

Hive DDL - 建库

- 语法

```
1 CREATE [REMOTE] (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
2 [COMMENT database_comment]
3 [LOCATION hdfs_path]
4 [MANAGEDLOCATION hdfs_path]
5 [WITH DBPROPERTIES (property_name=property_value, ...)];
```

- 建库案例

```
1 create database if not exists day06;
2 use day06;
3 # 确认当前数据库的位置
4 select current_database();
```

Hive DDL - 建表

- 常用

```
1 create table table_name
2 partitioned by (伪名 类型,伪名 类型...)
3 clustered by ()[sorted by ()] into buckets
4 row format delimited|serde fields terminated by ','
5 stored 名
6 location 路径
7 tblproperties ()
```

if not exists - 支持重跑机制

```
1 #如果表存在则跳过,不会报错
2 create table if not exists t_1(id int,name string,age int);
```

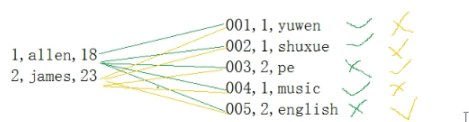
Hive的数据类型

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types>

- Hive 支持原生数据类型和符合数据类型
- Hive 原生数据类型包含

需求：查询出每个学生的选修课情况？

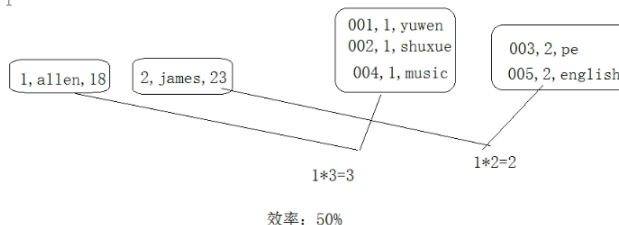
```
select * from t_a join t_b on t_a.id = t_b.snum
```



分桶表可以针对 join 场景进行优化。

关键：针对两边的表 对 join 的字段进行分桶操作。

2*5 = 10
符合条件的：5
效率：50%



- Hive除了支持sql类型之外,还支持Java数据类型 string等
- Hive除了支持基础数据类型之外,还支持符合类型,(array数组 map映射 union struct结构)
 - 针对复合类型的数据,要想直接从文件中解析成功,还必须配合分隔符指定的语法
- Hive中大小写不敏感
- 在建表的时候,最好表的字段类型要和文件中的类型保持一致
 - 如果不一致,Hive会尝试进行类型隐式转换,不保证转换成功,如果不成功,显示null

```
1 --创建数据库并切换使用
2 create database if not exists itheima;
3 use itheima;
4
5 --建表
6 create table t_archer(
7     id int comment "ID",
8     name string comment "英雄名称",
9     hp_max int comment "最大生命",
10    mp_max int comment "最大法力",
11    attack_max int comment "最高物攻",
12    defense_max int comment "最大物防",
13    attack_range string comment "攻击范围",
14    role_main string comment "主要定位",
15    role_assist string comment "次要定位"
16 ) comment "王者荣耀射手信息"
17 row format delimited
18 fields terminated by "\t";
19
20 --查看表数据 查看表的元数据信息
21 select * from t_archer;
22 desc formatted t_archer;
```

```

23
24 --上传文件到表对应的HDFS目录下
25 [root@node1 hivedata]# hadoop fs -put archer.txt
   /user/hive/warehouse/itheima.db/t_archer
26
27 [root@node1 hiedata]# pwd
28 /root/hivedata

```

Hive读写HDFS上文件

- 机制:SerDe(Serializer and Deserializer) 序列化机制
- Hive使用SerDe机制读写HDFS上文件
- 读文件 -- 序列化
 - HDFS files --> InputFileFormat --> <key, value> --> ==Deserializer== --> Row object

```

1 <br class="Apple-interchange-newline"><div></div>
2
3 #1、使用InputFileFormat（默认实现 TextInputFormat ）读取hdfs上文件
4     一行一行读取数据，返回<k, v>键值对类型
5 #2、返回<key, value>，其中数据存储在value中
6
7 #3、使用Deserializer反序列化动作读取value 解析成为对象（Row object）
8     默认的序列化类 LazysimpleSerDe

```

- 写文件 -- 反序列化
 - Row object --> ==Serializer== --> <key, value> --> OutputFileFormat --> HDFS files
- 分隔符指定语法
 - 语法格式

```

1 ROW FORMAT DELIMITED | SERDE
2
3 ROW FORMAT DELIMITED 表示使用LazySimpleSerDe类进行序列化解析数据
4
5 ROW FORMAT SERDE      表示使用其他SerDe类进行序列化解析数据

```

- ROW FORMAT DELIMITED具体的子语法

```

1 row format delimited
2 [fields terminated by char] #指定字段之间的分隔符
3 [collection items terminated by char] #指定集合元素之间的分隔符
4 [map keys terminated by char] #指定map类型kv之间的分隔符
5 [lines terminated by char] #指定换行符

```

- 默认分隔符

- Hive在建表的时候, 如果没有row format语法, 则该表使用==\001默认分隔符==进行字段分割;
- 如果此时文件中的数据字段之间的分隔符也是\001, 那么就可以直接映射成功。
- ==针对默认分隔符, 其是一个不可见分隔符, 在代码层面是\001表示==
- 在vim编辑器中, 连续输入ctrl+v、ctrl+a;
- 在实际工作中, Hive最喜欢的就是\001分隔符, 在清洗数据的时候, ==有意识==的把数据之间的分隔符指定为\001;

- 例子

```
1  --建表
2  create table t_hot_hero_skin_price(
3      id int,
4      name string,
5      win_rate int,
6      skin_price map<string,int>
7  )
8  row format delimited
9  fields terminated by ',' --字段之间分隔符
10 collection items terminated by '-' --集合元素之间分隔符
11 map keys terminated by ':'; --集合元素kv之间分隔符;
12
13 --上传数据
14 hadoop fs -put hot_hero_skin_price.txt
    /user/hive/warehouse/itheima.db/t_hot_hero_skin_price
15
16 select *
17 from t_hot_hero_skin_price;
18
19
20 --有点想法: 就把map数据类型当成字符串映射如何?
21 create table t_hot_hero_skin_price_str(
22     id int,
23     name string,
24     win_rate int,
25     skin_price string
26 )
27 row format delimited
28 fields terminated by ',';
```

```

30  hadoop fs -put hot_hero_skin_price.txt
   /user/hive/warehouse/itheima.db/t_hot_hero_skin_price_str
31
32  --结论
33      •1、不管使用map还是使用string来定义数据 都能解析映射成功
34      •2、区别在于使用的过程中 一个是针对map类型数据处理 一个是针对string类型的数据处理
35
36
37  select skin_price from t_hot_hero_skin_price;
38  select skin_price from t_hot_hero_skin_price_str;
39
40
41  select skin_price["至尊宝"] from t_hot_hero_skin_price limit 1;
42  select skin_price["至尊宝"] from t_hot_hero_skin_price_str limit 1; --语法错误
43
44
45  --建表 不指定分隔符
46  create table t_team_ace_player(
47      id int,
48      team_name string,
49      ace_player_name string
50  ); --没有指定row format语句 此时采用的是默认的\001作为字段的分隔符
51
52  hadoop fs -put team_ace_player.txt /user/hive/warehouse/itheima.db/t_team_ace_player
53
54  select * from t_team_ace_player;

```

内部表、外部表

```

1  --创建内部表
2  create table student_inner(Sno int,Sname string,Sex string,Sage int,Sdept string) row
   format delimited fields terminated by ',';
3
4  --创建外部表 关键字external
5  create external table student_external(Sno int,Sname string,Sex string,Sage int,Sdept
   string) row format delimited fields terminated by ',';
6
7  --上传文件到内部表、外部表中
8  hadoop fs -put students.txt /user/hive/warehouse/itheima.db/student_inner
9  hadoop fs -put students.txt /user/hive/warehouse/itheima.db/student_external

```

```

10
11 --好像没啥区别 都能映射成功 数据也都在HDFS上
12
13 --针对内部表、外部表 进行drop删除操作
14 drop table student_inner; --内部表在删除的时候 元数据和数据都会被删除
15 drop table student_external; --外部表在删除的时候 只删除元数据 而HDFS上的数据文件不会动

```

- 外部表有什么好处

1 最大的好处是防止误操作删除表的时候 把表的数据一起删除。

- 可以通过命令去查询表的元数据信息 获取表的类型

```

1 desc formatted table_name;
2
3 MANAGED_TABLE      内部表、受控表
4 EXTERNAL_TABLE     外部表

```

- 什么时候使用外部表

```

1 # Hive数据仓库
2 # Hive数据仓库分层 ODS > DW > APP
3 ODS数据内部表还是外部表？
4 外部表
5 APP层，自定义应用层，内部表还是内部表？
6 内部表

```

表数据在HDFS上存储路径

- 存储路径由hive.metastore.warehouse.dir 属性指定。默认值是：/user/hive/warehouse
- 不管是内部表，还是外部表，在HDFS上的路径如下：

```

1 /user/hive/warehouse/itcast.db/t_array
2
3 /user/hive/warehouse/数据库名.db/表名

```

例子

```

1 --在建表的时候 可以使用location关键字指定表的路径在HDFS任意位置
2 create table t_team_ace_player_location(
3   id int,
4   team_name string,
5   ace_player_name string)

```

```
6 location '/data'; --使用location关键字指定本张表数据在hdfs上的存储路径
7
8 --此时再上传数据 就必须上传到指定的目录下 否则就解析映射失败了
```

- 在实际开发中，最好集中维护管理Hive表数据，避免文件在HDFS随意存放。

分区表的引入

- 创建一个分区表 partitioned by(伪列 类型)
- 引入案例

```
1 create table t_all_hero_part(
2     id int,
3     name string,
4     hp_max int,
5     mp_max int,
6     attack_max int,
7     defense_max int,
8     attack_range string,
9     role_main string,
10    role_assist string
11 )
12 partitioned by(role_main string)
13 row format delimited
14 fields terminated by "\t";
15
16 --错误说 分区字段重复了 好家伙
17 Error: Error while compiling statement: FAILED: SemanticException [Error 10035]: Column
18 repeated in partitioning columns (state=42000,code=10035)
19
20 --分区表建表
21 create table t_all_hero_part(
22     id int,
23     name string,
24     hp_max int,
25     mp_max int,
26     attack_max int,
27     defense_max int,
28     attack_range string,
29     role_main string,
30     role_assist string
```

```

31 ) partitioned by (role string)--注意 这里是分区字段
32 row format delimited
33 fields terminated by "\t";
34
35 --查询分区表 发现分区字段也显示出来了
36 select * from t_all_hero_part;

```

分区表的数据加载

- 静态分区
- 场景
 - 分区的字段格式固定、分区数较小
 - 每个省的订单总量，统计每个学科的学生就业平均薪资

```

1 --静态加载分区表数据
2 load data local inpath '/root/data/archer.txt' into table t_all_hero_part
  partition(roles='sheshou');
3 load data local inpath '/root/data/assassin.txt' into table t_all_hero_part
  partition(roles='cike');
4 load data local inpath '/root/data/mage.txt' into table t_all_hero_part
  partition(roles='fashi');
5 load data local inpath '/root/data/support.txt' into table t_all_hero_part
  partition(roles='fuzhu');
6 load data local inpath '/root/data/tank.txt' into table t_all_hero_part
  partition(roles='tanke');
7 load data local inpath '/root/data/warrior.txt' into table t_all_hero_part
  partition(roles='zhanshi');
8
9 --查询一下验证是否加载成功
10 select * from t_all_hero_part;

```

- 动态分区
- 设置允许动态分区、设置动态分区模式

```

1 --动态分区
2 set hive.exec.dynamic.partition=true; --注意hive3已经默认开启了
3 set hive.exec.dynamic.partition.mode=nonstrict;
4
5 --模式分为strict严格模式 nonstrict非严格模式
6 严格模式要求 分区字段中至少有一个分区是静态分区。

```

动态分区加载数据

- ==insert + select==

- 插入的数据来自于后面的查询语句返回的结果。
- 查询返回的内容，其字段类型、顺序、个数要和待插入的表保持一致。

```
1  --创建一张新的分区表 t_all_hero_part_dynamic
2  create table t_all_hero_part_dynamic(
3      id int,
4      name string,
5      hp_max int,
6      mp_max int,
7      attack_max int,
8      defense_max int,
9      attack_range string,
10     role_main string,
11     role_assist string
12 ) partitioned by (role string)
13 row format delimited
14 fields terminated by "\t";
15
16 --执行动态分区插入  --注意 分区值并没有手动写死指定
17 insert into table t_all_hero_part_dynamic partition(role)
18 select tmp.*,tmp.role_main from t_all_hero_part tmp;
19
20 --查询验证结果
21 select * from t_all_hero_part_dynamic;
22
23 -- 在执行动态分区插入数据的时候，如果是严格模式strict，要求至少一个分区为静态分区？
24 --partition(guojia="zhongguo",sheng) --第一个分区写死了（静态） 符合严格模式。
25 --partition(guojia,sheng) --两个分区都是动态确定的 需要非严格模式
```

分区表的使用

```
1  --非分区表 全表扫描过滤查询
2  select count(*) from t_all_hero where role_main="archer" and hp_max >6000;
3
4  --分区表 先基于分区过滤 再查询
5  select count(*) from t_all_hero_part where roles="sheshou" and hp_max >6000;
```

分区表注意事项

- ==分区表的字段不能是表中已有的字段==；分区的字段也会显示在查询结果上；

- 分区的字段是虚拟的字段，出现在表所有字段的后面，其值来自于加载数据到表中的时候手动指定。
- 分区在底层的形式就是==以文件夹管理不同的文件==；不同文件夹就是表不同分区；文件夹的名字
- 分区表是一种优化表，建表的时候可以不使用，但是，当==创建分区表之后，使用分区字段查询可以减少全表扫描，提高查询的效率==。

申明分区

如果分区不存在则创建

```
1 alter table 表名 add partition if not exists partition(key=value)
```

多重分区表

- 分区表支持基于多个字段进行分区
 - partitioned by(字段1, 字段2....)
- 多个分区之间是一种==递进关系==，可以理解为在==前一个分区的基础上继续分区==；从底层来说就是文件夹下面继续划分子文件夹；
- ==常见的多分区就是2个分区==；

```
1 --以国家、省创建分区表
2 create table t_user_double_p(id int,name string,country string) partitioned by(guojia
  string,sheng string) row format delimited fields terminated by ',';
3
4 --加载数据到多分区表中
5 load data local inpath '/root/data/china_sh.txt' into table t_user_double_p
  partition(guojia="zhongguo",sheng="shanghai");
6
7 load data local inpath '/root/data/china_sz.txt' into table t_user_double_p
  partition(guojia="zhongguo",sheng="shenzhen");
8
9 load data local inpath '/root/data/usa_dezhou.txt' into table t_user_double_p
  partition(guojia="meiguo",sheng="dezhou");
10
11 --查询来自于中国深圳的用户有哪些？
12 select * from t_user_double_p where guojia="zhongguo"and sheng="shenzhen";
```

动静分区混合操作:


```
1 格式:
2 insert into|overwrite table 表名 partition(分区字段1=值1, 分区字段2 ...) + select...语
  句
3
```

- 4 如果使用动态分区，必须开启以下内容：
- 5 `set hive.exec.dynamic.partition=true; -- 开启动态分区支持`
- 6 `set hive.exec.dynamic.partition.mode=nonstrict; -- 关闭严格模式`
- 7
- 8 注意：在使用动态分区的时候，将动态分区字段数据放置查询的结果最后面(注意顺序一致)

有序动态分区

- 1 目的：此配置的目的，是为了解决在动态分区过程中，如果分区过多，资源不足的情况下使用的
- 2 一旦开启此配置后，整个MR将只会有一个reduce程序进行往外写出数据，减少资源占用情况，从而保证正常执行
- 3 弊端：执行效率变慢
- 4
- 5 如何配置呢？
- 6 `hive.optimize.sort.dynamic.partition=true` 直接在CM上对hive进行配置即可(暂时不要修改，后续遇到问题在修改)

CDH Test1

 Hive

操作 ▾

状态 实例 配置 命令 图表库 审核 HiveServer2 Web UI  快速链接 ▾

hive.optimize.sort.dynamic.partition

筛选器

▼ 范围

Hive (服务范围)

启用有序动态分区优化程序

hive.optimize.sort.dynamic.partition

☐ HiveServer2 Default Group

Hive分通表(分簇表)

- 作用

- 1 作用一：数据采样
- 2 第一种情况：假设数据源中数据非常庞大，需要对数据进行统计分析，此时需要先编写统计分析SQL，写完SQL后需要测试SQL是否可用，如果将SQL对整个表数据做测试，此时一条SQL测试时间会很长，这个时候，可以将整个表数据抽样出一部分来，然后测试SQL，此时效率比较高效
- 3 第二种情况：对数据结构校验操作(数据可行性校验)
- 4 第三种情况：在进行统计分析的时候，不需要大家计算出精确的结果，只需要计算出相对比例，此时也不需要全部数据进行分析，只需要抽样出一部分，进行计算统计即可
- 5
- 6 作用二：提升查询数据效率

- 分桶就是将数据根据某个字段分成不同的文件

- 如何创建分桶表

```
1 clustered by 字段名 into n buckets
2 # 将根据字段将文件分成多个文件（桶）
```

- 分桶类似于 MR 的分区
 - a. Hive中的分区就是分文件夹
 - b. Hive中的分桶就是分文件，底层就是MR的分区操作
 - 将 k2 进行hash，根据hash值求模 $\text{hashcode} \% \text{分区数(分桶个数)}$
 - 有几个分桶就有几个文件

从语法层面解析分桶含义

```
1 CLUSTERED BY xxx INTO N BUCKETS
2 --根据xxx字段把数据分成N桶
3 --根据表中的字段把数据文件成为N个部分
4
5 t_user(id int,name string);
6 --1、根据谁分？
7 CLUSTERED BY xxx ;    xxx必须是表中的字段
8 --2、分成几桶？
9 N BUCKETS    ; N的值就是分桶的个数4
10 --3、分桶的规则？
11 clustered by id into 3 bucket
12
13 hashfunc(分桶字段) % N bucket 余数相同的来到同一个桶中
14 1、如果分桶的字段是数字类型的字段，hashfunc(分桶字段)=分桶字段本身
15 2、如果分桶的字段是字符串或者其他字段，hashfunc(分桶字段) = 分桶字段.hashCode
```

分桶的创建

```
1 CREATE TABLE t_usa_covid19_bucket(
2     count_date string,
3     county string,
4     state string,
5     fips int,
6     cases int,
7     deaths int)
8 CLUSTERED BY(state) INTO 5 BUCKETS; --分桶的字段一定要是表中已经存在的字段
9
```

```

10 --根据state州分为5桶 每个桶内根据cases确诊病例数倒序排序
11 CREATE TABLE t_usa_covid19_bucket_sort(
12     count_date string,
13     county string,
14     state string,
15     fips int,
16     cases int,
17     deaths int)
18 CLUSTERED BY(state)
19 sorted by (cases desc) INTO 5 BUCKETS;--指定每个分桶内部根据 cases倒序排序

```

分桶表的数据加载

```

1 --step1:开启分桶的功能 从Hive2.0开始不再需要设置
2 set hive.enforce.bucketing=true;
3
4 -- 限制对桶表进行load操作
5 set hive.strict.checks.bucketing = true;
6 说明：开启此配置后，无法对桶表执行load 方式，一旦执行就会报错
7 --step2:把源数据加载到普通hive表中
8 CREATE TABLE t_usa_covid19(
9     count_date string,
10    county string,
11    state string,
12    fips int,
13    cases int,
14    deaths int)
15 row format delimited fields terminated by ",";
16
17 --将源数据上传到HDFS，t_usa_covid19表对应的路径下
18 hadoop fs -put us-covid19-counties.dat /user/hive/warehouse/itheima.db/t_usa_covid19
19
20 --step3:使用insert+select语法将数据加载到分桶表中
21 insert into t_usa_covid19_bucket select * from t_usa_covid19;
22
23 select * from t_usa_covid19_bucket;

```

分桶表的使用

```

1  --基于分桶字段state查询来自于New York州的数据
2  --不再需要进行全表扫描过滤
3  --根据分桶的规则hash_function(New York) mod 5计算出分桶编号
4  --查询指定分桶里面的数据 就可以找出结果 此时是分桶扫描而不是全表扫描
5  select *
6  from t_usa_covid19_bucket where state="New York";

```

分桶表采样

```

1  常规采样：不做分桶，也是可以进行采样：tablesample (N percent)
2  select * from tableName tablesample(N PERCENT);--按照文件大小的比例来进行采样
3  select * from temp_buck tablesample(10 PERCENT);
4
5  分桶采样
6  select ..... from tableName tablesample(bucket x out of y)
7  x: 采样的桶的编号
8  第一个桶: x
9      第二个桶: x+y
10     第三个桶: x+y+y
11     .....
12 y: 分桶因子
13     举个栗子
14     桶的个数为6: y = 1 2 3 6
15
16 例如: x=1 y=2
17 采样的桶的个数 = 桶的总个数 / 分桶因子 = 6 / 2 = 3个桶
18 第一个桶: 1
19 第二个桶: 3
20 第三个桶: 5
21 select * from test_buck tablesample(bucket 1 out of 3);
22
23
24 select * from table tablesample(bucket x out of y on column) a ;
25
26 抽样函数:
27     tablesample(bucket x out of y on column)
28 放置位置:
29     在SQL的表的后面，如果有别名，一定放置别名的前面
30 相关属性说明:

```

```

31    x: 从第几个桶开始抽样
32    y: 抽样比例, 此值必须桶数量的倍数或者因子
33 注意: x 不能大于 y
34 案例说明:
35    user表是一个分桶表, 共为分10个桶
36    select * from user tablesample(bucket 3 out of 10 on column);
37 请问上述SQL, 会抽取出几个桶, 以及抽的第几个桶?
38    抽取一个桶: 10 / 10 = 1
39    抽的第几个桶: 3    x=3
40
41    select * from user tablesample(bucket 2 out of 2 on column);
42 请问上述SQL, 会抽取出几个桶, 以及抽的第几个桶?
43    抽取 5个桶: 10 / y(2) = 5
44    会抽取那些桶:
45        2,4,6,8,10
46    select * from user tablesample(bucket 4 out of 5 on column);
47 请问上述SQL, 会抽取出几个桶, 以及抽的第几个桶?
48    抽取 2个桶 : 10 / y(5) = 2
49    会抽取那些桶:
50        4,9
51

```

Bucket Map Join: 分桶Join

- 第一种普通的分桶join: Bucket map Join
 - 语法: clustered by col into N buckets
 - 两张表必须为桶表, 并且桶的个数要相等或者成倍数
 - 分桶字段 = Join字段
- 第二种基于排序的分桶Join: Sort Merge Bucket Join => SMB Join
 - 语法: clustered by col sorted by col into N buckets
 - 两张表必须为桶表, 并且桶的个数要相等
 - 分桶字段 = Join字段 = 排序字段

```

1 开启map join
2 set hive.auto.convert.join=true;  -- 是否自动尝试map join (是否开启map join)
3 set hive.auto.convert.join.noconditionaltask.size=512000000 ;  -- 判断多小的表认为是小表
4    默认值为: 20971520 = 20M

```

- 中型表和大表进行join: bucket map join

```

1 1) set hive.optimize.bucketmapjoin = true;  -- 保证开启 bucket map join

```

- 2 2) 一个表的bucket数是另一个表bucket数的整数倍
- 3 3) bucket列 == join列
- 4 4) 必须是应用在map join的场景中
- 5 5) 表必须是分桶表: set hive.enforce.bucketing=true;

- 大表和大表进行join : SMB map join

```
1 1) 两个表必须都是桶表 : set hive.enforce.bucketing=true;
2 2) set hive.optimize.bucketmapjoin = true; -- 保证开启 bucket map join
3 3) 一个表的bucket数等于另一个表bucket数
4 4) bucket列 == join列 == sort列
5 5) 必须是应用在bucket map join的场景中
6
7 基于上述条件:
8 set hive.enforce.bucketing=true; -- 保证都是桶表支持
9 set hive.auto.convert.join=true; -- 开启mapjoin
10 set hive.auto.convert.join.noconditionaltask.size=512000000 ; -- 有默认值
11 set hive.optimize.bucketmapjoin = true; -- 开启 bucket mapjoin
12
13 set hive.enforce.sorting=true; -- 开启强制排序功能
14 set hive.auto.convert.sortmerge.join=true;
15 set hive.auto.convert.sortmerge.join.noconditionaltask=true; -- 开启 SMB join
16 set hive.optimize.bucketmapjoin.sortedmerge = true; -- 自动尝试SMB联接 (建议直接配置CM中)
17
18
19 一般再生产中, 建议大家都开启, 这样hive在执行过程中, 如果发现表可以走 SMBjoin 那么就会SMB join
    如果发现可以走 bucket map join 就会走 bucket 如果可以走 map join 那么就会走map join 否则走
    普通的reduce join
20
21
22 注意:
23 表创建时必须是CLUSTERED且SORTED
24
25 create table test_smb_2(mid string,age_id string)
26 CLUSTERED BY(mid) SORTED BY(mid) INTO 500 BUCKETS;
27
```

分桶的总结

- 分桶表也是一种优化表, 可以==减少join查询时笛卡尔积的数量==、==提高抽样查询的效率==。
- 分桶表的字段必须是表中已有的字段;

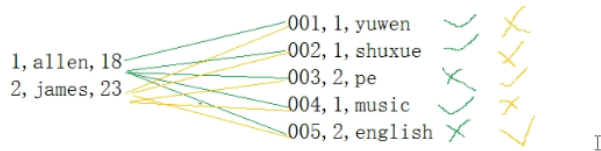
- 分桶表需要使用间接的方式才能把数据加载进入：insert+select
- 在join的时候，针对join的字段进行分桶，可以提高join的效率 减少笛卡尔积数量。

需求：查询出每个学生的选修课情况？

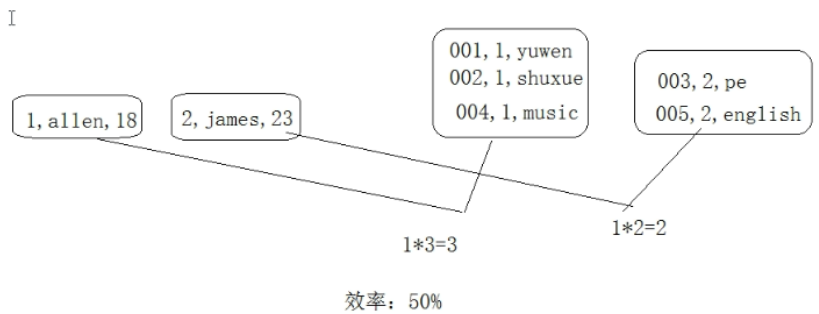
```
select * from t_a join t_b on t_a.id = t_b.snum
```

分桶表可以针对join场景进行优化。

关键：针对两边的表 对join的字段进行分桶操作。



2*5 = 10
符合条件的：5
效率：50%



效率：50%

Hive SQL--DDL其他操作

因为Hive建表、加载数据及其方便高效；在实际的应用中，==如果建表有问题，通常可以直接drop删除重新创建加载数据==。时间成本极低。

如果表是外部表的话，更加完美了。

Database 数据库 DDL操作

```
1  --创建数据库
2  create database if not exists itcast
3  comment "this is my first db"
4  with dbproperties ('createdBy'='Allen');
5
6  --描述数据库信息
7  describe database itcast;
8  describe database extended itcast;
9  desc database extended itcast;
10
11 --切换数据库
12 use default;
13 use itcast;
14 create table t_1(id int);
15
16 --删除数据库
```

```

17  --注意 CASCADE关键字慎重使用
18  DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];
19  drop database itcast cascade ;
20
21
22  --更改数据库属性
23  ALTER (DATABASE|SCHEMA) database_name SET DBPROPERTIES (property_name=property_value,
...);
24  --更改数据库所有者
25  ALTER (DATABASE|SCHEMA) database_name SET OWNER [USER|ROLE] user_or_role;
26  --更改数据库位置
27  ALTER (DATABASE|SCHEMA) database_name SET LOCATION hdfs_path;

```

Table 表 DDL操作

```

1  --下面这两个需要记住
2  --查询指定表的元数据信息
3  desc formatted itheima.t_all_hero_part;
4  show create table t_all_hero_part;
5
6  --1、更改表名
7  ALTER TABLE table_name RENAME TO new_table_name;
8  --2、更改表属性
9  ALTER TABLE table_name SET TBLPROPERTIES (property_name = property_value, ... );
10 --更改表注释
11 ALTER TABLE student SET TBLPROPERTIES ('comment' = "new comment for student table");
12 --3、更改SerDe属性
13 ALTER TABLE table_name SET SERDE serde_class_name [WITH SERDEPROPERTIES (property_name =
property_value, ... )];
14 ALTER TABLE table_name [PARTITION partition_spec] SET SERDEPROPERTIES serde_properties;
15 ALTER TABLE table_name SET SERDEPROPERTIES ('field.delim' = ',');
16 --移除SerDe属性
17 ALTER TABLE table_name [PARTITION partition_spec] UNSET SERDEPROPERTIES (property_name,
... );
18
19 --4、更改表的文件存储格式 该操作仅更改表元数据。现有数据的任何转换都必须在Hive之外进行。
20 ALTER TABLE table_name SET FILEFORMAT file_format;
21 --5、更改表的存储位置路径
22 ALTER TABLE table_name SET LOCATION "new location";
23

```

```

24 --6、更改列名称/类型/位置/注释
25 CREATE TABLE test_change (a int, b int, c int);
26 // First change column a's name to a1.
27 ALTER TABLE test_change CHANGE a a1 INT;
28 // Next change column a1's name to a2, its data type to string, and put it after column
  b.
29 ALTER TABLE test_change CHANGE a1 a2 STRING AFTER b;
30 // The new table's structure is:  b int, a2 string, c int.
31 // Then change column c's name to c1, and put it as the first column.
32 ALTER TABLE test_change CHANGE c c1 INT FIRST;
33 // The new table's structure is:  c1 int, b int, a2 string.
34 // Add a comment to column a1
35 ALTER TABLE test_change CHANGE a1 a1 INT COMMENT 'this is column a1';
36
37 --7、添加/替换列
38 --使用ADD COLUMNS, 您可以将新列添加到现有列的末尾但在分区列之前。
39 --REPLACE COLUMNS 将删除所有现有列, 并添加新的列集。
40 ALTER TABLE table_name ADD|REPLACE COLUMNS (col_name data_type,...);

```

Partition分区 DDL操作

比较重要的是==增加分区==、==删除分区==操作

```

1  --1、增加分区
2  --step1: 创建表 手动加载分区数据
3  drop table if exists t_user_province;
4  create table t_user_province (
5      num int,
6      name string,
7      sex string,
8      age int,
9      dept string) partitioned by (province string);
10
11 load data local inpath '/root/hivedata/students.txt' into table t_user_province
  partition(province ="SH");
12
13
14 --step2:手动创建分区的文件夹 且手动上传文件到分区中 绕开了hive操作 发现hive无法识别新分区
15 hadoop fs -mkdir /user/hive/warehouse/itheima.db/t_user_province/province=XM
16 hadoop fs -put students.txt /user/hive/warehouse/itheima.db/t_user_province/province=XM
17

```

```

18
19 --step3: 修改hive的分区, 添加一个分区元数据
20 ALTER TABLE t_user_province ADD PARTITION (province='XM') location
21     '/user/hive/warehouse/itheima.db/t_user_province/province=XM';
22
23
24 ----此外还支持一次添加多个分区
25 ALTER TABLE table_name ADD PARTITION (dt='2008-08-08', country='us') location
26     '/path/to/us/part080808'
27
28
29 --2、重命名分区
30 ALTER TABLE t_user_province PARTITION (province ="SH") RENAME TO PARTITION (province
31     ="Shanghai");
32
33 --3、删除分区
34 ALTER TABLE table_name DROP [IF EXISTS] PARTITION (dt='2008-08-08', country='us');
35
36 ALTER TABLE table_name DROP [IF EXISTS] PARTITION (dt='2008-08-08', country='us') PURGE;
37 --直接删除数据 不进垃圾桶 有点像skipTrash
38
39
40 --4、修复分区
41 MSCK [REPAIR] TABLE table_name [ADD/DROP/SYNC PARTITIONS];
42
43 --详细使用见课件资料
44
45
46 --5、修改分区
47 --更改分区文件存储格式
48 ALTER TABLE table_name PARTITION (dt='2008-08-09') SET FILEFORMAT file_format;
49
50 --更改分区位置
51 ALTER TABLE table_name PARTITION (dt='2008-08-09') SET LOCATION "new location";

```

Hive SQL中常见的show语法

```

show databases
show tables
show partitions
desc formatted table_name; 查看表的元数据信息
show create table table_name; 获取表的DDL建表语句
show functions;

```

1 --1、显示所有数据库 SCHEMAS和DATABASES的用法 功能一样

```
2  show databases;
3  show schemas;
4
5  --2、显示当前数据库所有表/视图/物化视图/分区/索引
6  show tables;
7  SHOW TABLES [IN database_name]; --指定某个数据库
8
9  --3、显示当前数据库下所有视图
10 --视图相当于没有数据临时表 虚拟表
11 Show Views;
12 SHOW VIEWS 'test_*'; -- show all views that start with "test_"
13 SHOW VIEWS FROM test1; -- show views from database test1
14 SHOW VIEWS [IN/FROM database_name];
15
16 --4、显示当前数据库下所有物化视图
17 SHOW MATERIALIZED VIEWS [IN/FROM database_name];
18
19 --5、显示表分区信息，分区按字母顺序列出，不是分区表执行该语句会报错
20 show partitions table_name;
21 show partitions itheima.student_partition;
22
23 --6、显示表/分区的扩展信息
24 SHOW TABLE EXTENDED [IN|FROM database_name] LIKE table_name;
25 show table extended like student;
26 describe formatted itheima.student;
27
28 --7、显示表的属性信息
29 SHOW TBLPROPERTIES table_name;
30 show tblproperties student;
31
32 --8、显示表、视图的创建语句
33 SHOW CREATE TABLE ([db_name.]table_name|view_name);
34 show create table student;
35
36 --9、显示表中的所有列，包括分区列。
37 SHOW COLUMNS (FROM|IN) table_name [(FROM|IN) db_name];
38 show columns in student;
39
40 --10、显示当前支持的所有自定义和内置的函数
41 show functions;
```

```

42
43 --11、Describe desc
44 --查看表信息
45 desc extended table_name;
46 --查看表信息（格式化美观）
47 desc formatted table_name;
48 --查看数据库相关信息
49 describe database database_name;

```

Hive DML 数据操纵语言

load

- 功能：load加载操作是将数据文件移动到与 Hive表对应的位置的==纯复制/移动==操作。

```

1 #1、如何理解这个纯字？
2 load命令是单纯数据搬运移到动作，移到加载的过程中，不会对数据进行任何修改操作。
3
4 #2、有的是移动操作，有的是复制操作

```

- 语法

```

1 LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO
2 TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

```

- Local关键字的含义
 - local表示是从本地文件系统加载数据到Hive的表中；
 - 如果没有local表示的是从HDFS文件系统加载数据到Hive表中；
 - ==问本地指的是哪个本地==？ HS2本地的文件系统
- 演示

```

1 --step1:建表
2 --建表student_local 用于演示从本地加载数据
3 create table student_local(num int,name string,sex string,age int,dept string) row
  format delimited fields terminated by ',';
4 --建表student_HDFS 用于演示从HDFS加载数据到非分区表
5 create table student_HDFS(num int,name string,sex string,age int,dept string) row
  format delimited fields terminated by ',';
6 --建表student_HDFS_p 用于演示从HDFS加载数据到分区表
7 create table student_HDFS_p(num int,name string,sex string,age int,dept string)
  partitioned by(country string) row format delimited fields terminated by ',';
8
9 --step2:加载数据

```

```

10 -- 从本地加载数据 数据位于HS2 (node1) 本地文件系统 本质是hadoop fs -put上传操作
11 --Loading data to table itheima.student_local from file:/root/hivedata/students.txt
12 LOAD DATA LOCAL INPATH '/root/hivedata/students.txt' INTO TABLE student_local;
13
14 --从HDFS加载数据 数据位于HDFS文件系统根目录下 本质是hadoop fs -mv 移动操作
15 --Loading data to table itheima.student_hdfs from hdfs://node1:8020/stu/students.txt
16
17 --先把数据上传到HDFS上 hadoop fs -put /root/hivedata/students.txt /
18 LOAD DATA INPATH '/students.txt' INTO TABLE student_HDFS;
19
20 ----从HDFS加载数据到分区表中并指定分区 数据位于HDFS文件系统根目录下
21 --先把数据上传到HDFS上 hadoop fs -put /root/hivedata/students.txt /
22 LOAD DATA INPATH '/students.txt' INTO TABLE student_HDFS_p partition(country ="China");

```

- 总结: ==建议在Hive中使用load命令加载数据到表中==, 这也是Hive比Mysql加载数据高效地方所在。
- 其实不管是使用load还是直接使用hadoop fs,目标就是把文件放置在表对应的目录下。

insert

- 探讨

```

1 --如果使用操作Mysql的思维来玩Hive会如何 使用insert+values的方式插入数据。
2
3 create table t_insert(id int,name string);
4
5 insert into table t_insert values(1,"allen");
6
7 --可以执行 但是效率极低 因为底层是通过MapReduce插入数据的 因此实际中推荐使用load加载数据

```

insert+select

==在hive中, insert主要是结合 select 查询语句使用, 将查询结果插入到表中==。

- 保证后面select**查询语句返回的结果字段个数、类型、顺序和待插入表一致**;
- 如果不一致, Hive会尝试帮你转换, 但是不保证成功;
- insert+select也是在数仓中ETL数据常见的操作。

```

1 --step1:创建一张源表student
2 drop table if exists student;
3 create table student(num int,name string,sex string,age int,dept string)
4 row format delimited
5 fields terminated by ',';
6 --加载数据

```

```

7 load data local inpath '/root/hivedata/students.txt' into table student;
8
9 select * from student;
10
11 --step2: 创建一张目标表 只有两个字段
12 create table student_from_insert(sno int,sname string);
13 --使用insert+select插入数据到新表中
14 insert into table student_from_insert
15 select num,name from student;
16
17 select *
18 from student_from_insert;

```

Multi Inserts 多重插入

- 功能: ==一次扫描, 多次插入==

```

1 create table source_table (id int, name string) row format delimited fields terminated
  by ',';
2
3 create table test_insert1 (id int) row format delimited fields terminated by ',';
4 create table test_insert2 (name string) row format delimited fields terminated by ',';
5
6
7 --普通插入:
8 insert into table test_insert1 select id from source_table;
9 insert into table test_insert2 select name from source_table;
10
11 --在上述需求实现中 从同一张表扫描了2次 分别插入不同的目标表中 性能低下。
12
13 --多重插入:
14 from source_table
15 insert overwrite table test_insert1
16 select id
17 insert overwrite table test_insert2
18 select name;
19 --只需要扫描一次表 分别把不同字段插入到不同的表中即可 减少扫描次数 提高效率

```

Dynamic partition inserts 动态分区插入

- 何谓动态分区, 静态分区


```
1 针对的是分区表。
2  --问题：分区表中分区字段值是如何确定的？
3
4 1、如果是在加载数据的时候人手动写死指定的 叫做静态分区
5 load data local inpath '/root/hivedata/usa_dezhou.txt' into table t_user_double_p
   partition(guojia="meiguo",sheng="dezhou");
6
7 2、如果是通过insert+select 动态确定分区值的，叫做动态分区
8 insert table partition (分区字段) +select
```

```
1  --1、首先设置动态分区模式为非严格模式 默认已经开启了动态分区功能
2 set hive.exec.dynamic.partition = true;
3 set hive.exec.dynamic.partition.mode = nonstrict;
4
5  --2、当前库下已有一张表student
6 select * from student;
7
8  --3、创建分区表 以sdept作为分区字段
9 create table student_partition(Sno int,Sname string,Sex string,Sage int) partitioned
   by(Sdept string);
10
11  --4、执行动态分区插入操作
12 insert into table student_partition partition(Sdept)
13 select num,name,sex,age,dept from student;
14  --其中，num,name,sex,age作为表的字段内容插入表中
15  --dept作为分区字段值
16
17 select *
18 from student_partition;
19
20 show partitions student_partition;
```

导出数据操作

- 功能：把select查询的结果导出成为一个文件。
- 注意：**==导出操作是一个overwrite操作==**，可能会让你凉凉。慎重！！！！
- 语法

```
1  --当前库下已有一张表student
```

```

2 select * from student;
3
4 --1、导出查询结果到HDFS指定目录下
5 insert overwrite directory '/tmp/hive_export/e1' select num,name,age from student limit
6 2; --默认导出数据字段之间的分隔符是\001
7
8 --2、导出时指定分隔符和文件存储格式
9 insert overwrite directory '/tmp/hive_export/e2' row format delimited fields terminated
10 by ','
11 stored as orc
12 select * from student;
13
14 --3、导出数据到本地文件系统指定目录下
15 insert overwrite local directory '/root/hive_export/e1' select * from student;

```

- Hive 导入的数据能不能是视频文件
- 不能解析视频、音频非结构化数据

Hive DQL数据查询语言

cluster by分桶查询

```

1 select * from student cluster by num ; -- 分桶查询 根据编号进行分桶查询
2 #cluster by 分桶字段,根据分桶字段分成几个部分
3 --如果用户没有设置,不指定reduce task个数,则hive该局输入数据量自己评估
4 -- 日志显示:Number of reduce tasks not specified. Estimated from input data size: 1
5 -- 手动设置reduce 个数
6 set mapreduce.job.reduces =2;
7 -- 日志显示:Number of reduce tasks not specified. Defaulting to jobconf value of: 2
8 -- 分桶查询的结果真的根据reduce tasks个数分为了两个部分,并且每个部分还根据了则断进行了排序
9 -- 总结:cluster by xx 分组排序的功能,分为几个部分,取决于reducetask个数,排序只能是正序,用户无法改变
10
11 -- 需求:把student表数据该局num分为两个部分,每个部分中根据年龄age倒序排序
12 set mapreduce.job.reduces=2;
13 select * from student cluster by num order by age desc; 报错
14 -- 另一种
15 select * from student cluster by num sort age desc ;报错
16 -- 总结
17 1.cluster by 字段 和distribute 字段 sort by 分区排序字段是或的关系
18 2.cluster by 字段 = distribute by 字段 sort by 字段。distribute by 和 sort by 字段相同时候
   等价于 cluster by

```

distribute by+sort by

- 功能:相当于把cluster by的功能一分为二
 - distribute by只负责分
 - sort by 只负责分之后的每个部分排序
 - 并且分和排序的字段可以不一样

```
1 -- 当后面分和排序的字段是同一个字段,加起来就相等cluster by
2 cluster by(分且排序) = distribute by (分) + sort by (排序)
3 -- 下面两个功能一样的
4 select * from student cluster by num;
5 select * from student distribute by num sort by num;
6
7 -- 最终实现
8 select * from student distribute by num sort by age desc;
```

order by

```
1 -- 首先设置一下reducetask个数,随便设置
2 set mapreduce.job.reduce=2;
3 -- 日志中显示的还是1
4 Number of reduce tasks determined at compile time: 1
5 --原因: order by是全局排序.全局排序意味着数据只能输出在一个文件中.因此也只能有一个reducetask
6 -- 在order by 出现的情况下,不管用户设置几个reducetask,在编译执行期间都会变为一个,满足全局
```

- order by 和sort by
 - order by 负责 全局排序 意味着整个mr作业只有一个reducetask 不管用户设置几个 编译期间hive都会把它设置为1
 - sort by 负责分完之后,局部排序

完整版select语法树

```
1 [WITH CommonTableExpression (, CommonTableExpression)*]
2 SELECT [ALL | DISTINCT] select_expr, select_expr, ...
3 FROM table_reference
4 [WHERE where_condition]
5 [GROUP BY col_list]
6 [ORDER BY col_list]
7 [CLUSTER BY col_list]
```

```
8      | [DISTRIBUTE BY col_list] [SORT BY col_list]
9  ]
10 [LIMIT [offset,] rows];
```

union联合查询

UNION用于将来自==多个SELECT语句的结果合并为一个结果集==。

```
1  -- 语法规则
2  select_statement union [distinct|all] select_statement union [all|distinct]
   select_statement
3  -- 使用distinct关键字与使用union默认值效果一样,都会删除重复行
4  select num,name from student_local
5  union
6  select num,name from student_hdfs;
7
8  -- 和上面一样
9  select num,name from student_local
10 union distinct
11 select num,name from student_hdfs limit 2;
12
13 -- 使用all关键字会保留重复行
14 select num,name from student_local
15 union all
16 select num,name from student_hdfs limit 2;
17
18 -- 如果要将order by ,sort by ,cluster by , distribute by 或者 limit 应用于单个select
19 -- 请将子句放在括住select的括号内
20 select num,name from(select num,name from student_local limit 2) subq1
21 union
22 select num,name from ( select num,name from student_hdfs limit 3)subq2;
23
24 -- 如果要将order by , sort by ,cluster by ,distribute by 或limit 子句应用于整个union结果
25 -- 请将order by ,sort by ,cluster by ,distribute by 或limit 放在最后一个之后
26 select num,name from student_local
27 union
28 select num,name from student_hdfs
29 order by num desc;
30
```

CTE表达式

通用表达式(CTE)是一个临时结果集,该结果集是从 with 子句中指定的简单查询 派生而来的 ,该查询紧接在select 或 insert 关键字之前.--

```
1  -- select语句中的CTE
2  with q1 as (select num,name,age from student where num=95002)
3  select * from q1;
4
5  -- from风格
6  with q1 as(select num,name,age from student where num=95002)
7  from q1 select *;
8
9  -- chaining CTEs链式
10 with q1 as (select * from student where num=95002),
11     q2 as (select num,name,age from q1)
12 select * from (select num from q2) a ;
13
14 -- union
15 with q1 as (select * from student where num=95002),
16     q2 as (select * from student where num=95004)
17 select * from q1 union all select * from q2 ;
18
19 -- ctas
20 -- create table as select 创建一张表来自于后面的查询语句,表的字段个数,名字,顺序和数据行数都取决于查询
21 -- create table t_ctas as select num,name from student limit 2;
22 create table s2 as with q1 as (select * from student where num=95002)
23 select * from q1;
24
25 -- view
26 create view v1 as with q1 as
27 (select * from student where num=95002)
28 select * from q1;
29
30 select * from v1;
```

Hive SQL join 查询

```

1  语法树
2  join_table:
3      table_reference [INNER] JOIN table_factor [join_condition]
4      | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference join_condition
5      | table_reference LEFT SEMI JOIN table_reference join_condition
6      | table_reference CROSS JOIN table_reference [join_condition] (as of Hive 0.10)
7
8  join_condition:
9      ON expression

```

```

1  -- join语法练习 建表
2  drop table if exists employee_address;
3  drop table if exists employee_connection;
4  drop table if exists employee;
5  --table1: 员工表
6  CREATE TABLE employee(
7      id int,
8      name string,
9      deg string,
10     salary int,
11     dept string
12 ) row format delimited
13 fields terminated by ',';
14
15 --table2:员工家庭住址信息表
16 CREATE TABLE employee_address (
17     id int,
18     hno string,
19     street string,
20     city string
21 ) row format delimited
22 fields terminated by ',';
23
24 --table3:员工联系方式信息表
25 CREATE TABLE employee_connection (
26     id int,
27     phno string,
28     email string

```

```

29 ) row format delimited
30 fields terminated by ',';
31 --加载数据到表中
32 load data local inpath '/root/data/employee.txt' into table employee;
33 load data local inpath '/root/data/employee_address.txt' into table employee_address;
34 load data local inpath '/root/data/employee_connection.txt' into table
employee_connection;
35 select * from employee;
36 select * from employee_address;
37 select * from employee_connection;

```

```

1 -- 1、内连接 inner join==join
2 -- 返回左右两边同时满足条件的数据
3 select e.*,e_a.* from employee e inner join employee_address e_a on e.id = e_a.id;
4 -- 等价于 inner join
5 select e.*,e_a.* from employee e join employee_address e_a on e.id=e_a.id;
6
7 -- 2、左连接 left join == left outer join
8 -- 左表为准,左表全部显示,右表与之关联,满足条件的返回,不满足条件显示null
9 select e.*,e_conn.* from employee e left join employee_connection e_conn on e.id =
e_conn.id;
10 -- 等价于 left outer join左外连接
11 select e.id,e.*,e_conn.* from employee e left outer join employee_connection e_conn on
e.id =e_conn.id;
12 -- 3.右连接 right join == right outer join 右外连接
13 -- 右表为准,右表全部显示,左表与之关联,满足条件的返回,不满足条件显示null
14 select e.id,e.*,e_conn.* from employee e right join employee_connection e_conn on e.id
=e_conn.id ;
15 -- 等价于right outer join
16 select e.id,e.*,e_conn. from employee e right outer join employee_connection e_conn on
e.id = e_conn.id ;
17
18 -- 4、外连接 全外连接full join== full outer join
19 full outer join关键字只要左表(table1)和右表(table2)其中一个表中存在匹配,则返回行
20 full outer join关键字结合了 left join 和right join 的结果
21 select e.*,e_a.* from employee e full outer join employee_address e_a on e.id=e_a.id;
22 -- 等价于
23 select e.*,e_a.* from employee e full join employee_address e_a on e.id=e_a.id;
24
25 -- 5、左半连接 left semi join

```

```

26 select * from employee e left semi join employee_address e_addr on e.id =e_addr.id;
27 -- 相当于inner join 但是只返回左表全部数据,只不过效率高一些,高于in/exists这种条件查询
28 select e.* from employee e inner join employee_address e_addr on e.id = e_addr.id;
29
30 -- 6、交叉连接 cross join
31 将会返回被连接的两个表的笛卡尔积,返回结果的行数等于表行数的乘积。
32 对于大表来说,cross join慎用
33 在SQL标准中定义的cross join就是无条件的inner join 返回两个表的笛卡尔积,无需指定关键字
34 在hiveSQL语法中,cross join 后面可以跟where子句进行过滤,或者on条件过滤。

```

join查询优化及注意事项

- 允许使用复杂的连接表达式
- 同一查询中可以连接两个以上的表
- 如果每个表在连接子句中使用相同的列,则hive将多个表上的连接转换为单个mr作业
- join时的最后一个表会通过reduce流式传输,并在其中缓冲之前的其他表,因此,将大表放置在最后有助于减少reduce阶段缓存数据所需要的内存

Hive shell命令行

- 批处理:一次连接,一次交互,执行结束断开连接
- 交互式处理:保存持续连接,一直交互
- 注意:如果说hive的shell客户端指的是第一代客户端bin/hive 而第二代客户端bin/beeline 属于jdbc客户端 不是shell

bin/hive

- 功能1: 作为第一代客户端连接访问 metastore 服务,使用hive,交互式方式
- 功能2:启动hive服务

```

1 /export/server/apache-hive/bin/hive --server metastore
2 /export/server/apache-hive/bin/hive --server hiveserver2

```

- 功能3:批处理执行hiveSQL

```

1 #-e执行后面的SQL语句 -S 静默执行
2 /export/server/apache-hive/bin/hive -e 'select * from day06.student'
3 /export/server/apache-hive/bin/hive -S -e 'select * from day06.student'
4
5 #-f 执行后面的SQL文件
6 vim hive.sql
7 select * from day06.student limit 2;
8 /export/server/apache-hive/bin/hive -f hive.sql

```


- 9 #SQL文件不一定是.SQL 要保证文件中是正确的hql语法
- 10 #-f调用SQL文件执行的方式 是企业中hive生产环境主流的调用方式

Hive参数配置方式

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

- 配置方式
 - 方式1:配置文件 conf/hive-site.xml
 - 方式2: 配置参数 hiveconf

```
1 /export/server/apache-hive/bin/hive --server metastore
2 /export/server/apache-hive/bin/hive --server hiveserver2 --hiveconf
3 hive.root.logger=DEBUG,console
4 #影响的是session会话级别的
```

- 方式3: set 命令

```
1 session会话级别的 设置完之后将会对后面的sql执行生效
2 session结束 set设置的参数将失效
3 也是推荐搭建使用的设置参数方式,谁需要 谁设置 谁生效
```

Hive的函数高阶应用

explode函数

- explode属于UDTF函数,表生成函数,输入一行数据输出多行数据

```
1 explode() takes in an array (or a map) as an input and outputs the elements of
2 the array (map) as sparate rows.
3 -- explode接收map array类型的参数 把map或者array的元素输出,一行一个元素
4 explode(array(11,22,33))
5 select explode (array(11,22,33,44,55));
6 select explode(`map`("id",10086,"name","allen","age",18));
```

例子:

- 将NBA总冠军球队数据使用explode进行拆分,并且根据夺冠年份进行倒序排序

```
1 -- step1 :建表
2 create table the_nba_championship(
3     team_name string,
4     champion_year array<string>
5 )row format delimited fields terminate by ','
```

```

6 collection items terminated by '|';
7 -- step2 : 加载数据文件列表中
8 load data local inpath '/root/data/The_NBA_Championship.txt' into table
  the_nba_championship;
9 -- step3 :验证
10 select * from the_nba_championship;
11
12 -- step4:使用explode 函数对champion进行拆分,俗称炸开
13 select explode(champion_year) from the_nba_championship;
14 -- 想法是正确的SQL执行确是错误的
15 select team_name,explode(champion_year) from the_nba_championship;
16 -- 错误信息
17 UDTF's are not supported outside the SELECT clause, nor nested in expressions
18 UDTF 在 SELECT 子句之外不受支持,也不在表达式中嵌套???
19

```

- 如果数据不是map或者array如何使用explode函数呢？
 - 想方设法使用split substr regex_replace等函数组合使用把数据变成array或者map

```

1 create table the_nba_championship_str(
2     team_name string,
3     champion_year string
4 )row format delimited fields terminated by ',';
5
6 load data local inpath '/root/data/The_NBA_Championship.txt' into table
  the_nba_championship_str;

```

1、explode函数属于UDTF函数，即表生成函数；

- explode函数执行返回的结果可以理解为一张虚拟的表，其数据来源于源表；
- 在select中只查询源表数据没有问题，只查询explode生成的虚拟表数据也没问题
- 但是不能在只查询源表的时候，既想返回源表字段又想返回explode生成的虚拟表字段
- 通俗点讲，有两张表，不能只查询一张表但是返回分别属于两张表的字段；
- 从SQL层面上来说应该对两张表进行关联查询
- Hive专门提供了语法lateral View侧视图，专门用于搭配explode这样的UDTF函数，以满足上述需要。

lateral view 侧视图

侧视图的原理是==将UDTF的结果构建成一个类似于视图的表，然后将原表中的每一行和UDTF函数输出的每一行进行连接，生成一张新的虚拟表==

- 背景

- UDTF函数生成的结果可以当成一张虚拟的表，但是无法和原始表进行组合查询
- 在select条件中，如果只有explode函数表达式，程序执行是没有任何问题的；
 - 但是在select条件中，包含explode和其他字段，就会报错。

```
1 select name,explode(location) from test_message;--这个sql就是错误的 相当于执行组合查询
```

- 从理论层面推导，对两份数据进行join就可以了
- 但是，hive专门推出了lateral view侧视图的语，满足上述需要。
- 功能：==把UDTF函数生成的结果和原始表进行关联，便于用户在select时间组合查询==、lateral view是UDTF的好基友好搭档，实际中经常配合使用。
- 语法：

```
1 --lateral view侧视图基本语法如下
2 select ..... from tabelA lateral view UDTF(xxx) 别名 as col1,col2,col3.....;
3
4 --针对上述NBA冠军球队年份排名案例，使用explode函数+lateral view侧视图，可以完美解决
5 select a.team_name ,b.year
6 from the_nba_championship a lateral view explode(champion_year) b as year;
7
8 --根据年份倒序排序
9 select a.team_name ,b.year
10 from the_nba_championship a lateral view explode(champion_year) b as year
11 order by b.year desc;
12
13 --统计每个球队获取总冠军的次数 并且根据倒序排序
14 select a.team_name ,count(*) as nums
15 from the_nba_championship a lateral view explode(champion_year) b as year
16 group by a.team_name
17 order by nums desc;
```

行列转换

多行转单列

- ==数据收集函数==

```
1 collect_set --把多行数据收集为一行 返回set集合 去重无序
2 collect_list --把多行数据收集为一行 返回list集合 不去重有序
```

- 字符串拼接函数

```
1 concat --直接拼接字符串
```

```

2 concat_ws --指定分隔符拼接
3
4 select concat("it","cast","And","heima");
5 select concat("it","cast","And",null);
6
7 select concat_ws("-", "itcast","And","heima");
8 select concat_ws("-", "itcast","And",null);

```

- 栗子

```

1 --原表
2 +-----+-----+-----+---+
3 | row2col2.col1 | row2col2.col2 | row2col2.col3 |
4 +-----+-----+-----+---+
5 | a           | b           | 1           |
6 | a           | b           | 2           |
7 | a           | b           | 3           |
8 | c           | d           | 4           |
9 | c           | d           | 5           |
10 | c           | d           | 6           |
11 +-----+-----+-----+---+
12
13 --目标表
14 +-----+-----+-----+---+
15 | col1 | col2 | col3 |
16 +-----+-----+-----+---+
17 | a    | b    | 1-2-3 |
18 | c    | d    | 4-5-6 |
19 +-----+-----+-----+---+
20
21 --建表
22 create table row2col2(
23             col1 string,
24             col2 string,
25             col3 int
26 )row format delimited fields terminated by '\t';
27
28 --加载数据到表中
29 load data local inpath '/root/data/r2c2.txt' into table row2col2;
30 select * from row2col2;
31

```

```

32 --最终SQL实现
33 select
34     col1,
35     col2,
36     concat_ws(',', collect_list(cast(col3 as string))) as col3
37 from
38     row2col2
39 group by
40     col1, col2;

```

单列转多行

- 技术原理: explode+lateral view
- 例子

```

1  --原表
2  +-----+-----+-----+---+
3  | col1  | col2  | col3  |
4  +-----+-----+-----+---+
5  | a      | b      | 1,2,3  |
6  | c      | d      | 4,5,6  |
7  +-----+-----+-----+---+
8
9  --目标表
10 +-----+-----+-----+---+
11 | row2col2.col1 | row2col2.col2 | row2col2.col3 |
12 +-----+-----+-----+---+
13 | a              | b              | 1              |
14 | a              | b              | 2              |
15 | a              | b              | 3              |
16 | c              | d              | 4              |
17 | c              | d              | 5              |
18 | c              | d              | 6              |
19 +-----+-----+-----+---+
20
21 --创建表
22 create table col2row2(
23     col1 string,
24     col2 string,
25     col3 string
26 )row format delimited fields terminated by '\t';

```

```

27
28 --加载数据
29 load data local inpath '/root/data/c2r2.txt' into table col2row2;
30
31 select * from col2row2;
32
33 select explode(split(col3,',')) from col2row2;
34
35 --SQL最终实现
36 select
37     col1,
38     col2,
39     lv.col3 as col3
40 from
41     col2row2
42     lateral view
43         explode(split(col3, ',')) lv as col3;

```

json格式数据处理

- 在hive中，没有json类的存在，一般使用string类型来修饰，叫做json字符串，简称json串。
- 在hive中，处理json数据的两种方式
 - hive内置了两个用于解析json的函数

```

1 json_tuple
2 --是UDTF 表生成函数 输入一行，输出多行 一次提取读个值 可以单独使用 也可以配合lateral view侧
  视图使用
3
4 get_json_object
5 --是UDF普通函数，输入一行 输出一行 一次只能提取一个值 多次提取多次使用

```

- 使用JsonSerDe 类解析，在加载json数据到表中的时候完成解析动作
- 栗子

```

1 --创建表
2 create table tb_json_test1 (
3     json string
4 );
5
6 --加载数据
7 load data local inpath '/root/data/device.json' into table tb_json_test1;

```

```

8
9 select * from tb_json_test1;
10
11 -- get_json_object UDF函数 最大弊端是一次只能解析提取一个字段
12 select
13     --获取设备名称
14     get_json_object(json,"$.device") as device,
15     --获取设备类型
16     get_json_object(json,"$.deviceType") as deviceType,
17     --获取设备信号强度
18     get_json_object(json,"$.signal") as signal,
19     --获取时间
20     get_json_object(json,"$.time") as stime
21 from tb_json_test1;
22
23 --json_tuple 这是一个UDTF函数 可以一次解析提取多个字段
24 --单独使用 解析所有字段
25 select
26     json_tuple(json,"device","deviceType","signal","time") as
    (device,deviceType,signal,stime)
27 from tb_json_test1;
28
29 --搭配侧视图使用
30 select
31     json,device,deviceType,signal,stime
32 from tb_json_test1
33     lateral view json_tuple(json,"device","deviceType","signal","time") b
34     as device,deviceType,signal,stime;
35
36
37 --方式2: 使用JsonSerDe类在建表的时候解析数据
38 --建表的时候直接使用JsonSerDe解析
39 create table tb_json_test2 (
40     device string,
41     deviceType string,
42     signal double,
43     `time` string
44 )
45 ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
46 STORED AS TEXTFILE;

```

```

47
48 load data local inpath '/root/data/device.json' into table tb_json_test2;
49
50 select * from tb_json_test2;

```

- 总结：
 - a. Hive读取 json 字符串分为两类：
 - b. 内置两种函数
 - json_tuple 一对多，给个 json 返回多列
 - get_json_object 一对一，输入一个 json 返回对应的json的对象集合()
 - c. 创建表的时候指定SerDe 序列化和反序列方式 JsonSerDe 而不是 Delimited 或者 LazySimpleSerde

hive 窗口函数

- 回顾

```

1  分析函数() over ([partition by] 分组字段 [order by] 分组内排序字段 窗口大小)
2  分析函数分类：
3  max/min sum/count/avg ntile lead/lag first_value/last_value等
4  排名：
5  rank dense_rank row_number
6  窗口大小：
7  rows between ... and ....
8  unbounded preceding
9  unbounded following
10 current row
11 n preceding
12 n following

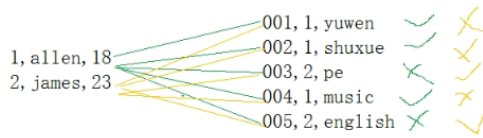
```

快速理解窗口函数功能

- window function 窗口函数、开窗函数、olap分析函数。
- 窗口：可以理解为操作数据的范围，窗口有大有小，本窗口中操作的数据有多有少。
- 可以简单地解释为类似于聚合函数的计算函数，但是通过GROUP BY子句组合的常规聚合会隐藏正在聚合的各个行，最终输出一行；而==窗口函数聚合后还可以访问当中====的各个行，并且可以将这些行中的某些属性添加到结果集中==。
-

需求：查询出每个学生的选修课情况？

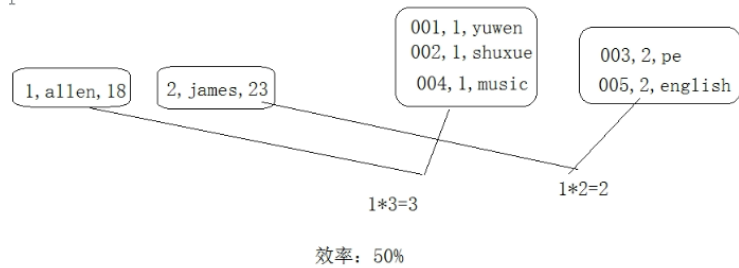
```
select * from t_a join t_b on t_a.id = t_b.snum
```



2*5 = 10
符合条件的：5
效率：50%

分桶表可以针对join场景进行优化。

关键：针对两边的表 对join的字段进行分桶操作。



```
1  --建表加载数据
2  CREATE TABLE employee(
3      id int,
4      name string,
5      deg string,
6      salary int,
7      dept string
8  ) row format delimited
9      fields terminated by ',';
10
11  load data local inpath '/root/data/employee.txt' into table employee;
12
13  select * from employee;
14
15  ----sum+group by普通常规聚合操作-----
16  select dept,sum(salary) as total from employee group by dept;
17
18  select id,dept,sum(salary) as total from employee group by dept; --添加id至结果, 错误sql
19
20  +-----+-----+
21  | dept | total |
22  +-----+-----+
23  | AC   | 60000 |
24  | TP   | 120000 |
25  +-----+-----+
26
```

```

27 ----sum+窗口函数聚合操作-----
28 select id,name,deg,salary,dept,sum(salary) over(partition by dept) as total from
   employee;
29
30 +-----+-----+-----+-----+-----+-----+
31 |  id   | name   | deg   | salary | dept  | total |
32 +-----+-----+-----+-----+-----+-----+
33 | 1204  | prasanth | dev   | 30000  | AC    | 60000 |
34 | 1203  | khalil  | dev   | 30000  | AC    | 60000 |
35 | 1206  | kranthi | admin | 20000  | TP    | 120000 |
36 | 1202  | manisha | cto   | 50000  | TP    | 120000 |
37 | 1201  | gopal   | manager | 50000  | TP    | 120000 |
38 +-----+-----+-----+-----+-----+-----+

```

窗口函数语法

具有==OVER语句==的函数叫做窗口函数。

```

1 Function(arg1,..., argn) OVER ([PARTITION BY <...>] [ORDER BY <....>]
  [<window_expression>])
2
3 --其中Function(arg1,..., argn) 可以是下面分类中的任意一个
4     --聚合函数: 比如sum max avg等
5     --排序函数: 比如rank row_number等
6     --分析函数: 比如lead lag first_value等
7
8 --OVER [PARTITION BY <...>] 类似于group by 用于指定分组 每个分组你可以把它叫做窗口
9 --如果没有PARTITION BY 那么整张表的所有行就是一组
10
11 --[ORDER BY <....>] 用于指定每个分组内的数据排序规则 支持ASC、DESC
12
13 --[<window_expression>] 用于指定每个窗口中 操作的数据范围 默认是窗口中所有行

```

- 建表加载数据 后续练习使用

```

1 ---建表并且加载数据
2 create table website_pv_info(
3     cookieid string,
4     createtime string,  --day
5     pv int
6 ) row format delimited
7 fields terminated by ',';

```

```

8
9 create table website_url_info (
10     cookieid string,
11     createtime string, --访问时间
12     url string        --访问页面
13 ) row format delimited
14 fields terminated by ',';
15
16
17 load data local inpath '/root/data/website_pv_info.txt' into table website_pv_info;
18 load data local inpath '/root/data/website_url_info.txt' into table website_url_info;
19
20 select * from website_pv_info;
21 select * from website_url_info;

```

窗口聚合函数

- 语法

```
1 sum|max|min|avg OVER ([PARTITION BY <...>] [ORDER BY <....>] [<window_expression>])
```

- 重点：==有PARTITION BY 没有PARTITION BY的区别；有ORDER BY没有ORDER BY的区别==。
 - 有没有partition by 影响的是全局聚合 还是分组之后 每个组内聚合。
 - 有没有==order by的区别==：
 - 没有order by,聚合的时候是组内所有的数据聚合再一起 全局聚合
 - 如果有order by, 聚合的时候是累加聚合，默认是第一行聚合到当前行。
- 栗子

```

1 --1、求出每个用户总pv数 sum+group by普通常规聚合操作
2 select cookieid,sum(pv) as total_pv from website_pv_info group by cookieid;
3 +-----+-----+
4 | cookieid | total_pv |
5 +-----+-----+
6 | cookie1  | 26       |
7 | cookie2  | 35       |
8 +-----+-----+
9
10 --2、sum+窗口函数 总共有四种用法 注意是整体聚合 还是累积聚合
11 --sum(...) over( )对表所有行求和，可以带上不相干的字段，与 group by ... sum
12 --sum(...) over( order by ... ) 连续累积求和
13 --sum(...) over( partition by... ) 同组内所行求和

```

```

14 --sum(...) over( partition by... order by ... ) 在每个分组内，连续累积求和
15
16 --需求： 求出网站总的pv数 所有用户所有访问加起来
17 --sum(...) over( )对表所有行求和
18 select cookieid,createtime,pv,
19         sum(pv) over() as total_pv
20 from website_pv_info;
21
22 --需求： 求出每个用户总pv数
23 --sum(...) over( partition by... ), 同组内所行求和
24 select cookieid,createtime,pv,
25         sum(pv) over(partition by cookieid) as total_pv
26 from website_pv_info;
27
28 --需求： 求出每个用户截止到当天，累积的总pv数
29 --sum(...) over( partition by... order by ... ), 在每个分组内，连续累积求和
30 select cookieid,createtime,pv,
31         sum(pv) over(partition by cookieid order by createtime) as current_total_pv
32 from website_pv_info;
33 +-----+-----+-----+-----+
34 | cookieid | createtime | pv | current_total_pv |
35 +-----+-----+-----+-----+
36 | cookie1  | 2018-04-10 | 1  | 1                |
37 | cookie1  | 2018-04-11 | 5  | 6                |
38 | cookie1  | 2018-04-12 | 7  | 13               |
39 | cookie1  | 2018-04-13 | 3  | 16               |
40 | cookie1  | 2018-04-14 | 2  | 18               |
41 | cookie1  | 2018-04-15 | 4  | 22               |
42 | cookie1  | 2018-04-16 | 4  | 26               |
43 | cookie2  | 2018-04-10 | 2  | 2                |
44 | cookie2  | 2018-04-11 | 3  | 5                |
45 | cookie2  | 2018-04-12 | 5  | 10               |
46 | cookie2  | 2018-04-13 | 6  | 16               |
47 | cookie2  | 2018-04-14 | 3  | 19               |
48 | cookie2  | 2018-04-15 | 9  | 28               |
49 | cookie2  | 2018-04-16 | 7  | 35               |
50 +-----

```

cookieid	createtime	pv	pv5
cookie1	2018-04-10	1	6
cookie1	2018-04-11	5	13
cookie1	2018-04-12	7	16
cookie1	2018-04-13	3	18
cookie1	2018-04-14	2	21
cookie1	2018-04-15	4	20
cookie1	2018-04-16	4	13
cookie2	2018-04-10	2	5
cookie2	2018-04-11	3	10
cookie2	2018-04-12	5	16
cookie2	2018-04-13	6	19
cookie2	2018-04-14	3	26
cookie2	2018-04-15	9	30
cookie2	2018-04-16	7	25

51

•

window_expression

直译叫做window表达式，通俗叫法称之为window子句。

- 功能：控制窗口操作的范围。
- 语法

```

1 rows between
2     - preceding: 往前
3     - following: 往后
4     - current row: 当前行
5     - unbounded: 起点
6     - unbounded preceding 表示从前面的起点 第一行
7     - unbounded following: 表示到后面的终点 最后一行

```

- 栗子

```

1 --默认从第一行到当前行

```

```

2  select cookieid,createtime,pv,
3         sum(pv) over(partition by cookieid order by createtime) as pv1
4  from website_pv_info;
5
6  --第一行到当前行 等效于rows between不写 默认就是第一行到当前行
7  select cookieid,createtime,pv,
8         sum(pv) over(partition by cookieid order by createtime rows between unbounded
9         preceding and current row) as pv2
10 from website_pv_info;
11
12 --向前3行至当前行
13 select cookieid,createtime,pv,
14         sum(pv) over(partition by cookieid order by createtime rows between 3 preceding
15         and current row) as pv4
16 from website_pv_info;
17
18 --向前3行 向后1行
19 select cookieid,createtime,pv,
20         sum(pv) over(partition by cookieid order by createtime rows between 3 preceding
21         and 1 following) as pv5
22 from website_pv_info;
23
24 --当前行至最后一行
25 select cookieid,createtime,pv,
26         sum(pv) over(partition by cookieid order by createtime rows between current row
27         and unbounded following) as pv6
28 from website_pv_info;
29
30 --第一行到最后一行 也就是分组内的所有行
31 select cookieid,createtime,pv,
32         sum(pv) over(partition by cookieid order by createtime rows between unbounded
33         preceding and unbounded following) as pv6
34 from website_pv_info;

```

●

cookieid	createtime	pv	pv5
cookie1	2018-04-10	1	6
cookie1	2018-04-11	5	13
cookie1	2018-04-12	7	16
cookie1	2018-04-13	3	18
cookie1	2018-04-14	2	21
cookie1	2018-04-15	4	20
cookie1	2018-04-16	4	13
cookie2	2018-04-10	2	5
cookie2	2018-04-11	3	10
cookie2	2018-04-12	5	16
cookie2	2018-04-13	6	19
cookie2	2018-04-14	3	26
cookie2	2018-04-15	9	30
cookie2	2018-04-16	7	25

窗口排序函数、窗口序列函数

- 功能：主要对数据分组排序之后，组内顺序标号。
- 核心函数：==row_number==、rank、dense_rank
- 适合场景：==分组TopN问题==（注意哦 不是全局topN）
- 栗子

```

1  SELECT
2      cookieid,
3      createtime,
4      pv,
5      RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn1,
6      DENSE_RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn2,
7      ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn3
8  FROM website_pv_info;
9
10 --需求：找出每个用户访问pv最多的Top3 重复并列的不考虑
11 SELECT * from
12 (SELECT
13     cookieid,
```

```

14     createtime,
15     pv,
16     ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS seq
17 FROM website_pv_info) tmp where tmp.seq <4;

```

- ==ntile==函数

- 功能：将分组排序之后的数据分成指定的若干个部分（若干个桶）
- 规则：尽量平均分配，优先满足最小的桶，彼此最多不相差1个。
- 栗子

```

1  --把每个分组内的数据分为3桶
2  SELECT
3      cookieid,
4      createtime,
5      pv,
6      NTILE(3) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn2
7  FROM website_pv_info
8  ORDER BY cookieid, createtime;
9
10 --需求：统计每个用户pv数最多的前3分之1天。
11 --理解：将数据根据cookieid分 根据pv倒序排序 排序之后分为3个部分 取第一部分
12 SELECT * from
13 (SELECT
14     cookieid,
15     createtime,
16     pv,
17     NTILE(3) OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn
18  FROM website_pv_info) tmp where rn =1;

```

其他窗口函数

```

1  --LAG 用于统计窗口内往上第n行值
2  SELECT cookieid,
3      createtime,
4      url,
5      ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
6      LAG(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY
createtime) AS last_1_time,
7      LAG(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS last_2_time

```



```

8 FROM website_url_info;
9
10
11 --LEAD 用于统计窗口内往下第n行值
12 SELECT cookieid,
13         createtime,
14         url,
15         ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
16         LEAD(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY
createtime) AS next_1_time,
17         LEAD(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS next_2_time
18 FROM website_url_info;
19
20 --FIRST_VALUE 取分组内排序后，截止到当前行，第一个值
21 SELECT cookieid,
22         createtime,
23         url,
24         ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
25         FIRST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS first1
26 FROM website_url_info;
27
28 --LAST_VALUE 取分组内排序后，截止到当前行，最后一个值
29 SELECT cookieid,
30         createtime,
31         url,
32         ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
33         LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1
34 FROM website_url_info;

```

Hive的数据压缩

- Hive的默认执行引擎是MapReduce，因此通常所说的==Hive压缩指的是MapReduce的压缩==。
- 压缩是指通过==算法对数据进行重新编排==，降低存储空间。无损压缩。
- MapReduce可以在两个阶段进行数据压缩
 - map的输出
 - ==减少shuffle的数据量== 提高shuffle时网络IO的效率
 - reduce的输出
 - 减少输出文件的大小 ==降低磁盘的存储空间==
- 压缩的弊端
 - 浪费时间

- 消耗CPU、内存
- 某些优秀的压缩算法需要钱
- 考虑两个问题
 - a. 压缩和解压缩的时间要短
 - b. 压缩比尽量要高
- 压缩的算法 (==推荐使用snappy==)

```
1 Snappyorg.apache.hadoop.io.compress.SnappyCodec
```

- Hive中压缩的设置：注意 本质还是指的是MapReduce的压缩

```
1 --设置Hive的中间压缩 也就是map的输出压缩
2 1) 开启 hive 中间传输数据压缩功能
3 set hive.exec.compress.intermediate=true;
4 2) 开启 mapreduce 中 map 输出压缩功能
5 set mapreduce.map.output.compress=true;
6 3) 设置 mapreduce 中 map 输出数据的压缩方式
7 set mapreduce.map.output.compress.codec = org.apache.hadoop.io.compress.SnappyCodec;
8
9 --设置Hive的最终输出压缩, 也就是Reduce输出压缩
10 1) 开启 hive 最终输出数据压缩功能
11 set hive.exec.compress.output=true;
12 2) 开启 mapreduce 最终输出数据压缩
13 set mapreduce.output.fileoutputformat.compress=true;
14 3) 设置 mapreduce 最终数据输出压缩方式
15 set mapreduce.output.fileoutputformat.compress.codec
    =org.apache.hadoop.io.compress.SnappyCodec;
16 4) 设置 mapreduce 最终数据输出压缩为块压缩 还可以指定RECORD
17 set mapreduce.output.fileoutputformat.compress.type=BLOCK;
18 --通过load data 不走MR上传数据 student_txt 表看是否有压缩?
19 结论:
20 --设置完毕之后 只有当HiveSQL底层通过MapReduce程序执行 才会涉及压缩。
21 --已有普通格式的表
22 select * from student;
23
24 --ctas语句
25 create table student_snappy as select * from student;
```

```
1 yarn配置: 在CM中直接配置
2 mapreduce.map.output.compress 是否开启map端压缩配置 默认开启的
3 mapreduce.map.output.compress.codec map端采用何种压缩方案
```

4 建议配置为: `org.apache.hadoop.io.compress.SnappyCodec`

5

6 `mapreduce.output.fileoutputformat.compress` 是否开启`reduce`端压缩配置 默认不开启的

7 `mapreduce.output.fileoutputformat.compress.codec` `reduce`端需要采用何种压缩操作

8 建议配置为: `org.apache.hadoop.io.compress.SnappyCodec`

9

10 `mapreduce.output.fileoutputformat.compress.type` 采用压缩的方式

11 建议: block 块压缩

12

13 `hive`配置: 此配置需要在会话中执行

14 `set hive.exec.compress.intermediate=true`; 开启中间结果的压缩

15 `set hive.exec.compress.output=true`; 是否开启最终结果压缩

16

Hive的数据存储格式

- 列式存储、行式存储
 - 数据最终在文件中底层以什么样的形成保存。
 -

数据的存储分为两类:

1. 按行来存储
关系型数据库都是按行来存储
2. 按列来存储
NoSQL数据库 HBase数据库按照列存储

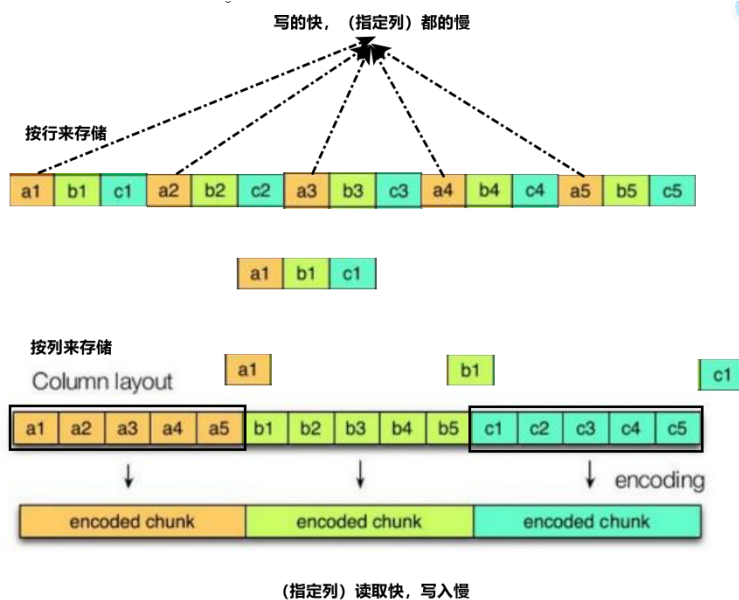
总结:

1. 对于行存储, 按行来存储效率更高;
- 对于列存储, 存储效率低;
2. 对于行 (指定列) 读取, 行的效率低;
- 对于列存储方式, 列读取效率高。

数据仓库中一般会使用列存储, 读取和计算的场景多, 写入的场景少

Logical table representation

a	b	c
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4
a5	b5	c5



- Hive中表的数据存储格式, 不是只支持text文本格式, 还支持其他很多格式。
- hive表的文件格式是如何指定的呢? 建表的时候通过`==STORED AS` 语法指定。如果没有指定默认都是`textfile==`。
- Hive中主流的几种文件格式。
 - `textfile` 文件格式 - 行式存储格式。
 - ORC、Parquet 列式存储格式。

2 二进制意味着肉眼无法直接解析，hive可以自解析。

- o 栗子
- o 分别使用3种不同格式存储数据，去HDFS上查看底层文件存储空间的差异。

```
1  --1、创建表，存储数据格式为TEXTFILE
2  create table log_text (
3  track_time string,
4  url string,
5  session_id string,
6  referer string,
7  ip string,
8  end_user_id string,
9  city_id string
10 )
11 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
12 STORED AS TEXTFILE;  --如果不写stored as textfile 默认就是textfile
13
14 --加载数据
15 load data local inpath '/root/data/log.data' into table log_text;
16
17 --2、创建表，存储数据格式为ORC
18 create table log_orc(
19 track_time string,
20 url string,
21 session_id string,
22 referer string,
23 ip string,
24 end_user_id string,
25 city_id string
26 )
27 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
28 STORED AS orc ;
29
30 --向表中插入数据 思考为什么不能使用load命令加载？ 因为load是纯复制移动操作 不会调整文件格式。
31 insert into table log_orc select * from log_text;
32
33 --3、创建表，存储数据格式为parquet
34 create table log_parquet(
35 track_time string,
36 url string,
```

```
37 session_id string,
38 referer string,
39 ip string,
40 end_user_id string,
41 city_id string
42 )
43 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
44 STORED AS PARQUET ;
45
46 --向表中插入数据
47 insert into table log_parquet select * from log_text ;
```

- 在实际开发中，可以根据需求选择不同的文件格式并且搭配不同的压缩算法。可以得到更好的存储效果。

```
1 --不指定压缩格式 代表什么呢？
2 --orc 存储文件默认采用ZLIB 压缩。比 snappy 压缩的小
3 STORED AS orc;    --2.78M
4
5 --以ORC格式存储 不压缩
6 STORED AS orc tblproperties ("orc.compress"="NONE");  --7.69M
7
8 --以ORC格式存储 使用snappy压缩
9 STORED AS orc tblproperties ("orc.compress"="SNAPPY"); --3.78M
```

- 结论建议：在Hive中推荐使用==ORC+snappy== 或 parquet + snappy 压缩。

Avro格式

常见格式介绍

类型	介绍
TextFile	Hive默认的文件格式，最简单的数据格式，便于查看和编辑，耗费存储空间，I/O性能较低
SequenceFile	含有键值对的二进制文件，优化磁盘利用率和I/O，并行操作数据，查询效率高，但存储空间消耗最大
AvroFile	特殊的二进制文件，设计的主要目标是为了满足schema evolution，Schema和数据保存在一起
OrcFile	列式存储，Schema存储在footer中，不支持schema evolution，高度压缩比并包含索引，查询速度非常快

ParquetFile

列式存储，与Orc类似，压缩比不如Orc，但是查询性能接近，支持的工具更多，通用性更强

- SparkCore缺点：RDD【数据】：没有Schema
- SparkSQL优点：DataFrame【数据 + Schema】
- Schema：列的信息【名称、类型】

Avro格式特点

- 优点
 - 二进制数据存储，性能好、效率高
 - 使用JSON描述模式，支持场景更丰富
 - Schema和数据统一存储，消息自描述
 - 模式定义允许定义数据的排序
- 缺点
 - 只支持Avro自己的序列化格式
 - 少量列的读取性能比较差，压缩比较低
- 场景：基于行的大规模结构化数据写入、列的读取非常多或者Schema变更操作比较频繁的场景
- Sqoop使用Avro格式

```
1  --as-avrodatafile
2  #Imports data to Avro datafiles
3  用这种格式会生成一个Java和avsc文件（表的schema信息）文件下面是指定文件存在哪里 是sqoop命令
4  --outdir logs/java_code    （本地路径）
```

```
1  注意：如果使用了MR的Uber模式，必须在程序加上以下参数避免类冲突问题
2  -Dmapreduce.job.user.classpath.first=true
```

sqoop 测试

```
1  sqoop import \  
2  -Dmapreduce.job.user.classpath.first=true \  
3  --connect jdbc:oracle:thin:@oracle.bigdata.cn:1521:helowin \  
4  --username ciss \  
5  --password 123456 \  
6  --table CISS4.CISS_SERVICE_WORKORDER \  
7  --delete-target-dir \  
8  --target-dir /test/full_imp/ciss4.ciss_service_workorder \  
9  --as-avrodatafile \  
10 --fields-terminated-by "\001" \  
11 -m 1
```

hive建表

```
1 create external table test_avro(  
2 line string  
3 )  
4 stored as avro
```

Schema备份及上传--建表有用

```
1 Avro文件本地存储  
2 workhome=/opt/sqoop/one_make  
3 --outdir ${workhome}/java_code  
4  
5 Avro文件HDFS存储  
6 hdfs_schema_dir=/data/dw/ods/one_make/avsc  
7 hdfs dfs -put ${workhome}/java_code/*.avsc ${hdfs_schema_dir}  
8  
9 Avro文件本地打包  
10 local_schema_backup_filename=schema_${biz_date}.tar.gz  
11 tar -czf ${local_schema_backup_filename} ./java_code/*.avsc  
12 Avro文件HDFS备份  
13 hdfs_schema_backup_filename=${hdfs_schema_dir}/avro_schema_${biz_date}.tar.gz  
14 hdfs dfs -put ${local_schema_backup_filename} ${hdfs_schema_backup_filename}  
15  
16 运行测试  
17 cd /opt/sqoop/one_make/  
18 ./upload_avro_schema.sh  
19
```

备份--建表有用

```
1 #!/usr/bin/env bash  
2 # 上传  
3 # /bin/bash  
4 workhome=/opt/sqoop/one_make  
5 hdfs_schema_dir=/data/dw/ods/one_make/avsc  
6 biz_date=20210101  
7 biz_fmt_date=2021-01-01  
8 local_schema_backup_filename=schema_${biz_date}.tar.gz  
9 hdfs_schema_backup_filename=${hdfs_schema_dir}/avro_schema_${biz_date}.tar.gz  
10 log_file=${workhome}/log/upload_avro_schema_${biz_fmt_date}.log  
11  
12 # 打印日志
```

```

13 log() {
14     cur_time=`date "+%F %T"`
15     echo "${cur_time} $" >> ${log_file}
16 }
17
18 source /etc/profile
19 cd ${workhome}
20
21 # hadoop fs [generic options] [-test [-defsz] <path>]
22 # -test [-defsz] <path> :
23 # Answer various questions about <path>, with result via exit status.
24 # -d return 0 if <path> is a directory.
25 # -e return 0 if <path> exists.
26 # -f return 0 if <path> is a file.
27 # -s return 0 if file <path> is greater than zero bytes in size.
28 # -z return 0 if file <path> is zero bytes in size, else return 1.
29
30 log "Check if the HDFS Avro schema directory ${hdfs_schema_dir}..."
31 #判断文件是否存在
32 hdfs dfs -test -e ${hdfs_schema_dir} > /dev/null
33
34 #如果不存在则创建
35 if [ $? != 0 ]; then
36     log "Path: ${hdfs_schema_dir} is not exists. Create a new one."
37     log "hdfs dfs -mkdir -p ${hdfs_schema_dir}"
38     hdfs dfs -mkdir -p ${hdfs_schema_dir}
39 fi
40
41 log "Check if the file ${hdfs_schema_dir}/CISS4_CISS_BASE_AREAS.avsc has uploaded to
the HFDS..."
42 #判断文件是否存在
43 hdfs dfs -test -e ${hdfs_schema_dir}/CISS4_CISS_BASE_AREAS.avsc.avsc > /dev/null
44 #不存在则上传文件
45 if [ $? != 0 ]; then
46     log "Upload all the .avsc schema file."
47     log "hdfs dfs -put ${workhome}/java_code/*.avsc ${hdfs_schema_dir}"
48     hdfs dfs -put ${workhome}/java_code/*.avsc ${hdfs_schema_dir}
49 fi
50
51 # backup

```



```

52 log "Check if the backup tar.gz file has generated in the local server..."
53 # ! 取反 判断文件是否存在 不存在则执行以下命令
54 if [ ! -e ${local_schema_backup_filename} ]; then
55     log "package and compress the schema files"
56     log "tar -czf ${local_schema_backup_filename} ./java_code/*.avsc"
57     tar -czf ${local_schema_backup_filename} ./java_code/*.avsc
58 fi
59
60 log "Check if the backup tar.gz file has upload to the HDFS..."
61 #判断hadoop文件是否存在 不存在则上传
62 hdfs dfs -test -e ${hdfs_schema_backup_filename} > /dev/null
63 if [ $? != 0 ]; then
64     log "upload the schema package file to HDFS"
65     log "hdfs dfs -put ${local_schema_backup_filename} ${hdfs_schema_backup_filename}"
66     hdfs dfs -put ${local_schema_backup_filename} ${hdfs_schema_backup_filename}
67 fi

```

Avro建表语法

- Hive官网: <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateTable>
- DataBricks官网: <https://docs.databricks.com/spark/2.x/spark-sql/language-manual/create-table.html>
- Avro用法: <https://cwiki.apache.org/confluence/display/Hive/AvroSerDe>

指定文件类型

```

1 方式一：指定类型（更简单）
2 stored as avro
3
4 方式二：指定解析类（可以指定配置）
5 --解析表的文件的时候，用哪个类来解析
6 ROW FORMAT SERDE
7 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
8 --读取这张表的数据用哪个类来读取
9 STORED AS INPUTFORMAT
10 'org.apache.hadoop.hive.q1.io.avro.AvroContainerInputFormat'
11 --写入这张表的数据用哪个类来写入
12 OUTPUTFORMAT
13 'org.apache.hadoop.hive.q1.io.avro.AvroContainerOutputFormat'

```

指定Schema

```
1 方式一：手动定义Schema
2 CREATE TABLE embedded
3 COMMENT "这是表的注释"
4 ROW FORMAT SERDE
5   'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
6 STORED AS INPUTFORMAT
7   'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
8 OUTPUTFORMAT
9   'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
10 TBLPROPERTIES (
11   'avro.schema.literal'='{
12     "namespace": "com.howdy",
13     "name": "some_schema",
14     "type": "record",
15     "fields": [ { "name":"string1","type":"string"}]
16   }'
17 );
18
19 方式二：加载Schema文件
20 CREATE TABLE embedded
21 COMMENT "这是表的注释"
22 ROW FORMAT SERDE
23   'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
24 STORED as INPUTFORMAT
25   'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
26 OUTPUTFORMAT
27   'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
28 TBLPROPERTIES (
29   'avro.schema.url'='file:///path/to/the/schema/embedded.avsc'
30 );
```

具体建表语法：

```
1 方式一：指定类型和加载Schema文件
2 create external table one_make_ods_test.ciss_base_areas
3 comment '行政地理区域表'
4 PARTITIONED BY (dt string)
5 stored as avro
```

```

6 location '/data/dw/ods/one_make/full_imp/ciss4.ciss_base_areas'
7 TBLPROPERTIES
  ('avro.schema.url'='/data/dw/ods/one_make/avsc/CISS4_CISS_BASE_AREAS.avsc');
8
9 方式二：指定解析类和加载Schema文件
10 create external table one_make_ods_test.ciss_base_areas
11 comment '行政地理区域表'
12 PARTITIONED BY (dt string)
13 ROW FORMAT SERDE
14   'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
15 STORED AS INPUTFORMAT
16   'org.apache.hadoop.hive ql.io.avro.AvroContainerInputFormat'
17 OUTPUTFORMAT
18   'org.apache.hadoop.hive ql.io.avro.AvroContainerOutputFormat'
19 location '/data/dw/ods/one_make/full_imp/ciss4.ciss_base_areas'
20 TBLPROPERTIES
  ('avro.schema.url'='/data/dw/ods/one_make/avsc/CISS4_CISS_BASE_AREAS.avsc');
21
22 create external table 数据库名称.表名
23 comment '表的注释'
24 partitioned by
25 ROW FORMAT SERDE
26   'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
27 STORED AS INPUTFORMAT
28   'org.apache.hadoop.hive ql.io.avro.AvroContainerInputFormat'
29 OUTPUTFORMAT
30   'org.apache.hadoop.hive ql.io.avro.AvroContainerOutputFormat'
31 location '这张表在HDFS上的路径'
32 TBLPROPERTIES ('这张表的Schema文件在HDFS上的路径')

```

Hive通用调优

Fetch抓取机制

- 功能：在执行sql的时候，==能不走===MapReduce程序处理就尽量不走MapReduce程序处理==。
- 尽量直接去操作数据文件。
- 设置：hive.fetch.task.conversion= more。

1 --在下述3种情况下 sql不走mr程序

```
2
3 --全局查找
4 select * from student;
5 --字段查找
6 select num,name from student;
7 --limit 查找
8 select num,name from student limit 2;
```

mapreduce本地模式

- 功能：如果非要执行==MapReduce程序，能够本地执行的，尽量不提交yarn上执行==。
- 默认是关闭的。意味着只要走MapReduce就提交yarn执行。

```
1 mapreduce.framework.name = local 本地模式
2 mapreduce.framework.name = yarn 集群模式
```

- Hive提供了一个参数，自动切换MapReduce程序为本地模式，如果不满足条件，就执行yarn模式。

```
1 set hive.exec.mode.local.auto = true;
2
3 --3个条件必须都满足 自动切换本地模式
4 The total input size of the job is lower than: hive.exec.mode.local.auto.inputbytes.max
  (128MB by default) --数据量小于128M
5
6 The total number of map-tasks is less than: hive.exec.mode.local.auto.tasks.max (4 by
  default) --maptask个数少于4个
7
8 The total number of reduce tasks required is 1 or 0. --reducetask个数是0 或者 1
```

- 切换Hive的执行引擎

```
1 WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future
  versions. Consider using a different execution engine (i.e. spark, tez) or using Hive
  1.X releases.
2
3 如果针对Hive的调优依然无法满足你的需求 还是效率低， 尝试使用spark计算引擎 或者Tez.
```

join优化

- 底层还是MapReduce的join优化。
- MapReduce中有两种join方式。指的是join的行为发生什么阶段。
 - map端join
 - reduce端join

- 优化1: ==Hive自动尝试选择map端join提高join的效率== 省去shuffle的过程。

```
1 开启 mapjoin 参数设置:
2  (1) 设置自动选择 mapjoin
3  set hive.auto.convert.join = true;  --默认为 true
4  (2) 大表小表的阈值设置:
5  set hive.mapjoin.smalltable.filesize= 25000000;
```

- 优化2: 大表join大表

```
1  --背景:
2  大表join大表本身数据就十分具体, 如果join字段存在null空值 如何处理它?
3
4  --方式1: 空key的过滤 此行数据不重要
5  参与join之前 先把空key的数据过滤掉
6  SELECT a.* FROM (SELECT * FROM nullidtable WHERE id IS NOT NULL ) a JOIN ori b ON a.id
   =b.id;
7
8  --方式2: 空Key转换
9  CASE WHEN a.id IS NULL THEN 'xxx任意字符串' ELSE a.id END
10 CASE WHEN a.id IS NULL THEN concat('hive', rand()) ELSE a.id  --避免转换之后数据倾斜 随机
   分布打散
11
12 分类表
13 cid categoryName
14 c001 手机
15 c002 家电
16 ...
17
18 商品表
19 pid pname price cid
20 p001 苹果13pro 10000 c001
21 p002 扫地机器人 200 c002
22 ...
23
24 select * from 商品表 join (select * from 分类表 where cid='手机')tmp on 商品
   表.cid=tmp.cid;
```

- 优化3: 桶表join提高优化效率。bucket mapjoin

```
1 1.1 条件
2      1) set hive.optimize.bucketmapjoin = true;
```

```
3      2) 一个表的bucket数是另一个表bucket数的整数倍
4      3) bucket列 == join列
5      4) 必须是应用在map join的场景中
6
7 1.2 注意
8      1) 如果表不是bucket的, 只是做普通join。
```

- 将那些容易产生倾斜的key, 从整个MR剔除掉, 使用一个单独的MR进行处理, 从而保证不会有倾斜问题
 - 运行期优化:
 - 配置信息: **set hive.optimize.skewjoin=true;** 默认为false
 - 优化过程: hive SQL在执行运行过程中, 动态的去监测每一个组内的key, 如果发现某个key数据量远远大于其他key的数量, 认为此key存在倾斜问题, 将这个倾斜的key剔除掉, 单独放置在一个MR中运行即可
 - 思考: 当这个key 的值达到多少个的时候, 认为出现倾斜呢?
 - **set hive.skewjoin.key=100000;**
 - 编译期优化:
 - 配置信息: **set hive.optimize.skewjoin.compiletime=true;** 默认为false
 - 优化过程: 在执行之前, 我们已经提前知道某些key的值可能会有倾斜, 前提将其排除掉
 - 注意: 在建表的时候, 就需要提前定义好哪些值会出现倾斜

```
1 建表语句:
2 CREATE TABLE list_bucket_single (key STRING, value STRING)
3 -- 倾斜的字段和需要拆分的key值
4 SKEWED BY (key) ON (1,5,6)
5 -- 为倾斜值创建子目录单独存放
6 [STORED AS DIRECTORIES];
```

在实际生产中, 如何配置呢? 两个都配置, 这样满足了那种 就按照那种即可

- 注意: 不管使用运行时优化 还是编译期优化, 都会将倾斜的key单独找一个MR来运行, 运行之后得到的结果和之前的MR进行union all合并操作, 最终得到合并后结果即可
- 执行union all 也需要执行一个MR来运行, 此处依然可以进行优化操作:

```
1 配置:
2 set hive.optimize.union.remove=true;
3
4 一旦开启此配置后, 在执行union all合并操作的时候, 可以不走MR, 也不会合并为一个文件了, 执行让两个MR将结果输出到目标地即可
```

group by 数据倾斜优化

```
1 (1) 是否在 Map 端进行聚合, 默认为 True
2 set hive.map.aggr = true;
```

```

3  (2) 在 Map 端进行聚合操作的条目数目
4  set hive.groupby.mapaggr.checkinterval = 100000;
5  (3) 有数据倾斜的时候进行负载均衡 (默认是 false)
6  set hive.groupby.skewindata = true;
7
8
9  --Q:在hive中数据倾斜开启负载均衡之后 底层执行机制是什么样?
10
11  --step1:启动一个MapReduce程序 将倾斜的数据随机发送到各个reduce中 进行打散
12      每个reduce进行聚合都是局部聚合
13
14  --step2:再启动第二个MapReduce程序 将上一步局部聚合的结果汇总起来进行最终的聚合

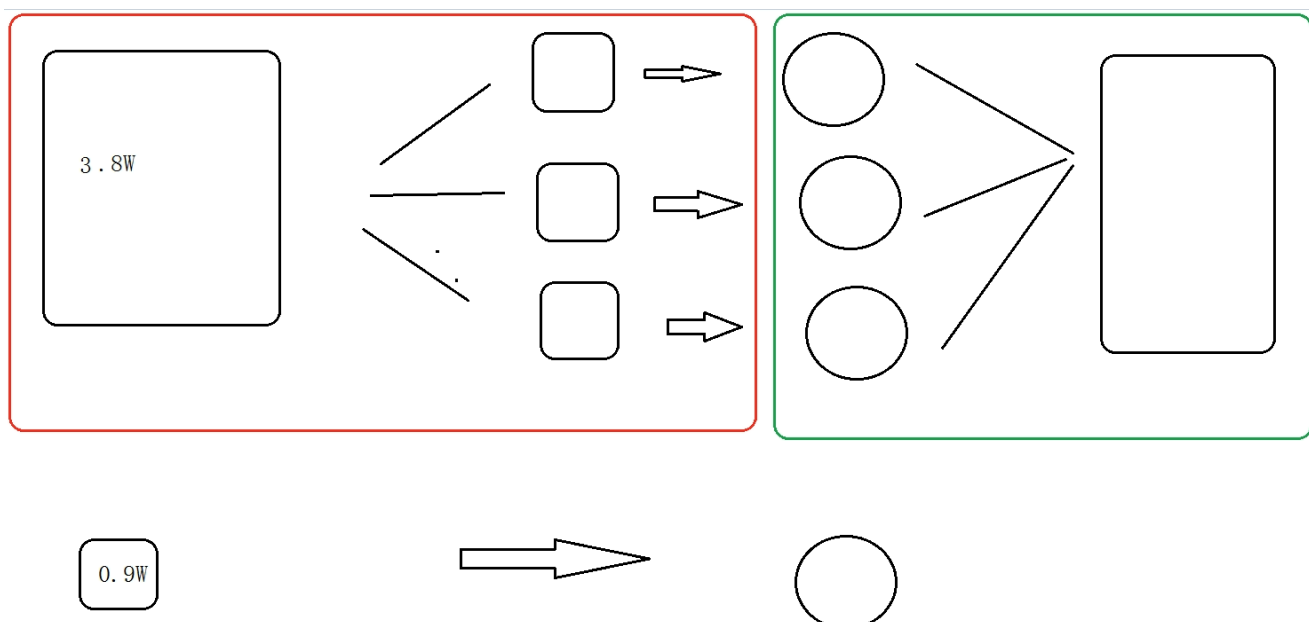
```

Union优化

```

1  应用了表连接倾斜优化以后，会在执行计划中插入一个新的union操作，此时建议开启对union
2  的优化配置：
3  set hive.optimize.union.remove=true;
4  默认关闭。
5  此项配置减少对Union all子查询中间结果的二次读写，可以避免union输出的额外扫描过程，
6  当我们开启了skewjoin时尤其有用，建议同时开启。
7  set hive.optimize.skewjoin=true;
8  set hive.optimize.skewjoin.compiletime=true;
9  set hive.optimize.union.remove=true;

```



hive中如何调整底层MapReduce中task的个数（并行度）

- maptask个数
 - 如果是在MapReduce中 maptask是通过==逻辑切片==机制决定的。
 - 但是在hive中，影响的因素很多。比如逻辑切片机制，文件是否压缩、压缩之后是否支持切割。
 - 因此在==Hive中，调整MapTask的个数，直接去HDFS调整文件的大小和个数，效率较高==。

```
1  如果小文件多，就进行小文件的合并  合并的大小最好=block size如果大文件多，就调整block size
```

- reducetask个数
 - 如果在MapReduce中，通过代码可以直接指定 job.setNumReduceTasks(N)
 - 在Hive中，reducetask个数受以下几个条件控制的

```
1  (1) 每个 Reduce 处理的数据量默认是 256MB
2  hive.exec.reducers.bytes.per.reducer=256000000
3  (2) 每个任务最大的 reduce 数，默认为 1009
4  hive.exec.reducers.max=1009
5  (3) mapreduce.job.reduces
6  该值默认为-1，由 hive 自己根据任务情况进行判断。
7
8
9  --如果用户不设置 hive将会根据数据量或者sql需求自己评估reducetask个数。
10 --用户可以自己通过参数设置reducetask的个数
11  set mapreduce.job.reduces = N
12 --用户设置的不一定生效，如果用户设置的和sql执行逻辑有冲突，比如order by，在sql编译期间，hive又
   会将reducetask设置为合理的个数。
13
14  Number of reduce tasks determined at compile time: 1
```

其他几个通用调优

- 执行计划explain
 - 通过执行计划可以看出==hive接下来是如何打算执行这条sql的==。
 - 语法格式：explain + sql语句
 - 栗子

```
1  explain select * from student;
2
3  +-----+
4  |                  Explain                  |
5  +-----+
6  | STAGE DEPENDENCIES:                      |
7  |   Stage-0 is a root stage                |
```



```

8 |
9 | STAGE PLANS:
10 |   Stage: Stage-0
11 |     Fetch Operator
12 |       limit: -1
13 |     Processor Tree:
14 |       TableScan
15 |         alias: student
16 |         Statistics: Num rows: 1 Data size: 5260 Basic stats: COMPLETE Column stats:
    NONE |
17 |       Select Operator
18 |         expressions: num (type: int), name (type: string), sex (type: string),
    age (type: int), dept (type: string) |
19 |         outputColumnNames: _col0, _col1, _col2, _col3, _col4 |
20 |         Statistics: Num rows: 1 Data size: 5260 Basic stats: COMPLETE Column
    stats: NONE |
21 |       ListSink
22 |
23 +-----+

```

- 并行执行机制

- 如果hivesql的底层某些stage阶段可以并行执行，就可以提高执行效率。
- 前提是==stage之间没有依赖== 并行的弊端是瞬时服务器压力变大。
- 参数

```

1 set hive.exec.parallel=true; --是否并行执行作业。适用于可以并行运行的 MapReduce 作业，例如在
  多次插入期间移动文件以插入目标
2 set hive.exec.parallel.thread.number=16; --最多可以并行执行多少个作业。默认为8。

```

- Hive的严格模式

- 注意。不要和动态分区的严格模式搞混淆。
- 这里的严格模式指的是开启之后 ==hive会禁止一些用户都影响不到的错误包括效率低下的操作==，不允许运行一些有风险的查询。
- 设置

```

1 set hive.mapred.mode = strict --默认是严格模式  nonstrict

```

- 解释

```

1 1、如果是分区表，没有where进行分区裁剪 禁止执行
2 2、order by语句必须+limit限制

```

- 推测执行机制

- MapReduce中task的一个机制。
- 功能：
 - 一个job底层可能有多个task执行，如果某些拖后腿的task执行慢，可能会导致最终job失败。
 - 所谓的==推测执行机制就是通过算法找出拖后腿的task,为其启动备份的task==。
 - 两个task同时处理一份数据，谁先处理完，谁的结果作为最终结果。
- 推测执行机制默认是开启的，但是在企业生产环境中==建议关闭==。

```
1  --分区
2  SET hive.exec.dynamic.partition=true;--开启动态分区功能
3  SET hive.exec.dynamic.partition.mode=nonstrict;--设置为非严格模式
4  set hive.exec.max.dynamic.partitions.pernode=10000;--在每个执行MR的节点上，最大可以创建多少个动态分区。
5  set hive.exec.max.dynamic.partitions=100000;--在每个执行MR的节点上，最大一共可以创建多少个动态分区。
6  set hive.exec.max.created.files=150000;--整个MR Job中，最大可以创建多少个HDFS文件。
7  --当有空分区生成时，是否抛出异常。一般不需要设置。
8  hive.error.on.empty.partition=false
9  --hive压缩
10 set hive.exec.compress.intermediate=true;
11 set hive.exec.compress.output=true;
12 --写入时压缩生效
13 set hive.exec.orc.compression.strategy=COMPRESSION;
14 --分桶
15 set hive.enforce.bucketing=true;
16 set hive.enforce.sorting=true;
17 set hive.optimize.bucketmapjoin = true;
18 set hive.auto.convert.sortmerge.join=true;
19 set hive.auto.convert.sortmerge.join.noconditionaltask=true;
20 --并行执行
21 set hive.exec.parallel=true;
22 set hive.exec.parallel.thread.number=8;
23 --小文件合并
24 #设置Hive中底层MapReduce读取数据的输入类：将所有文件合并为一个大文件作为输入
25 hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
26 #如果hive的程序，只有maptask，将MapTask产生的所有小文件进行合并
27 hive.merge.mapfiles=true;--是否开启合并Map端小文件，在Map-only的任务结束时合并小文件，true是打开。
28 hive.merge.mapredfiles=true;--是否开启合并Reduce端小文件，在map-reduce作业结束时合并小文件。true是打开。
29 hive.merge.size.per.task=256000000;--合并后MR输出文件的大小，默认为256M。
```

```
30 hive.merge.smallfiles.avgsize=16000000;--当输出文件的平均大小小于此设置值时，启动一个独立的
map-reduce任务进行文件merge，默认值为16M。hive.merge.smallfiles.avgsize
31 --矢量化查询：注意：要使用矢量化查询执行，就必须以ORC格式存储数据。
32 hive.vectorized.execution.enabled = true;
33 hive.vectorized.execution.reduce.enabled = true;
34 --关联优化器
35 --在Hive的一些复杂关联查询中，可能同时还包含有group by等能够触发shuffle的操作，有些时候shuffle
操作是可以共享的，通过关联优化器选项，可以尽量减少复杂查询中的shuffle，从而提升性能。
36 --作用：如果SQL中，有一些shuffle是相同操作，可以选择让其共享，以此来提升当前SQL的执行效率
37 set hive.optimize.correlation=true;
38 --读取零拷贝
39 set hive.exec.orc.zerocopy=true;
40 --JVM重用,随着Hadoop版本的升级，已自动优化了JVM重用选项，MRv2开始不再支持JVM重用。
41 mapreduce.job.jvm.numtasks=10
42 --本地模式
43 hive.exec.mode.local.auto=true;
44 --推测执行
45 mapreduce.map.speculative=true
46 mapreduce.reduce.speculative=true
47 hive.mapred.reduce.tasks.speculative.execution=true
48 --Fetch抓取
49 hive.fetch.task.conversion=more
50 --CBO优化器
51 hive.cbo.enable=true;
52 hive.compute.query.using.stats=true;
53 hive.stats.fetch.column.stats=true;
54 hive.stats.fetch.partition.stats=true;
55 --索引优化
56 hive.optimize.index.filter=true
57 --谓词下推PPD
58 hive.optimize.ppd=true;
59 --Map Join
60 hive.auto.convert.join=true
61 hive.auto.convert.join.noconditionaltask.size=512000000
62 --Bucket Join
63 hive.optimize.bucketmapjoin = true;
64 hive.auto.convert.sortmerge.join=true;
65 hive.optimize.bucketmapjoin.sortedmerge = true;
66 hive.auto.convert.sortmerge.join.noconditionaltask=true;
67 --Task内存
```

```
68 mapreduce.map.java.opts=-Xmx6000m;
69 mapreduce.map.memory.mb=6096;
70 mapreduce.reduce.java.opts=-Xmx6000m;
71 mapreduce.reduce.memory.mb=6096;
72 --yarn内存
73 yarn.scheduler.maximum-allocation-mb=4096
74 yarn.scheduler.minimum-allocation-mb=1024
75 --application-core
76 yarn.app.mapreduce.am.resource.cpu-vcores=2
77 --mapreduce-map-core
78 mapreduce.map.cpu.vcores=3
79 --mapreduce-reduce-core
80 mapreduce.reduce.cpu.vcores=3
81 --缓冲区大小
82 mapreduce.task.io.sort.mb=100
83 --Spill阈值
84 mapreduce.map.sort.spill.percent=0.8
85 --Merge线程
86 mapreduce.task.io.sort.factor=10
87 --Reduce拉取并行度
88 mapreduce.reduce.shuffle.parallelcopies=8
89 mapreduce.reduce.shuffle.read.timeout=180000
90 --开启Combiner: Map端聚合【规约】,hive不支持多列上的去重操作,并报错:
91 hive.map.aggr=true
92
93 --groupby时开启随机分区
94 hive.groupby.skewindata=true
95 --join时开启skewjoin
96     如果大表和大表进行join操作,则可采用skewjoin(倾斜关联)来开启对倾斜数据的优化。
97     skewjoin原理:
98         对于skewjoin.key,在执行job时,将它们存入临时的HDFS目录,其它数据正常执行
99         对倾斜数据开启map join操作(多个map并行处理),对非倾斜值采取普通join操作
100         将倾斜数据集和非倾斜数据集进行合并Union操作。
101 --开启运行过程中skewjoin
102 set hive.optimize.skewjoin=true;
103 --如果这个key的出现次数超过这个范围,就认为这个key产生了数据倾斜,则会对其进行分拆优化。
104 set hive.skewjoin.key=100000;
105 --在编译时判断是否会产生数据倾斜
106 set hive.optimize.skewjoin.compiletime=true;
```

```

107      --不合并，提升性能,应用了表连接倾斜优化以后，会在执行计划中插入一个新的union操作，此时建议
      开启对union的优化配置：
108      set hive.optimize.union.remove=true;
109      --如果Hive的底层走的是MapReduce，必须开启这个属性，才能实现不合并
110      set mapreduce.input.fileinputformat.input.dir.recursive=true;
111
112  --大表join大表空key过滤
113  5) 测试过滤空id
114  insert overwrite table jointable
115  select n.* from (select * from nullidtable where id is not null ) n left join ori o on
      n.id = o.id;
116  --大表join大表空key转换
117  设置reduce数量
118  mapreduce.job.reduces = 5; (可以不设置)
119  insert overwrite table jointable
120  select n.* from nullidtable n full join ori o on
121  case when n.id is null then concat('hive', rand()) else n.id end = o.id;

```

hive 并行操作

```

1  1.1) Hive编译查询限制
2  说明：在hive中执行SQL的时候，SQL会先进行编译操作，当如果有多个会话一起来执行SQL操作，hive同
      时只能对一个会话中SQL进行编译，其他的SQL只能等待操作，此时效率比较低
3  如何解决呢？
4      hive.driver.parallel.compilation 将此参数设置为 true 表示开启同时编译操作
5      hive.driver.parallel.compilation.global.limit 在进行同时编译的时候，一次性最多运行多
      多少个会话进行编译，默认值为 3 此值如果设置为 0 或者 负值 表示 无限制
6
7      以上两个配置，建议是直接CM中进行配合操作
8
9  1.2) hive的并行执行操作
10 说明：在执行hive的SQL的时候，一个SQL可能会编译为多个阶段，而在某些情况下，各个阶段之间的依
      赖关系并不强，是没有先后顺序的，此时可以安排其并行执行，从而提升效率，默认是串行执行的
11 如何设置呢？
12      set hive.exec.parallel=true, 可以开启并发执行，默认为false。
13      set hive.exec.parallel.thread.number=16; //同一个sql允许的最大并行度，默认为8。
14
15 例如：
16      select
17          *
18      from (select * FROM 表 A where starts_time = '2021-03-05') TEMP1
19      join

```

```
20      (select * FROM 表 B where starts_time = '2021-03-05') TEMP2 ;
21      比如还有 union all的操作
```

```
1  设置执行引擎
2  hive.execution.engine=mr
3  hive.execution.engine=spark
4
5  针对mr端设置参数
6  设置每隔map处理的大小
7  mapred.max.split.size=1000000;
8  # 设置map数
9  set mapred.max.split.size=112345600;
10 set mapred.min.split.size.per.node=112345600;
11 set mapred.min.split.size.per.rack=112345600;
12 set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
13 # 增大map数
14 set mapred.reduce.tasks=10;
15 设置申请map资源 内存
16 mapreduce.map.memory.mb=4096
17 设置申请cpu资源（可能无用）
18 mapreduce.map.cpu.vcores=1
19
20 动态生成分区的线程数
21 说明：在执行动态分区过程中，可以运行多少个线程来生产分区数据，线程数量越多，执行效率越高，前提是有资源
22 hive.load.dynamic.partitions.thread 默认值为 15
23
24 监听输入文件线程数
25 说明：hive在读取数据的线程数量配置，此配置越高 读取数据效率越高，前提是有资源
26 hive.exec.input.listing.max.threads 默认值为 15
```

问题

- beeline 的免密登录

```
1  # 编辑环境变量
2  vim /etc/profile
3  # 将此命令放到最后
```

```
4 alias beeline="beeline -u jdbc:hive2://node1:10000 -n root -p 123456"
5 # 快速生效环境变量
6 source /etc/profile
```

- 动态分区如何加载每天的数据

```
1 # 直接从 HDFS 加载到 ods 层的数据
2 #2.每日调度 每天跑一次
3 dt=date 'interval 1 days ago' +'%y%m%d' # 昨天的时间
4 alter table t_student_partition add partition (dt=$dt)
5 location '/user/hive/warehouse/ods.db/t_student_partition/"dt=$dt"'
6
7 #1.加载进来
8 insert into t_student_partition partition(dt) select * from t_student where
dt='20220103' join where;
```

- Hive Join 在MR 的底层原理
- <https://blog.csdn.net/u013668852/article/details/79768266>

• grouping sets

- grouping sets
 - 功能：根据给定的不同的维度组合进行分组聚合，相当于挨个分组聚合然后union合并
 - 语法：grouping sets (维度1,维度2,维度3) ;
 - 特点：语法简单，性能更好，只对表的数据进行一次查询

cube函数

- 功能：将给定的维度自动按照**所有维度子集**进行分组聚合合并的结果
 - 全集： (A,B,C)
 - 子集

```
1 A
2 B
3 C
4 AB
5 AC
6 BC
7 (A,B,C)
8 ( )
```

rollup函数

- 功能：将给定的维度**自动按照从左往右逐级递减**对所有维度子集进行分组聚合合并的结果
 - 全集： (A,B,C)
 - 子集

```
1 (A,B,C)
2 (A,B)
3 (A)
4 (
```

- **特殊函数grouping的功能 (presto适用)**
- **功能：**通过grouping来判断是否基于当前维度实现了分组，实现过滤和判断
- 如果基于这个维度则为0 spark相反则为1
- 用grouping(字段)=十进制，字段可以多个，可以输入想要筛选的字段=0
 - grouping(字段) = 0
- 精准则把全部分组的字段放入里面，需要筛选的则=多少

```
1 (create_date,max_class_id,max_class_name,mid_class_id,mid_class_name,min_class_id,min_class_name) = 01111000
2 #=十进制数字
3 grouping
  (create_date,store_id,trade_area_id,city_id,brand_id,min_class_id,mid_class_id,max_class_id) = 127
```

• grouping__id (hive适用)

- 使用方法,pycharm爆红

```
1 select book_name,output,grouping__id from book group by book_name,output grouping sets
  (book_name,output,(book_name,output));
```

- 将 group by 的所有字段 倒序 排列。对于每个字段，如果该字段出现在了当前粒度中，则该字段位置赋值为1，否则为0。
- spark相反，不会把 所有字段 倒序 排列，对于每个字段，如果该字段出现在了当前粒度中，则该字段位置赋值为0，否则为1。
- spark的语法是grouping_id()

• 其他函数

- trim：去除字符串两边的空格
 - trim (" abc ") 返回abc
- regexp_replace:正则表达式
 - select regexp_replace('foobar', 'oo|ar', '') 返回fb
- regexp_extract：正则表达式解析函数：将字符串subject按照pattern正则表达式的规则拆分，返回index指定的字符。
 - select regexp_extract('foothebar', 'foo(.?)(bar)', 1) from dual; ##返回值为the
- unix时间戳转字符串

- `select from_unixtime(unix_timestamp(),'yyyy/MM/dd HH:mm:ss');`
- 时间转时间戳
 - `select unix_timestamp(sso.signin_time,'yyyy-MM-dd HH:mm:ss')`
- rlike: 正则匹配

mask脱敏函数

```

1  •mask
2  •mask_first_n(string str[, int n])
3  •mask_last_n(string str[, int n])
4  •mask_show_first_n(string str[, int n])
5  •mask_show_last_n(string str[, int n])
6  •mask_hash(string|char|varchar str)
7  --mask
8  --将查询回的数据，大写字母转换为 X，小写字母转换为 x，数字转换为 n。
9  select mask("abc123DEF");
10 select mask("abc123DEF","-",".",'^'); --自定义替换的字母
11 --mask_first_n(string str[, int n])
12 --对前 n 个进行脱敏替换
13 select mask_first_n("abc123DEF",4);
14 --mask_last_n(string str[, int n])
15 select mask_last_n("abc123DEF",4);
16 --mask_show_first_n(string str[, int n])
17 --除了前 n 个字符，其余进行掩码处理select mask_show_first_n("abc123DEF",4);
18 --mask_show_last_n(string str[, int n])
19 select mask_show_last_n("abc123DEF",4);
20 --mask_hash(string|char|varchar str)
21 --返回字符串的 hash 编码。
22 select mask_hash("abc123DEF");

```

将数据表转换为外部表

```
alter table 表名 set tblproperties('EXTERNAL'='TRUE');
```

将数据表转换为内部表

```
alter table 表名 set tblproperties('EXTERNAL'='FALSE');
```