Hive

第一章 简介及核心概念

简介

Hive 是一个构建在 Hadoop 之上的`数据仓库`，它可以将`结构化的数据文件映射成表`，并提供类`SQL 查询功能`，用于查询的 SQL 语句`会被转化为 MapReduce 作业`，然后提交到 Hadoop 上运行。

特点：

1. 简单、容易上手 (提供了类似 sql 的查询语言 hql)，使得精通 sql 但是不了解 Java 编程的人也能很好地进行大数据分析；
2. 灵活性高，可以自定义用户函数 (UDF) 和存储格式；
3. 为超大的数据集设计的计算和存储能力，集群扩展容易；
4. 统一的元数据管理，可与 presto / impala / sparksql 等共享数据；
5. 执行延迟高，不适合做数据的实时处理，但适合做海量数据的离线处理。

Hive的体系架构

2.1 command-line shell & thrift/jdbc

可以用 command-line shell 和 thrift / jdbc 两种方式来操作数据：

- **command-line shell**：通过`hive命令行`的方式来操作数据；
- **thrift / jdbc**：通过`thrift协议`按照标准的 JDBC 的方式操作数据。

2.2 Metastore

在 Hive 中，表名、表结构、字段名、字段类型、表的分隔符等统一被称为`元数据`。`所有的元数据默认存储在 Hive内置的 derby数据库中`，但由于 derby 只能有一个实例，也就是说`不能有多个命令行客户端同时访问`，所以在实际生产环境中，通常使用 MySQL 代替 derby。

Hive 进行的是统一的元数据管理，就是说你在 Hive 上创建了一张表，然后在 presto / impala / sparksql 中都是可以直接使用的，它们会从 Metastore 中获取统一的元数据信息，同样的你在 presto / impala / sparksql 中创建一张表，在 Hive 中也可以直接使用。

2.3 HQL的执行流程

Hive 在执行一条 HQL 的时候，会经过以下步骤：

1. 语法解析：Antlr 定义 SQL 的语法规则，完成 SQL 词法，语法解析，将 SQL 转化为抽象 语法树 AST Tree；
2. 语义解析：遍历 AST Tree，抽象出查询的基本组成单元 QueryBlock；
3. 生成逻辑执行计划：遍历 QueryBlock，翻译为执行操作树 OperatorTree；
4. 优化逻辑执行计划：逻辑层优化器进行 OperatorTree 变换，合并不必要的 ReduceSinkOperator，减少 shuffle 数据量；
5. 生成物理执行计划：遍历 OperatorTree，翻译为 MapReduce 任务；
6. 优化物理执行计划：物理层优化器进行 MapReduce 任务的变换，生成最终的执行计划。

数据类型

3.1 基本数据类型

Hive 表中的列支持以下基本数据类型：

大类	类型
Integers（整型）	TINYINT—1 字节的有符号整数 SMALLINT—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
Boolean（布尔型）	BOOLEAN—TRUE/FALSE
Floating point numbers（浮点型）	FLOAT— 单精度浮点型 DOUBLE—双精度浮点型
Fixed point numbers（定点数）	DECIMAL—用户自定义精度定点数，比如 DECIMAL(7,2)
String types（字符串）	STRING—指定字符集的字符序列 VARCHAR—具有最大长度限制的字符序列 CHAR—固定长度的字符序列
Date and time types（日期时间类型）	TIMESTAMP — 时间戳 TIMESTAMP WITH LOCAL TIME ZONE — 时间戳，纳秒精度 DATE—日期类型
Binary types（二进制类型）	BINARY—字节序列

> TIMESTAMP 和 TIMESTAMP WITH LOCAL TIME ZONE 的区别如下：
>
> - **TIMESTAMP WITH LOCAL TIME ZONE**：用户提交时间给数据库时，会被转换成数据库所在的时区来保存。查询时则按照查询客户端的不同，转换为查询客户端所在时区的时间。
> - **TIMESTAMP** ：提交什么时间就保存什么时间，查询时也不做任何转换。

3.2 隐式转换

Hive 中基本数据类型遵循以下的层次结构，按照这个层次结构，子类型到祖先类型允许隐式转换。例如 INT 类型的数据允许隐式转换为 BIGINT 类型。额外注意的是：按照类型层次结构允许将 STRING 类型隐式转换为 DOUBLE 类型。

3.3 复杂类型

类型	描述	示例
STRUCT	类似于对象，是字段的集合，字段的类型可以不同，可以使用 `名称.字段名` 方式进行访问	STRUCT ('xiaoming', 12 , '2018-12-12')
MAP	键值对的集合，可以使用 `名称[key]` 的方式访问对应的值	map('a', 1, 'b', 2)
ARRAY	数组是一组具有相同类型和名称的变量的集合，可以使用 `名称[index]` 访问对应的值	ARRAY('a', 'b', 'c', 'd')

3.4 示例

如下给出一个基本数据类型和复杂数据类型的使用示例：

```
CREATE TABLE students(
  name         STRING,     -- 姓名
  age          INT,        -- 年龄
  subject      ARRAY<STRING>,    --学科
  score        MAP<STRING,FLOAT>,    --各个学科考试成绩
  address      STRUCT<houseNumber:int, street:STRING, city:STRING, province: STRING>   --家庭居住地址
) ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";
```

内容格式

当数据存储在文本文件中，`必须按照一定格式区别行和列`，如使用逗号作为分隔符的 CSV 文件 (Comma-Separated Values) 或者使用制表符作为分隔值的 TSV 文件 (Tab-Separated Values)。但此时也存在一个缺点，就是正常的文件内容中也可能出现逗号或者制表符。

所以 Hive 默认使用了几个平时很少出现的字符，这些字符一般不会作为内容出现在文件中。Hive 默认的行和列分隔符如下表所示。

分隔符	描述
\n	对于文本文件来说，每行是一条记录，所以可以使用换行符来分割记录
^A (Ctrl+A)	分割字段 (列)，在 CREATE TABLE 语句中也可以使用八进制编码 `\001` 来表示
^B	用于分割 ARRAY 或者 STRUCT 中的元素，或者用于 MAP 中键值对之间的分割，在 CREATE TABLE 语句中也可以使用八进制编码 `\002` 表示
^C	用于 MAP 中键和值之间的分割，在 CREATE TABLE 语句中也可以使用八进制编码 `\003` 表示

使用示例如下：

```
CREATE TABLE page_view(viewTime INT, userid BIGINT)
 ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\001'
    COLLECTION ITEMS TERMINATED BY '\002'
    MAP KEYS TERMINATED BY '\003'
 STORED AS SEQUENCEFILE;
```

存储格式

5.1 支持的存储格式

`Hive 会在 HDFS 为每个数据库上创建一个目录，数据库中的表是该目录的子目录，表中的数据会以文件的形式存储在对应的表目录下`。Hive 支持以下几种文件存储格式：

格式	说明
TextFile	存储为纯文本文件。 这是 Hive 默认的文件存储格式。这种存储方式数据不做压缩，磁盘开销大，数据解析开销大。
SequenceFile	SequenceFile 是 Hadoop API 提供的一种二进制文件，它将数据以<key,value>的形式序列化到文件中。这种二进制文件内部使用 Hadoop 的标准的 Writable 接口实现序列化和反序列化。它与 Hadoop API 中的 MapFile 是互相兼容的。Hive 中的 SequenceFile 继承自 Hadoop API 的 SequenceFile，不过它的 key 为空，使用 value 存放实际的值，这样是为了避免 MR 在运行 map 阶段进行额外的排序操作。
RCFile	RCFile 文件格式是 FaceBook 开源的一种 Hive 的文件存储格式，首先将表分为几个行组，对每个行组内的数据按列存储，每一列的数据都是分开存储。
ORC Files	ORC 是在一定程度上扩展了 RCFile，是对 RCFile 的优化。
Avro Files	Avro 是一个数据序列化系统，设计用于支持大批量数据交换的应用。它的主要特点有：支持二进制序列化方式，可以便捷，快速地处理大量数据；动态语言友好，Avro 提供的机制使动态语言可以方便地处理 Avro 数据。
Parquet	Parquet 是基于 Dremel 的数据模型和算法实现的，面向分析型业务的列式存储格式。它通过按列进行高效压缩和特殊的编码技术，从而在降低存储空间的同时提高了 IO 效率。

> 以上压缩格式中 ORC 和 `Parquet` 的综合性能突出，使用较为广泛，推荐使用这两种格式。

5.2 指定存储格式

通常在创建表的时候使用 `STORED AS` 参数指定：

```
CREATE TABLE page_view(viewTime INT, userid BIGINT)
 ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\001'
    COLLECTION ITEMS TERMINATED BY '\002'
    MAP KEYS TERMINATED BY '\003'
 STORED AS SEQUENCEFILE;
```

各个存储文件类型指定方式如下：

- STORED AS TEXTFILE
- STORED AS SEQUENCEFILE
- STORED AS ORC
- STORED AS PARQUET
- STORED AS AVRO
- STORED AS RCFILE

内部表和外部表

`内部表又叫做管理表` (Managed/Internal Table)，创建表时不做任何指定，默认创建的就是内部表。想要创建外部表 (External Table)，则需要使用 `External` 进行修饰。 内部表和外部表主要区别如下：

	内部表	外部表
数据存储位置	内部表数据存储的位置由 hive.metastore.warehouse.dir 参数指定，默认情况下表的数据存储在 HDFS 的 `/user/hive/warehouse/数据库名.db/表名/` 目录下	外部表数据的存储位置创建表时由 `Location` 参数指定；
导入数据	在导入数据到内部表，内部表将数据移动到自己的数据仓库目录下，数据的生命周期由 Hive 来进行管理	外部表不会将数据移动到自己的数据仓库目录下，只是在元数据中存储了数据的位置
删除表	`删除元数据（metadata）和文件`	`只删除元数据（metadata）`

第二章 Hive常用DML操作

加载文件数据到表

1.1 语法

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

- `LOCAL` 关键字代表 `从本地文件系统加载文件`，省略则代表从 HDFS 上加载文件：
- 从本地文件系统加载文件时， `filepath` 可以是 `绝对路径也可以是相对路径` (建议使用绝对路径)；
- 从 HDFS 加载文件时候， `filepath` 为文件完整的 URL 地址：如 `hdfs://namenode:port/user/hive/project/ data1`
- `filepath` 可以是文件路径 (在这种情况下 Hive 会将文件移动到表中)，也可以目录路径 (在这种情况下，Hive 会将该目录中的所有文件移动到表中)；
- 如果使用 `OVERWRITE` 关键字， `则将删除目标表（或分区）的内容，使用新的数据填充` ；不使用此关键字，则数据以 `追加的方式` 加入；
- 加载的目标可以是表或分区。如果是分区表，则必须指定加载数据的分区；
- 加载文件的格式必须与建表时使用 `STORED AS` 指定的存储格式相同。

> 使用建议：
>
> **不论是本地路径还是 URL 都建议使用完整的。**虽然可以使用不完整的 URL 地址，此时 Hive 将使用 hadoop 中的 fs.default.name 配置来推断地址，但是为避免不必要的错误，建议使用完整的本地路径或 URL 地址；
>
> **加载对象是分区表时建议显示指定分区。**在 Hive 3.0 之后，内部将加载 (LOAD) 重写为 INSERT AS SELECT，此时如果不指定分区，INSERT AS SELECT 将假设最后一组列是分区列，如果该列不是表定义的分区，它将抛出错误。为避免错误，还是建议显示指定分区。

1.2 示例

新建分区表：

```
    CREATE TABLE emp_ptn(
      empno INT,
      ename STRING,
      job STRING,
      mgr INT,
      hiredate TIMESTAMP,
      sal DECIMAL(7,2),
      comm DECIMAL(7,2)
       )
      PARTITIONED BY (deptno INT)    -- 按照部门编号进行分区
       ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";
```

从 HDFS 上加载数据到分区表：

```
LOAD DATA  INPATH "hdfs://hadoop001:8020/mydir/emp.txt" OVERWRITE INTO TABLE emp_ptn PARTITION (deptno=20);
```

查询结果插入到表

2.1 语法

```
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]]
select_statement1 FROM from_statement;

INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)]
select_statement1 FROM from_statement;
```

- Hive 0.13.0 开始，建表时可以通过使用 TBLPROPERTIES（“immutable”=“true”）来创建不可变表 (immutable table)，如果不可以变表中存在数据，则 INSERT INTO 失败。（注：INSERT OVERWRITE 的语句不受 `immutable` 属性的影响）；
- 可以对表或分区执行插入操作。如果表已分区，则必须通过指定所有分区列的值来指定表的特定分区；
- 从 Hive 1.1.0 开始，TABLE 关键字是可选的；
- 从 Hive 1.2.0 开始，可以采用 INSERT INTO tablename(z，x，c1) 指明插入列；
- 可以将 SELECT 语句的查询结果插入多个表（或分区），称为 `多表插入`。语法如下：

```
FROM from_statement
INSERT OVERWRITE TABLE tablename1
[PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select_statement1
[INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select_statement2]
[INSERT INTO TABLE tablename2 [PARTITION ...] select_statement2] ...;
```

2.2 动态插入分区

```
INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...)
select_statement FROM from_statement;

INSERT INTO TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...)
select_statement FROM from_statement;
```

在向分区表插入数据时候，分区列名是必须的，但是列值是可选的。如果给出了分区列值，我们将其称为静态分区，否则它是动态分区。动态分区列必须在 SELECT 语句的列中最后指定，并且与它们在 PARTITION() 子句中出现的顺序相同。

注意：Hive 0.9.0 之前的版本动态分区插入是默认禁用的，而 0.9.0 之后的版本则默认启用。以下是动态分区的相关配置：

配置	默认值	说明
hive.exec.dynamic.partition	true	需要设置为 true 才能启用动态分区插入
hive.exec.dynamic.partition.mode	strict	在严格模式 (strict) 下，用户必须至少指定一个静态分区，以防用户意外覆盖所有分区，在非严格模式下，允许所有分区都是动态的
hive.exec.max.dynamic.partitions.pernode	100	允许在每个 mapper/reducer 节点中创建的最大动态分区数
hive.exec.max.dynamic.partitions	1000	允许总共创建的最大动态分区数
hive.exec.max.created.files	100000	作业中所有 mapper/reducer 创建的 HDFS 文件的最大数量
hive.error.on.empty.partition	false	如果动态分区插入生成空结果，是否抛出异常

2.3 示例

1. 新建 emp 表，作为查询对象表

```
CREATE TABLE emp(
     empno INT,
     ename STRING,
     job STRING,
     mgr INT,
     hiredate TIMESTAMP,
     sal DECIMAL(7,2),
     comm DECIMAL(7,2),
     deptno INT)
      ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";

 -- 加载数据到 emp 表中 这里直接从本地加载
load data local inpath "/usr/file/emp.txt" into table emp;
```

2. 为清晰演示，先清空 emp_ptn 表中加载的数据：

```
TRUNCATE TABLE emp_ptn;
```

3. 静态分区演示：从 emp 表中查询部门编号为 20 的员工数据，并插入 emp_ptn 表中，语句如下：

```
INSERT OVERWRITE TABLE emp_ptn PARTITION (deptno=20)
SELECT empno,ename,job,mgr,hiredate,sal,comm FROM emp WHERE deptno=20;
```

4. 接着演示动态分区：

```
-- 由于我们只有一个分区，且还是动态分区，所以需要关闭严格默认。因为在严格模式下，用户必须至少指定一个静态分区
set hive.exec.dynamic.partition.mode=nonstrict;

-- 动态分区    此时查询语句的最后一列为动态分区列，即 deptno
INSERT OVERWRITE TABLE emp_ptn PARTITION (deptno)
SELECT empno,ename,job,mgr,hiredate,sal,comm,deptno FROM emp WHERE deptno=30;
```

使用SQL语句插入值

```
INSERT INTO TABLE tablename [PARTITION (partcol1[=val1], partcol2[=val2] ...)]
VALUES ( value [, value ...] )
```

- 使用时必须为表中的每个列都提供值。不支持只向部分列插入值（可以为缺省值的列提供空值来消除这个弊端）；
- 如果目标表表支持 ACID 及其事务管理器，则插入后自动提交；
- 不支持支持复杂类型 (array, map, struct, union) 的插入。

更新和删除数据

4.1 语法

更新和删除的语法比较简单，和关系型数据库一致。需要注意的是这两个操作都只能在支持 ACID 的表，也就是事务表上才能执行。

```
-- 更新
UPDATE tablename SET column = value [, column = value ...] [WHERE expression]

--删除
DELETE FROM tablename [WHERE expression]
```

4.2 示例

1. 修改配置

首先需要更改 `hive-site.xml` ，添加如下配置，开启事务支持，配置完成后需要重启 Hive 服务。

```
<property>
      <name>hive.support.concurrency</name>
      <value>true</value>
</property>
<property>
      <name>hive.enforce.bucketing</name>
      <value>true</value>
</property>
<property>
      <name>hive.exec.dynamic.partition.mode</name>
      <value>nonstrict</value>
</property>
<property>
```

```
        <name>hive.txn.manager</name>
        <value>org.apache.hadoop.hive.ql.lockmgr.DbTxnManager</value>
  </property>
  <property>
        <name>hive.compactor.initiator.on</name>
        <value>true</value>
  </property>
  <property>
        <name>hive.in.test</name>
        <value>true</value>
  </property>
```

2. 创建测试表

创建用于测试的事务表，建表时候指定属性 `transactional = true` 则代表该表是事务表。需要注意的是，按照 **官方文档** 的说明，目前 Hive 中的事务表有以下限制：

- 必须是 buckets Table;
- 仅支持 ORC 文件格式；
- 不支持 LOAD DATA ...语句。

```
CREATE TABLE emp_ts(
  empno int,
  ename String
)
CLUSTERED BY (empno) INTO 2 BUCKETS STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

3. 插入测试数据

```
INSERT INTO TABLE emp_ts   VALUES (1,"ming"),(2,"hong");
```

插入数据依靠的是 MapReduce 作业，执行成功后数据如下：

4. 测试更新和删除

```
--更新数据
UPDATE emp_ts SET ename = "lan"   WHERE  empno=1;

--删除数据
DELETE FROM emp_ts WHERE empno=2;
```

更新和删除数据依靠的也是 MapReduce 作业，执行成功后数据如下：

查询结果写出到文件系统

5.1 语法

```
INSERT OVERWRITE [LOCAL] DIRECTORY directory1
  [ROW FORMAT row_format] [STORED AS file_format]
  SELECT ... FROM ...
```

- OVERWRITE 关键字表示输出文件存在时，先删除后再重新写入；
- 和 Load 语句一样，建议无论是本地路径还是 URL 地址都使用完整的；

- 写入文件系统的数据被序列化为文本，其中列默认由^A 分隔，行由换行符分隔。如果列不是基本类型，则将其序列化为 JSON 格式。其中行分隔符不允许自定义，但列分隔符可以自定义，如下：

```
-- 定义列分隔符为'\t'
insert overwrite local directory './test-04'
row format delimited
FIELDS TERMINATED BY '\t'
COLLECTION ITEMS TERMINATED BY ','
MAP KEYS TERMINATED BY ':'
select * from src;
```

5.2 示例

这里我们将上面创建的 `emp_ptn` 表导出到本地文件系统，语句如下：

```
INSERT OVERWRITE LOCAL DIRECTORY '/usr/file/ouput'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
SELECT * FROM emp_ptn;
```

第三章 Hive常用DDL操作

Database

1.1 查看数据列表

```
show databases;
```

1.2 使用数据库

```
USE database_name;
```

1.3 新建数据库

语法：

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name    --DATABASE|SCHEMA 是等价的
   [COMMENT database_comment] --数据库注释
   [LOCATION hdfs_path] --存储在 HDFS 上的位置
   [WITH DBPROPERTIES (property_name=property_value, ...)]; --指定额外属性
```

示例：

```
CREATE DATABASE IF NOT EXISTS hive_test
   COMMENT 'hive database for test'
   WITH DBPROPERTIES ('create'='lwPigKing');
```

1.4 查看数据库信息

语法：

```
DESC DATABASE [EXTENDED] db_name; --EXTENDED 表示是否显示额外属性
```

示例：

```
DESC DATABASE   EXTENDED hive_test;
```

1.5 删除数据库

语法：

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];
```

- 默认行为是 RESTRICT，如果数据库中存在表则删除失败。要想删除库及其中的表，可以使用 CASCADE 级联删除。

示例：

```
    DROP DATABASE IF EXISTS hive_test CASCADE;
```

创建表

2.1 建表语法

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name       --表名
   [(col_name data_type [COMMENT col_comment],
     ... [constraint_specification])]   --列名 列数据类型
   [COMMENT table_comment]    --表描述
   [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]   --分区表分区规则
   [
     CLUSTERED BY (col_name, col_name, ...)
    [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS
   ]   --分桶表分桶规则
   [SKEWED BY (col_name, col_name, ...) ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)
    [STORED AS DIRECTORIES]
   ]   --指定倾斜列和值
   [
    [ROW FORMAT row_format]
    [STORED AS file_format]
       | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]
   ]   -- 指定行分隔符、存储文件格式或采用自定义存储格式
   [LOCATION hdfs_path]   -- 指定表的存储位置
   [TBLPROPERTIES (property_name=property_value, ...)]   --指定表的属性
   [AS select_statement];    --从查询结果创建表
```

2.2 内部表

```
CREATE TABLE emp(
  empno INT,
  ename STRING,
  job STRING,
  mgr INT,
  hiredate TIMESTAMP,
  sal DECIMAL(7,2),
  comm DECIMAL(7,2),
  deptno INT)
   ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";
```

2.3 外部表

```
CREATE EXTERNAL TABLE emp_external(
  empno INT,
  ename STRING,
  job STRING,
  mgr INT,
  hiredate TIMESTAMP,
  sal DECIMAL(7,2),
  comm DECIMAL(7,2),
  deptno INT)
   ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
  LOCATION '/hive/emp_external';
```

使用 `desc format  emp_external` 命令可以查看表的详细信息如下:

2.4 分区表

```
CREATE EXTERNAL TABLE emp_partition(
  empno INT,
  ename STRING,
  job STRING,
  mgr INT,
  hiredate TIMESTAMP,
  sal DECIMAL(7,2),
  comm DECIMAL(7,2)
   )
  PARTITIONED BY (deptno INT)    -- 按照部门编号进行分区
   ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
  LOCATION '/hive/emp_partition';
```

2.5 分桶表

```
CREATE EXTERNAL TABLE emp_bucket(
  empno INT,
  ename STRING,
  job STRING,
  mgr INT,
  hiredate TIMESTAMP,
  sal DECIMAL(7,2),
  comm DECIMAL(7,2),
  deptno INT)
  CLUSTERED BY(empno) SORTED BY(empno ASC) INTO 4 BUCKETS   --按照员工编号散列到四个 bucket 中
   ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
  LOCATION '/hive/emp_bucket';
```

2.6 倾斜表

通过指定一个或者多个列经常出现的值（严重偏斜），Hive 会自动将涉及到这些值的数据拆分为单独的文件。在查询时，如果涉及到倾斜值，它就直接从独立文件中获取数据，而不是扫描所有文件，这使得性能得到提升。

```
CREATE EXTERNAL TABLE emp_skewed(
  empno INT,
  ename STRING,
  job STRING,
  mgr INT,
  hiredate TIMESTAMP,
  sal DECIMAL(7,2),
  comm DECIMAL(7,2)
   )
  SKEWED BY (empno) ON (66,88,100)   --指定 empno 的倾斜值 66,88,100
   ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
  LOCATION '/hive/emp_skewed';
```

2.7 临时表

临时表仅对当前 session 可见，临时表的数据将存储在用户的暂存目录中，并在会话结束后删除。如果临时表与永久表表名相同，则对该表名的任何引用都将解析为临时表，而不是永久表。临时表还具有以下两个限制：

- 不支持分区列；
- 不支持创建索引。

```
CREATE TEMPORARY TABLE emp_temp(
  empno INT,
  ename STRING,
  job STRING,
  mgr INT,
  hiredate TIMESTAMP,
  sal DECIMAL(7,2),
  comm DECIMAL(7,2)
   )
   ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";
```

2.8 CTAS创建表

支持从查询语句的结果创建表：

```
CREATE TABLE emp_copy AS SELECT * FROM emp WHERE deptno='20';
```

2.9 复制表结构

语法：

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name   --创建表表名
    LIKE existing_table_or_view_name   --被复制表的表名
    [LOCATION hdfs_path]; --存储位置
```

示例：

```
CREATE TEMPORARY EXTERNAL TABLE   IF NOT EXISTS  emp_co   LIKE emp
```

2.10 加载数据到表

加载数据到表中属于 DML 操作，这里为了方便大家测试，先简单介绍一下加载本地数据到表中：

```
-- 加载数据到 emp 表中
load data local inpath "/usr/file/emp.txt" into table emp;
```

```
 7369   SMITH   CLERK   7902    1980-12-17 00:00:00 800.00      20
 7499   ALLEN   SALESMAN    7698    1981-02-20 00:00:00 1600.00 300.00  30
 7521   WARD    SALESMAN    7698    1981-02-22 00:00:00 1250.00 500.00  30
 7566   JONES   MANAGER 7839    1981-04-02 00:00:00 2975.00      20
 7654   MARTIN  SALESMAN    7698    1981-09-28 00:00:00 1250.00 1400.00 30
 7698   BLAKE   MANAGER 7839    1981-05-01 00:00:00 2850.00      30
 7782   CLARK   MANAGER 7839    1981-06-09 00:00:00 2450.00      10
 7788   SCOTT   ANALYST 7566    1987-04-19 00:00:00 1500.00      20
 7839   KING    PRESIDENT       1981-11-17 00:00:00 5000.00      10
 7844   TURNER  SALESMAN    7698    1981-09-08 00:00:00 1500.00 0.00    30
 7876   ADAMS   CLERK   7788    1987-05-23 00:00:00 1100.00      20
 7900   JAMES   CLERK   7698    1981-12-03 00:00:00 950.00       30
 7902   FORD    ANALYST 7566    1981-12-03 00:00:00 3000.00      20
 7934   MILLER  CLERK   7782    1982-01-23 00:00:00 1300.00      10
```

加载后可查询表中数据：

修改表

3.1 重命名表

语法：

```
ALTER TABLE table_name RENAME TO new_table_name;
```

示例：

```
ALTER TABLE emp_temp RENAME TO new_emp; --把 emp_temp 表重命名为 new_emp
```

3.2 修改列

语法：

```
ALTER TABLE table_name [PARTITION partition_spec] CHANGE [COLUMN] col_old_name col_new_name
column_type
   [COMMENT col_comment] [FIRST|AFTER column_name] [CASCADE|RESTRICT];
```

示例：

```
-- 修改字段名和类型
ALTER TABLE emp_temp CHANGE empno empno_new INT;

-- 修改字段 sal 的名称 并将其放置到 empno 字段后
ALTER TABLE emp_temp CHANGE sal sal_new decimal(7,2)   AFTER ename;

-- 为字段增加注释
ALTER TABLE emp_temp CHANGE mgr mgr_new INT COMMENT 'this is column mgr';
```

3.3 新增列

示例：

```
ALTER TABLE emp_temp ADD COLUMNS (address STRING COMMENT 'home address');=
```

清空表/删除表

4.1 清空表

语法：

```
-- 清空整个表或表指定分区中的数据
TRUNCATE TABLE table_name [PARTITION (partition_column = partition_col_value,  ...)];
```

- 目前只有内部表才能执行 TRUNCATE 操作，外部表执行时会抛出异常 `Cannot truncate non-managed table XXXX`。

示例：

```
TRUNCATE TABLE emp_mgt_ptn PARTITION (deptno=20);
```

4.2 删除表

语法：

```
DROP TABLE [IF EXISTS] table_name [PURGE];
```

- 内部表：不仅会删除表的元数据，同时会删除 HDFS 上的数据；
- 外部表：只会删除表的元数据，不会删除 HDFS 上的数据；
- 删除视图引用的表时，不会给出警告（但视图已经无效了，必须由用户删除或重新创建）。

其他命令

5.1 Describe

查看数据库：

```
DESCRIBE|Desc DATABASE [EXTENDED] db_name;   --EXTENDED 是否显示额外属性
```

查看表：

```
DESCRIBE|Desc [EXTENDED|FORMATTED] table_name --FORMATTED 以友好的展现方式查看表详情
```

5.2 Show

1. 查看数据库列表

```
-- 语法
SHOW (DATABASES|SCHEMAS) [LIKE 'identifier_with_wildcards'];

-- 示例：
SHOW DATABASES like 'hive*';
```

LIKE 子句允许使用正则表达式进行过滤，但是 SHOW 语句当中的 LIKE 子句只支持 `*`（通配符）和 `|`（条件或）两个符号。例如 `employees`，`emp *`，`emp * | * ees`，所有这些都将匹配名为 `employees` 的数据库。

2. 查看表的列表

```
-- 语法
SHOW TABLES [IN database_name] ['identifier_with_wildcards'];

-- 示例
SHOW TABLES IN default;
```

3. 查看视图列表

```
SHOW VIEWS [IN/FROM database_name] [LIKE 'pattern_with_wildcards'];    --仅支持 Hive 2.2.0 +
```

4. 查看表的分区列表

```
SHOW PARTITIONS table_name;
```

5. 查看表/视图的创建语句

```
SHOW CREATE TABLE ([db_name.]table_name|view_name);
```

第四章 Hive数据查询详解

数据准备

为了演示查询操作，这里需要预先创建三张表，并加载测试数据。

1.1 员工表

```
-- 建表语句
CREATE TABLE emp(
      empno INT,        -- 员工表编号
      ename STRING,    -- 员工姓名
```

```
        job STRING,       -- 职位类型
        mgr INT,
        hiredate TIMESTAMP,    --雇佣日期
        sal DECIMAL(7,2),    --工资
        comm DECIMAL(7,2),
        deptno INT)     --部门编号
       ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";

   --加载数据
LOAD DATA LOCAL INPATH "/usr/file/emp.txt" OVERWRITE INTO TABLE emp;
```

1.2 部门表

```
-- 建表语句
CREATE TABLE dept(
      deptno INT,     --部门编号
      dname STRING,    --部门名称
      loc STRING      --部门所在的城市
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";

--加载数据
LOAD DATA LOCAL INPATH "/usr/file/dept.txt" OVERWRITE INTO TABLE dept;
```

1.3 分区表

这里需要额外创建一张分区表，主要是为了演示分区查询：

```
CREATE EXTERNAL TABLE emp_ptn(
        empno INT,
        ename STRING,
        job STRING,
        mgr INT,
        hiredate TIMESTAMP,
        sal DECIMAL(7,2),
        comm DECIMAL(7,2)
    )
 PARTITIONED BY (deptno INT)    -- 按照部门编号进行分区
 ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t";


--加载数据
LOAD DATA LOCAL INPATH "/usr/file/emp.txt" OVERWRITE INTO TABLE emp_ptn PARTITION (deptno=20)
LOAD DATA LOCAL INPATH "/usr/file/emp.txt" OVERWRITE INTO TABLE emp_ptn PARTITION (deptno=30)
LOAD DATA LOCAL INPATH "/usr/file/emp.txt" OVERWRITE INTO TABLE emp_ptn PARTITION (deptno=40)
LOAD DATA LOCAL INPATH "/usr/file/emp.txt" OVERWRITE INTO TABLE emp_ptn PARTITION (deptno=50)
```

单表查询

2.1 SELECT

```
-- 查询表中全部数据
SELECT * FROM emp;
```

2.2 WHERE

```
-- 查询 10 号部门中员工编号大于 7782 的员工信息
SELECT * FROM emp WHERE empno > 7782 AND deptno = 10;
```

2.3 DISTINCT

Hive 支持使用 DISTINCT 关键字去重。

```
-- 查询所有工作类型
SELECT DISTINCT job FROM emp;
```

2.4 分区查询

分区查询 (Partition Based Queries)，可以指定某个分区或者分区范围。

```
-- 查询分区表中部门编号在[20,40]之间的员工
SELECT emp_ptn.* FROM emp_ptn
WHERE emp_ptn.deptno >= 20 AND emp_ptn.deptno <= 40;
```

2.5 LIMIT

```
-- 查询薪资最高的 5 名员工
SELECT * FROM emp ORDER BY sal DESC LIMIT 5;
```

2.6 GROUP BY

Hive 支持使用 GROUP BY 进行分组聚合操作。

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

-- 查询各个部门薪酬综合
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno;
```

`hive.map.aggr` 控制程序如何进行聚合。默认值为 false。如果设置为 true，Hive 会在 map 阶段就执行一次聚合。这可以提高聚合效率，但需要消耗更多内存。

2.7 ORDER AND SORT

可以使用 ORDER BY 或者 Sort BY 对查询结果进行排序，排序字段可以是整型也可以是字符串：如果是整型，则按照大小排序；如果是字符串，则按照字典序排序。ORDER BY 和 SORT BY 的区别如下：

- 使用 ORDER BY 时会有一个 Reducer 对全部查询结果进行排序，可以保证数据的全局有序性；
- 使用 SORT BY 时只会在每个 Reducer 中进行排序，这可以保证每个 Reducer 的输出数据是有序的，但不能保证全局有序。

由于 ORDER BY 的时间可能很长，如果你设置了严格模式 (hive.mapred.mode = strict)，则其后面必须再跟一个 `limit` 子句。

> 注： hive.mapred.mode 默认值是 nonstrict ， 也就是非严格模式。

```
-- 查询员工工资，结果按照部门升序，按照工资降序排列
SELECT empno, deptno, sal FROM emp ORDER BY deptno ASC, sal DESC;
```

2.8 HAVING

可以使用 HAVING 对分组数据进行过滤。

```
-- 查询工资总和大于 9000 的所有部门
SELECT deptno,SUM(sal) FROM emp GROUP BY deptno HAVING SUM(sal)>9000;
```

2.9 DISTRIBUTE BY

默认情况下，MapReduce 程序会对 Map 输出结果的 Key 值进行散列，并均匀分发到所有 Reducer 上。如果想要把具有相同 Key 值的数据分发到同一个 Reducer 进行处理，这就需要使用 DISTRIBUTE BY 字句。

需要注意的是，DISTRIBUTE BY 虽然能保证具有相同 Key 值的数据分发到同一个 Reducer，但是不能保证数据在 Reducer 上是有序的。情况如下：

把以下 5 个数据发送到两个 Reducer 上进行处理：

```
k1
k2
k4
k3
k1
```

Reducer1 得到如下乱序数据：

```
k1
k2
k1
```

Reducer2 得到数据如下：

```
k4
k3
```

如果想让 Reducer 上的数据时有序的，可以结合 `SORT BY` 使用 (示例如下)，或者使用下面我们将要介绍的 CLUSTER BY。

```
-- 将数据按照部门分发到对应的 Reducer 上处理
SELECT empno, deptno, sal FROM emp DISTRIBUTE BY deptno SORT BY deptno ASC;
```

2.10 CLUSTER BY

如果 `SORT BY` 和 `DISTRIBUTE BY` 指定的是相同字段，且 SORT BY 排序规则是 ASC，此时可以使用 `CLUSTER BY` 进行替换，同时 `CLUSTER BY` 可以保证数据在全局是有序的。

```
SELECT empno, deptno, sal FROM emp CLUSTER   BY deptno ;
```

多表联结查询

Hive 支持内连接，外连接，左外连接，右外连接，笛卡尔连接，这和传统数据库中的概念是一致的，可以参见下图。

需要特别强调：JOIN 语句的关联条件必须用 ON 指定，不能用 WHERE 指定，否则就会先做笛卡尔积，再过滤，这会导致你得不到预期的结果 (下面的演示会有说明)。

3.1 INNER JOIN

```
-- 查询员工编号为 7369 的员工的详细信息
SELECT e.*,d.* FROM
emp e JOIN dept d
ON e.deptno = d.deptno
WHERE empno=7369;

--如果是三表或者更多表连接，语法如下
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)
```

3.2 LEFT OUTER JOIN

LEFT OUTER JOIN 和 LEFT JOIN 是等价的。

```
-- 左连接
SELECT e.*,d.*
FROM emp e LEFT OUTER  JOIN  dept d
ON e.deptno = d.deptno;
```

3.3 RIGHT OUTER JOIN

```
--右连接
SELECT e.*,d.*
FROM emp e RIGHT OUTER JOIN  dept d
ON e.deptno = d.deptno;
```

执行右连接后，由于 40 号部门下没有任何员工，所以此时员工信息为 NULL。这个查询可以很好的复述上面提到的——JOIN 语句的关联条件必须用 ON 指定，不能用 WHERE 指定。你可以把 ON 改成 WHERE，你会发现无论如何都查不出 40 号部门这条数据，因为笛卡尔运算不会有 (NULL, 40) 这种情况。

3.4 FULL OUTER JOIN

```
SELECT e.*,d.*
FROM emp e FULL OUTER JOIN  dept d
ON e.deptno = d.deptno;
```

3.5 LEFT SEMI JOIN

LEFT SEMI JOIN （左半连接）是 IN/EXISTS 子查询的一种更高效的实现。

- JOIN 子句中右边的表只能在 ON 子句中设置过滤条件;
- 查询结果只包含左边表的数据，所以只能 SELECT 左表中的列。

```
-- 查询在纽约办公的所有员工信息
SELECT emp.*
FROM emp LEFT SEMI JOIN dept
ON emp.deptno = dept.deptno AND dept.loc="NEW YORK";

--上面的语句就等价于
SELECT emp.* FROM emp
WHERE emp.deptno IN (SELECT deptno FROM dept WHERE loc="NEW YORK");
```

3.6 JOIN

笛卡尔积连接，这个连接日常的开发中可能很少遇到，且性能消耗比较大，基于这个原因，如果在严格模式下 (hive.mapred.mode = strict)，Hive 会阻止用户执行此操作。

```
SELECT * FROM emp JOIN dept;
```

JOIN优化

4.1 STREAMTABLE

在多表进行联结的时候，如果每个 ON 字句都使用到共同的列（如下面的 `b.key` ），此时 Hive 会进行优化，将多表 JOIN 在同一个 map / reduce 作业上进行。同时假定查询的最后一个表（如下面的 c 表）是最大的一个表，在对每行记录进行 JOIN 操作时，它将尝试将其他的表缓存起来，然后扫描最后那个表进行计算。因此用户需要保证查询的表的大小从左到右是依次增加的。

```
`SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key) JOIN c ON (c.key = b.key)`
```

然后，用户并非需要总是把最大的表放在查询语句的最后面，Hive 提供了 `/*+ STREAMTABLE() */` 标志，用于标识最大的表，示例如下：

```
SELECT /*+ STREAMTABLE(d) */  e.*,d.*
FROM emp e JOIN dept d
ON e.deptno = d.deptno
WHERE job='CLERK';
```

4.2 MAPJOIN

如果所有表中只有一张表是小表，那么 Hive 把这张小表加载到内存中。这时候程序会在 map 阶段直接拿另外一个表的数据和内存中表数据做匹配，由于在 map 就进行了 JOIN 操作，从而可以省略 reduce 过程，这样效率可以提升很多。Hive 中提供了 `/*+ MAPJOIN() */` 来标记小表，示例如下：

```
SELECT /*+ MAPJOIN(d) */ e.*,d.*
FROM emp e JOIN dept d
ON e.deptno = d.deptno
WHERE job='CLERK';
```

SELECT的其他用途

查看当前数据库：

```
SELECT current_database()
```

本地模式

在上面演示的语句中，大多数都会触发 MapReduce, 少部分不会触发，比如 `select * from emp limit 5` 就不会触发 MR，此时 Hive 只是简单的读取数据文件中的内容，然后格式化后进行输出。在需要执行 MapReduce 的查询中，你会发现执行时间可能会很长，这时候你可以选择开启本地模式。

```
--本地模式默认关闭，需要手动开启此功能
SET hive.exec.mode.local.auto=true;
```

启用后，Hive 将分析查询中每个 map-reduce 作业的大小，如果满足以下条件，则可以在本地运行它：

- 作业的总输入大小低于：hive.exec.mode.local.auto.inputbytes.max（默认为 128MB）；
- map-tasks 的总数小于：hive.exec.mode.local.auto.tasks.max（默认为 4）；

- 所需的 reduce 任务总数为 1 或 0。

因为我们测试的数据集很小，所以你再次去执行上面涉及 MR 操作的查询，你会发现速度会有显著的提升。

第五章 Hive分区表和分桶表

分区表

1.1 概念

Hive 中的表对应为 HDFS 上的指定目录，在查询数据时候，默认会对全表进行扫描，这样时间和性能的消耗都非常大。

分区为 HDFS 上表目录的子目录，数据按照分区存储在子目录中。如果查询的 `where` 字句的中包含分区条件，则直接从该分区去查找，而不是扫描整个表目录，合理的分区设计可以极大提高查询速度和性能。

> 这里说明一下分区表并 Hive 独有的概念，实际上这个概念非常常见。比如在我们常用的 Oracle 数据库中，当表中的数据量不断增大，查询数据的速度就会下降，这时也可以对表进行分区。表进行分区后，逻辑上表仍然是一张完整的表，只是将表中的数据存放到多个表空间（物理文件上），这样查询数据时，就不必要每次都扫描整张表，从而提升查询性能。

1.2 使用场景

通常，在管理大规模数据集的时候都需要进行分区，比如将日志文件按天进行分区，从而保证数据细粒度的划分，使得查询性能得到提升。

1.3 创建分区表

在 Hive 中可以使用 `PARTITIONED BY` 子句创建分区表。表可以包含一个或多个分区列，程序会为分区列中的每个不同值组合创建单独的数据目录。下面的我们创建一张雇员表作为测试：

```
CREATE EXTERNAL TABLE emp_partition(
    empno INT,
    ename STRING,
    job STRING,
    mgr INT,
    hiredate TIMESTAMP,
    sal DECIMAL(7,2),
    comm DECIMAL(7,2)
    )
    PARTITIONED BY (deptno INT)    -- 按照部门编号进行分区
    ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
    LOCATION '/hive/emp_partition';
```

1.4 加载数据到分区表

加载数据到分区表时候必须要指定数据所处的分区：

```
# 加载部门编号为20的数据到表中
LOAD DATA LOCAL INPATH "/usr/file/emp20.txt" OVERWRITE INTO TABLE emp_partition PARTITION (deptno=20)
# 加载部门编号为30的数据到表中
LOAD DATA LOCAL INPATH "/usr/file/emp30.txt" OVERWRITE INTO TABLE emp_partition PARTITION (deptno=30)
```

1.5 查看分区目录

这时候我们直接查看表目录，可以看到表目录下存在两个子目录，分别是 `deptno=20` 和 `deptno=30` ,这就是分区目录，分区目录下才是我们加载的数据文件。

```
# hadoop fs -ls  hdfs://hadoop001:8020/hive/emp_partition/
```

这时候当你的查询语句的 `where` 包含 `deptno=20`，则就去对应的分区目录下进行查找，而不用扫描全表。

分桶表

1.1 简介

分区提供了一个隔离数据和优化查询的可行方案，但是并非所有的数据集都可以形成合理的分区，分区的数量也不是越多越好，过多的分区条件可能会导致很多分区上没有数据。同时 Hive 会限制动态分区可以创建的最大分区数，用来避免过多分区文件对文件系统产生负担。鉴于以上原因，Hive 还提供了一种更加细粒度的数据拆分方案：分桶表 (bucket Table)。

分桶表会将指定列的值进行哈希散列，并对 bucket（桶数量）取余，然后存储到对应的 bucket（桶）中。

1.2 理解分桶表

单从概念上理解分桶表可能会比较晦涩，其实和分区一样，分桶这个概念同样不是 Hive 独有的，对于 Java 开发人员而言，这可能是一个每天都会用到的概念，因为 Hive 中的分桶概念和 Java 数据结构中的 HashMap 的分桶概念是一致的。

当调用 HashMap 的 put() 方法存储数据时，程序会先对 key 值调用 hashCode() 方法计算出 hashcode，然后对数组长度取模计算出 index，最后将数据存储在数组 index 位置的链表上，链表达到一定阈值后会转换为红黑树 (JDK1.8+)。下图为 HashMap 的数据结构图：

1.3 创建分桶表

在 Hive 中，我们可以通过 `CLUSTERED BY` 指定分桶列，并通过 `SORTED BY` 指定桶中数据的排序参考列。下面为分桶表建表语句示例：

```
    CREATE EXTERNAL TABLE emp_bucket(
      empno INT,
      ename STRING,
      job STRING,
      mgr INT,
      hiredate TIMESTAMP,
      sal DECIMAL(7,2),
      comm DECIMAL(7,2),
      deptno INT)
      CLUSTERED BY(empno) SORTED BY(empno ASC) INTO 4 BUCKETS   --按照员工编号散列到四个 bucket 中
       ROW FORMAT DELIMITED FIELDS TERMINATED BY "\t"
      LOCATION '/hive/emp_bucket';
```

1.4 加载数据到分桶表

这里直接使用 `Load` 语句向分桶表加载数据，数据时可以加载成功的，但是数据并不会分桶。

这是由于分桶的实质是对指定字段做了 hash 散列然后存放到对应文件中，这意味着向分桶表中插入数据是必然要通过 MapReduce，且 Reducer 的数量必须等于分桶的数量。由于以上原因，分桶表的数据通常只能使用 CTAS(CREATE TABLE AS SELECT) 方式插入，因为 CTAS 操作会触发 MapReduce。加载数据步骤如下：

1. 设置强制分桶

```
set hive.enforce.bucketing = true; --Hive 2.x 不需要这一步
```

在 Hive 0.x and 1.x 版本，必须使用设置 `hive.enforce.bucketing = true`，表示强制分桶，允许程序根据表结构自动选择正确数量的 Reducer 和 cluster by column 来进行分桶。

2. CTAS导入数据

```
INSERT INTO TABLE emp_bucket SELECT *  FROM emp;   --这里的 emp 表就是一张普通的雇员表
```

分区表和分桶表结合使用

分区表和分桶表的本质都是将数据按照不同粒度进行拆分，从而使得在查询时候不必扫描全表，只需要扫描对应的分区或分桶，从而提升查询效率。两者可以结合起来使用，从而保证表数据在不同粒度上都能得到合理的拆分。下面是 Hive 官方给出的示例：

```
 CREATE TABLE page_view_bucketed(
     viewTime INT,
      userid BIGINT,
      page_url STRING,
      referrer_url STRING,
      ip STRING )
   PARTITIONED BY(dt STRING)
   CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
   ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '\001'
      COLLECTION ITEMS TERMINATED BY '\002'
      MAP KEYS TERMINATED BY '\003'
   STORED AS SEQUENCEFILE;
```

此时导入数据时需要指定分区：

```
INSERT OVERWRITE page_view_bucketed
PARTITION (dt='2009-02-25')
SELECT * FROM page_view WHERE dt='2009-02-25';
```

第六章 Hive视图和索引

视图

1.1 简介

Hive 中的视图和 RDBMS 中视图的概念一致，都是一组数据的逻辑表示，本质上就是一条 SELECT 语句的结果集。视图是纯粹的逻辑对象，没有关联的存储 (Hive 3.0.0 引入的物化视图除外)，当查询引用视图时，Hive 可以将视图的定义与查询结合起来，例如将查询中的过滤器推送到视图中。

1.2 创建视图

```
CREATE VIEW [IF NOT EXISTS] [db_name.]view_name    -- 视图名称
   [(column_name [COMMENT column_comment], ...) ]      --列名
   [COMMENT view_comment]   --视图注释
   [TBLPROPERTIES (property_name = property_value, ...)]   --额外信息
   AS SELECT ...;
```

在 Hive 中可以使用 `CREATE VIEW` 创建视图，如果已存在具有相同名称的表或视图，则会抛出异常，建议使用 `IF NOT EXISTS` 预做判断。在使用视图时候需要注意以下事项：

- 视图是只读的，不能用作 LOAD / INSERT / ALTER 的目标；
- 在创建视图时候视图就已经固定，对基表的后续更改（如添加列）将不会反映在视图；
- 删除基表并不会删除视图，需要手动删除视图；
- 视图可能包含 ORDER BY 和 LIMIT 子句。如果引用视图的查询语句也包含这类子句，其执行优先级低于视图对应字句。例如，视图 `custom_view` 指定 LIMIT 5，查询语句为 `select * from custom_view  LIMIT 10`，此时结果最多返回 5 行。
- 创建视图时，如果未提供列名，则将从 SELECT 语句中自动派生列名；
- 创建视图时，如果 SELECT 语句中包含其他表达式，例如 x + y，则列名称将以_C0，_C1 等形式生成；

```
CREATE VIEW   IF NOT EXISTS custom_view AS SELECT empno, empno+deptno , 1+2 FROM emp;
```

1.3 查看视图

```
-- 查看所有视图： 没有单独查看视图列表的语句，只能使用 show tables
show tables;
-- 查看某个视图
desc view_name;
-- 查看某个视图详细信息
desc formatted view_name;
```

1.4 删除视图

```
DROP VIEW [IF EXISTS] [db_name.]view_name;
```

删除视图时，如果被删除的视图被其他视图所引用，这时候程序不会发出警告，但是引用该视图其他视图已经失效，需要进行重建或者删除。

1.5 修改视图

```
ALTER VIEW [db_name.]view_name AS select_statement;
```

被更改的视图必须存在，且视图不能具有分区，如果视图具有分区，则修改失败。

1.6 修改视图属性

语法：

```
ALTER VIEW [db_name.]view_name SET TBLPROPERTIES table_properties;

table_properties:
   : (property_name = property_value, property_name = property_value, ...)
```

示例：

```
ALTER VIEW custom_view SET TBLPROPERTIES ('create'='heibaiying','date'='2019-05-05');
```

索引

2.1 简介

Hive 在 0.7.0 引入了索引的功能，索引的设计目标是提高表某些列的查询速度。如果没有索引，带有谓词的查询（如'WHERE table1.column = 10'）会加载整个表或分区并处理所有行。但是如果 column 存在索引，则只需要加载和处理文件的一部分。

2.2 索引原理

在指定列上建立索引，会产生一张索引表（表结构如下），里面的字段包括：索引列的值、该值对应的 HDFS 文件路径、该值在文件中的偏移量。在查询涉及到索引字段时，首先到索引表查找索引列值对应的 HDFS 文件路径及偏移量，这样就避免了全表扫描。

```
+--------------+----------------+----------+--+
|   col_name   |   data_type    | comment  |
+--------------+----------------+----------+--+
| empno        | int            | 建立索引的列 |
| _bucketname  | string         | HDFS 文件路径 |
| _offsets     | array<bigint>  | 偏移量      |
+--------------+----------------+----------+--+
```

2.3 创建索引

```
CREATE INDEX index_name       --索引名称
   ON TABLE base_table_name (col_name, ...)   --建立索引的列
   AS index_type      --索引类型
   [WITH DEFERRED REBUILD]      --重建索引
   [IDXPROPERTIES (property_name=property_value, ...)]   --索引额外属性
   [IN TABLE index_table_name]      --索引表的名字
   [
       [ ROW FORMAT ...] STORED AS ...
       | STORED BY ...
   ]    --索引表行分隔符 、 存储格式
   [LOCATION hdfs_path]   --索引表存储位置
   [TBLPROPERTIES (...)]    --索引表表属性
   [COMMENT "index comment"];   --索引注释
```

2.4 查看索引

```
--显示表上所有列的索引
SHOW FORMATTED INDEX ON table_name;
```

2.4 删除索引

删除索引会删除对应的索引表。

```
DROP INDEX [IF EXISTS] index_name ON table_name;
```

如果存在索引的表被删除了，其对应的索引和索引表都会被删除。如果被索引表的某个分区被删除了，那么分区对应的分区索引也会被删除。

2.5 重建索引

```
ALTER INDEX index_name ON table_name [PARTITION partition_spec] REBUILD;
```

重建索引。如果指定了 PARTITION，则仅重建该分区的索引。

索引案例

3.1 创建索引

在 emp 表上针对 `empno` 字段创建名为 `emp_index`,索引数据存储在 `emp_index_table` 索引表中

```
create index emp_index on table emp(empno) as
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'
with deferred rebuild
in table emp_index_table ;
```

此时索引表中是没有数据的，需要重建索引才会有索引的数据。

3.2 重建索引

```
alter index emp_index on emp rebuild;
```

Hive 会启动 MapReduce 作业去建立索引，建立好后查看索引表数据如下。三个表字段分别代表：索引列的值、该值对应的 HDFS 文件路径、该值在文件中的偏移量。

3.3 自动使用索引

默认情况下，虽然建立了索引，但是 Hive 在查询时候是不会自动去使用索引的，需要开启相关配置。开启配置后，涉及到索引列的查询就会使用索引功能去优化查询。

```
SET hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
SET hive.optimize.index.filter=true;
SET hive.optimize.index.filter.compact.minsize=0;
```

3.4 查看索引

```
SHOW INDEX ON emp;
```

索引的缺陷

索引表最主要的一个缺陷在于：索引表无法自动 rebuild，这也就意味着如果表中有数据新增或删除，则必须手动 rebuild，重新执行 MapReduce 作业，生成索引表数据。

同时按照 **官方文档** 的说明，Hive 会从 3.0 开始移除索引功能，主要基于以下两个原因：

- 具有自动重写的物化视图 (Materialized View) 可以产生与索引相似的效果（Hive 2.3.0 增加了对物化视图的支持，在 3.0 之后正式引入）。
- 使用列式存储文件格式（Parquet，ORC）进行存储时，这些格式支持选择性扫描，可以跳过不需要的文件或块。

第七章 Hive窗口函数

个人对over()的窗口理解：这个永远是`一行对应一个窗口`，至于这个窗口的范围是什么就要看over()函数里面对窗口范围的约束是什么了（partition by order by between ... and）通过`partition by关键字来对窗口分组`，特殊注意：通过order by 来对order by字段排序后的行进行开窗，只不过注意的是第一行数据的窗口大小是1，第二行数据的窗口范围是前2行，第n行的窗口范围是前n行，以此类推。如果里面没有条件，则每一行对应整张表。

特殊的窗口函数如rank(),rownumber(),dense()等，即使后面over()里面没有条件，默认的开窗类似order by效果，即第一行窗口大小为1，第二行窗口大小为2，以此类推，但是数据只不过没有什么统计意义，所以一般还是会在over()里加入partiton by和order by（分组，排序）等，为其赋予意义，如排名等。

over(partition by) 和 普通的group by的区别，为什么不同group by，因为有group by，只能select group by 后面的字段，和一些聚合函数 sum(),avg(),max(),min()等，而用了over(partition by)，还能select 别的非 partition by 字段 或者能直接“select *”，而且对于join 等有更好的支持。

窗口函数语法

avg()、sum()、max()、min()是分析函数，而over()才是窗口函数，下面我们来看看over()窗口函数的语法结构、及常与over()一起使用的分析函数

- over()窗口函数的语法结构
- 常与over()一起使用的分析函数
- 窗口函数总结

over()窗口函数的语法结构

```
分析函数 over(partition by 列名 order by 列名 rows between 开始位置 and 结束位置)
```

over()函数中包括三个函数：包括分区`partition by 列名`、排序`order by 列名`、指定窗口范围`rows between 开始位置 and 结束位置`。我们在使用over()窗口函数时，over()函数中的这三个函数可组合使用也可以不使用。

over()函数中如果不使用这三个函数，窗口大小是针对查询产生的所有数据，如果指定了分区，窗口大小是针对每个分区的数据。

over() 默认此时每一行的窗口都是所有的行

```
select *,count(1) over() from business;
```

over(order by orderdate)

orderdate=1的窗口只有一行，orderdate=2的窗口包括orderdate=2017-01-01,orderdate=2017-01-02

```
select *,count(1) over(order by orderdate) from business;
```

over(partition by name)每一行根据 name来区分窗口

```
select *,sum(cost) over(partition by name) from business;
```

over(partition by name order by id) 每一行根据name来区分窗口,再根据order by取具体的范围

```
select *,sum(cost) over(partition by name order by orderdate) from business;
```

over()函数中的三个函数讲解

A、 partition by
`partition by`可理解为group by 分组。`over(partition by 列名)`搭配分析函数时，分析函数按照每一组每一组的数据进行计算的。

B、 rows between 开始位置 and 结束位置
是指定窗口范围，比如第一行到当前行。而这个范围是随着数据变化的。`over(rows between 开始位置 and 结束位置)`搭配分析函数时，分析函数按照这个范围进行计算的。

窗口范围说明：
我们常使用的窗口范围是`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW（表示从起点到当前行）`，常用该窗口来计算累加。

```
PRECEDING：往前
```

```
FOLLOWING：往后
CURRENT ROW：当前行
UNBOUNDED：起点（一般结合PRECEDING，FOLLOWING使用）
UNBOUNDED PRECEDING 表示该窗口最前面的行（起点）
UNBOUNDED FOLLOWING：表示该窗口最后面的行（终点）
```

```
比如说：
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW（表示从起点到当前行）
ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING（表示往前2行到往后1行）
ROWS BETWEEN 2 PRECEDING AND 1 CURRENT ROW（表示往前2行到当前行）
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING（表示当前行到终点）
```

常与over()一起使用的分析函数

聚合类

```
avg()、sum()、max()、min()
```

排名类

```
row_number()按照值排序时产生一个自增编号，不会重复（如：1、2、3、4、5、6）
rank() 按照值排序时产生一个自增编号，值相等时会重复，会产生空位（如：1、2、3、3、3、6）
dense_rank() 按照值排序时产生一个自增编号，值相等时会重复，不会产生空位（如：1、2、3、3、3、4）
```

其他类

```
lag(列名,往前的行数,[行数为null时的默认值，不指定为null])，可以计算用户上次购买时间，或者用户下次购买时间。或者上次登录时间和下次登录时间
lead(列名,往后的行数,[行数为null时的默认值，不指定为null])
ntile(n) 把有序分区中的行分发到指定数据的组中，各个组有编号，编号从1开始，对于每一行，ntile返回此行所属的组的编号
```

练习题

```
 测试数据

 20191020,11111,85
 20191020,22222,83
 20191020,33333,86
 20191021,11111,87
 20191021,22222,65
 20191021,33333,98
 20191022,11111,67
 20191022,22222,34
 20191022,33333,88
 20191023,11111,99
 20191023,22222,33
```

```
create table test_window
(logday string,     #logday时间
userid string,
score int)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

#加载数据
load data local inpath '/home/xiaowangzi/hive_test_data/test_window.txt' into table test_window;
```

使用 over() 函数进行数据统计，统计每个用户及表中数据的总数

```
select *, count(userid) over() as total  from  test_window;
```

这里使用 over() 与 select count(*) 有相同的作用，好处就是，在需要计算总数时不用再进行一次关联。

求用户明细并统计每天的用户总数

可以使用 partition by 按日期列对数据进行分区处理，如：over(partition by logday)

```
select  *,count() over(partition by logday) as day_total from  test_window;
```

求每天的用户数可以使用 `select logday, count(userid) from recommend.test_window group by logday`，但是当想要得到 userid 信息时，这种用法的优势就很明显。

计算从第一天到现在的所有 score 大于80分的用户总数

此时简单的分区不能满足需求，需要将 order by 和 窗口定义结合使用。

```
select *,count() over(order by logday rows between unbounded preceding and current row)as total
from test_window where score > 80;
```

计算每个用户到当前日期分数大于80的天数

```
select *, count() over(partition by userid order by logday rows between unbounded preceding and
current row) as total from test_window where score > 80 order by logday, userid;
```