# HDFS面试题整理

1. HDFS是什么

* hdfs（Hadoop Distribute File System）是分布式文件系统。
* 分布式文件系统(distributed file system)是指文件系统管理的物理存储资源不一定直接链接在本地节点上，而是通过计算机网络与节点相连，可让多机器上的多用户分享文件和存储空间。
* 说简单点就是将一个大文件分布式存储在不同的机器节点上

# HDFS架构

* HDFS为主从架构，由单个NameNode和多个DataNode组成，另外还包括SecondaryNameNode。
* NameNode主要负责管理dataNode和记录元数据。由FsImage和EditLog两部分组成。
  + FsImage用户维护文件目录和元数据
  + EditLog则记录所有对文件操作的操作。
* DataNode负责存储数据，同时定期与NameNode进行通信，汇报自己存储的信息。
* SecondaryNameNode负责合并日志，定期将新的FsImage拷贝回dataNode。

1. HDFS相关概念

* block：文件上传前需要分块，这个块就是block，一般为128MB，当然你可以去改，不顾不推荐。因为块太小：寻址时间占比过高。块太大：Map任务数太少，作业执行速度变慢。它是最大的一个单位。
* packet：packet是第二大的单位，它是client端向DataNode，或DataNode的PipLine之间传数据的基本单位，默认64KB。
* chunk：chunk是最小的单位，它是client向DataNode，或DataNode的PipLine之间进行数据校验的基本单位，默认512Byte，因为用作校验，故每个chunk需要带有4Byte的校验位。所以实际每个chunk写入packet的大小为516Byte。由此可见真实数据与校验值数据的比值约为128 : 1。（即64\*1024 / 512）

1. HDFS中小文件过多有什么坏处？

* 主要是NameNode内存会受限。由于NameNode会将文件系统的元数据都存储在内存中，因此所能存储的文件总量受限于NameNode的内存总容量。每个文件、目录和数据块大约占150个字节，过多的小文件会大量消耗NameNode的内存量
* 增加NameNode寻址的时间

1. 怎么处理小文件过多的问题？

* 数据未上传到HDFS前的处理：从根源上解决问题，每小时抽取一次数据改为每天抽取一次数据积累数据量
* 数据已上传到HDFS：通过程序将小文件和并成大文件

1. HDFS中NameNode内存总共有多大

* Hadoop 2.x系列，默认为2000m
* Hadoop 3.x系列，动态分配内存，最小值为1G，每增加100万个Block，增加1G内存

1. HDFS支持并发写入么？

* 不支持，客户端想写数据必须续约（许可证），同时只能有一个客户端写数据，直到这个客户端契约过期，下个客户端才可以获取契约并写入数据，详见写数据流程

1. HDFS的特性与优点

* 大数据分而治之，主要针对一次写入多次读取的场景
* 分布式，扩容方便
* 高可靠，稳定
* 成本低，只需要几台磁盘大点的服务器

1. NameNode 和 SecondaryNameNode的区别与联系

* NameNode负责管理元数据
* SecondaryNameNode见名字像是NameNode的备份，实则不是，这也是hdfs命名的缺陷，SecondaryNameNode主要负责 standby namenode中的checkporint动作，将内存中的fsimage合并到磁盘上的fsimage中

1. fsimage与editlogs是的作用？

* NameNode主要维护了两个文件，fsimage与editlogs
* fsimage：最近元数据的检查点，包括数据块描述信息、修改时间、访问时间、元数据的权限、副本个数等。相当于快照
* editlogs：所有元数据，写入、删除、移动的操作，都会记录在这里

1. HDFS有哪些设计不合理的地方

* 数据不能实时处理
* 不能并发写入
* 小文件太多会占用过多namenode中的内存，增加namenode的寻址地址的时间

1. 简述hadoop的二级排序原理

* 通过自定义分区实现，因为如果采用默认分区策略很有可能产生数据倾斜现象
* 哪个key分配到哪个分区这个过程自定义控制，将key分区好后，将key进行排序
* 再将每个key中的值进行排序

1. MapReduce跑的慢的原因

* key设计不合理 或 数据倾斜
* 小文件过多
* 数据太大导致spill和merge的次数过多
* Map和Reduce的个数设计的不合理

1. 什么是数据倾斜, 数据倾斜的原因

* key分布不均匀
* 业务数据本身的特性
  + 数据混乱，空值过多，可将空值的key变成一段字符串+随机数，把倾斜的数据分布到不同的reduce上，由于值是null关联不上( where 值 != null )，不会影响最终结果
* 建表时考虑不周
  + 表设计的不好，没有按照建模思想建表
* sql 语句本身就有数据倾斜
  + join
    - 做好列裁剪和数据过滤操作
    - map join ，大小表join ，让小表先进入到内存
  + group by
  + count distinct
    - 可用sum() + group by 方式替换

1. Hadoop Rpc 原理

* Rpc 是远程（不同进程）间过程（方法）的调用 的缩写形式，分为客户端和服务端，服务端会一直开启一个服务等待别人的调用，客户端获取服务端的代理，调用服务端的方法后，方法在服务端上执行
* Hadoop Rpc是通过构建者设计模式构建的

# hdfs故障检测机制

故障的类型主要有以下三种，针对这三种故障类型，HDFS提供了不同的故障检测机制：

* 针对DataNode失效问题， HDFS使用了心跳机制，DataNode定期向NameNode发送心跳信息，NameNode根据心跳信息判断DataNode是否存活；
* 针对网络故障而导致无法收发数据的问题，HDFS提供了ACK的机制，在发送端发送数据后，如果没有收到ACK并且经过多次重试后仍然如此，则认为网络故障；
* 针对数据损坏问题，所有DataNode会定期向NameNode发送自身存储的块清单，在传输数据的同时会发送总和校验码，NameNode依次来判断数据是否丢失或损坏。

1. **hdfs的文件上传流程**

* 首先客户端通过Distributed FileSystem模块向NameNode请求上传文件，NameNode检查目标文件是否已存在，父目录是否存在
* NameNode返回是否可以上传。
* 如果文件大于128M则分块存储，客户端请求第一个 Block上传到哪几个DataNode服务器上。
* NameNode根据副本储存策略返回3个DataNode节点，假如为dn1、dn2、dn3。
* 客户端通过FSDataOutputStream模块请求dn1上传数据，dn1收到请求调用dn2，dn2调用dn3，建立通信管道完成,dn1、dn2、dn3逐级应答客户端。
* 客户端以Packet为单位往dn1上传第一个Block数据,dn1收到Packet就会传给dn2，dn2传给dn3；dn1,dn2,dn3每接收packet会放入一个待写队列等待写入数据，落盘。
* 当一个Block传输完成之后，客户端再次请求NameNode上传第二个Block的服务器，重复执行3-6步。

1. **文件的下载流程**

* 客户端通过Distributed FileSystem向NameNode请求下载文件，NameNode通过查询元数据，找到文件块所在的DataNode地址。
* 挑选一台DataNode（就近原则，然后随机）服务器，请求读取数据。
* DataNode开始传输数据给客户端（从磁盘里面读取数据输入流，以Packet为单位来做校验）。
* 客户端以Packet为单位接收，先在本地缓存，然后写入目标文件。

1. HDFS在上传文件的时候，如果其中一个DataNode突然挂掉了怎么办

* 客户端上传文件时与DataNode建立pipeline管道，管道的正方向是客户端向DataNode发送的数据包，管道反向是DataNode向客户端发送ack确认，也就是正确接收到数据包之后发送一个已确认接收到的应答。
* 当DataNode突然挂掉了，客户端接收不到这个DataNode发送的ack确认，客户端会通知NameNode，NameNode检查该块的副本与规定的不符，NameNode会通知DataNode去复制副本，并将挂掉的DataNode作下线处理，不再让它参与文件上传与下载。

1. HDFS的常用命令是什么?

* hdfs:查看hdfs命令所有参数
* hdfs dfsadmin -report //报告当前集群的情况
* hdfs dfsadmin -safemode //安全模式
* hdfs dfsamin -safemode leave // leave代表关闭,安全模式关闭才是读写正常的
* hdfs dfsadmin -safemode enter //进入安全模式.如正在做集群维护的时候,就可以手动进入安全模式,维护完就离开
* hdfs fasck //对hdfs做检查.如发现Target Replicas is 3 but 1 live replica(s) 代表有副本丢失.
* hdfs balancer //集群平衡. DN1与DN2间节点平衡.
* hdfs balancer -policy datanode -threshold 10 //让每个节点磁盘使用率减去平均磁盘使用率小于10%
* hdfs diskbalancer //磁盘平衡. DN1中,多块磁盘的平衡,前提条件:hdfs.disk.balancer.enabled = true;

# MapReduce面试题整理

1. **MapReduce核心思想** **分而治之，先分后和（只有一个模型）** 【将一个大的、复杂的工作或任务，拆分成多个小的任务，并行处理，最终进行合并。适用于大量复杂的、时效性不高的任务处理场景（大规模离线数据处理场景）。】 Map负责数据拆分 map: [k1,v1] → [(k2,v2)] Reduce负责数据合并 reduce: [k2, {v2,…}] → [k3, v3]

**2.偏移量** ：指的是每行行首字母移动到文办的最前面需要一定的字符。

**3.Suffle包含哪些步骤** partition(分区)-sort(排序)-combiner（局部聚合）=group(分组)

**4.MR从读取数据开始到将最终结果写入HDFS经过哪些步骤：**

**第一步：InputFormat** InputFormat 在HDFS文件系统中读取要进行计算的数据输出 给Split

**第二步：Split** Split 将数据进行逻辑切分，切分成多个任务。输出给RR

**第三步：RR( RecordReader)** RR 将切分后的数据转换成key value进行输出 key : 每 一行行首字母的偏移量 value: 每一行数据输出给Map

**第四步：Map** 接收一条一条的数据（有多少行数据Map运行多少次，输出的次数根据 实际业务需求而定）根域业务需求编写代码 Map的输出是 key value的 list 输出给 Shuffle（partition） ---------------------------------------Map------------------------------------------------------

**第五步： partition** partition: 按照一定的规则对 \*\*key value的 list进行分区输出给 Shuffle（sort）

**第六步：Sort** Sort :对每个分区内的数据进行排序。 输出给Shuffle（Combiner）

**第七步：Combiner** Combiner: 在Map端进行局部聚合（汇总） 目的是为了减少网络 带宽的开销输出给Shuffle（Group）

**第八步：Group** Group: 将相同key的key提取出来作为唯一的key 将相同key对应 的value提取出来组装成一个value 的List 输出给Shuffle（reduce） ------------------------------------Shuffle--------------------------------------------

**第九步：reduce reduce：** 根据业务需求对传入的数据进行汇总计算。输出给Shuffle （outputFormat）

**第十步：outputFormat** outputFormat:将最终的额结果写入HDFS

**5.如何设置ReduceTask的数量** reducetask的设置： job.setNumReduceTasks(2);

**6.combiner的作用** Map结束后，在Map端进行局部聚和。 作用：较少网络传入次数，降低了网络开销。

**7.combiner运行在MapReduce的哪一端？** map 每一个map都可能会产生大量的本地输出，Combiner的作用就是对map端的输出先做一次合并，以减少在map和reduce节点之间的数据传输量，以提高网络IO性能.

**8.Maptask的数量是可以人为设置的吗？** 不可以。

**9.Maptask的个数由什么决定？**一个job的map阶段MapTask并行度（个数），由客户端提交job时的切片个数决定。

**10.Shuffle阶段的Partition分区算法是什么** 计算逻辑：对map输出的key 取哈希值，用这个哈希值与reducetask的值取余。 余几，就将这个key，value放在对应的分区编号里（分区有多个编号）

**11.Split逻辑切分数据，节分大小是多大？** hdfs数据块大小是128 ，split 逻辑切分数据块大小是128M HDFS 128 是存储层面的概念， 是切分数据的分界点。 split 128 是一个逻辑切分。 这两个128没有关系。

**12内存角度介绍Map的输出到Reduce的输入的过程。** **1.Map 输出数据到内存:** map输出的数据写入环形缓冲区（内存），缓冲区的默认大小是100M（可修改）。当数据达到阈值（默认0.8-可修改）时，环形缓冲区进行flash, 环形缓冲区：数据在输出的同时，数据也可以写入空余的空间内。 当flash的数据个数达到一定的数量时（默认4个）。对数据进行合并（merge）。 **2:Reduce在Map拷贝数据** Map 输出的结果写入本地，reduce主动发出拷贝进程到Map端拷贝数据。 reduce获取[大数据](https://cloud.tencent.com/solution/bigdata?from=10680" \t "_blank)后，将数据写入内存，当数据达到阈值时进行flash. 当flash的个数达到一定的量时，进行合并，最终发送给reduce

**13.最优的Map效率是什么？** Map端的最高效率:尽量减少环形缓冲区flush的次数（减少磁盘IO的使用次数）

**14.最优的reduce是什么？** 1.尽量减少环形缓冲区flush的次数 2.尽量将所有的数据在内存计算。

**15.在MapReduce阶段，有哪些优化的点？（至少两个点）** 1、加大环形缓冲区的内存 2、增大缓冲区阈值的大小 （考虑剩余的空间是不是够系统使用） 3、对输出的进行压缩（压缩-解压的过程会消耗CPU）

**16.集群优化的核心思路是什么？** 在网络带宽，磁盘IO是瓶颈的前期下 能不使用IO网络就不使用，在必须使用的前提下，能少用就少用。 所有的只要能够减少网络带宽的开销，只要能够减少磁盘IO的使用的次数的配置项，都是集群调优的可选项。 （可选项包括：软件层面【系统软件和集群软件】，硬件层面，网络层面）

**17.如果没有定义partitioner，那数据在被送达reducer前是如何被分区的？**

如果没有自定义的 partitioning，则默认的 partition 算法，即根据每一条数据的 key 的 hashcode 值摸运算（%）reduce 的数量，得到的数字就是“分区号“。

**18.什么样的计算不能用mr来提速？**

1）数据量很小。2）繁杂的小文件。3）索引是更好的存取机制的时候。

4）事务处理。5）只有一台机器的时候。

**19.MapReduce Framework 中的分布式缓存是什么？请说明。**

答：分布式缓存是 MapReduce 框架的重要组成部分。它用于在执行期间跨操作缓存文件，并确保更快地执行任务。该框架使用分布式缓存来存储在该特定节点上执行任务经常需要的重要文件。

1. **如何减少Hadoop Map端到Reduce端的数据传输量**

1.使用combiner函数

2.减少map个数(修改参数)

3.减少reduce个数。

1. **简述Shuffle过程，如何对其IO进行优化**

Shuffle过程包括collect，sort，spill，merge，copy，Merge Sort的过程

Collect过程将Map的结果以kv的形式，放在一个环形数组kvbuffer中，以尽可能有效利用内存空间。

Sort将kvbuffer中的kv按照partition和key值进行升序排列，排序只移动索引。排序结果为kvmeta中的数据，按照partition为单位聚集，同一partition内按照key有序。

Spill创建磁盘文件，把kvmeta挨个partition输出到磁盘中。

Merge，如果map输出较大，可能有多个spill，merge要将这些out和index文件按照相同的key，merge成group。

Copy，reduce端通过http向map任务拖取数据，每个节点有一个httpserver，读取相应Map输出文件对应这个reduce的部分，通过网络流输出给reduce

Merge Sort，和map端类似，Map输出数据是有序的，merge sort进行一次归并排序。Reduce端一边进行copy，一边merge sort。

**Shuffle IO优化**：

Shuffle如果生成map reduce的数量的乘积个临时文件，将面临极大IO压力。

可以使用sort shuffle，map端输出M个数据文件和M个索引文件，reducer直接访问有序的数据，从而减少文件数目。

而在map和reduce数量不多时，sort shuffle会多此一举，可以退化为hash shuffle，让文件数和map数脱钩。

# HIVE面试题整理

Hive面试题整理：

1. hive中存的是什么？

表

存的是和hdfs的映射关系，hive是逻辑上的数据仓库，实际上操作的都是hdfs上的文件，hql就是用sql语法来写的MR程序

1. hive和关系型数据库的关系？

没有关系，hive是数据仓库，不能和数据库一样直接进行curb操作。

是一次性进行多次读取的操作，可以看成是ETL的工具。

1. hive和hbase的关系

Hbase是Hadoop database的简称，是一种NoSQL数据库，主要适用于海量明细数据（十亿、百亿）的随机实时查询，如日志明细、交易清单、轨迹行为等。

Hive是Hadoop数据仓库，通过SQL来处理和计算HDFS的数据，Hive会将SQL翻译为Mapreduce来处理数据，适用于离线的批量数据计算。

在大数据架构中，Hive和HBase是协作关系，在数据引入到数据存储上密切配合，共同完成任务，通过ETL工具将数据源抽取到HDFS存储；通过Hive清洗、处理和计算原始数据；HIve清洗处理后的结果，如果是面向海量数据随机查询场景的可存入Hbase；数据应用从HBase查询数据。

1. hive和hbase的区别

Hive是为了简化mapreduce操作而生的，hive是hadoop的一个数据仓库工具，可以将结构化工具映射成一张表数据库，并提供sql查询功能，hive适用于离线查询，因为查询时间较长

Hbase是为查询而生的，hbase是hadoop的数据库，是一个非关系型数据库，非常适合进行大数据的实时查询

1. 试说明hive中sort by, order by,cluster by,distribute by各代表什么意思

Order by：会对输入做全局排序，因此只有一个reducer（多个reducer无法保证全局有序），只有一个reducer会导致输入过大时，需要较长的计算时间。

Sort by：不是全局排序，其在数据进入reducer前完成排序，

Distribute by：根据指定字段对数据进行划分，输出到不同的reduce中

cluster by：除了Distribute by的功能外还具有Sort by的功能

1. hive底层和数据库交互的原理

Hive的查询功能是由hdfs+mapreduce结合起来实现的

Hive和mysql的关系：只是借助mysql来存储hive中的表的元数据的信息，成为metastore

1. hive内部表和外部表的区别

未被external修饰的是内部表，被external修饰的为外部表。

内部表数据由Hive自身管理，外部表数据由HDFS管理；

内部表数据存储的位置是hive.metastore.warehouse.dir（默认：/user/hive/warehouse）， 外部表数据的存储位置由自己制定（如果没有LOCATION，Hive将在HDFS上 的/user/hive/warehouse文件夹下以外部表的表名创建一个文件夹，并将属于这个表的数据存 放在这里）；

删除内部表会直接删除元数据（metadata）及存储数据；删除外部表仅仅会删除元数据，HDFS上的文件并不会被删除。

1. hive有索引吗

Hive支持索引（3.0版本之前），但是Hive的索引与关系型数据库中的索引并不相同。并且Hive索引提供的功能很有限，效率也并不高，因此Hive索引很少使用。

索引适用的场景：

适用于不更新的静态字段。以免总是重建索引数据。每次建立、更新数据后，都要重建索引以构建索引表。

1. 为什么要对数据仓库进行分层

用空间换时间，通过大量的预处理来提升应用系统的用户体验（效率），因此数据仓库会存在大量冗余的数据。如果不分层的话，如果源业务系统的业务规则发生变化将会影响整个数据清洗过程，工作量巨大。

通过数据分层管理可以简化数据清洗的过程，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

1. hive小文件过多怎么办？

使用 hive 自带的 concatenate 命令，自动合并小文件

调整参数减少Map数量

减少Reduce的数量

使用hadoop的archive将小文件归档

1. hive的优化有哪些？

数据存储及压缩

通过调参优化

有效地减小数据集将大表拆分成子表；结合使用外部表和分区表

SQL优化

1. 写出Hive中split、coalesce及collect\_list函数的用法（可举例）？

split将字符串转化为数组，即：split('a,b,c,d' , ',') ==> ["a","b","c","d"]。

coalesce(T v1, T v2, …) 返回参数中的第一个非空值；如果所有值都为 NULL，那么返回NULL。

collect\_list列出该字段所有的值，不去重 => select collect\_list(id) from table。

1. 所有的Hive任务都会有MapReduce的执行吗？

不是，从Hive0.10.0版本开始，对于简单的不需要聚合的类似SELECT from LIMIT n语句，不需要起MapReduce job，直接通过Fetch task获取数据。

Fetch抓取是指，Hive中对某些情况的查询可以不必使用MapReduce计算。例如：SELECT \* FROM employees;在这种情况下，Hive可以简单地读取employee对应的存储目录下的文件，然后输出查询结果到控制台。

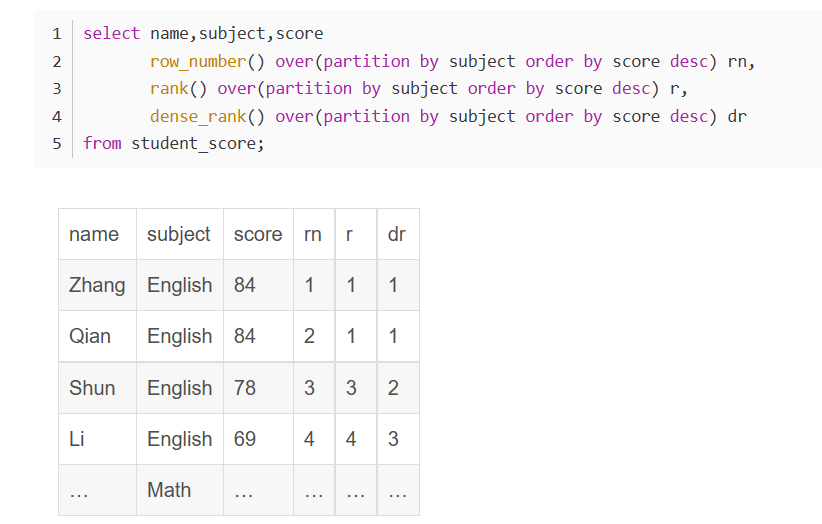
1. row\_number()，rank()和dense\_rank()的区别

都有对数据进行排序的功能

row\_number()：根据查询结果的顺序计算排序，多用于分页查询

rank()：排序相同时序号重复，总序数不变

dense\_rank()：排序相同时序号重复时，总序数减少



1. Hive导入数据的五种方式

1. Load方式，可以从本地或HDFS上导入，本地是copy，HDFS是移动本地：load data local inpath ‘/root/student.txt’ into table student;

HDFS：load data inpath ‘/user/hive/data/student.txt’ into table student;

2. Insert方式，往表里插入

insert into table student values(1,’zhanshan’);

3. As select方式，根据查询结果创建表并插入数据

create table if not exists stu1 as select id,name from student;

4. Location方式，创建表并指定数据的路径

Create external if not exists stu2 like student location '/user/hive/warehouse/student/student.txt';

5. Import方式，先从hive上使用export导出在导入

import table stu3 from ‘/user/export/student’;

16、Hive 有哪些方式保存元数据，各有哪些优缺点。

1)存储于 derby 数据库，此方法只能开启一个hive客户端，不推荐使用

2)存储于mysql数据库中，可以多客户端连接，推荐使用。

17、数据倾斜造成的原因有哪些？解决方案是什么？

1、不同数据类型关联产生数据倾斜

情形：比如用户表中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。

后果：处理此特殊值的reduce耗时；只有一个reduce任务。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。

解决方案：把数据类型转换成字符串类型

1. 在生产环境经常会用大量空值数据进入到一个reduce中去，导致数据倾斜

解决方案：自定义分区，将为空的key转变为字符串加随机数或纯随机数，将因空值而造成倾斜的数据分不到多个Reducer。

注意：对于异常值如果不需要的话，最好是提前在where条件里过滤掉，这样可以使计算量大大减少

18、Union与Union all区别

union会将联合的结果集去重，效果较union all差

union all 不会对结果集去重，所以效率高

1. Hive分区分桶的区别

Hive分区：是指按照数据表的某列或者某些列分为多个区，区从形式上可以理解为文件夹，比如我们要收集某个大型网站的日志数据，一个网站每天的日志数据存在同一张表上，由于每天会生成大量的日志，导致数据表的内容巨大，在查询时进行全表扫描耗费的资源非常多。那其实这个情况下，我们可以按照日期对数据表进行分区，不同日期的数据存放在不同的分区，在查询时只要指定分区字段的值就可以直接从该分区查找。

Hive分桶：分桶是相对分区进行更细粒度的划分。分桶将整个数据内容安装某列属性值得hash值进行区分，如要安装name属性分为3个桶，就是对name属性值的hash值对3取摸，按照取模结果对数据分桶。如取模结果为0的数据记录存放到一个文件，取模为1的数据存放到一个文件，取模为2的数据存放到一个文件。

1. Hive的执行顺序

from --> join on --> where --> group by --> 聚合函数 -->having -->select -->distinct--> order by

1. Hive文件存储格式

支持 Apache Hadoop 中使用的几种熟悉的文件格式，比如 TextFile、SequenceFile、RCFile、Avro、ORC、ParquetFile等。

TEXTFILE 和 SEQUENCEFILE 的存储格式都是基于行存储的，ORC 和 PARQUET 是基于列式存储的。

**第一种：TextFile**

1、存储方式：行存储。默认格式，如果建表时不指定默认为此格式。

2、每一行都是一条记录，每行都以换行符"\n"结尾。默认是不压缩，但可以采用多种压缩方式，但是部分压缩算法压缩数据后生成的文件是不支持split。

3、可结合Gzip、Bzip2等压缩方式一起使用（系统会自动检查，查询时会自动解压）,推荐选用可切分的压缩算法。

4、该类型的格式可以识别在hdfs上的普通文件格式（如txt、csv），因此该模式常用语仓库数据接入和导出层；

5、无法区分数据类型，各个字段都被认为是文本，但需要制定列分隔符和行分隔符。

**第二种：Sequence File**

1、SequenceFile是一种二进制文件，以<key,value>的形式序列化到文件中。存储方式：行存储；

2、支持三种压缩选择：NONE、RECORD、BLOCK。RECORD压缩率低，一般建议使用BLOCK压缩。

3、优势是文件和Hadoop API的MapFile是相互兼容的

4、缺点是由于该种模式是在textfile基础上加了些其他信息，故该类格式的大小要大于textfile，现阶段基本上不用。

**第三种：RC File**

1、存储方式：数据按行分块，每块按照列存储 。

A、首先，将数据按行分块，保证同一个record在一个块上，避免读一个记录需要读取多个block。

B、其次，块数据列式存储，有利于数据压缩和快速的列存取。

2、相对来说，RCFile对于提升任务执行性能提升不大，但是能节省一些存储空间。可以使用升级版的ORC格式。

**第四种：ORC File**

1、存储方式：数据按行分块，每块按照列存储

2、Hive提供的新格式，属于RCFile的升级版，性能有大幅度提升，而且数据可以压缩存储，压缩快，快速列存取。

3、ORC File会基于列创建索引，当查询的时候会很快，现阶段主要使用的文件格式。

**第五种：Parquet File**

1、存储方式：列式存储。

2、Parquet对于大型查询的类型是高效的。对于扫描特定表格中的特定列查询，Parquet特别有用。

Parquet一般使用Snappy、Gzip压缩，默认是Snappy。

1. Hive连接优化
2. MapReduce任务优化：

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 Map Reduce job。

即Hive通过一个优化可以在同一个Map Reduce job 中连接多个表（需要每个子句中有相同的连接键）。

1. JOIN连接时表的顺序（内连接）

Hive假定查询中最后一个表是最大的那个表，在对每行记录进行连接操作时，它会尝试将其他表缓存起来，然后扫描最后那个表进行计算。因此用户需要保证连续查询中表的大小从左往右是依次增加的。

用户也可以人为显示告诉查询优化器哪张表是大表，即使它在查询中不是位于最后面的

1. 连接前通过WHERE过滤数据加快查询数据，最直接的方式是使用嵌套SELECT语句：
2. map-side JOIN 优化含有小表的连接：

如果所有表中只有一张表是小表，则可以在最大的表通过mapper的时候将小表完全放到内存中。Hive可以在map端执行连接过程（称为mao-side JOIN），这是因为

Hive可以和内存中的小表逐一匹配，从而省略常规连接操作需要的reduce过程。不仅减少了reduce过程，有时也可以同时减少map过程的执行步骤。

1. Bucket Map Join

Bucket map join 需要待连接的两个表在连接字段上进行分桶（每个分桶对应 hdfs上的一个文件），而且小表的桶数需要时大表桶数的倍数。

1. 设置属性hive.optimize.bucketmapjoin= true控制 hive 执行 bucket map join；
2. 对小表的每个分桶文件建立一个 hashtable，并分发到所有做连接的 map 端；
3. map 端接受了 N（N 为小表分桶的个数）个小表的 hashtable，做连接 操作的时候，只需要将小表的一个 hashtable 放入内存即可，然后将大表的对应的 split 拿出来进行连接，所以其内存限制为小表中最大的那个 hashtable 的大小
4. Sort Merge Bucket Map Join

对于 bucket map join 中的两个表，如果每个桶内分区字段也是有序的，则还可以进行 sort merge bucket map join。

这样一来当两边 bucket 要做局部 join 的时候，只需要用类似 merge sort 算法中的 merge 操作一样把两个 bucket 顺序遍历一遍即可完成，这样甚至都不用把一个 bucket 完整的加载成 hashtable，而且可以做全连接操作。

操作时需要设置Hive的部分属性，从而使其进行Sort Merge Bucket Map Join

# 分布式一致性

**1. 为什么要做数据冗余？保障数据一致性方法？**

防止数据丢失、出错，进行容灾、读写加速。

* 2PC
* 3PC
* paxos
* raft
* zookeeper

**2. 什么是脑裂？zookeeper怎么解决脑裂？**

脑裂：一个整体内的节点互相联系着，一切运转正常。当网络问题导致心跳检测误判原master不在了，主备切换产生新的master。而后续被判定“死”的master由于网络恢复复活，此时网络出现两个主master，出现争抢共享资源造成系统混乱、数据损坏的现象，即：脑裂。

ZooKeeper默认采⽤了Quorums(法定⼈数)的⽅式: 只有获得超过半数节点的投票, 才能选举出leader，这种⽅式可以确保要么选出唯⼀的leader，要么选举失败。

**3. Zookeeper锁机制，Zookeeper读写锁实现？**

客户端尝试创建一个 znode 节点，比如 /lock 。那么第一个客户端就创建成功了，相当于拿到了锁；而其它的 客户端会创建失败（znode 已存在），获取锁失败。znode 应该被创建成 ephemeral 的。这是znode的一个特性，它保证如果创建 znode 的那个客户端崩溃了，那么相应的 znode 会被自动删除。这保证了锁一定会被释放。这个特性避免了设置锁的过期时间。

如果是写锁：

获取/lock 下的子节点

按照顺序号排序

检查此写锁之前是否还有其他锁，若有先注册对该写锁前一个锁的监听，然后阻塞该写锁获取，若监听到该写锁前一个锁已释放，则该写锁打开阻塞。

如果是读锁：

获取/lock 下的子节点

按照顺序号排序

检查此读锁之前是否有写锁，若有先注册对该读锁的前一个写锁的监听，然后阻塞该读锁的获取。若监听到该读锁前一个写锁已释放，则该读锁打开阻塞

**4. 什么是“羊群效应”，在Zookeeper中怎么优化？**

持有锁的客户端访问共享资源完成后，将 znode 删掉，这样其它客户端接下来就能来获取锁了。如上所述的基于ZooKeeper 的分布式锁的实现，并不是最优的,它会引发 “herd effect”（羊群效应），降低获取锁的性能。可以设置锁节点为顺序临时节点，后面的节点 watch 前面的节点，当前面的节点删除时唤醒后面的节点从而避免羊群效应。

**5. Zookeeper为什么是CP架构？AP结构用在哪里？**

CAP理论：在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

强调的是一致性（C），在事务消息写入时，为尽最大努力保证数据视图一致性

ZooKeeper为了保证在集群脑裂分区（P）情况下数据一致性（C），其放弃了可用性（A），所以在集群部分节点或者网络发生故障，master离线的时候，集群将进行崩溃恢复选注，在选主期间，整个集群对外是不可用的，一直持续到选主完成，主从数据同步成功为止

明显AP结构选择了高可用和分区容错性，此时，那个失去联系的节点依然可以向系统提供服务，不过它的数据就不能保证是同步的了（失去了C属性）。Eureka就是一个AP架构的例子，当Eureka客户端心跳消失的时候，那Eureka服务端就会启动自我保护机制，不会剔除该EurekaClient客户端的服务，依然可以提供需求；

**6. 什么是BASE理论，什么是最终一致性？**

是在实践过程中基于CAP理论演化而来的，基本可用，软状态，最终一致性

基本可用：是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。即保证核心可用

软状态：指系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延迟

最终一致性：所有数据副本经过一定时间后，最终能够达到一致的状态

核心思想：既然无法做到强一致性，但每个应用都可以跟据自身业务特点，采用适当的方式来使系统达到最终一致性，也就是牺牲数据的一致性来满足系统的高可用性，系统中一部分数据不可用或不一致时，仍需要保持系统整体基本可用。

**7. Zookeeper的ZAB协议和Paxos有什么区别？**

Paxos算法中，一个新的选举产生的主进程会进行两个阶段的工作

1、读阶段，新的主进程会通过和所有其他进程进行通信的方式来搜集上一个主进程提出的提案，并将它们提交。

2、写阶段，当前主进程开始提出它自己的提案。

ZAB在Paxos基础上额外添加一个同步阶段。同步阶段之前，ZAB协议存在一个和Paxos读阶段类似的发现（Discovery）阶段

同步阶段中，新的Leader会确保存在过半的Follower已经提交了之前Leader周期中的所有事务Proposal

发现阶段的存在，确保所有进程都已经完成对之前所有事物Proposal的提交

ZAB协议主要用于构建一个高可用的分布式数据主备系统，例如ZooKeeper

Paxos算法则是用于构建一个分布式的一致性状态机系统

**8. ZAB协议和Raft有什么区别？**







**9. Zookeeper的选举算法：**

Zookeeper 在配置文件中并没有指定 Master 和 Slave。但是，Zookeeper 工作时， 是有一个节点为 Leader，其他则为 Follower，而这个 Leader 是通过内部的选举机制临时产生的。

每个 Server 首先都提议自己是 Leader，并为自己投票，然后将投票结果与其他 Server 的选票进行对比，权重大的胜出，使用权重较大的选票更新自身的投票箱，我们介绍下服务器启动时期的 Leader 选举。

1）每一个 Server 都会发出一个投票

在集群初次启动时，每个 Server 都会推荐自己为 Leader，然后各自将这个投票发给集群中其他 Server。

2）接收来自各个 Server 的投票

每个 Server 在接收到其他 Server 的投票后，首先会判断该票的有效性，包括检查是否本轮投票，是否来自 Looking 状态的 Server。（Looking 状态表示当前集群正处于选举状态）

3）处理投票

针对每一个投票，Server 都会将别人的投票和自己的投票进行 PK，计算出 Zxid 最大的 Server，并将该 Server 设置成下一次投票推荐的 Server。

4）统计投票

每次投票结束之后，都会统计所有投票，获取投票最多的 Server 将成为获胜者，如果获胜者的票数超过集群个数的一半，则该 Server 将为推选为 Leader。否则继续投票，直至 Leader 被选举出来。

5）改变服务器状态

一旦 Leader 确定后，Leader 会通知其他 Follower 集群已经成为 Uptodate 状态，Follower 在收到 Uptodate 消息后，接收 Client 的请求并开始对外提供服务。

**10. 什么是Paxos协议，其与拜占庭将军协议应用场景存在什么区别？**

在常见的分布式系统中，总会发生诸如机器宕机或网络异常（包括消息的延迟，丢失，重复，乱序，网络分区）等情况。Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中，快速且正确的在集群内布对某个数据的值达成一致，并且保证不论发生以上任何异常，都不会破坏整个系统的一致性。

proposer、Acceptor、Learner

是否存在拜占庭错误。人为恶意篡改包括“精神分裂式投票”，中继篡改上一个办事处的投票信息。对应到分布式系统面临的问题就是：消息被伪造、系统安全攻击等等。发生的人为恶意篡改的过程就可以称之为系统发生了拜占庭错误

**11. raft选举过程，prevote**

节点启动时都是follower状态；在一段时间内如果没有收到来自leader的心跳，从follower切换到candidate，发起选举；如果收到majority的造成票（含自己的一票）则切换到leader状态；如果发现其他节点比自己更新，则主动切换到follower。

增加节点本地的 current term ，切换到candidate状态

投自己一票

并行给其他节点发送 RequestVote RPCs

等待其他节点的回复

为了防止网络分区后，被隔离的几点始终自增任期，再log落后的情况下成为leader，需要进行预选举。在预选举，不增加任期，如果预选举获胜，则增加任期参加选举

**12. raft怎么解决脑裂**

网络分区导致老的leader在少数区，那么，由于其发现自己无法联系大多数节点，因此不会进行数据写。多数区重新选举出新leader

老leader在多数区，少数区没法选举，问题不大

**13. raft怎么保证数据一致性，raft提案过程描述**

整个系统只有一个领导者，所有的外部请求只会由领导者处理，即使是跟随者收到请求，也会转发给领导者处理，这是数据一致性的保障之一。

整个系统对外可用需要达到多数节点存活的情况下才行，也就是说领导者是由多数节点达成一致选举出来的，这是数据一致性的又一个保障。

Raft的数据持久化分为快照和事务日志。跟随者会始终保持跟领导者的数据同步，这是数据一致性的又一个保障。

**14. 什么是分布式事务？分布式事务有哪些解决方案？**

一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

2PC，3PC，TCC

**15. 两阶段提交协议和三阶段提交协议的区别，三阶段提交协议解决了什么问题？还存在什么问题？**

与二阶段相比，三阶段：

1. 引入超时机制：同时在协调者和参与者中都引入超时机制。

2. 在第一阶段和第二阶段中插入一个准备阶段：保证了在最后提交阶段之前各参与节点的状态是一致的，即三阶段提交就有 CanCommit（无锁状态）、PreCommit（无锁状态）、DoCommit 三个阶段。

存在问题：

1. 三阶段的超级机制，解决了阻塞问题。

2. CanCommit的预先铺垫 过渡到 PreCommit 的预备阶段，相当于让我们有理由相信 DoCommit 成功提交的几率很大，但是由于网络原因导致的数据不一致问题依然存在。

**16. 什么是TCC，TCC存在什么问题？**

TCC方案分为Try Confirm Cancel三个阶段，属于补偿性分布式事务。

Try：尝试待执行的业务

这个过程并未执行业务，只是完成所有业务的一致性检查，并预留好执行所需的全部资源

Confirm：执行业务

这个过程真正开始执行业务，由于Try阶段已经完成了一致性检查，因此本过程直接执行，而不做任何检查。并且在执行的过程中，会使用到Try阶段预留的业务资源。

Cancel：取消执行的业务

若业务执行失败，则进入Cancel阶段，它会释放所有占用的业务资源，并回滚Confirm阶段执行的操作。

TCC需要编写大量的补偿事务的代码，比较冗余