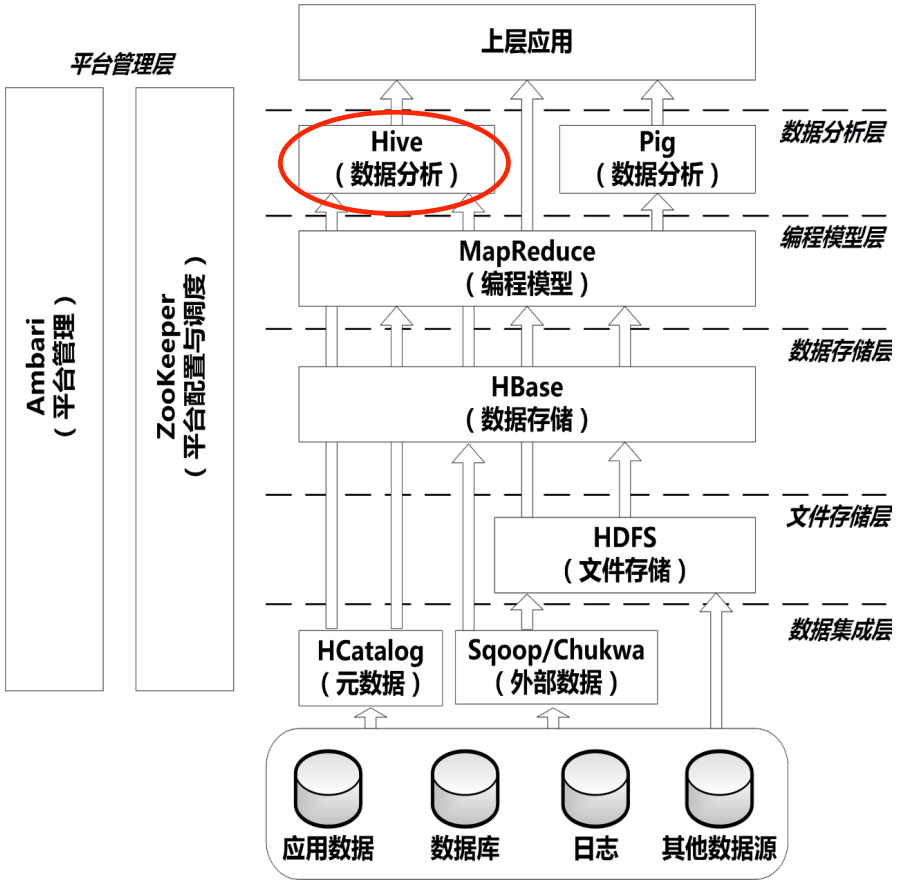
C9. Hive

刘军（liujun@bupt.edu.cn）
北京邮电大学 数据科学中心

Hive的位置



- 数据分析员 vs. 程序员
  - **数据抽象**
  - **操作方式**
  - **执行环境**

Hive的定义

- 来源： Ashish Thusoo, Joydeep Sen Sarma, et al., Facebook, “Hive: A Warehousing Solution over A Map-Reduce Framework”, Proceedings of the VLDB Endowment, Aug. 2009.
- Why Hive ?（http://hive.apache.org/）
  - Hive is a data warehouse system for Hadoop that facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in Hadoop compatible file systems. Hive provides a mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL.
  - 数据库 vs. 数据仓库：存取（面向事务） vs. 分析（面向主题）

感受Hive

代码

```
#include "mapreduce/mapreduce.h"
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
   const string& text = input.value();
   const int n = text.size();
   for (int i = 0; i < n; ) {
   while ((i < n) && isspace(text[i]))    i++;
   int start = i;
   while ((i < n) && !isspace(text[i]))    i++;
   if (start < i)
    Emit(text.substr(start,i-start),"1"); }}};
REGISTER_MAPPER(WordCounter);
```

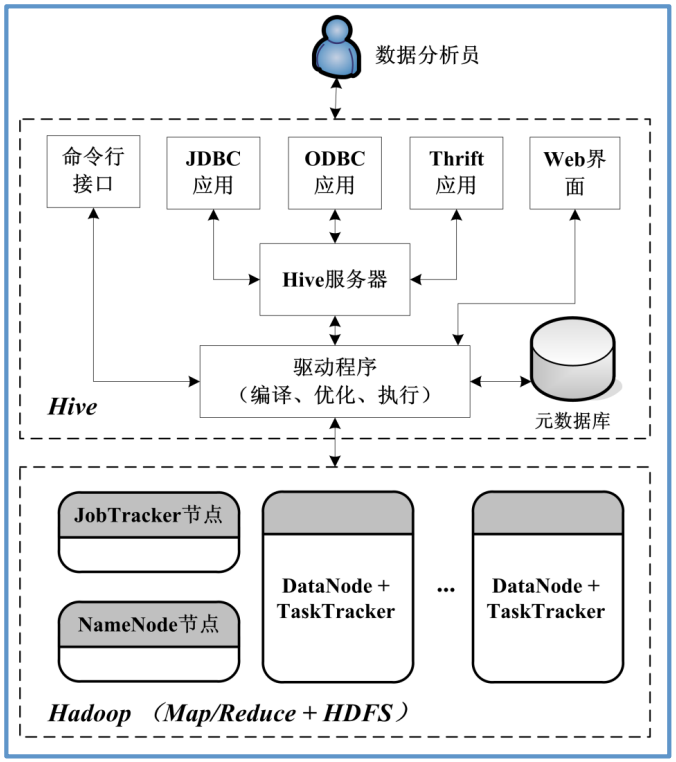
```
class Adder : public Reducer {
 virtual void Reduce(ReduceInput* input) {
  int64 value = 0;
  while (!input->done()) {
   value += StringToInt(input->value());
   input->NextValue();
  }
  Emit(IntToString(value)); }};
REGISTER_REDUCER(Adder);
```

```
int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
  MapReduceSpecification spec;
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");
  out->set_combiner_class("Adder");
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();
  return 0;
}
```

Hive

```
SELECT * FROM log WHERE date > '2012-12-01' ;
```

Hive架构与组件



- **用户操作接口**：
  - 命令行接口（shell）
  - Web界面
  - Hive应用
- **Hive服务器**：
  - Hive应用可以以JDBC、ODBC和Thirft接口访问指定地址和端口的Hive服务器。
- **驱动程序**：
  - 负责处理Hive语句，完成编译、优化和执行的工作
  - 生成相应的MapReduce任务与HDFS节点进行数据交互
- **元数据库**：
  - 存储Hive中与数据表相关的元数据，包括数据的库、表和分区组织等信息
- **Hadoop框架**
  - HDFS：文件存储
  - MapReduce：命令执行

Hive数据组织

- 库（Database）、表（Table）、分区（Partition）和桶（Bucket）
  - **库（Database）**：与传统的关系型数据库类似，库就是若干数据表的集合，代表了用户希望管理的一个数据集。
  - **表（Table）**：与关系型数据库中表的概念类似，表在逻辑上由存储的数据和描述表格中数据形式的相关元数据组成，每张表中的数据以序列化文件的形式存储在相应的HDFS目录下。
  - **分区（Partition）**：分区是为了在数据量过大时提高数据存储效率而对表进行划分的机制，每个表可以包含一个或多个分区，每个分区在物理上是HDFS存储表数据的目录下的子目录。
  - **桶（Bucket）**：通过对指定列值进行哈希并将结果除以桶的个数取余数的方法，将一张表或分区分到不同桶中，从而在查询少量数据只需在对应的桶中进行查找，提高查询效率。一个桶对应一个HDFS文件，存在于一个分区或一张表的目录中，文件按照字典顺序排列的。

- 分区及桶示例：

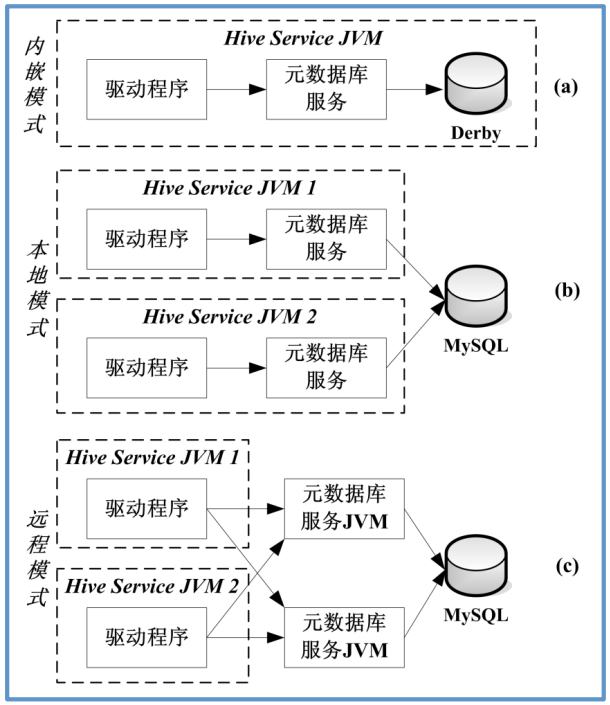
字段	userID	deviceID	type	roamType	url	time
类型	String	String	int	int	String	String

/user/hive/warehouse/logs/Type=1/file_0000
/user/hive/warehouse/logs/Type=1/file_0001
…
/user/hive/warehouse/logs/Type=2/file_0000

  - 分区后，Type列作为划分列在文件系统的目录中体现，因此此列不会出现在数据文件的列值中

Hive中的元数据

- 元数据是描述数据的库/表/分区/桶组织形式和关系的基础数据
  - 元数据集中存放在元数据库（Metastore）中
  - 采用了传统的关系数据库（MySQL或Derby）存储元数据
- 元数据的三种存储模式



- **内嵌（Embedded）模式**：
  - Hive元数据库的默认运行模式
  - 元数据库服务和Hive服务运行在同一个JVM中
  - 元数据存储在内嵌在本地磁盘的Derby数据库中
  - 只能存在一个Hive会话，适合Hive的简单试用
- **本地（Local）模式**：
  - 使用独立的数据库作为元数据的存储组件，例如MySQL
  - 元数据存储独立，支持多个Hive服务共享一个元数据库
- **远程（Remote）模式**：
  - Hive服务和元数据库服务运行在不同的JVM中
  - Hive服务器可以访问多个元数据库服务，具有很好的可扩展性
  - 元数据库可部署于防火墙之后，通过JDBC或ODBC远程访问，具有更好的安全性

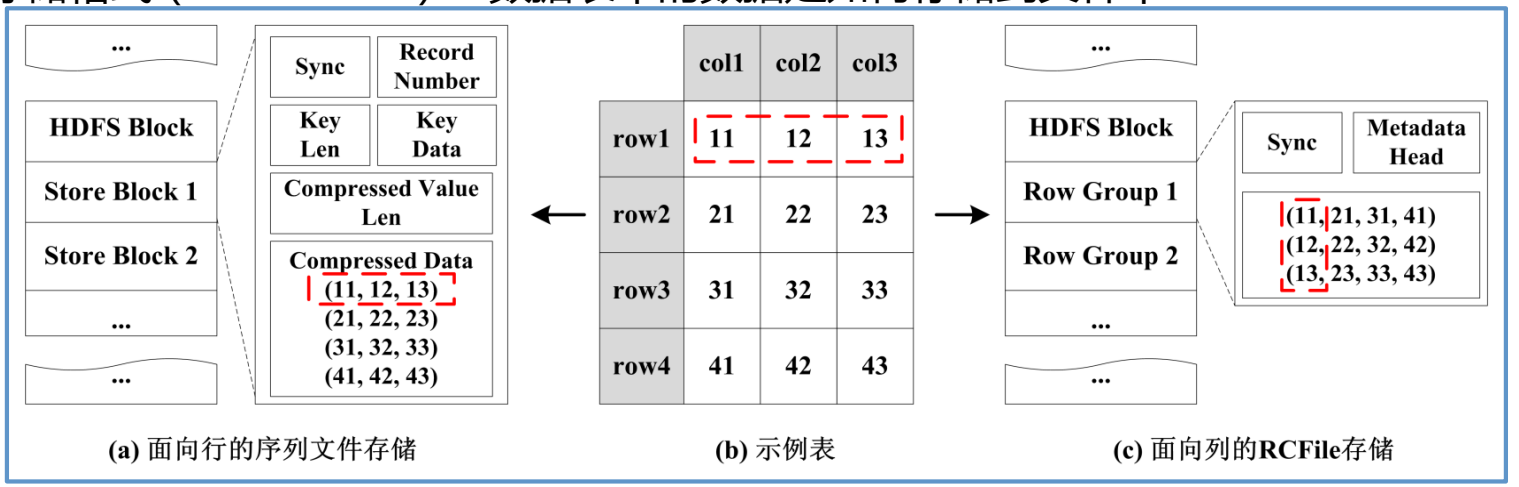
Hive数据存储 - 行格式

- Hive数据存储格式的两个维度：
  - 行格式（row format）
  - 文件存储格式（file format）

- 行格式
  - SerDe（Serializer-Deserializer）
    - ✓ 每行数据是在存储时要进行序列化操作（Serializer），即将每行数据的属性结构及属性值转换为二进制数据流存入文件中
    - ✓ 读取时要进行反序列化操作（Deserializer），即将文件中存储的二进制数据量还原为每行数据的属性结构和属性值
  - Hive提供了多种序列化反序列化接口，开发者也可以扩展自己的SerDe接口设置数据存储格式
  - 默认使用的SerDe接口为：延迟简单SerDe（LazySimpleSerDe）
    - ✓ 以回车符（ASCII码13）区分不同的行
    - ✓ 以CTRL-A（ASCII码1）区分一行中的不同列
    - ✓ 延迟（Lazy）指只有当某列数据被访问时才会进行反序列化操作，以提高数据存储效率

Hive数据存储 - 文件格式

- 文件存储格式（file format）：数据表中的数据是如何存储到文件中



(a) 面向行的序列文件存储　(b) 示例表　(c) 面向列的RCFile存储

文本文件存储

- 带分隔符的文本文件
- 行以回车符进行分割，列用CTRL-A分割
- 更多分隔符支持复杂数据类型的存储，例如使用CTRL-B对集合（Collection）中的元素进行分割
- 优点：结构简单，适合MapReduce程序
- 缺点：占用空间大，IO效率低

面向行的序列文件（图a）

- 按顺序存放的KV对
- 支持可分割并行的压缩
- 数据面向行存储
- 优点：同一行记录的位于同一HDFS节点上，适应快速数据加载和动态负载均衡
- 缺点：不支持部分列的快速查询，磁盘空间利用率低

面向列的RCFile文件（图c）

- “先水平切分，再垂直切分”
- 若干行组合为行组（Row Group），每个行组存放于一个HDFS Block中，同一行的数据存储在同一节点上
- 不同行的同一列数据顺序存放，然后再存储下一列数据
- 每个行组在存储时包含三个部分：同步标识，元数据头，存储数据的数据段
- 结合了行存储和列存储的各自优点

HQL - Hive Query Language

- 提供给数据分析人员使用的类似SQL的命令语法
  - 适应海量数据分析实际应用环境的需求
  - 为使用者和程序提供一个与传统SQL使用习惯相近的分析语言
- 不是SQL语言所遵循的SQL-92标准中的全集：非全集，有扩展

对比点	SQL	HiveQL
更新方式	UPDATE,INSERT,DELETE	INSERT OVERWRITE TABLE （整表填充）
事务支持	支持	不支持
索引	支持	不支持
处理时延	秒级	分钟级
数据类型	整数、浮点型、定点数、文本和二进制串、时间	整数、浮点数、布尔型、字符串、数组、映射、结构
内置函数	多（数百个）	少（十几个）
多表插入	不支持	支持
查询语句	满足SQL-92标准	FROM条件中只支持单表或单视图， SORT BY只支持部分排序方式， LIMIT只可限制返回行的数量，不支持HAVING
连接查询	满足SQL-92标准	内连接、外连接、半连接和带提示的SQL-92语法
子查询	完全支持	只能在FROM条件中，不支持相关子查询
视图	视图可更新，可以实体化化也可以非实体化	视图只读，不支持实体化
扩展函数	支持用户定义函数和存储过程	支持用户定义函数和MapReduce脚本

HQL中的数据类型

分类	数据类型	描述	示例
基本数据类型	TINYINT	有符号整数，1个字节，范围从-128到127	1
	SMALLINT	有符号整数，2个字节，范围从-32768到32767	1
	INT	有符号整数，4个字节，范围从-2147483648到2147483647	1
	BIGINT	有符号整数，8个字节，范围从-922337203685477580 8到9223372036854775807	1
	FLOAT	单精度浮点数，4个字节	1.0
	DOUBLE	双精度浮点数，8个字节	1.0
	BOOLEAN	true/false	true
	STRING	字符串	'b', "a"
复杂数据类型	ARRAY	一组有序字段。字段的类型必须相同	array(1,2)
	MAP	一组无序的键/值对。键的类型必须是基本类型；值可以是任意类型。同一个映射的键的类型必须相同，值的类型也必须相同	map(1, "a")
	STRUCT	一组命名的字段。字段的类型可以不同	struct(1, "a", 2.0)

- STRUCT支持将多个不同类型的字段封装为一个结构型数据

  STRUCT<index:INT，name:STRING，weight:DOUBLE>

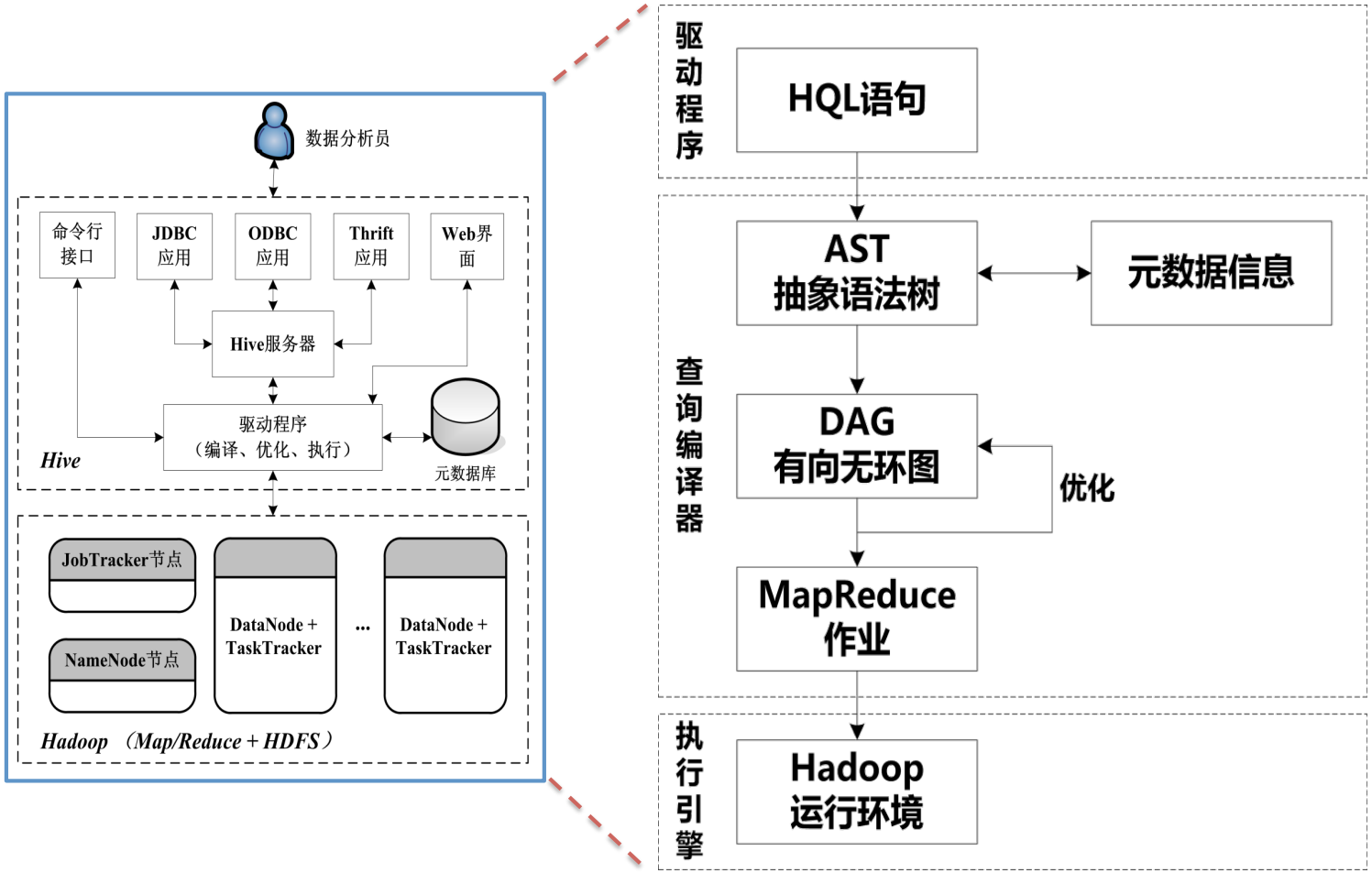
- 复杂数据类型在定义时要使用尖括号指明其数据字段的数据类型，并且允许任意层次的嵌套关系

  col1 ARRAY<INT>

  col2 MAP<INT, STRING>

  col3 STRUCT<index:INT, name:STRING, weight:DOUBLE>

HQL执行流程


数据分析员
命令行接口
JDBC应用
ODBC应用
Thrift应用
Web界面
Hive服务器
驱动程序
（编译、优化、执行）
元数据库
Hive
JobTracker节点
NameNode节点
DataNode + TaskTracker
...
DataNode + TaskTracker
Hadoop （Map/Reduce + HDFS）
驱动程序
HQL语句
查询编译器
AST
抽象语法树
元数据信息
DAG
有向无环图
优化
MapReduce
作业
执行引擎
Hadoop
运行环境

HQL实例（0）- 源数据

- 源数据
  - 源数据存放在HDFS的/data/目录下
  - 用户信息存放于user.txt文件

```
1111，25，male
2222，30，male
3333，30，female
...
```

  - 访问日志存放于log.txt文件

```
20130601，3g.qq.com，1111，portal，10
20130602，cnn.com，2222，news，20
20130602，baidu.com，3333，search，15
20130603，news.qq.com，1111，news，100
20130603，baidu.com，3333，search，15
...
```

HQL实例（1） - 创建数据表

- user.txt → user

user		
userID	**age**	**gender**
STRING	*INT*	*STRING*
1111	25	male
2222	30	male
3333	30	female

- log.txt → log

log				
date	**URL**	**userID**	**category**	**traffic**
STRING	*STRING*	*STRING*	*STRING*	*INT*
20130601	3g.qq.com	1111	portal	10
20130602	cnn.com	2222	social	20
20130602	baidu.com	3333	search	15
20130603	news.qq.com	1111	news	100
20130603	baidu.com	3333	search	15

HQL实例（1） - 创建数据表

- Hive中的两种数据表：内部数据表、外部数据表
  - 内部数据表：Hive将把数据文件移动到它管理的数据仓库目录下
  - 外部数据表：Hive只创建并管理外部数据表的元数据，数据文件则仍使用原始文件
  - 区别
    - ✓ 创建外部表时需要指定外部数据源文件所在的位置
    - ✓ 加载数据（LOAD）和删除表（DROP）两类操作

- user表

```
CREATE TABLE user (userID STRING, age INT, gender STRING);
```

- log表

```
CREATE EXTERNAL TABLE log (date STRING, URL STRING, userID, STRING, category STRING,
  traffic INT) LOCATION '/data/log.txt';
```

HQL实例（1）- 创建数据表（优化）

- 分区：将数据表按照某个列进行切分然后存放在不同的目录下，提高对相应列的查询效率
  - 例如：将日志文件中的大量记录按照日期进行分区存放，提高日期查询效率

```
CREATE TABLE log (date STRING, URL STRING, userID, STRING, category STRING,
    traffic INT) PARTITIONED BY (date STRING);

LOAD DATA INPATH '/data/log_20130601.txt' INTO table log PARTITIONED (date='20130601');
```

  - 注意：date没有在列描述区域出现，仅在指定分区的语句段出现
  - 查看分区命令：SHOW PARTITIONS

```
hive> SHOW PARTITIONS log;
        date=20130601
        date=20130602
```

- 桶：支持快速生成抽样样本集及提高查询效率
  - 例如：使用用户ID作为划分桶的字段，并生成5个桶

```
CREATE TABLE user (userID STRING, age INT, gender STRING)
    CLUSTERED BY (userID) INT 5 BUCKETS;
```

  - 在数据存储时将对用户ID进行哈希并用结果除以5后获得的余数决定分配到哪个桶中
  - 桶就是分区目录下的文件

HQL实例（2） - 加载数据

- 加载数据：从本地文件系统或HDFS中导入数据到数据表中
- 命令：LOAD DATA

- 内部数据表
  - 加载数据相当于一个移动操作，即将源数据文件移动到Hive管理的文件目录下
  - 对于user表，原始数据文件hdfs://data/user.txt将被移动到Hive管理的数据仓库hdfs://user/hive/warehouse/user目录下

```
LOAD DATA INPATH '/data/user.txt' INTO table user;
```

- 外部数据表
  - 加载操作仅仅是建立元数据
  - 不会检查加载命令中的原始数据文件是否存在，留在之后进行数据操作时才进行

```
LOAD DATA INPATH '/data/log.txt' INTO table log;
```

HQL实例（3）- 修改数据表

- 命令：ALTER TABLE
- 可修改表名、列名、列字段类型、增加列、替换列
- 将原来名为user的表重命名为new_user

```
ALTER TABLE user RENAME TO new_user;
```

  - 内部数据表：
    - ✓ 数据文件所在的目录将被重命名为新的表名
    - ✓ 例如将hdfs://user/hive/warehouse/user目录被重命名为hdfs://user/hive/warehouse/new_user
  - 外部数据表：
    - ✓ 只修改元数据，不会操作数据文件和目录
- 增加列

```
ALTER TABLE user ADD COLUMNS (addres STRING);
```

- 修改列名和列字段类型

```
ALTER TABLE user CHANGE addres address STRING;
```

- 删除列

```
ALTER TABLE user REPLACE COLUMNS (userID STRING, age INT, gender STRING);
```

- 注意：HiveQL修改表结构后，仅修改元数据中的表定义信息，并不真正更新数据文件中的内容，因此需要确保一致性

HQL实例（4） - 删除数据表

- 删除数据表命令：DROP TABLE

- 例如删除clone_log_1表

```
DROP TABLE clone_log_1;
```

- 区别：
  - 内部数据表：删除表操作是一个真正的删除操作，会将表对应的元数据和数据文件一并删除
  - 外部数据表：删除操作仅是删除元数据，真正的数据文件不会删除

HQL实例（5） - 数据查询

- 数据查询命令：SELECT ... FROM ...

- 支持与SQL类似的子句
  - 条件（WHERE）
  - 分组（GROUP）
  - 排序（ORDER BY）
  - 限制返回数量（LIMIT）等

```
SELECT * FROM log WHERE date > '20130601';
```

HQL实例（7） - 数据查询（Group By）

- 查询每类网页的流量和

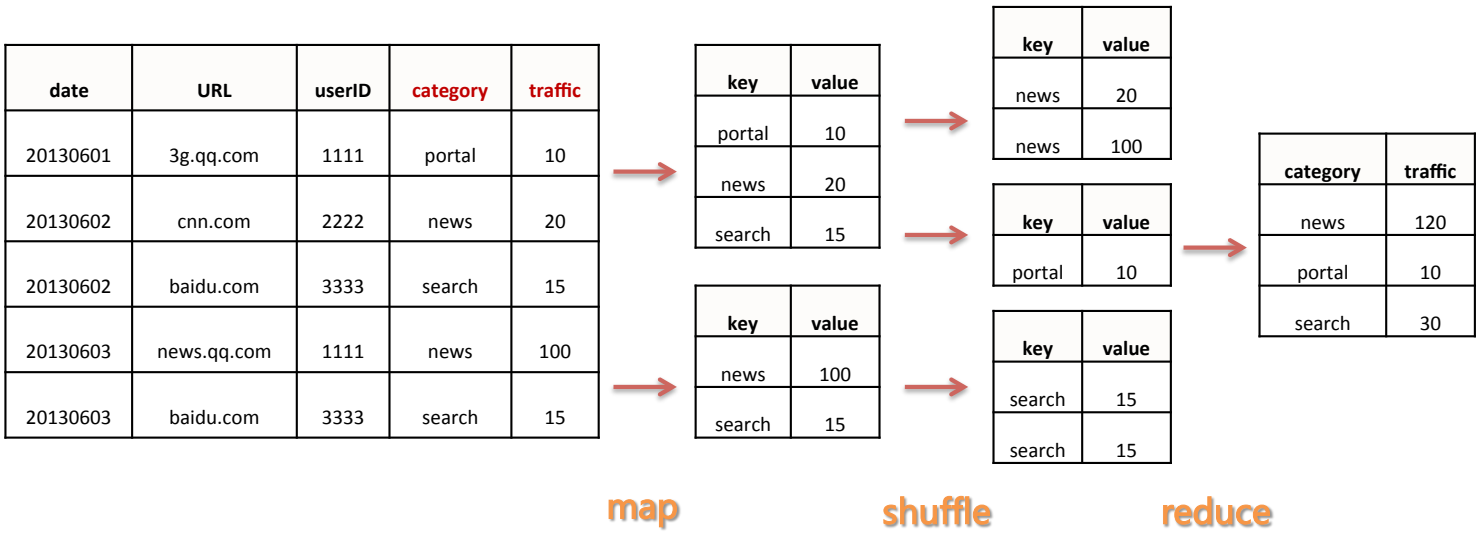
```
INSERT INTO TABLE sum_traffic
    SELECT category, SUM(traffic) FROM log GROUP BY category;
```

log				
date	**URL**	**userID**	**category**	**traffic**
STRING	*STRING*	*STRING*	*STRING*	*INT*
20130601	3g.qq.com	1111	portal	10
20130602	cnn.com	2222	news	20
20130602	baidu.com	3333	search	15
20130603	news.qq.com	1111	news	100
20130603	baidu.com	3333	search	15

sum_traffic	
category	**traffic**
STRING	*INT*
news	120
portal	10
search	30

HQL Group By的实现

```
INSERT INTO TABLE sum_traffic
    SELECT category, SUM(traffic) FROM log GROUP BY category;
```



HQL内建函数

- 内建函数
  - count()
  - sum()
  - avg()
  - max()
  - min()
  - ...

```
SELECT category, SUM(traffic) FROM log GROUP BY category;
SELECT URL, COUNT(DISTINCT userID) FROM log GROUP BY URL;
```

HQL自定义函数（1）- 使用方法

- 内建函数不能满足一些特性需求，因此提供了用户自定义函数扩展功能
  - 普通自定义函数（UDF）
  - 聚集自定义函数（UDAF）
  - 表生成自定函数（UDTF）

- 自定义函数的使用步骤：

① 将编写好的程序进行打包，例如将myfunction类打包为myfunction.jar包，并将myfunction.jar包存放到/home/myjar目录下。

② 使用ADD JAR命令将myfunction.jar包注册到Hive里。命令为：

```
ADD JAR /home/myjar/ myfunction.jar;
```

③ 使用CREATE TEMPORARY命令为自定义函数编定别名，命令为：

```
CREATE TEMPORARY FUNCTION myfunction AS 'com.hadoopbook.hive.myfunction';
```

④ 在HQL查询语句中使用自定义函数，例如：

```
SELECT myfunction(col) FROM table;
```

HQL自定义函数（2）- 普通自定义函数

- 普通自定义函数（User Defined Function，UDF）
  - 输入单行数据，处理后输出一行数据
  - 编写UDF函数的两个条件：
    - ✓ 继承org.apache.hadoop.hive.ql.exec.UDF类
    - ✓ 实现UDF类中的evaluate方法

- 示例：将表中的age列值加1后输出

```
1:  import org.apache.hadoop.hive.ql.exec.UDF;
2:  public class AddOne extends UDF{
3:      public int evaluate (int age) {
4:          int newAge = age + 1;
5:          return newAge;
6:      }
7:  }
```

- 使用：SELECT addone(age) FROM user;

HQL自定义函数（3）- 自定义聚合函数

- 自定义聚合函数（User-Defined Aggregate Function，UDAF）
  - 输入多行数据，处理后输出一行数据
  - 编写UDF函数的两个条件：
    - 继承

    org.apache.hadoop.hive.ql.exec.UDAF类
    - 实现接口类及其5个方法

    org.apache.hadoop.hive.ql.exec.UDAFEvaluator
- 例：查询user表中age列的最大值

```
1:  public class Maximum extends UDAF{
2:    public static class MaximumIntEvaluator
3:     implements UDAFEvaluator{
4:     private int result;
5:     public void init() {
6:       result = 0;
7:     }
8:     public boolean iterate(int value) {
9:       result = Math.max(result, value);
10:      return true;
11:    }
12:    public int terminatePartial() {
13:      return result;
14:    }
15:    public Boolean merge() {
16:      return true;
17:    }
18:    public int terminate{
19:      return result;
20:    }
21:  }
22: }
```

HQL自定义函数（4） - 表生成自定义函数

- 表生成自定义函数（User Defined Table-generating Function，UDTF）
  - 输入单行数据，处理后输出多行数据
  - 编写UDF函数的两个条件：
    - ✓继承类 org.apache.Hadoop.hive.ql.udf.generic.GenericUDTF
    - ✓实现initialize(), process(), close()三个方法
- 示例：查询表中URL列的值，并将URL值以点号进行分割，分割后的每一个值作为一整行输出到结果表中

```
1:  public class StringUDTF extends GenericUDTF{
2:    public void close() throws HiveException {
3:    }
4:    public StructObjectInspector
5:      initialize(ObjectInspector[] args) {
6:    }
7:    public void process(Object[] args) {
8:      String input = args[0].toString();
9:      String[] test = input.split(".");
10:     for (int i=0; i<test.length; i++) {
11:       String result = test[i];
12:       forward(result);
13:     }
14:   }
15: }
```

下面，让我们实践一下……