什么是Hive

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张表，并提供类 SQL 查询功能。用于解决海量结构化日志的数据统计。

本质是：将 HQL 转化成MapReduce 程序

- Hive 处理的数据存储在 HDFS
- Hive 分析数据底层的默认实现是 MapReduce
- 执行程序运行在 Yarn 上

优缺点

优点

- 操作接口采用类 SQL 语法，提供快速开发的能力（简单、容易上手）。
- 避免了去写 MapReduce，减少开发人员的学习成本。
- Hive 的执行延迟比较高，因此Hive 常用于数据分析，对实时性要求不高的场合。
- Hive 优势在于处理大数据，对于处理小数据没有优势，因为Hive 的执行延迟比较高。
- Hive 支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

缺点

- Hive 的 HQL 表达能力有限
  - 迭代式算法无法表达
  - 数据挖掘方面不擅长
- Hive 的效率比较低
  - Hive 自动生成的 MapReduce 作业，通常情况下不够智能化
  - Hive 调优比较困难，粒度较粗

配置

操作语句都跟sql类似

- 启动hive

```
bin/hive
```

- 退出 hive

```
quit;
```

文件导入hive

- 先在本地准备数据文件（数据用`\t`分割开）

- 需要导入的表创建时需要指定好分隔符

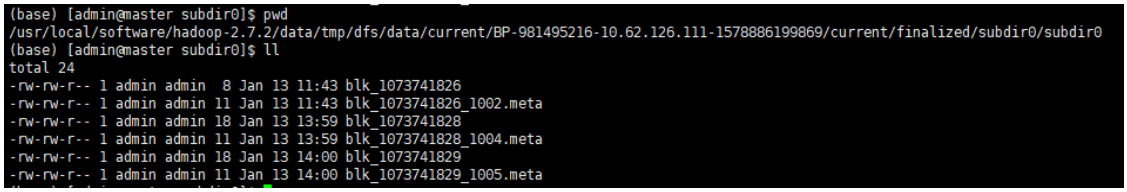
```
create table student(id int, name string) row format delimited fields
terminated by '\t';
```

- 加载命令

```
# local 关键字表示在本地导入
load data local inpath '/home/admin/stu.txt' into table student;

# 此时的路径是hdfs路径
load data inpath '/home/admin/stu.txt' into table student;
```

> 本地文件导入时，实际上时使用hdfs的 `-put` 命令，将文件上传到hive表的路径中
>
> hdfs上的文件导入时，效果上时移动数据文件到hive表的路径里，实际上，是改动了namenode中的元数据路径信息，并非文件的移动。下图是hdfs存储数据的路径



MySql权限设置

```
# 查询 user表
select User, Host, Password from mysql.user;

# 修改 user 表，把 Host 表内容修改为%
update user set host='%' where host='localhost';

# 删除root 用户的其他 host
delete from user where Host='master';
delete from user where Host='127.0.0.1';
delete from user where Host='::1';

# 刷新
flush privileges;
```

配置Metastore到MySql

```
# 将mysql-connector-java-5.1.27-bin.jar拷贝到conf目录下

cd hive/conf
vim hive-site.xml
```

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:mysql://master:3306/metastore?
createDatabaseIfNotExist=true</value>
```

```
        <description>JDBC connect string for a JDBC metastore</description>
    </property>

    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
        <description>Driver class name for a JDBC metastore</description>
    </property>

    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>root</value>
        <description>username to use against metastore database
</description>
    </property>

    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>123456</value>
        <description>password to use against metastore database
</description>
    </property>
</configuration>

```

常用命令

- -e : 不进入 hive 的交互窗口执行 sql 语句

```
bin/hive -e "select id from student;"
```

- -f : 不进入hive的交互窗口执行文件里的sql语句

```
bin/hive -f /home/admin/stu.txt

# 将输出结果输出到aa.txt，属于linux命令
bin/hive -f /home/admin/stu.txt > /home/admin/aa.txt
```

- 不常用操作
  - 在hive client 命令窗口中查看 hdfs 文件系统

```
hive> dfs -ls /;
```

  - 在hive client 命令窗口中查看本地文件系统

```
hive> ! ls ./;
```

  - 查看在 hive 中输入的所有历史命令

```
# 进入到当前用户的根目录/root 或/home/admin
# 查看. hivehistory 文件
cat .hivehistory
```

常见配置

数据仓库路径配置

- Default 数据仓库的最原始位置是在hdfs 上的：/user/hive/warehouse 路径下。
- 在仓库目录下，没有对默认的数据库 default 创建文件夹。如果某张表属于 default数据库，直接在数据仓库目录下创建一个文件夹。
- 修改 default 数据仓库原始位置（将 hive-default.xml.template 如下配置信息拷贝到hive-site.xml 文件中）

```
<property>
    <name>hive.metastore.warehouse.dir</name>
    <value>/user/hive/warehouse</value>
    <description>location   of  default warehouse</description>
</property>
```

查询后信息显示配置

- 在 hive-site.xml 文件中添加如下配置信息，就可以实现显示当前数据库，以及查询表的头信息配置

```
<!--打印字段 -->
<property>
    <name>hive.cli.print.header</name>
    <value>true</value>
    <description>Whether to print the names of the columns in query output.</description>
</property>

<!--显示使用的数据库名 -->
<property>
    <name>hive.cli.print.current.db</name>
    <value>true</value>
    <description>Whether to include the current database in the Hive prompt.</description>
</property>
```

运行日志信息配置

- Hive 的 log 默认存放在 `/tmp/admin/hive.log` 目录下（当前用户名下）
- 修改 hive 的 log 存放日志到 `/usr/local/software/hive-1.2.1/logs`
  - 修改 `/usr/local/software/hive-1.2.1/conf/hive-log4j.properties.template` 文件名称为 `hive-log4j.properties`
  - 在 `hive-log4j.properties` 文件中修改 log 存放位置

```
hive.log.dir=/usr/local/software/hive-1.2.1/logs
```

参数配置方式

查看配置信息

```
hive> set;
hive> set 参数;
```

参数的配置三种方式

- 配置文件方式

  默认配置文件：hive-default.xml

  用户自定义配置文件：hive-site.xml

  > **用户自定义配置会覆盖默认配置。**另外，Hive 也会读入Hadoop 的配置，因为 Hive 是作为 Hadoop 的客户端启动的，Hive 的配置会覆盖Hadoop 的配置。配置文件的设定对本机启动的所有Hive 进程都有效

- 命令行参数方式

  启动Hive 时，可以在命令行添加-hiveconf param=value 来设定参数

  ```
  bin/hive -hiveconf mapred.reduce.tasks=10
  ```

  > 仅对本次 hive 启动有效

- 参数声明方式

  ```
  hive (default)> set mapred.reduce.tasks=100;

  # 查看
  hive (default)> set mapred.reduce.tasks;
  ```

  > 仅对本次 hive 启动有效

> 上述三种设定方式的优先级依次递增。即配置文件<命令行参数<参数声明。注意某些系统级的参数，例如 log4j 相关的设定，必须用前两种方式设定，因为那些参数的读取在会话建立以前已经完成了。类似HDFS配置的优先等级

数据类型

基本数据类型

Hive 数据类型	Java 数据类型	长度	例子
TINYINT	byte	1byte 有符号整数	20
SMALINT	short	2byte 有符号整数	20
INT	int	4byte 有符号整数	20
BIGINT	long	8byte 有符号整数	20
BOOLEAN	boolean	布尔类型，true 或者 false	TRUE FALSE
FLOAT	float	单精度浮点数	3.14159
DOUBLE	double	双精度浮点数	3.14159
STRING	string	字符系列。可以指定字符集。可以使用单引号或者双引号。	‘now is the time’ “for allgood men”
TIMESTAMP		时间类型	
BINARY		字节数组	

> 对于Hive 的 String 类型相当于数据库的 varchar 类型，该类型是一个可变的字符串，不过它不能声明其中最多能存储多少个字符，理论上它可以存储 2GB 的字符数，不过通常一条日志长度为 2KB。

集合数据类型

数据类型	描述	语法示例
STRUCT	和c 语言中的 struct 类似，都可以通过“点”符号访问元素内容。例如，如果某个列的数据类型是 STRUCT{first STRING,last STRING},那么第 1 个元素可以通过字段.first 来引用。	struct()
MAP	MAP 是一组键-值对元组集合，使用数组表示法可以访问数据。例如，如果某个列的数据类型是 MAP，其中键->值对是’first’->’John’和’last’->’Doe’，那么可以通过字段名[‘last’]获取最后一个元素	map()
ARRAY	数组是一组具有相同类型和名称的变量的集合。这些变量称为数组的元素，每个数组元素都有一个编号，编号从零开始。例如，数组值为[‘John’, ‘Doe’]，那么第 2 个元素可以通过数组名[1]进行引用。	Array()

```
{
    "name": "songsong",
    "friends": ["bingbing" , "lili"] ,       //列表 Array,
    "children": {                     //键值 Map,
        "xiao song": 18 ,
        "xiaoxiao song": 19
    }
    "address": {                      //结构 Struct,
        "street": "hui long guan" ,
        "city": "beijing"
    }
}
```

```
# 数据，hive若想使用json数据，需要转成一行数据/条，并且设置分隔符
songsong,bingbing_lili,xiao song:18_xiaoxiao song:19,hui long guan_beijing
```

```
# 例子，建表语句
create table test(
    name string,
    friends array<string>,
    children map<string, int>,
    address struct<street:string, city:string>
)
row format delimited fields terminated by ','
collection items terminated by '_'
map keys terminated by ':'
lines terminated by '\n'; # lines的分隔符默认为'\n'，可不指定

# 访问方式
select friends[1], children['xiao song'], address.city from test;
```

类型转换

隐式类型转换

类似Java中的小转大，可以直接转换

- 任何整数类型都可以隐式地转换为一个范围更广的类型，如 TINYINT 可以转换成 INT，INT 可以转换成BIGINT。
- 所有整数类型、FLOAT 和 STRING 类型都可以隐式地转换成 DOUBLE。
- TINYINT、SMALLINT、INT 都可以转换为 FLOAT。
- BOOLEAN 类型不可以转换为任何其它的类型。

显式类型转换

使用 CAST 操作显示进行数据类型转换，CAST('1' AS INT)将把字符串'1' 转换成整数 1；如果强制类型转换失败，如执行CAST('X' AS INT)，表达式返回空值 NULL。

```
select cast('1' as int);
```

DDL数据定义

创建表

管理表（内部表）

默认创建的表都是所谓的管理表，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。**

外部表

用`external`关键字

因为表是外部表，所以 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。

```
create external table if not exists default.dept(
    deptno int,
    dname string,
    loc int
) row format delimited fields terminated by '\t';
```

管理表与外部表的使用场景

hive表的元数据信息存储在mysql数据库中（默认在derby），原始数据存储在hdfs中，hive sql语句操作时，是从mysql中读取元数据信息，得到了原始数据在hdsf中的路径，再去读取原始数据。内部表与外部表的主要区别就是：在删除表时，是否删除hdfs中的原始数据，而元数据信息都会删除掉（元数据删除了，自然就读取不到原始数据的路径了，也就查询不到）。

每天将收集到的网站日志定期流入HDFS 文本文件。在外部表（原始日志表）的基础上做大量的统计分析，用到的中间表、结果表使用内部表存储，数据通过 SELECT+INSERT 进入内部表

管理表与外部表的互相转换

- 查询表类型

```
desc formatted stu;
# Table Type:   EXTERNAL_TABLE(或 MANAGED_TABLE)
```

- 修改内部表 stu 为外部表

```
alter table student2 set tblproperties('EXTERNAL'='TRUE');
```

- 修改外部表 stu 为内部表

```
alter table student2 set tblproperties('EXTERNAL'='FALSE');
```

> ('EXTERNAL'='TRUE')和('EXTERNAL'='FALSE')为固定写法，区分大小写！
>
> 若小写，则表示加了一个新属性。

分区表

基本操作

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。

- 创建分区表语法

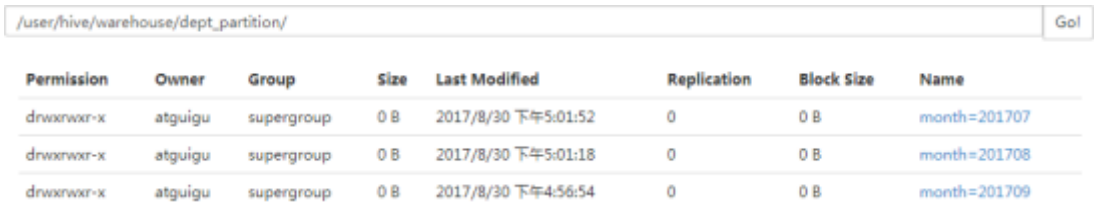
```
create table dept_partition(
    deptno int,
    dname string,
    loc string
)
partitioned by (month string);
# 额外指定了一个month字段
```

> 将month，当成普通字段使用即可，通常用于where子句，能提高查询效率。在hdfs存储的形式为，一个分区对应一个目录。
>
> 分区字段也可以指定表的字段名，但只是个名字，在下面的动态分区补充

- 加载数据，需要指定分区

```
load data local inpath 'datapath' into table dept_partition
partition(month='2020-01');
```

> 原先没有分区的话会先添加分区元数据信息，在添加数据进hdfs中



- 添加分区

```
alter table dept_partition add partition(month='2019-11')
partition(month='2019-12');
```

> 多个分区之间用空格隔开

- 删除分区

```
alter table dept_partition drop partition(month='2019-
11'),partition(month='2019-12');
```

> 多个分区之间用逗号隔开

- 查看分区表有多少分区

```
show partitions dept_partition;
```

- 查看分区表结构

```
desc formatted dept_partition;
# Partition Information
```

注意事项

创建二级分区表（多级同理）

- 创建

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

- 加载数据，需要指定分区

```
load data local inpath 'datapath' into table dept_partition2
partition(month='2020-01', day='01');
```

- 查询

```
select * from dept_partition2 where month='2020-01' and day='01';
```

数据修复

- 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

> 也就是先上传数据到hdfs中，但是没有该路径的元数据，hive sql语句查询时会查不到先上传数据

```
# 创建一个新的分区目录，hive中没有该元数据信息
hadoop fs -mkdir -p /user/hive/warehouse/dept_partition/2020-02
# 上传数据
hadoop fs -put ./data.txt /user/hive/warehouse/dept_partition/2020-02
```

  - 上传数据后修复，适用于多个分区目录没有元数据

```
msck repair table dept_partition2;
```

  - 上传数据后添加分区，适用于少数个分区目录没有元数据（常用）

```
alter table dept_partition add partition(month='2020-02');
```

  - load 数据到分区（少用）

```
load data inpath 'datapath' into table dept_partition
partition(month='2020-02');
```

分桶表

分桶表与分区表的区别

- 分区表针对的是数据的存储路径；分桶表针对的是数据文件。
- 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。

创建分桶表

创建分桶表时，数据通过子查询的方式导入

```
set hive.enforce.bucketing=true;
# 开启分桶参数
set mapreduce.job.reduces=-1;
# 设置reducer个数为又hive自定义，就算是设置为3，但实际的reducer个数是由桶的个数决定的

create table cc_buck(id int, name string, empno string)
clustered by(id) into 4 buckets
row format delimited fields terminated by '\t';
# 指定字段和桶数

insert into table cc_buck select * from cc;
# 同过子查询的方式导入数据到分桶表中
```

```
-rwxr-xr-x   3 admin supergroup         16 2020-01-15 11:57 /user/hive/warehouse/aa.db/cc_buck/000000_0
-rwxr-xr-x   3 admin supergroup         25 2020-01-15 11:57 /user/hive/warehouse/aa.db/cc_buck/000001_0
-rwxr-xr-x   3 admin supergroup         33 2020-01-15 11:57 /user/hive/warehouse/aa.db/cc_buck/000002_0
-rwxr-xr-x   3 admin supergroup         16 2020-01-15 11:57 /user/hive/warehouse/aa.db/cc_buck/000003_0
```

查询数据，查询结果为是按照id字段分桶（对桶数取余）

```
hive (aa)> select * from cc_buck;
OK
cc_buck.id      cc_buck.name    cc_buck.empno
8       tt      80
4       hh      20
13      rr      50
5       gg      10
1       aa      10
10      pp      60
2       qq      50
6       yy      30
2       bb      10
7       ll      50
3       cc      30
```

分桶抽样查询

```
select * from stu_buck tablesample(bucket 1 out of 4 on id);
# 语法 TABLESAMPLE(BUCKET x OUT OF y)
```

满足的条件：

- y 必须是 table 总 bucket 数的倍数或者因子，否则查询的结果没有规律
- x 的值必须小于等于 y 的值，否则报错

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

含义：

- x 决定从哪个 bucket 开始抽取
- y 决定了抽取多少个桶，由桶数(bucket) / y 得出取多少个桶

- 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 3(x+y)个 bucket 的数据。
- 若y = 8，则4/8=0.5，表示从第一个桶中取一半的数据

修改表

- 更新列

```
ALTER TABLE table_name CHANGE [COLUMN] col_old_name col_new_name
column_type [COMMENT col_comment] [FIRST|AFTER column_name]

# 例子
alter table dept_partition change column deptdesc desc int;
```

- 增加和替换列

```
ALTER TABLE table_name ADD|REPLACE COLUMNS (col_name data_type [COMMENT
col_comment], ...)

# 例子
# 添加deptdesc字段
alter table dept_partition add columns(deptdesc string);
# 将表的字段替换成deptno, dname, loc
alter table dept_partition replace columns(deptno string, dname string,
loc string);
```

> ADD 是代表新增一字段，字段位置在所有列后面(partition 列前)，REPLACE 则是表示替换表中所有字段。

删除表

```
drop table dept_partition;
```

> hive表的修改操作都是针对元数据修改的，不会修改hdfs中的原始数据
>
> Truncate 只能删除管理表，不能删除外部表中数据

DML数据操作

数据导入

load加载

```
load data [local] inpath '/opt/module/datas/student.txt' [overwrite] into
table student [partition (partcol1=val1,…)];
```

- load data:表示加载数据
- local:表示从本地加载数据到 hive 表；否则从HDFS 加载数据到 hive 表
- inpath:表示加载数据的路径
- overwrite:表示覆盖表中已有数据，否则表示追加
- into table:表示加载到哪张表
- student:表示具体的表
- partition:表示上传到指定分区

通过查询语句向表中插入数据

- 基本模式插入（根据单张表查询结果）

```
insert overwrite table student partition(month='2020-01')
    select id, name from student where month = '2020-01';
# overwrite为覆盖写， into为插入
# 将select语句的查询结果插入表中，且是分区表
```

- 多插入模式（根据多张表查询结果）

```
# 将from student提取出来了，也可以按正常方式写子句
from student
insert overwrite table student partition(month='2020-01')
    select id, name from student where month = '2020-01'
insert overwrite table student partition(month='2020-02')
    select id, name from student where month = '2020-02';
```

查询语句中创建表并加载数据

根据查询结果创建表（查询的结果会添加到新创建的表中）

```
create table if not exists student3
as select id, name from student;
```

建表时通过 Location 指定加载数据路径

创建表，并指定在 hdfs 上的位置

```
 create table if not exists student5(
     id int,
     name string
)
row format delimited fields terminated by '\t'
location '/user/hive/warehouse/student5';
# 将hdfs的路径信息绑定到student5的元数据
```

Import 数据到指定 Hive 表中（少用）

```
import table student2 partition(month='201709') from
'/user/hive/warehouse/export/student';

# 得先用 export 导出后，再将数据导入
```

数据导出

Insert导出

```
insert overwrite [local] directory 'datapath'
    [row format delimited fields ternimated by '\t']
    select * from student;


# local 表示的是将查询结果导出到本地目录中，'/home/admin/stu'
# 若没有local，则表示的是导出到hdfs，'/'
# row format 表示的是以'\t'为分隔符的形式导出，否则是行数据字段是连一起的，最好以创表时的分隔符来分隔
```

hive shell导出

利用linux > 导出符，将查询结果导出到本地文件中

```
bin/hive -e ‘select * from student’ > /opt/module/datas/export/student4.txt
```

Export导出到hdfs中

会在student目录下多个 _metedata 文件，存储元数据，用于import导入方式使用

```
hive(default)> export table default.student to
'/user/hive/warehouse/export/student';
```

清空表中数据

```
hive (default)> truncate table student;
```

> Truncate 只能删除管理表，不能删除外部表中数据
>
> 只删除hdfs中的数据，元数据、表结构信息不删除

查询

跟mysql一样，只列举一些不一样的

join

- Hive 支持通常的 SQL JOIN 语句，但是只支持等值连接，不支持非等值连接。

  ```
  hive (default)> select e.empno, e.ename, d.deptno, d.dname from emp e
  join dept d
  on e.deptno = d.deptno;
  # 不支持 > < != 等非等值连接
  ```

  > 同样支持内连接，左连接，右连接等

- 连接谓词中不支持or

```
hive (default)> select
              >    e.empno,
              >    e.ename,
              >    d.deptno
              > from
              >    emp e
              > join
              >    dept d
              > on
              >    e.deptno=d.deptno or e.ename=d.dname;
FAILED: SemanticException [Error 10019]: Line 10:3 OR not
supported in JOIN currently 'dname'
```

- 支持满外连接（full join Mysql不支持）

```
select e.empno, e.ename, d.deptno from emp e full join dept d on
e.deptno = d.deptno;
```

多表连接

```
SELECT e.ename, d.deptno, l.loc_name FROM emp e
JOIN dept d
ON d.deptno = e.deptno JOIN location l
ON d.loc = l.loc;
```

大多数情况下，Hive 会对每对 JOIN 连接对象启动一个 MapReduce 任务。本例中会首先启动一个 MapReduce job 对表e 和表 d 进行连接操作，然后会再启动一个 MapReduce job 将第一个 MapReduce job 的输出和表 l;进行连接操作。

排序

全局排序（order by）

全局排序，只会产生一个 Reducer

即使设置`mapreduce.job.reduces`参数，依然只会是1个reducer

每个 MapReduce 内部排序（sort by）

每个 Reducer 内部进行排序，对全局结果集来说不是排序。

> 采用随机分区，使得分区后的数据量尽量均匀
>
> hash分区容易导致数据倾斜

- 设置reduce 个数

```
set mapreduce.job.reduces=3;
```

- 将查询结果导入到文件中（按照部门编号降序排序）

```
insert overwrite local directory '/opt/module/datas/sortby-result'
select * from emp sort by deptno desc;
# 结束后结果目录下会有3（reducer个数）个文件，每个文件里的结果按照deptno降序排序显示
```

分区排序（distribute by）

类似 MR 中 partition，进行分区

通常结合 sort by 使用。

```
set mapreduce.job.reduces=3;

insert overwrite local directory '/opt/module/datas/sortby-result'
select * from emp
distribute by deptno
sort by salary desc;

# 按照deptno分区，默认是hash分区
# 在分区内按照salary降序排序
```

> Hive 要求 DISTRIBUTE BY 语句要写在SORT BY 语句之前。

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。但是排序只能是升序排序，不能指定排序规则为 ASC 或者 DESC。

```
hive (default)> select * from emp cluster by deptno;
hive (default)> select * from emp distribute by deptno sort by deptno;
# 结果等价
```

常用查询函数

空字段复制

NVL(s1, s2)

如果s1为 NULL，则 NVL 函数返回 s2的值，否则返回 s1的值，如果两个参数都为 NULL ，则返回 NULL。

时间类

- date_format:格式化时间

```
hive (default)> select date_format('2019-06-29','yyyy-MM-dd'); OK
_c0
2019-06-29
```

- date_add:时间跟天数相加 / date_sub:时间跟天数相减

```
hive (default)> select date_add('2019-06-29',5); OK
_c0
2019-07-04

hive (default)> select date_add('2019-06-29',-5); OK
_c0
2019-06-24
```

- datediff:两个时间相减

```
hive (default)> select datediff('2019-06-29','2019-06-24'); OK
_c0
5

hive (default)> select datediff('2019-06-24 12:12:12','2019-06-29'); OK
_c0
-5

hive (default)> select datediff('2019-06-24 12:12:12','2019-06-29
13:13:13');
OK
_c0
-5
```

> 时间类只能处理 '-' 分隔符的，其他格式的如'2020/01/15'不能处理，只能先将 '/' 转成 '-' 。
>
> regexp_replace(col, old_str, new_str)：将字段col中的old_str替换成new_str

case when

同Mysql用法

```
select dept_id,
sum(case sex when '男' then 1 else 0 end) male_count,
sum(case sex when '女' then 1 else 0 end) female_count
from emp_sex
group by dept_id;
# 求每个部门中的男女数量，输出如下，A B为dept_id
dept_id male_count  female_count
A   2   1
B   1   2
```

行转列

- 案例需求
  -

信息 结果1 状态

id	userid	subject	score
1	001	语文	90
2	001	数学	92
3	001	英语	80
4	002	语文	88
5	002	数学	90
6	002	英语	75.5
7	003	语文	70
8	003	数学	85
9	003	英语	90
10	003	政治	82

先来看一下转换后的结果：

信息 结果1 状态

userid	语文	数学	英语	政治
001	90	92	80	0
002	88	90	75.5	0
003	70	85	90	82

```
SELECT userid,
SUM(CASE `subject` WHEN '语文' THEN score ELSE 0 END) as '语文',
SUM(CASE `subject` WHEN '数学' THEN score ELSE 0 END) as '数学',
SUM(CASE `subject` WHEN '英语' THEN score ELSE 0 END) as '英语',
SUM(CASE `subject` WHEN '政治' THEN score ELSE 0 END) as '政治'
FROM tb_score
GROUP BY userid
# MAX函数也行
```

-

name	constellation	blood_type
孙悟空	白羊座	A
大海	射手座	A
宋宋	白羊座	B
猪八戒	白羊座	A
凤姐	射手座	A

3. 需求

把星座和血型一样的人归类到一起。结果如下：

```
射手座,A            大海|凤姐
白羊座,A            孙悟空|猪八戒
白羊座,B            宋宋
```

```
select t1.base,
concat_ws('|', collect_set(t1.name)) name from
(
    select name, concat(constellation, ",", blood_type) base
    from person_info
) t1
group by t1.base;
```

- concat(str1, str2...)

返回输入字符串连接后的结果，支持任意个输入字符串;

> 允许其他类型的参数连接

- concat_ws(sep, str1, str2...)

  它是一个特殊形式的 concat()。第一个参数是剩余参数间的分隔符。分隔符可以是与剩余参数一样的字符串。如果分隔符是 NULL， 返回值也将为 NULL。这个函数会跳过分隔符参数后的任何 NULL 和空字符串。分隔符将被加到被连接的字符串之间;

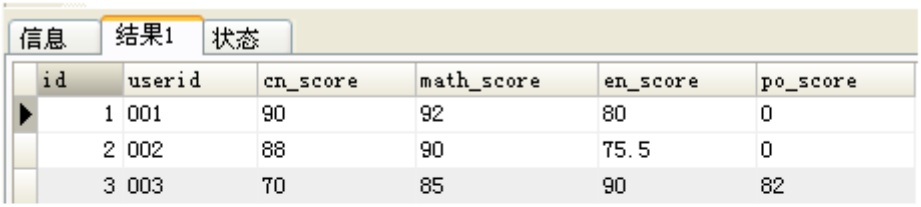
  > 可以用array类型当参数
  >
  > 只允许string类型的参数

- collect_set(col)

  函数只接受基本数据类型，它的主要作用是将某字段的值进行去重汇总，产生 array 类型字段。

  列转行

列转行

- 案例需求
  - 

id	userid	cn_score	math_score	en_score	po_score
1	001	90	92	80	0
2	002	88	90	75.5	0
3	003	70	85	90	82

    转换后：

userid	course	score
001	政治	0
001	英语	80
001	数学	92
001	语文	90
002	政治	0
002	英语	75.5
002	数学	90
002	语文	88
003	政治	82
003	英语	90
003	数学	85
003	语文	70

    ```sql
    SELECT userid,'语文' AS course,cn_score AS score FROM tb_score1
    UNION ALL
    SELECT userid,'数学' AS course,math_score AS score FROM tb_score1
    UNION ALL
    SELECT userid,'英语' AS course,en_score AS score FROM tb_score1
    UNION ALL
    SELECT userid,'政治' AS course,po_score AS score FROM tb_score1
    ORDER BY userid
    ```

  -

movie	category
《疑犯追踪》	悬疑,动作,科幻,剧情
《Lie to me》	悬疑,警匪,动作,心理,剧情
《战狼 2》	战争,动作,灾难

3. 需求

将电影分类中的数组数据展开。结果如下：

```
《疑犯追踪》      悬疑
《疑犯追踪》      动作
《疑犯追踪》      科幻
《疑犯追踪》      剧情
《Lie to me》   悬疑
《Lie to me》   警匪
《Lie to me》   动作
《Lie to me》   心理
《Lie to me》   剧情
《战狼 2》        战争
《战狼 2》        动作
《战狼 2》        灾难
```

```
select movie, category_name
from movie_info lateral view explode(category) table_tmp as category_name;
# category_name 是explode(category)的别名
```

- explode(col)

  将hive 一列中复杂的 array 或者 map 结构拆分成多行。

- lateral view

  用法：LATERAL VIEW udtf(expression) tableAlias AS columnAlias

  解释：用于和 split, explode 等UDTF 一起使用，它能够将一列数据拆成多行数据，在此基础上可以对拆分后的数据进行聚合。lateral view首先为原始表的每行调用UDTF，UTDF会把一行拆分成一或者多行，lateral view再把结果组合，产生一个支持别名表的虚拟表。

窗口函数

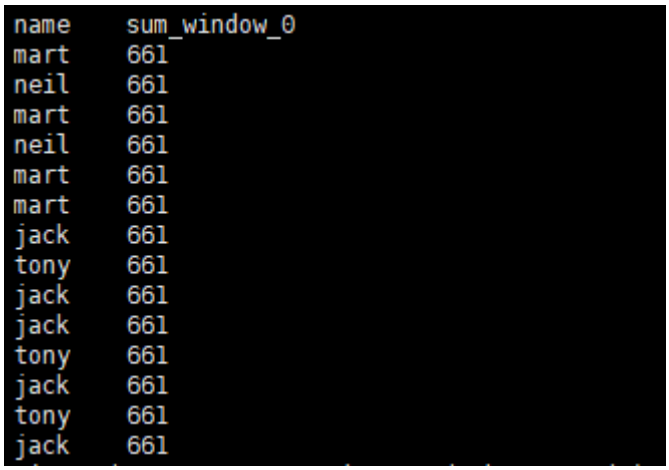
over()：指定分析函数工作的数据窗口大小，这个数据窗口大小可能会随着行的变化而变化；

用法：聚合函数 + over()，表示聚合函数只作用在窗口大小的范围内

原始数据：

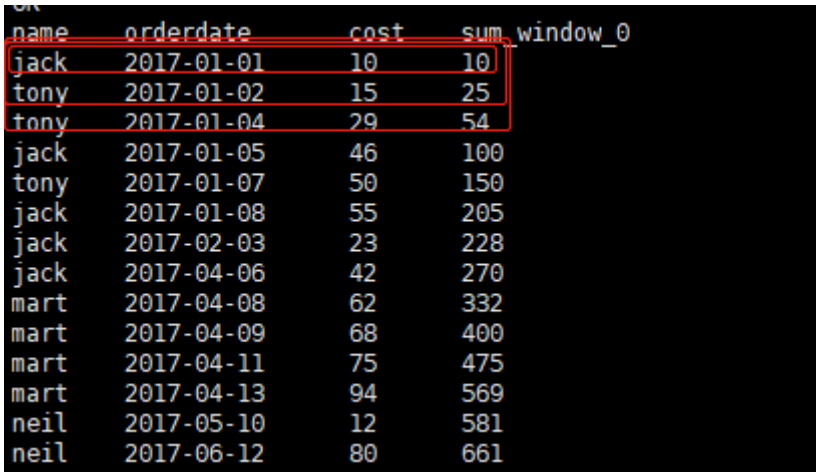
```
hive (default)> select * from business;
OK
business.name   business.orderdate      business.cost
jack    2017-01-01      10
tony    2017-01-02      15
jack    2017-02-03      23
tony    2017-01-04      29
jack    2017-01-05      46
jack    2017-04-06      42
tony    2017-01-07      50
jack    2017-01-08      55
mart    2017-04-08      62
mart    2017-04-09      68
neil    2017-05-10      12
mart    2017-04-11      75
neil    2017-06-12      80
mart    2017-04-13      94
```

```sql
# over()不带参数表示窗口大小为所有行
select name, sum(cost) over() from business;
```

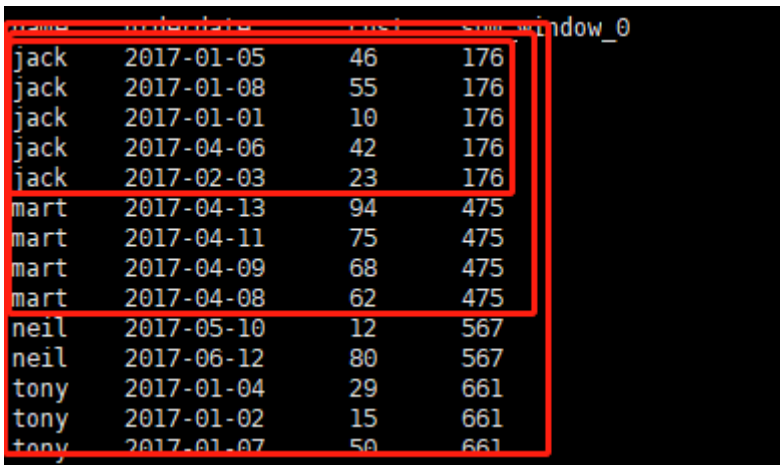


name	sum_window_0
mart	661
neil	661
mart	661
neil	661
mart	661
mart	661
jack	661
tony	661
jack	661
jack	661
tony	661
jack	661
tony	661
jack	661

```sql
# 排序起了累计的作用，窗口大小随着order by 后的字段的不同而变化
select name, orderdate, cost, sum(cost) over(order by orderdate) from
business;
select name, orderdate, cost, sum(cost) over(order by name) from business;
```

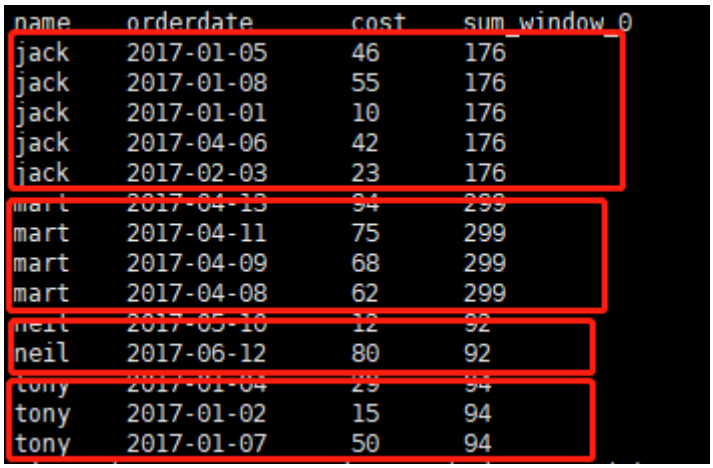


name	orderdate	cost	sum_window_0
jack	2017-01-01	10	10
tony	2017-01-02	15	25
tony	2017-01-04	29	54
jack	2017-01-05	46	100
tony	2017-01-07	50	150
jack	2017-01-08	55	205
jack	2017-02-03	23	228
jack	2017-04-06	42	270
mart	2017-04-08	62	332
mart	2017-04-09	68	400
mart	2017-04-11	75	475
mart	2017-04-13	94	569
neil	2017-05-10	12	581
neil	2017-06-12	80	661



[illegible]	[illegible]	[illegible]	[illegible]indow_0
jack	2017-01-05	46	176
jack	2017-01-08	55	176
jack	2017-01-01	10	176
jack	2017-04-06	42	176
jack	2017-02-03	23	176
mart	2017-04-13	94	475
mart	2017-04-11	75	475
mart	2017-04-09	68	475
mart	2017-04-08	62	475
neil	2017-05-10	12	567
neil	2017-06-12	80	567
tony	2017-01-04	29	661
tony	2017-01-02	15	661
tony	2017-01-07	50	661

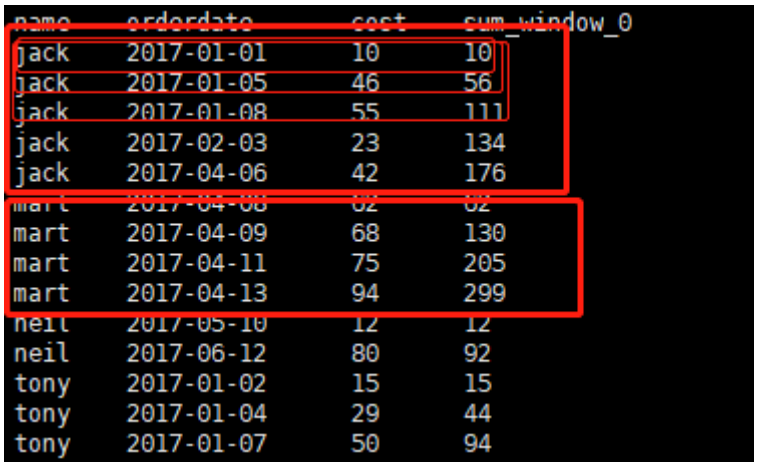
```
# over(distribute by name)表示每行数据的窗口大小为name相同的区域
select name, orderdate, cost, sum(cost) over(distribute by name) from
business;
```



name	orderdate	cost	sum_window_0
jack	2017-01-05	46	176
jack	2017-01-08	55	176
jack	2017-01-01	10	176
jack	2017-04-06	42	176
jack	2017-02-03	23	176
mart	2017-04-13	94	299
mart	2017-04-11	75	299
mart	2017-04-09	68	299
mart	2017-04-08	62	299
neil	2017-05-10	12	92
neil	2017-06-12	80	92
tony	2017-01-04	29	94
tony	2017-01-02	15	94
tony	2017-01-07	50	94

```
# 窗口大小为每个分区（并不是按照哈希值分区，而是数值相同的），且每个分区内的窗口大小逐渐变大，相当于做累加
# 用count(*)函数可以看得出来，排序起累计的作用

# distribute by ... sort by ... 和 partition by ... order by ...，但是组合不能乱
select name, orderdate, cost, sum(cost) over(distribute by name sort by
orderdate) from business;
```



name	orderdate	cost	sum_window_0
jack	2017-01-01	10	10
jack	2017-01-05	46	56
jack	2017-01-08	55	111
jack	2017-02-03	23	134
jack	2017-04-06	42	176
mart	2017-04-08	62	62
mart	2017-04-09	68	130
mart	2017-04-11	75	205
mart	2017-04-13	94	299
neil	2017-05-10	12	12
neil	2017-06-12	80	92
tony	2017-01-02	15	15
tony	2017-01-04	29	44
tony	2017-01-07	50	94

over()里的字段表示：

- current row：当前行
- n preceding：往前n行数据
- n following：往后n行数据
- unbounded：无边界
  - unbounded preceding：表示数据的起点，第一行
  - unbounded following：表示数据的终点，最后一行

```
# 窗口大小为 当前行 前一行 后一行，总共3行
# 也可结合着distribute by等子句使用
select name, orderdate, cost, sum(cost) over(rows between 1 preceding and 1 following) from business;
```

name	orderdate	cost	sum_window_0
mart	2017-04-13	94	174
neil	2017-06-12	80	249
mart	2017-04-11	75	167
neil	2017-05-10	12	155
mart	2017-04-09	68	142
mart	2017-04-08	62	185
jack	2017-01-08	55	167
tony	2017-01-07	50	147
jack	2017-04-06	42	138
jack	2017-01-05	46	117
tony	2017-01-04	29	98
jack	2017-02-03	23	67
tony	2017-01-02	15	48
jack	2017-01-01	10	25

lag(col, n, default)：字段col的往前第n行数据

lead(col, n, default)：字段col的往后第n行数据

> default为，将NULl值替换为default，可忽略不写

```
# 后面得跟着over()函数，且可以指定窗口大小
select *, lag(orderdate, 1) over() from business;
```

business.name	business.orderdate	business.cost	lag_window_0
mart	2017-04-13	94	NULL
neil	2017-06-12	80	2017-04-13
mart	2017-04-11	75	2017-06-12
neil	2017-05-10	12	2017-04-11
mart	2017-04-09	68	2017-05-10
mart	2017-04-08	62	2017-04-09
jack	2017-01-08	55	2017-04-08
tony	2017-01-07	50	2017-01-08
jack	2017-04-06	42	2017-01-07
jack	2017-01-05	46	2017-04-06
tony	2017-01-04	29	2017-01-05
jack	2017-02-03	23	2017-01-04
tony	2017-01-02	15	2017-02-03
jack	2017-01-01	10	2017-01-02

```
# 设置窗口大小
select * , lag(orderdate, 1) over(distribute by name sort by orderdate) from business;
```

jack	2017-01-01	10	NULL
jack	2017-01-05	46	2017-01-01
jack	2017-01-08	55	2017-01-05
jack	2017-02-03	23	2017-01-08
jack	2017-04-06	42	2017-02-03
mart	2017-04-08	62	NULL
mart	2017-04-09	68	2017-04-08
mart	2017-04-11	75	2017-04-09
mart	2017-04-13	94	2017-04-11
neil	2017-05-10	12	NULL
neil	2017-06-12	80	2017-05-10
tony	2017-01-02	15	NULL
tony	2017-01-04	29	2017-01-02
tony	2017-01-07	50	2017-01-04

ntile(n)

```
# 按照排序将数据分成n 份
select name,orderdate,cost, ntile(5) over(order by orderdate) sorted from
business;
```

name	orderdate	cost	sorted
jack	2017-01-01	10	1
tony	2017-01-02	15	1
tony	2017-01-04	29	1
jack	2017-01-05	46	2
tony	2017-01-07	50	2
jack	2017-01-08	55	2
jack	2017-02-03	23	3
jack	2017-04-06	42	3
mart	2017-04-08	62	3
mart	2017-04-09	68	4
mart	2017-04-11	75	4
mart	2017-04-13	94	4
neil	2017-05-10	12	5
neil	2017-06-12	80	5

```
# 取前20% ， 也就是组号为1的；前40%，则是组号为1和2的
select * from (
select name,orderdate,cost, ntile(5) over(order by orderdate) sorted
from business
) t
where sorted = 1;
```

Rank

- RANK() 排序相同时会重复，总数不会变
- DENSE_RANK() 排序相同时会重复，总数会减少
- ROW_NUMBER() 会根据顺序计算

> rank()同样是窗口函数，后面需要跟over()

```sql
# 按学科分区，按成绩排序，根据over里指定排序的字段来排名
select
name,
subject,
score,
rank() over(partition by subject order by score desc) rp,
dense_rank() over(partition by subject order by score desc) drp,
row_number() over(partition by subject order by score desc) rmp
from score;
```

name	subject	score	rp	drp	rmp
孙悟空	数学	95	1	1	1
宋宋	数学	86	2	2	2
婷婷	数学	85	3	3	3
大海	数学	56	4	4	4
宋宋	英语	84	1	1	1
大海	英语	84	1	1	2
婷婷	英语	78	3	2	3
孙悟空	英语	68	4	3	4
大海	语文	94	1	1	1
孙悟空	语文	87	2	2	2
婷婷	语文	65	3	3	3
宋宋	语文	64	4	4	4

案例

表：

```
记录了用户每天的蚂蚁森林低碳生活领取的记录流水
table_name：user_low_carbon
user_id data_dt  low_carbon
用户      日期       减少碳排放（g）

蚂蚁森林植物换购表，用于记录申领环保植物所需要减少的碳排放量
table_name:  plant_carbon
plant_id plant_name low_carbon
植物编号     植物名  换购植物所需要的碳
```

第一题

```sql
#1.蚂蚁森林植物申领统计
#问题：假设2017年1月1日开始记录低碳数据（user_low_carbon），假设2017年10月1日之前满足申领条件的用户#都申领了一颗p004-胡杨，
#剩余的能量全部用来领取“p002-沙柳” 。
#统计在10月1日累计申领“p002-沙柳” 排名前10的用户信息；以及他比后一名多领了几颗沙柳。
#得到的统计结果如下表样式：
user_id  plant_count less_count(比后一名多领了几颗沙柳)
u_101    1000          100
u_088    900           400
u_103    500           …

select user_id, sum(u.low_carbon) sum_carbon
from user_low_carbon u
```

```
group by user_id; t1

# 领取胡杨后还剩余的排碳量
select t1.user_id, t1.sum_carbon - p.low_carbon left_carbon
from () t1, plant_carbon p
where p.plant_id='p004'; t2

# 计算每位用户能领取的沙柳
select t2.user_id, t2.left_carbon div p.low_carbon as plant_count
from ()t2, plant_carbon p
where p.plant_id='p002'; t3

# 计算比后一名多领了几颗沙柳
select user_id, plant_count, plant_count-lead(plant_count, 1, 0) over(order
by plant_count desc) less_count
from (select t2.user_id, t2.left_carbon div p.low_carbon as plant_count
from (select t1.user_id, t1.sum_carbon - p.low_carbon left_carbon
from (select user_id, sum(u.low_carbon) sum_carbon
from user_low_carbon u group by user_id
) t1, plant_carbon p
where p.plant_id='p004')t2, plant_carbon p
where p.plant_id='p002') t3
limit 10;
```

第二题

```
#2、蚂蚁森林低碳用户排名分析
#问题：查询user_low_carbon表中每日流水记录，条件为：
#用户在2017年，连续三天（或以上）的天数里，
#每天减少碳排放（low_carbon）都超过100g的用户低碳流水。
#需要查询返回满足以上条件的user_low_carbon表中的记录流水。
#例如用户u_002符合条件的记录如下，因为2017/1/2~2017/1/5连续四天的碳排放量之和都大于等
于100g：

#求出2017年超过100g的用户&时间
select user_id, data_dt
from user_low_carbon u
where substring(data_dt, 1, 4) = '2017'
group by user_id, data_dt
having sum(low_carbon) >= 100; t1

#给出每位用户按时间的排名
select user_id, data_dt, row_number() over(partition by user_id order by
data_dt) rk
from () t1; t2

#将时间减去排名，时间需要转换成合法的'-'分隔符
select user_id, data_dt, date_sub(date_format(regexp_replace(data_dt, '/',
'-'), 'yyyy-MM-dd'), rk) diff
from () t2; t3

#求出连续超过3天的，即每位用户的diff相同数超过3个
select user_id, data_dt, count(*) over(partition by user_id, diff)
valid_days
from () t3; t4

select user_id, data_dt
```

```
from (select user_id, data_dt, count(*) over(partition by user_id, diff) valid_days
from (select user_id, data_dt, date_sub(date_format(regexp_replace(data_dt, '/', '-'), 'yyyy-MM-dd'), rk) diff
from (select user_id, data_dt, row_number() over(partition by user_id order by data_dt) rk
from (select user_id, data_dt
from user_low_carbon u
where substring(data_dt, 1, 4) = '2017'
group by user_id, data_dt
having sum(low_carbon) >= 100) t1) t2) t3) t4
where valid_days >= 3
order by user_id, data_dt;

u_002   2017/1/2
u_002   2017/1/3
u_002   2017/1/4
...
u_013   2017/1/5
u_014   2017/1/5
u_014   2017/1/6
u_014   2017/1/7
```

自定义函数

- 创建一个maven工程
- 导入依赖

```
<dependencies>
    <!--https://mvnrepository.com/artifact/org.apache.hive/hive-exec -->
    <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>1.2.1</version>
    </dependency>
</dependencies>
```

- 创建一个类

```
// 继承UDF，实现evaluta方法，每次处理一行数据，并返回结果
// evaluta方法并不是重写，并且名字是固定的
// 可以重载方法，实现多参数调用
package com.zzh.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
public class MyUDF extends UDF {
    public String evaluate(String s) {
        return null;
    }
}

// 自定义UDTF
public class MyUDTF extends GenericUDTF {
    String[] dataOutPut = new String[1];

    // 定义输出数据的列名和输出类型
```

```
    @Override
    public StructObjectInspector initialize(StructObjectInspector argOIs) throws UDFArgumentException {
        // 想定义输出几列就添加几个列名
        List<String> fieldNames = new ArrayList<>();
        fieldNames.add("word");
//        fieldNames.add("number");

        // 关联输出列的数据类型
        List<ObjectInspector> fieldOIs = new ArrayList<>();

 fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
//
fieldOIs.add(PrimitiveObjectInspectorFactory.javaIntObjectInspector);

        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
    }

    @Override
    public void process(Object[] args) throws HiveException {
        // args是输入进来的参数
        String line = args[0].toString();
        String sep1 = args[1].toString();

        String[] words = line.split(sep1);
        for (String word : words) {
            dataOutPut[0] = word;
            // 要指定集合格式输出
            forward(dataOutPut);
        }

    }

    @Override
    public void close() throws HiveException {}
}
```

- 打成 jar 包上传到服务器

  通常上传到hive的lib目录下，在hive启动的时候回自动加载lib目录下jar包，而不用手动的去添加关联

- 将 jar 包添加到hive 的 classpath

```
# 若是jar包在lib下的话，可以不用添加，除非是客户端启动的时候添加的jar包就需要更新以下
hive (default)> add jar /usr/local/software/hive-1.2.1/lib/udf.jar;
```

- 创建临时函数与开发好的 java class 关联

```
# 语法
# temporary：只在当前hive client生效，一次性的
# dbname：指定调用范围，只能在dname里调用，省略则表示在当前数据库下
# function_name：函数名称
# class_name：全类名，如com.zzh.udf.MyUDF
create [temporary] function [dbname.]function_name AS class_name;
```

> UDAF UDTF同理实现上传

- 删除函数

```
drop [temporary] function [if exists] [dbname.]function_name;
```

压缩和存储

文件存储格式

- 行式存储

  查询满足条件的一整行数据的时候，列存储则需要去每个聚集的字段找到对应的每个列的值，行存储只需要找到其中一个值，其余的值都在相邻地方，所以此时行存储查询的速度更快。

- 列式存储

  因为每个字段的数据聚集存储，在查询只需要少数几个字段的时候，能大大减少读取的数据量；每个字段的数据类型一定是相同的，列式存储可以针对性的设计更好的设计压缩算法。

> textfile和 sequencefile的存储格式都是基于行存储的；
>
> orc 和parquet是基于列式存储的。结合了行式存储，可自行了解
>
> orc格式自带压缩
>
> orc、parquet在导入数据时通过insert子句，将子查询的结果导入

调优

Fetch抓取

Hive 中对某些情况的查询可以不必使用 MapReduce 计算

在 `hive-default.xml.template` 文件中 `hive.fetch.task.conversion` 默认是 more，老版本 hive 默认是 minimal，只对limit查找不走mapreduce。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。

```
# 设置属性
# none模式全部走mapreduce
# minimal模式，只有limit子句不走mapreduce
# more模式，下面3种情况不走mapreduce
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;
```

本地模式

在Hive 的输入数据量非常小的情况下，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。

```
#开启本地 mr
set hive.exec.mode.local.auto=true;
#设置 local mr 的最大输入数据量，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，默认为 4
set hive.exec.mode.local.auto.input.files.max=10;
# 开启本地模式需要同时满足以上两个条件，数据大小和操作的文件个数都要小于设置的值
```

表优化

- 小表join大表

  > 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。
  >
  > set hive.auto.convert.join = true;
  >
  > 自动开启了mapjoin

- 大表join大表

  空 KEY 过滤 / 转换

  表中过多的null值会在Reducer端分到一组造成数据倾斜，所以要么过滤掉（一般不可取，除非整行都是空），要么转换为随机值，均匀地分配到reduce task中

  - 过滤空key

    ```
    # a 为大表，b为小表
    select *
    from b join (select * from a where a is not null) a
    on b.id = a.id;
    ```

  - 转换空key

```sql
select *
from b join a
on b.id = (case when a.id is null then rand() else a.id) end;
```

- mapjoin

  由参数`hive.auto.convert.join`开启，默认是开启的

  参数`hive.mapjoin.smalltable.filesize`决定了小表的上限值，默认为25000000，大约25M

  开启的情况下，大表join小表时会自动mapjoin，当内存足够时，可以调大小表的上限值来调优

GroupBy

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。

开启 Map 端聚合参数设置

- 是否在 Map 端进行聚合，默认为True（`hive.map.aggr`）
- 在 Map 端进行聚合操作的条目数目，默认为100000（`hive.groupby.mapaggr.checkinterval`）
- 有数据倾斜的时候进行负载均衡，默认是 false，需要的时候再开启（`hive.groupby.skewindata`）

  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。

  相当于是先将key加个随机值，均匀的分配到各个reduce中部分聚合，再通过另一个job将随机值去掉后，实现整体聚合。

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。

替换了之后执行数据不一定变快，但是在一个reduce完成不了的情况下，可以将任务量分配给其他reduce来完成

```sql
select count(distinct id) from bigtable;
# 替换
select count(id) from (select id from bigtable group by id) a;
```

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。通常通过参数将笛卡尔积关闭，在使用笛卡尔积时报错。

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。建议先过滤再关联。

```
# 先关联再过滤
select * from a join b on a.id = b.id where a.id > 10;
# 先过滤再关联（推荐使用）
select * from (select id from a where a.id > 10) join b on a.id = b.id;
```

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。

若不配置动态分区，则需要人为地静态指定好分区，不方便。

- 开启动态分区功能（默认 true，开启）

  `hive.exec.dynamic.partition`

- 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）

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

- 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，默认值为1000

  `hive.exec.max.dynamic.partitions`

- 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`

- 整个MR Job 中，最大可以创建多少个HDFS 文，默认值为100000

  `hive.exec.max.created.files`

- 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`

案例：

- 建表

```
# 动态分区建表时不需要指定分区的具体值
create table dept_par(dname string, loc int) partitioned by(deptno int)
row format delimited fields terminated by '\t';

# 建普通表
create table dept(deptno int, dname string, loc int) row format
delimited fields terminated by '\t';
```

- 将dept的数据插入到dept_par

```
# 注意字段位置，分区表默认是最后的字段是分区字段
# 所以添加数据时，得注意字段的位置
insert into dept_par partition(deptno) select dname, loc, deptno from
dept;
```

```
hive (default)> select * from dept;
OK
dept.deptno     dept.dname      dept.loc
10      ACCOUNTING      1700
20      RESEARCH        1800
30      SALES   1900
40      OPERATIONS      1700
```

```
hive (default)> select * from dept_par;
OK
dept_par.dname  dept_par.loc    dept_par.deptno
ACCOUNTING      1700    10
RESEARCH        1800    20
SALES   1900    30
OPERATIONS      1700    40
```

```
hive (default)> show partitions dept_par;
OK
partition
deptno=10
deptno=20
deptno=30
deptno=40
Time taken: 0.108 seconds, Fetched: 4 row(s)
```

分区&分桶

前面已提到

MR优化

合理设置map数

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。

根据公式$Math.max(minSize, Math.min(maxSize, blocksize))$来设置块大小

其中

`mapreduce.input.fileinputformat.split.maxsize`设置maxsize，默认值为1;

`mapreduce.input.fileinputformat.split.minsize`设置minsize，默认值为256000000，大约256MB;

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）。

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

合理设置reduce数

- 方法一
  - 每个Reduce 处理的数据量默认是 256MB

    `hive.exec.reducers.bytes.per.reducer`
  - 每个任务最大的 reduce 数，默认为 1009

    `hive.exec.reducers.max`

    - 计算 reducer 数的公式，当`mapreduce.job.reduces=-1`时

      $$N = min(参数2，总输入数据量/参数1)$$

- 方法二

  在 hadoop 的 mapred-default.xml 文件中修改

  或者在hive客户端里设置参数`mapreduce.job.reduces`修改

> map数跟reduce数过多时，且处理文件过小，则jvm的开启关闭时间都长于任务的执行时间，此时则可以考虑jvm重用来进行优化。
>
> `mapred.job.reuse.jvm.num.tasks`，默认值为1

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

- 打开任务并行执行，默认为fasle

  `hive.exec.parallel`

- 同一个sql允许最大并行度，默认为8

  `hive.exec.parallel.thread.number`

严格模式

在严格模式下，以下几种情况不能执行：

- 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。

- 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。

- 限制笛卡尔积的查询。

  在执行join语句时，Hive 的优化器不可以将WHERE 语句转化成ON 语句，而关系数据库可以高效地完成。

- 不允许比较`bigint`和`string`、`bigint`和`double`。

通过参数`hive.mapred.mode`来开启，默认值为nonstrict，需要设置为strict。

推测执行

原理同yarn的推测执行，是为了防止一个job中，少数的task由于机器、网络等原因执行慢而重新备份执行一个task。最终选用成功执行的task的结果。

如果数据量本身就大，或者有数据倾斜的情况，就关掉。

压缩

执行计划 Explain