大数据离线阶段

08

一、 课程计划

目录

二、 Hive 函数高阶

1. UDTF 之 explode 函数

1.1. explode 语法功能

对于UDTF表生成函数，很多人难以理解什么叫做输入一行，输出多行。

为什么叫做表生成？能够产生表吗？下面我们就来学习Hive当做内置的一个非常著名的UDTF函数，名字叫做explode函数，中文戏称之为"爆炸函数"，可以炸开数据。

explode函数接收map或者array类型的数据作为参数，然后把参数中的每个元素炸开变成一行数据。一个元素一行。这样的效果正好满足于输入一行输出多行。

explode函数在关系型数据库中本身是不该出现的。

因为他的出现本身就是在操作不满足第一范式的数据（每个属性都不可再分）。本身已经违背了数据库的设计原理，但是在面向分析的数据库或者数据仓库中，这些规范可以发生改变。

```
explode(a) - separates the elements of array a into multiple
rows, or the elements of a map into multiple rows and columns
```

```
describe function extended explode;
```

3 rows

	tab_name
1	explode(a) - separates the elements of array a into multiple rows, or the element
2	Function class:org.apache.hadoop.hive.ql.udf.generic.GenericUDTFExplode
3	Function type:BUILTIN

explode(array)将array列表里的每个元素生成一行；

explode(map)将map里的每一对元素作为一行，其中key为一列，value为一列；

一般情况下，explode函数可以直接使用即可，也可以根据需要结合lateral view侧视图使用。

1.2. explode 函数的使用

```
select explode(`array`(11,22,33)) as item;

select explode(`map`("id",10086,"name","zhangsan","age",18));
```

```
select explode(`array`(11,22,33)) as item;
```

3 rows

	item
1	11
2	22
3	33

```
select explode(`map`("id",10086,"name","zhangsan","age",18));
```

3 rows

	key	value
1	id	10086
2	name	zhangsan
3	age	18

1.3. 案例：NBA 总冠军球队名单

业务需求

有一份数据《The_NBA_Championship.txt》，关于部分年份的NBA总冠军球队名单：

```
Chicago Bulls,1991|1992|1993|1996|1997|1998
San Antonio Spurs,1999|2003|2005|2007|2014
Golden State Warriors,1947|1956|1975|2015
Boston Celtics,1957|1959|1960|1961|1962|1963
L.A. Lakers,1949|1950|1952|1953|1954|1972|198
Miami Heat,2006|2012|2013
Philadelphia 76ers,1955|1967|1983
Detroit Pistons,1989|1990|2004
```

第一个字段表示的是球队名称，第二个字段是获取总冠军的年份，字段之间以，分割；

获取总冠军年份之间以|进行分割。

需求：使用Hive建表映射成功数据，对数据拆分，要求拆分之后数据如下所示：

```
Chicago Bulls 1991
Chicago Bulls 1992
Chicago Bulls 1993
Chicago Bulls 1996
Chicago Bulls 1997
Chicago Bulls 1998
```

并且最好根据年份的倒序进行排序。

代码实现

```
--step1:建表
create table the_nba_championship(
    team_name string,
    champion_year array<string>
) row format delimited
fields terminated by ','
collection items terminated by '|';

--step2:加载数据文件到表中
load data local inpath
'/root/hivedata/The_NBA_Championship.txt' into table
the_nba_championship;

--step3:验证
select *
from the_nba_championship;
```

	team_name	champion_year
1	Chicago Bulls	["1991","1992","1993","1996","1997","1998"]
2	San Antonio Spurs	["1999","2003","2005","2007","2014"]
3	Golden State Warriors	["1947","1956","1975","2015"]
4	Boston Celtics	["1957","1959","1960","1961","1962","1963","1964","1965","1966
5	L.A. Lakers	["1949","1950","1952","1953","1954","1972","1980","1982","1985
6	Miami Heat	["2006","2012","2013"]
7	Philadelphia 76ers	["1955","1967","1983"]
8	Detroit Pistons	["1989","1990","2004"]

下面使用explode函数：

```
--step4:使用 explode 函数对 champion_year 进行拆分 俗称炸开
select explode(champion_year) from the_nba_championship;

select team_name,explode(champion_year) from
the_nba_championship;
```

explode使用限制

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；

但是如果在select条件中，包含explode和其他字段，就会报错。错误信息为：

org.apache.hadoop.hive.ql.parse.SemanticException:UDTF's are not supported outside the SELECT clause, nor nested in expressions

那么如何理解这个错误呢？为什么在select的时候，explode的旁边不支持其他字段的同时出现？

explode语法限制原因

1、 explode函数属于UDTF函数，即表生成函数；

2、 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；

3、 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

4、 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

5、 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；

6、 从SQL层面上来说应该对两张表进行关联查询

7、 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。

```
--根据年份倒序排序
select a.team_name ,b.year
from the_nba_championship a lateral view explode(champion_year) b as year
order by b.year desc;
```

61 rows

	team_name	year
1	Golden State Warriors	2015
2	San Antonio Spurs	2014
3	Miami Heat	2013
4	Miami Heat	2012
5	L.A. Lakers	2010

2. Lateral View 侧视图

2.1. 概念

Lateral View是一种特殊的语法，主要用于搭配UDTF类型功能的函数一起使用，用于解决UDTF函数的一些查询限制的问题。

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。这样就避免了UDTF的使用限制问题。使用lateral view时也可以对UDTF产生的记录设置字段名称，产生的字段可以用于group by、order by 、limit等语句中，不需要再单独嵌套一层子查询。

一般只要使用UDTF，就会固定搭配lateral view使用。

官方链接：

https://cwiki.apache.org/confluence/display/Hive/LanguageManual+LateralView

2.2. UDTF 配合侧视图使用

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：

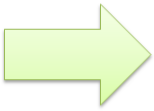
```
--lateral view 侧视图基本语法如下
select …… from tabelA lateral view UDTF(xxx) 别名 as col1,col2,col3……;

select a.team_name ,b.year
from the_nba_championship a lateral view explode(champion_year) b as year

--根据年份倒序排序
select a.team_name ,b.year
from the_nba_championship a lateral view explode(champion_year) b as year
order by b.year desc;
```

3. 行列转换应用与实现

3.1. 工作应用场景

实际工作场景中经常需要实现对于Hive中的表进行行列转换操作，例如统计得到每个小时不同维度下的UV、PV、IP的个数，而现在为了构建可视化报表，得到每个小时的UV、PV的线图，观察访问趋势，我们需要构建如下的表结构：

小时	UV	PV	IP
00	100	200	50
01	101	201	51
02	102	202	52
03	103	203	53
04	104	204	54
05	105	205	55
06	106	206	56
07	107	207	57
08	108	208	58
09	109	209	59
10	110	210	60
11	111	211	61
12	112	212	62
13	113	213	63
14	114	214	64
15	115	215	65
16	116	216	66
17	117	217	67
18	118	218	68
19	119	219	69
20	120	220	70
21	121	221	71
22	122	222	72
23	123	223	73



指标	00	01	02	03	04	……	21	22	23
UV	100	101	102	103	104	……	121	122	123
PV	200	201	202	203	204	……	221	222	223
IP	50	51	52	53	54	……	71	72	73

在Hive中，我们可以通过函数来实现各种复杂的行列转换。

3.2. 行转列：多行转单列

需求

- 原始数据表

col1	col2	col3
a	b	1
a	b	2
a	b	3
c	d	4
c	d	5
c	d	6

- 目标数据表

col1	col2	col3
a	b	1, 2, 3
c	d	4, 5, 6

concat

- 功能：用于实现字符串拼接，不可指定分隔符
- 语法

```
concat(element1,element2,element3……)
```

- 测试

```
select concat("it","cast","And","heima");
+-----------------+
| itcastAndheima  |
+-----------------+
```

- 特点：如果任意一个元素为null，结果就为null

```
select concat("it","cast","And",null);
```

concat_ws

- 功能：用于实现字符串拼接，可以指定分隔符
- 语法

```
concat_ws(SplitChar，element1，element2.....)
```

- 测试

```
select concat_ws("-","itcast","And","heima");
+-------------------+
| itcast-And-heima  |
+-------------------+
```

- 特点：任意一个元素不为null，结果就不为null

```
select concat_ws("-","itcast","And",null);
+-------------+
| itcast-And  |
+-------------+
```

collect_list

- 功能：用于将一列中的多行合并为一行，不进行去重
- 语法

```
collect_list ( colName )
```

- 测试

```
select collect_list(col1) from row2col1;
+---------------------------+
| ["a","a","a","b","b","b"] |
+---------------------------+
```

concat_set

- 功能：用于将一列中的多行合并为一行，并进行去重
- 语法

```
collect_set ( colName )
```

- 测试

```
select collect_set(col1) from row2col1;
+------------+
| ["b","a"]  |
+------------+
```

实现

- 创建原始数据表，加载数据

```
--建表
create table row2col2(
   col1 string,
   col2 string,
   col3 int
)row format delimited fields terminated by '\t';

--加载数据到表中
load data local inpath '/export/data/r2c2.txt' into table row2col2;
```

- SQL实现转换

```
select
  col1,
  col2,
  concat_ws(',', collect_list(cast(col3 as string))) as col3
from
  row2col2
group by
  col1, col2;
```

```
+-------+-------+--------+
| col1  | col2  |  col3  |
+-------+-------+--------+
| a     | b     | 1,2,3  |
| c     | d     | 4,5,6  |
+-------+-------+--------+
```

3.3. 列转行：单列转多行

需求

- 原始数据表

col1	col2	col3
a	b	1, 2, 3
c	d	4, 5, 6

- 目标结果表

col1	col2	col3
a	b	1
a	b	2
a	b	3
c	d	4
c	d	5
c	d	6

explode

- 功能：用于将一个集合或者数组中的每个元素展开，将每个元素变成一行
- 语法

```
explode( Map | Array)
```

- 测试

```
select explode(split("a,b,c,d",","));
```

```
+------+
| col  |
+------+
| a    |
| b    |
| c    |
| d    |
+------+
```

实现

- 创建原始数据表，加载数据

```
--切换数据库
use db_function;

--创建表
create table col2row2(
   col1 string,
   col2 string,
   col3 string
)row format delimited fields terminated by '\t';


--加载数据
load data local inpath '/export/data/c2r2.txt' into table col2row2;
```

- SQL实现转换

```
select
  col1,
  col2,
  lv.col3 as col3
from
  col2row2
    lateral view
  explode(split(col3, ',')) lv as col3;
```

```
+-------+-------+-------+
| col1  | col2  | col3  |
+-------+-------+-------+
| a     | b     | 1     |
| a     | b     | 2     |
| a     | b     | 3     |
| c     | d     | 4     |
| c     | d     | 5     |
| c     | d     | 6     |
+-------+-------+-------+
```

4. JSON 数据处理

4.1. 应用场景

JSON数据格式是数据存储及数据处理中最常见的结构化数据格式之一，很多场景下公司都会将数据以JSON格式存储在HDFS中，当构建数据仓库时，需要对JSON格式的数据进行处理和分析，那么就需要在Hive中对JSON格式的数据进行解析读取。

例如，当前我们JSON格式的数据如下：

```
{"device":"device_30","deviceType":"kafka","signal":98.0,"time":1616817201390}
{"device":"device_40","deviceType":"route","signal":99.0,"time":1616817201887}
{"device":"device_21","deviceType":"bigdata","signal":77.0,"time":1616817202142}
{"device":"device_31","deviceType":"kafka","signal":98.0,"time":1616817202405}
{"device":"device_20","deviceType":"bigdata","signal":12.0,"time":1616817202513}
{"device":"device_54","deviceType":"bigdata","signal":14.0,"time":1616817202913}
{"device":"device_10","deviceType":"db","signal":39.0,"time":1616817203356}
{"device":"device_94","deviceType":"bigdata","signal":59.0,"time":1616817203771}
{"device":"device_32","deviceType":"kafka","signal":52.0,"time":1616817204010}
{"device":"device_21","deviceType":"bigdata","signal":85.0,"time":1616817204229}
{"device":"device_74","deviceType":"bigdata","signal":27.0,"time":1616817204720}
{"device":"device_91","deviceType":"bigdata","signal":50.0,"time":1616817205164}
{"device":"device_62","deviceType":"db","signal":89.0,"time":1616817205328}
{"device":"device_21","deviceType":"bigdata","signal":25.0,"time":1616817205457}
{"device":"device_76","deviceType":"bigdata","signal":62.0,"time":1616817205984}
{"device":"device_74","deviceType":"bigdata","signal":44.0,"time":1616817206571}
{"device":"device_42","deviceType":"route","signal":43.0,"time":1616817206681}
{"device":"device_32","deviceType":"kafka","signal":65.0,"time":1616817207131}
{"device":"device_32","deviceType":"kafka","signal":95.0,"time":1616817207714}
{"device":"device_71","deviceType":"bigdata","signal":45.0,"time":1616817207907}
{"device":"device_32","deviceType":"kafka","signal":81.0,"time":1616817208320}
{"device":"device_10","deviceType":"db","signal":81.0,"time":1616817208907}
{"device":"device_20","deviceType":"bigdata","signal":69.0,"time":1616817209287}
{"device":"device_61","deviceType":"db","signal":98.0,"time":1616817209785}
{"device":"device_30","deviceType":"kafka","signal":95.0,"time":1616817210104}
{"device":"device_43","deviceType":"route","signal":57.0,"time":1616817210540}
{"device":"device_10","deviceType":"db","signal":36.0,"time":1616817211134}
{"device":"device_20","deviceType":"bigdata","signal":75.0,"time":1616817211248}
```

每条数据都以JSON形式存在，每条数据中都包含4个字段，分别为设备名称【device】、设备类型【deviceType】、信号强度【signal】和信号发送时间【time】，现在我们需要将这四个字段解析出来，在Hive表中以每一列的形式存储，最终得到以下Hive表：

device	devicetype	signal	stime
device_30	kafka	98.0	1616817201390
device_40	route	99.0	1616817201887
device_21	bigdata	77.0	1616817202142
device_31	kafka	98.0	1616817202405
device_20	bigdata	12.0	1616817202513
device_54	bigdata	14.0	1616817202913
device_10	db	39.0	1616817203356
device_94	bigdata	59.0	1616817203771
device_32	kafka	52.0	1616817204010
device_21	bigdata	85.0	1616817204229
device_74	bigdata	27.0	1616817204720
device_91	bigdata	50.0	1616817205164
device_62	db	89.0	1616817205328
device_21	bigdata	25.0	1616817205457
device_76	bigdata	62.0	1616817205984
device_74	bigdata	44.0	1616817206571
device_42	route	43.0	1616817206681

4.2. 处理方式

Hive中为了实现JSON格式的数据解析，提供了两种解析JSON数据的方式，在实际工作场景下，可以根据不同数据，不同的需求来选择合适的方式对JSON格式数据进行处理。

- 方式一：使用JSON函数进行处理

  Hive中提供了两个专门用于解析JSON字符串的函数：get_json_object、json_tuple，这两个函数都可以实现将JSON数据中的每个字段独立解析出来，构建成表。

- 方式二：使用Hive内置的JSON Serde加载数据

  Hive中除了提供JSON的解析函数以外，还提供了一种专门用于加载JSON文件的Serde来实现对JSON文件中数据的解析，在创建表时指定Serde，加载文件到表中，会自动解析为对应的表格式。

4.3. JSON 函数：get_json_object

功能

用于解析JSON字符串，可以从JSON字符串中返回指定的某个对象列的值

语法

- 语法

```
get_json_object(json_txt, path) - Extract a json object from path
```

- 参数
  - 第一个参数：指定要解析的JSON字符串
  - 第二个参数：指定要返回的字段，通过$.columnName的方式来指定path
- 特点：每次只能返回JSON对象中一列的值

使用

- 创建表

```
--切换数据库
use db_function;

--创建表
create table tb_json_test1 (
  json string
);
```

- 加载数据

```
--加载数据
load data local inpath '/export/data/device.json' into table
tb_json_test1;
```

- 查询数据

```
select * from tb_json_test1;
```

```
+-----------------------------------------------------+
| {"device":"device_30","deviceType":"kafka","signal":98.0,"time":1616817201390} |
| {"device":"device_40","deviceType":"route","signal":99.0,"time":1616817201887} |
| {"device":"device_21","deviceType":"bigdata","signal":77.0,"time":1616817202142} |
| {"device":"device_31","deviceType":"kafka","signal":98.0,"time":1616817202405} |
| {"device":"device_20","deviceType":"bigdata","signal":12.0,"time":1616817202513} |
| {"device":"device_54","deviceType":"bigdata","signal":14.0,"time":1616817202913} |
| {"device":"device_10","deviceType":"db","signal":39.0,"time":1616817203356} |
| {"device":"device_94","deviceType":"bigdata","signal":59.0,"time":1616817203771} |
| {"device":"device_32","deviceType":"kafka","signal":52.0,"time":1616817204010} |
| {"device":"device_21","deviceType":"bigdata","signal":85.0,"time":1616817204229} |
| {"device":"device_74","deviceType":"bigdata","signal":27.0,"time":1616817204720} |
| {"device":"device_91","deviceType":"bigdata","signal":50.0,"time":1616817205164} |
| {"device":"device_62","deviceType":"db","signal":89.0,"time":1616817205328} |
| {"device":"device_21","deviceType":"bigdata","signal":25.0,"time":1616817205457} |
| {"device":"device_76","deviceType":"bigdata","signal":62.0,"time":1616817205984} |
| {"device":"device_74","deviceType":"bigdata","signal":44.0,"time":1616817206571} |
| {"device":"device_42","deviceType":"route","signal":43.0,"time":1616817206681} |
| {"device":"device_32","deviceType":"kafka","signal":65.0,"time":1616817207131} |
| {"device":"device_32","deviceType":"kafka","signal":95.0,"time":1616817207714} |
| {"device":"device_71","deviceType":"bigdata","signal":45.0,"time":1616817207907} |
| {"device":"device_32","deviceType":"kafka","signal":81.0,"time":1616817208320} |
| {"device":"device_10","deviceType":"db","signal":81.0,"time":1616817208907} |
| {"device":"device_20","deviceType":"bigdata","signal":69.0,"time":1616817209287} |
| {"device":"device_61","deviceType":"db","signal":98.0,"time":1616817209785} |
| {"device":"device_30","deviceType":"kafka","signal":95.0,"time":1616817210104} |
```

- 获取设备名称字段

```
select
        json,
        get_json_object(json,"$.device") as device
from tb_json_test1;
```

```
+-----------------------------------------------------+------------+
|                        json                         |   device   |
+-----------------------------------------------------+------------+
| {"device":"device_30","deviceType":"kafka","signal":98.0,"time":1616817201390} | device_30  |
| {"device":"device_40","deviceType":"route","signal":99.0,"time":1616817201887} | device_40  |
| {"device":"device_21","deviceType":"bigdata","signal":77.0,"time":1616817202142} | device_21  |
| {"device":"device_31","deviceType":"kafka","signal":98.0,"time":1616817202405} | device_31  |
| {"device":"device_20","deviceType":"bigdata","signal":12.0,"time":1616817202513} | device_20  |
| {"device":"device_54","deviceType":"bigdata","signal":14.0,"time":1616817202913} | device_54  |
| {"device":"device_10","deviceType":"db","signal":39.0,"time":1616817203356} | device_10  |
| {"device":"device_94","deviceType":"bigdata","signal":59.0,"time":1616817203771} | device_94  |
| {"device":"device_32","deviceType":"kafka","signal":52.0,"time":1616817204010} | device_32  |
| {"device":"device_21","deviceType":"bigdata","signal":85.0,"time":1616817204229} | device_21  |
| {"device":"device_74","deviceType":"bigdata","signal":27.0,"time":1616817204720} | device_74  |
| {"device":"device_91","deviceType":"bigdata","signal":50.0,"time":1616817205164} | device_91  |
| {"device":"device_62","deviceType":"db","signal":89.0,"time":1616817205328} | device_62  |
| {"device":"device_21","deviceType":"bigdata","signal":25.0,"time":1616817205457} | device_21  |
| {"device":"device_76","deviceType":"bigdata","signal":62.0,"time":1616817205984} | device_76  |
| {"device":"device_74","deviceType":"bigdata","signal":44.0,"time":1616817206571} | device_74  |
| {"device":"device_42","deviceType":"route","signal":43.0,"time":1616817206681} | device_42  |
| {"device":"device_32","deviceType":"kafka","signal":65.0,"time":1616817207131} | device_32  |
| {"device":"device_32","deviceType":"kafka","signal":95.0,"time":1616817207714} | device_32  |
| {"device":"device_71","deviceType":"bigdata","signal":45.0,"time":1616817207907} | device_71  |
| {"device":"device_32","deviceType":"kafka","signal":81.0,"time":1616817208320} | device_32  |
| {"device":"device_10","deviceType":"db","signal":81.0,"time":1616817208907} | device_10  |
| {"device":"device_20","deviceType":"bigdata","signal":69.0,"time":1616817209287} | device_20  |
```

- 获取设备名称及信号强度字段

```
select
        --获取设备名称
        get_json_object(json,"$.device") as device,
        --获取设备信号强度
```

```
        get_json_object(json,"$.signal") as signal
from tb_json_test1;
```

```
+------------+---------+
|   device   | signal  |
+------------+---------+
| device_30  | 98.0    |
| device_40  | 99.0    |
| device_21  | 77.0    |
| device_31  | 98.0    |
| device_20  | 12.0    |
| device_54  | 14.0    |
| device_10  | 39.0    |
| device_94  | 59.0    |
| device_32  | 52.0    |
| device_21  | 85.0    |
| device_74  | 27.0    |
| device_91  | 50.0    |
| device_62  | 89.0    |
| device_21  | 25.0    |
```

- 实现需求

```
select
        --获取设备名称
        get_json_object(json,"$.device") as device,
        --获取设备类型
          get_json_object(json,"$.deviceType") as deviceType,
        --获取设备信号强度
        get_json_object(json,"$.signal") as signal,
        --获取时间
        get_json_object(json,"$.time") as stime
from tb_json_test1;
```

```
+------------+-------------+---------+----------------+
|   device   | devicetype  | signal  |     stime      |
+------------+-------------+---------+----------------+
| device_30  | kafka       | 98.0    | 1616817201390  |
| device_40  | route       | 99.0    | 1616817201887  |
| device_21  | bigdata     | 77.0    | 1616817202142  |
| device_31  | kafka       | 98.0    | 1616817202405  |
| device_20  | bigdata     | 12.0    | 1616817202513  |
| device_54  | bigdata     | 14.0    | 1616817202913  |
| device_10  | db          | 39.0    | 1616817203356  |
| device_94  | bigdata     | 59.0    | 1616817203771  |
| device_32  | kafka       | 52.0    | 1616817204010  |
| device_21  | bigdata     | 85.0    | 1616817204229  |
| device_74  | bigdata     | 27.0    | 1616817204720  |
| device_91  | bigdata     | 50.0    | 1616817205164  |
| device_62  | db          | 89.0    | 1616817205328  |
| device_21  | bigdata     | 25.0    | 1616817205457  |
| device_76  | bigdata     | 62.0    | 1616817205984  |
| device_74  | bigdata     | 44.0    | 1616817206571  |
| device_42  | route       | 43.0    | 1616817206681  |
```

4.4. JSON 函数：json_tuple

功能

用于实现JSON字符串的解析，可以通过指定多个参数来解析JSON返回多列的值

语法

- 语法

```
json_tuple(jsonStr, p1, p2, ..., pn)
like get_json_object, but it takes multiple names and return a tuple
```

- 参数
  - 第一个参数：指定要解析的JSON字符串
  - 第二个参数：指定要返回的第1个字段
  - ……
  - 第N+1个参数：指定要返回的第N个字段
- 特点
  - 功能类似于get_json_object，但是可以调用一次返回多列的值。属于UDTF类型函数
  - 返回的每一列都是字符串类型
  - 一般搭配lateral view使用

使用

- 获取设备名称及信号强度字段

```
select
        --返回设备名称及信号强度
        json_tuple(json,"device","signal") as (device,signal)
from tb_json_test1;
```

```
+------------+---------+
|   device   | signal  |
+------------+---------+
| device_30  | 98.0    |
| device_40  | 99.0    |
| device_21  | 77.0    |
| device_31  | 98.0    |
| device_20  | 12.0    |
| device_54  | 14.0    |
| device_10  | 39.0    |
| device_94  | 59.0    |
| device_32  | 52.0    |
| device_21  | 85.0    |
| device_74  | 27.0    |
```

- 实现需求，单独使用

```
select
        --解析所有字段
        json_tuple(json,"device","deviceType","signal","time") as
(device,deviceType,signal,stime)
from tb_json_test1;
```

```
+------------+-------------+---------+----------------+
|   device   | devicetype  | signal  |     stime      |
+------------+-------------+---------+----------------+
| device_30  | kafka       | 98.0    | 1616817201390  |
| device_40  | route       | 99.0    | 1616817201887  |
| device_21  | bigdata     | 77.0    | 1616817202142  |
| device_31  | kafka       | 98.0    | 1616817202405  |
| device_20  | bigdata     | 12.0    | 1616817202513  |
| device_54  | bigdata     | 14.0    | 1616817202913  |
| device_10  | db          | 39.0    | 1616817203356  |
| device_94  | bigdata     | 59.0    | 1616817203771  |
| device_32  | kafka       | 52.0    | 1616817204010  |
| device_21  | bigdata     | 85.0    | 1616817204229  |
| device_74  | bigdata     | 27.0    | 1616817204720  |
| device_91  | bigdata     | 50.0    | 1616817205164  |
| device_62  | db          | 89.0    | 1616817205328  |
| device_21  | bigdata     | 25.0    | 1616817205457  |
| device_76  | bigdata     | 62.0    | 1616817205984  |
| device_74  | bigdata     | 44.0    | 1616817206571  |
| device_42  | route       | 43.0    | 1616817206681  |
| device_32  | kafka       | 65.0    | 1616817207131  |
| device_32  | kafka       | 95.0    | 1616817207714  |
```

- 实现需求，搭配侧视图

```
select
        json,device,deviceType,signal,stime
from tb_json_test1
lateral                                                    view
json_tuple(json,"device","deviceType","signal","time") b
as device,deviceType,signal,stime;
```

4.5. JSONSerde

功能

上述解析JSON的过程中是将数据作为一个JSON字符串加载到表中，再通过JSON解析函数对JSON字符串进行解析，灵活性比较高，但是对于如果整个文件就是一个JSON文件，在使用起来就相对比较麻烦。Hive中为了简化对于JSON文件的处理，内置了一种专门用于解析JSON文件的Serde解析器，在创建表时，只要指定使用JSONSerde解析表的文件，就会自动将JSON文件中的每一列进行解析。

使用

- 创建表

```
--切换数据库
use db_function;

--创建表
create table tb_json_test2 (
   device string,
   deviceType string,
   signal double,
   `time` string
 )
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE;
```

- 加载数据

```
load data local inpath '/export/data/device.json' into table tb_json_test2;
```

- 查询数据

```
select * from tb_json_test2;
```

tb_json_test2.device	tb_json_test2.devicetype	tb_json_test2.signal	tb_json_test2.time
device_30	kafka	98.0	1616817201390
device_40	route	99.0	1616817201887
device_21	bigdata	77.0	1616817202142
device_31	kafka	98.0	1616817202405
device_20	bigdata	12.0	1616817202513
device_54	bigdata	14.0	1616817202913
device_10	db	39.0	1616817203356
device_94	bigdata	59.0	1616817203771
device_32	kafka	52.0	1616817204010
device_21	bigdata	85.0	1616817204229
device_74	bigdata	27.0	1616817204720
device_91	bigdata	50.0	1616817205164
device_62	db	89.0	1616817205328
device_21	bigdata	25.0	1616817205457
device_76	bigdata	62.0	1616817205984
device_74	bigdata	44.0	1616817206571
device_42	route	43.0	1616817206681

4.6. 总结

不论是Hive中的JSON函数还是自带的JSONSerde，都可以实现对于JSON数据的解析，工作中一般根据数据格式以及对应的需求来实现解析。如果数据中每一行只有个别字段是JSON格式字符串，就可以使用JSON函数来实现处理，但是如果数据加载的文件整体就是JSON文件，每一行数据就是一个JSON数据，那么建议直接使用JSONSerde来实现处理最为方便。

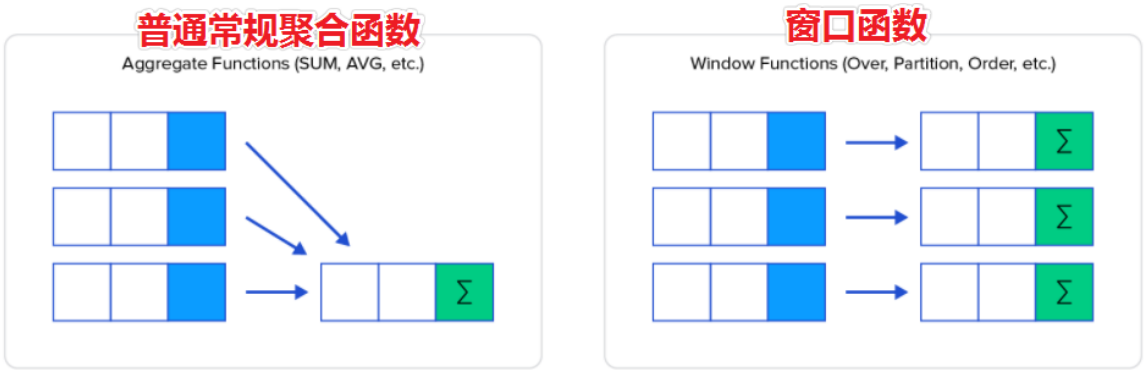
三、 Window functions 窗口函数

1. 窗口函数概述

窗口函数（Window functions）是一种SQL函数，非常适合于数据分析，因此也叫做OLAP函数，其最大特点是：输入值是从SELECT语句的结果集中的一行或多行的“窗口”中获取的。你也可以理解为窗口有大有小（行有多有少）。

通过OVER子句，窗口函数与其他SQL函数有所区别。如果函数具有OVER子句，则它是窗口函数。如果它缺少OVER子句，则它是一个普通的聚合函数。

窗口函数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，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。



为了更加直观感受窗口函数，我们通过sum聚合函数进行普通常规聚合和窗口聚合，一看效果。

```
----sum+group by 普通常规聚合操作------------
select sum(salary) as total from employee group by dept;

----sum+窗口函数聚合操作------------
select id,name,deg,salary,dept,sum(salary) over(partition by dept) as total from employee;
```

```
select * from employee;
```

5 rows | DDL | DML

	id	name	deg	salary	dept
1	1201	gopal	manager	50000	TP
2	1202	manisha	cto	50000	TP
3	1203	khalil	dev	30000	AC
4	1204	prasanth	dev	30000	AC
5	1206	kranthi	admin	20000	TP

```
----sum+group by普通常规聚合操作------------
select sum(salary) as total from employee group by dept;
```

2 rows

	total
1	60000
2	120000

group by之后分为两组，
每组不管多少条数据，sum聚合之后返回一行

```
----sum+窗口函数聚合操作------------
select id,name,deg,salary,dept,sum(salary) over(partition by dept) as total from employee;
```

1、根据dept分为两组

5 rows

2、分组后组内聚合

	id	name	deg	salary	dept	total
1	1204	prasanth	dev	30000	AC	60000
2	1203	khalil	dev	30000	AC	60000
3	1206	kranthi	admin	20000	TP	120000
4	1202	manisha	cto	50000	TP	120000
5	1201	gopal	manager	50000	TP	120000

sum聚合

sum聚合

3、但是聚合之后的结果还可以和分组内的每一行数据进行交互，并没有像普通聚合过程那样直接最终输出一行

2. 窗口函数语法

```
Function(arg1,..., argn) OVER ([PARTITION BY <...>] [ORDER BY <....>] [<window_expression>])

--其中Function(arg1,..., argn) 可以是下面分类中的任意一个
    --聚合函数：比如sum max avg等
    --排序函数：比如rank row_number等
    --分析函数：比如lead lag first_value等

--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
--如果没有PARTITION BY 那么整张表的所有行就是一组

--[ORDER BY <....>]  用于指定每个分组内的数据排序规则 支持ASC、DESC

--[<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
```

3. 案例：网站用户页面浏览次数分析

在网站访问中，经常使用cookie来标识不同的用户身份，通过cookie可以追踪不同用户的页面访问情况，有下面两份数据：

```
cookie1,2018-04-10,1
cookie1,2018-04-11,5
cookie1,2018-04-12,7
cookie1,2018-04-13,3
cookie1,2018-04-14,2
cookie1,2018-04-15,4
cookie1,2018-04-16,4
cookie2,2018-04-10,2
cookie2,2018-04-11,3
cookie2,2018-04-12,5
```

字段含义：cookieid 、访问时间、pv数（页面浏览数）

```
cookie1,2018-04-10 10:00:02,url2
cookie1,2018-04-10 10:00:00,url1
cookie1,2018-04-10 10:03:04,1url3
cookie1,2018-04-10 10:50:05,url6
cookie1,2018-04-10 11:00:00,url7
cookie1,2018-04-10 10:10:00,url4
cookie1,2018-04-10 10:50:01,url5
cookie2,2018-04-10 10:00:02,url22
cookie2,2018-04-10 10:00:00,url11
cookie2,2018-04-10 10:03:04,1url33
cookie2,2018-04-10 10:50:05,url66
cookie2,2018-04-10 11:00:00,url77
cookie2,2018-04-10 10:10:00,url44
cookie2,2018-04-10 10:50:01,url55
```

字段含义：cookieid、访问时间、访问页面url

在Hive中创建两张表表，把数据加载进去用于窗口分析。

```
---建表并且加载数据
create table website_pv_info(
   cookieid string,
```

```
   createtime string,   --day
   pv int
) row format delimited
fields terminated by ',';

create table website_url_info (
    cookieid string,
    createtime string,  --访问时间
    url string      --访问页面
) row format delimited
fields terminated by ',';


load data local inpath '/root/hivedata/website_pv_info.txt' into
table website_pv_info;
load data local inpath '/root/hivedata/website_url_info.txt'
into table website_url_info;

select * from website_pv_info;
select * from website_url_info;
```

3.1. 窗口聚合函数

从Hive v2.2.0开始，支持DISTINCT与窗口函数中的聚合函数一起使用。

这里以sum（）函数为例，其他聚合函数使用类似。

```
-----窗口聚合函数的使用-----------
--1、求出每个用户总pv数  sum+group by普通常规聚合操作
select cookieid,sum(pv) as total_pv from website_pv_info group
by cookieid;

--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
--sum(...) over( )对表所有行求和
--sum(...) over( order by ... ) 连续累积求和
--sum(...) over( partition by... ) 同组内所有行求和
--sum(...) over( partition by... order by ... ) 在每个分组内，连续
```

累积求和

```
--需求：求出网站总的pv数 所有用户所有访问加起来
--sum(...) over( )对表所有行求和
select cookieid,createtime,pv,
       sum(pv) over() as total_pv
from website_pv_info;

--需求：求出每个用户总pv数
--sum(...) over( partition by... ), 同组内所行求和
select cookieid,createtime,pv,
       sum(pv) over(partition by cookieid) as total_pv
from website_pv_info;

--需求：求出每个用户截止到当天，累积的总pv数
--sum(...) over( partition by... order by ... ), 在每个分组内，连续累积求和
select cookieid,createtime,pv,
       sum(pv) over(partition by cookieid order by createtime) as current_total_pv
from website_pv_info;
```

```
--需求：求出每个用户总pv数
--sum(...) over( partition by... ), 同组内所行求和
select cookieid,createtime,pv,
       sum(pv) over(partition by cookieid) as total_pv
from website_pv_info;
```

没有order by
就是所有行整体求和

14 rows

	cookieid	createtime	pv	total_pv
1	cookie1	2018-04-10	1	26
2	cookie1	2018-04-16	4	26
3	cookie1	2018-04-15	4	26
4	cookie1	2018-04-14	2	26
5	cookie1	2018-04-13	3	26
6	cookie1	2018-04-12	7	26
7	cookie1	2018-04-11	5	26
8	cookie2	2018-04-16	7	35
9	cookie2	2018-04-15	9	35

26

```
--需求：求出每个用户截止到当天，累积的总pv数
--sum(...) over( partition by... order by ... )，在每个分组内，连续累积求和
select cookieid,createtime,pv,
       sum(pv) over(partition by cookieid order by createtime) as current_total_pv
from website_pv_info;
```

有order by 注意这是累积聚合

14 rows

	cookieid	createtime	pv	current_total_pv
1	cookie1	2018-04-10	1	1
2	cookie1	2018-04-11	5	6
3	cookie1	2018-04-12	7	13
4	cookie1	2018-04-13	3	16
5	cookie1	2018-04-14	2	18
6	cookie1	2018-04-15	4	22
7	cookie1	2018-04-16	4	26
8	cookie2	2018-04-10	2	2
9	cookie2	2018-04-11	3	5
10	cookie2	2018-04-12	5	10
11	cookie2	2018-04-13	6	16
12	cookie2	2018-04-14	3	19

新的一组

3.2. 窗口表达式

我们知道，在sum(...) over(partition by... order by ...)语法完整的情况下，进行的累积聚合操作，默认累积聚合行为是：从第一行聚合到当前行。

Window expression窗口表达式给我们提供了一种控制行范围的能力，比如向前2行，向后3行。

语法如下：

关键字是rows between，包括下面这几个选项

- preceding：往前
- following：往后
- current row：当前行
- unbounded：边界
- unbounded preceding 表示从前面的起点
- unbounded following：表示到后面的终点

```
---窗口表达式
--第一行到当前行
select cookieid,createtime,pv,
      sum(pv) over(partition by cookieid order by createtime rows
between unbounded preceding and current row) as pv2
from website_pv_info;

--向前 3 行至当前行
select cookieid,createtime,pv,
      sum(pv) over(partition by cookieid order by createtime rows
between 3 preceding and current row) as pv4
from website_pv_info;

--向前 3 行 向后 1 行
```

```
select cookieid,createtime,pv,
      sum(pv) over(partition by cookieid order by createtime rows
between 3 preceding and 1 following) as pv5
from website_pv_info;

--当前行至最后一行
select cookieid,createtime,pv,
      sum(pv) over(partition by cookieid order by createtime rows
between current row and unbounded following) as pv6
from website_pv_info;

--第一行到最后一行 也就是分组内的所有行
select cookieid,createtime,pv,
      sum(pv) over(partition by cookieid order by createtime rows
between unbounded preceding  and unbounded following) as pv6
from website_pv_info;
```

3.3. 窗口排序函数

窗口排序函数用于给每个分组内的数据打上排序的标号。注意窗口排序函数不支持窗口表达式。总共有4个函数需要掌握：

row_number：在每个分组中，为每行分配一个从1开始的唯一序列号，递增，不考虑重复；

rank: 在每个分组中，为每行分配一个从1开始的序列号，考虑重复，挤占后续位置；

dense_rank: 在每个分组中，为每行分配一个从1开始的序列号，考虑重复，不挤占后续位置；

```
-----窗口排序函数
SELECT
    cookieid,
    createtime,
    pv,
    RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn1,
```

```
    DENSE_RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS
rn2,
    ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS
rn3
FROM website_pv_info
WHERE cookieid = 'cookie1';
```

7 rows

cookieid	createtime	pv	rn1	rn2	rn3
cookie1	2018-04-12	7	1	1	1
cookie1	2018-04-11	5	2	2	2
cookie1	2018-04-16	4	3	3	3
cookie1	2018-04-15	4	3	3	4
cookie1	2018-04-13	3	5	4	5
cookie1	2018-04-14	2	6	5	6
cookie1	2018-04-10	1	7	6	7

注意数据重复 RANK DENSE_RANK ROW_NUMBER

上述这三个函数用于分组TopN的场景非常适合。

```
--需求：找出每个用户访问 pv 最多的 Top3 重复并列的不考虑
SELECT * from
(SELECT
    cookieid,
    createtime,
    pv,
    ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC)
AS seq
FROM website_pv_info) tmp where tmp.seq <4;
```

6 rows

	cookieid	createtime	pv	seq
1	cookie1	2018-04-12	7	1
2	cookie1	2018-04-11	5	2
3	cookie1	2018-04-16	4	3
4	cookie2	2018-04-15	9	1
5	cookie2	2018-04-16	7	2
6	cookie2	2018-04-13	6	3

还有一个函数，叫做ntile函数，其功能为：将每个分组内的数据分为指定的若干个桶里（分为若干个部分），并且为每一个桶分配一个桶编号。

如果不能平均分配，则优先分配较小编号的桶，并且各个桶中能放的行数最多相差1。

有时会有这样的需求:如果数据排序后分为三部分，业务人员只关心其中的一部分，如何将这中间的三分之一数据拿出来呢?NTILE函数即可以满足。

```
--把每个分组内的数据分为 3 桶
SELECT
    cookieid,
    createtime,
    pv,
    NTILE(3) OVER(PARTITION BY cookieid ORDER BY createtime) AS
rn2
FROM website_pv_info
ORDER BY cookieid,createtime;
```

cookieid	createtime	pv	rn2
cookie1	2018-04-10	1	1
cookie1	2018-04-11	5	1
cookie1	2018-04-12	7	1
cookie1	2018-04-13	3	2
cookie1	2018-04-14	2	2
cookie1	2018-04-15	4	3
cookie1	2018-04-16	4	3
cookie2	2018-04-10	2	1
cookie2	2018-04-11	3	1
cookie2	2018-04-12	5	1
cookie2	2018-04-13	6	2
cookie2	2018-04-14	3	2
cookie2	2018-04-15	9	3
cookie2	2018-04-16	7	3

```
--需求：统计每个用户 pv 数最多的前 3 分之 1 天。
--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
SELECT * from
(SELECT
    cookieid,
    createtime,
    pv,
    NTILE(3) OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn
 FROM website_pv_info) tmp where rn =1;
```

6 rows

	cookieid	createtime	pv	rn
1	cookie1	2018-04-12	7	1
2	cookie1	2018-04-11	5	1
3	cookie1	2018-04-16	4	1
4	cookie2	2018-04-15	9	1
5	cookie2	2018-04-16	7	1
6	cookie2	2018-04-13	6	1

3.4. 窗口分析函数

LAG(col,n,DEFAULT) 用于统计窗口内往上第n行值

第一个参数为列名，第二个参数为往上第n行（可选，默认为1），第三个参数为默认值（当往上第n行为NULL时候，取默认值，如不指定，则为NULL）；

LEAD(col,n,DEFAULT) 用于统计窗口内往下第n行值

第一个参数为列名，第二个参数为往下第n行（可选，默认为1），第三个参数为默认值（当往下第n行为NULL时候，取默认值，如不指定，则为NULL）；

FIRST_VALUE 取分组内排序后，截止到当前行，第一个值；

LAST_VALUE 取分组内排序后，截止到当前行，最后一个值；

```
----------窗口分析函数----------
--LAG
SELECT cookieid,
       createtime,
       url,
       ROW_NUMBER()   OVER(PARTITION   BY   cookieid   ORDER   BY
createtime) AS rn,
       LAG(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY
cookieid ORDER BY createtime) AS last_1_time,
       LAG(createtime,2)  OVER(PARTITION  BY  cookieid  ORDER  BY
createtime) AS last_2_time
FROM website_url_info;


--LEAD
SELECT cookieid,
       createtime,
       url,
       ROW_NUMBER()   OVER(PARTITION   BY   cookieid   ORDER   BY
createtime) AS rn,
       LEAD(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY
cookieid ORDER BY createtime) AS next_1_time,
       LEAD(createtime,2)  OVER(PARTITION  BY  cookieid  ORDER  BY
createtime) AS next_2_time
FROM website_url_info;

--FIRST_VALUE
SELECT cookieid,
       createtime,
       url,
       ROW_NUMBER()   OVER(PARTITION   BY   cookieid   ORDER   BY
createtime) AS rn,
       FIRST_VALUE(url)  OVER(PARTITION  BY  cookieid  ORDER  BY
createtime) AS first1
FROM website_url_info;

--LAST_VALUE
SELECT cookieid,
```

```
        createtime,
        url,
        ROW_NUMBER()    OVER(PARTITION    BY    cookieid    ORDER    BY
createtime) AS rn,
        LAST_VALUE(url)    OVER(PARTITION    BY    cookieid    ORDER    BY
createtime) AS last1
FROM website_url_info;
```

```
        LAG(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY createtime) AS last_1_time,
        LAG(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS last_2_time
FROM website_url_info;
```

向上1行 且有默认值
因此不会出现null

14 rows

	cookieid	createtime	url	rn	last_1_time	last_2_time
1	cookie1	2018-04-10 10:00:00	url1	1	1970-01-01 00:00:00	<null>
2	cookie1	2018-04-10 10:00:02	url2	2	2018-04-10 10:00:00	<null>
3	cookie1	2018-04-10 10:03:04	1url3	3	2018-04-10 10:00:02	2018-04-10 10:00:00
4	cookie1	2018-04-10 10:10:00	url4	4	2018-04-10 10:03:04	2018-04-10 10:00:02
5	cookie1	2018-04-10 10:50:01	url5	5	2018-04-10 10:10:00	2018-04-10 10:03:04
6	cookie1	2018-04-10 10:50:05	url6	6	2018-04-10 10:50:01	2018-04-10 10:10:00
7	cookie1	2018-04-10 11:00:00	url7	7	2018-04-10 10:50:05	2018-04-10 10:50:01

四、 Hive 数据压缩

1. 优缺点

优点：

减少存储磁盘空间，降低单节点的磁盘 IO。

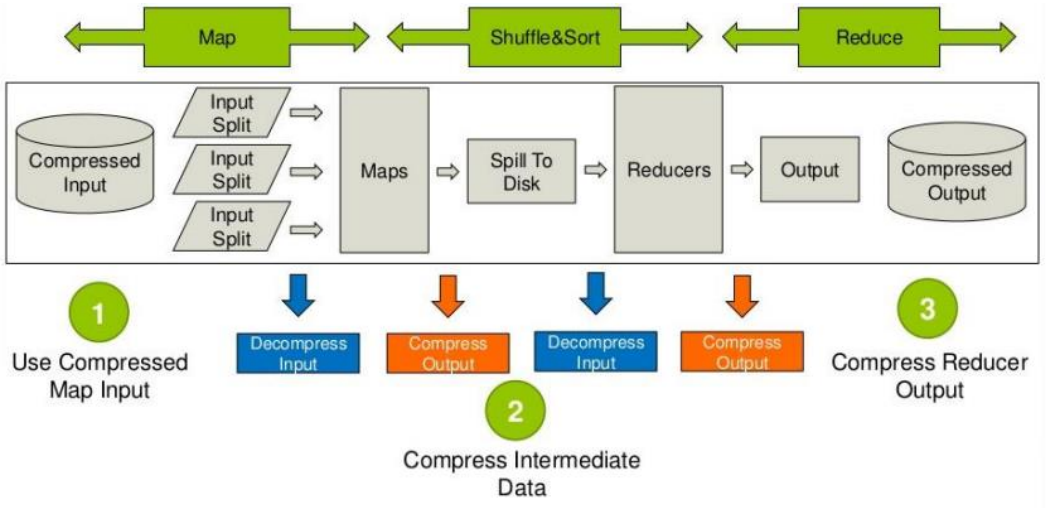
由于压缩后的数据占用的带宽更少，因此可以加快数据在 Hadoop 集群流动的速度，减少网络传输带宽。

缺点：

需要花费额外的时间/CPU 做压缩和解压缩计算。

2. 压缩分析

首先说明 mapreduce 哪些过程可以设置压缩：需要分析处理的数据在进入 map 前可以压缩，然后解压处理，map 处理完成后的输出可以压缩，这样可以减少网络 I/O(reduce 通常和 map 不在同一节点上)，reduce 拷贝压缩的数据后进行解压，处理完成后可以压缩存储在 hdfs 上，以减少磁盘占用量。



3. Hadoop 中支持的压缩算法

压缩格式	压缩格式所在的类
Zlib	org.apache.hadoop.io.compress.DefaultCodec
Gzip	org.apache.hadoop.io.compress.GzipCodec
Bzip2	org.apache.hadoop.io.compress.BZip2Codec
Lzo	com.hadoop.compression.lzo.LzoCodec
Lz4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

4. Hive 的压缩设置

4.1. 开启 hive 中间传输数据压缩功能

1）开启 hive 中间传输数据压缩功能

```
set hive.exec.compress.intermediate=true;
```

2）开启 mapreduce 中 map 输出压缩功能

```
set mapreduce.map.output.compress=true;
```

3）设置 mapreduce 中 map 输出数据的压缩方式

```
set mapreduce.map.output.compress.codec=
org.apache.hadoop.io.compress.SnappyCodec;
```

4.2. 开启 Reduce 输出阶段压缩

1）开启 hive 最终输出数据压缩功能

```
set hive.exec.compress.output=true;
```

2）开启 mapreduce 最终输出数据压缩

```
set mapreduce.output.fileoutputformat.compress=true;
```

3）设置 mapreduce 最终数据输出压缩方式

```
set mapreduce.output.fileoutputformat.compress.codec =
org.apache.hadoop.io.compress.SnappyCodec;
```

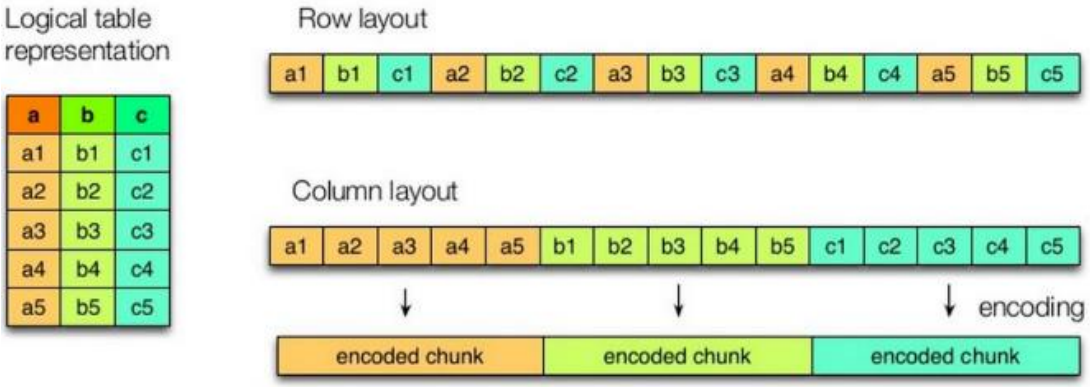
4）设置 mapreduce 最终数据输出压缩为块压缩

```
set mapreduce.output.fileoutputformat.compress.type=BLOCK;
```

五、 Hive 数据存储格式

1. 列式存储和行式存储

逻辑表中的数据，最终需要落到磁盘上，以文件的形式存储，有两种常见的存储形式。行式存储和列式存储。



1.1. 行式存储

优点：

相关的数据是保存在一起，比较符合面向对象的思维，因为一行数据就是一条记录

这种存储格式比较方便进行 INSERT/UPDATE 操作

缺点：

如果查询只涉及某几个列，它会把整行数据都读取出来，不能跳过不必要的列读取。当然数据比较少，一般没啥问题，如果数据量比较大就比较影响性能

由于每一行中，列的数据类型不一致，导致不容易获得一个极高的压缩比，也就是空间利用率不高

不是所有的列都适合作为索引

1.2. 列式存储

优点：

查询时，只有涉及到的列才会被查询，不会把所有列都查询出来，即可以跳过不必要的列查询；

高效的压缩率，不仅节省储存空间也节省计算内存和 CPU；

任何列都可以作为索引；

缺点：

INSERT/UPDATE 很麻烦或者不方便；

不适合扫描小量的数据

Hive 支持的存储数的格式主要有：TEXTFILE（行式存储） 、SEQUENCEFILE(行式存储)、ORC（列式存储）、PARQUET（列式存储）。

2. TEXTFILE 格式

默认格式，数据不做压缩，磁盘开销大，数据解析开销大。可结合 Gzip、Bzip2 使用(系统自动检查，执行查询时自动解压)，但使用这种方式，hive 不会对数据进行切分，从而无法对数据进行并行操作。

3. ORC 格式

ORC 的全称是(Optimized Row Columnar)，ORC 文件格式是一种 Hadoop 生态圈中的列式存储格式，它的产生早在 2013 年初，最初产生自 Apache Hive，用于降低 Hadoop 数据存储空间和加速 Hive 查询速度。它并不是一个单纯的列式存储格式，仍然是首先根据行组分割整个表，在每一个行组内进行按列存储。

优点如下：

ORC 是列式存储，有多种文件压缩方式，并且有着很高的压缩比。

文件是可切分（Split）的。因此，在 Hive 中使用 ORC 作为表的文件存储格式，不仅节省 HDFS 存储资源，查询任务的输入数据量减少，使用的 MapTask 也就减少了。

ORC 可以支持复杂的数据结构（比如 Map 等）。

ORC 文件也是以二进制方式存储的，所以是不可以直接读取，ORC 文件也是自解析的。

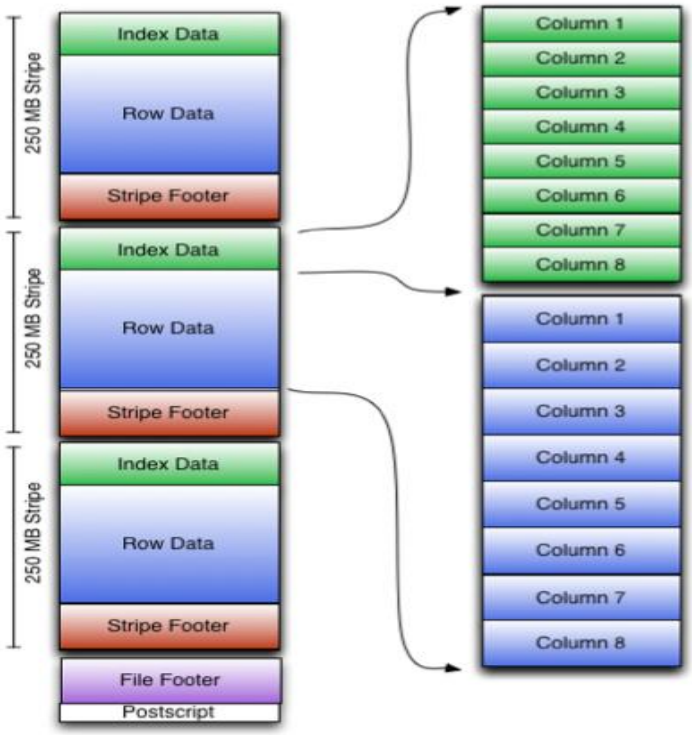
3.1. 了解 ORC 结构

一个 orc 文件可以分为若干个 Stripe，一个 stripe 可以分为三个部分：

indexData：某些列的索引数据

rowData :真正的数据存储

StripFooter：stripe 的元数据信息



1）Index Data：一个轻量级的 index，默认是每隔 1W 行做一个索引。这里做的索引只是记录某行的各字段在 Row Data 中的 offset。

2）Row Data：存的是具体的数据，先取部分行，然后对这些行按列进行存储。对每个列进行了编码，分成多个 Stream 来存储。

3）Stripe Footer：存的是各个 stripe 的元数据信息。

每个文件有一个 File Footer，这里面存的是每个 Stripe 的行数，每个 Column 的数据类型信息等；每个文件的尾部是一个 PostScript，这里面记录了整个文件的压缩类型以及 FileFooter 的长度信息等。在读取文件时，会 seek 到文件尾部读 PostScript，从里面解析到 File Footer 长度，再读 FileFooter，从里面解析到各个 Stripe 信息，再读各个 Stripe，即从后往前读。

4. PARQUET 格式

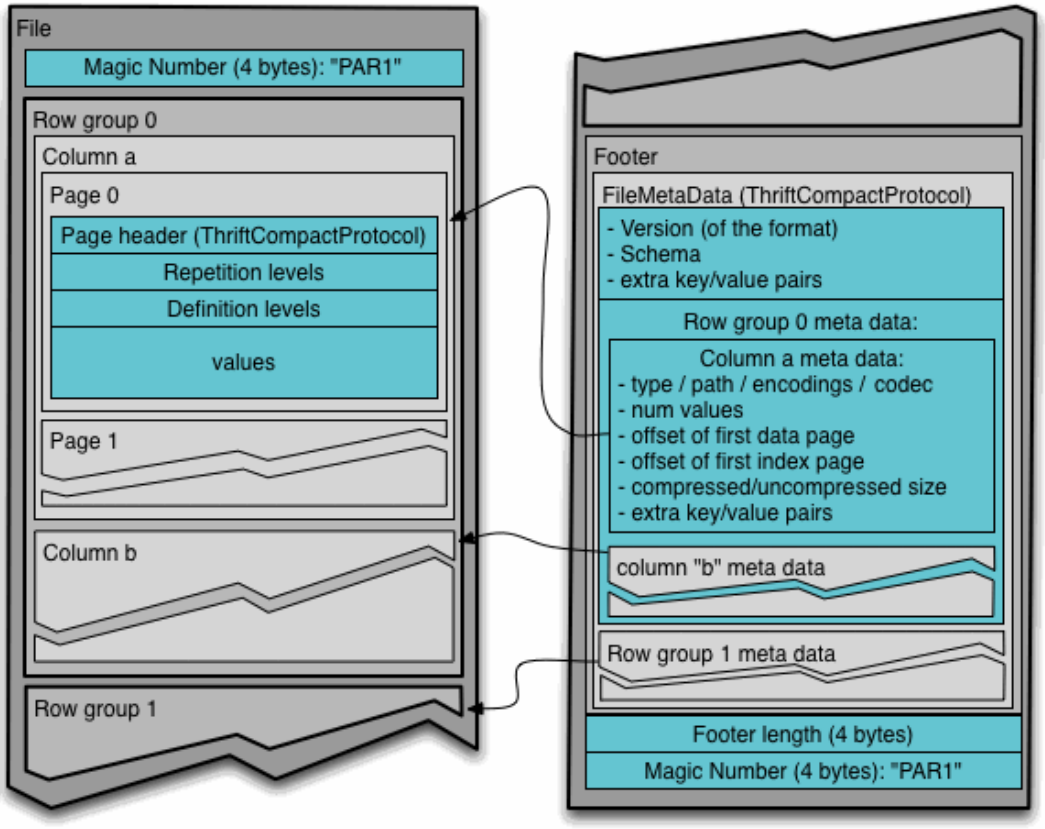
Parquet 是面向分析型业务的列式存储格式，由 Twitter 和 Cloudera 合作开发，2015 年 5 月从 Apache 的孵化器里毕业成为 Apache 顶级项目。

Parquet 文件是以二进制方式存储的，所以是不可以直接读取的，文件中包括该文件的数据和元数据，因此 Parquet 格式文件是自解析的。

通常情况下，在存储 Parquet 数据的时候会按照 Block 大小设置行组的大小，由于一般情况下每一个 Mapper 任务处理数据的最小单位是一个 Block，这样可以把每一个行组由一个 Mapper 任务处理，增大任务执行并行度。

4.1. 了解 PARQUET 格式

Parquet 文件的格式如下图所示：



上图展示了一个 Parquet 文件的内容，一个文件中可以存储多个行组，文件的首位都是该文件的 Magic Code，用于校验它是否是一个 Parquet 文件，Footer length 了文件元数据的大小，通过该值和文件长度可以计算出元数据的偏移量，文件的元数据中包括每一个行组的元数据信息和该文件存储数据的 Schema 信息。除了文件中每一个行组的元数据，每一页的开始都会存储该页的元数据，在 Parquet 中，有三种类型的页：数据页、字典页和索引页。数据页用于存储当前行组中该列的值，字典页存储该列值的编码字典，每一个列块中最多包含一个字典页，索引页用来存储当前行组下该列的索引，目前 Parquet 中还不支持索引页，但是在后面的版本中增加。

5. 文件格式存储对比

从存储文件的压缩比和查询速度两个角度对比。

测试数据 参见附件资料 log.data。

5.1. TEXTFILE

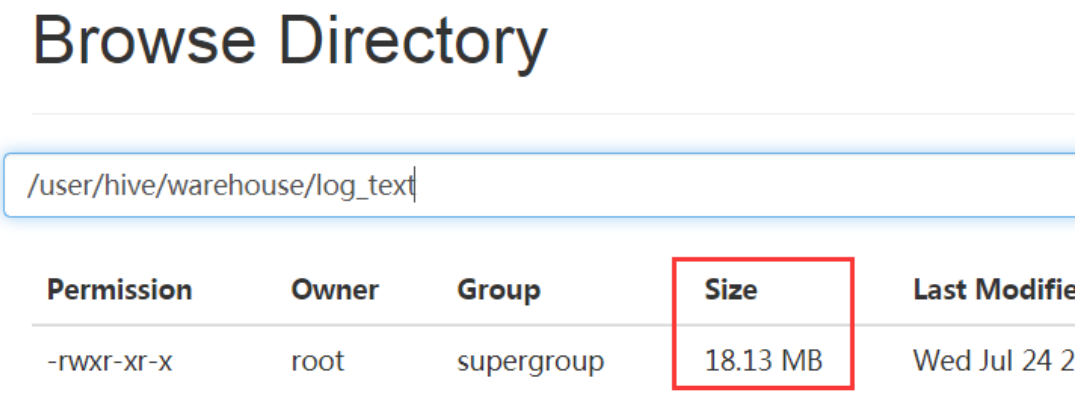
创建表，存储数据格式为 TEXTFILE。

```
create table log_text (
track_time string,
url string,
session_id string,
referer string,
ip string,
end_user_id string,
city_id string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE ;
```

加载数据：

```
load data local inpath '/root/hivedata/log.data' into table log_text ;
```

查看表中数据大小：



5.2. ORC

创建表，存储数据格式为 ORC。

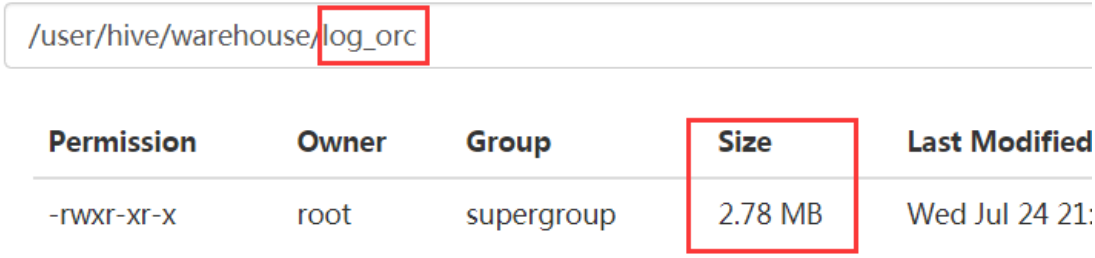
```
create table log_orc(
track_time string,
url string,
session_id string,
referer string,
ip string,
end_user_id string,
city_id string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS orc ;
```

向表中加载数据：

```
insert into table log_orc select * from log_text ;
```

查看表中数据大小：

```
dfs -du -h /user/hive/warehouse/log_orc;
```



5.3. PARQUET

创建表，存储数据格式为 parquet。

```
create table log_parquet(
track_time string,
url string,
session_id string,
referer string,
ip string,
end_user_id string,
city_id string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS PARQUET ;
```

向表中加载数据：

```
insert into table log_parquet select * from log_text ;
```

查看表中数据大小：

```
dfs -du -h /user/hive/warehouse/log_parquet;
```

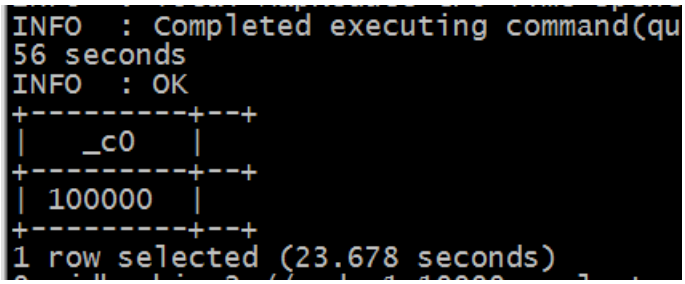
存储文件的压缩比总结：

ORC > Parquet > textFile

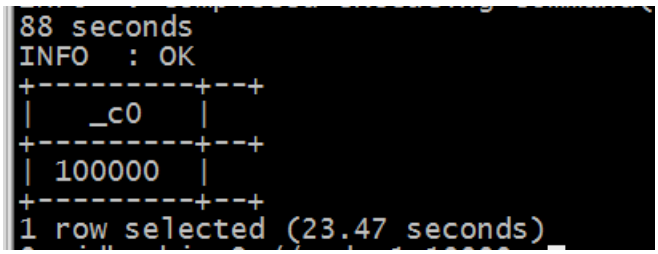
6. 存储文件查询速度对比

可以针对三张表，使用 sql 统计表数据个数。查看执行时间。

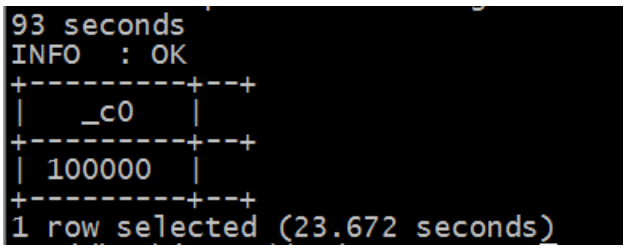
select count(*) from log_orc;



select count(*) from log_orc;



select count(*) from log_parquet;



7. 存储格式和压缩的整合

7.1. 非压缩 ORC 文件

建表语句：

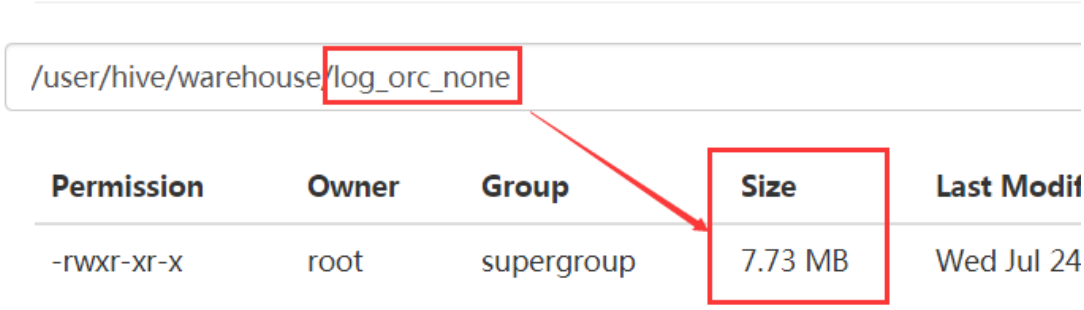
```
create table log_orc_none(
track_time string,
url string,
session_id string,
referer string,
ip string,
end_user_id string,
city_id string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS orc tblproperties ("orc.compress"="NONE");
```

插入数据：

```
insert into table log_orc_none select * from log_text ;
```

查看插入后数据：

```
dfs -du -h /user/hive/warehouse/log_orc_none;
```



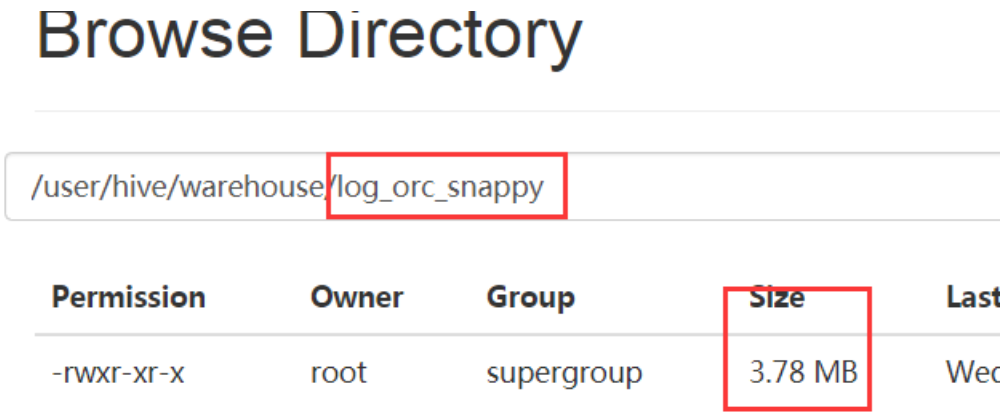
7.2. Snappy 压缩 ORC 文件

建表语句：

```
create table log_orc_snappy(
track_time string,
url string,
session_id string,
referer string,
ip string,
end_user_id string,
city_id string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS orc tblproperties ("orc.compress"="SNAPPY");
```

插入数据：

```
insert into table log_orc_snappy select * from log_text ;
```



上一节中默认创建的 ORC 存储方式，导入数据后的大小为比 Snappy 压缩的还小。原因是 orc 存储文件默认采用 ZLIB 压缩。比 snappy 压缩的小。

在实际的项目开发当中，hive 表的数据存储格式一般选择：orc 或 parquet。压缩方式一般选择 snappy。

六、 Hive 调优

1. Fetch 抓取机制

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

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

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

```
set hive.fetch.task.conversion=none;
select * from score;
select s_score from score;
select s_score from score limit 3;
```

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

```
set hive.fetch.task.conversion=more;
select * from score;
select s_score from score;
select s_score from score limit 3;
```

2. mapreduce 本地模式

mapreduce 程序除了可以提交到 yarn 执行之外，还可以使用本地模拟环境运行，此时就不是分布式执行的程序，但是针对小文件小数据处理特别有效果。

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。

hive 自动根据下面三个条件判断是否启动本地模式：

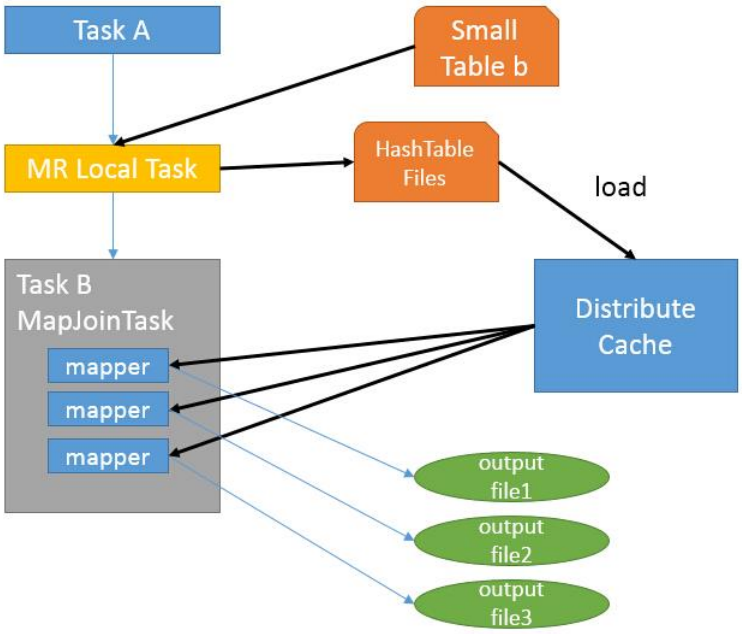
```
The total input size of the job is lower than:
 hive.exec.mode.local.auto.inputbytes.max (128MB by default)
The total number of map-tasks is less than:
 hive.exec.mode.local.auto.tasks.max (4 by default)
The total number of reduce tasks required is 1 or 0.
```

3． join 查询的优化

多个表关联时，最好分拆成小段 sql 分段执行，避免一个大 sql（无法控制中间 Job）。

3.1． map side join

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



首先是 Task A，它是一个 Local Task（在客户端本地执行的 Task），负责扫描小表 b 的数据，将其转换成一个 HashTable 的数据结构，并写入本地的文件中，之后将该文件加载到 DistributeCache 中。

接下来是 Task B，该任务是一个没有 Reduce 的 MR，启动 MapTasks 扫描大表 a,在 Map 阶段，根据 a 的每一条记录去和 DistributeCache 中 b 表对应的 HashTable 关联，并直接输出结果。

由于 MapJoin 没有 Reduce，所以由 Map 直接输出结果文件，有多少个 Map Task，就有多少个结果文件。

map 端 join 的参数设置：

开启 mapjoin 参数设置：

（1）设置自动选择 mapjoin

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

（2）大表小表的阈值设置：

set hive.mapjoin.smalltable.filesize=25000000;

小表的输入文件大小的阈值（以字节为单位）;如果文件大小小于此阈值，它将尝试将 common join 转换为 map join。

因此在实际使用中，只要根据业务把握住小表的阈值标准即可，hive 会自动帮我们完成 mapjoin，提高执行的效率。

3.2. 大表 join 大表

空 key 过滤

有时 join 超时是因为某些 key 对应的数据太多，而相同 key 对应的数据都会发送到相同的 reducer 上，从而导致内存不够。

此时我们应该仔细分析这些异常的 key，很多情况下，这些 key 对应的数据是异常数据，我们需要在 SQL 语句中进行过滤。例如 key 对应的字段为空，操作如下：

准备环境：

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

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

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

load data local inpath '/root/hivedata/hive_big_table/*' into table ori;
load data local inpath '/root/hivedata/hive_have_null_id/*' into table nullidtable;
```

不过滤查询：

```
INSERT OVERWRITE TABLE jointable
SELECT a.* FROM nullidtable a JOIN ori b ON a.id = b.id;
结果：
No rows affected (152.135 seconds)
```

过滤查询：

```
INSERT OVERWRITE TABLE jointable
SELECT a.* FROM (SELECT * FROM nullidtable WHERE id IS NOT NULL ) a JOIN ori b ON a.id = b.id;
结果：
No rows affected (141.585 seconds)
```

空 key 转换

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

不随机分布：

```
set hive.exec.reducers.bytes.per.reducer=32123456;
set mapreduce.job.reduces=7;

INSERT OVERWRITE TABLE jointable
SELECT a.*
FROM nullidtable a
LEFT JOIN ori b ON CASE WHEN a.id IS NULL THEN 'hive' ELSE a.id END = b.id;
No rows affected (41.668 seconds)
```

结果：这样的后果就是所有为 null 值的 id 全部都变成了相同的字符串，及其容易造成数据的倾斜（所有的 key 相同，相同 key 的数据会到同一个 reduce 当中去）。

为了解决这种情况，我们可以通过 hive 的 rand 函数，随记的给每一个为空的 id 赋上一个随机值，这样就不会造成数据倾斜。

随机分布：

```
set hive.exec.reducers.bytes.per.reducer=32123456;
set mapreduce.job.reduces=7;

INSERT OVERWRITE TABLE jointable
SELECT a.*
FROM nullidtable a
LEFT JOIN ori b ON CASE WHEN a.id IS NULL THEN concat('hive', rand()) ELSE a.id END = b.id;
```

3.3. 大小表、小大表 join

在当下的 hive 版本中，大表 join 小表或者小表 join 大表，就算是关闭 map 端 join 的情况下，基本上没有区别了（hive 为了解决数据倾斜的问题，会自动进行过滤）。

数据环境准备：

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

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

create table jointable2(id bigint, time bigint, uid string, keyword string, url_rank int, click_num int, click_url string) row format delimited fields terminated by '\t';

load data local inpath '/root/hivedata/big_data' into table bigtable;
load data local inpath '/root/hivedata/small_data' into table smalltable;
```

大小表、小大表 join 测试：

```
INSERT OVERWRITE TABLE jointable2
SELECT b.id, b.time, b.uid, b.keyword, b.url_rank, b.click_num, b.click_url
FROM smalltable s
left JOIN bigtable    b
ON b.id = s.id;

INSERT OVERWRITE TABLE jointable2
SELECT b.id, b.time, b.uid, b.keyword, b.url_rank, b.click_num, b.click_url
FROM bigtable    b
left JOIN smalltable    s
ON s.id = b.id;
```

4. group by 优化—map 端聚合

默认情况下，当进行 group by 的时候，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。

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

（1）是否在 Map 端进行聚合，默认为 True

```
set hive.map.aggr = true;
```

（2）在 Map 端进行聚合操作的条目数目

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

（3）有数据倾斜的时候进行负载均衡（默认是 false）

```
set hive.groupby.skewindata = true;
```

当选项设定为 true，生成的查询计划会有两个 MR Job。第一个 MR Job 中，Map 的输出结果会随机分布到 Reduce 中，每个 Reduce 做部分聚合操作，并输出结果，这样处理的结果是相同的 Group By Key 有可能被分发到不同的 Reduce 中，从而达到负载均衡的目的；第二个 MR Job 再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中（这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中），最后完成最终的聚合操作。

5. MapReduce 引擎并行度调整

5.1. maptask 个数调整

小文件场景

对已经存在的小文件做出的解决方案：

使用 Hadoop achieve 把小文件进行归档

重建表，建表时减少 reduce 的数量

通过参数调节，设置 map 数量（下面的 API 属于 hadoop 低版本的 API）

```
//每个 Map 最大输入大小(这个值决定了合并后文件的数量)
set mapred.max.split.size=112345600;
//一个节点上 split 的至少的大小(这个值决定了多个 DataNode 上的文件是否需要合并)
set mapred.min.split.size.per.node=112345600;
//一个交换机下 split 的至少的大小(这个值决定了多个交换机上的文件是否需要合并)
set mapred.min.split.size.per.rack=112345600;
//执行 Map 前进行小文件合并
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
```

大文件场景

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

如果表 a 只有一个文件，大小为 120M，但包含几千万的记录，如果用 1 个 map 去完成这个任务，肯定是比较耗时的，这种情况下，我们要考虑将这一个文件合理的拆分成多个，这样就可以用多个 map 任务去完成。

```
set mapreduce.job.reduces =10;

create table a_1 as
select * from a
distribute by rand(123);
```

这样会将 a 表的记录，随机的分散到包含 10 个文件的 a_1 表中，再用 a_1 代替上面 sql 中的 a 表，则会用 10 个 map 任务去完成。

每个 map 任务处理大于 12M（几百万记录）的数据，效率肯定会好很多。

5.2. reducetask 个数调整

总共受 3 个参数控制：

（1）每个 Reduce 处理的数据量默认是 256MB

hive.exec.reducers.bytes.per.reducer=256123456

（2）每个任务最大的 reduce 数，默认为 1009

hive.exec.reducers.max=1009

（3）mapreduce.job.reduces

该值默认为-1，由 hive 自己根据任务情况进行判断。

因此，可以手动设置每个 job 的 Reduce 个数

```
set mapreduce.job.reduces = 8;
```

reduce 个数并不是越多越好

1）过多的启动和初始化 reduce 也会消耗时间和资源；

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

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

6. 执行计划—explain（了解）

基本语法

```
EXPLAIN [EXTENDED | DEPENDENCY | AUTHORIZATION] query
```

案例实操

（1）查看下面这条语句的执行计划

```
explain select * from course;
explain select s_id ,avg(s_score) avgscore from score group by s_id;
```

（2）查看详细执行计划

```
explain extended select * from course;
explain extended select s_id ,avg(s_score) avgscore from score group by s_id;
```

7. 并行执行机制

Hive 会将一个查询转化成一个或者多个阶段。这样的阶段可以是 MapReduce 阶段、抽样阶段、合并阶段、limit 阶段。或者 Hive 执行过程中可能需要的其他阶段。

默认情况下，Hive 一次只会执行一个阶段。不过，某个特定的 job 可能包含众多的阶段，而这些阶段可能并非完全互相依赖的，也就是说有些阶段是可以并行执行的，这样可能使得整个 job 的执行时间缩短。不过，如果有更多的阶段可以并行执行，那么 job 可能就越快完成。

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

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

当然，得是在系统资源比较空闲的时候才有优势，否则，没资源，并行也起不来。

8. 严格模式

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

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

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

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

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

9. 推测执行机制

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

hadoop 中默认两个阶段都开启了推测执行机制。

hive 本身也提供了配置项来控制 reduce-side 的推测执行：

```
<property>
    <name>hive.mapred.reduce.tasks.speculative.execution</name>
    <value>true</value>
</property>
```

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