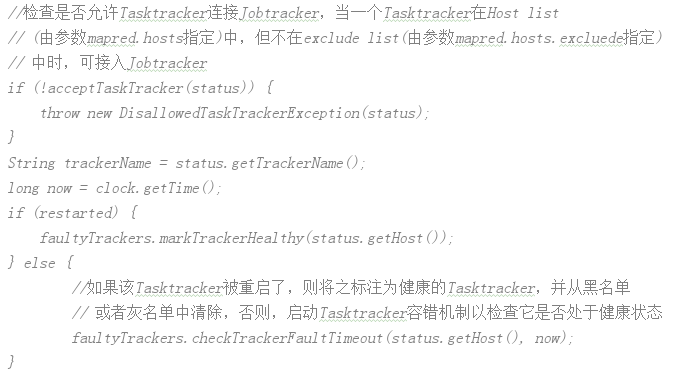
该函数的返回值为一个HeartbeatResponse 对象，该对象主要封装了Jobtracker向tasktracker下达的命令



该函数的内部实现逻辑主要分为两个步骤：更新状态和下达命令。Jobtracker首先将Tasktracker汇报的最新任务运行状态保存到相应的数据结构中，然后根据这些状态信息和外界需求为其下达相应的命令。

**A. 更新状态**

函数heartbeat首先会更新Tasktracker/job/task的状态信息。代码如下





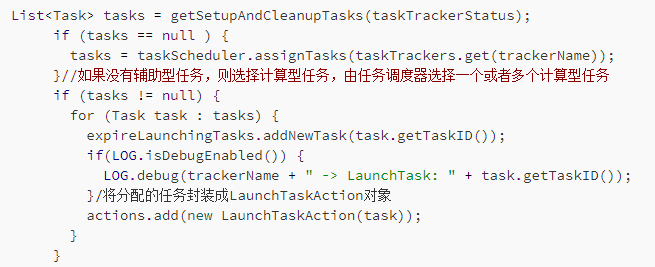
**B. 下达命令**

在Jobtracker更新状态以后，Jobtracker要为Tasktracker构造一个HeartbeatResponse对象作为心跳应答，该对象主要有2个部分内容下达给**Tasktracker的命令和下次心跳的时间。**

**下达命令** ：Jobtracker将下达给Tasktracker的命令封装成**TasktrackerAction类**，主要包括了ReinitTrackerAction(重新初始化)、LaunchTaskAction(运行新任务)、KillTaskAction(杀死任务)、KillJobAction(杀死作业)、CommitTaskAction(提交任务)五种。

* **ReinitTrackerAction**：Jobtracker接到Tasktracker发送过来的心跳信息后，首先要进行一致性检查。如果发现异常情况，则会要求Tasktracker重新对自己进行初始化，已恢复到一致性的状态。当出现下面2中不一致情况时，Jobtracker会向Tasktracker下达ReinitTrackerAction命令：
* 丢失上次心跳应答信息：Jobtracker会保存向每个Tasktracker发送的最近心跳应答信息，如果Jobtracker未刚刚重启,且一个Tasktracker并非初次连接Jobtracker(initialContact！=true)，而最近心跳应答丢失了；
* 丢失Tasktracker状态信息：jobTracker接收到任何一个心跳信息后，会将Tasktracker状态(封装在类TaskTrackerStatus中)信息保存起来，如果一个Tasktracker非初次连接Jobtracker，但状态信息却不存在；
* **LaunchTaskAction**：该类封装了Tasktracker新分配的任务，Tasktracker接收到该命令后会启动一个子进程运行该任务，Hadoop将一个作业分解后的任务分成2大类：**计算型任务和辅助型任务**。其中，计算型任务是处理实际数据的任务，包括Map Task和Reduce Task2种(对应的TaskType类中的MAP和REDUCE2种类型)，由专门的任务调度器对它们进行调度。而辅助型任务则不会处理实际的数据，通常用户同步计算型任务或者清理磁盘上无用的目录，包括job-setup task、job-cleanup task、和task-cleanup task三种(对应的ActionType是JOB\_SETUP JOB\_CLEANUP TASK\_CLEANUP)，其中job-setup task 和job-cleanup task 分别用作计算型任务开始运行同步标识和结束运行同步标识，而task-cleanup task 则用于清理失败计算型任务已经写到磁盘的部分结果，这种任务由Jobtracker负责调度，运行优先级高于计算型任务。

如果一个正常的Tasktracker尚有空闲的slot(acceptNewTasks=true)，则Jobtracker会为该Tasktracker分配新任务，任务顺序是先辅助型任务在计算型任务，选择顺序依次为job-cleanup task 、task-cleanup task 、job-setup task . **//优先选择辅助型任务**，选择优先级从高到低依次是 job-cleanup task 、task-cleanup task 、job-setup task这样可以要完成了的作业快速结束，提交的的作业立刻进入运行状态

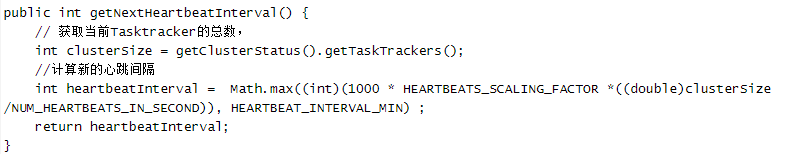


* **KillTaskAction**：该类封装了Tasktracker需杀死的任务，Tasktracker收到该命令后杀掉对应任务、清理工作目录和释放slot，**导致Jobtracker向Tasktracker发送该命令的原因有很多主要**：
* 用户使用 hadoop job -kill-task 或者 hadoop job -fail-task 杀死一个任务或者使一个任务失败
* 启用了**推测执行机制后**，同一份数据可能同时由2个task attempt 处理，当其中一个task attempt执行成功后，另一个处理相同数据的task attempt将会被杀掉
* 某个作业运行失败，它的其他任务将被杀掉
* Tasktracker在一定时间内**未汇报心跳**，则Jobtracker认为其死掉，它上面所有task均标注为死亡
* **KillJobAction**：该类封装了Tasktracker待清理的作业，tasktracker接收到该命令后，会清理作业的临时目录，导致Jobtracker向Tasktracker发送该命令的原因很多：
* 用户使用 hadoop job -kill 或者hadoop job -fail杀死一个作业或者使一个作业失败
* 作业完成，通知Tasktracker清理该作业的工作目录
* 作业运行失败，即同一个作业失败的task数目超过一定比例
* **CommitTaskAction**：该类封装了Tasktracker需要提交的任务。为了防止同一个TaskInProcess的2个同时运行的Task Attempt同时打开一个文件或者往一个文件中写数据而产生冲突，Hadoop让每个Task Attempt 写到单独一个文件中(以TaskAttemptID命名，比如attempt\_201606092312\_0008\_r\_000000\_0)中，通常而言，hadoop让每个Task Attempt将计算结果写到临时目录{mapred.output.dir}/temporary/\_{taskid}中，当某个task Attempt成功运行完成，再将运算结果转移到最终目录${mapred.output.dir}中，hadoop将一个成功运行完成的task Attempt结果文件从临时目录“提升”至最终目录的过程，称为“提交任务”。当TaskInProgress中一个任务被提交后，其他任务将会被杀死，同时意味着该TaskInProgress运行完成。

**C. 调整心跳间隔**

Tasktracker心跳时间间隔大小应该适度，如果太小，则Jobtracker需要处理高并发的心跳连接请求，必然产生不小的并发压力，如果太大，空闲的资源不能及时汇报给Jobtracker(进而不能为tasktracker分配任务)，造成资源浪费，进而降低了系统的吞吐率。Tasktracker汇报心跳的时间间隔并不是一成不变的，它会随着集群的规模的动态调整而变化，以便能够合理利用Jobtracker的并发处理能力，在hadoop mapreduce中，**只有Jobtracker知道某一时刻集群的规模**，因此由Jobtracker为每个Tasktracker计算下一次汇报心跳的时间间隔，并通过心跳机制告诉Tasktracker。

Jobtracker允许用户通过参数配置心跳的时间间隔加速比，即每增加mapred.heartbeats.in.second(默认是100，最小1)个节点心跳时间间隔增加mapreduce.jobtracker.heartbeats.scaling.factor(默认是1，最小是0.01)秒。同时为了防止用户参数设置不合理而对Jobtracker产生较大的负载，**Jobtracker要求心跳时间间隔至少为3秒**。



**心跳机制：**

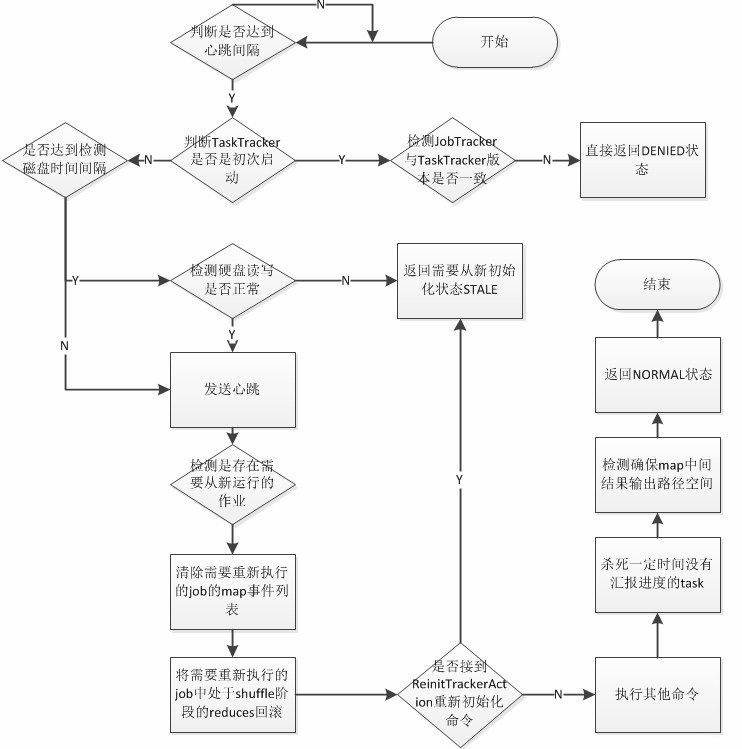
1、hadoop集群是master/slave模式，master包括Namenode和Jobtracker，slave包括Datanode和Tasktracker。

2、master启动的时候，会开一个**ipc server**在那里，等待slave心跳。

3、slave启动时，会连接master，并每隔3秒钟主动向master发送一个“心跳”，这个时间可以通过”heartbeat.recheck.interval”属性来设置。将自己的状态信息告诉master，然后master也是通过这个心跳的返回值，向slave节点传达指令。

4、需要指出的是：namenode与datanode之间的通信，jobtracker与tasktracker之间的通信，都是通过“心跳”完成的。

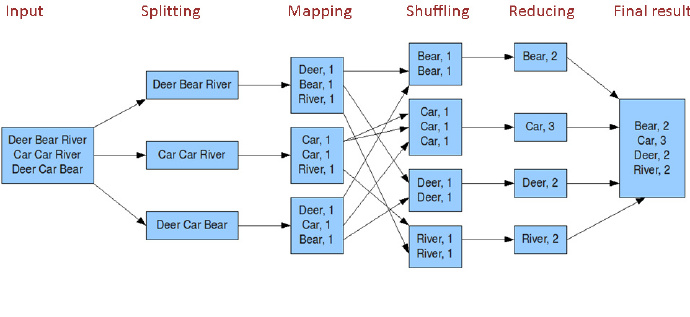
从整个源码看，TaskTracker向JobTracker发送一次心跳的流程如下：

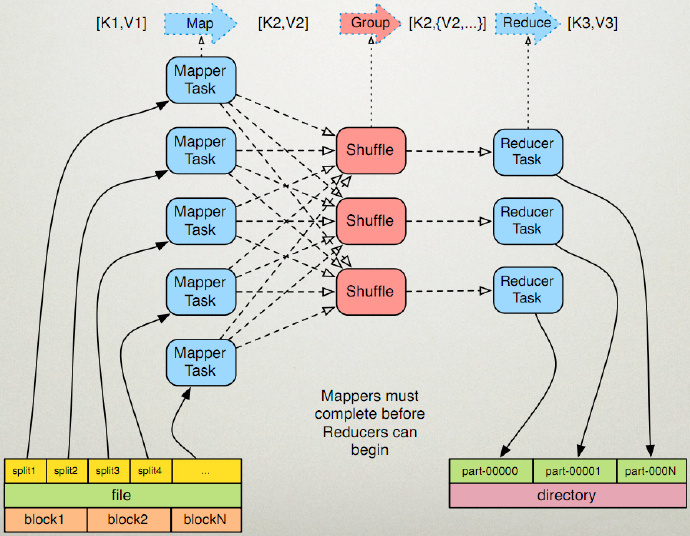


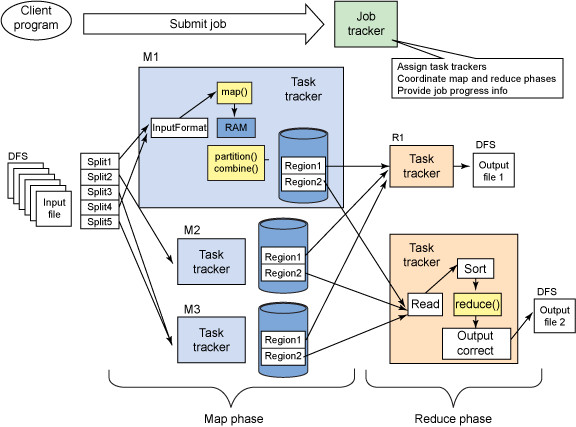
**5、mapreduce运行机制**

<http://blog.jobbole.com/84089/>

<http://www.cnblogs.com/wujing-hubei/p/6009849.html>







参入mapreduce作业执行涉及4个独立的实体：

* **客户端（client）**：编写mapreduce程序，配置作业，提交作业，这就是程序员完成的工作；
* **JobTracker**：初始化作业，分配作业，与TaskTracker通信，协调整个作业的执行；
* **TaskTracker**：保持与JobTracker的通信，在分配的数据片段上执行Map或Reduce任务，TaskTracker和JobTracker的不同有个很重要的方面，就是在执行任务时候TaskTracker可以有n多个，JobTracker则只会有一个（JobTracker只能有一个就和hdfs里namenode一样存在单点故障，我会在后面的mapreduce的相关问题里讲到这个问题的）
* **Hdfs**：保存作业的数据、配置信息等等，最后的结果也是保存在hdfs上面

**1）那么mapreduce到底是如何运行的呢？**

首先是客户端要编写好mapreduce程序，配置好mapreduce的**作业**也就是**job**，接下来就是提交job了，提交job是提交到JobTracker上的，这个时候JobTracker就会构建这个job，具体就是分配一个新的job任务的ID值，接下来它会做**检查操作**，这个检查就是确定输出目录是否存在，如果存在那么job就不能正常运行下去，JobTracker会抛出错误给客户端，接下来还要检查输入目录是否存在，如果不存在同样抛出错误，如果存在JobTracker会根据输入计算输入分片（Input Split），如果分片计算不出来也会抛出错误，至于输入分片我后面会做讲解的，这些都做好了JobTracker就会配置Job需要的资源了。分配好资源后，JobTracker就会初始化作业，*初始化主要做的是将Job放入一个内部的队列，让配置好的作业调度器能调度到这个作业，作业调度器会初始化这个job，初始化就是创建一个正在运行的job对象（封装任务和记录信息），以便JobTracker跟踪job的状态和进程*。

初始化完毕后，作业调度器会获取输入分片信息（input split），每个分片创建一个map任务。接下来就是任务分配了，这个时候tasktracker会运行一个简单的循环机制定期发送心跳给jobtracker，心跳间隔是3秒，程序员可以配置这个时间，心跳就是jobtracker和tasktracker沟通的桥梁，通过心跳，jobtracker可以监控tasktracker是否存活，也可以获取tasktracker处理的状态和问题，同时tasktracker也可以通过心跳里的返回值获取jobtracker给它的操作指令。任务分配好后就是执行任务了。在任务执行时候jobtracker可以通过心跳机制监控tasktracker的状态和进度，同时也能计算出整个job的状态和进度，而tasktracker也可以本地监控自己的状态和进度。当jobtracker获得了最后一个完成指定任务的tasktracker操作成功的通知时候，jobtracker会把整个job状态置为成功，然后当客户端查询job运行状态时候（注意：这个是**异步操作**），客户端会查到job完成的通知的。如果job中途失败，mapreduce也会有相应机制处理，一般而言如果不是程序员程序本身有bug，mapreduce错误处理机制都能保证提交的job能正常完成。

从**逻辑实体**的角度**讲解mapreduce运行机制**，这些按照时间顺序包括：输入分片（input split）、map阶段、combiner阶段、shuffle阶段和reduce阶段。

* **输入分片（input split）：**在进行map计算之前，mapreduce会根据输入文件计算输入分片（input split），每个输入分片（input split）针对一个map任务，输入分片（input split）存储的并非数据本身，而是一个分片长度和一个记录数据的位置的数组，输入分片（input split）往往和hdfs的block（块）关系很密切，假如我们设定hdfs的块的大小是64mb，如果我们输入有三个文件，大小分别是3mb、65mb和127mb，那么mapreduce会把3mb文件分为一个输入分片（input split），65mb则是两个输入分片（input split）而127mb也是两个输入分片（input split），换句话说我们如果在map计算前做输入分片调整，例如合并小文件，那么就会有5个map任务将执行，而且每个map执行的数据大小不均，这个也是**mapreduce优化计算**的一个关键点。
* **map阶段：**就是程序员编写好的map函数了，因此map函数效率相对好控制，而且一般map操作都是**本地化操作**也就是在数据存储节点上进行；
* **combiner阶段：**combiner阶段是程序员可以选择的，combiner其实也是一种reduce操作，因此我们看见WordCount类里是用reduce进行加载的。**Combiner是一个本地化的reduce操作**，它是map运算的后续操作，主要是在map计算出**中间文件**前做一个简单的合并重复key值的操作，例如我们对文件里的单词频率做统计，map计算时候如果碰到一个hadoop的单词就会记录为1，但是这篇文章里hadoop可能会出现n多次，那么map输出文件冗余就会很多，因此在reduce计算前对相同的key做一个合并操作，那么文件会变小，这样就提高了宽带的传输效率，毕竟hadoop计算力宽带资源往往是计算的瓶颈也是最为宝贵的资源，但是combiner操作是有风险的，使用它的原则是combiner的输入不会影响到reduce计算的最终输入，例如：如果计算只是求总数，最大值，最小值可以使用combiner，但是做平均值计算使用combiner的话，最终的reduce计算结果就会出错。
* **shuffle阶段：**将map的输出作为reduce的输入的过程就是shuffle了，这个是mapreduce优化的重点地方。这里我不讲怎么优化shuffle阶段，讲讲**shuffle阶段的原理**，因为大部分的书籍里都没讲清楚shuffle阶段。Shuffle一开始就是map阶段做输出操作，一般mapreduce计算的都是海量数据，map输出时候不可能把所有文件都放到**内存**操作，因此map写入**磁盘**的过程十分的复杂，更何况map输出时候要对结果进行排序，内存开销是很大的，map在做输出时候会在内存里开启一个环形内存缓冲区，这个缓冲区专门用来输出的，默认大小是100mb，并且在配置文件里为这个缓冲区设定了一个阀值，默认是0.80（这个大小和阀值都是可以在配置文件里进行配置的），同时map还会为输出操作启动一个**守护线程**，如果缓冲区的内存达到了阀值的80%时候，这个守护线程就会把内容写到磁盘上，这个过程叫spill，另外的20%内存可以继续写入要写进磁盘的数据，**写入磁盘和写入内存操作是互不干扰的**，如果缓存区被撑满了，那么map就会阻塞写入内存的操作，让写入磁盘操作完成后再继续执行写入内存操作，前面我讲到写入磁盘前会有个**排序操作**，这个是在写入磁盘操作时候进行，不是在写入内存时候进行的，如果我们定义了combiner函数，那么排序前还会执行combiner操作。

每次spill操作也就是写入磁盘操作时候就会写一个溢出文件，也就是说在做map输出有几次spill就会产生多少个溢出文件，等map输出全部做完后，map会合并这些输出文件。这个过程里还会有一个Partitioner操作，对于这个操作很多人都很迷糊，其实Partitioner操作和map阶段的输入分片（Input split）很像，一个Partitioner对应一个reduce作业，如果我们mapreduce操作只有一个reduce操作，那么Partitioner就只有一个，如果我们有多个reduce操作，那么Partitioner对应的就会有多个，**Partitioner因此就是reduce的输入分片**，这个程序员可以编程控制，主要是根据实际key和value的值，根据实际业务类型或者为了更好的reduce负载均衡要求进行，这是提高reduce效率的一个关键所在。到了reduce阶段就是合并map输出文件了，Partitioner会找到对应的map输出文件，然后进行复制操作，复制操作时reduce会开启几个复制线程，这些线程默认个数是5个，程序员也可以在配置文件更改复制线程的个数，这个复制过程和map写入磁盘过程类似，也有阀值和内存大小，阀值一样可以在配置文件里配置，而内存大小是直接使用reduce的tasktracker的内存大小，复制时候reduce还会进行排序操作和合并文件操作，这些操作完了就会进行reduce计算了。

* **reduce阶段：**和map函数一样也是程序员编写的，最终结果是存储在hdfs上的

**2）Mapreduce的相关问题**

① jobtracker的单点故障：jobtracker和hdfs的namenode一样也存在单点故障，单点故障一直是hadoop被人诟病的大问题，为什么hadoop的文件系统和mapreduce计算框架都是高容错的，但是最重要的管理节点的故障机制却如此不好，我认为主要是namenode和jobtracker在实际运行中都是在内存操作，而做到内存的容错就比较复杂了，只有当内存数据被持久化后容错才好做，namenode和jobtracker都可以备份自己持久化的文件，但是这个持久化都会有延迟，因此真的出故障，任然不能整体恢复，另外hadoop框架里包含zookeeper框架，zookeeper可以结合jobtracker，用几台机器同时部署jobtracker，保证一台出故障，有一台马上能补充上，不过这种方式也没法恢复正在跑的mapreduce任务。

② 做mapreduce计算时候，输出一般是一个文件夹，而且该文件夹是不能存在，我在出面试题时候提到了这个问题，而且这个检查做的很早，当我们提交job时候就会进行，mapreduce之所以这么设计是保证数据可靠性，如果输出目录存在，reduce就搞不清楚你到底是要追加还是覆盖，不管是追加和覆盖操作都会有可能导致最终结果出问题，mapreduce是做海量数据计算，一个生产计算的成本很高，例如一个job完全执行完可能要几个小时，因此一切影响错误的情况mapreduce是零容忍的。

③ Mapreduce还有一个InputFormat和OutputFormat，我们在编写map函数时候发现map方法的参数是之间操作行数据，没有牵涉到InputFormat，这些事情在我们new Path时候mapreduce计算框架帮我们做好了，而OutputFormat也是reduce帮我们做好了，我们使用什么样的输入文件，就要调用什么样的InputFormat，InputFormat是和我们输入的文件类型相关的，mapreduce里常用的InputFormat有FileInputFormat普通文本文件，SequenceFileInputFormat是指hadoop的序列化文件，另外还有KeyValueTextInputFormat。OutputFormat就是我们想最终存储到hdfs系统上的文件格式了

**6、HDFS数据的读写过程**

<http://blog.csdn.net/u014470581/article/details/51461540>

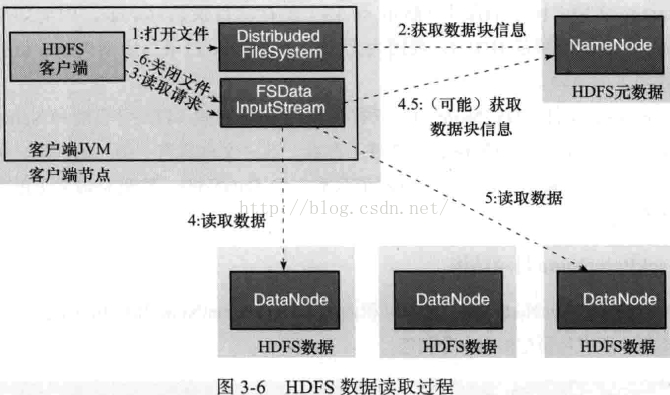
<http://blog.csdn.net/ruidongliu/article/details/8815006>

<http://blog.csdn.net/mmd0308/article/details/75003494>

<http://blog.csdn.net/u011239443/article/details/51751462>

**1）读数据**

一般的文件读取操作包括：open 、read、close等。客户端读取数据过程，其中1、3、6步由客户端发起。用户首先获取FileSystem的一个实例，这里就是HDFS对应的实例：



* ①客户端调用FileSystem实例的open方法，获得这个**文件对应的输入流**，在HDFS中就是DFSInputStream；
* ②构造第一步中的输入流DFSInputStream时，通过RPC远程调用NameNode可以获得NameNode中此文件对应的数据块的保存位置，包括这个**文件的副本的保存位置**（主要是各DataNode的地址）。注意，在输入流中会按照网络拓扑结构，根据与客户端距离对DataNode进行简单排序；
* ③④获得此输入流之后，客户端调用read方法读取数据。输入流DFSInputStream会根据前面的**排序结果**，选择最近的DataNode建立连接并读取数据。如果客户端和其中一个DataNode位于一个机器中（比如MapReduce过程中的mapper和reducer），那么就会直接**从本地读取数据**。
* ⑤如果已达到数据块末端，那么关闭与这个DataNode的连接，然后重新查找下一个数据块。不断执行②-⑤直到数据全部读完，然后调用close；
* ⑥客户端调用close，关闭输入流DFSInputStream。
* 另外如果DFSInputStream和DataNode通信时出现错误，或者是数据校验出错，那么DFSInputStream就会重新选择DataNode传输数据。

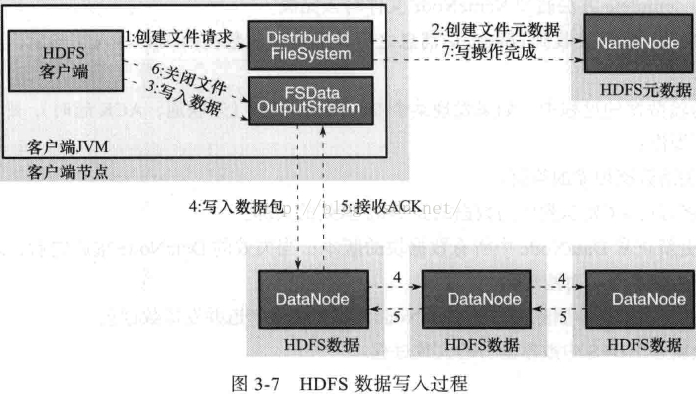
或者：

文件读取的过程如下：

* 使用HDFS提供的客户端开发库，向远程的Namenode发起RPC请求；
* Namenode会视情况返回文件的部分或者全部**block列表**，对于每个block，Namenode都会返回持有该block拷贝的datanode地址；
* 客户端开发库会选取离客户端最接近的datanode来读取block；
* 读取完当前block的数据后，关闭与当前的datanode连接，并为读取下一个block寻找最佳的datanode；
* 当读完列表的block后，且文件读取还没有结束，客户端开发库会继续向Namenode获取下一批的block列表。
* 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拷贝的datanode继续读。

**2）写数据**

数据的写入过程 一般文件写入操作不外乎create、write、close几种。客户端写入数据过程，其中1、3、6由客户端发起，客户端首先要获取FileStream的一个实例，这里就是HDFS的实例：



* ①②客户端调用**FileSystem实例的create方法**，创建文件。NameNode通过检查，比如文件是否存在，客户端是否拥有创建权限等；通过检查之后，在NameNode添加文件信息。**注意**，因为此时文件没有数据，所以NameNode上也没有文件数据块信息。创建结束后，HDFS会返回一个输出流DFSDataOutputStream给客户端。
* ③客户端调用输出流DFSDataOutputStream的write方法向HDFS中对应的文件写入数据。数据首先会被分包，这些分包会写入一个输入流内部队列Data队列中，接收完整数据分包，输出流DFSDataOutputStream会向nameNode申请保存文件和副本数据块的若干个DataNode，这若干个DataNode会形成一个数据传输管道。
* ④DFSDataOutputStream会（根据网络拓扑结构排序）将数据传输给距离上最短的DataNode，这个DataNode接收到数据包之后会传递给下一个DataNode，数据在各DataNode之间通过管道流动，而不是全部由输出流分发，这样可以减少传输开销。
* ⑤因为DataNode位于不同机器上，数据需要通过网络发送，所以，为了保证所有的DataNode的数据都是准确的，接收到数据的DataNode要向发送者发送确认包（ACKPacket）。对于某个数据块，只有当DFSDataOutputStream收到了所有DataNode的正确ACK，才能确认传输结束。DFSDataOutputStream内部专门维护了一个等待ACK队列，这一队列保存已经进入管道传输数据、但是并未被完全确认的数据包。
* 不断③-⑤直到数据全部写完，客户端调用close关闭文件。
* ⑥客户端调用close方法，DFSDataOutputStream继续等待直到所有数据写入完毕并被确认，调用complete方法通知NameNode文件写入完成。
* ⑦NameNode接收到complete消息之后，等待相应数量的副本写入完毕后，告知客户端即可。

或者

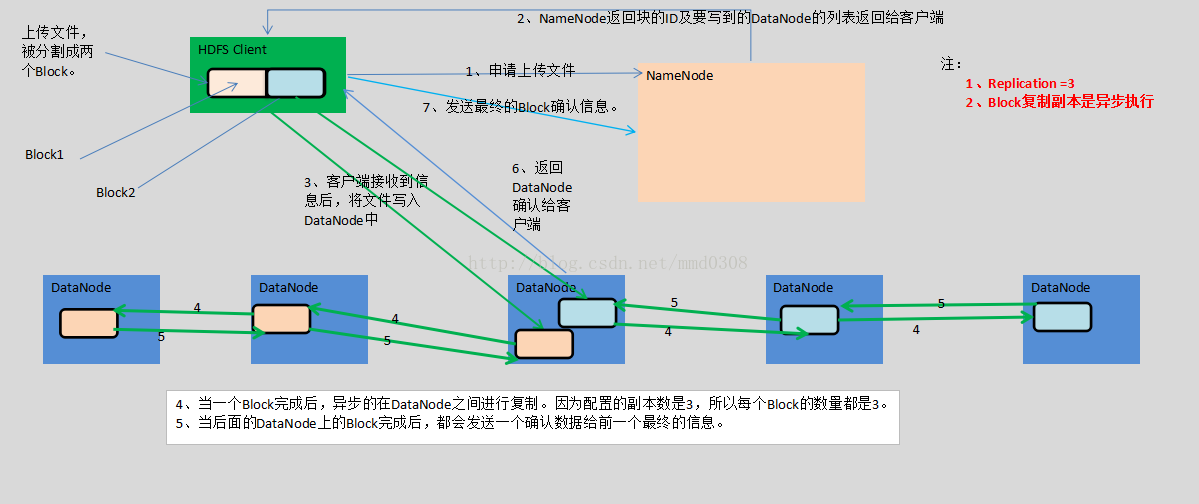
写入文件的过程比读取较为复杂：

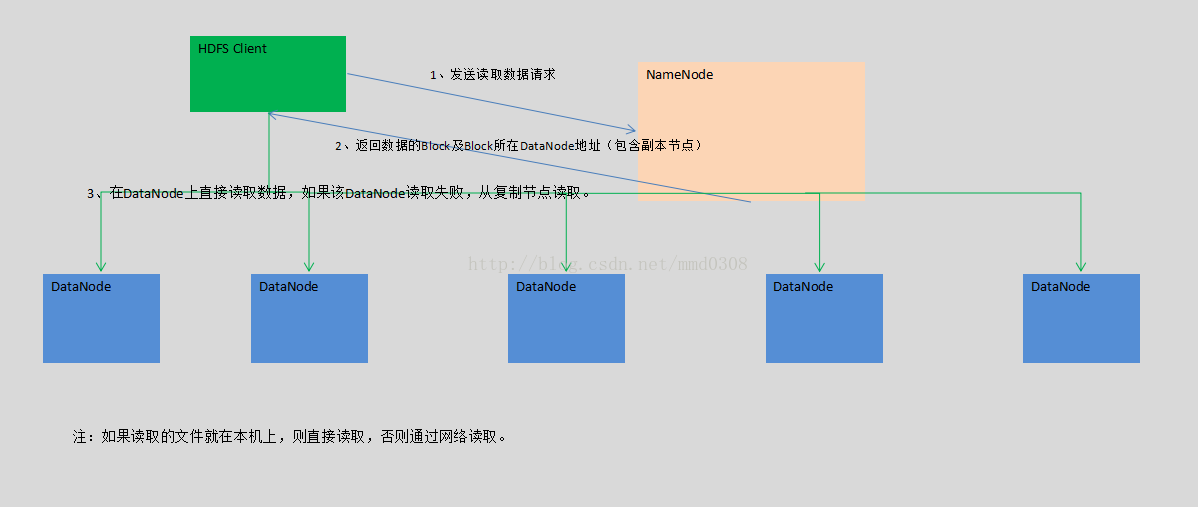
* 使用HDFS提供的客户端开发库，向远程的Namenode发起**RPC请求**；
* Namenode会检查要创建的文件是否已经存在，创建者是否有权限进行操作，成功则会为文件创建一个记录，否则会让客户端抛出异常；
* 当客户端开始写入文件的时候，开发库会将文件切分成多个packets（信息包），并在内部以"data queue"的形式管理这些packets，并向Namenode申请新的blocks，获取用来存储replicas（复制品）的合适的datanodes列表，列表的大小根据在Namenode中对replication的设置而定。
* 开始以pipeline（**管道**）的形式将packet写入所有的replicas中。开发库把packet以流的方式写入第一个datanode，该datanode把该packet存储之后，再将其传递给在此pipeline（管道）中的下一个datanode，直到最后一个datanode，这种写数据的方式呈流水线的形式。
* 最后一个datanode成功存储之后会返回一个ack packet，在pipeline里传递至客户端，在客户端的开发库内部维护着"ack queue"，成功收到datanode返回的ack packet后会从"ack queue"移除相应的packet。
* 如果传输过程中，有某个datanode出现了故障，那么当前的pipeline会被关闭，出现故障的datanode会从当前的pipeline中移除，剩余的block会继续剩下的datanode中继续以pipeline的形式传输，同时Namenode会分配一个新的datanode，保持replicas设定的数量。

**3）注意**

在传输过程中，如果发现某个DataNode失效（未联通，ACK超时），那么HDFS执行如下操作：

* ①关闭数据传输的管道
* ②将等待ACK队列中的数据放到Data队列的头部
* ③更新正常DataNode中所有数据块的版本；当失效的DataNode重启之后，之前的数据块会因为版本不对而被清除。
* ④在传输管道中删除失效DataNode，重新建立管道并发送数据包。





**7、HDFS分布式文件系统NameNode和Secondary NameNode**

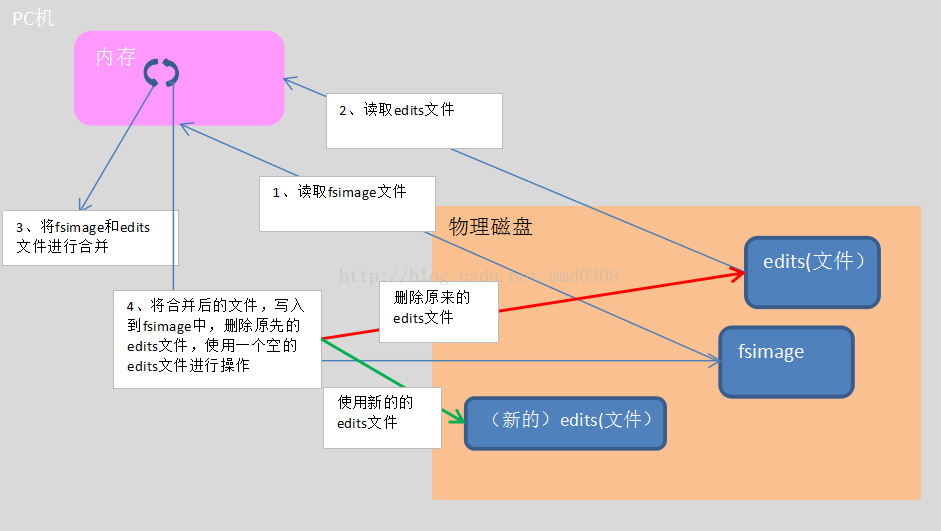
<http://blog.csdn.net/mmd0308/article/details/74674524>

**1）NameNode启动时如何维护元数据**

* **Edits文件：**NameNode在本地操作系统的文件都会保存在**Edits日志文件**中。也就是说当文件系统中的任何元数据产生操作时，都会记录在Edits日志文件中。eg：在HDFS上创建一个文件，NameNode就会在Edits中插入一条记录。同样如果修改或者删除等操作，也会在Edits日志文件中新增一条数据。
* **FsImage映像文件：**整个文件系统的名字空间，包括数据块到文件的映射，文件的属性等等，都存储在一个称为FsImage的文件中，这个文件也是放在NameNode所在的文件系统中。

流程介绍：

* ①加载fsimage映像文件到内存
* ②加载edits文件到内存
* ③在内存将fsimage映像文件和edits文件进行合并
* ④将合并后的文件写入到fsimage中
* ⑤清空原先edits中的数据，使用一个空的edits文件进行正常操作



因为NameNode只有在启动的阶段才合并fsimage和edits，那么如果运行时间长了，edits文件可能会越来越大，在下一次启动NameNode时会花很长的时间，请问能否让fsimage映像文件和edits日志文件**定期合并呢？SecondaryNameNode**

**2）Secondary NameNode工作流程**

Secondary NameNode和NameNode的**区别：**

* NameNode：

①、存储文件的metadata，运行时所有数据都保存在内存中，这个的HDFS可存储的文件受限于NameNode的内存。

②、NameNode失效则整个HDFS都失效了，所以要保证NameNode的可用性。

* Secondary NameNode：

①、定时与NameNode进行同步，定期的将fsimage映像文件和Edits日志文件进行合并，并将合并后的传入给NameNode，替换其镜像，并清空编辑日志。如果NameNode失效，需要手动的将其设置成主机。

②、Secondary NameNode**保存最新检查点checkpoints的目录**和NameNode的目录结构相同。所以NameNode可以在需要的时候应用Secondary NameNode上的检查点镜像。

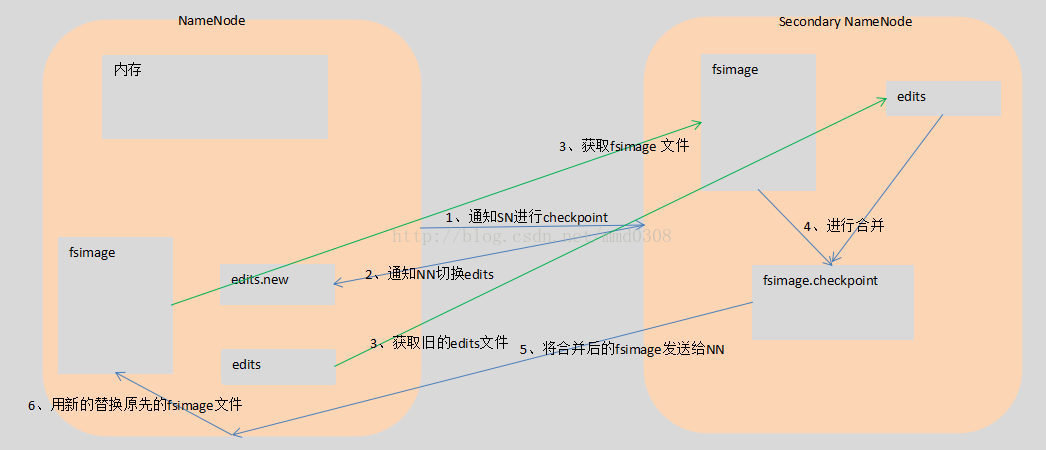
**什么时候checkpoint：**

①、连续两次的检查点最大时间间隔，默认是3600秒，可以通过配置“fs.checkpoint.period”进行修改

②、Edits日志文件的最大值，如果超过这个值就会进行合并即使不到1小时也会进行合并。可以通过“fs.checkpoint.size”来配置，默认是64M；

**工作流程：**

* ①、NameNode通知Secondary NameNode进行checkpoint。
* ②、Secondary NameNode通知NameNode切换edits日志文件，使用一个空的。
* ③、Secondary NameNode通过Http获取NmaeNode上的fsimage映像文件和切换前的edits日志文件。
* ④、Secondary NameNode在内容中合并fsimage和Edits文件。
* ⑤、Secondary NameNode将合并之后的fsimage文件发送给NameNode。
* ⑥、NameNode用Secondary NameNode 传来的fsImage文件替换原先的fsImage文件。



**8、HDFS可靠性保障**

<http://blog.csdn.net/login_sonata/article/details/54801232>

HDFS具备了较为完善的**冗余备份和故障恢复机制**，可以实现在集群中可靠地存储海量文件。

* **冗余备份：**为了容错，文件的所有数据块都可以有副本（replication默认3个）。当DataNode启动的时候，它会遍历本地文件系统，产生一份HDFS数据块和本地文件对应关系的列表，并把这个报告发送给NameNode，这就是报告块（BlockReport），报告块上包含了DataNode上所有块的列表。
* **副本存放：**HDFS集群一般运行在多个机架上，不同机架上机器的通信需要通过交换机，机架内节点之间的带宽比跨机架节点之间的带宽要大。**HDFS的存放策略**是将一个副本存放在本地机架节点上，一个副本存放在同一个机架的另一个节点上，最后一个副本放在不同机架的节点上。这种**机架感知（Rack-aware）**的策略减少了机架间的数据传输，提高了写操作的效率。机架的错误远远比节点的错误少，所以这种策略不会影响到数据的可靠性和可用性。
* **心跳检测：**NameNode周期性地从集群中的每个DataNode接受心跳包和块报告，如果DataNode不能发送心跳信息，NameNode会标记最近没有心跳的DataNode为**宕机**，并且不会给他们发送任何I/O请求，同时**采取数据备份措施**。
* **数据完整性检测：**NameNode在创建HDFS文件时，会计算每个数据的校验和并储存起来。当客户端从DataNode获取数据时，他会将获取的**数据的校验和**与之前**储存的校验和**进行对比。
* **空间回收：**从HDFS中删除的文件会首先被放入到/trash中，/trash文件夹中的内容是被删除文件最后的副本，该文件夹会被定时清空。该文件夹中不存在的文件就彻底不存在了。
* **安全模式：**HDFS启动时，会进入安全模式，此时不允许写操作。这时，NameNode会收到所有DataNode节点的数据块报告，在确认安全之后，系统自动退出安全模式。
* **核心文件备份：**HDFS的核心文件是映像文件（FsImage）和事务日志（Edit），系统支持对这两个文件的备份，以确保NameNode宕机后的恢复。
* **快照**

**9、Hadoop作业提交方式**

<http://blog.csdn.net/gaokao2011/article/details/41682515>

客户端提交任务的时候，使用方法job.setWaitForCompletion(true)的时候，这个方法会调用**job的submit()**方法，submit()方法又会调用**jobclient的submitJobInternal(conf)**方法向master提交一个任务，这个方法会向hadoop文件系统提交三个文件job.jar，job.split和job.xml那么它报那个错误就是因为job.jar文件没有提交上去（报jar file not found 的错误），也就是说我们像在伪分布下那样提交任务的话我们的jar包并没有打好，所以就需要我们想办法将jar包提交上去。

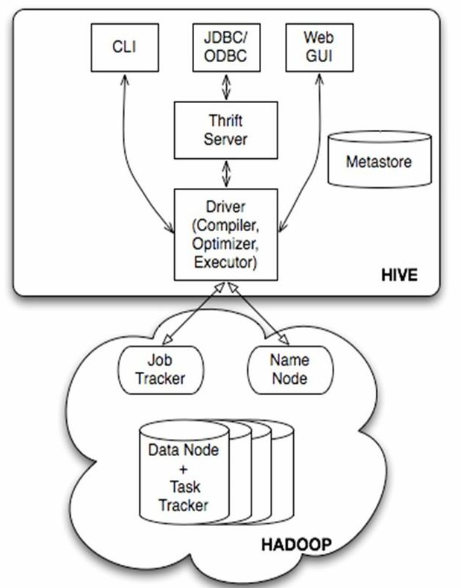
**二、HBase知识**

**三、Hive知识**

**1、hive架构及运行过程**

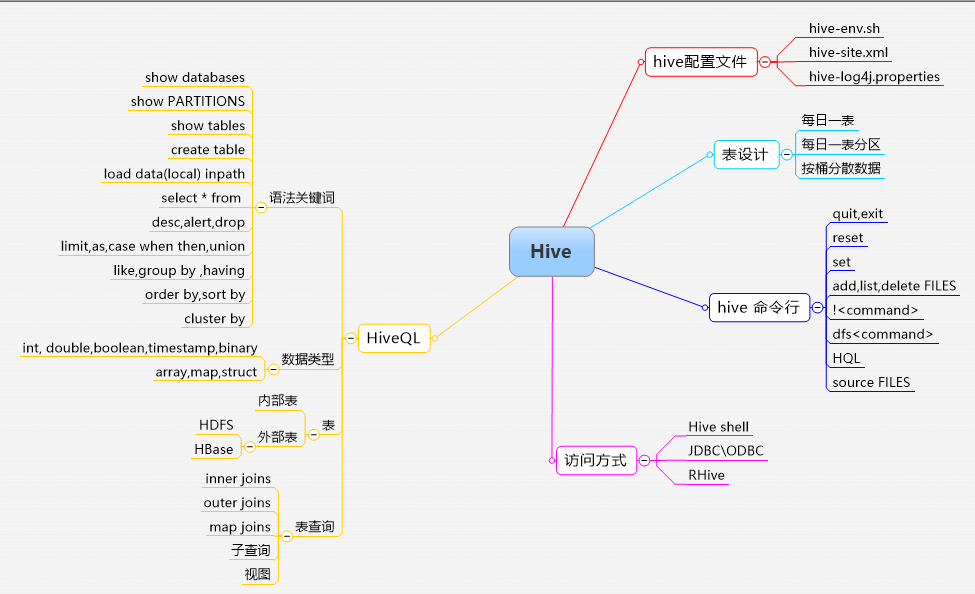
<http://www.cnblogs.com/tgzhu/p/5759610.html>

* **hive架构：**

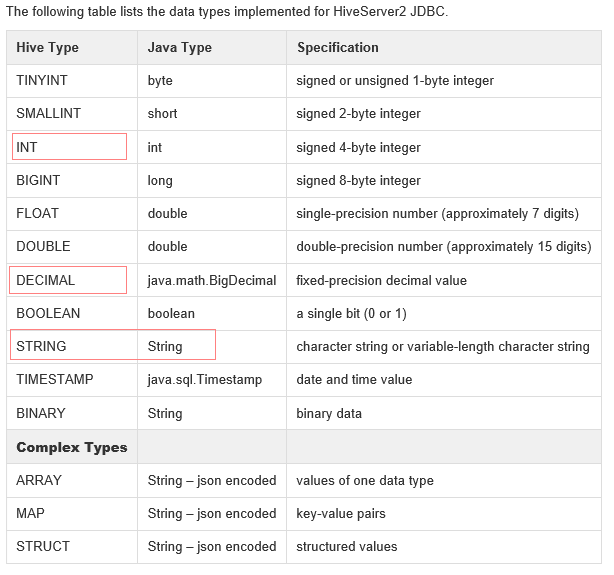


* **用户接口**：CLI（命令行）、JDBC/ODBC客户端、web GUI
* **metaStore**: hive 的元数据结构描述信息库，可选用不同的关系型数据库来存储，通过配置文件修改、查看数据库配置信息
* **Driver**: 解释器、编译器、优化器完成HQL查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在HDFS中，并在随后由MapReduce调用执行
* Hive的数据存储在HDFS中，大部分的查询、计算由MapReduce完成
* **知识体系**：

包含shell命令语法、HiveQl语法、访问方式等，如下图：



* **数据类型：**

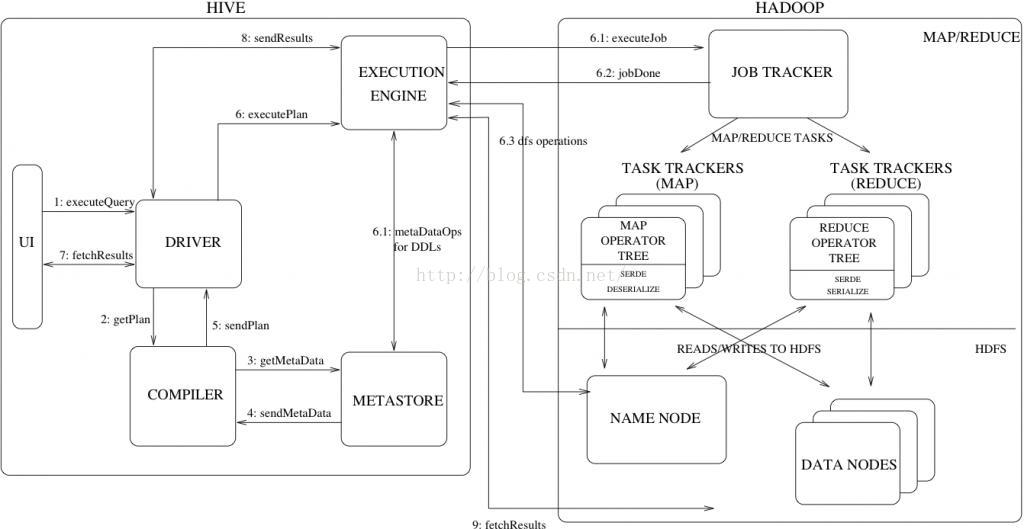


**Hive SQL的编译过程：**

<https://tech.meituan.com/hive-sql-to-mapreduce.html>

**1）Hive的工作原理**

<http://blog.csdn.net/wangyang1354/article/details/50570903>



**流程大致步骤为：**

1. 用户提交查询等任务给Driver。
2. 编译器获得该用户的任务Plan。
3. 编译器Compiler根据用户任务去MetaStore中获取需要的Hive的元数据信息。
4. 编译器Compiler得到元数据信息，对任务进行编译，先将HiveQL转换为抽象语法树，然后将抽象语法树转换成查询块，将查询块转化为逻辑的查询计划，重写逻辑查询计划，将逻辑计划转化为物理的计划（MapReduce）, 最后选择最佳的策略。
5. 将最终的计划提交给Driver。
6. Driver将计划Plan转交给ExecutionEngine去执行，获取元数据信息，提交给JobTracker或者SourceManager执行该任务，任务会直接读取HDFS中文件进行相应的操作。
7. 获取执行的结果。
8. 取得并返回执行结果。

**2、Hive数据倾斜**

<http://blog.csdn.net/s646575997/article/details/51510661>

<http://blog.csdn.net/guohecang/article/details/51567434>

在做Shuffle阶段的优化过程中，遇到了**数据倾斜的问题**，造成了对一些情况下优化效果不明显。主要是因为在Job完成后的所得到的Counters（计数器）是整个Job的总和，优化是基于这些Counters得出的平均值，而由于数据倾斜的原因造成map处理数据量的差异过大，使得这些平均值能代表的价值降低。Hive的执行是**分阶段**的，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。规避错误来更好的运行比解决错误更高效。

**1）数据倾斜的原因**

* **操作：**



* **原因：**
* key分布不均匀；
* 业务数据本身的特性；
* 建表时考虑不周；
* 某些SQL语句本身就有数据倾斜；
* **表现：**任务进度长时间维持在99%（或100%），查看任务监控页面，发现只有少量（1个或几个）reduce子任务未完成。因为其处理的数据量和其他reduce差异过大。单一reduce的记录数与平均记录数差异过大，通常可能达到3倍甚至更多。最长时长远大于平均时长。

**2）数据倾斜解决方案**

**A. 参数调节**

* hive.map.aggr=true：Map 端部分聚合，相当于Combiner
* hive.groupby.skewindata=true：有数据倾斜的时候进行负载均衡，当选项设定为 true，生成的查询计划会有两个 MR Job。第一个 MR Job 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 有可能被分发到不同的 Reduce 中，从而达到**负载均衡**的目的；第二个 MR Job 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 被分布到同一个 Reduce 中），最后完成最终的聚合操作。

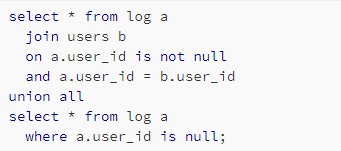
**B. SQL语句调节**

* **如何Join**：关于驱动表的选取，选用join key分布最均匀的表作为驱动表。做好列裁剪和filter操作，以达到两表做join的时候，数据量相对变小的效果；
* **大小表Join**：使用map join让小的维度表（1000条以下的记录条数） 先进内存。在map端完成reduce；
* **大表Join大表**：把空值的key变成一个字符串加上随机数，把倾斜的数据分到不同的reduce上，由于null值关联不上，处理后并不影响最终结果；
* **count distinct大量相同特殊值**：count distinct时，将值为空的情况单独处理，如果是计算count distinct，可以不用处理，直接过滤，在最后结果中加1。如果还有其他计算，需要进行group by，可以先将值为空的记录单独处理，再和其他计算结果进行union；
* **group by维度过小**：采用sum() group by的方式来替换count(distinct)完成计算；
* **特殊情况特殊处理**：在业务逻辑优化效果的不大情况下，有些时候是可以将倾斜的数据单独拿出来处理。最后union回去。

**3）典型的业务场景**

**A. 空值产生的数据倾斜**

* **场景**：如日志中，常会有信息丢失的问题，比如日志中的 user\_id，如果取其中的 user\_id 和 用户表中的user\_id 关联，会碰到数据倾斜的问题。
* **解决方法1**： user\_id为空的不参与关联



* **解决方法2** ：赋与空值分新的key值

select \* from log a left outer join users b on **case when** a.user\_id is null **then** concat(‘hive’,rand()) **else** a.user\_id = b.user\_id end;

* **结论：**方法2比方法1效率更好，不但io少了，而且作业数也少了。解决方法1中 log读取两次，jobs是2。解决方法2 job数是1 。这个优化适合无效 id (比如 -99 , ’’, null 等) 产生的倾斜问题。把空值的 key 变成一个字符串加上随机数，就能把倾斜的数据分到不同的reduce上 ,解决数据倾斜问题。

**B. 不同数据类型关联产生数据倾斜**

* **场景**：用户表中user\_id字段为int，log表中user\_id字段既有string类型也有int类型。当按照user\_id进行两个表的Join操作时，默认的Hash操作会按int型的id来进行分配，这样会导致所有string类型id的记录都分配到一个Reducer中。
* **解决方法**：把数字类型转换成字符串类型：

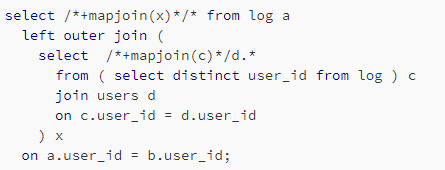
select \* from users a left outer join logs b on a.usr\_id = cast(b.user\_id as string)

**C. 小表不小不大，怎么用 map join 解决倾斜问题**

使用 map join 解决小表(记录数少)关联大表的数据倾斜问题，这个方法使用的频率非常高，但如果小表很大，大到map join会出现bug或异常，这时就需要特别的处理。 以下例子：select \* from log a left outer join users b on a.user\_id = b.user\_id;

users 表有 600w+ 的记录，把 users 分发到所有的 map 上也是个不小的开销，而且 map join 不支持这么大的小表。如果用普通的 join，又会碰到数据倾斜的问题。

**解决方法：**



假如，log里user\_id有上百万个，这就又回到原来map join问题。所幸，每日的会员uv不会太多，有交易的会员不会太多，有点击的会员不会太多，有佣金的会员不会太多等等。所以这个方法能解决很多场景下的数据倾斜问题。

**4）总结**

使map的输出数据更均匀的分布到reduce中去，是我们的最终目标。由于Hash算法的局限性，按key Hash会或多或少的造成数据倾斜。大量经验表明数据倾斜的原因是人为的建表疏忽或业务逻辑可以规避的。在此给出较为**通用的步骤：**

* 采样log表，哪些user\_id比较倾斜，得到一个结果表tmp1。由于对计算框架来说，所有的数据过来，他都是不知道数据分布情况的，所以**采样**是并不可少的。
* 数据的分布符合社会学统计规则，贫富不均。倾斜的key不会太多，就像一个社会的富人不多，奇特的人不多一样。所以tmp1记录数会很少。把tmp1和users做**map join**生成tmp2，把tmp2读到distribute file cache。这是一个map过程。
* map读入users和log，假如记录来自log,则检查user\_id是否在tmp2里，如果是，输出到本地文件a,否则生成<user\_id,value>的key,value对，假如记录来自member,生成<user\_id,value>的key,value对，进入reduce阶段。
* 最终把a文件，把Stage3 reduce阶段输出的文件合并起写到hdfs。

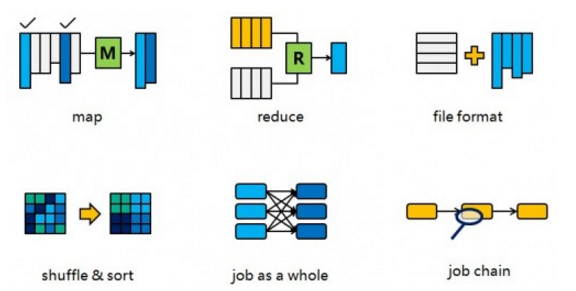
如果确认业务需要这样倾斜的逻辑，考虑以下的优化方案：

* 对于join，在判断小表不大于1G的情况下，使用map join
* 对于group by或distinct，设定 hive.groupby.skewindata=true
* 尽量使用上述的SQL语句调节进行优化

**3、HiveSQL性能优化**

<http://www.csdn.net/article/2015-01-13/2823530>

Hive查询生成多个map reduce job，一个map reduce job又有map，reduce，spill，shuffle，sort等多个阶段，所以针对hive查询的优化可以大致分为针对MR中单个步骤的优化，针对MR全局的优化以及针对整个查询的优化。



**4、hive优化十大原则**

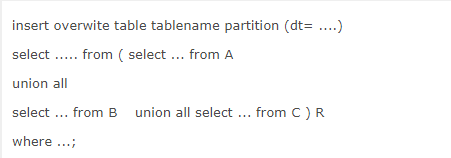
<http://www.cnblogs.com/sandbank/p/6408762.html>

**1）表连接优化**

* 将大表放后头：Hive假定查询中最后的一个表是大表。它会将其它表缓存起来，然后扫描最后那个表。因此通常需要将小表放前面，或者标记哪张表是大表：/\*streamtable(table\_name) \*/；
* 使用相同的连接键：当对3个或者更多个表进行join连接时，如果每个on子句都使用相同的连接键的话，那么只会产生一个MapReduce job；
* 尽量尽早地过滤数据：减少每个阶段的数据量，对于分区表要加分区，同时只选择需要使用到的字段；
* 尽量原子化操作：尽量避免一个SQL包含复杂逻辑，可以使用中间表来完成复杂的逻辑

**2）用insert into替换union all**

如果union all的部分个数大于2，或者每个union部分数据量大，应该拆成多个insert into 语句，实际测试过程中，执行时间能提升50%



改为：



**3）order by & sort by**

* order by : 对查询结果进行全局排序，消耗时间长。需要 set hive.mapred.mode=nostrict
* sort by : 局部排序，并非全局有序，提高效率

**4）transform+python**

一种嵌入在hive取数流程中的自定义函数，通过transform语句可以把在hive中不方便实现的功能在python中实现，然后写入hive表中。

语法：

select transform({column names1}) using '\*\*.py' as {column names2} from {table name}

如果除python脚本外还有其它依赖资源，可以使用ADD ARVHIVE

**5）limit 语句快速出结果**

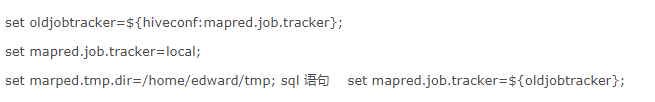
一般情况下，Limit语句还是需要执行整个查询语句，然后再返回部分结果。有一个配置属性可以开启，避免这种情况---对数据源进行抽样。

* hive.limit.optimize.enable=true --- 开启对数据源进行采样的功能
* hive.limit.row.max.size --- 设置最小的采样容量
* hive.limit.optimize.limit.file --- 设置最大的采样样本数

**缺点：**有可能部分数据永远不会被处理到

**6）本地模式**

对于小数据集，为查询触发执行任务消耗的时间>实际执行job的时间，因此可以通过本地模式，在单台机器上（或某些时候在单个进程上）处理所有的任务。



* -- 可以通过设置属性hive.exec.mode.local.auto的值为true，来让hve在适当的时候自动启动这个优化，也可以将这个配置写在$HOME/.hiverc文件中。
* -- 当一个job满足如下条件才能真正使用本地模式：
* job的输入数据大小必须小于参数：hive.exec.mode.local.auto.inputbytes.max(默认128MB)
* job的map数必须小于参数：hive.exec.mode.local.auto.tasks.max(默认4)
* job的reduce数必须为0或者1

可用参数hive.mapred.local.mem(默认0)控制child jvm使用的最大内存数。

**7）并行执行**

hive会将一个查询转化为一个或多个阶段，包括：MapReduce阶段、抽样阶段、合并阶段、limit阶段等。默认情况下，一次只执行一个阶段。 不过，如果某些阶段不是互相依赖，是可以并行执行的。会比较耗系统资源。

* set hive.exec.parallel=true,可以开启并发执行。
* set hive.exec.parallel.thread.number=16; //同一个sql允许最大并行度，默认为8。

**5、hive经验**

<http://www.iteye.com/news/31896>

**1）hive的使用场景有哪些？适合实时性强的分析场景使用么？**

Hive是建立在Hadoop上的数据仓库基础构架，它的最佳使用场合是大数据集的批处理作业，一般延迟性比较高，并不能够在大规模数据集上实现低延迟快速的查询，所以不适合实时性强的分析场景。不过目前Hive底层的计算框架除了支持原生的MapReduce，还支持Tez和Spark，这对提升Hive的查询处理性能帮助很大，从Hive 2.0版本开始，推荐使用Tez或Spark作为Hive的计算引擎。

如果希望实时查询分析，可以结合Impala，Presto，Drill等开源的交互式、实时的查询引擎使用，它们能够访问Hive中的表进行数据查询分析。

**2）hive的技术特点有哪些？**

Hive应该是第一个出现的SQL on Hadoop的产品，技术特点如下：

* 类SQL查询方式，支持标准SQL也比较全面；
* 支持索引，加快数据查询；
* 元数据保存在关系型数据库中，比如MySQL，可以减少查询过程中执行语义检查的时间；
* 支持多种数据存储格式类型，比如Text，Sequence，RCFile，Parquet和ORC等，针对不同的场景进行选择；
* 数据存储在HDFS分布式文件系统中，实现冗余高可用；
* 内置常用的基本函数，以及**窗口分析型函数**，同时支持用户自定义UDF，UDAF，UDTF函数；
* 底层计算引擎支持MapReduce，Tez和Spark，根据需要进行动态选择。

**3）Hive创建的内部表和外部表有何异同？**

和很多传统数据库中的内部表和外部表一样，没有什么差别。从表的创建和删除两方面简单介绍一下：

* Hive 创建内部表时，后面执行导入操作时会将用户数据移动到表所在的数据仓库指向的路径；
* 若创建外部表时，只会记录表对应的用户数据所在的路径，不对用户数据的位置做任何改变。
* 在删除表的时候，内部表的元数据和用户数据会被一起删除；
* 而外部表只会删除元数据，不删除用户数据。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。

**4）Hive的优化技巧有哪些？**

比如如何处理数据倾斜、大表与小表join时，如何优化性能。

Hive优化其实涉及到几方面，其中一方面是**计算引擎方面的优化**，比如你使用MapReduce作为计算引擎，那么就需要优化MapReduce；如果你选择Spark作为计算引擎，那么需要对Spark进行优化。

**A. 表设计层面优化**

* 合理利用中间结果集，避免查过就丢的资源浪费，减低Hadoop的IO负载
* 合理设计**表分区**，包括静态分区和动态分区
* 尽量不使用复杂或低效函数，比如count(distinct)，可以使用其他方式实现
* 选择**合适的表**存储格式和压缩格式
* 如果某些逻辑使用系统函数可能嵌套好几层，那么可以使用自定义函数实现
* 适当使用索引

**B. 语法和参数层面优化**

* 合理控制mapper和reducer数
* 设置map和reduce的内存大小
* 合并小文件
* 避免数据倾斜，解决数据倾斜问题：
* 处理数据倾斜的方法其实有很多，不论是Group by还是Join时出现数据倾斜，其实都是数据热点的问题，即某些Key值太多，导致都分发到一个节点执行，那么我们可以将数据量比较大的Key拿出来单独处理，最后再合并到结果集中。如果出现数据倾斜的Key值对结果无关紧要，比如空值，那么我们可以**过滤处理**，或者将空值加上随机数，进行分发到集群的所有节点并行处理。当然也可以利用Hive自带的参数进行优化，设置当分组或关联的Key值超过多少数量时，进行单独处理，即额外启动一个MapReduce作业处理。减少Job数；Join优化
* 尽量将小表放到join的左边。小表和大表join时，如果差一个以及以上数量级并且小表数据量很小，可以使用mapjoin方式，将小表全部读入内存中，在map阶段进行表关联匹配。大表和大表进行关联时，要注意数据倾斜的问题。如果两个表以相同Key进行分桶，以及表的桶个数是倍数关系，可以使用bucket join，加快关联查询。避免笛卡尔积；提前裁剪数据，减少处理的数据量，避免资源浪费

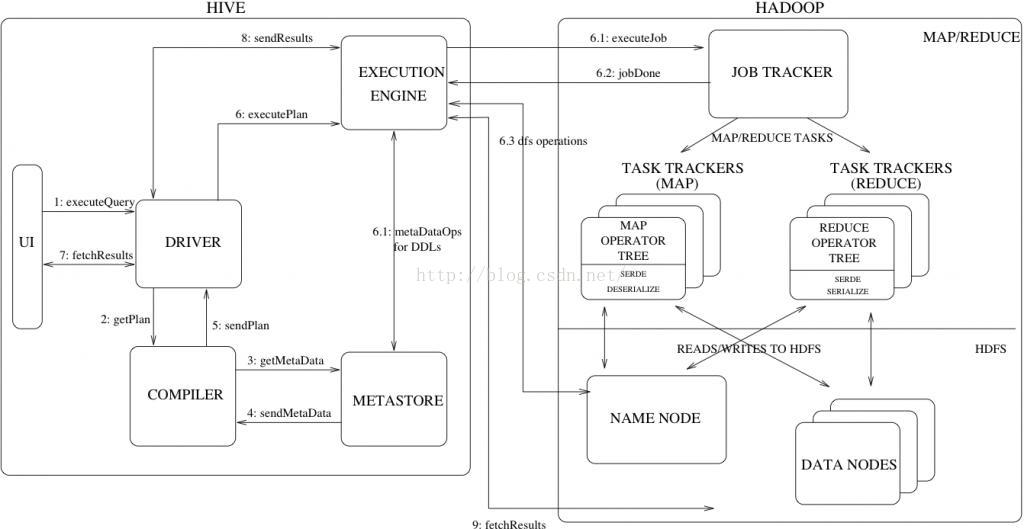
**C. Hive Job优化**

* 并行化执行——每个查询被Hive转化成多个阶段，有些阶段关联性不大，则可以并行化执行，减少执行时间。
* 本地化执行
* JVM重利用——JVM重利用可以是Job长时间保留slot，直到作业结束，这在对于有较多任务和较多小文件的任务是非常有意义的，减少执行时间。
* 推测执行——所谓的推测执行，就是当所有的task都开始运行之后，Job Tracker会统计所有任务的平均进度，如果某个task所在的节点配置内存比较低或者CPU负载很大，导致任务执行比总体任务的平均执行要慢，此时Job Tracker就会在其他节点启动一个新的相同的任务，原有任务和新任务哪个先执行完就把其他节点的另外一个任务kill掉。
* Hive中间结果压缩数据——中间压缩就是处理Hive查询的多个job之间的数据，对于中间压缩，最好选择一个节省CPU耗时的压缩方式

**6、HQL如何转化成mapreduce的过程**

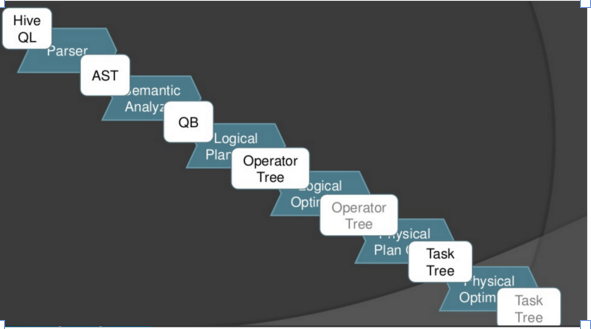
<http://www.jianshu.com/p/660fd157c5eb>

编译阶段，将HQL转换为mapreduce任务：



**1）基本转换步骤**

hiveSQL转换成MapReduce的执行计划包括如下几个步骤：HiveSQL ->AST(抽象语法树) -> QB(查询块) ->OperatorTree（操作树）->优化后的操作树->mapreduce任务树->优化后的mapreduce任务树

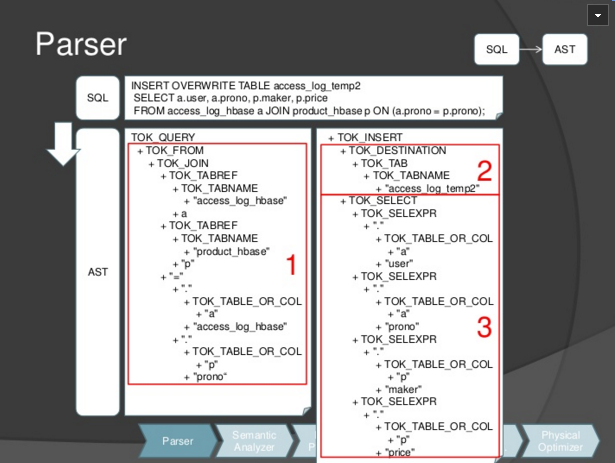


* **SQL Parser：**Antlr定义SQL的语法规则，完成SQL词法，语法解析，将SQL转化为抽象语法树AST Tree；
* **Semantic Analyzer：**遍历AST Tree，抽象出查询的基本组成单元QueryBlock；
* **Logical plan：**遍历QueryBlock，翻译为执行操作树OperatorTree；
* **Logical plan optimizer:** 逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量；
* **Physical plan：**遍历OperatorTree，翻译为MapReduce任务；
* **Logical plan optimizer：**物理层优化器进行MapReduce任务的变换，生成最终的执行计划；

**A. SQL Parser**

如下图所示， sql语句可以解析为三个部分

* AST中第一个部分对应SQL语句中FROM access\_log\_hbase a JOIN product\_hbase p ON (a.prono=p.prono)。
* insert overwrite table对应第二部分。
* select a.user, a.prono, p.maker, p.price对应第三部分。



**B. Semantic Analyzer**

这个步骤把AST转换成基本的查询块QB，如下图所示

QB的对象包括如下属性：

aliasToTabs:保存表格别名的信息

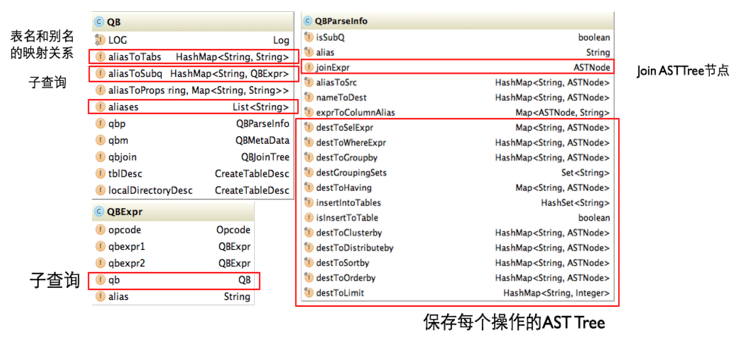
aliasToSubq:保存子查询的信息

qbm：保存每个输入表的元信息，比如表在HDFS上的路径，保存表数据的文件格式等

QBParseInfo对象包括如下属性：

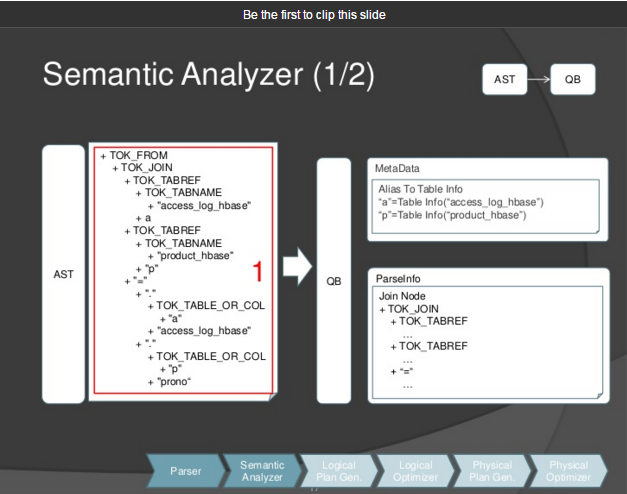
joinExpr: 保存TOK\_JOIN节点信息

destToxx:保存输出和各个操作的ASTNode节点的对应关系。

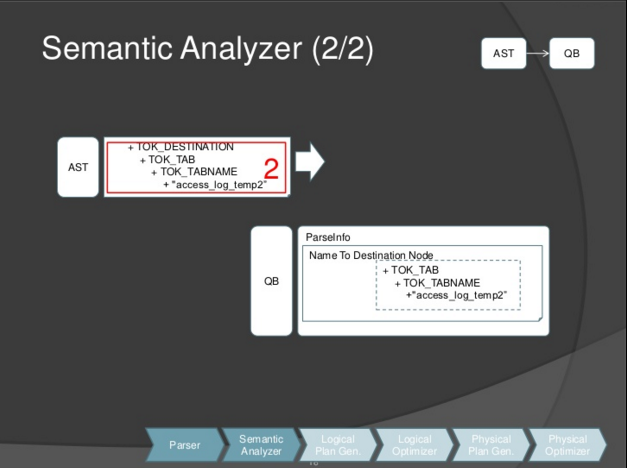


如下图中， 表格别名a, p保存到aliasTotabs, 分别对应“access\_log\_hbase", " product\_hbase"。

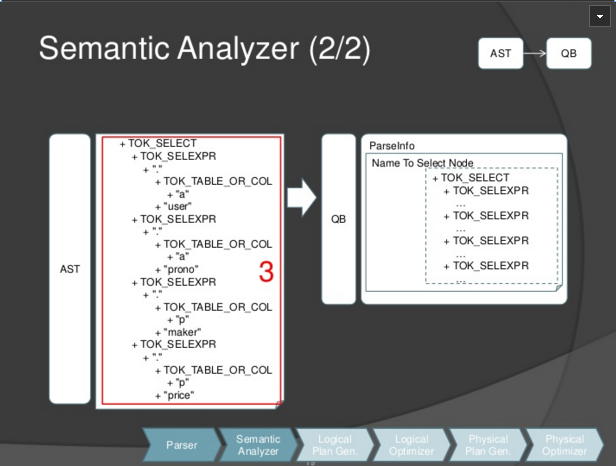
TOK\_JOIN信息保存到ParseInfo对象:joinExpr



下图所示，TOK\_DESTINATION节点保存到nameToDest属性中。



下图所示，TOK\_SELECT节点保存到nameToDest属性中。



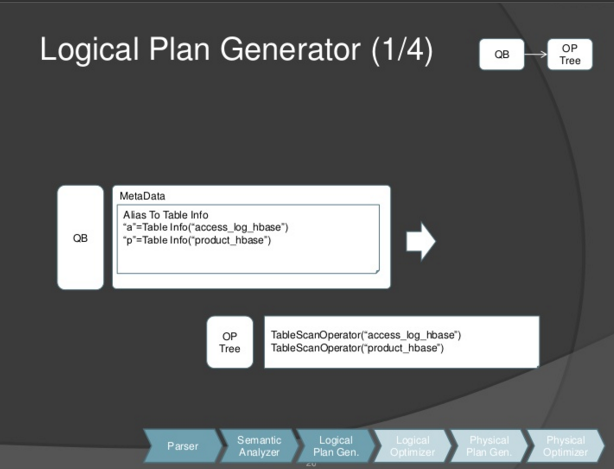
**C. Logical Plan**

该步骤是把查询块QB转换操作树。

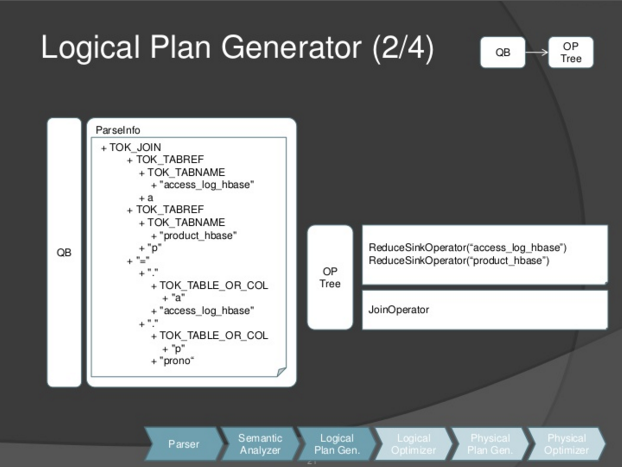
操作树基本的操作符包括TableScanOperator，SelectOperator，FilterOperator，JoinOperator，GroupByOperator，ReduceSinkOperator。

* TableScanOperator： 扫描数据表中数据，从原表中取数据。
* JoinOperator完成Join操作。
* FilterOperator完成过滤操作，对应sql里面的where语句功能
* ReduceSinkOperator：标志着Hive Map阶段的结束， Reduce阶段的开始。
* SelectOperator：reduce阶段输出select中的列
* FileSinkOperator: 生成结果数据到输出文件。

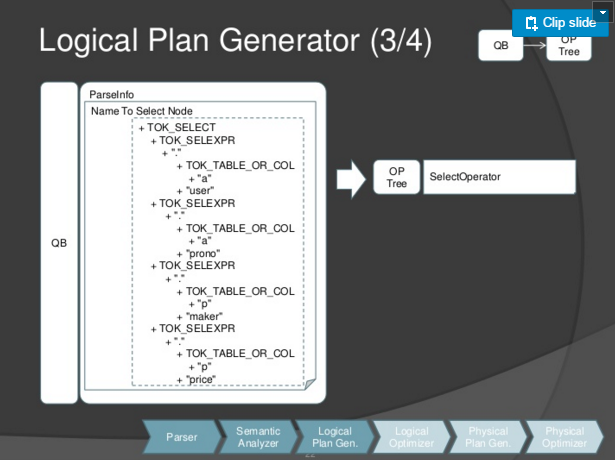
从两个输入表格中读入数据， 用operator树表示为两个TableScanOperator节点



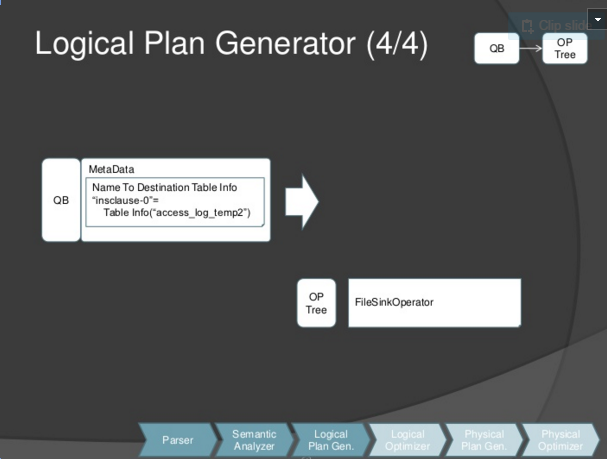
Join放在reduce阶段执行，执行join节点前，加入两个ReduceSinkOperator节点，表示当前map阶段结束， 进入到reduce阶段



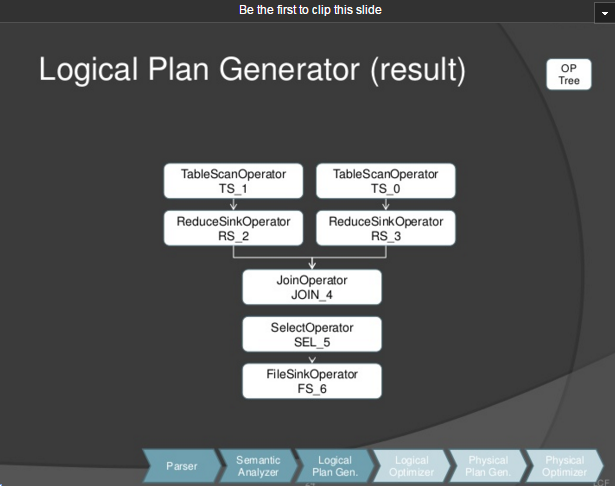
selectoperator节点，从reduce节点获取select指定的列值。



nameToDest ASTNode节点，转换为FileSinkOperator节点， 把结果写入到目标文件。



通过上面几个转换步骤， 最终生成的logical计划树。



logical plan tree还可以通过logical plan optimizer进一步优化， 优化完成的**逻辑优化树**还有转换成物理执行计划和物理执行计划优化。本文不做详细介绍， 后续有时间再补充。

PS: 查看hive sql编译后的执行计划

hive> **explain** select \* from tablename;

**2）简单总结**

Hive在执行一条HQL的时候，会经过以下步骤：

* **语法解析Parser**：Antlr定义SQL的语法规则，完成SQL词法，语法解析，将SQL转化为抽象语法树AST Tree；
* **语义解析**：遍历AST Tree，抽象出查询的基本组成单元QueryBlock；QueryBlock是一条SQL最基本的组成单元，包括三个部分：输入源，计算过程，输出。简单来讲一个QueryBlock就是一个子查询。
* **生成逻辑执行计划**：遍历QueryBlock，翻译为执行操作树OperatorTree；
* **优化逻辑执行计划**：逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量；
* **生成物理执行计划**：遍历OperatorTree，翻译为MapReduce任务；
* **优化物理执行计划**：物理层优化器进行MapReduce任务的变换，生成最终的执行计划；

参考文档：  
<http://tech.meituan.com/hive-sql-to-mapreduce.html> （很详细）  
<http://www.slideshare.net/recruitcojp/internal-hive>  
<http://lxw1234.com/archives/2015/09/476.htm>

**四、Spark知识**

**五、Kafka知识**

<http://www.cnblogs.com/cyfonly/p/5954614.html>

**六、ZooKeeeper知识**

**1、ZooKeeper的角色以及相应的Zookepper工作原理?**

<https://zhuanlan.zhihu.com/p/24383239>

Zookeeper的角色大概有如下四种：**leader、learner(follower)、observer、client**。其中leader主要用来决策和调度，follower和observer的区别仅仅在于后者没有写的职能，但都有将client请求提交给leader的职能，而observer的出现是为了应对当投票压力过大这种情形的，client就是用来发起请求的。

而Zookeeper所用的**分布式一致性算法**，包括leader的选举其实和-原始部落的获得神器为酋长，或者得玉玺者为皇帝类似，谁id最小，谁为leader，会根据你所配置的相应的文件在相应的节点机下生成id，然后相应的节点会通过getchildren()这个函数获取之前设置的节点下生成的id，谁最小，谁是leader。并且如果万一这个leader挂掉了或者堕落了，则由次小的顶上。而且在配置相应的zookeeper文件的时候回有类似于如下字样的信息：Server.x=AAAA:BBBB:CCCC。其中的x即为你的节点号哈，AAAA对应你所部属zookeeper所在的ip地址，BBBB为接收client请求的端口，CCCC为重新选举leader端口。

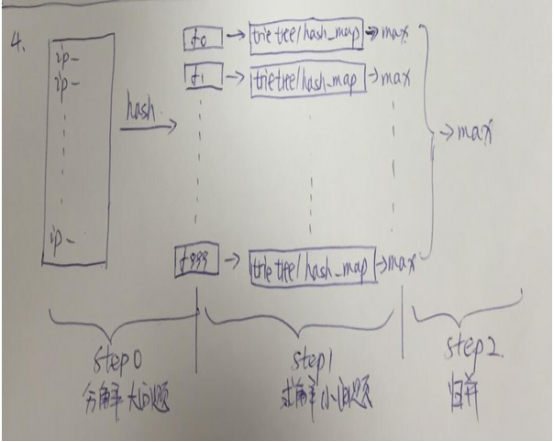
**七、大数据经典面试题及答案**

<http://www.offcn.com/it/2017/0517/9051.html>

<https://zhuanlan.zhihu.com/p/24383239>

**1、海量日志数据，提取出某日访问百度次数最多的那个IP？**

**解析：将大文件映射分割多个小文件，然后分别统计每个小文件内出现频率最大的前几个IP，最后对比从所有小文件统计的前几个IP，找出出现频率最大的IP即为所求。**首先是将这一天，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。



**2、检索最热门的查询串**

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。

**解析**：

* 思想：首先对海量的数据进行预处理，即使用Hash表对这些数据进行去重排序处理，而后采用**小根堆结构**遍历hash后的数据，并维护最热门的10个查询串，即为所求。
* 第一步、先对这批**海量数据预处理**，在O(N)的时间内用Hash表完成排序（hash去重并排序）；
* 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)。

**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。

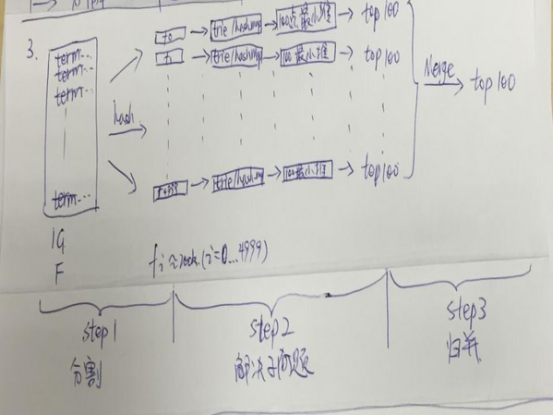
**3、统计出现频数最高的单词（内存限制）**

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

**解析：大文件分割为多个小文件，已满足内存限制条件**

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。

**分割大问题，求解小问题，归并：**



**4、大文件数据query频度排序**

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

**解析**：

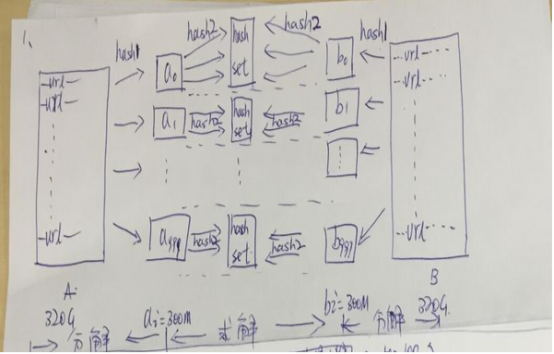
* **方案1：**顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件(记为)中。这样新生成的文件每个的大小大约也1G(假设hash函数是随机的)。找一台内存在2G左右的机器，依次对用hash\_map(query, query\_count)来统计每个query出现的次数。利用快**速/堆/归并排序按照出现次数进行排序**。将排序好的query和对应的query\_cout输出到文件中。这样得到了10个排好序的文件(记为)。对这10个文件进行归并排序(**内排序与外排序相结合**)。
* **方案2**：一般query的总量是有限的，只是重复的次数比较多而已，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。

**5、寻找大数据文件中共同的URL**

给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url?

**解析**：

* **方案1**：可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取**分而治之**的方法。
* 通读文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件(记为a0,a1,...,a999)中。这样每个小文件的大约为300M。
* 通读文件b，采取和a相同的方式将url分别存储到1000小文件(记为b0,b1,...,b999)。这样处理后，**所有可能相同的url都在对应的小文件**(a0vsb0,a1vsb1,...,a999vsb999)中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。
* 求每对小文件中相同的url时，可以把其中一个小文件的url存储到**hash\_set**中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。
* **方案2**：如果允许有一定的**错误率**，可以使用Bloom filter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是**共同的url**(注意会有一定的错误率)。



**6、寻找大数据中不重复的整数**

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

**解析**：

* **方案1（位图解析）**：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。
* **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。

**7、快速判断一个数是否在原数集中（腾讯面试题）**

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?

**解析**：

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。

**方案2：**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。

**附：**再简单介绍下，位图方法：使用位图法判断整形数组是否存在重复，判断集合中存在重复是常见编程任务之一，当集合中数据量比较大时我们通常希望少进行几次扫描，这时双重循环法就不可取了。

**位图法**比较适合于这种情况，它的做法是按照集合中最大元素max创建一个长度为max+1的新数组，然后再次扫描原数组，遇到几就给新数组的第几位置上1，如遇到5就给新数组的第六个元素置1，这样下次再遇到5想置位时发现新数组的第六个元素已经是1了，这说明这次的数据肯定和以前的数据存在着重复。这种给新数组初始化时，置零其后置一的做法类似于**位图的处理方法故称位图法**。它的运算次数最坏的情况为2N。如果已知数组的最大值，即能事先给新数组定长的话效率还能提高一倍。

**8、怎么在海量数据中找出重复次数最多的一个?**

**解析：**

先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求(具体参考前面的题)

**9、上千万或上亿数据(有重复)，统计其中出现次数最多的前N个数据**

**解析：**

千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用hash\_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前N个出现次数最多的数据了，可以用第2题提到的**堆机制**完成。

**10、统计词出现频度及复杂度分析**

一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析

**解析：**

* 方案1：这题是考虑时间效率。用trie树统计每个词出现的次数，时间复杂度是O(n\*le)(le表示单词的平准长度)。然后是找出出现**最频繁的前10个词，可以用堆来实现**，前面的题中已经讲到了，时间复杂度是O(n\*lg10)。所以总的时间复杂度，是O(n\*le)与O(n\*lg10)中较大的哪一个。

**11、100w个数中找出最大的100个数**

* **方案1**：在前面的题中，我们已经提到了，用一个含100个元素的最小堆完成。复杂度为O(100w\*lg100)。
* **方案2**：采用**快速排序**的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比100多的时候，采用传统排序算法排序，取前100个。复杂度为O(100w\*100)。
* **方案3**：采用**局部淘汰法**。选取前100个元素，并排序，记为序列L。然后一次扫描剩余的元素x，与排好序的100个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把x利用插入排序的思想，插入到序列L中。依次循环，知道扫描了所有的元素。复杂度为O(100w\*100)。

**八、十个海量数据处理方法大总结**

<https://zhuanlan.zhihu.com/p/24383239>

**1、BloomFilter**

* **适用范围：**可以用来实现数据字典，进行数据的判重，或者集合求交集
* **基本原理及要点**：对于原理来说很简单，位数组+k个独立hash函数。将hash函数对应的值的位数组置1，查找时如果发现所有hash函数对应位都是1说明存在，很明显这个过程并不保证查找的结果是100%正确的。同时也不支持删除一个已经插入的关键字，因为该关键字对应的位会牵动到其他的关键字。所以一个简单的改进就是countingBloomfilter，用一个counter数组代替位数组，就可以支持删除了。
* 还有一个比较重要的问题，**如何根据输入元素个数n，确定位数组m的大小及hash函数个数**。当hash函数个数**k=(ln2)\*(m/n)**时错误率最小。在错误率不大于E的情况下，m至少要等于n\*lg(1/E)才能表示任意n个元素的集合。但m还应该更大些，因为还要保证bit数组里至少一半为0，则m应该>=nlg(1/E)\*lge大概就是nlg(1/E)1.44倍(lg表示以2为底的对数)。

举个例子我们假设错误率为0.01，则此时m应大概是n的13倍。这样k大概是8个。

注意这里m与n的单位不同，m是bit为单位，而n则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多bit的。所以使用bloomfilter内存上通常都是节省的。

* **扩展：**

Bloomfilter将集合中的元素映射到位数组中，用k(k为哈希函数个数)个映射位是否全1表示元素在不在这个集合中。Countingbloomfilter(CBF)将位数组中的每一位扩展为一个counter，从而支持了元素的删除操作。SpectralBloomFilter(SBF)将其与集合元素的出现次数关联。SBF采用counter中的最小值来近似表示元素的出现频率。

**问题实例**：给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你找出A,B文件共同的URL。如果是三个乃至n个文件呢?

**解析：**根据这个问题我们来计算下内存的占用，4G=2^32大概是40亿\*8大概是340亿，n=50亿，如果按出错率0.01算需要的大概是650亿个bit。现在可用的是340亿，相差并不多，这样可能会使出错率上升些。另外如果这些url-ip是一一对应的，就可以转换成ip，则大大简单了。

**2、Hashing**

* **适用范围**：快速查找，删除的基本数据结构，通常需要总数据量可以放入内存
* **基本原理及要点**：hash函数选择，针对字符串，整数，排列，具体相应的hash方法。碰撞处理，一种是openhashing，也称为拉链法；另一种就是closedhashing，也称开地址法，openedaddressing。
* **扩展：**

d-lefthashing中的d是多个的意思，我们先简化这个问题，看一看2-lefthashing。2-lefthashing指的是将一个哈希表分成长度相等的两半，分别叫做T1和T2，给T1和T2分别配备一个哈希函数，h1和h2。在存储一个新的key时，同时用两个哈希函数进行计算，得出两个地址h1[key]和h2[key]。这时需要检查T1中的h1[key]位置和T2中的h2[key]位置，哪一个位置已经存储的(有碰撞的)key比较多，然后将新key存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个key，就把新key存储在左边的T1子表中，2-left也由此而来。在查找一个key时，必须进行两次hash，同时查找两个位置。

* **问题实例**：海量日志数据，提取出某日访问百度次数最多的那个IP。
* **解析**：IP的数目还是有限的，最多2^32个，所以可以考虑使用hash将ip直接存入内存，然后进行统计。

**3、Bit-map**

* **适用范围**：可进行数据的快速查找，判重，删除，一般来说数据范围是int的10倍以下
* **基本原理及要点**：使用bit数组来表示某些元素是否存在，比如8位电话号码
* **扩展**：bloomfilter可以看做是对bit-map的扩展
* **问题实例：**

1）已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。

8位最多99999999，大概需要99m个bit，大概10几m字节的内存即可。

2）2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

将bit-map扩展一下，用**2bit表示一个数**即可，0表示未出现，1表示出现一次，2表示出现2次及以上。或者我们不用2bit来进行表示，我们用两个bit-map即可模拟实现这个2bit-map。

**4、堆**

* **适用范围**：海量数据前n大，并且n比较小，堆可以放入内存
* **基本原理及要点**：最大堆求前n小，最小堆求前n大。方法，比如求前n小，我们比较当前元素与最大堆里的最大元素，如果它小于最大元素，则应该替换那个最大元素。这样最后得到的n个元素就是最小的n个。适合大数据量，求前n小，n的大小比较小的情况，这样可以扫描一遍即可得到所有的前n元素，效率很高。
* **扩展**：双堆，一个最大堆与一个最小堆结合，可以用来维护中位数。
* **问题实例：**

1）100w个数中找最大的前100个数。

用一个100个元素大小的最小堆即可

**5、数据库索引**

**适用范围**：大数据量的增删改查

**基本原理及要点**：利用数据的设计实现方法，对海量数据的增删改查进行处理

**6、倒排索引(Invertedindex)**

* **适用范围**：搜索引擎，关键字查询
* **基本原理及要点**：为何叫倒排索引?一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。

**7、双层桶划分**

其实本质上就是【分而治之】的思想，重在“分”的技巧上!

* **适用范围**：第k大，中位数，不重复或重复的数字
* **基本原理及要点**：因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。可以通过多次缩小，双层只是一个例子。
* **问题实例：**

1）2.5亿个整数中找出不重复的整数的个数，内存空间不足以容纳这2.5亿个整数。

有点像**鸽巢原理**，整数个数为2^32,也就是，我们可以将这2^32个数，划分为2^8个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。

2）5亿个int找它们的中位数。

这个例子比上面那个更明显。首先我们将int划分为2^16个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成2^24个区域，然后确定区域的第几大数，在将该区域分成2^20个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有2^20，就可以直接利用directaddrtable进行统计了。

**8、外排序**

* **适用范围**：大数据的排序，去重
* **基本原理及要点**：外排序的归并方法，置换选择败者树原理，最优归并树
* **问题实例：**

1）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个字节，内存限制大小是1M。返回频数最高的100个词。

这个数据具有很明显的特点，词的大小为16个字节，但是内存只有1m做hash有些不够，所以可以用来排序。内存可以当输入缓冲区使用。

**9、trie树**

* **适用范围**：数据量大，重复多，但是数据种类小可以放入内存
* **基本原理及要点**：实现方式，节点孩子的表示方式
* **问题实例**：

1）有10个文件，每个文件1G，每个文件的每一行都存放的是用户的query，每个文件的query都可能重复。要你按照query的频度排序。

2）1000万字符串，其中有些是相同的(重复),需要把重复的全部去掉，保留没有重复的字符串。请问怎么设计和实现?

3）**寻找热门查询**：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，每个不超过255字节。

**10、分布式处理mapreduce**

* **适用范围**：数据量大，但是数据种类小可以放入内存
* **基本原理及要点**：将数据交给不同的机器去处理，数据划分，结果归约。