Hadoop生态圈技术栈(上)笔记

课程主要内容

```
数据仓库工具：Hive【重
点】
```

数据仓库工具 -- Hive

第一部分 Hive概述

HDFS => 海量数据的存储

MapReduce => 海量数据的分析和处理

YARN => 集群资源的管理和作业调度

第 1 节 Hive产生背景

直接使用MapReduce处理大数据，将面临以下问题：

```
- MapReduce 开发难度大，学习成本高(wordCount => Hello World)
- Hdfs文件没有字段名、没有数据类型，不方便进行数据的有效管理
- 使用MapReduce框架开发，项目周期长，成本高
```

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(类似于RDBMS中的表)，并提供类SQL查询功能；Hive是由Facebook开源，用于解决海量结构化日志的数据统计。

- Hive本质是：将 SQL 转换为 MapReduce 的任务进行运算
- 底层由HDFS来提供数据存储
- 可以将Hive理解为一个：**将 SQL 转换为 MapReduce 任务的工具**

数据仓库(Data Warehouse)是一个面向主题的、集成的、相对稳定的、反映历史变化的数据集合，主要用于管理决策。(数据仓库之父比尔·恩门，1991年提出)。

- 数据仓库的目的：构建面向分析的、集成的数据集合；为企业提供决策支持
- 数据仓库本身不产生数据，数据来源与外部
- 存储了大量数据，对这些数据的分析和处理不可避免的用到Hive

第 2 节 Hive和RDBMS对比

由于 Hive 采用了类似SQL 的查询语言 HQL(Hive Query Language)，因此很容易将 Hive 理解为数据库。其实从结构上来看，Hive 和传统的关系数据库除了拥有类似的查询语言，再无类似之处。

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

数据规模。Hive存储海量数据；RDBMS只能处理有限的数据集；
由于Hive建立在集群上并可以利用MapReduce进行并行计算，因此可以支持很大规模的数据；而RDBMS可以支持的数据规模较小。

执行引擎。Hive的引擎是MR/Tez/Spark/Flink；RDBMS使用自己的执行引擎
Hive中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的。而RDBMS通常有自己的执行引擎。

数据存储。Hive保存在HDFS上；RDBMS保存在本地文件系统 或 裸设备
Hive 的数据都是存储在 HDFS 中的。而RDBMS是将数据保存在本地文件系统或裸设备中。

执行速度。Hive相对慢（MR/数据量）；RDBMS相对快；
Hive存储的数据量大，在查询数据的时候，通常没有索引，需要扫描整个表；加之Hive使用MapReduce作为执行引擎，这些因素都会导致较高的延迟。而RDBMS对数据的访问通常是基于索引的，执行延迟较低。当然这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive的并行计算显然能体现出并行的优势。

可扩展性。Hive支持水平扩展；通常RDBMS支持垂直扩展，对水平扩展不友好
Hive建立在Hadoop之上，其可扩展性与Hadoop的可扩展性是一致的（Hadoop集群规模可以轻松超过1000个节点）。而RDBMS由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有100台左右。

数据更新。Hive对数据更新不友好；RDBMS支持频繁、快速数据更新

Hive是针对数据仓库应用设计的，数据仓库的内容是读多写少的。因此，Hive中不建议对数据的改写，所有的数据都是在加载的时候确定好的。而RDBMS中的数据需要频繁、快速的进行更新。

第 3 节 Hive的优缺点

Hive的优点

学习成本低。Hive提供了类似SQL的查询语言，开发人员能快速上手；

处理海量数据。底层执行的是MapReduce 任务；

系统可以水平扩展。底层基于Hadoop；

功能可以扩展。Hive允许用户自定义函数；

良好的容错性。某个节点发生故障，HQL仍然可以正常完成；

统一的元数据管理。元数据包括：有哪些表、表有什么字段、字段是什么类型

Hive的缺点

HQL表达能力有限；

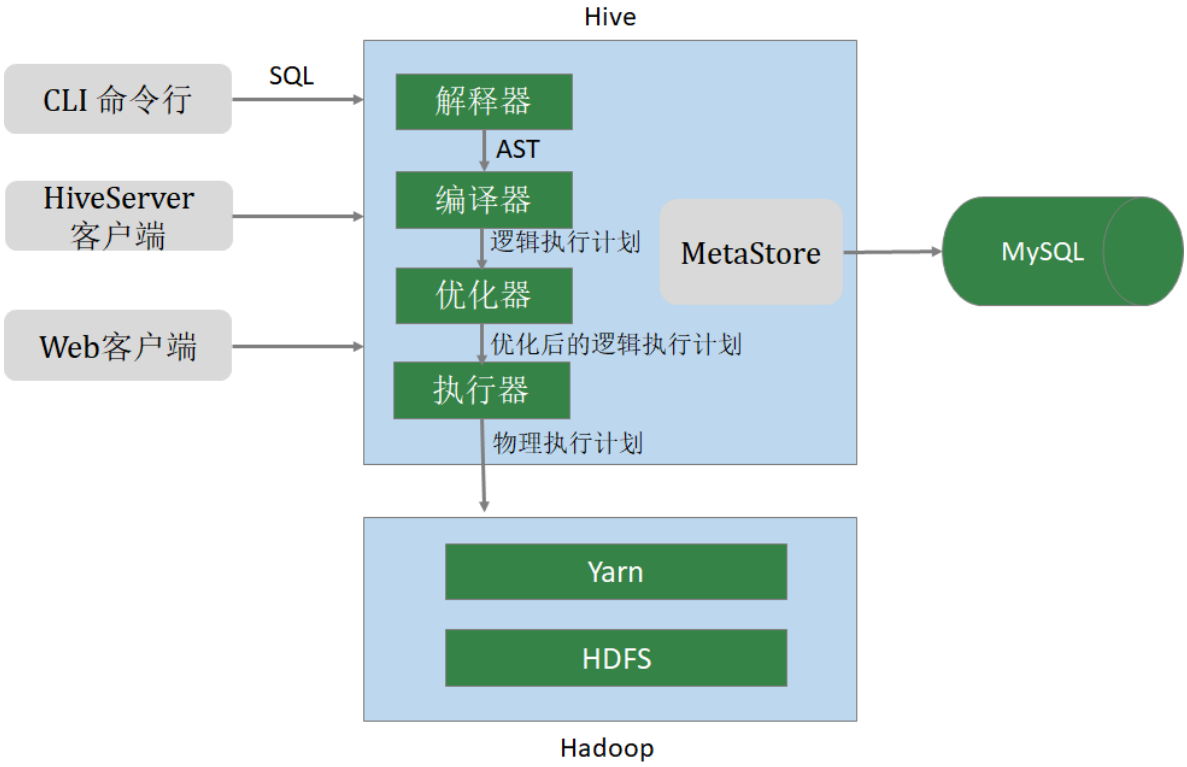
迭代计算无法表达；

Hive的执行效率不高(基于MR的执行引擎)；

Hive自动生成的MapReduce作业，某些情况下不够智能；

Hive的调优困难；

第 4 节 Hive架构



1. **用户接口** CLI(Common Line Interface)：Hive的命令行，用于接收HQL，并返回结果； JDBC/ODBC：是指Hive的java实现，与传统数据库JDBC类似；WebUI：是指可通过浏览器访问Hive；
2. **Thrift Server**

   Hive可选组件，是一个软件框架服务，允许客户端使用包括Java、C++、Ruby和其他很多种语言，通过 编程的方式远程访问Hive；
3. **元数据管理(MetaStore)** Hive将元数据存储在关系数据库中(如mysql、derby)。Hive的元数据包括：数据库名、表名及类型、字段名称及数据类型、数据所在位置等；
4. **驱动程序(Driver)**
   - **解析器** (SQLParser)： 使用第三方工具（antlr）将HQL字符串转换成抽象语法树（AST）；对AST进行语法分析，比如字段是否存在、SQL语义是否有误、表是否存在；
   - **编译器** (Compiler)： 将抽象语法树编译生成逻辑执行计划；
   - **优化器** (Optimizer)： 对逻辑执行计划进行优化，减少不必要的列、使用分区等；
   - **执行器** (Executr)： 把逻辑执行计划转换成可以运行的物理计划；

第二部分 Hive安装与配置

第 1 节 Hive安装配置

Hive官网：http://hive.apache.org

下载网址：http://archive.apache.org/dist/hive/

文档网址：https://cwiki.apache.org/confluence/display/Hive/LanguageManual

安装前提：3台虚拟机，安装了Hadoop

安装软件：Hive(2.3.7) + MySQL (5.7.26)

备注：Hive的元数据默认存储在自带的 derby 数据库中，生产中多采用MySQL

derby：java语言开发占用资源少，单进程，单用户。仅仅适用于个人的测试。

软件	linux121	linux122	linux123
Hadoop	√	√	√
MySQL			√
Hive			√

```
# hive安装包
apache-hive-2.3.7-bin.tar.gz

# MySQL安装包
mysql-5.7.26-1.el7.x86_64.rpm-bundle.tar

# MySQL的JDBC驱动程序
mysql-connector-java-5.1.46.jar

# 整体的安装步骤：
1、安装MySQL
2、安装配置Hive
3、Hive添加常用配置
```

2.1.1、MySQL安装

Hive中使用MySQL存储元数据，MySQL的版本 5.7.26。安装步骤：

```
1、环境准备(删除有冲突的依赖包、安装必须的依赖包)
2、安装MySQL
3、修改root口令(找到系统给定的随机口令、修改口令)
4、在数据库中创建hive用户
```

1、删除MariaDB

centos7.6自带的 MariaDB(MariaDB是MySQL的一个分支)，与要安装的MySQL有冲突，需要删除。

```
# 查询是否安装了mariadb
rpm -aq | grep mariadb

# 删除mariadb。-e 删除指定的套件；--nodeps 不验证套件的相互关联性
rpm -e --nodeps mariadb-libs
```

2、安装依赖

```
yum install perl -y
yum install net-tools -y
```

3、安装MySQL

```
# 解压缩
tar xvf mysql-5.7.26-1.el7.x86_64.rpm-bundle.tar

# 依次运行以下命令
rpm -ivh mysql-community-common-5.7.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-libs-5.7.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-client-5.7.26-1.el7.x86_64.rpm
rpm -ivh mysql-community-server-5.7.26-1.el7.x86_64.rpm
```

4、启动数据库

```
systemctl start mysqld
```

5、查找root密码

```
grep password /var/log/mysqld.log
```

6、修改 root 口令

```
# 进入MySQL，使用前面查询到的口令
mysql -u root -p

# 设置口令强度；将root口令设置为12345678；刷新
set global validate_password_policy=0;
set password for 'root'@'localhost' =password('12345678');
flush privileges;
```

validate_password_policy 密码策略(默认是1)，可配置的值有以下：

- 0 or LOW 仅需需符合密码长度（由参数validate_password_length【默认为8】指定）
- 1 or MEDIUM 满足LOW策略，同时还需满足至少有1个数字，小写字母，大写字母和特殊字符
- 2 or STRONG 满足MEDIUM策略，同时密码不能存在字典文件（dictionary file）中

备注：个人开发环境，出于方便的目的设比较简单的密码；生产环境一定要设复杂密码！

7、创建 hive 用户

```
-- 创建用户设置口令、授权、刷新
CREATE USER 'hive'@'%' IDENTIFIED BY '12345678';
GRANT ALL ON *.* TO 'hive'@'%';
FLUSH PRIVILEGES;
```

2.1.2、Hive 安装

安装步骤：
1、下载、上传、解压缩
2、修改环境变量
3、修改hive配置
4、拷贝JDBC的驱动程序
5、初始化元数据库

1、下载Hive软件，并解压缩

```
cd /opt/software
tar zxvf apache-hive-2.3.7-bin.tar.gz -C ../servers/
cd ../servers
mv apache-hive-2.3.7-bin hive-2.3.7
```

2、修改环境变量

```
# 在 /etc/profile 文件中增加环境变量
export HIVE_HOME=/opt/servers/hive-2.3.7
export PATH=$PATH:$HIVE_HOME/bin

# 执行并生效
source /etc/profile
```

3、修改 Hive 配置

cd $HIVE_HOME/conf vi hive-site.xml 增加以下内容：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <!-- hive元数据的存储位置 -->
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:mysql://linux123:3306/hivemetadata?
createDatabaseIfNotExist=true&amp;useSSL=false</value>
        <description>JDBC connect string for a JDBC
metastore</description>
    </property>

    <!-- 指定驱动程序 -->
```

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

    <!-- 连接数据库的用户名 -->
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>hive</value>
        <description>username to use against metastore
database</description>
    </property>

    <!-- 连接数据库的口令 -->
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>12345678</value>
        <description>password to use against metastore
database</description>
    </property>
</configuration>
```

备注：

- 注意jdbc的连接串，如果没有 useSSL=false 会有大量警告
- 在xml文件中 `&` 表示 `&`

4、拷贝 MySQL JDBC 驱动程序

将 mysql-connector-java-5.1.46.jar 拷贝到 $HIVE_HOME/lib

5、初始化元数据库

```
[root@linux123 ~]$ schematool -dbType mysql -initSchema
```

6、启动Hive，执行命令

```
# 启动hive服务之前，请先启动hdfs、yarn的服务
[root@linux123 ~]$ hive
hive> show functions;
```

2.1.3、Hive 属性配置

可在 hive-site.xml 中增加以下常用配置，方便使用。

数据存储位置

```
<property>
    <!-- 数据默认的存储位置(HDFS) -->
    <name>hive.metastore.warehouse.dir</name>
    <value>/user/hive/warehouse</value>
    <description>location of default database for the
warehouse</description>
</property>
```

显示当前库

```
<property>
    <!-- 在命令行中，显示当前操作的数据库 -->
    <name>hive.cli.print.current.db</name>
    <value>true</value>
    <description>Whether to include the current database in the
Hive prompt.</description>
</property>
```

显示表头属性

```
<property>
    <!-- 在命令行中，显示数据的表头 -->
    <name>hive.cli.print.header</name>
    <value>true</value>
</property>
```

本地模式

```
<property>
    <!-- 操作小规模数据时，使用本地模式，提高效率 -->
    <name>hive.exec.mode.local.auto</name>
    <value>true</value>
    <description>Let Hive determine whether to run in local mode
automatically</description>
</property>
```

备注：当 Hive 的输入数据量非常小时，Hive 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，执行时间会明显被缩短。当一个job满足如下条件才能真正使用本地模式：

- job的输入数据量必须小于参数：hive.exec.mode.local.auto.inputbytes.max (默认128MB)
- job的map数必须小于参数：hive.exec.mode.local.auto.tasks.max (默认4)
- job的reduce数必须为0或者1

Hive的日志文件

Hive的log默认存放在 /tmp/root 目录下（root为当前用户名）；这个位置可以修改。

```
vi $HIVE_HOME/conf/hive-log4j2.properties

# 添加以下内容：
property.hive.log.dir = /opt/servers/hive-2.3.7/logs
```

可以不修改，但是要知道位置。

Hadoop 2.x 中 NameNode RPC缺省的端口号：8020

对端口号要敏感

附录：添加第三方用户（Hadoop）

```
groupadd hadoop
# -m：自动建立用户的登入目录
# -g：指定用户所属的起始群组
# -G<群组>：指定用户所属的附加群组
# —s：指定用户登入后所使用的shell
useradd -m hadoop -g hadoop -s /bin/bash

passwd hadoop

visudo
# 在100行后添加。允许用户执行sudo，免密
hadoop  ALL=(ALL)       NOPASSWD:ALL
```

建议：现阶段使用root用户

小结：

1、添加了配置，使用Hive更方便；

2、删除了有冲突的软件包(hive)

3、Hive的日志在哪里(/tmp/root)

4、第三方用户使用Hive。建议使用root用户

5、NameNode 缺省的RPC(远程过程调用)端口号8020，经常使用的端口号9000

2.1.4、参数配置方式

查看参数配置信息：

```
-- 查看全部参数
hive> set;

-- 查看某个参数
hive> set hive.exec.mode.local.auto;
hive.exec.mode.local.auto=false
```

参数配置的三种方式：

```
1、用户自定义配置文件(hive-site.xml)
2、启动hive时指定参数(-hiveconf)
3、hive命令行指定参数(set)

配置信息的优先级：
set > -hiveconf > hive-site.xml > hive-default.xml
```

1、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

配置优先级：hive-site.xml > hive-default.xml

配置文件的设定对本机启动的所有Hive进程有效；

配置文件的设定对本机所有启动的Hive进程有效；

2、启动时指定参数值

启动Hive时，可以在命令行添加 -hiveconf param=value 来设定参数，这些设定仅对本次启动有效。

```
# 启动时指定参数
hive -hiveconf hive.exec.mode.local.auto=true

# 在命令行检查参数是否生效
hive> set hive.exec.mode.local.auto;
hive.exec.mode.local.auto=true
```

3、命令行修改参数

可在 Hive 命令行中使用SET关键字设定参数，同样仅对本次启动有效

```
hive> set hive.exec.mode.local.auto=false;
hive> set hive.exec.mode.local.auto;
hive.exec.mode.local.auto=false
```

set > -hiveconf > hive-site.xml > hive-default.xml

第 2 节 Hive命令

1、Hive

```
hive -help
usage: hive
 -d,--define <key=value>          Variable substitution to apply
to Hive
                                  commands. e.g. -d A=B or --
define A=B
    --database <databasename>     Specify the database to use
 -e <quoted-query-string>         SQL from command line
 -f <filename>                    SQL from files
 -H,--help                        Print help information
    --hiveconf <property=value>   Use value for given property
    --hivevar <key=value>         Variable substitution to apply
to Hive
                                  commands. e.g. --hivevar A=B
 -i <filename>                    Initialization SQL file
 -S,--silent                      Silent mode in interactive
shell
 -v,--verbose                     Verbose mode (echo executed SQL
to the
                                  console)
```

-e：不进入hive交互窗口，执行sql语句

```
hive -e "select * from users"
```

-f：执行脚本中sql语句

```
# 创建文件hqlfile1.sql，内容：select * from users

# 执行文件中的SQL语句
hive -f hqlfile1.sql

# 执行文件中的SQL语句，将结果写入文件
hive -f hqlfile1.sql >> result1.log
```

2、退出Hive命令行

exit; quit;

3、在命令行执行 shell 命令 / dfs 命令

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

第三部分 数据类型与文件格式

Hive支持关系型数据库的绝大多数**基本数据类型**，同时也支持4种**集合数据类型**。

第 1 节 基本数据类型及转换

Hive类似和java语言中一样，会支持多种不同长度的整型和浮点类型数据，同时也支持布尔类型、字符串类型，时间戳数据类型以及二进制数组数据类型等。详细信息见下表：

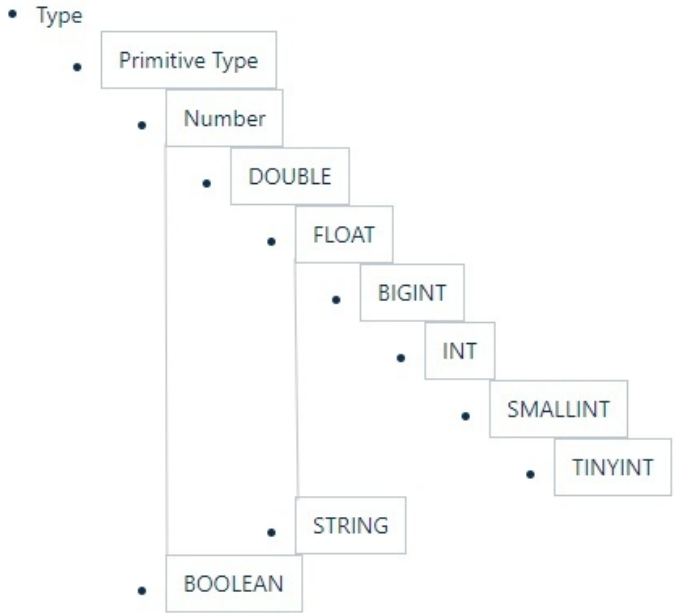
大类	类型
Integers(整型)	TINYINT -- 1字节的有符号整数 SAMLINT -- 2字节的有符号整数 INT -- 4字节的有符号整数 BIGINT -- 8字节的有符号整数
Floating point numbers(浮点数)	FLOAT -- 单精度浮点数 DOUBLE -- 双精度浮点数
Fixed point numbers(定点数)	DECIMAL--用户自定义精度定点数，如 DECIMAL(10,3)
String types(字符串)	STRTIMESTAMP -- 时间戳 TIMESTAMP WITH LOCAL TIME ZONE -- 时间戳，纳秒精度 DATE -- 日期类型
Boolean(布尔类型)	BOOLEAN -- TRUE / FALSE
Binary types(二进制类型)	BINARY -- 字节序列

这些类型名称都是 Hive 中保留字。这些基本的数据类型都是 java 中的接口进行实现的，因此与 java 中数据类型是基本一致的：

Hive数据类型	Java数据类型	长度	样例
TINYINT		1byte有符号整数	20
SMALLINT		2byte有符号整数	30
INT		4byte有符号整数	40
BIGINT		8byte有符号整数	50
BOOLEAN		布尔类型	TURE / FALSE
FLOAT		单精度浮点数	3.14159
DOUBLE		双精度浮点数	3.14159
STRING		字符系列，可指定字符集；可使用单引号或双引号	'The Apache Hive data warehouse software facilitates'
TIMESTAMP		时间类型	
BINARY		字节数组	

数据类型的隐式转换

Hive的数据类型是可以进行隐式转换的，类似于Java的类型转换。如用户在查询中将一种浮点类型和另一种浮点类型的值做对比，Hive会将类型转换成两个浮点类型中值较大的那个类型，即：将FLOAT类型转换成DOUBLE类型；当然如果需要的话，任意整型会转化成DOUBLE类型。 Hive 中基本数据类型遵循以下层次结构，按照这个层次结构，子类型到祖先类型允许隐式转换。



总的来说数据转换遵循以下规律：

-

```
hive> select '1.0'+2;
OK
3.0
hive> select '1111' > 10;
hive> select 1 > 0.8;
```

数据类型的显示转换

使用cast函数进行强制类型转换；如果强制类型转换失败，返回NULL

```
hive> select cast('1111s' as int);
OK
NULL
hive> select cast('1111' as int);
OK
1111
```

第 2 节 集合数据类型

Hive支持集合数据类型，包括**array、map、struct、union**

类型	描述	字面量示例
ARRAY	有序的相同数据类型的集合	array(1,2)
MAP	key-value对。key必须是基本数据类型，value不限	map('a', 1, 'b',2)
STRUCT	不同类型字段的集合。类似于C语言的结构体	struct('1',1,1.0), named_struct('col1', '1', 'col2', 1, 'clo3', 1.0)
UNION	不同类型的元素存储在同一字段的不同行中	create_union(1, 'a', 63)

和基本数据类型一样，这些类型的名称同样是保留字；

ARRAY 和 MAP 与 Java 中的 Array 和 Map 类似；

STRUCT 与 C 语言中的 Struct 类似，它封装了一个命名字段集合，复杂数据类型允许任意层次的嵌套；

```
hive> select array(1,2,3);
OK
[1,2,3]

-- 使用 [] 访问数组元素
hive> select arr[0] from (select array(1,2,3) arr) tmp;

hive> select map('a', 1, 'b', 2, 'c', 3);
OK
{"a":1,"b":2,"c":3}

-- 使用 [] 访问map元素
hive> select mymap["a"] from (select map('a', 1, 'b', 2, 'c', 3)
as mymap) tmp;

-- 使用 [] 访问map元素。 key 不存在返回 NULL
hive> select mymap["x"] from (select map('a', 1, 'b', 2, 'c', 3)
as mymap) tmp;
NULL

hive> select struct('username1', 7, 1288.68);
OK
{"col1":"username1","col2":7,"col3":1288.68}

-- 给 struct 中的字段命名
```

```
hive> select named_struct("name", "username1", "id", 7, "salary",
12880.68);
OK
{"name":"username1","id":7,"salary":12880.68}

-- 使用 列名.字段名 访问具体信息
hive> select userinfo.id
    >   from (select named_struct("name", "username1", "id", 7,
"salary", 12880.68) userinfo) tmp;

-- union 数据类型
hive> select create_union(0, "zhansan", 19, 8000.88) uinfo;
```

第 3 节 文本文件数据编码

Hive表中的数据在存储在文件系统上，Hive定义了默认的存储格式，也支持用户自定义文件存储格式。

Hive默认使用几个很少出现在字段值中的控制字符，来表示替换默认分隔符的字符。

Hive默认分隔符

```
id name age hobby(array) score(map)
字段之间：^A
元素之间：^B
key-value之间：^C
666^Alisi^A18^Aread^Bgame^Ajava^C97^Bhadoop^C87

create table s1(
  id int,
  name string,
  age int,
  hobby array<string>,
  score map<string, int>
);
load data local inpath '/home/hadoop/data/s1.dat' into table s1;
select * from s1;
```

分隔符	名称	说明
\n	换行符	用于分隔行。每一行是一条记录，使用换行符分割数据
^A	<Ctrl>+A	用于分隔字段。在CREATE TABLE语句中使用八进制编码\001表示
^B	<Ctrl>+B	用于分隔 ARRAY、MAP、STRUCT 中的元素。在CREATE TABLE语句中使用八进制编码\002表示
^C	<Ctrl+C>	Map中 key、value之间的分隔符。在CREATE TABLE语句中使用八进制编码\003表示

Hive 中没有定义专门的数据格式，数据格式可以由用户指定，用户定义数据格式需要指定三个属性：列分隔符（通常为空格、"\t"、"\x001"）、行分隔符（"\n"）以及读取文件数据的方法。

在加载数据的过程中，Hive 不会对数据本身进行任何修改，而只是将数据内容复制或者移动到相应的 HDFS 目录中。

将 Hive 数据导出到本地时，系统默认的分隔符是^A、^B、^C 这些特殊字符，使用 cat 或者 vim 是看不到的；

在 vi 中输入特殊字符：

- (Ctrl + v) + (Ctrl + a) => ^A
- (Ctrl + v) + (Ctrl + b) => ^B
- (Ctrl + v) + (Ctrl + c) => ^C

^A / ^B / ^C 都是特殊的控制字符，使用 more 、 cat 命令是看不见的；可以使用 `cat -A file.dat`

第 4 节 读时模式

在传统数据库中，在加载时发现数据不符合表的定义，则拒绝加载数据。数据在写入数据库时对照表模式进行检查，这种模式称为"写时模式"（schema on write）。

写时模式 -> 写数据检查 -> RDBMS；

Hive中数据加载过程采用"读时模式" (schema on read)，加载数据时不进行数据格式的校验，读取数据时如果不合法则显示NULL。这种模式的优点是加载数据迅速。

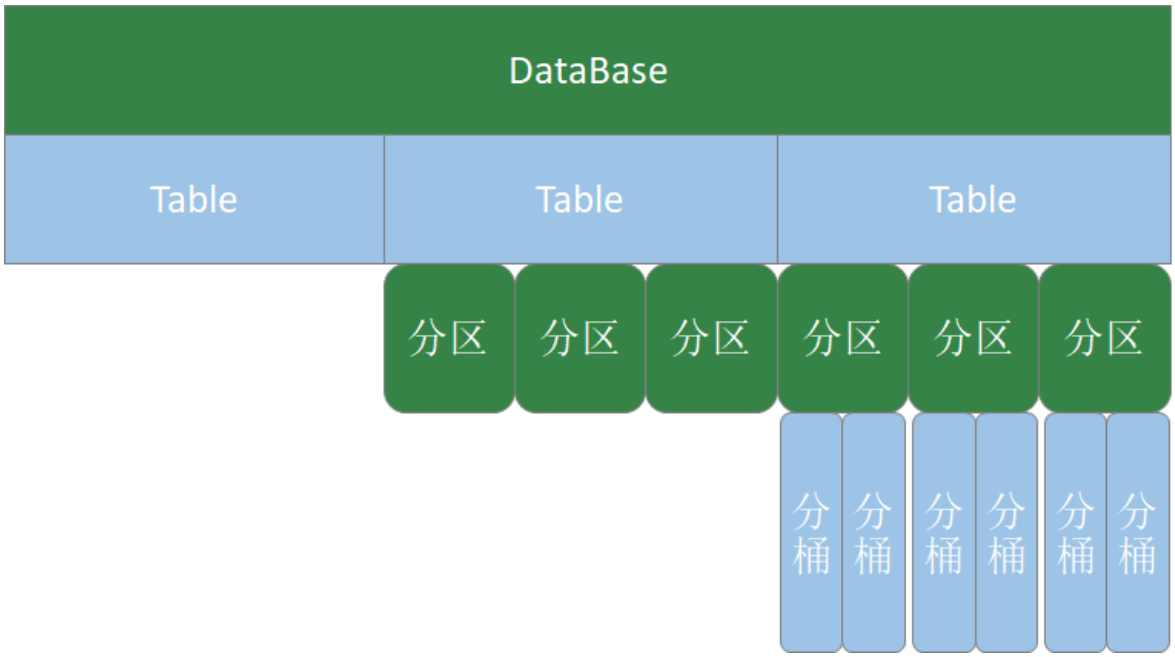
读时模式 -> 读时检查数据 -> Hive；好处：加载数据快；问题：数据显示NULL

第四部分 HQL操作之 -- DDL命令

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

DDL（data definition language）： 主要的命令有CREATE、ALTER、DROP等。

DDL主要是用在定义、修改数据库对象的结构 或 数据类型。



第 1 节 数据库操作

Hive有一个默认的数据库default，在操作HQL时，如果不明确的指定要使用哪个库，则使用默认数据库；

Hive的数据库名、表名均不区分大小写；

名字不能使用数字开头；

不能使用关键字，尽量不使用特殊符号；

创建数据库语法

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

```
-- 创建数据库，在HDFS上存储路径为 /user/hive/warehouse/*.db
hive (default)> create database mydb;
hive (default)> dfs -ls /user/hive/warehouse;

-- 避免数据库已经存在时报错，使用 if not exists 进行判断【标准写法】
hive (default)> create database if not exists mydb;

-- 创建数据库。添加备注，指定数据库在存放位置
hive (default)> create database if not exists mydb2
comment 'this is mydb2'
location '/user/hive/mydb2.db';
```

查看数据库

```
-- 查看所有数据库
show database;

-- 查看数据库信息
desc database mydb2;
desc database extended mydb2;
describe database extended mydb2;
```

使用数据库

```
use mydb;
```

删除数据库

```
-- 删除一个空数据库
drop database databasename;

-- 如果数据库不为空，使用 cascade 强制删除
drop database databasename cascade;
```

第 2 节 建表语法

```
```

```
create [external] table [IF NOT EXISTS] table_name
[(colName colType [comment 'comment'], ...)]
[comment table_comment]
[partition by (colName colType [comment col_comment], ...)]
[clustered BY (colName, colName, ...)
[sorted by (col_name [ASC|DESC], ...)] into num_buckets buckets]
[row format row_format]
[stored as file_format]
[LOCATION hdfs_path]
[TBLPROPERTIES (property_name=property_value, ...)]
[AS select_statement];

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS]
[db_name.]table_name
  LIKE existing_table_or_view_name
  [LOCATION hdfs_path];
```

1. CREATE TABLE。按给定名称创建表，如果表已经存在则抛出异常。可使用if not exists 规避。
2. EXTERNAL关键字。创建外部表，否则创建的是内部表(管理表)。

   删除内部表时，数据和表的定义同时被删除；

   删除外部表时，仅仅删除了表的定义，数据保留；

   在生产环境中，多使用外部表；
3. comment。表的注释
4. partition by。对表中数据进行分区，指定表的分区字段
5. clustered by。创建分桶表，指定分桶字段
6. sorted by。对桶中的一个或多个列排序，较少使用
7. 存储子句。

   ```
   ROW FORMAT DELIMITED
   [FIELDS TERMINATED BY char]
   [COLLECTION ITEMS TERMINATED BY char]
   [MAP KEYS TERMINATED BY char]
   [LINES TERMINATED BY char] | SERDE serde_name
   [WITH SERDEPROPERTIES (property_name=property_value,
   property_name=property_value, ...)]
   ```

   建表时可指定 SerDe 。如果没有指定 ROW FORMAT 或者 ROW FORMAT DELIMITED，将会使用默认的 SerDe。建表时还需要为表指定列，在指定列的同时也会指定自定义的 SerDe。Hive通过 SerDe 确定表的具体的列的数据。

SerDe是 Serialize/Deserilize 的简称， hive使用Serde进行行对象的序列与反序列化。

8. stored as SEQUENCEFILE|TEXTFILE|RCFILE。如果文件数据是纯文本，可以使用 STORED AS TEXTFILE（缺省）；如果数据需要压缩，使用 STORED AS SEQUENCEFILE（二进制序列文件）。
9. LOCATION。表在HDFS上的存放位置
10. TBLPROPERTIES。定义表的属性
11. AS。后面可以接查询语句，表示根据后面的查询结果创建表
12. LIKE。like 表名，允许用户复制现有的表结构，但是不复制数据

第 3 节 内部表 & 外部表

在创建表的时候，可指定表的类型。表有两种类型，分别是内部表(管理表)、外部表。

- 默认情况下，创建内部表。如果要创建外部表，需要使用关键字 external
- 在删除内部表时，表的定义(元数据) 和 数据 同时被删除
- 在删除外部表时，仅删除表的定义，数据被保留
- 在生产环境中，多使用外部表

内部表

t1.dat文件内容

```
2;zhangsan;book,TV,code;beijing:chaoyang,shagnhai:pudong
3;lishi;book,code;nanjing:jiangning,taiwan:taibei
4;wangwu;music,book;heilongjiang:haerbin
```

创建表 SQL

```
-- 创建内部表
create table t1(
    id int,
    name string,
    hobby array<string>,
    addr  map<string, string>
)
row format delimited
```

```
fields terminated by ";"
collection items terminated by ","
map keys terminated by ":";

-- 显示表的定义，显示的信息较少
desc t1;

-- 显示表的定义，显示的信息多，格式友好
desc formatted t1;

-- 加载数据
load data local inpath '/home/hadoop/data/t1.dat' into table t1;

-- 查询数据
select * from t1;

-- 查询数据文件
dfs -ls /user/hive/warehouse/mydb.db/t1;

-- 删除表。表和数据同时被删除
drop table t1;

-- 再次查询数据文件，已经被删除
```

外部表

```
-- 创建外部表
create external table t2(
    id int,
    name string,
    hobby array<string>,
    addr  map<string, string>
)
row format delimited
fields terminated by ";"
collection items terminated by ","
map keys terminated by ":";

-- 显示表的定义
desc formatted t2;

-- 加载数据
load data local inpath '/home/hadoop/data/t1.dat' into table t2;
```

```
-- 查询数据
select * from t2;

-- 删除表。表删除了，目录仍然存在
drop table t2;

-- 再次查询数据文件，仍然存在
```

内部表与外部表的转换

```
-- 创建内部表，加载数据，并检查数据文件和表的定义
create table t1(
    id int,
    name string,
    hobby array<string>,
    addr  map<string, string>
)
row format delimited
fields terminated by ";"
collection items terminated by ","
map keys terminated by ":";
load data local inpath '/home/hadoop/data/t1.dat' into table t1;
dfs -ls /user/hive/warehouse/mydb.db/t1;
desc formatted t1;

-- 内部表转外部表
alter table t1 set tblproperties('EXTERNAL'='TRUE');
-- 查询表信息，是否转换成功
desc formatted t1;

-- 外部表转内部表。EXTERNAL 大写，false 不区分大小
alter table t1 set tblproperties('EXTERNAL'='FALSE');
-- 查询表信息，是否转换成功
desc formatted t1;
```

小结

- 建表时：
    - 如果不指定external关键字，创建的是内部表；
    - 指定external关键字，创建的是外部表；

- 删表时
  - 删除外部表时，仅删除表的定义，表的数据不受影响
  - 删除内部表时，表的数据和定义同时被删除
- 外部表的使用场景
  - 想保留数据时使用。生产多用外部表

第 4 节 分区表

Hive在执行查询时，一般会扫描整个表的数据。由于表的数据量大，全表扫描消耗时间长、效率低。

而有时候，查询只需要扫描表中的一部分数据即可，Hive引入了分区表的概念，将表的数据存储在不同的子目录中，每一个子目录对应一个分区。只查询部分分区数据时，可避免全表扫描，提高查询效率。

在实际中，通常根据时间、地区等信息进行分区。

分区表创建与数据加载

```
-- 创建表
create table if not exists t3(
    id      int
   ,name    string
   ,hobby   array<string>
   ,addr    map<String,string>
)
partitioned by (dt string)
row format delimited
fields terminated by ';'
collection items terminated by ','
map keys terminated by ':';

-- 加载数据。
load data local inpath "/home/hadoop/data/t1.dat" into table t3
partition(dt="2020-06-01");
load data local inpath "/home/hadoop/data/t1.dat" into table t3
partition(dt="2020-06-02");
```

备注：分区字段不是表中已经存在的数据，可以将分区字段看成伪列

查看分区

```
show partitions t3;
```

新增分区并设置数据

```
-- 增加一个分区，不加载数据
alter table t3 add partition(dt='2020-06-03');

-- 增加多个分区，不加载数据
alter table t3
add partition(dt='2020-06-05') partition(dt='2020-06-06');

-- 增加多个分区。准备数据
hdfs dfs -cp /user/hive/warehouse/mydb.db/t3/dt=2020-06-01
/user/hive/warehouse/mydb.db/t3/dt=2020-06-07
hdfs dfs -cp /user/hive/warehouse/mydb.db/t3/dt=2020-06-01
/user/hive/warehouse/mydb.db/t3/dt=2020-06-08

-- 增加多个分区。加载数据
alter table t3 add
partition(dt='2020-06-07') location
'/user/hive/warehouse/mydb.db/t3/dt=2020-06-07'
partition(dt='2020-06-08') location
'/user/hive/warehouse/mydb.db/t3/dt=2020-06-08';

-- 查询数据
select * from t3;
```

修改分区的hdfs路径

```
alter table t3 partition(dt='2020-06-01') set location
'/user/hive/warehouse/t3/dt=2020-06-03';
```

删除分区

```
-- 可以删除一个或多个分区，用逗号隔开
alter table t3 drop partition(dt='2020-06-03'),
partition(dt='2020-06-04');
```

第 5 节 分桶表

当单个的分区或者表的数据量过大，分区不能更细粒度的划分数据，就需要使用分桶技术将数据划分成更细的粒度。将数据按照指定的字段进行分成多个桶中去，即将数据按照字段进行划分，数据按照字段划分到多个文件当中去。分桶的原理：

- MR中：key.hashCode % reductTask
- Hive中：分桶字段.hashCode % 分桶个数

```
-- 测试数据
1	java	90
1	c	78
1	python	91
1	hadoop	80
2	java	75
2	c	76
2	python	80
2	hadoop	93
3	java	98
3	c	74
3	python	89
3	hadoop	91
5	java	93
6	c	76
7	python	87
8	hadoop	88
```

```
-- 创建分桶表
create table course(
    id int,
    name string,
    score int
)
clustered by (id) into 3 buckets
row format delimited fields terminated by "\t";

-- 创建普通表
```

```
create table course_common(
    id int,
    name string,
    score int
)
row format delimited fields terminated by "\t";

-- 普通表加载数据
load data local inpath '/home/hadoop/data/course.dat' into table
course_common;

-- 通过 insert ... select ... 给桶表加载数据
insert into table course select * from course_common;

-- 观察分桶数据。数据按照：(分区字段.hashCode) % (分桶数) 进行分区
```

备注：

- 分桶规则：分桶字段.hashCode % 分桶数
- 分桶表加载数据时，使用 insert... select ... 方式进行
- 网上有资料说要使用分区表需要设置 hive.enforce.bucketing=true，那是Hive 1.x 以前的版本；Hive 2.x 中，删除了该参数，始终可以分桶；

第 6 节 修改表 & 删除表

```
-- 修改表名。rename
alter table course_common
rename to course_common1;

-- 修改列名。change column
alter table course_common1
change column id cid int;

-- 修改字段类型。change column
alter table course_common1
change column cid cid string;
-- The following columns have types incompatible with the
existing columns in their respective positions
-- 修改字段数据类型时，要满足数据类型转换的要求。如int可以转为string，但是
string不能转为int

-- 增加字段。add columns
alter table course_common1
```

```
add columns (common string);

-- 删除字段：replace columns
-- 这里仅仅只是在元数据中删除了字段，并没有改动hdfs上的数据文件
alter table course_common1
replace columns(
    id string, cname string, score int);

-- 删除表
drop table course_common1;
```

HQL DDL命令小结：

- 主要对象：数据库、表
- 表的分类：
  - 内部表。删除表时，同时删除元数据和表数据
  - 外部表。删除表时，仅删除元数据，保留表中数据；生产环境多使用外部表
  - 分区表。按照分区字段将表中的数据放置在不同的目录中，提高SQL查询的性能
  - 分桶表。按照分桶字段，将表中数据分开。 分桶字段.hashCode % 分桶数据
- 主要命令：create、alter 、drop

第五部分 HQL操作之--数据操作

第 1 节 数据导入

装载数据(Load)

基本语法：

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

- LOCAL：
  - LOAD DATA LOCAL ... 从本地文件系统加载数据到Hive表中。本地文件会拷贝到Hive表指定的位置

    - LOAD DATA ... 从HDFS加载数据到Hive表中。HDFS文件移动到Hive表指定的位置
- INPATH：加载数据的路径
- OVERWRITE：覆盖表中已有数据；否则表示追加数据
- PARTITION：将数据加载到指定的分区

准备工作：

```
-- 创建表
CREATE TABLE tabA (
  id int
 ,name string
 ,area string
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ;

数据文件（~/data/sourceA.txt）：
1,fish1,SZ
2,fish2,SH
3,fish3,HZ
4,fish4,QD
5,fish5,SR

-- 拷贝文件到 HDFS
hdfs dfs -put sourceA.txt data/
```

装载数据：

```
-- 加载本地文件到hive(tabA)
LOAD DATA LOCAL INPATH '/home/hadoop/data/sourceA.txt'
INTO TABLE tabA;
-- 检查本地文件还在

-- 加载hdfs文件到hive(tabA)
LOAD DATA INPATH 'data/sourceA.txt'
INTO TABLE tabA;
-- 检查HDFS文件，已经被转移

-- 加载数据覆盖表中已有数据
LOAD DATA INPATH 'data/sourceA.txt'
OVERWRITE INTO TABLE tabA;
```

```
-- 创建表时加载数据
hdfs dfs -mkdir /user/hive/tabB
hdfs dfs -put sourceA.txt /user/hive/tabB

CREATE TABLE tabB (
  id INT
 ,name string
 ,area string
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
Location '/user/hive/tabB';
```

插入数据(Insert)

```
-- 创建分区表
CREATE TABLE tabC (
  id INT
 ,name string
 ,area string
)
partitioned by (month string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

-- 插入数据
insert into table tabC
partition(month='202001')
values (5, 'wangwu', 'BJ'), (4, 'lishi', 'SH'), (3, 'zhangsan',
'TJ');

-- 插入查询的结果数据
insert into table tabC partition(month='202002')
select id, name, area from tabC where month='202001';

-- 多表（多分区）插入模式
from tabC
insert overwrite table tabC partition(month='202003')
select id, name, area where month='202002'
insert overwrite table tabC partition(month='202004')
select id, name, area where month='202002';
```

创建表并插入数据(as select)

```
-- 根据查询结果创建表
create table if not exists tabD
as select * from tabC;
```

使用import导入数据

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

第 2 节 数据导出

```
-- 将查询结果导出到本地
insert overwrite local directory '/home/hadoop/data/tabC'
select * from tabC;

-- 将查询结果格式化输出到本地
insert overwrite local directory '/home/hadoop/data/tabC2'
row format delimited fields terminated by ' '
select * from tabC;

-- 将查询结果导出到HDFS
insert overwrite directory '/user/hadoop/data/tabC3'
row format delimited fields terminated by ' '
select * from tabC;

-- dfs 命令导出数据到本地。本质是执行数据文件的拷贝
dfs -get /user/hive/warehouse/mydb.db/tabc/month=202001
/home/hadoop/data/tabC4

-- hive 命令导出数据到本地。执行查询将查询结果重定向到文件
hive -e "select * from tabC" > a.log

-- export 导出数据到HDFS。使用export导出数据时，不仅有数还有表的元数据信息
export table tabC to '/user/hadoop/data/tabC4';

-- export 导出的数据，可以使用 import 命令导入到 Hive 表中
-- 使用 like tname创建的表结构与原表一致。create ... as select ... 结构
可能不一致
create table tabE like tabc;
import table tabE from ''/user/hadoop/data/tabC4';
```

```
-- 截断表，清空数据。(注意：仅能操作内部表)
truncate table tabE;

-- 以下语句报错，外部表不能执行 truncate 操作
alter table tabC set tblproperties("EXTERNAL"="TRUE");
truncate table tabC;
```

小结：

数据导入：load data / insert / create table .... as select ..... / import table

数据导出：insert overwrite ... diretory ... / hdfs dfs -get / hive -e "select ..." > a.log / export table ...

Hive的数据导入与导出还可以使用其他工具：Sqoop、DataX等；

第六部分 HQL操作之--DQL命令【重点】

DQL -- Data Query Language 数据查询语言

select语法：

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
  FROM table_reference
  [WHERE where_condition]
  [GROUP BY col_list]
  [ORDER BY col_list]
  [CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY
col_list]]
 [LIMIT [offset,] rows]
```

SQL语句书写注意事项：

- SQL语句对大小写不敏感
- SQL语句可以写一行（简单SQL）也可以写多行(复杂SQL)
- 关键字不能缩写，也不能分行
- 各子句一般要分行
- 使用缩进格式，提高SQL语句的可读性(重要)

创建表，加载数据

```
-- 测试数据 /home/hadoop/data/emp.dat
7369,SMITH,CLERK,7902,2010-12-17,800,,20
7499,ALLEN,SALESMAN,7698,2011-02-20,1600,300,30
7521,WARD,SALESMAN,7698,2011-02-22,1250,500,30
7566,JONES,MANAGER,7839,2011-04-02,2975,,20
7654,MARTIN,SALESMAN,7698,2011-09-28,1250,1400,30
7698,BLAKE,MANAGER,7839,2011-05-01,2850,,30
7782,CLARK,MANAGER,7839,2011-06-09,2450,,10
7788,SCOTT,ANALYST,7566,2017-07-13,3000,,20
7839,KING,PRESIDENT,,2011-11-07,5000,,10
7844,TURNER,SALESMAN,7698,2011-09-08,1500,0,30
7876,ADAMS,CLERK,7788,2017-07-13,1100,,20
7900,JAMES,CLERK,7698,2011-12-03,950,,30
7902,FORD,ANALYST,7566,2011-12-03,3000,,20
7934,MILLER,CLERK,7782,2012-01-23,1300,,10

-- 建表并加载数据
CREATE TABLE emp (
    empno int,
    ename string,
    job string,
    mgr int,
    hiredate DATE,
    sal int,
    comm int,
    deptno int
)row format delimited fields terminated by ",";

-- 加载数据
LOAD DATA LOCAL INPATH '/home/hadoop/data/emp.dat'
INTO TABLE emp;
```

第 1 节 基本查询

```
-- 省略from子句的查询
select 8*888 ;
select current_date ;

-- 使用列别名
select 8*888 product;
select current_date as currdate;
```

```
-- 全表查询
select * from emp;

-- 选择特定列查询
select ename, sal, comm from emp;

-- 使用函数
select count(*) from emp;
-- count(colname) 按字段进行count，不统计NULL
select sum(sal) from emp;
select max(sal) from emp;
select min(sal) from emp;
select avg(sal) from emp;


-- 使用limit子句限制返回的行数
select * from emp limit 3;
```

第 2 节 where子句

WHERE子句紧随FROM子句，使用WHERE子句，过滤不满足条件的数据；

where 子句中不能使用列的别名；

```
select * from emp
 where sal > 2000;
```

where子句中会涉及到较多的比较运算 和 逻辑运算；

比较运算符

官方文档：https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF

比较运算符	描述
=、==、<=>	等于
<>、!=	不等于
<、<=、>、>=	大于等于、小于等于
is [not] null	如果A等于NULL，则返回TRUE，反之返回FALSE。使用NOT关键字结果相反。
in (value1, value2, ...)	匹配列表中的值
LIKE	简单正则表达式，也称通配符模式。'x%' 表示必须以字母 'x' 开头；'%x'表示必须以字母'x'结尾；'%x%'表示包含有字母'x'，可以位于字符串任意位置。使用NOT关键字结果相反。 % 代表匹配零个或多个字符(任意个字符)；_ 代表匹配一个字符。
[NOT] BETWEEN ... AND ...	范围的判断，使用NOT关键字结果相反。
RLIKE、REGEXP	基于java的正则表达式，匹配返回TRUE，反之返回FALSE。匹配使用的是JDK中的正则表达式接口实现的，因为正则也依据其中的规则。例如，正则表达式必须和整个字符串A相匹配，而不是只需与其字符串匹配。

备注：通常情况下NULL参与运算，返回值为NULL；**NULL<=>NULL的结果为true**

逻辑运算符

就是我们所熟悉的：and、or、not

```
-- 比较运算符，null参与运算
select null=null;
select null==null;
select null<=>null;
```

```sql
-- 使用 is null 判空
select * from emp where comm is null;

-- 使用 in
select * from emp where deptno in (20, 30);

-- 使用 between ... and ...
select * from emp where sal between 1000 and 2000;

-- 使用 like
select ename, sal from emp where ename like '%L%';

-- 使用 rlike。正则表达式，名字以A或S开头
select ename, sal from emp where ename rlike '^(A|S).*';
```

第 3 节 group by子句

GROUP BY语句通常与聚组函数一起使用，按照一个或多个列对数据进行分组，对每个组进行聚合操作。

```sql
-- 计算emp表每个部门的平均工资
select deptno, avg(sal)
  from emp
group by deptno;

-- 计算emp每个部门中每个岗位的最高薪水
select deptno, job, max(sal)
  from emp
group by deptno, job;
```

- where子句针对表中的数据发挥作用；having针对查询结果（聚组以后的结果）发挥作用
- where子句不能有分组函数；having子句可以有分组函数
- having只用于group by分组统计之后

```sql
-- 求每个部门的平均薪水大于2000的部门
select deptno, avg(sal)
  from emp
group by deptno
having avg(sal) > 2000;
```

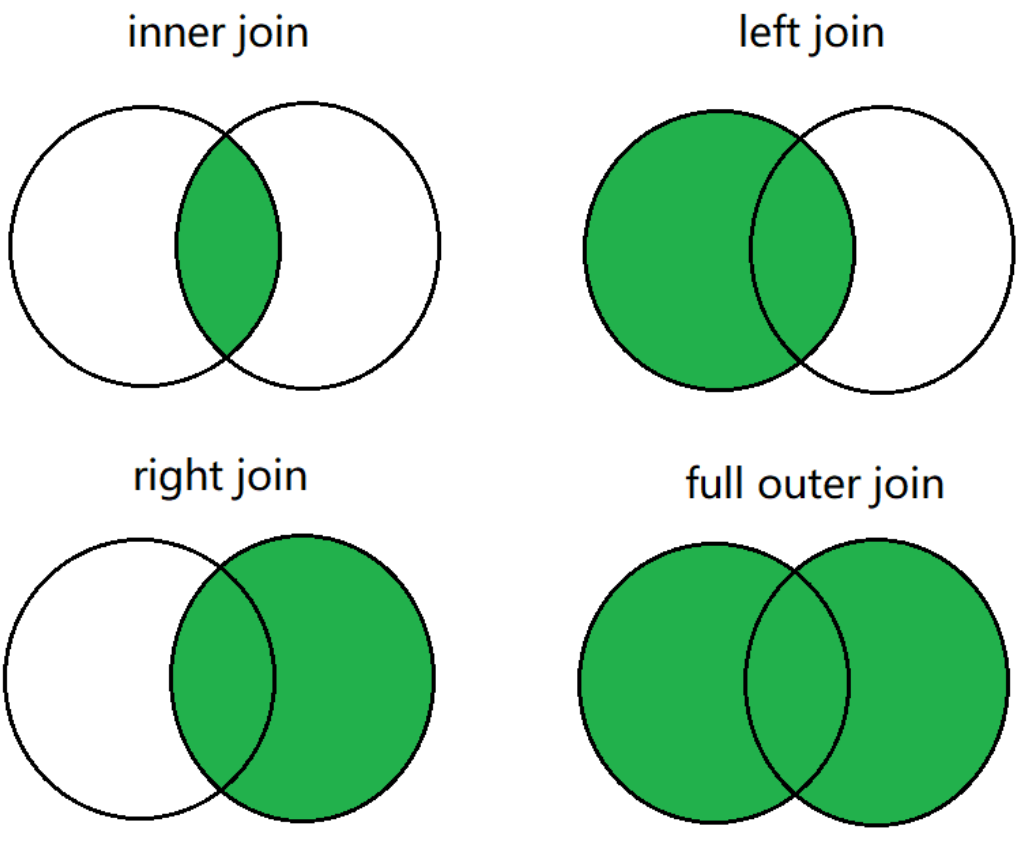
第 4 节 表连接

Hive支持通常的SQL JOIN语句。默认情况下，仅支持等值连接，不支持非等值连接。

JOIN 语句中经常会使用表的别名。使用别名可以简化SQL语句的编写，使用表名前缀可以提高SQL的解析效率。

连接查询操作分为两大类：内连接和外连接，而外连接可进一步细分为三种类型：

```
1. 内连接: [inner] join
2. 外连接 (outer join)
   - 左外连接。 left [outer] join，左表的数据全部显示
   - 右外连接。 right [outer] join，右表的数据全部显示
   - 全外连接。 full [outer] join，两张表的数据都显示
```



案例演示：

```
-- 准备数据
u1.txt数据：
1,a
2,b
3,c
```

```
4,d
5,e
6,f

u2.txt数据：
4,d
5,e
6,f
7,g
8,h
9,i

create table if not exists u1(
id int,
name string)
row format delimited fields terminated by ',';

create table if not exists u2(
id int,
name string)
row format delimited fields terminated by ',';

load data local inpath '/home/hadoop/data/u1.txt' into table u1;
load data local inpath '/home/hadoop/data/u2.txt' into table u2;
```

```
-- 内连接
select * from u1 join u2 on u1.id = u2.id;

-- 左外连接
select * from u1 left join u2 on u1.id = u2.id;

-- 右外连接
select * from u1 right join u2 on u1.id = u2.id;

-- 全外连接
select * from u1 full join u2 on u1.id = u2.id;
```

多表连接

连接 n张表，至少需要 n-1 个连接条件。例如：连接四张表，至少需要三个连接条件。

多表连接查询，查询老师对应的课程，以及对应的分数，对应的学生：

```
select *
  from techer t left join course c on t.t_id = c.t_id
                left join score  s on s.c_id = c.c_id
                left join student stu on s.s_id = stu.s_id;
```

Hive总是按照从左到右的顺序执行，Hive会对每对 JOIN 连接对象启动一个 MapReduce 任务。

上面的例子中会首先启动一个 MapReduce job 对表 t 和表 c 进行连接操作；然后再启动一个 MapReduce job 将第一个 MapReduce job 的输出和表 s 进行连接操作；然后再继续直到全部操作；

笛卡尔积

满足以下条件将会产生笛卡尔集：

- 没有连接条件
- 连接条件无效
- 所有表中的所有行互相连接

如果表A、B分别有M、N条数据，其笛卡尔积的结果将有 M*N 条数据；缺省条件下hive不支持笛卡尔积运算；

```
set hive.strict.checks.cartesian.product=false;

select * from u1, u2;
```

第 5 节 排序子句【重点】

全局排序(order by)

order by 子句出现在select语句的结尾；

order by子句对最终的结果进行排序；

默认使用升序(ASC)；可以使用DESC，跟在字段名之后表示降序；

ORDER BY执行全局排序，只有一个reduce；

```
-- 普通排序
select * from emp order by deptno;
```

```
-- 按别名排序
select empno, ename, job, mgr, sal + nvl(comm, 0) salcomm, deptno
  from emp
order by salcomm desc;


-- 多列排序
select empno, ename, job, mgr, sal + nvl(comm, 0) salcomm, deptno
  from emp
order by deptno, salcomm desc;

-- 排序字段要出现在select子句中。以下语句无法执行（因为select子句中缺少
deptno）：
select empno, ename, job, mgr, sal + nvl(comm, 0) salcomm
  from emp
order by deptno, salcomm desc;
```

每个MR内部排序(sort by)

对于大规模数据而言order by效率低；

在很多业务场景，我们并不需要全局有序的数据，此时可以使用sort by；

sort by为每个reduce产生一个排序文件，在reduce内部进行排序，得到局部有序的结果；

```
-- 设置reduce个数
set mapreduce.job.reduces=2;

-- 按照工资降序查看员工信息
select * from emp sort by sal desc;

-- 将查询结果导入到文件中（按照工资降序）。生成两个输出文件，每个文件内部数据按
工资降序排列
insert overwrite local directory '/home/hadoop/output/sortsal'
select * from emp sort by sal desc;
```

分区排序(distribute by)

distribute by 将特定的行发送到特定的reducer中，便于后继的聚合 与 排序操作；

distribute by 类似于MR中的分区操作，可以结合sort by操作，使分区数据有序；

distribute by 要写在sort by之前；

```
-- 启动2个reducer task；先按 deptno 分区，在分区内按 sal+comm 排序
set mapreduce.job.reduces=2;

-- 将结果输出到文件，观察输出结果
insert overwrite local directory '/home/hadoop/output/distBy'
select empno, ename, job, deptno, sal + nvl(comm, 0) salcomm
  from emp
distribute by deptno
sort by salcomm desc;
-- 上例中，数据被分到了统一区，看不出分区的结果

-- 将数据分到3个区中，每个分区都有数据
set mapreduce.job.reduces=3;
insert overwrite local directory '/home/hadoop/output/distBy1'
select empno, ename, job, deptno, sal + nvl(comm, 0) salcomm
  from emp
distribute by deptno
sort by salcomm desc;
```

Cluster By

当distribute by 与 sort by是同一个字段时，可使用cluster by简化语法；

cluster by 只能是剩下，不能指定排序规则；

```
-- 语法上是等价的
select * from emp distribute by deptno sort by deptno;
select * from emp cluster by deptno;
```

排序小结：

- order by。执行全局排序，效率低。生产环境中慎用
- sort by。使数据局部有序(在reduce内部有序)
- distribute by。按照指定的条件将数据分组，常与sort by联用，使数据局部有序
- cluster by。当distribute by 与 sort by是同一个字段时，可使用cluster by简化语法

第七部分 函数

Hive内置函数：https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-Built-inFunctions

第 1 节 系统内置函数

查看系统函数

```
-- 查看系统自带函数
show functions;

-- 显示自带函数的用法
desc function upper;
desc function extended upper;
```

日期函数【重要】

```
-- 当前前日期
select current_date;

select unix_timestamp();
-- 建议使用current_timestamp，有没有括号都可以
select current_timestamp();

-- 时间戳转日期
select from_unixtime(1505456567);
select from_unixtime(1505456567, 'yyyyMMdd');
select from_unixtime(1505456567, 'yyyy-MM-dd HH:mm:ss');

-- 日期转时间戳
select unix_timestamp('2019-09-15 14:23:00');

-- 计算时间差
select datediff('2020-04-18','2019-11-21');
select datediff('2019-11-21', '2020-04-18');

-- 查询当月第几天
select dayofmonth(current_date);
```

```
-- 计算月末:
select last_day(current_date);

-- 当月第1天:
select date_sub(current_date, dayofmonth(current_date)-1)

-- 下个月第1天:
select add_months(date_sub(current_date,
dayofmonth(current_date)-1), 1)

-- 字符串转时间（字符串必须为：yyyy-MM-dd格式）
select to_date('2020-01-01');
select to_date('2020-01-01 12:12:12');

-- 日期、时间戳、字符串类型格式化输出标准时间格式
select date_format(current_timestamp(), 'yyyy-MM-dd HH:mm:ss');
select date_format(current_date(), 'yyyyMMdd');
select date_format('2020-06-01', 'yyyy-MM-dd HH:mm:ss');

-- 计算emp表中，每个人的工龄
select *, round(datediff(current_date, hiredate)/365,1)
workingyears from emp;
```

字符串函数

```
-- 转小写。lower
select lower("HELLO WORLD");

-- 转大写。upper
select lower(ename), ename from emp;

-- 求字符串长度。length
select length(ename), ename from emp;

-- 字符串拼接。 concat / ||
select empno || " " ||ename idname from emp;
select concat(empno, " " ,ename) idname from emp;

-- 指定分隔符。concat_ws(separator, [string | array(string)]+)
SELECT concat_ws('.', 'www', array('baidu', 'com'));
select concat_ws(" ", ename, job) from emp;

-- 求子串。substr
```

```
SELECT substr('www.baidu.com', 5);
SELECT substr('www.baidu.com', -5);
SELECT substr('www.baidu.com', 5, 5);

-- 字符串切分。split，注意 '.' 要转义
select split("www.baidu.com", "\\.");
```

数学函数

```
-- 四舍五入。round
select round(314.15926);
select round(314.15926, 2);
select round(314.15926, -2);

-- 向上取整。ceil
select ceil(3.1415926);

-- 向下取整。floor
select floor(3.1415926);

-- 其他数学函数包括：绝对值、平方、开方、对数运算、三角运算等
```

条件函数【重要】

```
-- if (boolean testCondition, T valueTrue, T valueFalseOrNull)
select sal, if (sal<1500, 1, if (sal < 3000, 2, 3)) from emp;

-- CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END
-- 将emp表的员工工资等级分类：0-1500、1500-3000、3000以上
select sal, if (sal<=1500, 1, if (sal <= 3000, 2, 3)) from emp;

-- CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END
-- 复杂条件用 case when 更直观
select sal, case when sal<=1500 then 1
                 when sal<=3000 then 2
                 else 3 end sallevel
  from emp;


-- 以下语句等价
select ename, deptno,
```

```
       case deptno when 10 then 'accounting'
                   when 20 then 'research'
                   when 30 then 'sales'
                   else 'unknown' end deptname
  from emp;
select ename, deptno,
       case when deptno=10 then 'accounting'
            when deptno=20 then 'research'
            when deptno=30 then 'sales'
            else 'unknown' end deptname
  from emp;

-- COALESCE(T v1, T v2, ...)。返回参数中的第一个非空值；如果所有值都为
NULL，那么返回NULL
select sal, coalesce(comm, 0) from emp;

-- isnull(a) isnotnull(a)
select * from emp where isnull(comm);
select * from emp where isnotnull(comm);

-- nvl(T value, T default_value)
select empno, ename, job, mgr, hiredate, deptno, sal +
nvl(comm,0) sumsal
  from emp;

-- nullif(x, y) 相等为空，否则为a
SELECT nullif("b", "b"), nullif("b", "a");
```

UDTF函数【重要】

UDTF : User Defined Table-Generating Functions。用户定义表生成函数，一行输入，多行输出。

```
-- explode，炸裂函数
-- 就是将一行中复杂的 array 或者 map 结构拆分成多行
select explode(array('A','B','C')) as col;
select explode(map('a', 8, 'b', 88, 'c', 888));

-- UDTF's are not supported outside the SELECT clause, nor nested
in expressions
-- SELECT pageid, explode(adid_list) AS myCol... is not supported
-- SELECT explode(explode(adid_list)) AS myCol... is not
supported
```

```
-- lateral view 常与 表生成函数explode结合使用

-- lateral view 语法:
lateralView: LATERAL VIEW udtf(expression) tableAlias AS
columnAlias (',' columnAlias)*
fromClause: FROM baseTable (lateralView)*


-- lateral view 的基本使用
with t1 as (
  select 'OK' cola, split('www.baidu.com', '\\.') colb )
select cola, colc

  from t1
       lateral view explode(colb) t2 as colc;
```

UDTF 案例1:

```
-- 数据(uid tags):
1 1,2,3
2 2,3
3 1,2

--编写sql,实现如下结果:
1 1
1 2
1 3
2 2
2 3
3 1
3 2

-- 建表加载数据
create table market(
    id int,
    storage string,
    allocation string,
    outdt string
)
row format delimited fields terminated by '\t';
load data local inpath '/hivedata/market.txt' into table market;

-- SQL
```

```
select uid, tag
  from t1
       lateral view explode(split(tags,",")) t2 as tag;
```

UDTF 案例2:

```
-- 数据准备
lisi|Chinese:90,Math:80,English:70
wangwu|Chinese:88,Math:90,English:96
maliu|Chinese:99,Math:65,English:60

-- 创建表
create table studscore(
   name string
  ,score map<String,string>)
row format delimited
fields terminated by '|'
collection items terminated by ','
map keys terminated by ':';

-- 加载数据
load data local inpath '/home/hadoop/data/score.dat' overwrite
into table studscore;

-- 需求：找到每个学员的最好成绩
-- 第一步，使用 explode 函数将map结构拆分为多行
select explode(score) as (subject, socre) from studscore;
--但是这里缺少了学员姓名，加上学员姓名后出错。下面的语句有是错的
select name, explode(score) as (subject, socre) from studscore;

-- 第二步：explode常与 lateral view 函数联用，这两个函数结合在一起能关联其他字段
select name, subject, score1 as score from studscore
lateral view explode(score) t1 as subject, score1;

-- 第三步：找到每个学员的最好成绩
select name, max(mark) maxscore
  from (select name, subject, mark
          from studscore lateral view explode(score) t1 as
subject, mark) t1
group by name;

with tmp as (
```

```
select name, subject, mark
  from studscore lateral view explode(score) t1 as subject, mark
)
select name, max(mark) maxscore
  from tmp
group by name;
```

小结：

- 将一行数据转换成多行数据，可以用于array和map类型的数据；
- lateral view 与 explode 联用，解决 UDTF 不能添加额外列的问题

第 2 节 窗口函数【重要】

窗口函数又名开窗函数，属于分析函数的一种。用于解决复杂报表统计需求的功能强大的函数，很多场景都需要用到。窗口函数用于计算基于组的某种聚合值，它和聚合函数的不同之处是：对于每个组返回多行，而聚合函数对于每个组只返回一行。

窗口函数指定了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。

over 关键字

使用窗口函数之前一般要要通过over()进行开窗

```
-- 查询emp表工资总和
select sum(sal) from emp;

-- 不使用窗口函数，有语法错误
select ename, sal, sum(sal) salsum from emp;

-- 使用窗口函数，查询员工姓名、薪水、薪水总和
select ename, sal, sum(sal) over() salsum,
       concat(round(sal / sum(sal) over()*100, 1) || '%')
ratiosal
  from emp;
```

注意：窗口函数是针对每一行数据的；如果over中没有参数，默认的是全部结果集；

partition by子句

在over窗口中进行分区，对某一列进行分区统计，窗口的大小就是分区的大小

```
-- 查询员工姓名、薪水、部门薪水总和
select ename, sal, sum(sal) over(partition by deptno) salsum
  from emp;
```

order by 子句

order by 子句对输入的数据进行排序

```
-- 增加了order by子句；sum：从分组的第一行到当前行求和
select ename, sal, deptno, sum(sal) over(partition by deptno
order by sal) salsum
  from emp;
```

Window子句

```
rows between ... and ...
```

如果要对窗口的结果做更细粒度的划分，使用window子句，有如下的几个选项：!clear

- unbounded preceding。组内第一行数据
- n preceding。组内当前行的前n行数据
- current row。当前行数据
- n following。组内当前行的后n行数据
- unbounded following。组内最后一行数据

```
-- rows between ... and ... 子句
-- 等价。组内，第一行到当前行的和
select ename, sal, deptno,
       sum(sal) over(partition by deptno order by ename) from
emp;
select ename, sal, deptno,
       sum(sal) over(partition by deptno order by ename
                     rows between unbounded preceding and current
row
                    )
  from emp;

-- 组内，第一行到最后一行的和
select ename, sal, deptno,
       sum(sal) over(partition by deptno order by ename
                     rows between unbounded preceding and
unbounded following
                    )
  from emp;

-- 组内，前一行 + 当前行 +后一行
select ename, sal, deptno,
       sum(sal) over(partition by deptno order by ename
                     rows between 1 preceding and 1 following
                    )
  from emp;
```

排名函数

都是从1开始，生成数据项在分组中的排名。

- row_number()。排名顺序增加不会重复；如1、2、3、4、... ...
- RANK()。 排名相等会在名次中留下空位；如1、2、2、4、5、... ...
- DENSE_RANK()。 排名相等会在名次中不会留下空位； 如1、2、2、3、4、... ...

```
-- row_number / rank / dense_rank
100 1   1   1
100 2   1   1
100 3   1   1
99  4   4   2
98  5   5   3
```

```
98  6   5   3
97  7   7   4

-- 数据准备
class1  s01 100
class1  s03 100
class1  s05 100
class1  s07 99
class1  s09 98
class1  s02 98
class1  s04 97
class2  s21 100
class2  s24 99
class2  s27 99
class2  s22 98
class2  s25 98
class2  s28 97
class2  s26 96

-- 创建表加载数据
create table t2(
    cname string,
    sname string,
    score int
) row format delimited fields terminated by '\t';
load data local inpath '/home/hadoop/data/t2.dat' into table t2;

-- 按照班级，使用3种方式对成绩进行排名
select cname, sname, score,
       row_number() over (partition by cname order by score desc)
rank1,
       rank() over (partition by cname order by score desc)
rank2,
       dense_rank() over (partition by cname order by score desc)
rank3
  from t2;

-- 求每个班级前3名的学员--前3名的定义是什么--假设使用dense_rank
select cname, sname, score, rank
  from (select cname, sname, score,
               dense_rank() over (partition by cname order by
score desc) rank
        from t2) tmp
 where rank <= 3;
```

序列函数

- lag。返回当前数据行的上一行数据
- lead。返回当前数据行的下一行数据
- first_value。取分组内排序后，截止到当前行，第一个值
- last_value。分组内排序后，截止到当前行，最后一个值
- ntile。将分组的数据按照顺序切分成n片，返回当前切片值

```
-- 测试数据 userpv.dat。cid ctime pv
cookie1,2019-04-10,1
cookie1,2019-04-11,5
cookie1,2019-04-12,7
cookie1,2019-04-13,3
cookie1,2019-04-14,2
cookie1,2019-04-15,4
cookie1,2019-04-16,4
cookie2,2019-04-10,2
cookie2,2019-04-11,3
cookie2,2019-04-12,5
cookie2,2019-04-13,6
cookie2,2019-04-14,3
cookie2,2019-04-15,9
cookie2,2019-04-16,7

-- 建表语句
create table userpv(
    cid string,
    ctime date,
    pv int
)
row format delimited fields terminated by ",";

-- 加载数据
Load data local inpath '/home/hadoop/data/userpv.dat' into table
userpv;

-- lag。返回当前数据行的上一行数据
-- lead。功能上与lag类似
select cid, ctime, pv,
       lag(pv) over(partition by cid order by ctime) lagpv,
       lead(pv) over(partition by cid order by ctime) leadpv
  from userpv;
```

```
-- first_value / last_value
select cid, ctime, pv,
       first_value(pv) over (partition by cid order by ctime rows
between unbounded preceding and unbounded following) as firstpv,
       last_value(pv) over (partition by cid order by ctime rows
between unbounded preceding and unbounded following) as lastpv
  from userpv;

-- ntile。按照cid进行分组，每组数据分成2份
select cid, ctime, pv,
       ntile(2) over(partition by cid order by ctime) ntile
from userpv;
```

SQL面试题

1、连续7天登录的用户

```
-- 数据。uid dt status(1 正常登录，0 异常)
1 2019-07-11 1
1 2019-07-12 1
1 2019-07-13 1
1 2019-07-14 1
1 2019-07-15 1
1 2019-07-16 1
1 2019-07-17 1
1 2019-07-18 1
2 2019-07-11 1
2 2019-07-12 1
2 2019-07-13 0
2 2019-07-14 1
2 2019-07-15 1
2 2019-07-16 0
2 2019-07-17 1
2 2019-07-18 0
3 2019-07-11 1
3 2019-07-12 1
3 2019-07-13 1
3 2019-07-14 0
3 2019-07-15 1
3 2019-07-16 1
3 2019-07-17 1
3 2019-07-18 1
```

```
-- 建表语句
create table ulogin(
    uid int,
    dt date,
    status int
)
row format delimited fields terminated by ' ';

-- 加载数据
load data local inpath '/home/hadoop/data/ulogin.dat' into table
ulogin;

-- 连续值的求解，面试中常见的问题。这也是同一类，基本都可按照以下思路进行
-- 1、使用 row_number 在组内给数据编号(rownum)
-- 2、某个值 - rownum = gid，得到结果可以作为后面分组计算的依据
-- 3、根据求得的gid，作为分组条件，求最终结果
select uid, dt,
       date_sub(dt, row_number() over (partition by uid order by
dt)) gid
  from ulogin
 where status=1;

select uid, count(*) countlogin
  from (select uid, dt,
               date_sub(dt, row_number() over (partition by uid
order by dt)) gid
          from ulogin
         where status=1) t1
group by uid, gid
having countlogin >= 7;
```

2、编写sql语句实现每班前三名，分数一样并列，同时求出前三名按名次排序的分差

```
-- 数据。sid class score
1 1901 90
2 1901 90
3 1901 83
4 1901 60
5 1902 66
6 1902 23
7 1902 99
8 1902 67
9 1902 87
```

```
-- 待求结果数据如下：
class   score   rank    lagscore
1901    90      1       0
1901    90      1       0
1901    83      2       -7
1901    60      3       -23
1902    99      1       0
1902    87      2       -12
1902    67      3       -20

-- 建表语句
create table stu(
    sno int,
    class string,
    score int
)row format delimited fields terminated by ' ';

-- 加载数据
load data local inpath '/home/hadoop/data/stu.dat' into table
stu;

-- 求解思路：
-- 1、上排名函数，分数一样并列，所以用dense_rank
-- 2、将上一行数据下移，相减即得到分数差
-- 3、处理 NULL
with tmp as (
select sno, class, score,
       dense_rank() over (partition by class order by score desc)
as rank
  from stu
)
select class, score, rank,
       nvl(score - lag(score) over (partition by class order by
score desc), 0) lagscore
  from tmp
 where rank<=3;
```

3、行 <=> 列

```
-- 数据：id course
1 java
1 hadoop
1 hive
```

```
1 hbase
2 java
2 hive
2 spark
2 flink
3 java
3 hadoop
3 hive
3 kafka

-- 建表加载数据
create table rowline1(
    id string,
    course string
)row format delimited fields terminated by ' ';
load data local inpath '/root/data/data1.dat' into table
rowline1;

-- 编写sql，得到结果如下(1表示选修，0表示未选修)
id      java    hadoop  hive    hbase   spark   flink   kafka
1       1       1       1       1       0       0       0
2       1       0       1       0       1       1       0
3       1       1       1       0       0       0       1

-- 使用case when；group by + sum
select id,
sum(case when course="java"  then 1 else 0 end) as java,
sum(case when course="hadoop" then 1 else 0 end) as hadoop,
sum(case when course="hive"  then 1 else 0 end) as hive,
sum(case when course="hbase" then 1 else 0 end) as hbase,
sum(case when course="spark" then 1 else 0 end) as spark,
sum(case when course="flink" then 1 else 0 end) as flink,
sum(case when course="kafka" then 1 else 0 end) as kafka
  from rowline1
group by id;
```

```
-- 数据。id1 id2 flag
a b 2
a b 1
a b 3
c d 6
c d 8
c d 8
```

```
-- 编写sql实现如下结果
id1 id2  flag
a   b   2|1|3
c   d   6|8

-- 创建表 & 加载数据
create table rowline2(
    id1 string,
    id2 string,
    flag int
) row format delimited fields terminated by ' ';
load data local inpath '/root/data/data2.dat' into table
rowline2;
```

```
-- 第一步 将元素聚拢
select id1, id2, collect_set(flag) flag from rowline2 group by
id1, id2;
select id1, id2, collect_list(flag) flag from rowline2 group by
id1, id2;
select id1, id2, sort_array(collect_set(flag)) flag from rowline2
group by id1, id2;

-- 第二步 将元素连接在一起
select id1, id2, concat_ws("|", collect_set(flag)) flag
  from rowline2
group by id1, id2;

-- 这里报错，CONCAT_WS must be "string or array<string>"。加一个类型
转换即可
select id1, id2, concat_ws("|", collect_set(cast (flag as
string))) flag
  from rowline2
group by id1, id2;

-- 创建表 rowline3
create table rowline3 as
select id1, id2, concat_ws("|", collect_set(cast (flag as
string))) flag
  from rowline2
group by id1, id2;

-- 第一步：将复杂的数据展开
```

```
select explode(split(flag, "\\|")) flat from rowline3;

-- 第二步：lateral view 后与其他字段关联
select id1, id2, newflag
  from rowline3 lateral view explode(split(flag, "\\|")) t1 as
newflag;

lateralView: LATERAL VIEW udtf(expression) tableAlias AS
columnAlias (',' columnAlias)*
fromClause: FROM baseTable (lateralView)*
```

```
小结：
case when + sum + group by
collect_set、collect_list、concat_ws
sort_array
explode + lateral view
```

第 3 节 自定义函数

当 Hive 提供的内置函数无法满足实际的业务处理需要时，可以考虑使用用户自定义函数进行扩展。用户自定义函数分为以下三类：

- UDF（User Defined Function）。用户自定义函数，一进一出
- UDAF（User Defined Aggregation Function）。用户自定义聚集函数，多进一出；类似于：count/max/min
- UDTF（User Defined Table-Generating Functions）。用户自定义表生成函数，一进多出；类似于：explode

UDF开发：

- 继承org.apache.hadoop.hive.ql.exec.UDF
- 需要实现evaluate函数；evaluate函数支持重载
- UDF必须要有返回类型，可以返回null，但是返回类型不能为void

UDF开发步骤

- 创建maven java 工程，添加依赖
- 开发java类继承UDF，实现evaluate 方法
- 将项目打包上传服务器
- 添加开发的jar包

- 设置函数与自定义函数关联
- 使用自定义函数

需求：扩展系统 nvl 函数功能：

```
nvl(ename, "OK"): ename==null => 返回第二个参数
nvl(ename, "OK"): ename==null or ename=="" or ename="      " =>
返回第二个参数
```

1、创建maven java 工程，添加依赖

```
<!-- pom.xml 文件 -->
<dependencies>
    <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>2.3.7</version>
    </dependency>
</dependencies>
```

2、开发java类继承UDF，实现evaluate 方法

```
package cn.lago.hive.udf;
import org.apache.hadoop.hive.ql.exec.UDF;

public class nvl extends UDF {
    public Text evaluate(final Text t, final Text x) {
        if (t == null || t.toString().trim().length()==0) {
            return x;
        }

        return t;
    }
}
```

3、将项目打包上传服务器

4、添加开发的jar包（在Hive命令行中）

```
add jar /home/hadoop/hiveudf.jar;
```

5、创建临时函数。指定类名一定要完整的路径，即包名加类名

```
create temporary function mynvl as "cn.lago.hive.udf.nvl";
```

6、执行查询

```
-- 基本功能还有
select mynvl(comm, 0) from mydb.emp;

-- 测试扩充的功能
select mynvl("", "OK");
select mynvl("  ", "OK");
```

7、退出Hive命令行，再进入Hive命令行。执行步骤6的测试，发现函数失效。

备注：创建临时函数每次进入Hive命令行时，都必须执行以下语句，很不方便：

```
add jar /home/hadoop/hiveudf.jar;
create temporary function mynvl as "cn.lago.hive.udf.nvl";
```

可创建永久函数：

1、将jar上传HDFS

```
hdfs dfs -put hiveudf.jar jar/
```

2、在Hive命令行中创建永久函数

```
create function mynvl1 as 'cn.lago.hive.udf.nvl' using jar
'hdfs:/user/hadoop/jar/hiveudf.jar';

-- 查询所有的函数，发现 mynvl1 在列表中
show functions;
```

3、退出Hive，再进入，执行测试

```
-- 基本功能还有
select mynvl(comm, 0) from mydb.emp;

-- 测试扩充的功能
select mynvl("", "OK");
select mynvl("  ", "OK");
```

4、删除永久函数，并检查

```
drop function mynvl1;
show functions;
```

第八部分 HQL操作之--DML命令

数据操纵语言DML(Data Manipulation Language)，DML主要有三种形式：插入(INSERT)、删除(DELETE)、更新(UPDATE)。

事务(transaction)是一组单元化操作，这些操作要么都执行，要么都不执行，是一个不可分割的工作单元。

事务具有的四个要素：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability），这四个基本要素通常称为ACID特性。

- **原子性**。一个事务是一个不可再分割的工作单位，事务中的所有操作要么都发生，要么都不发生。
- **一致性**。事务的一致性是指事务的执行不能破坏数据库数据的完整性和一致性，一个事务在执行之前和执行之后，数据库都必须处于一致性状态。
- **隔离性**。在并发环境中，并发的事务是相互隔离的，一个事务的执行不能被其他事务干扰。即不同的事务并发操纵相同的数据时，每个事务都有各自完整的数据空间，即一个事务内部的操作及使用的数据对其他并发事务是隔离的，并发执行的各个事务之间不能互相干扰。
- **持久性**。事务一旦提交，它对数据库中数据的改变就应该是永久性的。

第 1 节 Hive 事务

Hive从0.14版本开始支持事务 和 **行级更新**，但缺省是不支持的，需要一些附加的配置。要想支持行级insert、update、delete，需要配置Hive支持事务。

Hive事务的限制：

- Hive提供行级别的ACID语义
- BEGIN、COMMIT、ROLLBACK 暂时不支持，所有操作自动提交
- 目前只支持 ORC 的文件格式
- 默认事务是关闭的，需要设置开启
- 要是使用事务特性，表必须是分桶的
- 只能使用内部表
- 如果一个表用于ACID写入（INSERT、UPDATE、DELETE），必须在表中设置表属性："transactional=true"
- 必须使用事务管理器 org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
- 目前支持快照级别的隔离。就是当一次数据查询时，会提供一个数据一致性的快照
- LOAD DATA语句目前在事务表中暂时不支持

HDFS是不支持文件的修改；并且当有数据追加到文件，HDFS不对读数据的用户提供一致性的。为了在HDFS上支持数据的更新：

- 表和分区的数据都被存在基本文件中（base files）
- 新的记录和更新，删除都存在增量文件中（delta files）
- 一个事务操作创建一系列的增量文件
- 在读取的时候，将基础文件和修改，删除合并，最后返回给查询

第 2 节 Hive 事务操作示例

```
-- 这些参数也可以设置在hive-site.xml中
SET hive.support.concurrency = true;
-- Hive 0.x and 1.x only
SET hive.enforce.bucketing = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET hive.txn.manager =
org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;

-- 创建表用于更新。满足条件：内部表、ORC格式、分桶、设置表属性
create table zxz_data(
    name string,
    nid int,
    phone string,
    ntime date)
clustered by(nid) into 5 buckets
stored as orc
```

```
tblproperties('transactional'='true');

-- 创建临时表，用于向分桶表插入数据
create table temp1(
    name string,
    nid int,
    phone string,
    ntime date)
row format delimited
fields terminated by ",";

-- 数据
name1,1,010-83596208,2020-01-01
name2,2,027-63277201,2020-01-02
name3,3,010-83596208,2020-01-03
name4,4,010-83596208,2020-01-04
name5,5,010-83596208,2020-01-05

-- 向临时表加载数据；向事务表中加载数据
load data local inpath '/home/hadoop/data/zxz_data.txt' overwrite
into table temp1;
insert into table zxz_data select * from temp1;

-- 检查数据和文件
select * from zxz_data;
dfs -ls /user/hive/warehouse/mydb.db/zxz_data ;

-- DML 操作
delete from zxz_data where nid = 3;
dfs -ls /user/hive/warehouse/mydb.db/zxz_data ;

insert into zxz_data values ("name3", 3, "010-83596208",
current_date);   -- 不支持
insert into zxz_data values ("name3", 3, "010-83596208", "2020-
06-01");   -- 执行
insert into zxz_data select "name3", 3, "010-83596208",
current_date;
dfs -ls /user/hive/warehouse/mydb.db/zxz_data ;

insert into zxz_data values
("name6", 6, "010-83596208", "2020-06-02"),
("name7", 7, "010-83596208", "2020-06-03"),
("name8", 9, "010-83596208", "2020-06-05"),
("name9", 8, "010-83596208", "2020-06-06");
dfs -ls /user/hive/warehouse/mydb.db/zxz_data ;
```

```
update zxz_data set name=concat(name, "00") where nid>3;
dfs -ls /user/hive/warehouse/mydb.db/zxz_data ;

-- 分桶字段不能修改，下面的语句不能执行
-- Updating values of bucketing columns is not supported
update zxz_data set nid = nid + 1;
```

第九部分 元数据管理与存储

第 1 节 Metastore

在Hive的具体使用中，首先面临的问题便是如何定义表结构信息，跟结构化的数据映射成功。所谓的映射指的是一种对应关系。在Hive中需要描述清楚表跟文件之间的映射关系、列和字段之间的关系等等信息。这些描述映射关系的数据的称之为Hive的元数据。该数据十分重要，因为只有通过查询它才可以确定用户编写sql和最终操作文件之间的关系。

Metadata即元数据。元数据包含用Hive创建的database、table、表的字段等元信息。元数据存储在关系型数据库中。如hive内置的Derby、第三方如MySQL等。

Metastore即元数据服务，是Hive用来管理库表元数据的一个服务。有了它上层的服务不用再跟裸的文件数据打交道，而是可以基于结构化的库表信息构建计算框架。

通过metastore服务将Hive的元数据暴露出去，而不是需要通过对Hive元数据库mysql的访问才能拿到Hive的元数据信息；metastore服务实际上就是一种thrift服务，通过它用户可以获取到Hive元数据，并且通过thrift获取元数据的方式，屏蔽了数据库访问需要驱动，url，用户名，密码等细节。

metastore三种配置方式

1、内嵌模式

内嵌模式使用的是内嵌的Derby数据库来存储元数据，也不需要额外起Metastore服务。数据库和Metastore服务都嵌入在主Hive Server进程中。这个是默认的，配置简单，但是一次只能一个客户端连接，适用于用来实验，不适用于生产环境。

优点：配置简单，解压hive安装包 bin/hive 启动即可使用；

缺点：不同路径启动hive，每一个hive拥有一套自己的元数据，无法共享。

2、本地模式

本地模式采用外部数据库来存储元数据，目前支持的数据库有：MySQL、Postgres、Oracle、MS SQL Server。教学中实际采用的是MySQL。

本地模式不需要单独起metastore服务，用的是跟Hive在同一个进程里的metastore服务。也就是说当启动一个hive 服务时，其内部会启动一个metastore服务。Hive根据 hive.metastore.uris 参数值来判断，如果为空，则为本地模式。

缺点：每启动一次hive服务，都内置启动了一个metastore；在hive-site.xml中暴露的数据库的连接信息；

优点：配置较简单，本地模式下hive的配置中指定mysql的相关信息即可。

3. 远程模式

远程模式下，需要单独起metastore服务，然后每个客户端都在配置文件里配置连接到该metastore服务。远程模式的metastore服务和hive运行在不同的进程里。**在生产环境中，建议用远程模式来配置Hive Metastore。**

在这种模式下，其他依赖hive的软件都可以通过Metastore访问Hive。此时需要配置hive.metastore.uris 参数来指定 metastore 服务运行的机器ip和端口，并且需要单独手动启动metastore服务。metastore服务可以配置多个节点上，避免单节点故障导致整个集群的hive client不可用。同时hive client配置多个metastore地址，会自动选择可用节点。

metastore内嵌模式配置

```
1、下载软件解压缩
2、设置环境变量，并使之生效
3、初始化数据库。
schematool -dbType derby -initSchema
4、进入hive命令行
5、再打开一个hive命令行，发现无法进入
```

metastore远程模式配置

配置规划：

节点	metastore	client
linux121	√	
linux122		√
linux123	√	

配置步骤：

配置步骤：**

1、将 linux123 的 hive 安装文件拷贝到 linux121、linux122

2、在linux121、linux123上分别启动 metastore 服务

```
# 启动 metastore 服务
nohup hive --service metastore &

# 查询9083端口(metastore服务占用的端口)
lsof -i:9083

# 安装lsof
yum install lsof
```

3、修改 linux122 上hive-site.xml。删除配置文件中：MySQL的配置、连接数据库的用户名、口令等信息；增加连接metastore的配置：

```
    <!-- hive metastore 服务地址 -->
    <property>
        <name>hive.metastore.uris</name>

<value>thrift://linux121:9083,thrift://linux123:9083</value>
    </property>
```

4、启动hive。此时client端无需实例化hive的metastore，启动速度会加快。

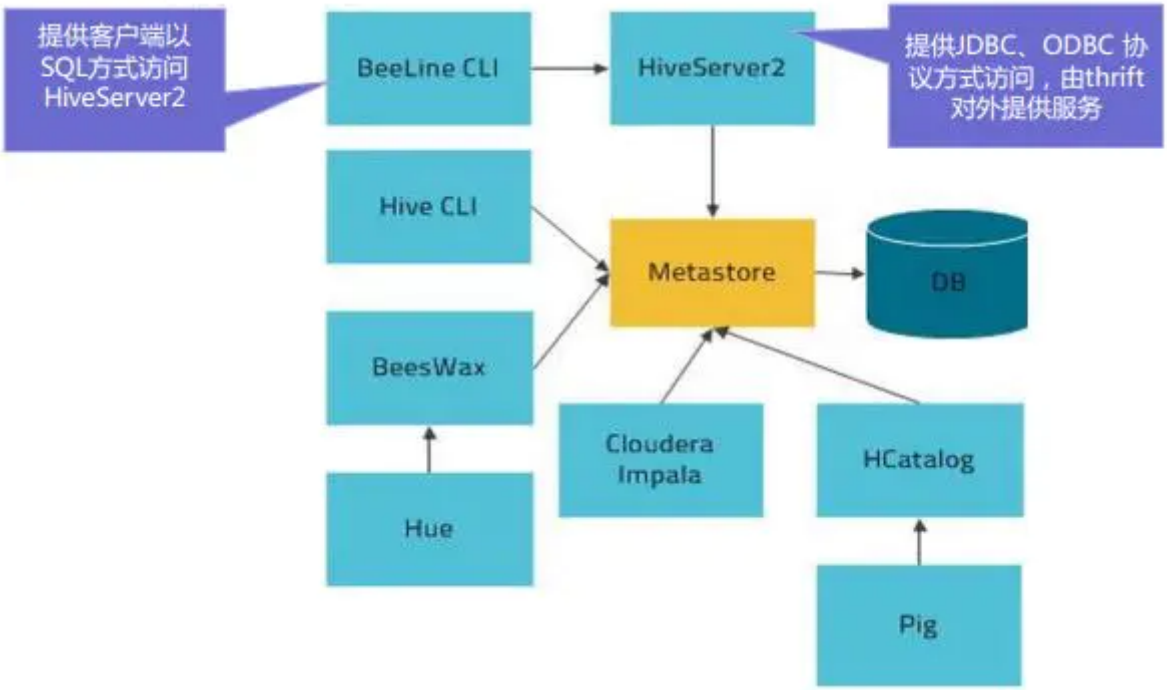
```
# 分别在linux121、linux123上执行以下命令，查看连接情况
lsof -i:9083
```

5、高可用测试。关闭已连接的metastore服务，发现hive连到另一个节点的服务上，仍然能够正常使用。

第 2 节 HiveServer2

HiveServer2是一个服务端接口，使远程客户端可以执行对Hive的查询并返回结果。目前基于Thrift RPC的实现是HiveServer的改进版本，并支持多客户端并发和身份验证，启动hiveServer2服务后，就可以使用jdbc、odbc、thrift 的方式连接。

Thrift是一种接口描述语言和二进制通讯协议，它被用来定义和创建跨语言的服务。它被当作一个远程过程调用（RPC）框架来使用，是由Facebook为“大规模跨语言服务开发”而开发的。



HiveServer2（HS2）是一种允许客户端对Hive执行查询的服务。HiveServer2是HiveServer1的后续 版本。HS2支持多客户端并发和身份验证，旨在为JDBC、ODBC等开放API客户端提供更好的支持。

HS2包括基于Thrift的Hive服务（TCP或HTTP）和用于Web UI 的Jetty Web服务器。

HiveServer2作用：

- 为Hive提供了一种允许客户端远程访问的服务
- 基于thrift协议，支持跨平台，跨编程语言对Hive访问
- 允许远程访问Hive

HiveServer2配置

配置规划：

节点	HiveServer2	client
linux121		
linux122		√
linux123	√	

配置步骤：

1、修改集群上的 core-site.xml，增加以下内容：

```
<!-- HiveServer2 连不上10000；hadoop为安装用户 -->
<!-- root用户可以代理所有主机上的所有用户 -->
<property>
        <name>hadoop.proxyuser.root.hosts</name>
        <value>*</value>
</property>
<property>
        <name>hadoop.proxyuser.root.groups</name>
        <value>*</value>
</property>
<property>
        <name>hadoop.proxyuser.hadoop.hosts</name>
        <value>*</value>
</property>
<property>
        <name>hadoop.proxyuser.hadoop.groups</name>
        <value>*</value>
</property>
```

2、修改 集群上的 hdfs-site.xml，增加以下内容：

```
<!-- HiveServer2 连不上10000；启用 webhdfs 服务 -->
<property>
        <name>dfs.webhdfs.enabled</name>
        <value>true</value>
</property>
```

3、启动linux123上的 HiveServer2 服务

```
# 启动 hiveserver2 服务
nohup hiveserver2 &

# 检查 hiveserver2 端口
lsof -i:10000

# 从2.0开始，HiveServer2提供了WebUI
# 还可以使用浏览器检查hiveserver2的启动情况。http://linux123:10002/
```

4、启动 linux122 节点上的 beeline

Beeline是从 Hive 0.11版本引入的，是 Hive 新的命令行客户端工具。

Hive客户端工具后续将使用Beeline 替代 Hive 命令行工具，并且后续版本也会废弃掉 Hive 客户端工具。

```
!connect jdbc:hive2://linux123:10000
use mydb;
show tables;
select * from emp;
create table tabtest1 (c1 int, c2 string);

!connect jdbc:mysql://linux123:3306
!help
!quit
```

第 3 节 HCatalog

HCatalog 提供了一个统一的元数据服务，允许不同的工具如 Pig、MapReduce 等通过 HCatalog 直接访问存储在 HDFS 上的底层文件。HCatalog是用来访问Metastore的Hive子项目，它的存在给了整个Hadoop生态环境一个统一的定义。

HCatalog 使用了 Hive 的元数据存储，这样就使得像 MapReduce 这样的第三方应用可以直接从 Hive 的数据仓库中读写数据。同时，HCatalog 还支持用户在 MapReduce 程序中只读取需要的表分区和字段，而不需要读取整个表，即提供一种逻辑上的视图来读取数据，而不仅仅是从物理文件的维度。

HCatalog 提供了一个称为 hcat 的命令行工具。这个工具和 Hive 的命令行工具类似，两者最大的不同就是 hcat 只接受不会产生 MapReduce 任务的命令。

```
# 进入 hcat 所在目录。$HIVE_HOME/hcatalog/bin
cd $HIVE_HOME/hcatalog/bin
```

```
# 执行命令，创建表
./hcat -e "create table default.test1(id string, name string, age int)"

# 长命令可写入文件，使用 -f 选项执行
./hcat -f createtable.txt

# 查看元数据
./hcat -e "use mydb; show tables"

# 查看表结构
./hcat -e "desc mydb.emp"

# 删除表
./hcat -e "drop table default.test1"
```

第 4 节 数据存储格式

Hive支持的存储数的格式主要有：TEXTFILE（默认格式） 、SEQUENCEFILE、RCFILE、ORCFILE、PARQUET。

- textfile为默认格式，建表时没有指定文件格式，则使用TEXTFILE，导入数据时会直接把数据文件拷贝到hdfs上不进行处理；
- sequencefile，rcfile，orcfile格式的表不能直接从本地文件导入数据，数据要先导入到textfile格式的表中， 然后再从表中用insert导入sequencefile、rcfile、orcfile表中。

行存储与列存储

行式存储下一张表的数据都是放在一起的，但列式存储下数据被分开保存了。

行式存储：

优点：数据被保存在一起，insert和update更加容易

缺点：选择（selection）时即使只涉及某几列，所有数据也都会被读取

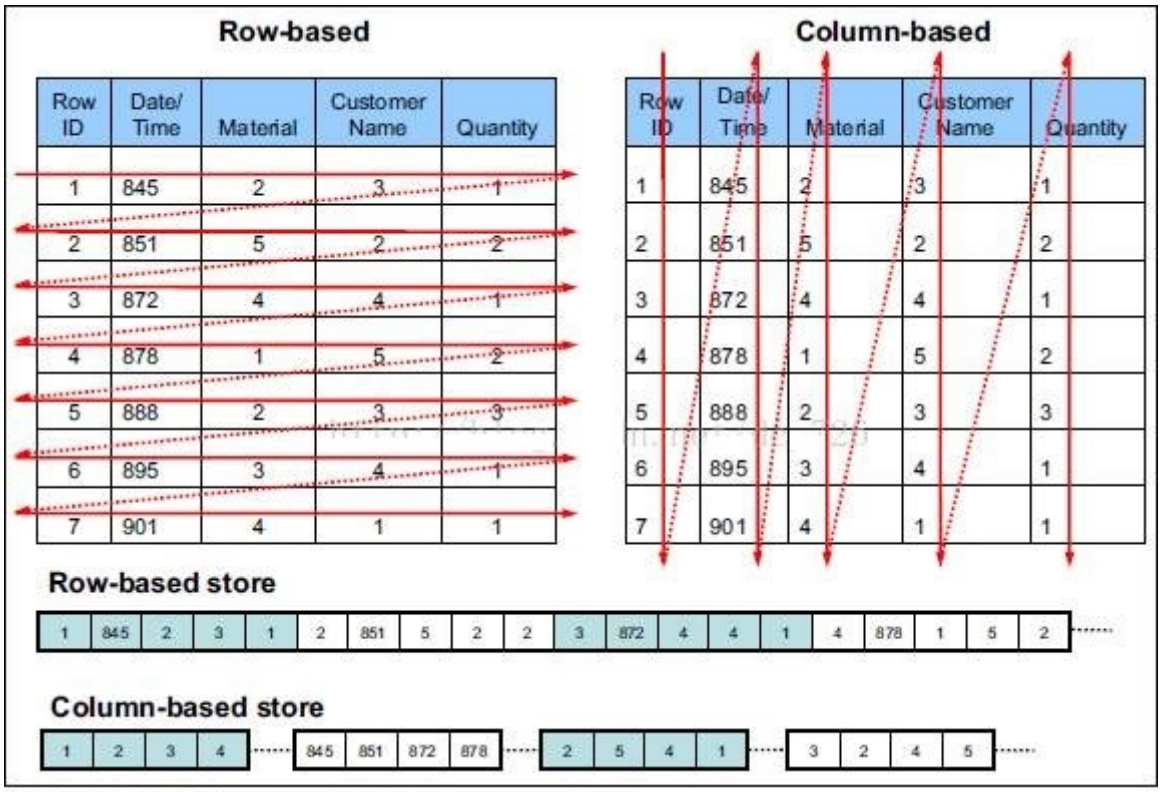
列式存储：

优点：查询时只有涉及到的列会被读取，效率高

缺点：选中的列要重新组装，insert/update比较麻烦

TEXTFILE、SEQUENCEFILE 的存储格式是基于行存储的；

ORC和PARQUET 是基于列式存储的。



TextFile

Hive默认的数据存储格式，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。

```
create table if not exists uaction_text(
  userid string,
  itemid string,
  behaviortype int,
  geohash string,
  itemcategory string,
  time string)
row format delimited fields terminated by ','
stored as textfile;

load data local inpath '/home/hadoop/data/useraction.dat'
overwrite into table uaction_text;
```

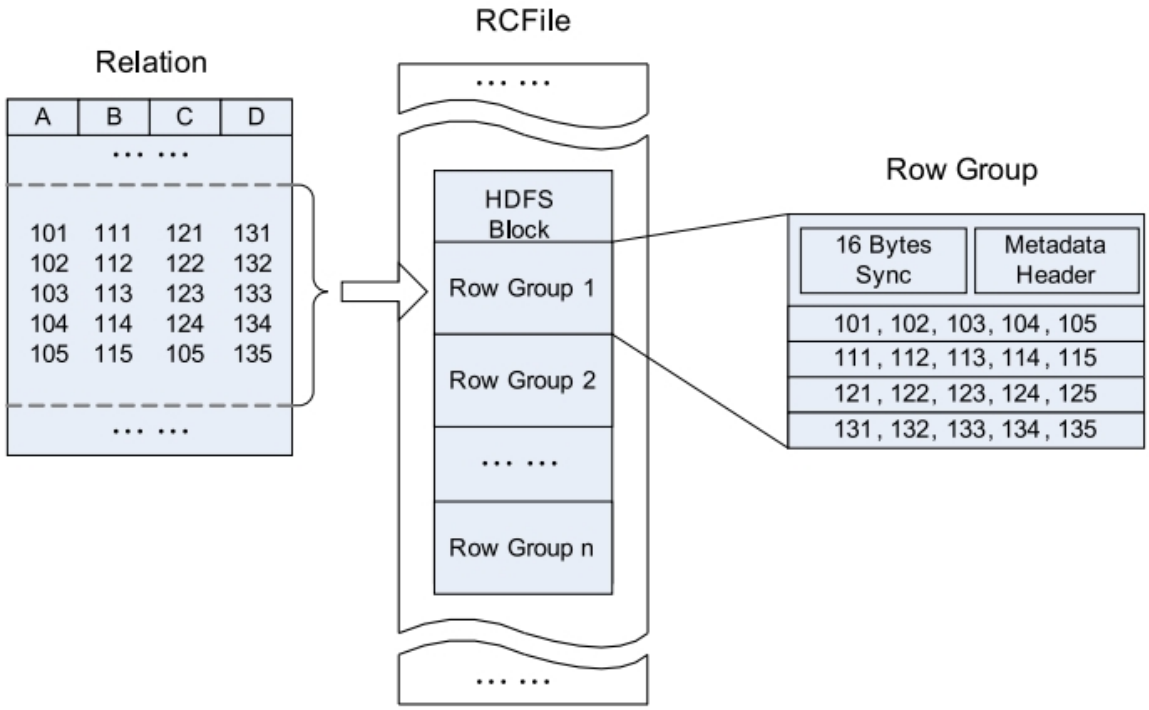
SEQUENCEFILE

SequenceFile是Hadoop API提供的一种二进制文件格式，其具有使用方便、可分割、可压缩的特点。 SequenceFile支持三种压缩选择：**none，record，block。** Record压缩率低，一般建议使用BLOCK压缩。

RCFile

RCFile全称Record Columnar File，列式记录文件，是一种类似于SequenceFile的键值对数据文件。RCFile结合列存储和行存储的优缺点，是基于行列混合存储的RCFile。

RCFile遵循的“**先水平划分，再垂直划分**”的设计理念。先将数据按行水平划分为行组，这样一行的数据就可以保证存储在同一个集群节点；然后在对行进行垂直划分。

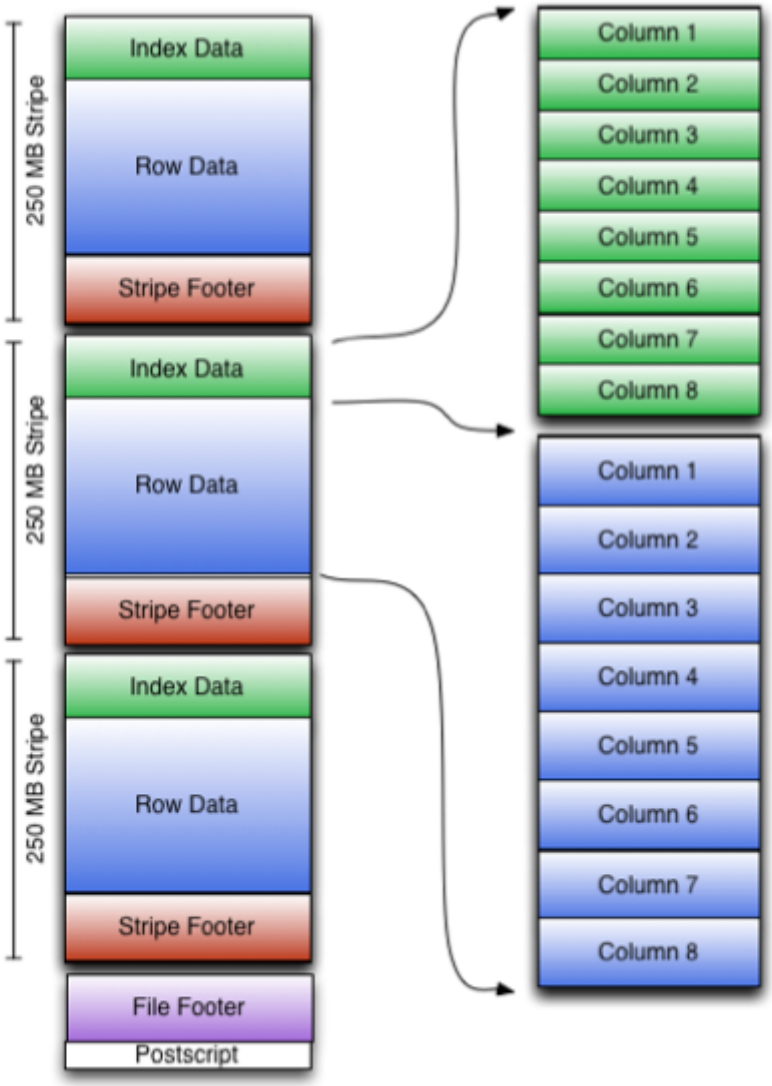


- 一张表可以包含多个HDFS block
- 在每个block中，RCFile以行组为单位存储其中的数据
- row group又由三个部分组成
    - 用于在block中分隔两个row group的16字节的标志区
    - 存储row group元数据信息的header
    - 实际数据区，表中的实际数据以列为单位进行存储

ORCFile

ORC File，它的全名是Optimized Row Columnar (ORC) file，其实就是对RCFile做了一些优化，在hive 0.11中引入的存储格式。这种文件格式可以提供一种高效的方法来存储Hive数据。它的设计目标是来克服Hive其他格式的缺陷。运用ORC File可以提高Hive的读、写以及处理数据的性能。ORC文件结构由三部分组成：

- 文件脚注(file footer)：包含了文件中 stripe 的列表，每个stripe行数，以及每个列的数据类型。还包括每个列的最大、最小值、行计数、求和等信息
- postscript：压缩参数和压缩大小相关信息
- 条带(stripe)：ORC文件存储数据的地方。在默认情况下，一个stripe的大小为250MB
    - Index Data：一个轻量级的index，默认是每隔1W行做一个索引。包括该条带的一些统计信息，以及数据在stripe中的位置索引信息
    - Rows Data：存放实际的数据。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
    - Stripe Footer：存放stripe的元数据信息



ORC在每个文件中提供了3个级别的索引：文件级、条带级、行组级。借助ORC提供的索引信息能加快数据查找和读取效率，规避大部分不满足条件的查询条件的文件和数据块。使用ORC可以避免磁盘和网络IO的浪费，提升程序效率，提升整个集群的工作负载。

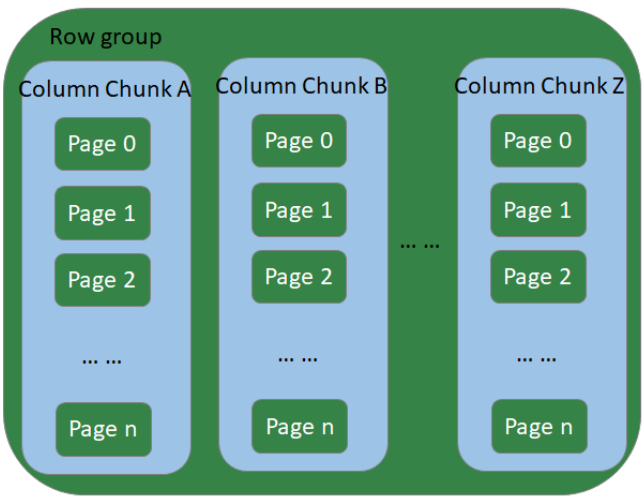
```
create table if not exists uaction_orc(
  userid string,
  itemid string,
  behaviortype int,
  geohash string,
  itemcategory string,
  time string)
stored as orc;

insert overwrite table uaction_orc select * from uaction_text;
```

Parquet

Apache Parquet是Hadoop生态圈中一种新型列式存储格式，它可以兼容Hadoop生态圈中大多数计算框架(Mapreduce、Spark等)，被多种查询引擎支持（Hive、Impala、Drill等），**与语言和平台无关的。**

Parquet文件是以二进制方式存储的，不能直接读取的，文件中包括实际数据和元数据，Parquet格式文件是自解析的。



一个文件由多个行组和元数据(Footer)构成。
每个行组：
- 写入数据是的最大缓存单元
- MR任务处理的最小并发单元
- 一般大小在50M-1GB之间

Column Chunk:
存储当前行组内的某一列数据
最小的IO并发单元

Page：
压缩、读取数据的最小单元
获取单条数据的最小读取单元
大小在8K-1M之间，越大压缩率越高

Footer:
数据的schema信息
每个行组的元数据信息：偏移量、大小
每个column chunk的元数据信息：每个列的编码格式、偏移量、大小、数据量等信息

Row group:

- 写入数据时的最大缓存单元
- MR任务的最小并发单元
- 一般大小在50MB-1GB之间

Column chunk：

- 存储当前Row group内的某一列数据
- 最小的IO并发单元

Page：

- 压缩、读数据的最小单元
- 获得单条数据时最小的读取数据单元
- 大小一般在8KB-1MB之间，越大压缩效率越高

Footer：

- 数据Schema信息
- 每个Row group的元信息：偏移量、大小
- 每个Column chunk的元信息：每个列的编码格式、首页偏移量、首索引页偏移量、个数、大小等信息

```
create table if not exists uaction_parquet(
  userid string,
  itemid string,
  behaviortype int,
  geohash string,
  itemcategory string,
  time string)
stored as parquet;

insert overwrite table uaction_parquet select * from
uaction_text;
```

文件存储格式对比测试

说明：

1、给 linux123 分配合适的资源。2core；2048G内存

2、适当减小文件的数据量（现有数据约800W，根据自己的实际选择处理100-300W条数据均可）

```
# 检查文件行数
wc -l uaction.dat

#
head -n 1000000 uaction.dat > uaction1.dat
tail -n 1000000 uaction.dat > uaction2.dat
```

文件压缩比

```
hive (mydb)> dfs -ls /user/hive/warehouse/mydb.db/ua*;
 13517070  /user/hive/warehouse/mydb.db/uaction_orc/000000_1000
 34867539
/user/hive/warehouse/mydb.db/uaction_parquet/000000_1000
 90019734
/user/hive/warehouse/mydb.db/uaction_text/useraction.dat
```

ORC > Parquet > text

执行查询

```
SELECT COUNT(*) FROM uaction_text;
SELECT COUNT(*) FROM uaction_orc;
SELECT COUNT(*) FROM uaction_parquet;

-- text : 14.446
-- orc: 0.15
-- parquet : 0.146
```

orc 与 parquet类似 > txt

在生产环境中，Hive表的数据格式使用最多的有三种：TextFile、ORCFile、Parquet。

- TextFile文件更多的是作为跳板来使用(即方便将数据转为其他格式)
- 有update、delete和事务性操作的需求，通常选择ORCFile
- 没有事务性要求，希望支持Impala、Spark，建议选择Parquet

第十部分 Hive调优策略

Hive作为大数据领域常用的数据仓库组件，在设计和开发阶段需要注意效率。

影响Hive效率的不仅仅是数据量过大；数据倾斜、数据冗余、job或I/O过多、MapReduce分配不合理等因素都对Hive的效率有影响。

对Hive的调优既包含对HiveQL语句本身的优化，也包含Hive配置项和MR方面的调整。

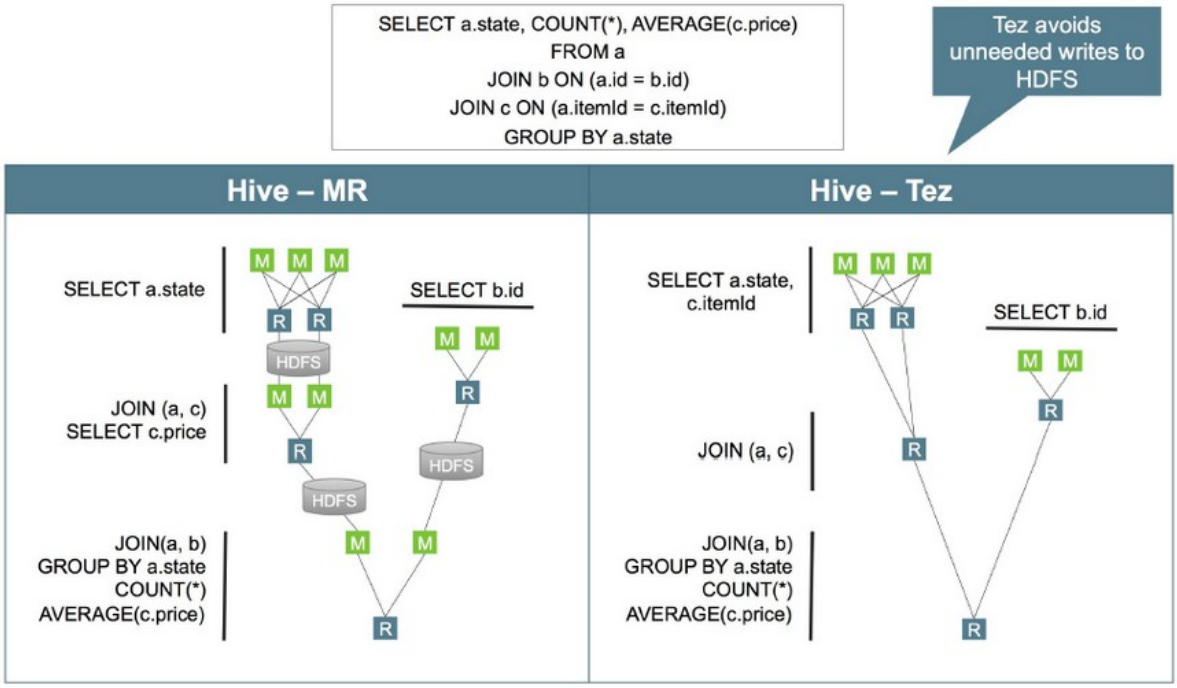
从以下三个方面展开：
架构优化
参数优化
SQL优化

第 1 节 架构优化

执行引擎

Hive支持多种执行引擎，分别是 MapReduce、Tez、Spark、Flink。可以通过hive-site.xml文件中的hive.execution.engine属性控制。

Tez是一个构建于YARN之上的支持复杂的DAG（有向无环图）任务的数据处理框架。由Hontonworks开源，将MapReduce的过程拆分成若干个子过程，同时可以把多个mapreduce任务组合成一个较大的DAG任务，减少了MapReduce之间的文件存储，同时合理组合其子过程从而大幅提升MR作业的性能。



优化器

与关系型数据库类似，Hive会在真正执行计算之前，生成和优化逻辑执行计划与物理执行计划。Hive有两种优化器：Vectorize(向量化优化器) 与 Cost-Based Optimization (CBO 成本优化器)。

- **矢量化查询执行**

矢量化查询(要求执行引擎为Tez)执行通过一次批量执行1024行而不是每行一行来提高扫描，聚合，过滤器和连接等操作的性能，这个功能一显着缩短查询执行时间。

```
set hive.vectorized.execution.enabled = true;                    -
- 默认 false
set hive.vectorized.execution.reduce.enabled = true;             -
- 默认 false
```

备注：要使用矢量化查询执行，必须用ORC格式存储数据

- **成本优化器**

Hive的CBO是基于apache Calcite的，Hive的CBO通过查询成本(有analyze收集的统计信息)会生成有效率的执行计划，最终会减少执行的时间和资源的利用，使用CBO的配置如下：

```
SET hive.cbo.enable=true;                          --从 v0.14.0默认
true
SET hive.compute.query.using.stats=true;           -- 默认false
SET hive.stats.fetch.column.stats=true;            -- 默认false
SET hive.stats.fetch.partition.stats=true;         -- 默认true
```

定期执行表（analyze）的分析，分析后的数据放在元数据库中。

分区表

对于一张比较大的表，将其设计成分区表可以提升查询的性能，对于一个特定分区的查询，只会加载对应分区路径的文件数据，所以执行速度会比较快。

分区字段的选择是影响查询性能的重要因素，尽量避免层级较深的分区，这样会造成太多的子文件夹。一些常见的分区字段可以是：

- 日期或时间。如year、month、day或者hour，当表中存在时间或者日期字段时
- 地理位置。如国家、省份、城市等

- 业务逻辑。如部门、销售区域、客户等等

分桶表

与分区表类似，分桶表的组织方式是将HDFS上的文件分割成多个文件。

分桶可以加快数据采样，也可以提升join的性能(join的字段是分桶字段)，因为分桶可以确保某个key对应的数据在一个特定的桶内(文件)，巧妙地选择分桶字段可以大幅度提升join的性能。

通常情况下，分桶字段可以选择经常用在过滤操作或者join操作的字段。

文件格式

在HiveQL的create table语句中，可以使用`stored as ...`指定表的存储格式。Hive表支持的存储格式有TextFile、SequenceFile、RCFile、ORC、Parquet等。

存储格式一般需要根据业务进行选择，生产环境中绝大多数表都采用TextFile、ORC、Parquet存储格式之一。

TextFile是最简单的存储格式，它是纯文本记录，也是Hive的默认格式。其磁盘开销大，查询效率低，更多的是作为跳板来使用。RCFile、ORC、Parquet等格式的表都不能由文件直接导入数据，必须由TextFile来做中转。

Parquet和ORC都是Apache旗下的开源列式存储格式。列式存储比起传统的行式存储更适合批量OLAP查询，并且也支持更好的压缩和编码。选择Parquet的原因主要是它支持Impala查询引擎，并且对update、delete和事务性操作需求很低。

数据压缩

压缩技术可以减少map与reduce之间的数据传输，从而可以提升查询性能，关于压缩的配置可以在hive的命令行中或者hive-site.xml文件中进行配置。

```
SET hive.exec.compress.intermediate=true
```

开启压缩之后，可以选择下面的压缩格式：

压缩格式	codec	扩展名	支持分割
Deflate	org.apache.hadoop.io.compress.DefaultCodec	.deflate	N
Gzip	org.apache.hadoop.io.compress.GzipCodec	.gz	N
Bzip2	org.apache.hadoop.io.compress.BZip2Codec	.gz	Y
LZO	com.apache.compression.lzo.LzopCodec	.lzo	N
LZ4	org.apache.hadoop.io.compress.Lz4Codec	.lz4	N
Snappy	org.apache.hadoop.io.compress.SnappyCodec	.snappy	N

关于压缩的编码器可以通过mapred-site.xml, hive-site.xml进行配置，也可以通过命令行进行配置，如：

```
-- 中间结果压缩
SET
hive.intermediate.compression.codec=org.apache.hadoop.io.compress
.SnappyCodec ;
-- 输出结果压缩
SET hive.exec.compress.output=true;
SET mapreduce.output.fileoutputformat.compress.codec =
org.apache.hadoop.io.compress.SnappyCodc
```

设计阶段：

```
执行引擎
优化器
分区、分桶
文件格式
数据压缩
```

第 2 节 参数优化

本地模式

当Hive处理的数据量较小时，启动分布式去处理数据会有点浪费，因为可能启动的时间比数据处理的时间还要长。Hive支持将作业动态地转为本地模式，需要使用下面的配置：

```
SET hive.exec.mode.local.auto=true; -- 默认 false
SET hive.exec.mode.local.auto.inputbytes.max=50000000;
SET hive.exec.mode.local.auto.input.files.max=5; -- 默认 4
```

一个作业只要满足下面的条件，会启用本地模式

- 输入文件的大小小于`hive.exec.mode.local.auto.inputbytes.max`配置的大小
- map任务的数量小于`hive.exec.mode.local.auto.input.files.max`配置的大小
- reduce任务的数量是1或者0

严格模式

所谓严格模式，就是强制不允许用户执行3种有风险的HiveQL语句，一旦执行会直接失败。这3种语句是：

- 查询分区表时不限定分区列的语句；
- 两表join产生了笛卡尔积的语句；
- 用order by来排序，但没有指定limit的语句。

要开启严格模式，需要将参数`hive.mapred.mode`设为strict(缺省值)。

该参数可以不在参数文件中定义，在执行SQL之前设置(set hive.mapred.mode=nostrict)

JVM重用

默认情况下，Hadoop会为为一个map或者reduce启动一个JVM，这样可以并行执行map和reduce。

当map或者reduce是那种仅运行几秒钟的轻量级作业时，JVM启动进程所耗费的时间会比作业执行的时间还要长。Hadoop可以重用JVM，通过共享JVM以串行而非并行的方式运行map或者reduce。

JVM的重用适用于同一个作业的map和reduce，对于不同作业的task不能够共享JVM。如果要开启JVM重用，需要配置一个作业最大task数量，默认值为1，如果设置为-1，则表示不限制：

```
# 代表同一个MR job中顺序执行的5个task重复使用一个JVM，减少启动和关闭的开销
SET mapreduce.job.jvm.numtasks=5;
```

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

并行执行

Hive的查询通常会被转换成一系列的stage，这些stage之间并不是一直相互依赖的，可以并行执行这些stage，通过下面的方式进行配置：

```
SET hive.exec.parallel=true; -- 默认false
SET hive.exec.parallel.thread.number=16; -- 默认8
```

并行执行可以增加集群资源的利用率，如果集群的资源使用率已经很高了，那么并行执行的效果不会很明显。

推测执行

在分布式集群环境下，因为程序Bug、负载不均衡、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采用了推测执行机制，它根据一定的规则推测出“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。

```
set mapreduce.map.speculative=true
set mapreduce.reduce.speculative=true
set hive.mapred.reduce.tasks.speculative.execution=true
```

合并小文件

- 在map执行前合并小文件，减少map数

```
# 缺省参数
set
hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFo
rmat;
```

- 在Map-Reduce的任务结束时合并小文件

```
# 在 map-only 任务结束时合并小文件，默认true
SET hive.merge.mapfiles = true;

# 在 map-reduce 任务结束时合并小文件，默认false
SET hive.merge.mapredfiles = true;

# 合并文件的大小，默认256M
SET hive.merge.size.per.task = 268435456;

# 当输出文件的平均大小小于该值时，启动一个独立的map-reduce任务进行文件merge
SET hive.merge.smallfiles.avgsize = 16777216;
```

Fetch模式

Fetch模式是指Hive中对某些情况的查询可以不必使用MapReduce计算。select col1, col2 from tab ;

可以简单地读取表对应的存储目录下的文件，然后输出查询结果到控制台。在开启fetch模式之后，在全局查找、字段查找、limit查找等都不启动 MapReduce 。

```
# Default Value: minimal in Hive 0.10.0 through 0.13.1, more in Hive 0.14.0 and later
hive.fetch.task.conversion=more
```

参数调整:

```
本地模式
严格模式
JVM重用
并行执行
推测还行
合并小文件
Fetch模式
```

Hive 参数说明的官方文档:

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

第 3 节 SQL优化

列裁剪和分区裁剪

列裁剪是在查询时只读取需要的列；分区裁剪就是只读取需要的分区。

简单的说：select 中不要有多余的列，坚决避免 select * from tab;

查询分区表，不读多余的数据；

```
select uid, event_type, record_data
  from calendar_record_log
 where pt_date >= 20190201 and pt_date <= 20190224
   and status = 0;
```

sort by 代替 order by

HiveQL中的order by与其他关系数据库SQL中的功能一样，是将结果按某字段全局排序，这会导致所有map端数据都进入一个reducer中，在数据量大时可能会长时间计算不完。

如果使用sort by，那么还是会视情况启动多个reducer进行排序，并且保证每个reducer内局部有序。为了控制map端数据分配到reducer的key，往往还要配合distribute by 一同使用。如果不加 distribute by 的话，map端数据就会随机分配到reducer。

group by 代替 count(distinct)

当要统计某一列的去重数时，如果数据量很大，count(distinct) 会非常慢。原因与order by类似，count(distinct)逻辑只会有很少的reducer来处理。此时可以用group by 来改写：

```
-- 原始SQL
select count(distinct uid)
  from tab;

-- 优化后的SQL
select count(1)
  from (select uid
          from tab
         group by uid) tmp;
```

这样写会启动两个MR job（单纯distinct只会启动一个），所以要确保数据量大到启动job的overhead远小于计算耗时，才考虑这种方法。当数据集很小或者key的倾斜比较明显时，group by还可能会比distinct慢。

group by 配置调整

map端预聚合

group by时，如果先起一个combiner在map端做部分预聚合，可以有效减少shuffle数据量。

```
-- 默认为true
set hive.map.aggr = true
```

Map端进行聚合操作的条目数

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

通过`hive.groupby.mapaggr.checkinterval`参数也可以设置map端预聚合的行数阈值，超过该值就会分拆job，默认值10W。

倾斜均衡配置项

group by时如果某些key对应的数据量过大，就会发生数据倾斜。Hive自带了一个均衡数据倾斜的配置项`hive.groupby.skewindata`，默认值false。

其实现方法是在group by时启动两个MR job。第一个job会将map端数据随机输入reducer，每个reducer做部分聚合，相同的key就会分布在不同的reducer中。第二个job再将前面预处理过的数据按key聚合并输出结果，这样就起到了均衡的效果。

但是，配置项毕竟是死的，单纯靠它有时不能根本上解决问题，建议了解数据倾斜的细节，并优化查询语句。

join 基础优化

Hive join的三种方式

1、common join

普通连接，在SQL中不特殊指定连接方式使用的都是这种普通连接。

缺点：性能较差(要将数据分区，有shuffle)

优点：操作简单，普适性强

2、map join

map端连接，与普通连接的区别是这个连接中不会有reduce阶段存在，连接在map端完成

适用场景：大表与小表连接，小表数据量应该能够完全加载到内存，否则不适用

优点：在大小表连接时性能提升明显

备注：Hive 0.6 的时候默认认为写在select 后面的是大表，前面的是小表， 或者使用 /*+mapjoin(map_table) */ *提示进行设定。select a.*, b.* from a join b on a.id = b.id【要求小表在前，大表之后】

hive 0.7 的时候这个计算是自动化的，它首先会自动判断哪个是小表，哪个是大表，这个参数由（hive.auto.convert.join=true）来控制，然后控制小表的大小由（hive.smalltable.filesize=25000000）参数控制（默认是25M），当小表超过这个大小，hive 会默认转化成common join。

Hive 0.8.1，hive.smalltable.filesize => hive.mapjoin.smalltable.filesize

缺点：使用范围较小，只针对大小表且小表能完全加载到内存中的情况。

3、bucket map join

分桶连接：Hive 建表的时候支持hash 分区通过指定clustered by (col_name,xxx) into number_buckets buckets 关键字.当连接的两个表的join key 就是bucket column 的时候，就可以通过设置hive.optimize.bucketmapjoin= true 来执行优化。

原理：通过两个表分桶在执行连接时会将小表的每个分桶映射成hash表，每个task节点都需要这个小表的所有hash表，但是在执行时只需要加载该task所持有大表分桶对应的小表部分的hash表就可以，所以对内存的要求是能够加载小表中最大的hash块即可。

备注：小表与大表的分桶数量需要是倍数关系，这个是因为分桶策略决定的，分桶时会根据分桶字段对桶数取余后决定哪个桶的，所以要保证成倍数关系。

优点：比map join对内存的要求降低，能在逐行对比时减少数据计算量（不用比对小表全量）

缺点：只适用于分桶表

利用map join特性

map join特别适合大小表join的情况。Hive会将build table和probe table在map端直接完成join过程，消灭了reduce，效率很高。

```
select a.event_type, b.upload_time
from calendar_event_code a
inner join (
  select event_type, upload_time from calendar_record_log
  where pt_date = 20190225
) b on a.event_type = b.event_type;
```

map join的配置项是`hive.auto.convert.join`，默认值true。

当build table大小小于`hive.mapjoin.smalltable.filesize`会启用map join，默认值25000000（约25MB）。还有`hive.mapjoin.cache.numrows`，表示缓存build table的多少行数据到内存，默认值25000。

分桶表map join

map join对分桶表还有特别的优化。由于分桶表是基于一列进行hash存储的，因此非常适合抽样（按桶或按块抽样）。它对应的配置项是`hive.optimize.bucketmapjoin`。

倾斜均衡配置项

这个配置与 group by 的倾斜均衡配置项异曲同工，通过`hive.optimize.skewjoin`来配置，默认false。

如果开启了，在join过程中Hive会将计数超过阈值`hive.skewjoin.key`（默认100000）的倾斜key对应的行临时写进文件中，然后再启动另一个job做map join生成结果。通过`hive.skewjoin.mapjoin.map.tasks`参数还可以控制第二个job的mapper数量，默认10000。

处理空值或无意义值

日志类数据中往往会有一些项没有记录到，其值为null，或者空字符串、-1等。如果缺失的项很多，在做join时这些空值就会非常集中，拖累进度【备注：这个字段是连接字段】。

若不需要空值数据，就提前写 where 语句过滤掉。需要保留的话，将空值key用随机方式打散，例如将用户ID为null的记录随机改为负值：

```
select a.uid, a.event_type, b.nickname, b.age
from (
  select
  (case when uid is null then cast(rand()*-10240 as int) else uid
end) as uid,
  event_type from calendar_record_log
  where pt_date >= 20190201
) a left outer join (
  select uid,nickname,age from user_info where status = 4
) b on a.uid = b.uid;
```

单独处理倾斜key

如果倾斜的 key 有实际的意义，一般来讲倾斜的key都很少，此时可以将它们单独抽取出来，对应的行单独存入临时表中，然后打上一个较小的随机数前缀（比如0~9），最后再进行聚合。

不要一个Select语句中，写太多的Join。一定要了解业务，了解数据。(A0-A9)

分成多条语句，分步执行；(A0-A4; A5-A9)；先执行大表与小表的关联；

调整 Map 数

通常情况下，作业会通过输入数据的目录产生一个或者多个map任务。主要因素包括：

- 输入文件总数
- 输入文件大小
- HDFS文件块大小

map越多越好吗。当然不是，合适的才是最好的。

如果一个任务有很多小文件（<< 128M），每个小文件也会被当做一个数据块，用一个 Map Task 来完成。

一个 Map Task 启动和初始化时间 >> 处理时间，会造成资源浪费，而且系统中同时可用的map数是有限的。

对于小文件采用的策略是合并。

每个map处理接近128M的文件块，会有其他问题吗。也不一定。

有一个125M的文件，一般情况下会用一个Map Task完成。假设这个文件字段很少，但记录数却非常多。如果Map处理的逻辑比较复杂，用一个map任务去做，性能也不好。

对于复杂文件采用的策略是增加 Map 数。

```
computeSliteSize(max(minSize, min(maxSize, blocksize))) =
blocksize
minSize : mapred.min.split.size （默认值1）
maxSize : mapred.max.split.size （默认值256M）

调整maxSize最大值。让maxSize最大值低于blocksize就可以增加map的个数。
建议用set的方式，针对SQL语句进行调整。
```

调整 Reduce 数

reducer数量的确定方法比mapper简单得多。使用参数`mapred.reduce.tasks`可以直接设定reducer数量。如果未设置该参数，Hive会进行自行推测，逻辑如下：

- 参数`hive.exec.reducers.bytes.per.reducer`用来设定每个reducer能够处理的最大数据量，默认值256M
- 参数`hive.exec.reducers.max`用来设定每个job的最大reducer数量，默认值999（1.2版本之前）或1009（1.2版本之后）
- 得出reducer数：`reducer_num = MIN(total_input_size / reducers.bytes.per.reducer, reducers.max)`

  即： min(输入总数据量 / 256M, 1009)

reducer数量与输出文件的数量相关。如果reducer数太多，会产生大量小文件，对HDFS造成压力。如果reducer数太少，每个reducer要处理很多数据，容易拖慢运行时间或者造成OOM。

第 4 节 优化小结

深入理解 Hadoop 的核心能力，对Hive优化很有帮助。Hadoop/Hive 处理数据过程，有几个显著特征：

- 不怕数据多，就怕数据倾斜
- 对 job 数比较多的作业运行效率相对比较低，比如即使有几百行的表，多次关联多次汇总，产生十几个jobs，执行也需要较长的时间。MapReduce 作业初始化的时间是比较长的
- 对sum、count等聚合操作而言，不存在数据倾斜问题
- count(distinct) 效率较低，数据量大容易出问题

从大的方面来说，优化可以从几个方面着手：

- 好的模型设计，事半功倍
- 解决数据倾斜问题。仅仅依靠参数解决数据倾斜，是通用的优化手段，收获有限。开发人员应该熟悉业务，了解数据规律，通过业务逻辑解决数据倾斜往往更可靠
- 减少 job 数
- 设置合理的map、reduce task数
- 对小文件进行合并，是行之有效的提高Hive效率的方法
- 优化把握整体，单一作业的优化不如整体最优

第十一部分 Hive案例

综合Hive知识，复习巩固。

第 1 节 需求描述

针对销售数据，完成统计：

1. 按年统计销售额
2. 销售金额在 10W 以上的订单
3. 每年销售额的差值
4. 年度订单金额前10位（年度、订单号、订单金额、排名）
5. 季度订单金额前10位（年度、季度、订单id、订单金额、排名）
6. 求所有交易日中订单金额最高的前10位
7. 每年度销售额最大的交易日
8. 年度最畅销的商品(即每年销售金额最大的商品)

第 2 节 数据说明

日期表(dimdate)		
dt	date	日期
yearmonth	int	年月
year	smallint	年
month	tinyint	月
day	tinyint	日
week	tinyint	周几
weeks	tinyint	第几周
quat	tinyint	季度
tendays	tinyint	旬
halfmonth	tinyint	半月
订单表(sale)		
orderid	string	订单号
locationid	string	交易位置
dt	date	交易日期
订单销售明细表(saledetail)		
orderid	string	订单号
rownum	int	行号
itemid	string	货品
num	int	数量
price	double	单价
amount	double	金额

第 3 节 实现

步骤一：创建表

将数据存放在ORC文件中

```
-- createtable.hql
drop database sale cascade;
create database if not exists sale;
create table sale.dimdate_ori(
    dt date,
    yearmonth int,
    year smallint,
    month tinyint,
    day tinyint,
    week tinyint,
    weeks tinyint,
    quat tinyint,
    tendays tinyint,
    halfmonth tinyint
)
row format delimited
fields terminated by ",";

create table sale.sale_ori(
    orderid string,
    locationid string,
    dt date
)
row format delimited
fields terminated by ",";

create table sale.saledetail_ori(
    orderid string,
    rownum int,
    goods string,
    num int,
    price double,
    amount double
)
row format delimited
fields terminated by ",";

create table sale.dimdate(
    dt date,
    yearmonth int,
    year smallint,
    month tinyint,
    day tinyint,
    week tinyint,
    weeks tinyint,
```

```
    quat tinyint,
    tendays tinyint,
    halfmonth tinyint
) stored as orc;

create table sale.sale(
    orderid string,
    locationid string,
    dt date
) stored as orc;

create table sale.saledetail(
    orderid string,
    rownum int,
    goods string,
    num int,
    price double,
    amount double
)stored as orc;

hive -f createtable.hql
```

步骤二：导入数据

```
-- 加载数据
use sale;
load data local inpath "/root/data/tbDate.dat" overwrite into
table dimdate_ori;
load data local inpath "/root/data/tbSale.dat" overwrite into
table sale_ori;
load data local inpath "/root/data/tbSaleDetail.dat" overwrite
into table saledetail_ori;

-- 导入数据
insert into table dimdate select * from dimdate_ori;
insert into table sale select * from sale_ori;
insert into table saledetail select * from saledetail_ori;

hive -f loaddata.hql
```

步骤三：SQL实现

1、按年统计销售额

```
SELECT year(B.dt) year, round(sum(A.amount)/10000, 2) amount
  FROM saledetail A join sale B on A.orderid=B.orderid
group by year(B.dt);
```

2、销售金额在 10W 以上的订单

```
SELECT orderid, round(sum(amount), 2) amount
  FROM saledetail
group by orderid
having sum(amount) > 100000
```

3、每年销售额的差值

```
SELECT year, round(amount, 2) amount, round(lag(amount) over
(ORDER BY year), 2) prioramount
      ,round(amount - lag(amount) over (ORDER BY year), 2) diff
  from (SELECT year(B.dt) year, sum(A.amount) amount
          from saledetail A join sale B on A.orderid=B.orderid
         group by year(B.dt)
         ) tmp;
```

4、年度订单金额前10位（年度、订单号、订单金额、排名）

```
-- 方法一
SELECT dt, orderid, amount, rank
  from (SELECT dt, orderid, amount,
               dense_rank() over(PARTITION BY dt ORDER BY amount
desc) rank
          from (SELECT year(B.dt) dt, A.orderid, sum(A.amount)
amount
                  from saledetail A join sale B on
A.orderid=B.orderid
                 GROUP BY year(B.dt), A.orderid
                 ) tmp1
        ) tmp2
 where rank <= 10;

-- 方法二
```

```
with tmp as (
SELECT year(B.dt) dt, A.orderid, sum(A.amount) amount
  from saledetail A join sale B on A.orderid=B.orderid
GROUP BY year(B.dt), A.orderid
)
SELECT dt, orderid, amount, rank
  from (SELECT dt, orderid, amount,
               dense_rank() over(PARTITION BY dt ORDER BY amount
desc) rank
          from tmp
        ) tmp2
 where rank <= 10;
```

5、季度订单金额前10位（年度、季度、订单id、订单金额、排名）

```
-- 方法一
with tmp as (
select C.year, C.quat, A.orderid, round(sum(B.amount), 2) amount
  from sale A join saledetail B on A.orderid=B.orderid
              join dimdate C    on A.dt=C.dt
group by C.year, C.quat, A.orderid
)
select year, quat, orderid, amount, rank
  from (
       select year, quat, orderid, amount,
              dense_rank() over (partition by year, quat order by
amount desc) rank
       from tmp
       ) tmp1
 where rank <= 10;


-- 方法二
with tmp as(
select year(A.dt) year,
       case when month(A.dt) <= 3 then 1
            when month(A.dt) <= 6 then 2
            when month(A.dt) <= 9 then 3
            else 4 end quat,
       A.orderid,
       round(sum(B.amount), 2) amount
  from sale A join saledetail B on A.orderid = B.orderid
group by year(A.dt),
```

```
        case when month(A.dt) <= 3 then 1
             when month(A.dt) <= 6 then 2
             when month(A.dt) <= 9 then 3
             else 4 end,
        A.orderid
)
select year, quat, orderid, amount, rank
  from (
       select year, quat, orderid, amount,
              dense_rank() over (partition by year, quat order by
amount desc) rank
       from tmp
       ) tmp1
 where rank <= 10;

-- 方法三。求季度
select floor(month(dt/3.1)) + 1;

with tmp as (
select year(A.dt) year, floor(month(A.dt)/3.1) + 1 quat,
       A.orderid,
       round(sum(B.amount), 2) amount
  from sale A join saledetail B on A.orderid=B.orderid
group by year(A.dt), floor(month(A.dt)/3.1) + 1, A.orderid
)
select year, quat, orderid, amount, rank
  from (
       select year, quat, orderid, amount,
              dense_rank() over (partition by year, quat order by
amount desc) rank
       from tmp
       ) tmp1
 where rank <= 10;
```

6、求所有交易日中订单金额最高的前10位

```
topN问题：
1、基础数据
2、上排名函数
3、解决N的问题

with tmp as (
select A.dt, A.orderid, round(sum(B.amount), 2) amount
```

```
  from sale A join saledetail B on A.orderid=B.orderid
group by A.dt, A.orderid
)
select dt, orderid, amount, rank
  from (
        select dt, orderid, amount, dense_rank() over(order by
amount desc) rank
        from tmp
       ) tmp1
 where rank <= 10;
```

7、每年度销售额最大的交易日

```
with tmp as (
select A.dt, round(sum(B.amount), 2) amount
  from sale A join saledetail B on A.orderid=B.orderid
group by A.dt
)
select year(dt) year, max(amount) dayamount
  from tmp
group by year(dt);
```

8、年度最畅销的商品(即每年销售金额最大的商品)

```
with tmp as (
select year(B.dt) year, goods, round(sum(amount),2) amount
  from saledetail A join sale B on A.orderid=B.orderid
group by year(B.dt), goods
)
select year, goods, amount
  from (select year, goods, amount, dense_rank() over (partition
by year order by amount desc) rank
        from tmp) tmp1
 where rank = 1;
```