尚硅谷大数据技术之大数据基础阶段考试题（二）

(作者：尚硅谷大数据研发部)

版本：V1.0

一 Zookeeper

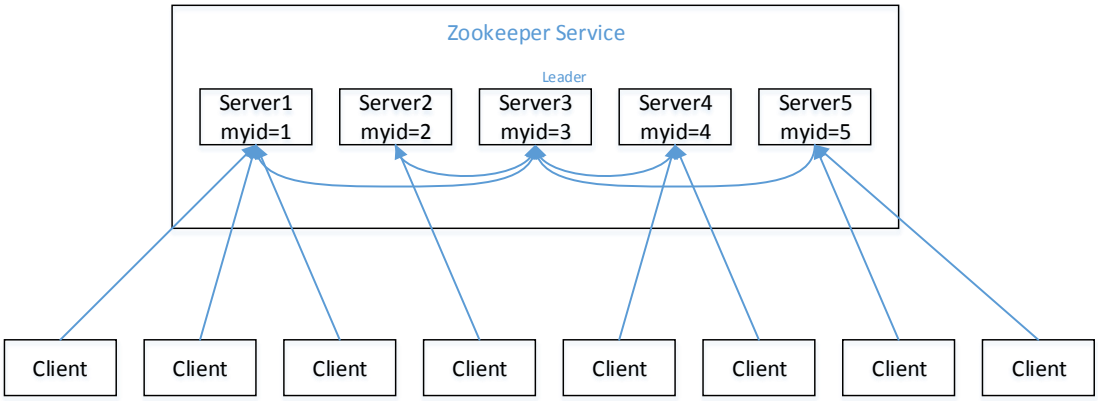
1 请简述 ZooKeeper 的选举机制

1）半数机制（Paxos 协议）：集群中半数以上机器存活，集群可用。所以 zookeeper 适合装在奇数台机器上。

2）Zookeeper 虽然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，Leader 是通过内部的选举机制临时产生的

3）以一个简单的例子来说明整个选举的过程。

假设有五台服务器组成的 zookeeper 集群，它们的 id 从 1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么。



（1）服务器 1 启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是 LOOKING 状态。

（2）服务器 2 启动，它与最开始启动的服务器 1 进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以 id 值较大的服务器 2 胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是 3)，所以服务器 1、2 还是继续保持 LOOKING 状态。

（3）服务器 3 启动，根据前面的理论分析，服务器 3 成为服务器 1、2、3 中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的 leader。

（4）服务器 4 启动，根据前面的分析，理论上服务器 4 应该是服务器 1、2、3、4 中最

大的，但是由于前面已经有半数以上的服务器选举了服务器 3，所以它只能接收当小弟的命了。

（5）服务器 5 启动，同 4 一样当小弟。

2 ZooKeeper 的监听原理是什么？

1）首先要有一个 main()线程

2）在 main 线程中创建 Zookeeper 客户端，这时就会创建两个线程，一个负责网络连接通信（connet），一个负责监听（listener）。

3）通过 connect 线程将注册的监听事件发送给 Zookeeper。

4）在 Zookeeper 的注册监听器列表中将注册的监听事件添加到列表中。

5）Zookeeper 监听到有数据或路径变化，就会将这个消息发送给 listener 线程。

6）listener 线程内部调用了 proces()方法。

3 ZooKeeper 的部署方式有哪几种？集群中的角色有哪些？集群最少需要几台机器？

1） 单机模式、伪集群模式和集群模式

2） Leader 和 Follower

3） 3

4 ZooKeeper 的常用命令

ls create get delete set…

二 Hive

1 Hive 表关联查询，如何解决数据倾斜的问题？

1） 过滤掉脏数据：如果大 key 是无意义的脏数据，直接过滤掉。本场景中大 key 无实际意义，为非常脏数据，直接过滤掉。

2）数据预处理：数据做一下预处理，尽量保证 join 的时候，同一个 key 对应的记录不要有太多。

3） 增加 reduce 个数：如果数据中出现了多个大 key，增加 reduce 个数，可以让这些大 key 落到同一个 reduce 的概率小很多。

4） 转换为 mapjoin：如果两个表 join 的时候，一个表为小表，可以用 mapjoin 做。

5） 大 key 单独处理：将大 key 和其他 key 分开处理

6）hive.optimize.skewjoin：会将一个 join sql 分为两个 job。另外可以同时设置下

hive.skewjoin.key，默认为 10000。参考：

https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties

参数对 full outer join 无效。

7）调整内存设置：适用于那些由于内存超限内务被 kill 掉的场景。通过加大内存起码能让任务跑起来，不至于被杀掉。该参数不一定会明显降低任务执行时间。如：

setmapreduce.reduce.memory.mb=5120 ;

setmapreduce.reduce.java.opts=-Xmx5000M -XX:MaxPermSize=128m ;

2 请谈一下 Hive 的特点，Hive 和 RDBMS 有什么异同？

特点：

1）Hive 处理的数据存储在 HDFS

2）Hive 分析数据底层的实现是 MapReduce

3）执行程序运行在 Yarn 上

异同：

1）查询语言：由于 SQL 被广泛的应用在数据仓库中，因此，专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便的使用 Hive 进行开发。

2） 数据存储位置：Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。

3）数据更新：由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的。因此，Hive 中不建议对数据的改写，所有的数据都是在加载的时候确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使用 INSERT INTO … VALUES 添加数据，使用 UPDATE … SET 修改数据。

4）索引：Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入， Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。

5） 执行：Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的。而数据库通常有自己的执行引擎。

6） 执行延迟：Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。

7） 可扩展性：由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性是和 Hadoop 的可扩展性是一致的（世界上最大的 Hadoop 集群在 Yahoo!，2009 年的规模在 4000 台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。

8）数据规模：由于 Hive 建立在集群上并可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。

3 请说明 Hive 中 Sort By，Order By，Cluster By，Distrbute By 各代表什么意思？

Sort By：每个 Reducer 内部有序；

Order By：全局排序，只有一个 Reducer；

Cluster By：当 distribute by 和 sorts by 字段相同时，可以使用 cluster by 方式。cluster by 除了具有 distribute by 的功能外还兼具 sort by 的功能。但是排序只能是倒序排序，不能指定排序规则为 ASC 或者 DESC。

Distrbute By：类似 MR 中 partition，进行分区，结合 sort by 使用，Hive 要求 DISTRIBUTE BY 语句要写在 SORT BY 语句之前。

4 Hive 有哪些方式保存元数据，各有哪些特点？

1）Single User Mode：默认安装 hive，hive 是使用 derby 内存数据库保存 hive 的元数据，这样是不可以并发调用 hive 的。

2）User Mode：通过网络连接到一个数据库中，是最经常使用到的模式。假设使用本机 mysql 服务器存储元数据。这种存储方式需要在本地运行一个 mysql 服务器，可并发调用

3）Remote Server Mode：在服务器端启动一个 MetaStoreServer，客户端利用 Thrift 协议通过 MetaStoreServer 访问元数据库。

5 Hive 内部表和外部表的区别？

1）默认创建的表都是管理表，有时也被称为内部表。因为这种表，Hive 会（或多或少地）控制着数据的生命周期。Hive 默认情况下会将这些表的数据存储在由配置项 hive.metastore.warehouse.dir(例如，/user/hive/warehouse)所定义的目录的子目录下。当我们删除一个管理表时，Hive 也会删除这个表中数据。管理表不适合和其他工具共享数据。

2）Hive 并非认为其完全拥有这份数据。删除该表并不会删除掉这份数据，不过描述表的元数据信息会被删除掉。

6 写出将 text.txt 文件放入 Hive 中 test 表‘2016-10-10’ 分区的语句，test 的分区字段是 l_date。

load data local inpath ‘/text.txt’ into table test partition(l_date=’ 2016-10-10’);

7 Hive 自定义 UDF 函数的流程?

1）自定义一个 Java 类

2）继承 UDF 类

3）重写 evaluate 方法
4）打成 jar 包
6）在 hive 执行 add jar 方法
7）在 hive 执行创建模板函数
8）hql 中使用

8 对于 Hive，你写过哪些 udf 函数，作用是什么？

时间日期类 UDF 函数，传入时间，返回年，月，日…

9 Hive 中的压缩格式 TextFile、SequenceFile、RCfile 、ORCfile 各有什么区别？

TextFile: 默认格式，数据不做压缩，磁盘开销大，数据解析开销大。可结合 Gzip、Bzip2 使用，但使用 Gzip 这种方式，hive 不会对数据进行切分，从而无法对数据进行并行操作。

SequenceFile: SequenceFile 是 Hadoop API 提供的一种二进制文件，它将数据以<key,value>的形式序列化到文件中。这种二进制文件内部使用 Hadoop 的标准的 Writable 接口实现序列化和反序列化。它与 Hadoop API 中的 MapFile 是互相兼容的。Hive 中的 SequenceFile 继承自 Hadoop API 的 SequenceFile，不过它的 key 为空，使用 value 存放实际的值， 这样是为了避免 MR 在运行 map 阶段的排序过程。

RCFile: RCFile 是 Hive 推出的一种专门面向列的数据格式。 它遵循“先按列划分，再垂直划分”的设计理念。当查询过程中，针对它并不关心的列时，它会在 IO 上跳过这些列。需要说明的是，RCFile 在 map 阶段从 远端拷贝仍然是拷贝整个数据块，并且拷贝到本地目录后 RCFile 并不是真正直接跳过不需要的列，并跳到需要读取的列， 而是通过扫描每一个 row group 的头部定义来实现的，但是在整个 HDFS Block 级别的头部并没有定义每个列从哪个 row group 起始到哪个 row group 结束。所以在读取所有列的情况下，RCFile 的性能反而没有 SequenceFile 高。

ORCfile: ORC 是列式存储，有多种文件压缩方式，并且有着很高的压缩比。文件是可切分（Split）的。因此，在 Hive 中使用 ORC 作为表的文件存储格式，不仅节省 HDFS 存储资源,查询任务的输入数据量减少，使用的 MapTask 也就减少了。提供了多种索引，row group index、bloom filter index。ORC 可以支持复杂的数据结构（比如 Map 等）

10 Hive join 过程中大表小表的放置顺序？

小表在前，大表在后。Hive 会将小表进行缓存。在 0.7 版本后。也能够用配置来自己主动优化：set hive.auto.convert.join=true;

11 Hive 的两张表关联，使用 MapReduce 怎么实现？

可以将小表缓存在 map 端实现 map 端 join 防止数据倾斜。并将 join 字段作为 key，在

Reducer 阶段实际执行相应的业务处理。

12 所有的 Hive 任务都会有 MapReduce 的执行吗？

不是。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。

在 hive-default.xml.template 文件中 hive.fetch.task.conversion 默认是 more，老版本 hive 默认是 minimal，该属性修改为 more 以后，在全局查找、字段查找、limit 查找等都不走 mapreduce。

13 Hive 的函数：UDF、UDAF、UDTF 的区别？

（1）UDF（User-Defined-Function）
一进一出
（2）UDAF（User-Defined Aggregation Function）
聚集函数，多进一出
类似于：count/max/min
（3）UDTF（User-Defined Table-Generating Functions）
一进多出
如 lateral view explore()

14 说说对 Hive 桶表的理解？

分区针对的是数据的存储路径，分桶针对的是数据文件。分区提供一个隔离数据和优化查询的便利方式。不过，并非所有的数据集都可形成合理的分区，特别是之前所提到过的要确定合适的划分大小这个疑虑。

分桶是将数据集分解成更容易管理的若干部分的另一个技术。

15 Hive 可以像关系型数据库那样建立多个库吗？

可以

16 Hive 实现统计的查询语句是什么？

聚合函数：count(),sum(),avg(),max(),min()

17 Hive 优化措施

17.1 Fetch 抓取

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。

在 hive-default.xml.template 文件中 hive.fetch.task.conversion 默认是 more，老版本 hive 默认是 minimal，该属性修改为 more 以后，在全局查找、字段查找、limit 查找等都不走 mapreduce。

```
<property>
    <name>hive.fetch.task.conversion</name>
    <value>more</value>
    <description>
      Expects one of [none, minimal, more].
      Some select queries can be converted to single FETCH task minimizing latency.
      Currently the query should be single sourced not having any subquery and
should not have
      any aggregations or distincts (which incurs RS), lateral views and joins.
      0. none : disable hive.fetch.task.conversion
      1. minimal : SELECT STAR, FILTER on partition columns, LIMIT only
      2. more  : SELECT, FILTER, LIMIT only (support TABLESAMPLE and virtual
columns)
    </description>
  </property>
```

案例实操：

1）把 hive.fetch.task.conversion 设置成 none，然后执行查询语句，都会执行 mapreduce 程序。

```
hive (default)> set hive.fetch.task.conversion=none;
hive (default)> select * from emp;
hive (default)> select ename from emp;
hive (default)> select ename from emp limit 3;
```

2）把 hive.fetch.task.conversion 设置成 more，然后执行查询语句，如下查询方式都不会执行 mapreduce 程序。

```
hive (default)> set hive.fetch.task.conversion=more;
hive (default)> select * from emp;
hive (default)> select ename from emp;
hive (default)> select ename from emp limit 3;
```

17.2 本地模式

大多数的 Hadoop Job 是需要 Hadoop 提供的完整的可扩展性来处理大数据集的。不过，有时 Hive 的输入数据量是非常小的。在这种情况下，为查询触发执行任务消耗的时间可能会比实际 job 的执行时间要多的多。对于大多数这种情况，Hive 可以通过本地模式在单台机器上处理所有的任务。对于小数据集，执行时间可以明显被缩短。

用户可以通过设置 hive.exec.mode.local.auto 的值为 true，来让 Hive 在适当的时候自动启动这个优化。

```
set hive.exec.mode.local.auto=true;  //开启本地 mr
//设置 local mr 的最大输入数据量，当输入数据量小于这个值时采用 local  mr 的方式，默认
为 134217728，即 128M
set hive.exec.mode.local.auto.inputbytes.max=50000000;
//设置 local mr 的最大输入文件个数，当输入文件个数小于这个值时采用 local mr 的方式，默
认为 4
```

```
set hive.exec.mode.local.auto.input.files.max=10;
```

案例实操：

1）开启本地模式，并执行查询语句

```
hive (default)> set hive.exec.mode.local.auto=true;
hive (default)> select * from emp cluster by deptno;
Time taken: 1.328 seconds, Fetched: 14 row(s)
```

2）关闭本地模式，并执行查询语句

```
hive (default)> set hive.exec.mode.local.auto=false;
hive (default)> select * from emp cluster by deptno;
Time taken: 20.09 seconds, Fetched: 14 row(s)
```

17.3 表的优化

17.3.1 小表、大表 Join

将 key 相对分散，并且数据量小的表放在 join 的左边，这样可以有效减少内存溢出错误发生的几率；再进一步，可以使用 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。

实际测试发现：新版的 hive 已经对小表 JOIN 大表和大表 JOIN 小表进行了优化。小表放在左边和右边已经没有明显区别。

案例实操

1．需求

测试大表 JOIN 小表和小表 JOIN 大表的效率

2．建大表、小表和 JOIN 后表的语句

```
// 创建大表
create table bigtable(id bigint, time bigint, uid string, keyword string, url rank int, click num int, click url string) row format delimited fields terminated by '\t';
// 创建小表
create table smalltable(id bigint, time bigint, uid string, keyword string, url_rank int, click_num int, click_url string) row format delimited fields terminated by '\t';
// 创建 join 后表的语句
create table jointable(id bigint, time bigint, uid string, keyword string, url_rank int, click_num int, click_url string) row format delimited fields terminated by '\t';
```

3．分别向大表和小表中导入数据

```
hive (default)> load data local inpath '/opt/module/datas/bigtable' into table bigtable;
hive (default)>load data local inpath '/opt/module/datas/smalltable' into table smalltable;
```

4．关闭 mapjoin 功能（默认是打开的）

```
set hive.auto.convert.join = false;
```

5．执行小表 JOIN 大表语句

```
insert overwrite table jointable
select b.id, b.time, b.uid, b.keyword, b.url_rank, b.click_num, b.click_url
from smalltable s
left join bigtable  b
on b.id = s.id;
```

Time taken: 35.921 seconds

6．执行大表 JOIN 小表语句

```
insert overwrite table jointable
select b.id, b.time, b.uid, b.keyword, b.url_rank, b.click_num, b.click_url
from bigtable  b
left join smalltable  s
```

```
on s.id = b.id;
```

Time taken: 34.196 seconds

17.3.2 大表 Join 大表

1．空 KEY 过滤

有时 join 超时是因为某些 key 对应的数据太多，而相同 key 对应的数据都会发送到相同的 reducer 上，从而导致内存不够。此时我们应该仔细分析这些异常的 key，很多情况下，这些 key 对应的数据是异常数据，我们需要在 SQL 语句中进行过滤。例如 key 对应的字段为空，操作如下：

案例实操

（1）配置历史服务器

配置 mapred-site.xml

```
<property>
<name>mapreduce.jobhistory.address</name>
<value>hadoop102:10020</value>
</property>
<property>
    <name>mapreduce.jobhistory.webapp.address</name>
    <value>hadoop102:19888</value>
</property>
```

启动历史服务器

```
sbin/mr-jobhistory-daemon.sh start historyserver
```

查看 jobhistory

http://192.168.1.102:19888/jobhistory

（2）创建原始数据表、空 id 表、合并后数据表

```
// 创建原始表
create table ori(id bigint, time bigint, uid string, keyword string, url rank int,
click_num int, click_url string) row format delimited fields terminated by '\t';
// 创建空 id 表
create table nullidtable(id bigint, time bigint, uid string, keyword string,
url_rank int, click_num int, click_url string) row format delimited fields
terminated by '\t';
// 创建 join 后表的语句
create table jointable(id bigint, time bigint, uid string, keyword string, url rank
int, click num int, click url string) row format delimited fields terminated by
'\t';
```

（3）分别加载原始数据和空 id 数据到对应表中

```
hive (default)> load data local inpath '/opt/module/datas/ori' into table ori;
hive (default)> load data local inpath '/opt/module/datas/nullid' into table
nullidtable;
```

（4）测试不过滤空 id

```
hive (default)> insert overwrite table jointable
select n.* from nullidtable n left join ori o on n.id = o.id;
```

Time taken: 42.038 seconds

Time taken: 37.284 seconds

（5）测试过滤空 id

```
hive (default)> insert overwrite table jointable
select n.* from (select * from nullidtable where id is not null ) n  left join ori
o on n.id = o.id;
```

Time taken: 31.725 seconds

Time taken: 28.876 seconds

2．空 key 转换

有时虽然某个 key 为空对应的数据很多，但是相应的数据不是异常数据，必须要包含在 join 的结果中，此时我们可以表 a 中 key 为空的字段赋一个随机的值，使得数据随机均匀地分不到不同的 reducer 上。例如：

案例实操：

不随机分布空 null 值：

（1）设置 5 个 reduce 个数

set mapreduce.job.reduces = 5;

（2）JOIN 两张表

```
insert overwrite table jointable
select n.* from nullidtable n left join ori b on n.id = b.id;
```

结果：如图 6-13 所示，可以看出来，出现了数据倾斜，某些 reducer 的资源消耗远大于其他 reducer。

Show 20 entries

Task							
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Shuffle Finish Time	Merge Finish Time
task 1506334829052 0015 r 000000	SUCCEEDED	Mon Sep 25 19:18:05 +0800 2017	Mon Sep 25 19:18:24 +0800 2017	18sec	Mon Sep 25 19:18:05 +0800 2017	Mon Sep 25 19:18:18 +0800 2017	Mon Sep 25 19:18:18 +0800 2017
task 1506334829052 0015 r 000001	SUCCEEDED	Mon Sep 25 19:18:05 +0800 2017	Mon Sep 25 19:18:18 +0800 2017	12sec	Mon Sep 25 19:18:05 +0800 2017	Mon Sep 25 19:18:13 +0800 2017	Mon Sep 25 19:18:13 +0800 2017
task 1506334829052 0015 r 000004	SUCCEEDED	Mon Sep 25 19:18:06 +0800 2017	Mon Sep 25 19:18:18 +0800 2017	11sec	Mon Sep 25 19:18:06 +0800 2017	Mon Sep 25 19:18:14 +0800 2017	Mon Sep 25 19:18:14 +0800 2017
task 1506334829052 0015 r 000003	SUCCEEDED	Mon Sep 25 19:18:06 +0800 2017	Mon Sep 25 19:18:17 +0800 2017	10sec	Mon Sep 25 19:18:06 +0800 2017	Mon Sep 25 19:18:14 +0800 2017	Mon Sep 25 19:18:14 +0800 2017
task 1506334829052 0015 r 000002	SUCCEEDED	Mon Sep 25 19:18:05 +0800 2017	Mon Sep 25 19:18:16 +0800 2017	10sec	Mon Sep 25 19:18:05 +0800 2017	Mon Sep 25 19:18:13 +0800 2017	Mon Sep 25 19:18:13 +0800 2017

图 6-13 空 key 转换

随机分布空 null 值

（1）设置 5 个 reduce 个数

set mapreduce.job.reduces = 5;

（2）JOIN 两张表

```
insert overwrite table jointable
select n.* from nullidtable n full join ori o on
case when n.id is null then concat('hive', rand()) else n.id end = o.id;
```

结果：如图 6-14 所示，可以看出来，消除了数据倾斜，负载均衡 reducer 的资源消耗

Task					Successful			
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Shuffle Finish Time	Merge Finish Time	F
task 1506334829052 0016 r 000000	SUCCEEDED	Mon Sep 25 19:20:43 +0800 2017	Mon Sep 25 19:21:04 +0800 2017	20sec	Mon Sep 25 19:20:43 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mon Sep 25 19:20:56 +0800 2017	Mo 19: 20:
task 1506334829052 0016 r 000001	SUCCEEDED	Mon Sep 25 19:20:43 +0800 2017	Mon Sep 25 19:21:00 +0800 2017	16sec	Mon Sep 25 19:20:43 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mo 19: 20:
task 1506334829052 0016 r 000003	SUCCEEDED	Mon Sep 25 19:20:44 +0800 2017	Mon Sep 25 19:21:00 +0800 2017	15sec	Mon Sep 25 19:20:44 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mo 19: 20:
task 1506334829052 0016 r 000002	SUCCEEDED	Mon Sep 25 19:20:43 +0800 2017	Mon Sep 25 19:20:59 +0800 2017	15sec	Mon Sep 25 19:20:43 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mo 19: 20:
task 1506334829052 0016 r 000004	SUCCEEDED	Mon Sep 25 19:20:44 +0800 2017	Mon Sep 25 19:20:59 +0800 2017	14sec	Mon Sep 25 19:20:44 +0800 2017	Mon Sep 25 19:20:54 +0800 2017	Mon Sep 25 19:20:55 +0800 2017	Mo 19: 20:
ID	State	Start Time	Finish Time	Elapsed	Start Time	Shuffle Time	Merge Time	Fin

图 6-14 随机分布空值

17.3.3 MapJoin

如果不指定 MapJoin 或者不符合 MapJoin 的条件，那么 Hive 解析器会将 Join 操作转换成 Common Join，即：在 Reduce 阶段完成 join。容易发生数据倾斜。可以用 MapJoin 把小表全部加载到内存在 map 端进行 join，避免 reducer 处理。

1．开启 MapJoin 参数设置

（1）设置自动选择 Mapjoin

```
set hive.auto.convert.join = true; 默认为 true
```

（2）大表小表的阈值设置（默认 25M 一下认为是小表）：

```
set hive.mapjoin.smalltable.filesize=25000000;
```

2．MapJoin 工作机制，如图 6-15 所示

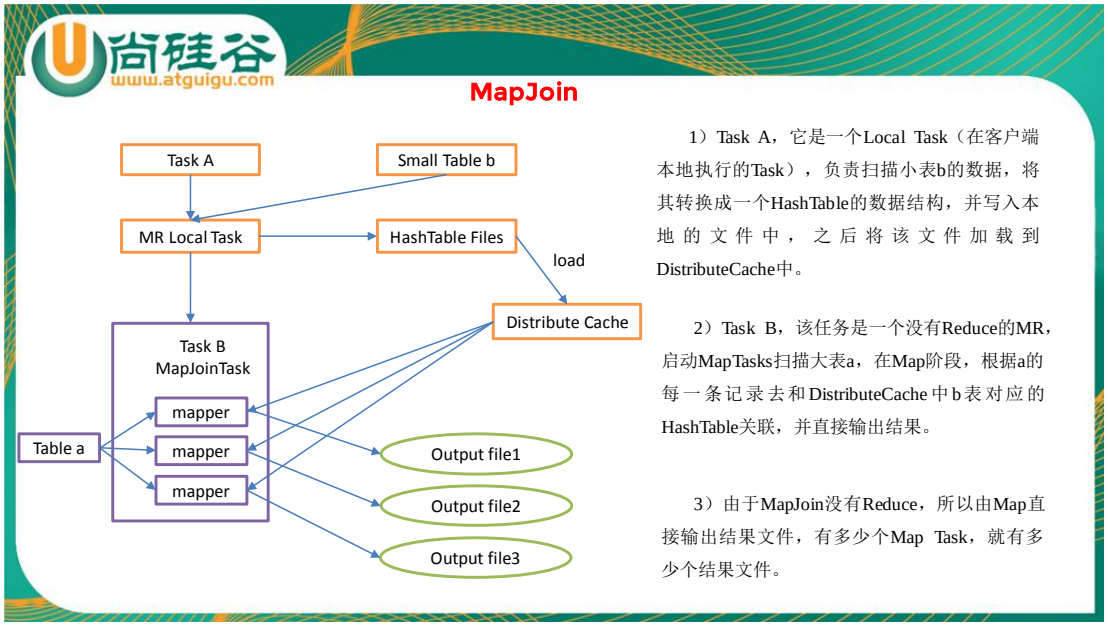


图 6-15　MapJoin 工作机制

案例实操：

（1）开启 Mapjoin 功能

```
set hive.auto.convert.join = true; 默认为 true
```

（2）执行小表 JOIN 大表语句

```
insert overwrite table jointable
select b.id, b.time, b.uid, b.keyword, b.url_rank, b.click_num, b.click_url
from smalltable s
join bigtable  b
on s.id = b.id;
```

Time taken: 24.594 seconds

（3）执行大表 JOIN 小表语句

```
insert overwrite table jointable
select b.id, b.time, b.uid, b.keyword, b.url_rank, b.click_num, b.click_url
from bigtable  b
join smalltable  s
on s.id = b.id;
```

Time taken: 24.315 seconds

17.3.4 Group By

默认情况下，Map 阶段同一 Key 数据分发给一个 reduce，当一个 key 数据过大时就倾斜了。

并不是所有的聚合操作都需要在 Reduce 端完成，很多聚合操作都可以先在 Map 端进行部分聚合，最后在 Reduce 端得出最终结果。

1．开启 Map 端聚合参数设置

（1）是否在 Map 端进行聚合，默认为 True

```
hive.map.aggr = true
```

（2）在 Map 端进行聚合操作的条目数目

```
hive.groupby.mapaggr.checkinterval = 100000
```

（3）有数据倾斜的时候进行负载均衡（默认是 false）

```
hive.groupby.skewindata = true
```

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。

17.3.5 Count(Distinct) 去重统计

数据量小的时候无所谓，数据量大的情况下，由于 COUNT DISTINCT 操作需要用一个 Reduce Task 来完成，这一个 Reduce 需要处理的数据量太大，就会导致整个 Job 很难完成，一般 COUNT DISTINCT 使用先 GROUP BY 再 COUNT 的方式替换：

案例实操

1．创建一张大表

```
hive (default)> create table bigtable(id bigint, time bigint, uid string, keyword
string, url_rank int, click_num int, click_url string) row format delimited
fields terminated by '\t';
```

2．加载数据

```
hive (default)> load data local inpath '/opt/module/datas/bigtable' into table
 bigtable;
```

3．设置 5 个 reduce 个数

```
set mapreduce.job.reduces = 5;
```

4．执行去重 id 查询

```
hive (default)> select count(distinct id) from bigtable;
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 7.12 sec   HDFS Read: 120741990
HDFS Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 7 seconds 120 msec
OK
c0
100001
Time taken: 23.607 seconds, Fetched: 1 row(s)
```

5．采用 GROUP by 去重 id

```
hive (default)> select count(id) from (select id from bigtable group by id) a;
Stage-Stage-1: Map: 1  Reduce: 5   Cumulative CPU: 17.53 sec   HDFS Read: 120752703
HDFS Write: 580 SUCCESS
Stage-Stage-2: Map: 1  Reduce: 1   Cumulative CPU: 4.29 sec   HDFS Read: 9409 HDFS
Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 21 seconds 820 msec
OK
_c0
100001
Time taken: 50.795 seconds, Fetched: 1 row(s)
```

虽然会多用一个 Job 来完成，但在数据量大的情况下，这个绝对是值得的。

17.3.6 笛卡尔积

尽量避免笛卡尔积，join 的时候不加 on 条件，或者无效的 on 条件，Hive 只能使用 1 个 reducer 来完成笛卡尔积。

17.3.7 行列过滤

列处理：在 SELECT 中，只拿需要的列，如果有，尽量使用分区过滤，少用 SELECT *。

行处理：在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在 Where 后面，那么就会先全表关联，之后再过滤，比如：

案例实操：

1．测试先关联两张表，再用 where 条件过滤

```
hive (default)> select o.id from bigtable b
join ori o on o.id = b.id
```

```
where o.id <= 10;
```

Time taken: 34.406 seconds, Fetched: 100 row(s)

2．通过子查询后，再关联表

```
hive (default)> select b.id from bigtable b
join (select id from ori where id <= 10 ) o on b.id = o.id;
```

Time taken: 30.058 seconds, Fetched: 100 row(s)

17.3.8 动态分区调整

关系型数据库中，对分区表 Insert 数据时候，数据库自动会根据分区字段的值，将数据插入到相应的分区中，Hive 中也提供了类似的机制，即动态分区(Dynamic Partition)，只不过，使用 Hive 的动态分区，需要进行相应的配置。

1．开启动态分区参数设置

（1）开启动态分区功能（默认 true，开启）

hive.exec.dynamic.partition=true

（2）设置为非严格模式（动态分区的模式，默认 strict，表示必须指定至少一个分区为静态分区，nonstrict 模式表示允许所有的分区字段都可以使用动态分区。）

```
hive.exec.dynamic.partition.mode=nonstrict
```

（3）在所有执行 MR 的节点上，最大一共可以创建多少个动态分区。

```
hive.exec.max.dynamic.partitions=1000
```

（4）在每个执行 MR 的节点上，最大可以创建多少个动态分区。该参数需要根据实际的数据来设定。比如：源数据中包含了一年的数据，即 day 字段有 365 个值，那么该参数就需要设置成大于 365，如果使用默认值 100，则会报错。

```
hive.exec.max.dynamic.partitions.pernode=100
```

（5）整个 MR Job 中，最大可以创建多少个 HDFS 文件。

```
hive.exec.max.created.files=100000
```

（6）当有空分区生成时，是否抛出异常。一般不需要设置。

```
hive.error.on.empty.partition=false
```

2．案例实操

需求：将 ori 中的数据按照时间(如：20111230000008)，插入到目标表 ori_partitioned_target 的相应分区中。

（1）创建分区表

```
create table ori_partitioned(id bigint, time bigint, uid string, keyword string,
 url_rank int, click_num int, click_url string)
partitioned by (p_time bigint)
row format delimited fields terminated by '\t';
```

（2）加载数据到分区表中

```
hive (default)> load data local inpath '/home/atguigu/ds1' into table
 ori_partitioned partition(p_time='20111230000010') ;
hive (default)> load data local inpath '/home/atguigu/ds2' into table
ori_partitioned partition(p_time='20111230000011') ;
```

（3）创建目标分区表

```
create table ori_partitioned_target(id bigint, time bigint, uid string,
 keyword string, url_rank int, click_num int, click_url string) PARTITIONED
BY (p_time STRING) row format delimited fields terminated by '\t';
```

（4）设置动态分区

```
set hive.exec.dynamic.partition = true;
set hive.exec.dynamic.partition.mode = nonstrict;
set hive.exec.max.dynamic.partitions = 1000;
set hive.exec.max.dynamic.partitions.pernode = 100;
set hive.exec.max.created.files = 100000;
set hive.error.on.empty.partition = false;

hive (default)> insert overwrite table ori partitioned target partition
(p_time)
select id, time, uid, keyword, url_rank, click_num, click_url, p_time from
ori_partitioned;
```

（5）查看目标分区表的分区情况

```
hive (default)> show partitions ori_partitioned_target;
```

17.3.9 分桶

1）分桶表数据存储

分区针对的是数据的存储路径；分桶针对的是数据文件。

分区提供一个隔离数据和优化查询的便利方式。不过，并非所有的数据集都可形成合理的分区，特别是之前所提到过的要确定合适的划分大小这个疑虑。

分桶是将数据集分解成更容易管理的若干部分的另一个技术。

1．先创建分桶表，通过直接导入数据文件的方式

（1）数据准备

student.txt

（2）创建分桶表

```
create table stu_buck(id int, name string)
clustered by(id)
into 4 buckets
row format delimited fields terminated by '\t';
```

（3）查看表结构

```
hive (default)> desc formatted stu_buck;
Num Buckets:            4
```

（4）导入数据到分桶表中

```
hive (default)> load data local inpath '/opt/module/datas/student.txt' into table
 stu_buck;
```

（5）查看创建的分桶表中是否分成 4 个桶，如图 6-7 所示

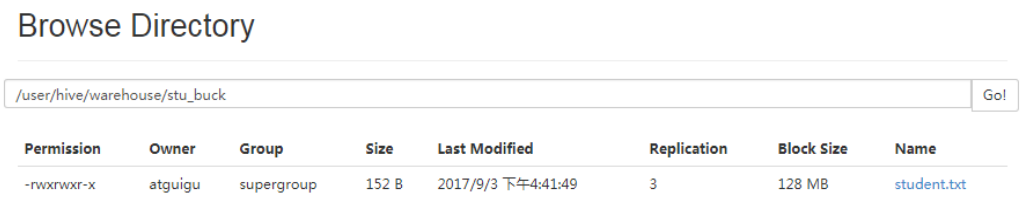


图 6-7 未分桶

发现并没有分成 4 个桶。是什么原因呢？

2．创建分桶表时，数据通过子查询的方式导入

（1）先建一个普通的 stu 表

```
create table stu(id int, name string)
row format delimited fields terminated by '\t';
```

（2）向普通的 stu 表中导入数据

```
load data local inpath '/opt/module/datas/student.txt' into table stu;
```

（3）清空 stu_buck 表中数据

```
truncate table stu_buck;
select * from stu_buck;
```

（4）导入数据到分桶表，通过子查询的方式

```
insert into table stu_buck
select id, name from stu;
```

（5）发现还是只有一个分桶，如图 6-8 所示

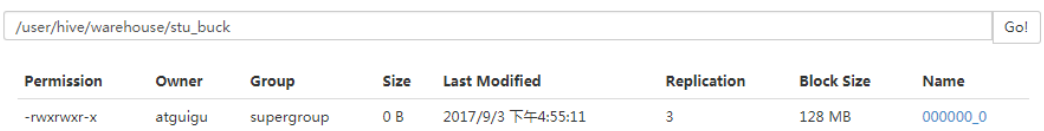


图 6-8 未分桶

（6）需要设置一个属性

```
hive (default)> set hive.enforce.bucketing=true;
hive (default)> set mapreduce.job.reduces=-1;
hive (default)> insert into table stu_buck
select id, name from stu;
```

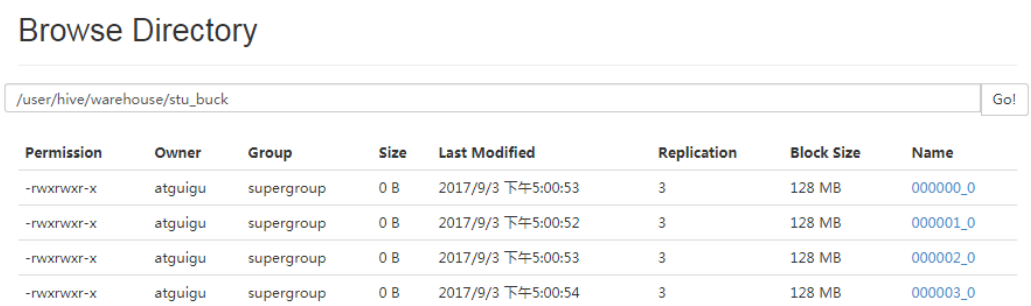


图 6-9 分桶

（7）查询分桶的数据

```
hive (default)> select * from stu_buck;
OK
stu_buck.id     stu_buck.name
1004    ss4
1008    ss8
1012    ss12
1016    ss16
1001    ss1
1005    ss5
1009    ss9
1013    ss13
1002    ss2
1006    ss6
1010    ss10
1014    ss14
1003    ss3
1007    ss7
1011    ss11
1015    ss15
```

2）分桶抽样查询

对于非常大的数据集，有时用户需要使用的是一个具有代表性的查询结果而不是全部结果。Hive 可以通过对表进行抽样来满足这个需求。

查询表 stu_buck 中的数据。

```
hive (default)> select * from stu_buck tablesample(bucket 1 out of 4 on id);
```

注：tablesample 是抽样语句，语法：TABLESAMPLE(BUCKET x OUT OF y) 。

y 必须是 table 总 bucket 数的倍数或者因子。hive 根据 y 的大小，决定抽样的比例。例如，table 总共分了 4 份，当 y=2 时，抽取(4/2=)2 个 bucket 的数据，当 y=8 时，抽取(4/8=)1/2 个 bucket 的数据。

x 表示从哪个 bucket 开始抽取，如果需要取多个分区，以后的分区号为当前分区号加上 y。例如，table 总 bucket 数为 4，tablesample(bucket 1 out of 2)，表示总共抽取（4/2=）2 个 bucket 的数据，抽取第 1(x)个和第 4(x+y)个 bucket 的数据。

注意：x 的值必须小于等于 y 的值，否则

FAILED: SemanticException [Error 10061]: Numerator should not be bigger than denominator in sample clause for table stu_buck

17.3.10 分区

分区表实际上就是对应一个 HDFS 文件系统上的独立的文件夹，该文件夹下是该分区所有的数据文件。Hive 中的分区就是分目录，把一个大的数据集根据业务需要分割成小的数据集。在查询时通过 WHERE 子句中的表达式选择查询所需要的指定的分区，这样的查

询效率会提高很多。

1）分区表基本操作

1．引入分区表（需要根据日期对日志进行管理）

```
/user/hive/warehouse/log_partition/20170702/20170702.log
/user/hive/warehouse/log_partition/20170703/20170703.log
/user/hive/warehouse/log_partition/20170704/20170704.log
```

2．创建分区表语法

```
hive (default)> create table dept partition(
            deptno int, dname string, loc string
            )
            partitioned by (month string)
            row format delimited fields terminated by '\t';
```

3．加载数据到分区表中

```
hive (default)> load data local inpath '/opt/module/datas/dept.txt' into table
default.dept_partition partition(month='201709');
hive (default)> load data local inpath '/opt/module/datas/dept.txt' into table
default.dept partition partition(month='201708');
hive (default)> load data local inpath '/opt/module/datas/dept.txt' into table
default.dept_partition partition(month='201707');
```

图 6-5 加载数据到分区表

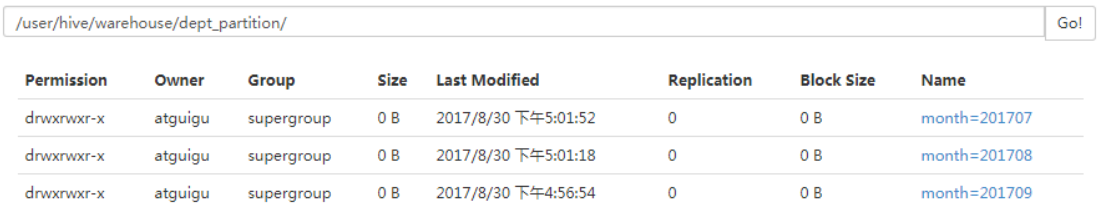


图 6-6 分区表

4．查询分区表中数据

单分区查询

```
hive (default)> select * from dept_partition where month='201709';
```

多分区联合查询

```
hive (default)> select * from dept_partition where month='201709'
            union
            select * from dept partition where month='201708'
            union
            select * from dept_partition where month='201707';

_u3.deptno      _u3.dname       _u3.loc _u3.month
10      ACCOUNTING      NEW YORK        201707
10      ACCOUNTING      NEW YORK        201708
10      ACCOUNTING      NEW YORK        201709
20      RESEARCH        DALLAS  201707
20      RESEARCH        DALLAS  201708
20      RESEARCH        DALLAS  201709
30      SALES   CHICAGO 201707
30      SALES   CHICAGO 201708
30      SALES   CHICAGO 201709
40      OPERATIONS      BOSTON  201707
40      OPERATIONS      BOSTON  201708
40      OPERATIONS      BOSTON  201709
```

5．增加分区

创建单个分区

```
hive (default)> alter table dept_partition add partition(month='201706') ;
```

同时创建多个分区

```
hive (default)> alter table dept partition add partition(month='201705')
partition(month='201704');
```

6．删除分区

删除单个分区

```
hive (default)> alter table dept_partition drop partition (month='201704');
```

同时删除多个分区

```
hive (default)> alter table dept partition drop partition (month='201705'),
partition (month='201706');
```

7．查看分区表有多少分区

```
hive> show partitions dept_partition;
```

8．查看分区表结构

```
  hive> desc formatted dept partition;

# Partition Information
# col_name            data_type            comment
month                string
```

2）分区表注意事项

1．创建二级分区表

```
hive (default)> create table dept_partition2(
            deptno int, dname string, loc string
            )
            partitioned by (month string, day string)
            row format delimited fields terminated by '\t';
```

2．正常的加载数据

（1）加载数据到二级分区表中

```
hive (default)> load data local inpath '/opt/module/datas/dept.txt' into table
 default.dept_partition2 partition(month='201709', day='13');
```

（2）查询分区数据

```
hive (default)> select * from dept_partition2 where month='201709' and day='13';
```

3．把数据直接上传到分区目录上，让分区表和数据产生关联的三种方式

（1）方式一：上传数据后修复

上传数据

```
hive (default)> dfs -mkdir -p
 /user/hive/warehouse/dept partition2/month=201709/day=12;
hive        (default)>        dfs        -put        /opt/module/datas/dept.txt
/user/hive/warehouse/dept_partition2/month=201709/day=12;
```

查询数据（查询不到刚上传的数据）

```
hive (default)> select * from dept_partition2 where month='201709' and day='12';
```

执行修复命令

```
  hive> msck repair table dept_partition2;
```

再次查询数据

```
hive (default)> select * from dept_partition2 where month='201709' and day='12';
```

（2）方式二：上传数据后添加分区

上传数据

```
hive (default)> dfs -mkdir -p
 /user/hive/warehouse/dept_partition2/month=201709/day=11;
hive        (default)>        dfs        -put        /opt/module/datas/dept.txt
/user/hive/warehouse/dept_partition2/month=201709/day=11;
```

执行添加分区

```
  hive (default)> alter table dept_partition2 add partition(month='201709',
 day='11');
```

查询数据

```
hive (default)> select * from dept_partition2 where month='201709' and day='11';
```

（3）方式三：上传数据后 load 数据到分区

创建目录

```
hive (default)> dfs -mkdir -p
 /user/hive/warehouse/dept_partition2/month=201709/day=10;
```

上传数据

```
hive (default)> load data local inpath '/opt/module/datas/dept.txt' into table
 dept_partition2 partition(month='201709',day='10');
```

查询数据

```
hive (default)> select * from dept_partition2 where month='201709' and day='10';
```

17.4 数据倾斜

17.4.1 合理设置 Map 数

1）通常情况下，作业会通过 input 的目录产生一个或者多个 map 任务。

主要的决定因素有：input 的文件总个数，input 的文件大小，集群设置的文件块大小。

2）是不是 map 数越多越好？

答案是否定的。如果一个任务有很多小文件（远远小于块大小 128m），则每个小文件也会被当做一个块，用一个 map 任务来完成，而一个 map 任务启动和初始化的时间远远大于逻辑处理的时间，就会造成很大的资源浪费。而且，同时可执行的 map 数是受限的。

3）是不是保证每个 map 处理接近 128m 的文件块，就高枕无忧了？

答案也是不一定。比如有一个 127m 的文件，正常会用一个 map 去完成，但这个文件只有一个或者两个小字段，却有几千万的记录，如果 map 处理的逻辑比较复杂，用一个 map 任务去做，肯定也比较耗时。

针对上面的问题 2 和 3，我们需要采取两种方式来解决：即减少 map 数和增加 map 数；

17.4.2 小文件进行合并

在 map 执行前合并小文件，减少 map 数：CombineHiveInputFormat 具有对小文件进行合并的功能（系统默认的格式）。HiveInputFormat 没有对小文件合并功能。

```
set hive.input.format= org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
```

17.4.3 复杂文件增加 Map 数

当 input 的文件都很大，任务逻辑复杂，map 执行非常慢的时候，可以考虑增加 Map 数，来使得每个 map 处理的数据量减少，从而提高任务的执行效率。

增加 map 的方法为：根据 computeSliteSize(Math.max(minSize,Math.min(maxSize,blocksize)))=blocksize=128M 公式，调整 maxSize 最大值。让 maxSize 最大值低于 blocksize 就可以增加 map 的个数。

案例实操：

1．执行查询

```
hive (default)> select count(*) from emp;
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
```

2．设置最大切片值为 100 个字节

```
hive (default)> set mapreduce.input.fileinputformat.split.maxsize=100;
hive (default)> select count(*) from emp;
Hadoop job information for Stage-1: number of mappers: 6; number of reducers: 1
```

17.4.4 合理设置 Reduce 数

1．调整 reduce 个数方法一

（1）每个 Reduce 处理的数据量默认是 256MB

```
hive.exec.reducers.bytes.per.reducer=256000000
```

（2）每个任务最大的 reduce 数，默认为 1009

```
hive.exec.reducers.max=1009
```

（3）计算 reducer 数的公式

```
N=min(参数 2，总输入数据量/参数 1)
```

2．调整 reduce 个数方法二

在 hadoop 的 mapred-default.xml 文件中修改

设置每个 job 的 Reduce 个数

```
set mapreduce.job.reduces = 15;
```

3．reduce 个数并不是越多越好

1）过多的启动和初始化 reduce 也会消耗时间和资源；

2）另外，有多少个 reduce，就会有多少个输出文件，如果生成了很多个小文件，那么如果这些小文件作为下一个任务的输入，则也会出现小文件过多的问题；

在设置 reduce 个数的时候也需要考虑这两个原则：处理大数据量利用合适的 reduce 数；使单个 reduce 任务处理数据量大小要合适；

17.5 并行执行

Hive 会将一个查询转化成一个或者多个阶段。这样的阶段可以是 MapReduce 阶段、抽样阶段、合并阶段、limit 阶段。或者 Hive 执行过程中可能需要的其他阶段。默认情况下，Hive 一次只会执行一个阶段。不过，某个特定的 job 可能包含众多的阶段，而这些阶段可能并非完全互相依赖的，也就是说有些阶段是可以并行执行的，这样可能使得整个 job 的执行时间缩短。不过，如果有更多的阶段可以并行执行，那么 job 可能就越快完成。

通过设置参数 hive.exec.parallel 值为 true，就可以开启并发执行。不过，在共享集群中，需要注意下，如果 job 中并行阶段增多，那么集群利用率就会增加。

```
set hive.exec.parallel=true;              //打开任务并行执行
set hive.exec.parallel.thread.number=16;  //同一个 sql 允许最大并行度，默认为 8。
```

当然，得是在系统资源比较空闲的时候才有优势，否则，没资源，并行也起不来。

17.6 严格模式

Hive 提供了一个严格模式，可以防止用户执行那些可能意向不到的不好的影响的查询。

通过设置属性 hive.mapred.mode 值为默认是非严格模式 nonstrict 。开启严格模式需要修改 hive.mapred.mode 值为 strict，开启严格模式可以禁止 3 种类型的查询。

```
<property>
   <name>hive.mapred.mode</name>
   <value>strict</value>
   <description>
     The mode in which the Hive operations are being performed.
     In strict mode, some risky queries are not allowed to run. They include:
       Cartesian Product.
       No partition being picked up for a query.
       Comparing bigints and strings.
       Comparing bigints and doubles.
       Orderby without limit.
   </description>
   </property>
```

1) 对于分区表，除非 where 语句中含有分区字段过滤条件来限制范围，否则不允许执行。换句话说，就是用户不允许扫描所有分区。进行这个限制的原因是，通常分区表都拥有非常大的数据集，而且数据增加迅速。没有进行分区限制的查询可能会消耗令人不可接受的巨大资源来处理这个表。

2) 对于使用了 order by 语句的查询，要求必须使用 limit 语句。因为 order by 为了执行排序过程会将所有的结果数据分发到同一个 Reducer 中进行处理，强制要求用户增加这个 LIMIT 语句可以防止 Reducer 额外执行很长一段时间。

3) 限制笛卡尔积的查询。对关系型数据库非常了解的用户可能期望在执行 JOIN 查询的时候不使用 ON 语句而是使用 where 语句，这样关系数据库的执行优化器就可以高效地将 WHERE 语句转化成那个 ON 语句。不幸的是，Hive 并不会执行这种优化，因此，如果表足够大，那么这个查询就会出现不可控的情况。

17.7 JVM 重用

JVM 重用是 Hadoop 调优参数的内容，其对 Hive 的性能具有非常大的影响，特别是对于很难避免小文件的场景或 task 特别多的场景，这类场景大多数执行时间都很短。

Hadoop 的默认配置通常是使用派生 JVM 来执行 map 和 Reduce 任务的。这时 JVM 的启动过程可能会造成相当大的开销，尤其是执行的 job 包含有成百上千 task 任务的情况。JVM 重用可以使得 JVM 实例在同一个 job 中重新使用 N 次。N 的值可以在 Hadoop 的 mapred-site.xml 文件中进行配置。通常在 10-20 之间，具体多少需要根据具体业务场景测试得出。

```
<property>
  <name>mapreduce.job.jvm.numtasks</name>
  <value>10</value>
  <description>How many tasks to run per jvm. If set to -1, there is
  no limit.
  </description>
</property>
```

这个功能的缺点是，开启 JVM 重用将一直占用使用到的 task 插槽，以便进行重用，直到任务完成后才能释放。如果某个“不平衡的”job 中有某几个 reduce task 执行的时间要比其他 Reduce task 消耗的时间多的多的话，那么保留的插槽就会一直空闲着却无法被其他的 job 使用，直到所有的 task 都结束了才会释放。

17.8 推测执行

在分布式集群环境下，因为程序 Bug（包括 Hadoop 本身的 bug），负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行速度不一致，有些任务的运行速度可能明显慢于其他任务（比如一个作业的某个任务进度只有 50%，而其他所有任务已经运行完毕），则这些任务会拖慢作业的整体执行进度。为了避免这种情况发生，Hadoop 采用了推测执行（Speculative Execution）机制，它根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

设置开启推测执行参数：Hadoop 的 mapred-site.xml 文件中进行配置

```
<property>
  <name>mapreduce.map.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some map tasks
               may be executed in parallel.</description>
</property>

<property>
  <name>mapreduce.reduce.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some reduce tasks
```

```
            may be executed in parallel.</description>
</property>
```

不过 hive 本身也提供了配置项来控制 reduce-side 的推测执行：

```
  <property>
    <name>hive.mapred.reduce.tasks.speculative.execution</name>
    <value>true</value>
    <description>Whether speculative execution for reducers should be turned on.
</description>
  </property>
```

关于调优这些推测执行变量，还很难给一个具体的建议。如果用户对于运行时的偏差非常敏感的话，那么可以将这些功能关闭掉。如果用户因为输入数据量很大而需要执行长时间的 map 或者 Reduce task 的话，那么启动推测执行造成的浪费是非常巨大大。

17.9 压缩

17.9.1 Hadoop 源码编译支持 Snappy 压缩

1）资源准备

1．CentOS 联网

配置 CentOS 能连接外网。Linux 虚拟机 ping www.baidu.com 是畅通的

注意：采用 root 角色编译，减少文件夹权限出现问题

2．jar 包准备(hadoop 源码、JDK8 、maven、protobuf)

（1）hadoop-2.7.2-src.tar.gz

（2）jdk-8u144-linux-x64.tar.gz

（3）snappy-1.1.3.tar.gz

（4）apache-maven-3.0.5-bin.tar.gz

（5）protobuf-2.5.0.tar.gz

17.9.2 jar 包安装

注意：所有操作必须在 root 用户下完成

1．JDK 解压、配置环境变量 JAVA_HOME 和 PATH，验证 java-version(如下都需要验证是否配置成功)

```
[root@hadoop101 software] # tar -zxf jdk-8u144-linux-x64.tar.gz -C /opt/module/
[root@hadoop101 software]# vi /etc/profile
#JAVA_HOME
export JAVA_HOME=/opt/module/jdk1.8.0_144
export PATH=$PATH:$JAVA_HOME/bin
[root@hadoop101 software]#source /etc/profile
```

验证命令：java -version

2．Maven 解压、配置 MAVEN_HOME 和 PATH

```
[root@hadoop101 software]# tar -zxvf apache-maven-3.0.5-bin.tar.gz -C /opt/module/
[root@hadoop101 apache-maven-3.0.5]# vi /etc/profile
#MAVEN HOME
export MAVEN_HOME=/opt/module/apache-maven-3.0.5
export PATH=$PATH:$MAVEN_HOME/bin
[root@hadoop101 software]#source /etc/profile
```

验证命令：mvn -version

17.9.3 编译源码

1．准备编译环境

```
[root@hadoop101 software]# yum install svn
[root@hadoop101 software]# yum install autoconf automake libtool cmake
[root@hadoop101 software]# yum install ncurses-devel
[root@hadoop101 software]# yum install openssl-devel
[root@hadoop101 software]# yum install gcc*
```

2．编译安装 snappy

```
[root@hadoop101 software]# tar -zxvf snappy-1.1.3.tar.gz -C /opt/module/
[root@hadoop101 module]# cd snappy-1.1.3/
[root@hadoop101 snappy-1.1.3]# ./configure
[root@hadoop101 snappy-1.1.3]# make
[root@hadoop101 snappy-1.1.3]# make install
# 查看 snappy 库文件
[root@hadoop101 snappy-1.1.3]# ls -lh /usr/local/lib |grep snappy
```

3．编译安装 protobuf

```
[root@hadoop101 software]# tar -zxvf protobuf-2.5.0.tar.gz -C /opt/module/
[root@hadoop101 module]# cd protobuf-2.5.0/
[root@hadoop101 protobuf-2.5.0]# ./configure
[root@hadoop101 protobuf-2.5.0]#  make
[root@hadoop101 protobuf-2.5.0]#  make install
# 查看 protobuf 版本以测试是否安装成功
[root@hadoop101 protobuf-2.5.0]# protoc --version
```

4．编译 hadoop native

```
[root@hadoop101 software]# tar -zxvf hadoop-2.7.2-src.tar.gz
[root@hadoop101 software]# cd hadoop-2.7.2-src/
[root@hadoop101  software]#  mvn  clean  package  -DskipTests  -Pdist,native  -Dtar
-Dsnappy.lib=/usr/local/lib -Dbundle.snappy
```

执行成功后，/opt/software/hadoop-2.7.2-src/hadoop-dist/target/hadoop-2.7.2.tar.gz 即为新生成的支持 snappy 压缩的二进制安装包。

17.9.4 Hadoop 压缩配置

1）MR 支持的压缩编码

表 6-8

压缩格式	工具	算法	文件扩展名	是否可切分
DEFAULT	无	DEFAULT	.deflate	否
Gzip	gzip	DEFAULT	.gz	否
bzip2	bzip2	bzip2	.bz2	是
LZO	lzop	LZO	.lzo	是
Snappy	无	Snappy	.snappy	否

为了支持多种压缩/解压缩算法，Hadoop 引入了编码/解码器，如下表所示：

表 6-9

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较：

表 6-10

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

http://google.github.io/snappy/

On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

2） 压缩参数配置

要在 Hadoop 中启用压缩，可以配置如下参数（mapred-site.xml 文件中）：

表 6-11

参数	默认值	阶段	建议
io.compression.codecs（在 core-site.xml 中配置）	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec, org.apache.hadoop.io.compress.Lz4Codec	输入压缩	Hadoop 使用文件扩展名判断是否支持某种编解码器
mapreduce.map.output.compress	false	mapper 输出	这个参数设为 true 启用压缩
mapreduce.map.output.compress.codec	org.apache.hadoop.io.compress.DefaultCodec	mapper 输出	使用 LZO、LZ4 或 snappy 编解码器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress	false	reducer 输出	这个参数设为 true 启用压缩
mapreduce.output.fileoutputformat.compress.codec	org.apache.hadoop.io.compress. DefaultCodec	reducer 输出	使用标准工具或者编解码器，如 gzip 和 bzip2
mapreduce.output.fileoutputformat.compress.type	RECORD	reducer 输出	SequenceFile 输出使用的压缩类型：NONE 和 BLOCK

17.9.5 开启 Map 输出阶段压缩

开启 map 输出阶段压缩可以减少 job 中 map 和 Reduce task 间数据传输量。具体配置如下：

案例实操：

1．开启 hive 中间传输数据压缩功能

```
hive (default)>set hive.exec.compress.intermediate=true;
```

2．开启 mapreduce 中 map 输出压缩功能

```
hive (default)>set mapreduce.map.output.compress=true;
```

3．设置 mapreduce 中 map 输出数据的压缩方式

```
hive (default)>set mapreduce.map.output.compress.codec=
 org.apache.hadoop.io.compress.SnappyCodec;
```

4．执行查询语句

```
    hive (default)> select count(ename) name from emp;
```

17.9.6 开启 Reduce 输出阶段压缩

当 Hive 将输出写入到表中时，输出内容同样可以进行压缩。属性 hive.exec.compress.output 控制着这个功能。用户可能需要保持默认设置文件中的默认值 false，这样默认的输出就是非压缩的纯文本文件了。用户可以通过在查询语句或执行脚本中设置这个值为 true，来开启输出结果压缩功能。

案例实操：

1．开启 hive 最终输出数据压缩功能

```
hive (default)>set hive.exec.compress.output=true;
```

2．开启 mapreduce 最终输出数据压缩

```
hive (default)>set mapreduce.output.fileoutputformat.compress=true;
```

3．设置 mapreduce 最终数据输出压缩方式

```
hive (default)> set mapreduce.output.fileoutputformat.compress.codec =
 org.apache.hadoop.io.compress.SnappyCodec;
```

4．设置 mapreduce 最终数据输出压缩为块压缩

```
hive (default)> set mapreduce.output.fileoutputformat.compress.type=BLOCK;
```

5．测试一下输出结果是否是压缩文件

```
hive (default)> insert overwrite local directory
 '/opt/module/datas/distribute-result' select * from emp distribute by deptno sort
by empno desc;
```

17.10 执行计划（Explain）

1．基本语法

EXPLAIN [EXTENDED | DEPENDENCY | AUTHORIZATION] query

2．案例实操

（1）查看下面这条语句的执行计划

```
hive (default)> explain select * from emp;
hive (default)> explain select deptno, avg(sal) avg_sal from emp group by deptno;
```

（2）查看详细执行计划

```
hive (default)> explain extended select * from emp;
hive (default)> explain extended select deptno, avg(sal) avg_sal from emp group by
deptno;
```

18 Hive 数据分析面试题

场景举例.北京市学生成绩分析.

成绩的数据格式:时间,学校,年纪,姓名,科目,成绩

样例数据如下:

2013,北大,1,裘容絮,语文,97

2013,北大,1,庆眠拔,语文,52

2013,北大,1,乌洒筹,语文,85

2012,清华,0,钦尧,英语,61

2015,北理工,3,冼殿,物理,81

2016,北科,4,况飘索,化学,92

2014,北航,2,孔须,数学,70

2012,清华,0,王脊,英语,59

2014,北航,2,方部盾,数学,49

2014,北航,2,东门雹,数学,77

问题:

18.1 情景题：分组 TOPN

```
# 1.分组 TOPN 选出  今年每个学校,每个年级,分数前三的科目.
hive -e "

set mapreduce.job.queuename=low;
select t.*
from
(
select
        school,
        class,
        subjects,
        score,
        row_number() over (partition by school,class,subjects order by score desc) rank_code
from spark_test_wx
where partition_id = "2017"
) t
where t.rank_code <= 3;
```

北大	0	英语	95	1
北大	0	英语	77	2
北大	0	英语	50	3
北大	2	数学	80	1
北大	2	数学	62	2
北大	2	数学	56	3
北大	3	物理	82	1
北大	3	物理	61	2
北大	3	物理	57	3
北大	4	化学	72	1
北大	4	化学	56	2
北大	4	化学	52	3
北大	5	生物	84	1
北大	5	生物	62	2
北大	5	生物	57	3
北理工	0	英语	93	1
北理工	0	英语	71	2
北理工	0	英语	59	3
北理工	1	语文	79	1
北理工	1	语文	62	2
北理工	1	语文	49	3
北理工	2	数学	93	1
北理工	2	数学	85	2
北理工	2	数学	73	3
北理工	3	物理	78	1
北理工	3	物理	49	2
北理工	4	化学	97	1
北理工	4	化学	89	2
北理工	4	化学	78	3
北理工	5	生物	86	1
北理工	5	生物	77	2
北理工	5	生物	57	3

18.2 情景题：where 与 having

```
-- 今年 清华 1 年级 总成绩大于 200 分的学生 以及学生数
SET mapreduce.job.queuename = low;
SELECT
    school,
    class,
    NAME,
    sum(score) AS total_score,
```

```
        count(1) over (PARTITION BY school, class) nct
    FROM
        spark_test_wx
    WHERE
        partition_id = "2017"
    AND school = "清华"
    AND class = 1
    GROUP BY
        school,
        class,
        NAME
    HAVING
        total_score > 200;
```

having 是分组（group by）后的筛选条件，分组后的数据组内再筛选，也就是说 HAVING 子句可以让我们筛选成组后的各组数据。

where则是在分组,聚合前先筛选记录。也就是说作用在GROUP BY子句和HAVING子句前。

18.3 情景题：数据倾斜

今年加入进来了 10 个学校,学校数据差异很大计算每个学校的平均分。简述需要注意的点。

该题主要是考察数据倾斜的处理方式。

Group by 方式很容易产生数据倾斜。需要注意一下几点

1）Map 端部分聚合

hive.map.aggr=true（用于设定是否在 map 端进行聚合，默认值为真，相当于 combine）

hive.groupby.mapaggr.checkinterval=100000（用于设定 map 端进行聚合操作的条数）

2）有数据倾斜时进行负载均衡

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。

18.4 情景题：分区表

假设我创建了一张表，其中包含了 2016 年客户完成的所有交易的详细信息：CREATE TABLE transaction_details (cust_id INT, amount FLOAT, month STRING, country STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘,’ ;

现在我插入了 100 万条数据，我想知道每个月的总收入。

问：如何高效的统计出结果。写出步骤即可。

19 Hive 中，建的表为压缩表，但是输入文件为非压缩格式，会产生怎样的现象或者结果？

使用 load 加载数据的时候，不会是压缩文件。使用查询插入数据的时候，存储的数据是压缩的。

20 已知表 a 是一张内部表，如何将它转换成外部表？请写出相应的 hive 语句。

alter table a set tblproperties('EXTERNAL'='TRUE');

21 Hive 中 mapjoin 的原理和实际应用？

原理就是在 Map 阶段将小表读入内存，顺序扫描大表完成 Join。

通过 MapReduce Local Task，将小表读入内存，生成 HashTableFiles 上传至 Distributed Cache 中，这里会对 HashTableFiles 进行压缩。

MapReduce Job 在 Map 阶段，每个 Mapper 从 Distributed Cache 读取 HashTableFiles 到内存中，顺序扫描大表，在 Map 阶段直接进行 Join，将数据传递给下一个 MapReduce 任务。

作用：优化 hive 语句，避免出现在 Reducer 端的数据倾斜。

22 订单详情表 ord_det(order_id 订单号，sku_id 商品编号，sale_qtty 销售数量，dt 日期分区)任务计算 2016 年 1 月 1 日商品销量的 Top100，并按销量降级排序

SELECT

```
    sku_id,
    sale_qtty
FROM
    ord_det PARTITION (dt = '20160101')
ORDER BY
    sale_qtty DESC
LIMIT 100;
```

23 某日志的格式如下：

pin|-|request_tm|-url|-|sku_id|-|amount

分隔符为“|-|”，

数据样例为:

张三|-|q2013-11-23 11:59:30|-|www.jd.com|-|100023|-|110.15

假设本地数据文件为 sample.txt,先将其导入到 hive 的 test 库的表 t_sample 中，并计算每个用户的总消费金额，写出详细过程包括表结构。

```
CREATE external TABLE t_sample (
    pin string,
    request_tm string,
    url string,
    sku_id string,
    amount string
) ROW format delimited FIELDS TERMINATED BY '|-|';'\|-\|','\\|-\\|','\|\-\|','\\|\\-\\|'... ...
'/|-/|','//|-//|','/|/-/|','//|//-//|'... ...
```

先清洗数据，改变其分割符变为“\t”，存入本地文件 jd.txt 中(注意 split()方法也是需要转义的)

```
CREATE external TABLE t_sample (
    pin string,
    request_tm string,
    url string,
    sku_id INT,
    amount DOUBLE
) ROW format delimited FIELDS TERMINATED BY "\t";

LOAD DATA LOCAL inpath '/opt/module/datas/jd.txt' INTO TABLE t_sample;

SELECT
    *
FROM
    t_sample;

SELECT
    pin,
```

```
    sum(amount)
FROM
    t_sample
GROUP BY
    pin;
```

24 有一张很大的表：TRLOG，该表大概有 2T 左右

```
CREATE TABLE TRLOG
(   PLATFORM string,
    USER_ID int,
    CLICK_TIME string,
    CLICK_URL string)
    row format delimited fields terminated by '\t';
```

数据:

PLATFORM	USER_ID	CLICK_TIME	CLICK_URL
WEB	12332321	2013-03-21 13:48:31.324	/home/
WEB	12332321	2013-03-21 13:48:32.954	/selectcat/er/
WEB	12332321	2013-03-21 13:48:46.365	/er/viewad/12.html
WEB	12332321	2013-03-21 13:48:53.651	/er/viewad/13.html
……	……	……	……

把上述数据处理为如下结构的表 ALLOG:

```
CREATE TABLE ALLOG
    (PLATFORM string,
    USER_ID int,
    SEQ int,
    FROM_URL string,
    TO_URL string)
    row format delimited fields terminated by '\t';
```

整理后的数据结构：

PLATFORM	USER_ID	SEQ	FROM_URL	TO_URL
WEB	12332321	1	NULL	/home/
WEB	12332321	2	/home/	/selectcat/er/
WEB	12332321	3	/selectcat/er/	/er/viewad/12.html
WEB	12332321	4	/er/viewad/12.html	/er/viewad/13.html
WEB	12332321	1	NULL	/m/home/
WEB	12332321	2	/m/home/	/m/selectcat/fang/

PLATFORM 和 USER_ID 还是代表平台和用户 ID:SEQ 字段代表用户按时间排序后的访问顺序，FROM_URL 和 TO_URL 分别代表用户从哪一页跳转到哪一页。某个用户的第一条访问记录的 FROM_URL 是 NULL（空值）。实现基于纯 Hive SQL 的 ETL 过程，从 TRLOG 表生成 ALLOG 表:（结果是一套 SQL）

```
INSERT INTO TABLE allog SELECT
    platform,
    user_id,
    row_number () over (
        PARTITION BY user_id
        ORDER BY
            click_time
    ) seq,
    lag (click_url, 1) over (
        PARTITION BY user_id
        ORDER BY
            click_time
    ) AS from_url,
    click_url AS to_url
FROM
    trlog;
```

25 已知一个表 STG.ORDER，有如下字段:Date，Order_id，User_id， amount。请给出 sql 进行统计：数据样例:2017-01-01,10029028,1000003251,33.57。

1) 给出 2017 年每个月的订单数、用户数、总成交金额。

```
SELECT
    count(Order_id) order_count,
    count(DISTINCT(User_id)) user_count,
    sum(amount) amount_sum,
    substring(Date, 1, 7) MONTH
FROM
    STG. ORDER
WHERE
    substring(Date, 1, 4) = ’2017’
GROUP BY
    MONTH;
```

2) 给出 2017 年 11 月的新客数(指在 11 月才有第一笔订单)。

```
SELECT
    count(1)
FROM
    (
        SELECT
            Order_id,
            DATE,
```

```
            LAG (DATE, 1) firstOrder
        FROM
            STG. ORDER
    ) t1
WHERE
    firstOrder IS NULL
AND SUBSTRING(DATE, 1, 7) = '2017-11';
```

三 Flume

1 flume 有哪些组件，flume 的 source、channel、sink 具体是做什么的

source 组件是专门用来收集数据的，可以处理各种类型、各种格式的日志数据,包括 avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy

source 组件把数据收集来以后，临时存放在 channel 中，即 channel 组件在 agent 中是专门用来存放临时数据的——对采集到的数据进行简单的缓存，可以存放在 memory、jdbc、file 等等。

sink 组件是用于把数据发送到目的地的组件，目的地包括 hdfs、logger、avro、thrift、ipc、file、null、Hbase、solr、自定义。

2 你是如何实现 flume 数据传输的监控的

使用第三方框架 Ganglia 实时监控 flume。

3 flume 的 source,sink,channel 的作用？你们 source 是什么类型？

source 组件是专门用来收集数据的，可以处理各种类型、各种格式的日志数据,包括 avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy

source 组件把数据收集来以后，临时存放在 channel 中，即 channel 组件在 agent 中是专门用来存放临时数据的——对采集到的数据进行简单的缓存，可以存放在 memory、jdbc、file 等等。

sink 组件是用于把数据发送到目的地的组件，目的地包括 hdfs、logger、avro、thrift、ipc、file、null、Hbase、solr、自定义。

监控后台日志：exec

监控后台产生日志的端口：netcat

四 Kafka

1 kafka 的 balance 是怎么做的

kafka 集群中的任何一个 broker,都可以向 producer 提供 metadata 信息,这些 metadata 中包含"集群中存活的 servers 列表"/"partitions leader 列表"等信息(请参看 zookeeper 中的节点信息). 当 producer 获取到 metadata 信息之后, producer 将会和 Topic 下所有 partition leader 保持 socket 连接;消息由 producer 直接通过 socket 发送到 broker,中间不会经过任何"路由层".

异步发送，将多条消息暂且在客户端 buffer 起来,并将他们批量发送到 broker;小数据 IO 太多,会拖慢整体的网络延迟,批量延迟发送事实上提升了网络效率;不过这也有一定的隐患,比如当 producer 失效时,那些尚未发送的消息将会丢失。

2 kafka 的消费者有几种模式

自动提交 offset
手动提交 offset
手动提交 partition 的 offset

3 为什么 kafka 可以实现高吞吐？单节点 kafka 的吞吐量也比其他消息队列大，为什么？

Kafka 是分布式消息系统，需要处理海量的消息，Kafka 的设计是把所有的消息都写入速度低容量大的硬盘，以此来换取更强的存储能力，但实际上，使用硬盘并没有带来过多的性能损失

kafka 主要使用了以下几个方式实现了超高的吞吐率

1）顺序读写

kafka 的消息是不断追加到文件中的，这个特性使 kafka 可以充分利用磁盘的顺序读写性能，顺序读写不需要硬盘磁头的寻道时间，只需很少的扇区旋转时间，所以速度远快于随机读写

Kafka 官方给出了测试数据(Raid-5，7200rpm)：

顺序 I/O: 600MB/s

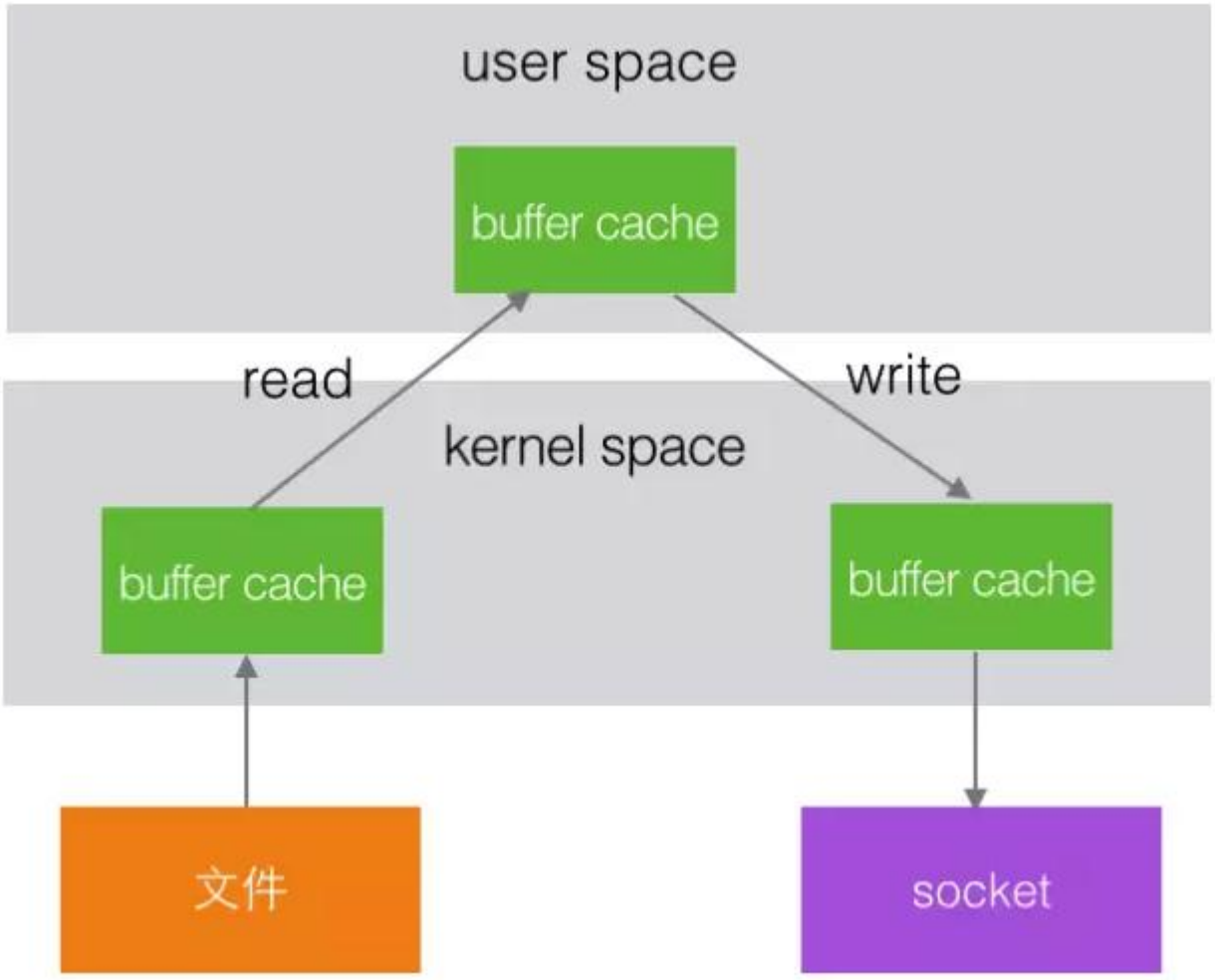
随机 I/O: 100KB/s

2）零拷贝

先简单了解下文件系统的操作流程，例如一个程序要把文件内容发送到网络

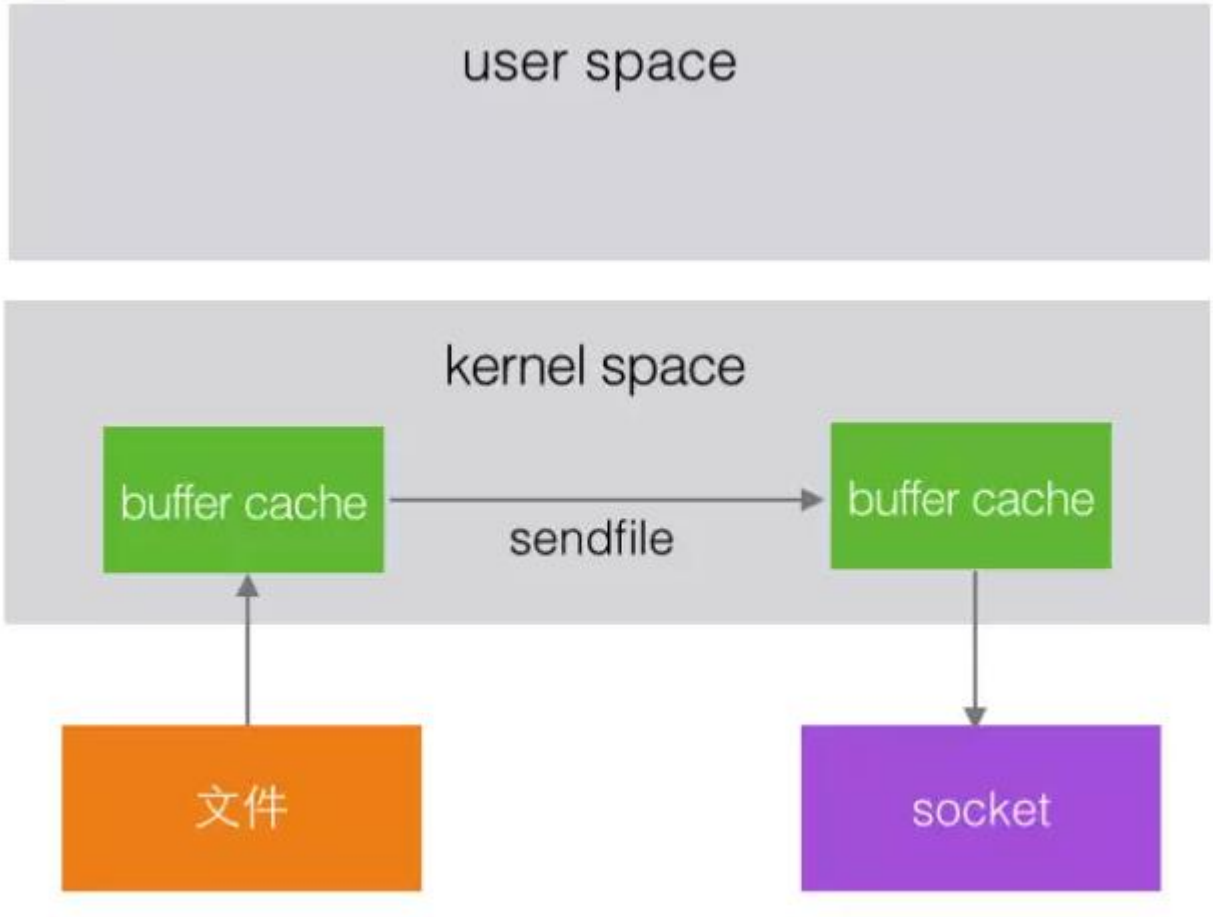
这个程序是工作在用户空间，文件和网络 socket 属于硬件资源，两者之间有一个内核空间

在操作系统内部，整个过程为：



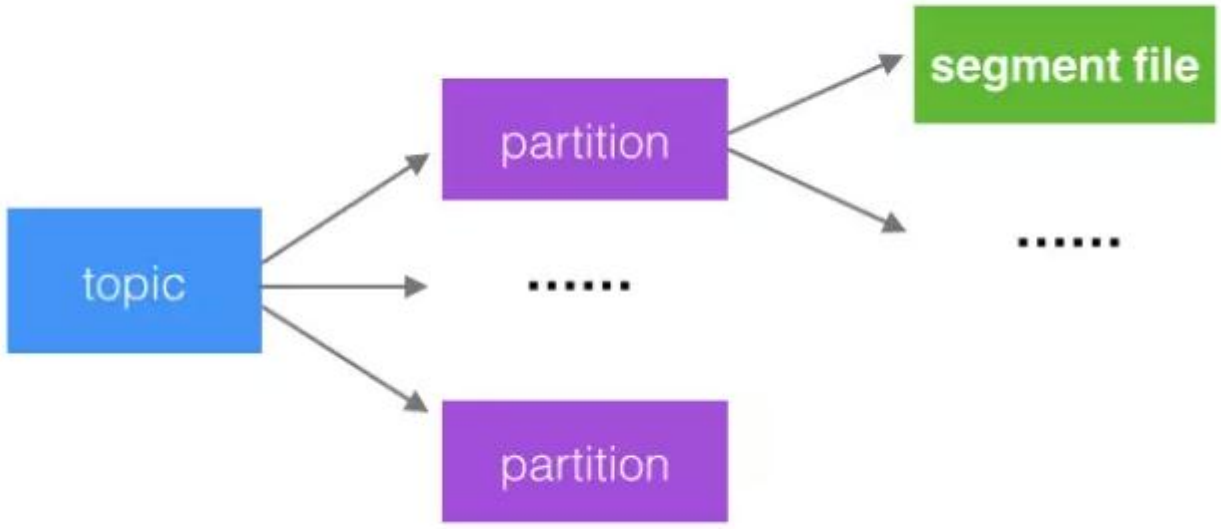
在 Linux kernel2.2 之后出现了一种叫做"零拷贝(zero-copy)"系统调用机制，就是跳过“用户缓冲区”的拷贝，建立一个磁盘空间和内存的直接映射，数据不再复制到“用户态缓冲区”

系统上下文切换减少为 2 次，可以提升一倍的性能



3）文件分段

kafka 的队列 topic 被分为了多个区 partition，每个 partition 又分为多个段 segment，所以一个队列中的消息实际上是保存在 N 多个片段文件中



通过分段的方式，每次文件操作都是对一个小文件的操作，非常轻便，同时也增加了并行处理能力

4）批量发送

Kafka 允许进行批量发送消息，先将消息缓存在内存中，然后一次请求批量发送出去，比如可以指定缓存的消息达到某个量的时候就发出去，或者缓存了固定的时间后就发送出去，如 100 条消息就发送，或者每 5 秒发送一次，这种策略将大大减少服务端的 I/O 次数

5）数据压缩

Kafka 还支持对消息集合进行压缩，Producer 可以通过 GZIP 或 Snappy 格式对消息集合进行压缩，压缩的好处就是减少传输的数据量，减轻对网络传输的压力，Producer 压缩之后，在 Consumer 需进行解压，虽然增加了 CPU 的工作，但在对大数据处理上，瓶颈在网络上而不是 CPU，所以这个成本很值得

4 kafka 的偏移量 offset 存放在哪儿，为什么？

从 kafka-0.9 版本及以后，kafka 的消费者组和 offset 信息就不存 zookeeper 了，而是存到 broker 服务器上，所以，如果你为某个消费者指定了一个消费者组名称（group.id），那么，一旦这个消费者启动，这个消费者组名和它要消费的那个 topic 的 offset 信息就会被记录在 broker 服务器上

1）概述

Kafka 版本[0.10.1.1]，已默认将消费的 offset 迁入到了 Kafka 一个名为 __consumer_offsets 的 Topic 中。其实，早在 0.8.2.2 版本，已支持存入消费的 offset 到 Topic 中，只是那时候默认是将消费的 offset 存放在 Zookeeper 集群中。那现在，官方默认将消费的 offset 存储在 Kafka 的 Topic 中，同时，也保留了存储在 Zookeeper 的接口，通过 offsets.storage 属性来进行设置。

2）内容

其实，官方这样推荐，也是有其道理的。之前版本，Kafka 其实存在一个比较大的隐患，就是利用 Zookeeper 来存储记录每个消费者/组的消费进度。虽然，在使用过程当中，JVM 帮助我们完成了一些优化，但是消费者需要频繁的去与 Zookeeper 进行交互，而利用 ZKClient 的 API 操作 Zookeeper 频繁的 Write 其本身就是一个比较低效的 Action，对于后期水平扩展也是一个比较头疼的问题。如果期间 Zookeeper 集群发生变化，那 Kafka 集群的吞吐量也跟着受影响。在此之后，官方其实很早就提出了迁移到 Kafka 的概念，只是，之前是一直默认存储在 Zookeeper 集群中，需要手动的设置，如果，对 Kafka 的使用不是很熟悉的话，一般我们就接受了默认的存储（即：存在 ZK 中）。在新版 Kafka 以及之后的版本，Kafka 消费的 offset 都会默认存放在 Kafka 集群中的一个叫 __consumer_offsets 的 topic 中。

5 Kafka 消费过的消息如何再消费

1） 修改 offset

2） 通过使用不同的 group 来消费

6 Kafka 里面用的什么方式拉的方式还是推的方式？如何保证数据不会出现丢失或者重复消费的情况？做过哪些预防措施，怎么解决以上问题的？Kafka 元数据存在哪？

pull 拉的方式

使用同步模式的时候，有 3 种状态保证消息被安全生产，在配置为 1（只保证写入 leader 成功）的话，如果刚好 leader partition 挂了，数据就会丢失。

还有一种情况可能会丢失消息，就是使用异步模式的时候，当缓冲区满了，如果配置为 0（还没有收到确认的情况下，缓冲池一满，就清空缓冲池里的消息），

数据就会被立即丢弃掉。

在数据生产时避免数据丢失的方法：

只要能避免上述两种情况，那么就可以保证消息不会被丢失。

就是说在同步模式的时候，确认机制设置为-1，也就是让消息写入 leader 和所有的副本。

还有，在异步模式下，如果消息发出去了，但还没有收到确认的时候，缓冲池满了，在配置文件中设置成不限制阻塞超时的时间，也就说让生产端一直阻塞，这样也能保证数据不会丢失。

在数据消费时，避免数据丢失的方法：如果使用了 storm，要开启 storm 的 ackfail 机制；如果没有使用 storm，确认数据被完成处理之后，再更新 offset 值。低级 API 中需要手动控制 offset 值。

数据重复消费的情况，如果处理？

（1）去重：将消息的唯一标识保存到外部介质中，每次消费处理时判断是否处理过；

（2）不管：大数据场景中，报表系统或者日志信息丢失几条都无所谓，不会影响最终的统计分析结果。

7 kafka 支不支持事物。

支持

1. 为什么要支持事务

我们在 Kafka 中设计事务的目的主要是为了满足“读取-处理-写入”这种模式的应用程序。这种模式下数据的读写是异步的，比如 Kafka 的 Topics。这种应用程序更广泛的被称之为流处理应用程序。

第一代流处理应用程序可以容忍不准确的数据处理，比如，查看网页点击数量的应用程序，能够允许计数器存在一些错误（多算或者漏算）。

然而，随着这些应用的普及，对于流处理计算语义有更多要求的需求也在增多。比如，一些金融机构部使用流处理应用来处理用户账户的借贷方，这种情况下，处理中的错误是不能容忍的，我们需要每一条消息都被处理一次，没有任何例外。

更正式的说，如果流处理应用程序消费消息 A 并产生消息 B，使得 B=F(A)，则 exactlyonce 则意味着仅当 B 成功时才认为 A 被消耗，反之亦然。

当在 Kafka 的 producer 和 consumer 的配置属性中使用 at-least-once 传入语义的时候，一个流处理应用程序能够处理下面的场景:

1. 由于内部重试，producer.send()方法使得消息 B 可能被重复写入。这将由 Procedure 的幂等特性解决，不是这篇文章其余部分的重点。
2. 我们可能会对消息 A 进行重新处理，这会导致重复的消息 B 被写入，违背了 exactly once 的处理语义，如果流处理应用程序在 B 写入成功但是在 A 被成功标记之前崩溃，则可能会被重新处理，因此，当它恢复时，他将在此消费 A 并再次写入 B，导致重复。
3. 最后，在分布式环境中，应用程序会崩溃或者更糟，一旦和系统其它部分连接丢失，通常情况下，新的实例会自动启动以取代丢失实例。通过这个过程，可能会有多个实例处理相同的输入 topic 并写入相同的输出 topic，从而导致重复的输出并违背 exactly once 的处理语义，这个我们称之为“僵尸实例”的问题。

我们在 Kafka 中设计了事务 API 来解决第二个和第三个问题，事务能够保证这些“读取-处理-写入”操作成为一个与那字操作并且在一个周期中保证精确处理，满足 exactly once 处理语义。

2. 事务语义

2.1. 多分区原子写入

事务能够保证 Kafka topic 下每个分区的原子写入。事务中所有的消息都将被成功写入或者丢弃。例如，处理过程中发生了异常并导致事务终止，这种情况下，事务中的消息都不会被 Consumer 读取。现在我们来看下 Kafka 是如何实现原子的“读取-处理-写入”过程的。

首先，我们来考虑一下原子“读取-处理-写入”周期是什么意思。简而言之，这意味着如果某个应用程序在某个 topic tp0 的偏移量 X 处读取到了消息 A，并且在对消息 A 进行了一些处理（如 B = F（A））之后将消息 B 写入 topic tp1，则只有当消息 A 和 B 被认为被成功地消费并一起发布，或者完全不发布时，整个读取过程写入操作是原子的。

现在，只有当消息 A 的偏移量 X 被标记为消耗时，消息 A 才被认为是从 topic tp0 消耗

的，消费到的数据偏移量（record offset）将被标记为提交偏移量（Committing offset）。在 Kafka 中，我们通过写入一个名为 offsets topic 的内部 Kafka topic 来记录 offset commit。消息仅在其 offset 被提交给 offsets topic 时才被认为成功消费。

由于 offset commit 只是对 Kafkatopic 的另一次写入，并且由于消息仅在提交偏移量时被视为成功消费，所以跨多个主题和分区的原子写入也启用原子“读取-处理-写入”循环：提交偏移量 X 到 offset topic 和消息 B 到 tp1 的写入将是单个事务的一部分，所以整个步骤都是原子的。

2.2. 粉碎“僵尸实例”

我们通过为每个事务 Producer 分配一个称为 transactional.id 的唯一标识符来解决僵尸实例的问题。在进程重新启动时能够识别相同的 Producer 实例。

API 要求事务性 Producer 的第一个操作应该是在 Kafka 集群中显示注册 transactional.id。当注册的时候，Kafka broker 用给定的 transactional.id 检查打开的事务并且完成处理。 Kafka 也增加了一个与 transactional.id 相关的 epoch。Epoch 存储每个 transactional.id 内部元数据。

一旦这个 epoch 被触发，任何具有相同的 transactional.id 和更旧的 epoch 的 Producer 被视为僵尸，并被围起来, Kafka 会拒绝来自这些 Procedure 的后续事务性写入。

2.3. 读事务消息

现在，让我们把注意力转向数据读取中的事务一致性。

Kafka Consumer 只有在事务实际提交时才会将事务消息传递给应用程序。也就是说，Consumer 不会提交作为整个事务一部分的消息，也不会提交属于中止事务的消息。

值得注意的是，上述保证不足以保证整个消息读取的原子性，当使用 Kafka consumer 来消费来自 topic 的消息时，应用程序将不知道这些消息是否被写为事务的一部分，因此他们不知道事务何时开始或结束；此外，给定的 Consumer 不能保证订阅属于事务一部分的所有 Partition，并且无法发现这一点，最终难以保证作为事务中的所有消息被单个 Consumer 处理。

简而言之：Kafka 保证 Consumer 最终只能提供非事务性消息或提交事务性消息。它将保留来自未完成事务的消息，并过滤掉已中止事务的消息。

3. 事务处理 Java API

事务功能主要是一个服务器端和协议级功能，任何支持它的客户端库都可以使用它。一个 Java 编写的使用 Kafka 事务处理 API 的“读取-处理-写入”应用程序示例：

[java] view plain copy

```
KafkaProducer producer = createKafkaProducer(
  “bootstrap.servers”, “localhost:9092”,
  “transactional.id”, “my-transactional-id”);

producer.initTransactions();

KafkaConsumer consumer = createKafkaConsumer(
  “bootstrap.servers”, “localhost:9092”,
  “group.id”, “my-group-id”,
  "isolation.level", "read_committed");

consumer.subscribe(singleton(“inputTopic”));

while (true) {
```

```
ConsumerRecords records = consumer.poll(Long.MAX_VALUE);
producer.beginTransaction();
for (ConsumerRecord record : records)
  producer.send(producerRecord("outputTopic", record));
producer.sendOffsetsToTransaction(currentOffsets(consumer), group);
producer.commitTransaction();
}
```

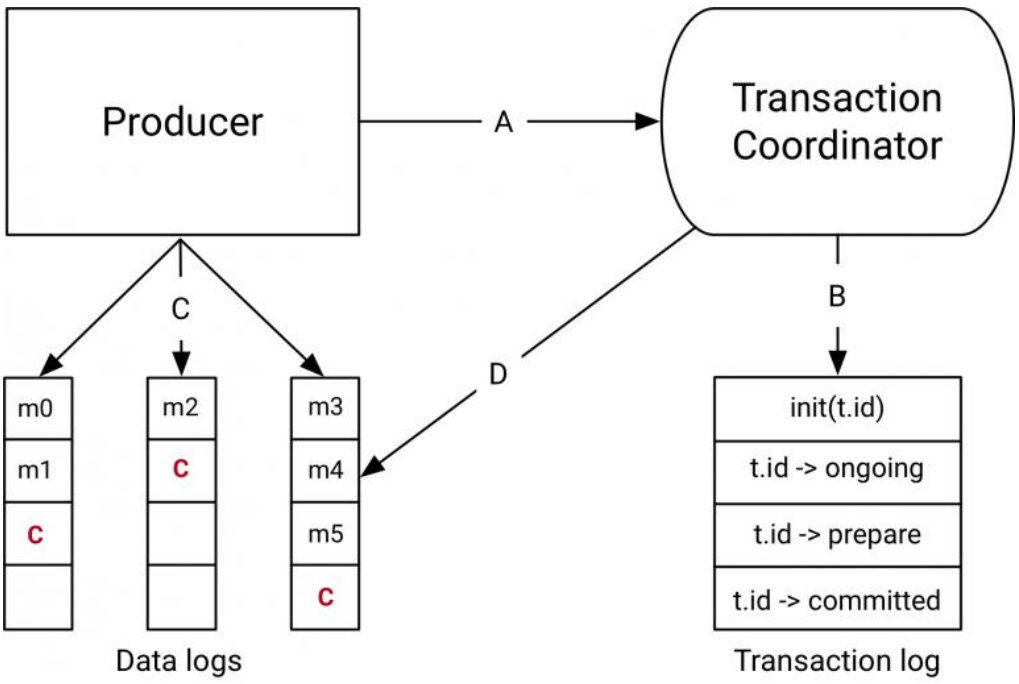
第 7-10 行指定 KafkaConsumer 只应读取非事务性消息，或从其输入主题中提交事务性消息。流处理应用程序通常在多个读取处理写入阶段处理其数据，每个阶段使用前一阶段的输出作为其输入。通过指定 read_committed 模式，我们可以在所有阶段完成一次处理。
第 1-5 行通过指定 transactional.id 配置并将其注册到 initTransactionsAPI 来设置 Procedure。在 producer.initTransactions（）返回之后，由具有相同的 transactional.id 的 Producer 的另一个实例启动的任何事务将被关闭和隔离。

第 14-21 行显示了“读取-处理-写入”循环的核心：读取一部分记录，启动事务，处理读取的记录，将处理的结果写入输出 topic，将消耗的偏移量发送到 offset topic，最后提交事务。有了上面提到的保证，我们就知道 offset 和输出记录将作为一个原子单位。

4. 事务工作原理

在本节中，我们将简要介绍上面介绍的事务 API 引入的新组件和新数据流。更详细的信息，你可以阅读原始设计文档，或观看介绍 Kafka MeetUp 的 Sliders。

下面示例的目标是在调试使用了事务的应用程序时，如何对事务进行优化以获得更好的性能。



5. 事务协调器和事务日志

在 Kafka 0.11.0 中与事务 API 一起引入的组件是上图右侧的事务 Coordinator 和事务日志。

事务 Coordinator 是每个 KafkaBroker 内部运行的一个模块。事务日志是一个内部的

Kafka Topic。每个 Coordinator 拥有事务日志所在分区的子集，即，这些 borker 中的分区都是 Leader。

每个 transactional.id 都通过一个简单的哈希函数映射到事务日志的特定分区。这意味着只有一个 Broker 拥有给定的 transactional.id。

通过这种方式，我们利用 Kafka 可靠的复制协议和 Leader 选举流程来确保事务协调器始终可用，并且所有事务状态都能够持久存储。

值得注意的是，事务日志只保存事务的最新状态而不是事务中的实际消息。消息只存储在实际的 Topic 的分区中。事务可以处于诸如“Ongoing”，“prepare commit”和“Completed”之类的各种状态中。正是这种状态和关联的元数据存储在事务日志中。

6. 数据流

数据流在抽象层面上有四种不同的类型。

A. producer 和事务 coordinator 的交互

执行事务时，Producer 向事务协调员发出如下请求：

1. initTransactions API 向 coordinator 注册一个 transactional.id。 此时，coordinator 使用该 transactional.id 关闭所有待处理的事务，并且会避免遇到僵尸实例。 每个 Producer 会话只发生一次。

2. 当 Producer 在事务中第一次将数据发送到分区时，首先向 coordinator 注册分区。

3. 当应用程序调用 commitTransaction 或 abortTransaction 时，会向 coordinator 发送一个请求以开始两阶段提交协议。

B. Coordinator 和事务日志交互

随着事务的进行，Producer 发送上面的请求来更新 Coordinator 上事务的状态。事务 Coordinator 会在内存中保存每个事务的状态，并且把这个状态写到事务日志中（这是以三种方式复制的，因此是持久保存的）。

事务 Coordinator 是读写事务日志的唯一组件。如果一个给定的 Borker 故障了，一个新的 Coordinator 会被选为新的事务日志的 Leader，这个事务日志分割了这个失效的代理，它从传入的分区中读取消息并在内存中重建状态。

C. Producer 将数据写入目标 Topic 所在分区

在 Coordinator 的事务中注册新的分区后，Producer 将数据正常地发送到真实数据所在分区。这与 producer.send 流程完全相同，但有一些额外的验证，以确保 Producer 不被隔离。

D. Topic 分区和 Coordinator 的交互

在 Producer 发起提交（或中止）之后，协调器开始两阶段提交协议。

在第一阶段，Coordinator 将其内部状态更新为“prepare_commit”并在事务日志中更新此状态。一旦完成了这个事务，无论发生什么事，都能保证事务完成。

Coordinator 然后开始阶段 2，在那里它将事务提交标记写入作为事务一部分的 Topic 分区。

这些事务标记不会暴露给应用程序，但是在 read_committed 模式下被 Consumer 使用来过滤掉被中止事务的消息，并且不返回属于开放事务的消息（即那些在日志中但没有事务标记与他们相关联）。

一旦标记被写入，事务协调器将事务标记为“完成”，并且 Producer 可以开始下一个事务。

7. 事务实践

现在我们已经理解了事务的语义以及它们是如何工作的，我们将注意力转向利用事务编写实际应用方面。

7.1. 如何选择事务 Id

transactional.id 在屏蔽僵尸中扮演着重要的角色。但是在一个保持一个在 Producer 会话中保持一致的标识符并且正确地屏蔽掉僵尸实例是有点棘手的。

正确隔离僵尸实例的关键在于确保读取进程写入周期中的输入 Topic 和分区对于给定的 transactional.id 总是相同的。如果不是这样，那么有可能丢失一部分消息。

例如，在分布式流处理应用程序中，假设 Topic 分区 tp0 最初由 transactional.idT0 处理。如果在某个时间点之后，它可以通过 transactional.id T1 映射到另一个 Producer，那么 T0 和 T1 之间就不会有栅栏了。所以 tp0 的消息可能被重新处理，违反了一次处理保证。

实际上，可能需要将输入分区和 transactional.id 之间的映射存储在外部存储中，或者对其进行静态编码。Kafka Streams 选择后一种方法来解决这个问题。

7.2. 事务性能以及如何优化？

Ø **Producer 打开事务之后的性能**

让我们把注意力转向事务如何执行。

首先，事务只造成中等的写入放大。额外的写入在于：

对于每个事务，我们都有额外的 RPC 向 Coordinator 注册分区。这些是批处理的，所以我们比事务中的 partition 有更少的 RPC。

在完成事务时，必须将一个事务标记写入参与事务的每个分区。同样，事务 Coordinator 在单个 RPC 中批量绑定到同一个 Borker 的所有标记，所以我们在那里保存 RPC 开销。但是在事务中对每个分区进行额外的写操作是无法避免的。

最后，我们将状态更改写入事务日志。这包括写入添加到事务的每批分区，“prepare_commit”状态和“complete_commit”状态。

我们可以看到，开销与作为事务一部分写入的消息数量无关。所以拥有更高吞吐量的关键是每个事务包含更多的消息。

实际上，对于 Producer 以最大吞吐量生产 1KB 记录，每 100ms 提交消息导致吞吐量仅降低 3%。较小的消息或较短的事务提交间隔会导致更严重的降级。

增加事务时间的主要折衷是增加了端到端延迟。回想一下，Consum 阅读事务消息不会传递属于公开传输的消息。因此，提交之间的时间间隔越长，消耗的应用程序就越需要等待，从而增加了端到端的延迟。

Ø **Consumer 打开之后的性能**

Consumer 在开启事务的场景比 Producer 简单得多，它需要做的是：

1. 过滤掉属于中止事务的消息。
2. 不返回属于公开事务一部分的事务消息。

因此，当以 read_committed 模式读取事务消息时，事务 Consumer 的吞吐量没有降低。这样做的主要原因是我们在读取事务消息时保持零拷贝读取。

此外，Consumer 不需要任何缓冲等待事务完成。相反，Broker 不允许提前抵消包括公开事务。

因此，Consumer 是非常轻巧和高效的。

8 Kafka 的原理？

1） Kafka 架构组件

Kafka 中发布订阅的对象是 topic。我们可以为每类数据创建一个 topic，把向 topic 发布消息的客户端称作 producer，从 topic 订阅消息的客户端称作 consumer。Producers 和 consumers

可以同时从多个 topic 读写数据。一个 kafka 集群由一个或多个 broker 服务器组成，它负责持久化和备份具体的 kafka 消息。

topic：消息存放的目录即主题

Producer：生产消息到 topic 的一方

Consumer：订阅 topic 消费消息的一方

Broker：Kafka 的服务实例就是一个 broker

2）Kafka Topic&Partition

消息发送时都被发送到一个 topic，其本质就是一个目录，而 topic 由是由一些 Partition Logs(分区日志)组成，每个 Partition 中的消息都是有序的，生产的消息被不断追加到 Partition log 上，其中的每一个消息都被赋予了一个唯一的 offset 值。

Kafka 集群会保存所有的消息，不管消息有没有被消费；我们可以设定消息的过期时间，只有过期的数据才会被自动清除以释放磁盘空间。比如我们设置消息过期时间为 2 天，那么这 2 天内的所有消息都会被保存到集群中，数据只有超过了两天才会被清除。

Kafka 需要维持的元数据只有一个–消费消息在 Partition 中的 offset 值，Consumer 每消费一个消息，offset 就会加 1。其实消息的状态完全是由 Consumer 控制的，Consumer 可以跟踪和重设这个 offset 值，这样的话 Consumer 就可以读取任意位置的消息。

把消息日志以 Partition 的形式存放有多重考虑，第一，方便在集群中扩展，每个 Partition 可以通过调整以适应它所在的机器，而一个 topic 又可以有多个 Partition 组成，因此整个集群就可以适应任意大小的数据了；第二就是可以提高并发，因为可以以 Partition 为单位读写了。

3）Kafka 核心组件

（1） Replications、Partitions 和 Leaders

通过上面介绍的我们可以知道，kafka 中的数据是持久化的并且能够容错的。Kafka 允许用户为每个 topic 设置副本数量，副本数量决定了有几个 broker 来存放写入的数据。如果你的副本数量设置为 3，那么一份数据就会被存放在 3 台不同的机器上，那么就允许有 2 个机器失败。一般推荐副本数量至少为 2，这样就可以保证增减、重启机器时不会影响到数据消费。如果对数据持久化有更高的要求，可以把副本数量设置为 3 或者更多。

Kafka 中的 topic 是以 partition 的形式存放的，每一个 topic 都可以设置它的 partition 数量，Partition 的数量决定了组成 topic 的 log 的数量。Producer 在生产数据时，会按照一定规则（这个规则是可以自定义的）把消息发布到 topic 的各个 partition 中。上面将的副本都是以 partition 为单位的，不过只有一个 partition 的副本会被选举成 leader 作为读写用。

关于如何设置 partition 值需要考虑的因素。一个 partition 只能被一个消费者消费（一个消费者可以同时消费多个 partition），因此，如果设置的 partition 的数量小于 consumer 的数量，就会有消费者消费不到数据。所以，推荐 partition 的数量一定要大于同时运行的 consumer 的数量。另外一方面，建议 partition 的数量大于集群 broker 的数量，这样 leader partition 就可以均匀的分布在各个 broker 中，最终使得集群负载均衡。在 Cloudera,每个 topic 都有上百个 partition。需要注意的是，kafka 需要为每个 partition 分配一些内存来缓存消息数据，如果 partition 数量越大，就要为 kafka 分配更大的 heap space。

（2）Producers

Producers 直接发送消息到 broker 上的 leader partition，不需要经过任何中介一系列的路由转发。为了实现这个特性，kafka 集群中的每个 broker 都可以响应 producer 的请求，并返回 topic 的一些元信息，这些元信息包括哪些机器是存活的，topic 的 leader partition 都在哪，现阶段哪些 leader partition 是可以直接被访问的。

Producer 客户端自己控制着消息被推送到哪些 partition。实现的方式可以是随机分配、实现一类随机负载均衡算法，或者指定一些分区算法。Kafka 提供了接口供用户实现自定义的分

区，用户可以为每个消息指定一个 partitionKey，通过这个 key 来实现一些 hash 分区算法。比如，把 userid 作为 partitionkey 的话，相同 userid 的消息将会被推送到同一个分区。

以 Batch 的方式推送数据可以极大的提高处理效率，kafka Producer 可以将消息在内存中累计到一定数量后作为一个 batch 发送请求。Batch 的数量大小可以通过 Producer 的参数控制，参数值可以设置为累计的消息的数量（如 500 条）、累计的时间间隔（如 100ms）或者累计的数据大小(64KB)。通过增加 batch 的大小，可以减少网络请求和磁盘 IO 的次数，当然具体参数设置需要在效率和时效性方面做一个权衡。

Producers 可以异步的并行的向 kafka 发送消息，但是通常 producer 在发送完消息之后会得到一个 future 响应，返回的是 offset 值或者发送过程中遇到的错误。这其中有个非常重要的参数“acks”,这个参数决定了 producer 要求 leader partition 收到确认的副本个数，如果 acks 设置数量为 0，表示 producer 不会等待 broker 的响应，所以，producer 无法知道消息是否发送成功，这样有可能会导致数据丢失，但同时，acks 值为 0 会得到最大的系统吞吐量。

若 acks 设置为 1，表示 producer 会在 leader partition 收到消息时得到 broker 的一个确认，这样会有更好的可靠性，因为客户端会等待直到 broker 确认收到消息。若设置为-1，producer 会在所有备份的 partition 收到消息时得到 broker 的确认，这个设置可以得到最高的可靠性保证。

Kafka 消息有一个定长的 header 和变长的字节数组组成。因为 kafka 消息支持字节数组，也就使得 kafka 可以支持任何用户自定义的序列号格式或者其它已有的格式如 Apache Avro、protobuf 等。Kafka 没有限定单个消息的大小，但我们推荐消息大小不要超过 1MB,通常一般消息大小都在 1~10kB 之前。

（3） Consumers

Kafka 提供了两套 consumer api，分为 high-level api 和 sample-api。Sample-api 是一个底层的 API，它维持了一个和单一 broker 的连接，并且这个 API 是完全无状态的，每次请求都需要指定 offset 值，因此，这套 API 也是最灵活的。

在 kafka 中，当前读到消息的 offset 值是由 consumer 来维护的，因此，consumer 可以自己决定如何读取 kafka 中的数据。比如，consumer 可以通过重设 offset 值来重新消费已消费过的数据。不管有没有被消费，kafka 会保存数据一段时间，这个时间周期是可配置的，只有到了过期时间，kafka 才会删除这些数据。

High-level API 封装了对集群中一系列 broker 的访问，可以透明的消费一个 topic。它自己维持了已消费消息的状态，即每次消费的都是下一个消息。

High-level API 还支持以组的形式消费 topic，如果 consumers 有同一个组名，那么 kafka 就相当于一个队列消息服务，而各个 consumer 均衡的消费相应 partition 中的数据。若 consumers 有不同的组名，那么此时 kafka 就相当与一个广播服务，会把 topic 中的所有消息广播到每个 consumer。

五 Hbase

1 Hbase 热点问题 读写请求会集中到某一个 RegionServer 上如何处理

对表采用预分区，将数据分散到不同的 region 中，region 交给不同的 regionserver 来维护

2 hbase 查询一条记录的方法是什么？Hbase 写入一条记录的方法是什么？

Get get = new Get(rowkey);
Put put = new Put (rowkey)
Put.add…

3 Hbase 优化及 rowkey 设计。

1） 高可用

在 HBase 中 Hmaster 负责监控 RegionServer 的生命周期，均衡 RegionServer 的负载，如果 Hmaster 挂掉了，那么整个 HBase 集群将陷入不健康的状态，并且此时的工作状态并不会维持太久。所以 HBase 支持对 Hmaster 的高可用配置。

1．关闭 HBase 集群（如果没有开启则跳过此步）

[atguigu@hadoop102 hbase]$ bin/stop-hbase.sh

2．在 conf 目录下创建 backup-masters 文件

[atguigu@hadoop102 hbase]$ touch conf/backup-masters

3．在 backup-masters 文件中配置高可用 HMaster 节点

[atguigu@hadoop102 hbase]$ echo hadoop103 > conf/backup-masters

4．将整个 conf 目录 scp 到其他节点

[atguigu@hadoop102 hbase]$ scp -r conf/ hadoop103:/opt/module/hbase/

[atguigu@hadoop102 hbase]$ scp -r conf/ hadoop104:/opt/module/hbase/

5．打开页面测试查看

http://hadooo102:16010

2）预分区

每一个 region 维护着 startRow 与 endRowKey，如果加入的数据符合某个 region 维护的 rowKey 范围，则该数据交给这个 region 维护。那么依照这个原则，我们可以将数据所要投放的分区提前大致的规划好，以提高 HBase 性能。

1．手动设定预分区

hbase> create 'staff1','info','partition1',SPLITS => ['1000','2000','3000','4000']

2．生成 16 进制序列预分区

create 'staff2','info','partition2',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

3．按照文件中设置的规则预分区

创建 splits.txt 文件内容如下：

aaaa

bbbb

cccc

dddd

然后执行：

create 'staff3','partition3',SPLITS_FILE => 'splits.txt'

4．使用 JavaAPI 创建预分区

```
//自定义算法，产生一系列 Hash 散列值存储在二维数组中
byte[][] splitKeys = 某个散列值函数
//创建 HBaseAdmin 实例
HBaseAdmin hAdmin = new HBaseAdmin(HBaseConfiguration.create());
//创建 HTableDescriptor 实例
HTableDescriptor tableDesc = new HTableDescriptor(tableName);
//通过 HTableDescriptor 实例和散列值二维数组创建带有预分区的 HBase 表
hAdmin.createTable(tableDesc, splitKeys);
```

3）RowKey 设计

一条数据的唯一标识就是 rowkey，那么这条数据存储于哪个分区，取决于 rowkey 处于哪个一个预分区的区间内，设计 rowkey 的主要目的 ，就是让数据均匀的分布于所有的 region 中，在一定程度上防止数据倾斜。接下来我们就谈一谈 rowkey 常用的设计方案。

1．生成随机数、hash、散列值

比如：

原本 rowKey 为 1001 的，SHA1 后变成：dd01903921ea24941c26a48f2cec24e0bb0e8cc7

原本 rowKey 为 3001 的，SHA1 后变成：49042c54de64a1e9bf0b33e00245660ef92dc7bd

原本 rowKey 为 5001 的，SHA1 后变成：7b61dec07e02c188790670af43e717f0f46e8913

在做此操作之前，一般我们会选择从数据集中抽取样本，来决定什么样的 rowKey 来 Hash 后作为每个分区的临界值。

2．字符串反转

20170524000001 转成 10000042507102

20170524000002 转成 20000042507102

这样也可以在一定程度上散列逐步 put 进来的数据。

3．字符串拼接

20170524000001_a12e

20170524000001_93i7

4）内存优化

HBase 操作过程中需要大量的内存开销，毕竟 Table 是可以缓存在内存中的，一般会分配整个可用内存的 70%给 HBase 的 Java 堆。但是不建议分配非常大的堆内存，因为 GC 过程持续太久会导致 RegionServer 处于长期不可用状态，一般 16~48G 内存就可以了，如果因为框架占用内存过高导致系统内存不足，框架一样会被系统服务拖死。

5）基础优化

1．允许在 HDFS 的文件中追加内容
hdfs-site.xml、hbase-site.xml
属性：dfs.support.append
解释：开启 HDFS 追加同步，可以优秀的配合 HBase 的数据同步和持久化。默认值为 true。
2．优化 DataNode 允许的最大文件打开数
hdfs-site.xml
属性：dfs.datanode.max.transfer.threads
解释：HBase 一般都会同一时间操作大量的文件，根据集群的数量和规模以及数据动作，设置为 4096 或者更高。默认值：4096
3．优化延迟高的数据操作的等待时间
hdfs-site.xml
属性：dfs.image.transfer.timeout
解释：如果对于某一次数据操作来讲，延迟非常高，socket 需要等待更长的时间，建议把该值设置为更大的值（默认 60000 毫秒），以确保 socket 不会被 timeout 掉。
4．优化数据的写入效率
mapred-site.xml
属性：
mapreduce.map.output.compress
mapreduce.map.output.compress.codec
解释：开启这两个数据可以大大提高文件的写入效率，减少写入时间。第一个属性值修改为 true，第二个属性值修改为：org.apache.hadoop.io.compress.GzipCodec 或者其他压缩方式。
5．优化 DataNode 存储
属性：dfs.datanode.failed.volumes.tolerated
解释：默认为 0，意思是当 DataNode 中有一个磁盘出现故障，则会认为该 DataNode shutdown 了。如果修改为 1，则一个磁盘出现故障时，数据会被复制到其他正常的 DataNode 上，当前的 DataNode 继续工作。
6．设置 RPC 监听数量
hbase-site.xml
属性：hbase.regionserver.handler.count
解释：默认值为 30，用于指定 RPC 监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。
7．优化 HStore 文件大小
hbase-site.xml
属性：hbase.hregion.max.filesize
解释：默认值 10737418240（10GB），如果需要运行 HBase 的 MR 任务，可以减小此值，因为一个 region 对应一个 map 任务，如果单个 region 过大，会导致 map 任务执行时间过长。该值的意思就是，如果 HFile 的大小达到这个数值，则这个 region 会被切分为两个 Hfile。
8．优化 hbase 客户端缓存
hbase-site.xml
属性：hbase.client.write.buffer
解释：用于指定 HBase 客户端缓存，增大该值可以减少 RPC 调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少 RPC 次数的目的。
9．指定 scan.next 扫描 HBase 所获取的行数
hbase-site.xml

属性：hbase.client.scanner.caching
解释：用于指定 scan.next 方法获取的默认行数，值越大，消耗内存越大。
10．flush、compact、split 机制
当 MemStore 达到阈值，将 Memstore 中的数据 Flush 进 Storefile；compact 机制则是把 flush 出来的小文件合并成大的 Storefile 文件。split 则是当 Region 达到阈值，会把过大的 Region 一分为二。
涉及属性：
即：128M 就是 Memstore 的默认阈值
hbase.hregion.memstore.flush.size：134217728
即：这个参数的作用是当单个 HRegion 内所有的 Memstore 大小总和超过指定值时，flush 该 HRegion 的所有 memstore。RegionServer 的 flush 是通过将请求添加一个队列，模拟生产消费模型来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发 OOM。
hbase.regionserver.global.memstore.upperLimit：0.4
hbase.regionserver.global.memstore.lowerLimit：0.38
即：当 MemStore 使用内存总量达到 hbase.regionserver.global.memstore.upperLimit 指定值时，将会有多个 MemStores flush 到文件中，MemStore flush 顺序是按照大小降序执行的，直到刷新到 MemStore 使用内存略小于 lowerLimit

① 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。
② 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。
③ Rowkey 唯一原则
必须在设计上保证其唯一性。

六 Oozie

1 有没有试过 OZ 调度 Sqoop

将每天的 MR 分析结果导入 mysql 中。

2 有没有使用 OZ 调度 hadoop 任务

定时每天分析前一天指标 MR。

七 Hadoop HA

1 HAnamenode 是如何工作的?

1） HDFS-HA 工作机制

通过双 namenode 消除单点故障

2）HDFS-HA 工作要点

（1）元数据管理方式需要改变：

内存中各自保存一份元数据；

Edits 日志只有 Active 状态的 namenode 节点可以做写操作；

两个 namenode 都可以读取 edits；

共享的 edits 放在一个共享存储中管理（qjournal 和 NFS 两个主流实现）；

（2）需要一个状态管理功能模块

实现了一个 zkfailover，常驻在每一个 namenode 所在的节点，每一个 zkfailover 负责监控自己所在 namenode 节点，利用 zk 进行状态标识，当需要进行状态切换时，由 zkfailover 来负责切换，切换时需要防止 brain split 现象的发生。

（3）必须保证两个 NameNode 之间能够 ssh 无密码登录。

（4）隔离（Fence），即同一时刻仅仅有一个 NameNode 对外提供服务

3） HDFS-HA 自动故障转移工作机制

前面学习了使用命令 hdfs haadmin -failover 手动进行故障转移，在该模式下，即使现役 NameNode 已经失效，系统也不会自动从现役 NameNode 转移到待机 NameNode，下面学习如何配置部署 HA 自动进行故障转移。自动故障转移为 HDFS 部署增加了两个新组件：ZooKeeper 和 ZKFailoverController（ZKFC）进程。ZooKeeper 是维护少量协调数据，通知客户端这些数据的改变和监视客户端故障的高可用服务。HA 的自动故障转移依赖于 ZooKeeper 的以下功能：

（1）故障检测：集群中的每个 NameNode 在 ZooKeeper 中维护了一个持久会话，如果机器崩溃，ZooKeeper 中的会话将终止，ZooKeeper 通知另一个 NameNode 需要触发故障转移。

（2）现役 NameNode 选择：ZooKeeper 提供了一个简单的机制用于唯一的选择一个节点为

active 状态。如果目前现役 NameNode 崩溃，另一个节点可能从 ZooKeeper 获得特殊的排外锁以表明它应该成为现役 NameNode。

ZKFC 是自动故障转移中的另一个新组件，是 ZooKeeper 的客户端，也监视和管理 NameNode 的状态。每个运行 NameNode 的主机也运行了一个 ZKFC 进程，ZKFC 负责：

（1）健康监测：ZKFC 使用一个健康检查命令定期地 ping 与之在相同主机的 NameNode，只要该 NameNode 及时地回复健康状态，ZKFC 认为该节点是健康的。如果该节点崩溃，冻结或进入不健康状态，健康监测器标识该节点为非健康的。

（2）ZooKeeper 会话管理：当本地 NameNode 是健康的，ZKFC 保持一个在 ZooKeeper 中打开的会话。如果本地 NameNode 处于 active 状态，ZKFC 也保持一个特殊的 znode 锁，该锁使用了 ZooKeeper 对短暂节点的支持，如果会话终止，锁节点将自动删除。

（3）基于 ZooKeeper 的选择：如果本地 NameNode 是健康的，且 ZKFC 发现没有其它的节点当前持有 znode 锁，它将为自己获取该锁。如果成功，则它已经赢得了选择，并负责运行故障转移进程以使它的本地 NameNode 为 active。故障转移进程与前面描述的手动故障转移相似，首先如果必要保护之前的现役 NameNode，然后本地 NameNode 转换为 active 状态。