Transwarp 开发者文档

开发者文档提供了实例演示以及样例代码和管理指令的参考指南，以帮助你快速熟悉并使用Transwarp Data Hub软件发行版产品。

产品使用教程

方便快速了解产品的功能及典型使用场景

产品选型

根据用户需求选择不同类型功能

实例演示

Inceptor和Hyperbase执行高性能离线分析和高并发实时查询

HDFS

分布式文件系统，具有高容错性，适合在大规模数据集中应用。

环境配置

HDFS在本地运行的配置要求

实例操作

对HDFS文件的基本操作，包括创建，删除以及导入等

Hyperbase

Hyperbase是一个高并发低延时，可做OLTP及OLAP的高性能分布式实时数据库。他建立于HBase之上，并添加索引加速的特性

环境准备

介绍了Hyperbase运行所需要的基本环境配置，包括lib库的建立

表格基础操作实例

提供了对Hyperbase表格操作的常用实例：插入，删除，查找等

Hyperbase特性

Hyperbase提供高性能计算分析引擎Inceptor接口以及使用索引以加快数据读取查询速度

Hyperbase API

Transwarp Hyperbase兼容Hbase的java类接口

Inceptor

Inceptor是一种高性能SQL分析工具

语法指南

Inceptor支持的SQL以及PLSQL语法详解

分布式内存列式存储

Inceptor可将数据从磁盘中加载进内存，并基于内存中进行运算

高级功能

分布式内存列式存储高级功能：MAPJOIN，设置过滤器以及利用过滤器优化SQL性能

Inceptor for Hyperbase

Inceptor支持创建与Hyperbase表关联的Inceptor表，利用索引技术加快SQL运行速度

应用工具

Sqoop

Sqoop是一个相互转移Hadoop和关系型数据库中的数据的工具，常用来将一个关系型数据库（例如：MySQL/Oracle/Postgres等）中的数据导入到HDFS

Flume

Flume提供了收集，聚集和转移海量日志文件的连接器

TranswarpR

TranswarpR为Transwarp提供的在R语言环境下使用高性能的分布式计算的接口，文档介绍了TranswarpR使用指南以及调用HDFS，MapReduce，HBase，Hive，Spark的API

TranswarpR 指南

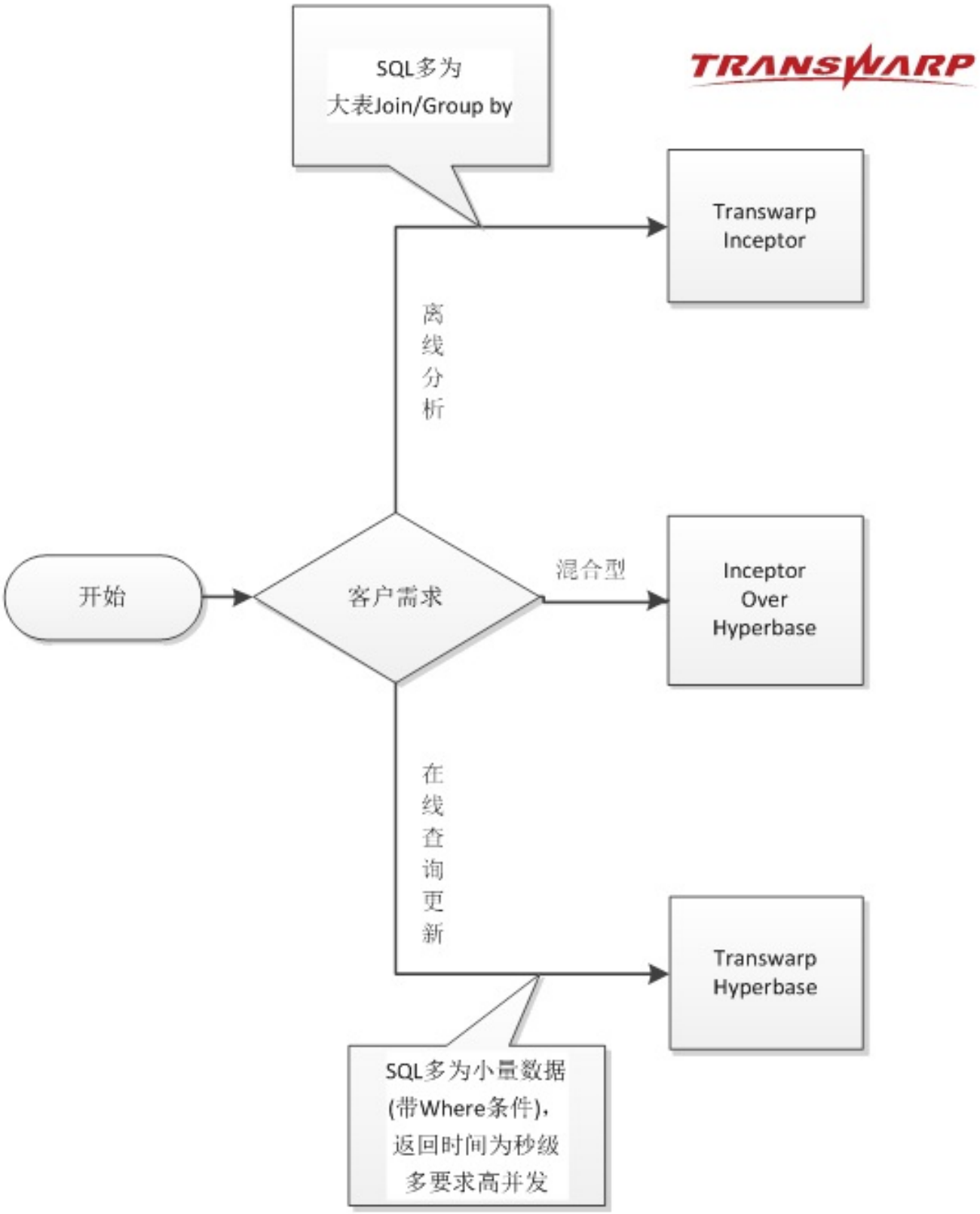
TranswarpR的相关配置及使用

TranswarpR APIs

TranswarpR调用HDFS，MapReduce等的API

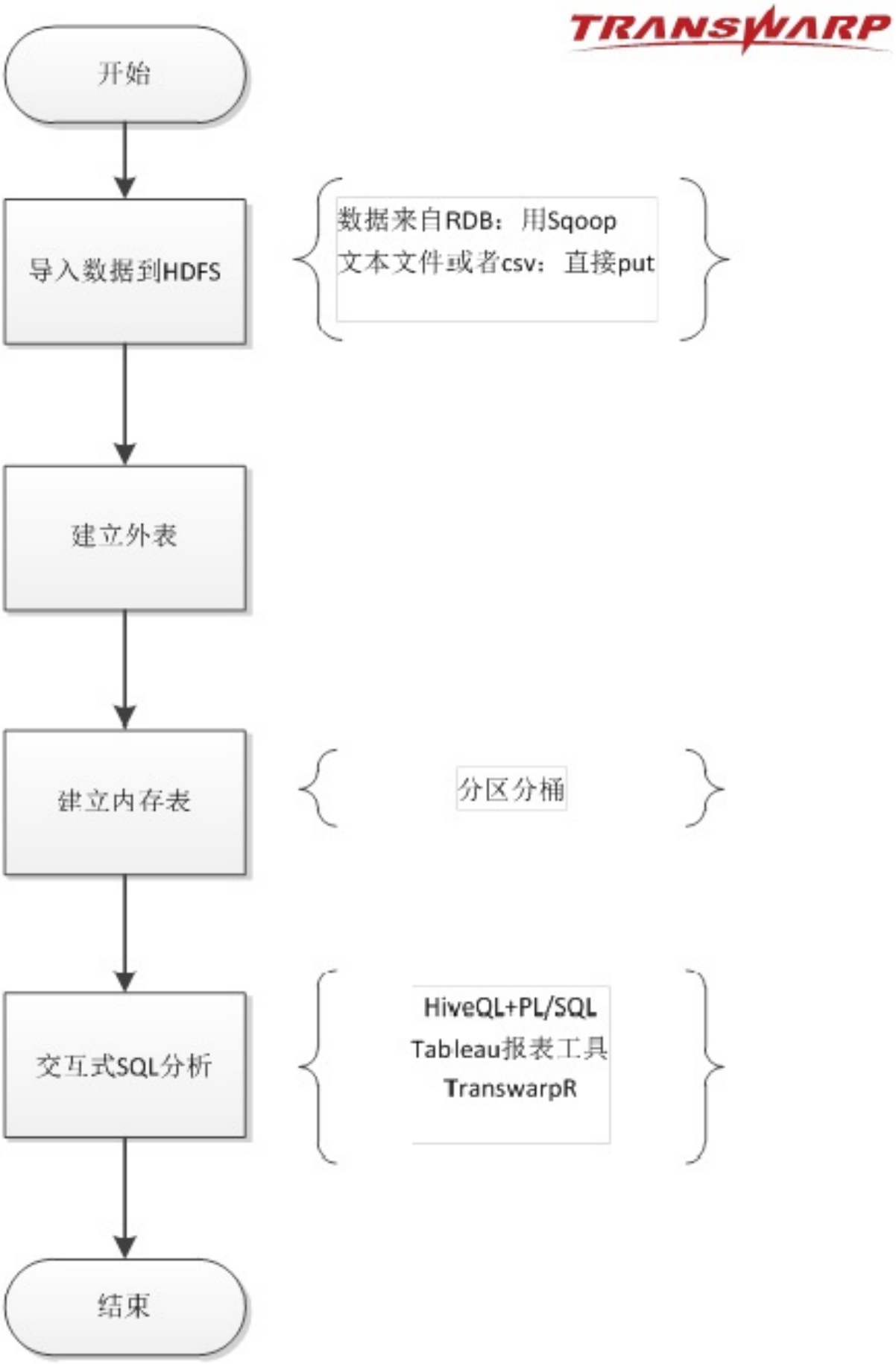
产品选型

Transwarp Data Hub是国内首个全面支持Spark和Hadoop2.2的大数据平台软件，可以用于处理大中小各类数据，根据不同的用户需求，需要选择不同类型的服务以满足要求，下图介绍了产品选型的基本原则：

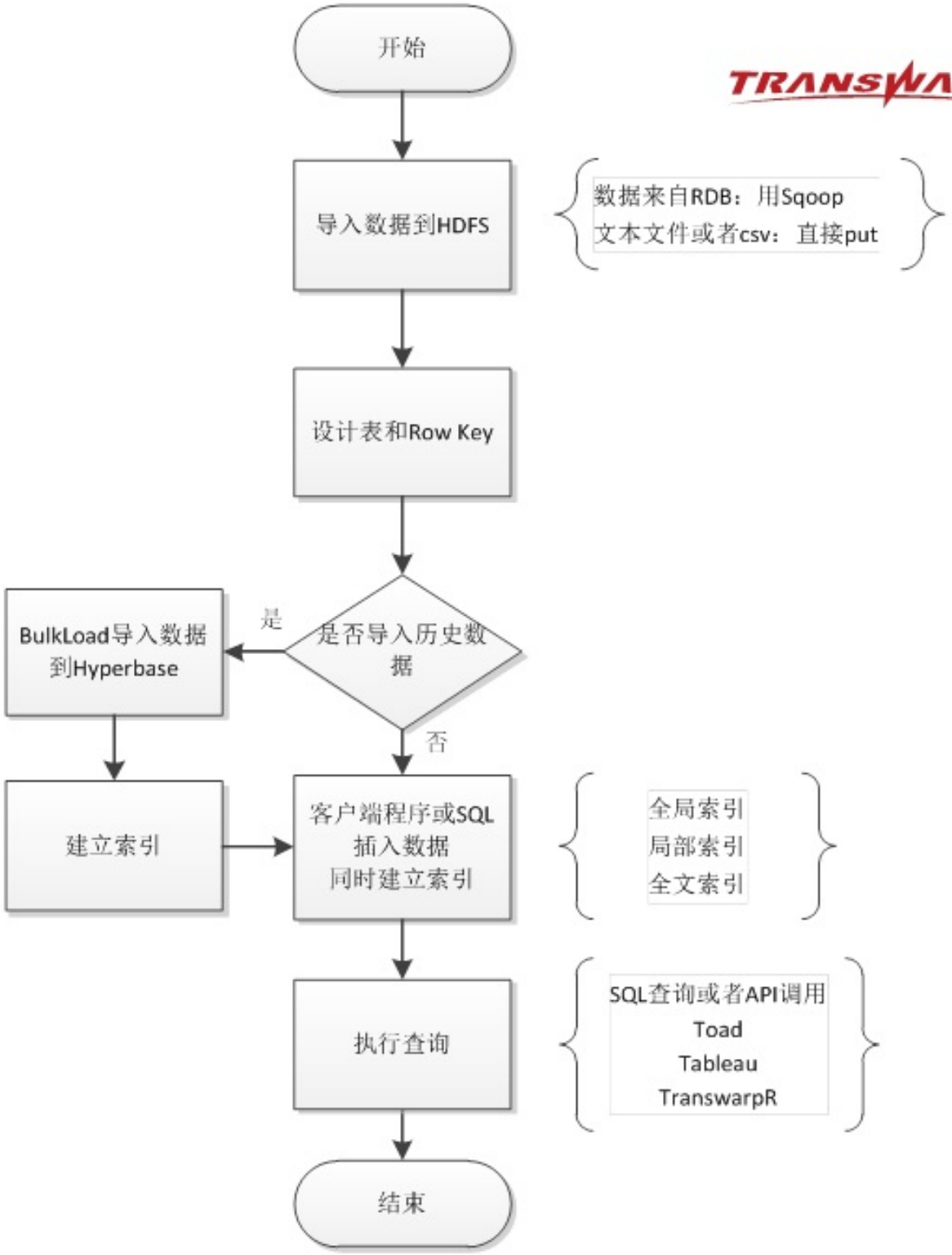


由上图，用户可根据不同需求选定使用Inceptor，Hyperbase或是二者结合版，Inceptor和Hyperbase的使用流程参见下面两张图表。

Inceptor使用流程：



Hyperbase使用流程：


开始
导入数据到HDFS
数据来自RDB：用Sqoop
文本文件或者csv：直接put
设计表和Row Key
是否导入历史数据
是
BulkLoad导入数据到Hyperbase
否
建立索引
客户端程序或SQL插入数据同时建立索引
全局索引
局部索引
全文索引
执行查询
SQL查询或者API调用
Toad
Tableau
TranswarpR
结束

Inceptor & Hyperbase实例演示

本页提供了一个模拟某运营商的通话记录以及用户信息的实例场景，并基于此场景从离线分析和高并发实时查询两个角度来说明Inceptor和Hyperbase的基本操作方式，内容包含了Inceptor的数据导入、加载内存、关联、统计以及Hyperbase的数据导入、建立索引、精确查询、统计分析。

实例场景

本实例场景模拟了某运营商的通话记录表和用户信息表，以导入这两张表到Inceptor和Hyperbase中进行查询、关联和汇聚统计。

通话记录表

- 通话记录表结构

列名	数据类型	含义
MSISDN	STRING	手机号码
OTHER_PARTY	STRING	对方号码
START_DATE	STRING	通话开始日期
START_TIME	STRING	通话开始时间
IMSI	STRING	imsi号

- 通话记录表数据：存放于本地目录`gsm`文件夹下的`gsm_data`文件中

  001|002|20140416|182701|101

  001|003|20140417|182702|101

  002|003|20140418|182708|102

  003|001|20140419|131709|103

  004|001|20140419|131709|104

用户信息表

- 用户信息表结构

列名	数据类型	含义
MSISDN	STRING	手机号码
CERNO	STRING	身份证号
CITY_ID	STRING	开通城市
REG_DATE	STRING	开通日期

- 用户信息表数据：存放于本地目录`userinfo`文件夹下的`userinfo_data`文件中

  001|1111111|SH|20130102

  002|2222222|SH|20130201

  003|3333333|NJ|20130612

  004|4444444|NT|20120612

  005|5555555|BJ|20120711

离线分析

本小节叙述了如何利用Inceptor对上述二表做离线分析：包括将数据导入HDFS以及加载到Inceptor内存表做离线分析。

上述SQL指令都是在Inceptor Shell下手工输入执行，用户同样可以通过JDBC连接到Inceptor Server以执行SQL语句。

数据导入HDFS

- 通话记录表数据存放于本地目录`gsm`文件夹下的`gsm_data`文件中，用户信息表数据存放于本地目录`userinfo`文件夹下的`userinfo_data`文件中：

```
-- 在HDFS目录/user/root下创建test文件夹
hadoop fs -mkdir /user/root/test
-- 本地文件夹导入hdfs的test文件夹
hadoop fs -put gsm /user/root/test
hadoop fs -put userinfo /user/root/test
```

- 除了将本地文件put到HDFS中以导入数据，还可以通过Sqoop将关系型数据库中的数据导入到HDFS，具体导入方法请参见语法参考手册的Sqoop

部分。

加载到Inceptor内存表做离线分析

- 进入Inceptor Shell (transwarp -t -h [Inceptor server]) 之后，使用SQL在Inceptor中建立外表gsm_ext和userinfo_ext：

```
[$host] transwarp> create external table gsm_ext(msisdn string,other_party string,start_date string,start_time string,imsi string) row format delimited fields terminated by '|' stored as textfile location '/user/root/test/gsm';
[$host] transwarp> create external table userinfo_ext(msisdn string,cerno string,city_id string,reg_date string) row format delimited fields terminated by '|' stored as textfile location '/user/root/test/userinfo';
```

- 使用SQL将gsm_ext表和userinfo_ext表中的数据按照msisdn分桶并加载到内存，此处的 set mapred.reduce.tasks 的值需根据reduce阶段的数据量动态调整，每个reduce处理100MB左右的数据量为宜：

```
[$host] transwarp> set mapred.reduce.tasks=[桶数];
[$host] transwarp> create table gsm tblproperties('cache'='ram','filters'='hashbucket(桶数):msisdn') as select * from gsm_ext distribute by msisdn;
[$host] transwarp> set mapred.reduce.tasks=[桶数];
[$host] transwarp> create table userinfo tblproperties('cache'='ram','filters'='hashbucket(桶数):msisdn') as select * from userinfo_ext distribute by msisdn;
```

- 使用SQL对Inceptor内存表进行统计分析：

```
-- 统计gsm表中所有通话记录条数和userinfo表中号码数量
[$host] transwarp> select count(*) from gsm;
[$host] transwarp> select count(*) from userinfo;
-- 统计gsm表中每个msisdn号的记录条数，按记录条数升序输出
[$host] transwarp> select msisdn, count(1) as count from gsm group by msisdn order by count;
-- 以msisdn为JOIN KEY关联表gsm和表userinfo，将结果输到表result中
[$host] transwarp> create table result as select gsm.*, userinfo.cerno, userinfo.city_id, userinfo.reg_date from gsm join userinfo on gsm.msisdn = userinfo.msisdn;
```

JDBC连接到Inceptor Server

- 为了使用JDBC连接到Inceptor Server以执行SQL进行数据分析，需要以下jar包：

```
hive-exec-0.12.0-transwarp.jar
hive-jdbc-0.12.0-transwarp.jar
hive-metastore-0.12.0-transwarp.jar
hive-service-0.12.0-transwarp.jar
libfb303-0.9.0.jar
commons-cli-1.2.jar
commons-logging-1.1.1.jar
hadoop-annotations-2.2.0-transwarp.jar
hadoop-auth-2.2.0-transwarp.jar
hadoop-common-2.2.0-transwarp.jar
hadoop-hdfs-2.2.0-transwarp.jar
log4j-1.2.17.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.6.1.jar
```

- 连接到Inceptor server并执行一些SQL语句的代码如下：

```
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;

public class inceptorJDBC{
   private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";
   public static void main(String[] args) throws SQLException{
      try{
       Class.forName(driverName);
      }catch(ClassNotFoundException e){
       e.printStackTrace();
       System.exit(1);
      }
      //172.16.1.110 is the IP address of Inceptor Server
      Connection connection = DriverManager.getConnection("jdbc:transwarp://172.16.1.110:10000/default","","");
      Statement st = connection.createStatement();
      //drop if exists table
      String tableName1 = "gsm";
      String dropIfExistsTable = "drop table if exists " + tableName1;
      st.execute(dropIfExistsTable);
      //create table
      st.execute("set mapred.reduce.tasks=4");
      String createTable = "create table " + tableName1 + " tblproperties('cache'='ram','filters'='hashbucket(4):msisdn') as select * from gsm_ext distribute by msisdn";
      st.execute(createTable);
      //make queries on existing tables gsm and userinfo
      String tableName2 = "userinfo";
      String sql1 = "select count(*) from " + tableName2;
      ResultSet res = st.executeQuery(sql1);
      while (res.next()){
         System.out.println(String.valueOf(res.getInt(1)));
      }
      String sql2 = "select msisdn, count(1) as count from " + tableName1 + " group by msisdn order by count";
      res = st.executeQuery(sql2);
      while (res.next()) {
       System.out.println(res.getString(1) + "\t" + String.valueOf(res.getInt(2)));
      }
   }
}
```

高并发实时查询

本小节叙述了如何建立带索引的Hyperbase表并将其映射到Inceptor中进行高并发查询。

建立带索引的Hyperbase表

- Hyperbase可以使用API或者Shell进行操作，兼容Hbase的API和Shell命令，并增加自定义接口及命令例如索引的使用，下文中建表和插入数据的操作采用Shell的方式。

  除了通过以上方法单条插入数据，用户同样可通过Bulkload工具从HDFS批量导入数据至Hyperbase。

  Hyperbase中有三种索引：本地索引(Local Index)，全局索引(Global Index)和全文索引。本小节通过对关键列建立本地索引和全局索引来加速查询。

  下面进入Hyperbase Shell创建带索引的gsm表并插入数据，基础hbase_gsm表结构如下所示：
  行键rowkey：[MSISDN]~[START_DATE]~[START_TIME]
  列族f的o列f:o，对该列创建本地索引：local_index_fo
  列簇f的i列f:i，对该列创建全局索引：hbase_gsm_imsi

```
create 'hbase_gsm', 'f', {NAME => 'local_index_fo', LOCAL_INDEX => 'SIMPLE_INDEX|INDEXED=f:o'}
put 'hbase_gsm', '001~20140416~182701', 'f:o', '002'
put 'hbase_gsm', '001~20140416~182701', 'f:i', '101'
put 'hbase_gsm', '001~20140417~182702', 'f:o', '003'
put 'hbase_gsm', '001~20140417~182702', 'f:i', '101'
put 'hbase_gsm', '002~20140418~182708', 'f:o', '003'
put 'hbase_gsm', '002~20140418~182708', 'f:i', '102'
put 'hbase_gsm', '003~20140419~131709', 'f:o', '001'
put 'hbase_gsm', '003~20140419~131709', 'f:i', '103'
put 'hbase_gsm', '004~20140419~131709', 'f:o', '001'
put 'hbase_gsm', '004~20140419~131709', 'f:i', '104'
add_index 'hbase_gsm','hbase_gsm_imsi','SIMPLE_INDEX|INDEXED=f:i,TYPE=STRING,DCOP=true'
rebuild_index 'hbase_gsm','hbase_gsm_imsi'
```

映射到Inceptor中进行高并发查询

- 在Inceptor中建立Hyperbase的gsm映射表：

```
[$host] transwarp> create external table hyperbase_gsm (key struct<msisdn:string, start_date:string, start_time:string>, fo string, imsi string) row format delimited
collection items terminated by '~' stored by 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' with serdeproperties ("hbase.columns.mapping"=":key,f:o,f:i")
tblproperties("hbase.table.name"="hbase_gsm");
```

- 多个线程执行下述查询，模拟多并发场景：

```
-- 查询号码001在2014年4月16日的拨出的所有电话记录（按rowkey查询）
[$host] transwarp> set ngmr.exec.mode=local;
[$host] transwarp> select * from hyperbase_gsm where key.msisdn = "001" and key.start_date = "20140416";

-- 查询号码001拨给号码002的所有通话记录（本地索引）
[$host] transwarp> set ngmr.exec.mode=local;
[$host] transwarp> select * from hyperbase_gsm where key.msisdn = "001" and fo = "002";
```

Hyperbase参数说明：set ngmr.exec.mode=local是用于加速精确查询，提高并发度，不能在大量数据统计分析时使用，在做全量数据分析时建议使用set ngmr.exec.mode=cluster

特别提醒：在做全量数据统计分析的时候，建议使用set hyperbase.reader=true

环境配置

HDFS的开发与使用需要导入特定jar包至本地客户端（如Eclipse）

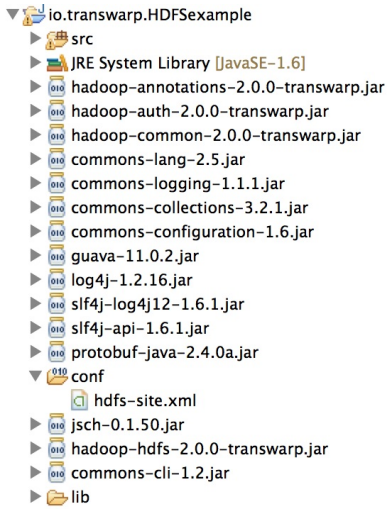
这些jar包可以从集群的以下目录获取：

```
/usr/lib/hadoop
/usr/lib/hadoop/lib
```

以及运行必要的配置文件`hdfs-site.xml`，此文件可在集群的`/etc/`目录下对应的HDFS服务名称，如`/etc/hdfs1/conf`中找到。

当导入这些文件至本地后，须将他们加入到对应的CLASSPATH中，以使程序得以正常运行。

下图显示了HDFS在Eclipse环境中开发运行的目录结构，以及最基本的jar包：



对目录的操作

此小节介绍了对目录的基本操作：创建目录与删除目录。

CreateDir.java

CreateDir.java提供了最基本的创建目录的实例：在localhost下的HDFS中新建了目录**newDir**及其父目录**user**

FileSystem实例提供了创建目录（及其父目录）的方法：`public boolean mkdirs(Path p) throws IOException`

如果目录（及所有父目录）创建成功，则返回**true**

通常情况下，不需要显式创建一个目录，因为调用`mkdirs()`方法写入文件时会自动创建父目录

```
package io.transwarp.HDFSexample;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CreateDir {
    public static void main(String[] args) throws IOException {
        String rootPath = "hdfs://localhost:8020"; // localhost为namenode的IP
        Path p = new Path(rootPath + "/tmp/user/newDir");
        Configuration conf = new Configuration();

        FileSystem fs = p.getFileSystem(conf);
        boolean b = fs.mkdirs(p);
        System.out.println(b);
        fs.close();
    }
}
```

Delete.java

FileSystem的`delete()`方法可以永久性删除文件或者目录：`public boolean delete(Path p, boolean recursive) throws IOException`

如果路径**p**是一个文件或者空目录，那么recursive的值会被忽略。当recursive为**true**时，一个非空目录以及内容会被删除，否则会抛出IOExcaption

例如，**Delete.java**删除了上例中创建的(父)目录**user**及其目录下所有文件。

```
package io.transwarp.HDFSexample;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class Delete {
    public static void main (String[] args) {
        String rootPath = "hdfs://localhost:8020";
        Path p = new Path(rootPath + "/tmp/user");
        Configuration conf = new Configuration();

        try {
            FileSystem fs = p.getFileSystem(conf);
            boolean b = fs.delete(p, true);
            System.out.println(b);
            fs.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

对文件的操作

本例中主要介绍了对HDFS文件的基本操作，包括了文件的创建，从本地上传文件以及在H文件末尾追加新的内容。

CreateFile.java

本例**CreateFile.java**实现了在IP地址为172.16.1.85的机器节点中创建新的文件file.txt并放置在目录/tmp/newFile/下，若目录/newFile不存在，`create()`方

法会自行创建此目录。

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class CreateFile {

    public static void main (String[] args) throws IOException {
        String rootPath = "hdfs://172.16.1.85:8020";
        // craete new file called "file.txt"
        Path p = new Path(rootPath + "/tmp/newFile/file.txt");
        Configuration conf = new Configuration();
        FileSystem fs = p.getFileSystem(conf);
        fs.create(p);

        fs.close();
    }
}
```

UploadFile.java

当通过`create()`创建文件后，可以从本地上传文件，导入至新建的H文件file.txt中。

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class UploadFile {
    public static void main (String[] args) throws IOException {
        String rootPath = "hdfs://172.16.1.85:8020";
        Configuration conf = new Configuration();
        String localFile = "/users/root/Desktop/hdfs.txt";
        InputStream in = new BufferedInputStream(new FileInputStream(localFile));

        Path p = new Path(rootPath + "/tmp/file.txt");
        FileSystem fs = p.getFileSystem(conf);
        OutputStream out= fs.create(p);

        IOUtils.copyBytes(in, out, conf);

        fs.close();
        IOUtils.closeStream(in);
    }
}
```

当然HDFS也支持直接从本地导入文件至HDFS，FileSystem 提供了

`public void copyFromLocalFile(src, dst);`

`public void copyFromLocalFile(boolean delSrc, path src, path dst);`

`public void copyFromLocalFile (boolean delSrc, boolean overwrite, path src, path dst);`

`public void copyFromLocalFile(boolean delSrc, boolean overwrite, path[] srcs, path dst);`

AppendFile.java

对于已经存在的文件，FileSystem提供方法`append(Path p)`在该文件中追加新的内容。值得注意的是，HDFS默认会有三份拷贝文件，若此处的节点少于三个，则在运行中会报错，故需要手工指定备份数：`setReplication(Path, short)`

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class AppendFile {

    public static void main (String[] args) throws IOException {
        String rootPath = "hdfs://172.16.1.85:8020";
        Path p = new Path(rootPath + "/tmp/file.txt");
        Configuration conf = new Configuration();

        FileSystem fs = p.getFileSystem(conf);
        fs.setReplication(p, (short) 1);
        OutputStream out = fs.append(p);
        InputStream in = new BufferedInputStream(new FileInputStream("/Users/Jnavie/Desktop/hdfs.txt"));

        IOUtils.copyBytes(in, out, conf);
        fs.close();
        IOUtils.closeStream(in);
    }
}
```

环境准备

Hyperbase的运行与使用须建立在HDFS, MapReduce和Hive已成功安装配置并运行的前提下。

此外，为了让Hyperbase Client正常运行，需要安装特定依赖包及进行一些必要的配置。

依赖包

Hyperbase的Java客户端(eg. Eclipse)须在CLASSPATH下建立 lib 文件夹，导入对应版本的.jar包。

这些.jar包可从集群的以下目录获取：

```
/usr/lib/hbase/
/usr/lib/hbase/lib
/usr/lib/hadoop
/usr/lib/hadoop/lib
/usr/lib/zookeeper
/usr/lib/hadoop-mapreduce
/usr/lib/hadoop-yarn
/usr/lib/hadoop-hdfs
```

用户可以根据自己的需求下载.jar包，以下.jar包（版本号可能根据软件的更新而有所变化）为维持基本Hyperbase运行所必须：

```
avro-1.5.4.jar
commons-cli-1.2.jar
commons-collections-3.2.1.jar
commons-configuration-1.6.jar
commons-httpclient-3.1.jar
commons-io-2.1.jar
commons-lang-2.5.jar
commons-logging-1.1.1.jar
guava-11.0.2.jar
hadoop-annotations-2.2.0-transwarp.jar
hadoop-auth-2.2.0-transwarp.jar
hadoop-common-2.2.0-transwarp.jar
hadoop-hdfs-2.2.0-transwarp.jar
hadoop-mapreduce-client-common-2.2.0-transwarp.jar
hadoop-mapreduce-client-core-2.2.0-transwarp.jar
hadoop-mapreduce-client-jobclient-2.2.0-transwarp.jar
hadoop-yarn-api-2.2.0-transwarp.jar
hadoop-yarn-client-2.2.0-transwarp.jar
hadoop-yarn-common-2.2.0-transwarp.jar
hbase-0.94.11-transwarp.jar
hdfs-raid-1.0.0.jar
jackson-core-asl-1.8.8.jar
jackson-jaxrs-1.8.8.jar
jackson-mapper-asl-1.8.8.jar
jackson-xc-1.8.8.jar
jsch-0.1.50.jar
log4j-1.2.17.jar
protobuf-java-2.4.0a.jar
slf4j-api-1.6.1.jar
slf4j-log4j12-1.6.1.jar
zookeeper-3.4.5-transwarp.jar
```

配置

所有配置文件储存在 conf 目录下。

系统必须的配置文件 hbase-site.xml 存在于集群中配有Hyperbase服务的机器中的 /etc/hyperbase1/conf 文件下。此处“hyperbase1”为此Hyperbase的服务名称，用户可根据自己的实际情况在相应文件夹下的 conf 中找到 hbase-site.xml ，并将其添加到本地目录的 /conf 中。

其它所需的配置文件还包括： core-site.xml , hdfs-site.xml , mapred-site.xml 和 yarn-site.xml 。用户亦可在相应的 conf 中找到文件并将其添加到本地目录的 conf 中。

ATTENTION：若本机上无法直接连接HDFS，须在 /etc/hosts 文件中添加或修改相关集群的IP与主机名称。

主页

本章节提供了对Hyperbase表格操作的常用实例。基本的，在Hbase Shell下，可通过`list`来查看数据库中有哪些已存在的表。对于一个已经存在的表，可通过`describe 'Hbase表名'`来查看其构造。

1. DDL（创建及删除表格）

本小节叙述如何在Hbase中创建表格以及删除表格。可通过Java和Hbase Shell两种方法实现。

- ## 创建表格

  HBase中表格的创建是通过对操作`HBaseAdmin`这一对象使其调用`createTable()`这一方法来实现。

  其中`HTableDescriptor`描述了表的schema，可在其上通过`addFamily()`这一方法增加列族。

  以下Java代码实现了建立一张简易的Hbase表格'table1'，该表有两个列族，分别为f1和f2。

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableExistsException;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.io.encoding.DataBlockEncoding;
import org.apache.hadoop.hbase.io.hfile.Compression.Algorithm;
import org.apache.hadoop.hbase.regionserver.StoreFile.BloomType;
import org.apache.hadoop.hbase.util.Bytes;

public class createTable{
   private static Configuration config;
   private static HBaseAdmin ha;

   public static void main(String[] args){
      try{
         config = HBaseConfiguration.create();
         config.addResource("core-site.xml");
         config.addResource("hdfs-site.xml");
         config.addResource("yarn-site.xml");
         config.addResource("mapred-site.xml");
         ha = new HBaseAdmin(config);
         //create table descriptor
         String tableName = "table1";
         HTableDescriptor htd = new HTableDescriptor(Bytes.toBytes(tableName));
         //create and configure column families
         HColumnDescriptor hcd1 = new HColumnDescriptor(Bytes.toBytes("family1"));
         hcd1.setBlocksize(65536);
         hcd1.setMaxVersions(1);
         hcd1.setBloomFilterType(BloomType.ROW);
         hcd1.setCompressionType(Algorithm.SNAPPY);
         hcd1.setDataBlockEncoding(DataBlockEncoding.PREFIX);
         hcd1.setTimeToLive(36000);
         hcd1.setInMemory(false);
         HColumnDescriptor hcd2 = new HColumnDescriptor(Bytes.toBytes("family2"));
         hcd2.setBlocksize(65536);
         hcd2.setMaxVersions(1);
         hcd2.setBloomFilterType(BloomType.ROW);
         hcd2.setCompressionType(Algorithm.SNAPPY);
         hcd2.setDataBlockEncoding(DataBlockEncoding.PREFIX);
         //TTL设置为36000，数据存活时间为10小时，超过10小时的数据会在
         //major compaction发生时被删除
         hcd2.setTimeToLive(36000);
         hcd2.setInMemory(false);
         //add column families to table descriptor
         htd.addFamily(hcd1);
         htd.addFamily(hcd2);
         //create table
         ha.createTable(htd);
         System.out.println("Hbase table created.");
      }catch (TableExistsException e){
         System.out.println("ERROR: attempting to create existing table!");
      }catch (IOException e){
         e.printStackTrace();
      }finally{
       try{
        ha.close();
         }catch(IOException e){
          e.printStackTrace();
         }
      }
   }
}
```

在Hbase Shell中，创建表格功能由`create ‘Hbase表名’，[‘列族名’...]`来实现。

例如，`create ‘table1’，‘family1’，‘family2’`同样可创建上述表格。

- **删除表格**

删除表也是通过`HBaseAdmin`来操作，删除表之前首先要disable表。这是一个比较耗时的操作，所以不建议频繁删除表。

以下Java代码实现了对表格“table1”的删除操作：

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class deleteTable{
   private static Configuration config;
   private static HBaseAdmin ha;
   public static void main(String[] args){
      try{
         config = HBaseConfiguration.create();
         config.addResource("core-site.xml");
         config.addResource("hdfs-site.xml");
         config.addResource("yarn-site.xml");
         config.addResource("mapred-site.xml");
         ha = new HBaseAdmin(config);
         String tableName = "table1";
         //Only an existing table can be dropped
         if (ha.tableExists(tableName)){
            //read&write denied
            ha.disableTable(tableName);
            ha.deleteTable(tableName);
            System.out.println("Hbase table dropped!");
         }
      }catch(IOException e){
         e.printStackTrace();
      }finally{
       try{
            ha.close();
         }catch(IOException e){
            e.printStackTrace();
         }
      }
   }
}
```

在Hbase Shell中，删除表格功能由`drop ‘Hbase表名’`来实现。

例如，先`disable ‘table1’`再`drop ‘table1’`同样可删除上述表格。

2. 数据插入

在Java操作中，put方法被用做插入数据。

put方法可以传递单个Put对象：`public void put(Put put) throws IOException`，也可以对很多Put对象进行批量插入：`public void put(List puts) throws IOException`

以下Java代码实现了对表格"table1"的批量数据插入操作。插入数据后，表格有10000行，列族“family1”，“family2”中都包含“q1”，“q2”两个列，其中列族“family1”储存整型数据(int)，列族“family2”储存字符串(string)。

ATTENTION：虽然Hbase支持多种类型储存，但为了应用高性能优化的HyperBase，表格值的储存类型建议一致使用为String。如上例所示，“family1：q1”中原为整数类型，须转制成string后再录入表中。

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;

public class insertTable{
   private static Configuration config;
 public static void main(String[] args) throws IOException{
      config = HBaseConfiguration.create();
      config.addResource("core-site.xml");
      config.addResource("hdfs-site.xml");
      config.addResource("yarn-site.xml");
      config.addResource("mapred-site.xml");
      String tableName = "table1";
      HTable table = new HTable(config, tableName);
      //set AutoFlush
      table.setAutoFlush(true);
      int count = 10000;
      String familyName1 = "family1";
      String familyName2 = "family2";
      String qualifier1 = "q1";
      String qualifier2 = "q2";
      //data to be inserted
      String[] f1q1 = new String[count];
      String[] f1q2 = new String[count];
      String[] f2q1 = new String[count];
      String[] f2q2 = new String[count];
      for(int i = 0; i < count; i++){
         f1q1[i] = Integer.toString(i);
         f1q2[i] = Integer.toString(i+10000);
         f2q1[i] = String.format("f2q1%d", i);
         f2q2[i] = String.format("f2q2%d", i);
      }
      List puts = new ArrayList();
      //insert by rows
      for (int j = 0; j < count; j++){
         //Create a Put object for a specified row-key
         Put p = new Put(Bytes.toBytes(String.format("Row%05d",j)));
         //fill columns
         p.add(Bytes.toBytes(familyName1), Bytes.toBytes(qualifier1), Bytes.toBytes(f1q1[j]));
         p.add(Bytes.toBytes(familyName1), Bytes.toBytes(qualifier2), Bytes.toBytes(f1q2[j]));
         p.add(Bytes.toBytes(familyName2), Bytes.toBytes(qualifier1), Bytes.toBytes(f2q1[j]));
         p.add(Bytes.toBytes(familyName2), Bytes.toBytes(qualifier2), Bytes.toBytes(f2q2[j]));
         puts.add(p);
         //put for every 100 rows
         if((j+1)%100==0){
            table.put(puts);
            puts.clear();
         }
      }
      table.close();
      System.out.println("Data inserted ! ");
   }
}
```

在Hbase Shell中，单条数据插入功能由`put ‘Hbase表名’，‘rowKey’，‘列族名：列名’，‘数据值’`来实现。

3. 数据查询

Hbase表格的数据查询可分为单条查询与批量查询。

- 单条查询是通过匹配rowkey在表格中查询某一行的数据。在Java中可通过`get()`这一方法来实现。
  下列Java代码实现了在表格“table1”中取出指定rowkey一行的所有列的数据：

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;

public class getFromTable{
   private static Configuration config;
   public static void main(String[] args) throws IOException{
      String tableName = "table1";
      config = HBaseConfiguration.create();
      config.addResource("core-site.xml");
      config.addResource("hdfs-site.xml");
      config.addResource("yarn-site.xml");
      config.addResource("mapred-site.xml");
      HTable table = new HTable(config, tableName);
      Get get = new Get(Bytes.toBytes("Row01230"));
      //add target columns for get
      get.addColumn(Bytes.toBytes("family1"), Bytes.toBytes("q1"));
      get.addColumn(Bytes.toBytes("family1"), Bytes.toBytes("q2"));
      get.addColumn(Bytes.toBytes("family2"), Bytes.toBytes("q1"));
      get.addColumn(Bytes.toBytes("family2"), Bytes.toBytes("q2"));
      Result result =  table.get(get);
      //get results
      byte[] rowKey = result.getRow();
      byte[] val1 = result.getValue(Bytes.toBytes("family1"), Bytes.toBytes("q1"));
      byte[] val2 = result.getValue(Bytes.toBytes("family1"),Bytes.toBytes("q2"));
      byte[] val3 = result.getValue(Bytes.toBytes("family2"), Bytes.toBytes("q1"));
      byte[] val4 = result.getValue(Bytes.toBytes("family2"), Bytes.toBytes("q2"));

      System.out.println("Row key: " + Bytes.toString(rowKey));
      System.out.println("value1: " + Bytes.toString(val1));
      System.out.println("value2: " + Bytes.toString(val2));
      System.out.println("value3: " + Bytes.toString(val3));
      System.out.println("value4: " + Bytes.toString(val4));
      table.close();
   }
}
```

在Hbase Shell中，单条数据查找功能由`get ‘Hbase表名’，‘rowKey’，‘列族名：列名’`来实现。

- 批量查询是通过制定一段rowkey的范围来查询。可通过Java中`getScanner()`这一方法来实现。
  下列Java代码实现了在表格“table1”中取出指定一段rowkey范围的所有列的数据：

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.util.Bytes;

public class scanFromTable {
   private static Configuration config;
   public static void main(String[] args) throws IOException{
    config = HBaseConfiguration.create();
      config.addResource("core-site.xml");
      config.addResource("hdfs-site.xml");
      config.addResource("yarn-site.xml");
      config.addResource("mapred-site.xml");
    String tableName = "table1";
      HTable table = new HTable(config, tableName);
      //Scan according to rowkey range
      Scan scan = new Scan();
      //set starting row(included), if not set, start from the first row
      scan.setStartRow(Bytes.toBytes("Row01000"));
      //set stopping row(excluded), if not set, stop at the last row
      scan.setStopRow(Bytes.toBytes("Row01100"));
      //specify columns to scan, if not specified, return all columns ;
      scan.addColumn(Bytes.toBytes("family1"), Bytes.toBytes("q1"));
      scan.addColumn(Bytes.toBytes("family1"), Bytes.toBytes("q2"));
      scan.addColumn(Bytes.toBytes("family2"), Bytes.toBytes("q1"));
      scan.addColumn(Bytes.toBytes("family2"), Bytes.toBytes("q2"));
      //specify maximum versions for one cell, if called without arguments, get all versions, if not called, get only the latest version
      scan.setMaxVersions();
      //specify maximum number of cells to avoid OutOfMemory error caused by huge amount of data in a single row
      scan.setBatch(10000);
      ResultScanner rs = table.getScanner(scan);
      for(Result r:rs){
         byte[] rowKey = r.getRow();
         byte[] val1 = r.getValue(Bytes.toBytes("family1"), Bytes.toBytes("q1"));
         byte[] val2 = r.getValue(Bytes.toBytes("family1"), Bytes.toBytes("q2"));
         byte[] val3 = r.getValue(Bytes.toBytes("family2"), Bytes.toBytes("q1"));
         byte[] val4 = r.getValue(Bytes.toBytes("family2"), Bytes.toBytes("q2"));
         System.out.print(Bytes.toString(rowKey)+": ");
         System.out.print(Bytes.toString(val1)+" ");
         System.out.print(Bytes.toString(val2)+" ");
         System.out.print(Bytes.toString(val3)+" ");
         System.out.println(Bytes.toString(val4));
      }
      rs.close();
      table.close();
   }
}
```

在Hbase Shell中，批量数据查找功能由 `scan ‘Hbase表名’，{COLUMNS=>‘列族名：列名’，STARTROW=>‘起始rowkey’，STOPROW=>‘终止rowkey’}` 来实现。

4. 利用过滤器筛选

过滤器是在Hbase服务器端上执行筛选操作，可以应用到行键(RowFilter)，列限定符(QualifierFilter)以及数据值(ValueFilter)。

这里列举了两个常用的过滤器：RowFilter和SingleColumnValueFilter。

- **RowFilter**

  RowFilter通过行键(rowkey)来筛选数据。

  其中BinaryComparator直接比较两个byte array，可选的比较符(CompareOp)有EQUAL,NOT_EQUAL,GREATER,GREATER_OR_EQUAL,LESS,LESS_OR_EQUAL。

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.BinaryComparator;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.Filter;
import org.apache.hadoop.hbase.filter.RowFilter;
import org.apache.hadoop.hbase.util.Bytes;

public class rowFilter{
   public static void main(String[] args) throws IOException{
      String tableName = "table1";
      Configuration config = HBaseConfiguration.create();
      config.addResource("core-site.xml");
      config.addResource("hdfs-site.xml");
      config.addResource("yarn-site.xml");
      config.addResource("mapred-site.xml");
      HTable table = new HTable(config, tableName);
      Scan scan = new Scan();
      scan.addColumn(Bytes.toBytes("family1"), Bytes.toBytes("q1"));
      Filter filter = new RowFilter(CompareFilter.CompareOp.EQUAL, new BinaryComparator(Bytes.toBytes("Row01234")));
      scan.setFilter(filter);
      ResultScanner scanner = table.getScanner(scan);
      for(Result res:scanner){
         byte[] value = res.getValue(Bytes.toBytes("family1"),Bytes.toBytes("q1"));
         System.out.println(new String(res.getRow())+" value is: "+Bytes.toString(value));
      }
      scanner.close();
      table.close();
   }
}
```

- ## SingleColumnValueFilter

  SingleColumnValueFilter对某一具体列的值进行筛选。

  其中SubstringComparator检查给定的字符串是否是列值的子字符串，可选的比较符(CompareOp)有EQUAL和NOT_EQUAL。

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.CompareFilter;
import org.apache.hadoop.hbase.filter.SingleColumnValueFilter;
import org.apache.hadoop.hbase.filter.SubstringComparator;
import org.apache.hadoop.hbase.util.Bytes;

public class singleColumnValueFilter{
   public static void main(String[] args) throws IOException{
      Configuration config = HBaseConfiguration.create();
      config.addResource("core-site.xml");
      config.addResource("hdfs-site.xml");
      config.addResource("yarn-site.xml");
      config.addResource("mapred-site.xml");
      String tableName = "table1";
      HTable table = new HTable(config,tableName);
      SingleColumnValueFilter filter = new SingleColumnValueFilter(
            Bytes.toBytes("family2"),
            Bytes.toBytes("q1"),
            CompareFilter.CompareOp.NOT_EQUAL,
            new SubstringComparator("45"));
      //when setting setFilterIfMissing(true), rows with "null" values are filtered
      filter.setFilterIfMissing(true);
      Scan scan = new Scan();
      scan.setFilter(filter);
      ResultScanner scanner = table.getScanner(scan);
      for (Result res:scanner){
         byte[] val = res.getValue(Bytes.toBytes("family1"), Bytes.toBytes("q1"));
         System.out.println(new String(res.getRow()));
         System.out.println("value: " + Bytes.toString(val));
      }
      scanner.close();
      table.close();
   }
}
```

- ## FusionRowFilter

FusionRowFilter是一个适用于rowkey定长，需要根据Rowkey中的进行匹配查询，可用于定值以及范围查询。例如：
Rowkey设计为车牌号（8位）+ 时间（YYYYMMDDHHMMSS）+其它此时需要查询所有在2014年5月12日18点10分到20分之间的记录：

```
List ranges = new ArrayList();
ranges.add(new BytesRange(8, 12, Bytes.toBytes(“201405121810”), Bytes.toBytes(“201405121820”)));
FusionRowFilter filter = new FusionRowFilter(ranges);
```

同时FusionRowFilter也支持多个范围，比如需要找5月12日到5月15日，每天18点10分到20分之间的记录：

```
List ranges = new ArrayList();
ranges.add(new BytesRange(8, 8, Bytes.toBytes(“20140512”), Bytes.toBytes(“20140512”)));
ranges.add(new BytesRange(16, 4, Bytes.toBytes(“1810”), Bytes.toBytes(“1820”)));
FusionRowFilter filter = new FusionRowFilter(ranges);
```

Hyperbase 特性

Hyperbase是基于Hbase的高性能数据库，具有两大特性：支持类型以使其更好的与支持类型的Inceptor对接，并利用自身的RowKey有序和根据Value排序的功能提升效率；以及使用索引以加快数据读取查询速度。

支持类型

- **Hyperbase支持类型的意义**

最初，Hyperbase并不支持类型，只是存储数据的二进制形式；虽然Inceptor支持类型，但其并不能保证有序和索引，而且转换效率极低。为了更好的利用Hyperbase本身根据RowKey有序、根据Value排序的特以及Inceptor对类型的支持和优化编码解码效率。TDH特地扩展了Inceptor的类型，使其可以无缝的为Hyperbase表支持类型。当然，为保证兼容性，用户可以选择不对Hyperbase使用类型。同时，Hyperbase支持类型可以保序、保宽。当Hyperbase中包括struct这种结构时，对于struct中某个数据进行查询时可以通过计算直接定位到数据所在的位置，加快查询效率。

- **类型分类**

Inceptor本身支持的简单类型包括boolean，tinyint，smallint，int，bigint，double，float，string；为保持与数据库中的数据来源类型一致，减少用户上层SQL迁移代价，Inceptor特意增加了VARCHAR(n)类型。TDH3.5将会继续增加对Date，DateTime，Binary，Timestamp，Interval以及Decimal类型的支持。

注意：

1. varchar类型必须指定长度，且varchar中长度表示的是byte数组的长度，用户在使用时需要注意中文字符的长度问题
2. varchar和string并不能自动互相转化，Inceptor视这两个类型为完全不相关的类型，如果需要转化需要手动cast(column as varchar(x))或者cast(column as string)
3. Hyperbase推荐使用varchar作为主键的一部分，不推荐使用string作为主键的一部分，这样可以更好的利用产品的优化

- **用SQL使Hyperbase数据批量入库**

```
--Inceptor中对需要入库的批量数据建立外表
drop table hb_src;
create external table hb_src (key varchar(10)，c1 boolean, c2 tinyint, c3 double, c4 int, c5 smallint, c6 bigint, c7 string, cc float, c8 struct<s1:boolean, s2:smallint, s3:int, s4:bigint, s5:float, s6:double, s7:string>) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' COLLECTION ITEMS TERMINATED BY '|' LOCATION '/extdata/hb_src';
--建立Hyperbase表(需要在hhase shell中执行以下指令)
disable 'FuzzyRow'
drop 'FuzzyRow'
create 'FuzzyRow', {NAME => 'f', DATA_BLOCK_ENCODING => 'PREFIX', BLOOMFILTER => 'ROW', TTL => '2678400', VERSIONS => '1'}
--Inceptor中创建外表与Hyperbase表建立映射关系：如果使用Transwarp带类型的编码方式那么stored by属性必须使用io.transwarp.hyperbase.HyperbaseStorageHandler，并且如果使用HyperbaseStorageHandler则是不需要指定分隔符的
--如果是采用文本文件作为底层存储，使用分隔符+字符串的编码方式那么请使用org.apache.hadoop.hive.hbase.HBaseStorageHandler作为StorageHandler的选项
drop table hb_table;
create external table hb_table(key struct<kchar:varchar(10),kint:int,klong:bigint,kstring:string,rd:bigint>, value struct<c1:boolean, c2:tinyint, c3:double, c4:int, c5:smallint, c6:bigint, c7:string, cc:float, ccc:varchar(8)>, sv1 string, sv2 varchar(5), sv3 double, sv4 int, sv5 boolean)
stored by 'io.transwarp.hyperbase.HyperbaseStorageHandler' with serdeproperties('hbase.columns.mapping'=':key,f:c,f:c1,f:c2,f:c3,f:c4,f:c5')
tblproperties('hbase.table.name'='FuzzyRow');
--将需要入库的数据导入到Hyperbase表对应的外表中
insert into table hb_table select named_struct('kchar', key, 'kint', c4, 'klong', c6, 'kstring', c8.s7, 'rd', uniq()), named_struct('c1', c1, 'c2', c2, 'c3', c3, 'c4', c4, 'c5', c5, 'c6', c6, 'c7', c7, 'cc', cc, 'ccc', cast(key as varchar(8))), key as sv1, cast(key as varchar(8)) as sv2, c3 as sv3, c6 as sv4, c1 as sv5
from hb_src;
```

使用索引

在Hyperbase中，通过使用索引来加快数据的查询速度。

Hyperbase中有三种索引：本地索引、全局索引、全文索引。

本章节将首先介绍Hyperbase中本地索引、全局索引两种索引在数据查询中的优势，之后详述两种索引的添加及使用：本地索引（Local Index），全局索引（Global Index）。

之后将阐述Local与Global两种索引的选择技巧。

最后本章节将介绍全文索引。

优势

使用索引的优势主要在于两点：查询快速以及使用方便

查询快速：索引的基本设计思想是对HBase表中的需要经常作为查询条件的列建立一个映射到主数据的索引。查询时可以利用索引特性进行快速定位并返回查询结果。实验证明，通过使用索引，用户查询的响应速度可以达到原来的20 ~ 100倍。

使用方便：索引使用非常方便。无论是通过HBase Shell还是JAVA API，用户都可以很快的学会使用。如果用户是通过上层TRANSWARP的SQL直接调用，用户只需要学会几条简单的SQL即可。两种索引的具体使用方式可见本章节后续部分

索引类型

- 索引相关信息

索引名称	适用范围	带类型	不带类型	LocalIndex	GlobalIndex	对应的Class
SmpleIndex	单一列	支持	支持	支持	支持	org.apache.hadoop.hbase.index.NoUpdateSimpleIndexBuilder
SegmentCombineIndex	多列组合	支持	支持	支持	支持	org.apache.hadoop.hbase.index.SegmentCombineIndexBuilder
StructIndex	struct结构	支持	暂不支持	支持	支持	org.apache.hadoop.hbase.index.StructIndexBuilder

本地索引（Local Index）

Local Index的索引是Region级别的索引：Local Index不涉及到网络交互，故其可以保证本地事务。索引与原始数据可以保证强一致性。不会出现数据不一致的情况。

Local Index使用简单，通过hbase shell或是JAVA API都可以对其进行操作

- 目前hbase shell可以支持Local Index的功能如下:创建表时同时创建索引，为已有表创建索引，为已有表中数据生成索引等。下面就各个功能如何使用进行举例。

创建表时同时创建索引：

为列族f1的列q1创建SimpleIndex索引，名称为index_f1q1 ， 为列族f1:q2和f1:q3创建SegmentCombineIndex索引，名称为index_f1q2q3。其中HBASE_TABLE_NAME为创建的Hbase表名，不能与已经存在的表名冲突。

```
create'HBASE_TABLE_NAME',{NAME=>'f1'},
{NAME=>'index_f1q1',LOCAL_INDEX=>'org.apache.hadoop.hbase.index.NoUpdateSimpleIndexBuilder|INDEXED=f1:q1，TYPE=INTEGER'}
create'HBASE_TABLE_NAME',{NAME=>'f1'},
{NAME=>'index_f1q2q3',LOCAL_INDEX=>'org.apache.hadoop.hbase.index.SegmentCombineIndexBuilder|INDEXED=f:a:8|f:b:9,TYPE=INTEGER:INTEGER'}
```

当Hyperbase中包括struct这种结构时，对于struct中某个数据进行查询时可以通过计算直接定位到数据所在的位置，可以创建StructIndex索引

```
create'HBASE_TABLE_NAME',{NAME=>'f1'},{NAME=>'org.apache.hadoop.hbase.index.StructIndexBuilder|INDEXED=f:q4,TYPE=INTEGER:INTEGER:INTEGER,
LOCATION=1'}
```

PS:根据原始表数据是否带类型，选择是否使用TYPE这个参数。StructIndex前提数据必须是带类型的，还有一个参数为UPDATE，下面会说明。

为已有表创建索引：对已有表创建索引必须先将表disable掉。

为列族f1的列q1创建索引，名称为index_f1q1，索引构建方式为SIMPLE_INDEX。其中HBASE_TABLE_NAME为存在的Hbase表名。(SegmentIndex,StructIndex类似)

```
disable 'HBASE_TABLE_NAME'
alter 'HBASE_TABLE_NAME',{NAME=>'index_f1q1',LOCAL_INDEX=>'org.apache.hadoop.hbase.index.NoUpdateSimpleIndexBuilder|INDEXED=f1:q1'}
enable 'HBASE_TABLE_NAME'
```

为已有表中数据生成索引：对已有表中数据生成索引必须先将表disable掉。

可以通过FAMILIES=>{['index1', 'index2']}指定为某几个Local Index生成数据，当此参数为空时，默认为该表所有的Local Index生成数据。其中HBASE_TABLE_NAME为存在的Hbase表名。

```
disable 'HBASE_TABLE_NAME'
build_local_index 'HBASE_TABLE_NAME',{FAMILIES=>['index_f1q1']}
或build_local_index 'HBASE_TABLE_NAME'
enable 'HBASE_TABLE_NAME'
```

若为已有表中数据生成索引时间过长，可通过以下命令消除构建的索引：此时，表中的索引列将被删除掉并被重新添加，之后再次调用生成索引的方法即可

```
delete_local_index_task 'HBASE_TABLE_NAME'
```

- 若是通过hbase shell方式无法满足用户的需求，那么用户可以通过JAVA API完成同样的操作（以SegmentCombineIndex为例）。

创建表时同时创建索引

```
String SEGMENT_COMBINE_INDEX = SegmentCombineIndexBuilder.class.getName() + "|INDEXED=f1:q1:2|f1:q2:2,DCOP=true";
//Create HTableDescriptor
String tableName = "table1";
HTableDescriptor htd = new HTableDescriptor(Bytes.toBytes(tableName));
//Create column family
String family1 = "family1";
HColumnDescriptor hcd1 = new HColumnDescriptor(Bytes.toBytes(family1));
//Create SimpleIndex for column family family1, qualifier q1
String qualifier1 = "q1";
IndexBuilder localIndex = IndexBuilderUtils.createLocalIndexByConf(indexConf);
//Create local index column descriptor
String indexFamily1 = "indexFamily1";
HColumnDescriptor indexHcd1 = new HColumnDescriptor(Bytes.toBytes(indexFamily1));
indexHcd1.setLocalIndex(localIndex);
//add column families
htd.addFamily(indexHcd1);
htd.addFamily(hcd1);
//create table
ha.createTable(htd);
```

为已有表创建索引

```
String SEGMENT_COMBINE_INDEX = SegmentCombineIndexBuilder.class.getName() + "|INDEXED=f1:q1:2|f1:q2:2,DCOP=true";
String tableName = "table1";
//before adding local index for existing table, disable it
ha.disableTable(tableName);
//new local index for family family4, column q1
IndexBuilder localIndex = IndexBuilderUtils.createLocalIndexByConf(SEGMENT_COMBINE_INDEX);
HColumnDescriptor indexFamily4 = new HColumnDescriptor("indexFamily4");
indexFamily4.setLocalIndex(localIndex);
ha.addColumn(Bytes.toBytes(tableName), indexFamily4);
ha.enableTable(tableName);
```

为已有表中数据生成索引

```
String tableName = "table1";
ha.disableTable(tableName);
ha.buildLocalIndexData(tableName);
while(true){
  if(ha.isTableInBuildingLocalIndex(Bytes.toBytes(tableName))){
     System.out.println("Waiting for create local index");
     Thread.sleep(1000*10);
     continue;
   }
break;
}
```

若生成索引中间出现问题，长时间无法完成，用以下方法删除构建索引

```
ha.deleteBuildTaskOfTable(tableName);
```

通过索引进行数据查询

```
HTable ht = new HTable(getConfiguration(), tableName);
Scan scan = new Scan();
Scan indexScan = new Scan();
indexScan.addFamily(indexColumn);
scan.setIndexScan(indexScan, indexColumn);
scan.setUseLocalIndex(true);    // true代表通过索引查询数据 ,默认为 false：代表不通过索引查询数据
scan.setAccessMain(false);     // false返回index数据，true返回数据库原始数据
ResultScanner rs = ht.getScanner(scan);
```

通过对某一列建立索引，可以方便的通过索引数据快速查询并返回数据。同时，Local Index的使用非常方便，用户只需要执行scan.setUseLocalIndex();即可。

此命令表示用户希望利用Local Index快速进行访问。Local Index会根据用户在scan中设置的filter（要有对Value建立Filter）自动选择使用Local Index。若用户是通过Hive操作HBase，那么用户可以直接通过Hive语法决定是否使用Local Index加快扫描。

Local Index使用要点：

1. HBase中不包括UPDATE操作，若用户每次插入的ROW_KEY都是新的ROW_KEY，那么可以将其称为NO_UPDATE，反之则是UPDATE。

2. SmpleIndex又分为SIMPLE_INDEX/NU_SIMPLE_INDEX，SIMPLE_INDEX每次插入时都会主动去查找原始表并将ROW_KEY、列族、列相同的数据的索引删除掉，故插入效率略低；NU_SIMPLE_INDEX则不会进行查找。若用户能够保证每次插入的数据ROW_KEY都是新的，可以使用NU_SIMPLE_INDEX方式加快插入效率。

全局索引（Global Index）

Global Index的索引存储与原表独立，索引本身是以一张表的形式存在。

SimpleIndex/SegmentCombineIndex/StructInndex同样适用于GlobalIndex。

类似Local Index，Global Index同样可以通过hbase shell或是JAVA API的方式进行创建、删除。

- 目前hbase shell可以支持Global Index的新建索引、删除索引及重建索引操作，下面就各个功能如何使用进行举例。

对已存在的表创建新的Global Index

HBASE_TABLE_NAME为存在的Hbase表名，为其列族f1的列q1创建索引，名称为INDEX_NAME，创建的索引表的名称为[HBASE_TABLE_NAME]_[INDEX_NAME]，需确保该名称不存在。

```
add_index 't1', 'index_name', 'org.apache.hadoop.hbase.index.NoUpdateSimpleIndexBuilder|INDEXED=f1:q1,TYPE=INTEGER'
```

对于新添加的索引表，其并不会自动对原来已存在的数据建立索引。因此对于插入数据后新添加的索引，或者由于某些误操作造成索引文件不一致的情况可以通过重建索引的方式重新生成索引数据。

HBASE_TABLE_NAME为存在的Hbase表名，要重建的索引名称为INDEX_NAME。

```
rebuild_index 'HBASE_TABLE_NAME','INDEX_NAME'
```

对于不再使用的索引可以通过删除索引的方式对数据进行删除

HBASE_TABLE_NAME为存在的Hbase表名，要删除的索引名称为INDEX_NAME。

```
delete_index 'HBASE_TABLE_NAME','INDEX_NAME'
```

注意：

1.重建索引的过程中最好停止插入新数据。

2.重建索引过程使用MapReduce方式，所以请保证YARN可用。如果需要终止任务，需要手工关闭对应的MapReduce任务。

3.删除索引表的时候，要使用delete_index这个命令，普通的删除表方式不能完全删除索引表信息。

- 用户可以通过JAVA API完成同样的操作，并且可以有更多更控制的细节。

对已存在的表创建新的Global Index

```
String SEGMENT_COMBINE_INDEX = SegmentCombineIndexBuilder.class.getName() + "|INDEXED=f1:q1:2|f1:q2:2,DCOP=true";
SegmentCombineIndexBuilder indexBuilder = (SegmentCombineIndexBuilder)IndexBuilderUtils.createLocalIndexByConf(SEGMENT_COMBINE_INDEX);
GlobalIndexAdmin globalIndexAdmin = new GlobalIndexAdmin(getConfiguration());
globalIndexAdmin.addGlobalIndexWithOutCompress(tableName, indexBuilder, indexName);
```

删除已有的Global Index

```
GlobalIndexAdmin indexAdmin = new GlobalIndexAdmin(configuration);
indexAdmin.dropGlobalIndex(Bytes.toBytes(tableName), Bytes.toBytes(indexFamily));
```

重建Global Index

```
GlobalIndexAdmin indexAdmin = new GlobalIndexAdmin(configuration);
indexAdmin.rebuildIndex(Bytes.toBytes(tableName), Bytes.toBytes(indexFamily));
```

若用户希望插入和查询表的时候的使用Global Index，需要用IndexHTable来替代原有的HTable进行插入查询操作

插入

```
IndexHTable table = new IndexHTable(configuration, tableName);
//Set do not auto flush
table.setAutoFlush(false);
Put put = new Put(Bytes.toBytes(rowKey));
put.add(Bytes.toBytes(family), Bytes.toBytes(qualifier), Bytes.toBytes("transwarp"));
table.put(put);
```

查询

```
IndexHTable table = new IndexHTable(configuration, tableName);
Filter filter = new SingleColumnValueFilter(Bytes.toBytes(family), Bytes.toBytes(qualifier), CompareFilter.CompareOp.EQUAL,Bytes.toBytes("transwarp"));
Scan scan = new Scan();
scan.setFilter(filter);
for(Result r : table.getScanner(scan)){
//Do something
}
```

选择技巧

Local Index和Global Index均可加快查询速度，但是用户在选择使用索引时需要注意以下几点：

1. 创建索引的字段需要有足够的区分度，即根据索引中的单一条件所获得的数据量远小于表的数据总量，类似于对性别、星期几这样的字段建立索引是没有意义的。

2. 对于Local Index及Global Index之间的选择，有一条比较简单的选择原则：

如果查询条件中会同时包含RowKey范围及索引查询条件的，使用Local Index，反之如果通常只单独包含索引查询条件的，选择Global Index。

注意：以上原则和表结构设计相关，因此在设计表结构的时候需要综合考虑索引的创建。

全文索引

全文索引通过建立词库的方式记录词的出现位置及次数，以加速数据查询。

Hyperbase支持高效的全文索引，本小节介绍了如何创建与使用全文索引。

创建全文索引

创建全文索引需要在创建表时同时指定需要对哪些column families的哪些columns创建全文索引，目前不支持对已存在的表新建全文索引。

- `IndexedHBaseColumn` 表示对Hbase表格的一个column做全文索引。
- `IndexFieldMetadata` 表示对Hbase表格的一个column family做全文索引，一个 `IndexFieldMetadata` 可包含多个 `IndexedHBaseColumn` 。可以为 `IndexFieldMetadata` 配置具体的检索参数，如分词算法。
- `IndexMetadata` 描述一张表的索引信息，一个 `IndexMetadata` 可包含多个 `IndexFieldMetadata` ，一张表只能提供一个 `IndexMetadata` 对象。

例如，如下代码在表的column family "f"，column "A" "C"上创建全文索引：

```
HTableDescriptor htd = new HTableDescriptor(table);
String family = "f";
String[] columns = {"A","C"};
//column family
HColumnDescriptor cf = new HColumnDescriptor(family);
htd.addFamily(cf);
//index information
IndexMetadata indexMetadata = new IndexMetadata();
IndexFieldMetadata field = new IndexFieldMetadata();
field.setName(Bytes.toBytes(family));
field.setIndexed(true);
field.setTokenized(true);
field.setAnalyzerMetadataName("org.apache.hadoop.hbase.search.IndexContext$MMSegComplexAnalyzerMetadata");
IndexedHBaseColumn ic = new IndexedHBaseColumn(Bytes.toBytes(family));
for(String column:columns){
  ic.addQualifierName(Bytes.toBytes(column));
}
field.addIndexedHBaseColumn(ic);
indexMetadata.addFieldMetadata(field);
htd.setIndexMetadata(indexMetadata);
```

使用全文索引

Hbase支持两种使用全文索引的方式：

（1）直接使用检索API-IndexSearcherClient

例如：以下代码返回column family "f"下包含"hbase"关键字的记录，取评分最高的10条记录。

```
HIndexedTable table = new HIndexedTable(configuration, tableName);
IndexSearcherClient searcher = new IndexSearcherClient(table);
IndexSearchResponse response = searcher.search(new ContainQueryFilter("f", "hbase"), 0, 10);
IndexableRecord[] records = response.getRecords();
```

（2）使用scan接口

例如：以下代码同样返回column family "f"下包含"hbase"关键字的记录，取评分最高的10条记录。

```
Scan s = new Scan();
s.setFilter(new ContainQueryFilter("f", "hbase"));
ResultScanner scanner = table.getScanner(s);
Result r = null;
int count = 0;
while((r = scanner.next()) != null && count < 10){
  count++;
  dumpResult(r);
}
```

Inceptor

Inceptor是一种高性能SQL分析工具。

本页内容包括Inceptor的语法指南，创建列式存储，高级功能以及Inceptor for Hyperbase。

Inceptor语法指南

Inceptor的语法全面扩展于HiveQL，使用metadata store存储表结构和属性，支持对已经存在metadata store中的数据表进行计算。并在此基础上融入了大量PLSQL特性，从而极大程度丰富了Inceptor的分析语法。

在介绍其语法特性前，Inceptor常用的相关命令将被首先简述，包括进入Inceptor的命令行参数以及交互模式常用命令。

类似于HiveQL，Inceptor也拥有一套完备的Data Definition Language(DDL)语句，以及相应所支持的数据类型。

同时，Inceptor拥有Data Manipulation Language(DML)以及SELECT语句用来执行数据导入及分析计算，并包含大量的内置运算符和内置函数予以辅助。此外，Inceptor同样支持用户自定义函数(UDF)以方便分析计算。

进入Inceptor的命令行参数

```
transwarp -t -h [Inceptor server IP]
-e: #命令行sql语句
-f <filename>: #SQL文件
-H, --help: #帮助
--hiveconf: #指定配置文件
-i: #初始化文件
-S, --silent: #静态模式（不将错误输出）
-v,--verbose: #详细模式
```

交互模式常用命令

```
[$host] transwarp> show tables; #查看所有表名
[$host] transwarp> show tables 'ad*';  #查看以'ad'开头的表名
[$host] transwarp> set 命令; #设置变量与查看变量；
[$host] transwarp> set -v; #查看所有的变量
[$host] transwarp> set hive.stats.atomic; #查看hive.stats.atomic变量
[$host] transwarp> set hive.stats.atomic=false; #设置hive.stats.atomic变量
[$host] transwarp> dfs  -ls; #查看hadoop所有文件路径
[$host] transwarp> dfs  -ls /inceptor1/user/hive/warehouse/; #查看inceptor所有文件
[$host] transwarp> dfs  -ls /inceptor1/user/hive/warehouse/ptest; #查看ptest文件
[$host] transwarp> source file <filepath>; #在client里执行一个inceptor脚本文件
[$host] transwarp> quit; #退出交互式shell
[$host] transwarp> exit; #退出交互式shell
[$host] transwarp> reset; #重置配置为默认值
[$host] transwarp> !命令; #从Inceptor shell执行一个shell命令,如 [$host] transwarp>!ls;
```

Data Definition Language(DDL)

Inceptor的DDL语句实现了对数据库，表，视图的创建，删除以及修改，包括 CREATE DATABASE语句， DROP DATABASE语句， CREATE TABLE语句， ALTER TABLE语句， DROP TABLE语句， CREATE VIEW语句， 以及DROP VIEW语句。

- **CREATE DATABASE语句**

  数据库可被看作是一个含有多个相关信息的表的集合，创建数据库的语法如下：

  ```
  CREATE DATABASE [IF NOT EXISTS] database_name [COMMENT 'database_comment']
    [LOCATION hdfs_path];
  ```

  示例：

  ```
  create database new_db;
  --进入Inceptor后，默认进入名为default的数据库，使用use语句切换到指定的数据库
  use new_db;
  ```

- **DROP DATABASE语句**

  已创建的数据库可被删除，删除数据库将同时删除其下所有的表和视图，因此应提前做好备份，删除数据库的语法如下：

```
DROP DATABASE [IF EXISTS] database_name;
```

示例：

```
drop database new_db;
--当一个database被use语句选用时，其不可以被删除
create database temp;
use temp;
drop database temp;
ERROR: AnalysisException: Cannot drop current default database: temp
```

- **CREATE TABLE语句**

Inceptor中表的创建是核心内容，具体语法如下：

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
 [(col_name data_type [COMMENT col_comment], …)]
 [COMMENT table_comment]
 [PARTITIONED BY (col_name data_type [COMMENT col_comment], …)]
 [CLUSTERED BY (col_name, col_name, ...)
   [SORTED BY (col_name [ASC|DESC], …)]
 INTO num_buckets BUCKETS
 ]
 [
   [ROW FORMAT row_format]
 [STORED AS file_format]|[STORED BY `storage.handler.class.name` [WITH SERDEPROPERTIES(...)]]
 ]
 [TBLPROPERTIES (property_name=property_value, ...)]
 [AS select_statement]
 [LOCATION hdfs_path]
```

注解如下：
可使用EXTERNAL关键字创建外表，同时用LOCATION关键字指定HDFS路径来指向实际数据
若不加EXTERNAL，则创建内部表，此时数据会移动到数据仓库指向的路径(/inceptor1/user/hive/warehouse/)
若删除外部表，只有元数据被删除，不删除数据；若删除内部表，元数据和数据会被一起删除
支持CTAS(create table as select)建表，此时不能指定表中有哪些列，只能通过select语句选择
分区是表的部分行的集合, 可以为频繁使用的数据建立分区, 这样查找分区中的数据时就不需要扫描全表, 这对于提高查找效率很有帮助，分区用PARTITIONED BY语句实现
桶是按行分开组织特定字段，每个桶对应一个reduce操作和一个数据仓库下的文件，采用分桶字段哈希并对桶数取余的方式来分桶，桶数量最好为质数以尽可能保证每个桶中数据量分布均匀
分桶用CLUSTERED BY语句实现，同时可用SORTED BY语句保证桶内有序
row_format处语法为DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char][MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]
分别用来指定列分隔符，struct分隔符，map分隔符，行分隔符
支持的存储格式file_format：textfile，sequencefile，rcfile以及orc

示例：

```
--创建外部表，指定列分隔符和数据路径
create external table tb_ext(col1 string, col2 string) row format delimited fields terminated by '\t' location '/user/hadoop/input';
--创建分区表，分区列的值将被转化为存储路径的文件夹名
create table tb_test (userid int) partitioned by (name string) row format delimited fields terminated by '\t';
--导入数据进分区，LOAD DATA信息详见LOAD DATA语句部分
load data local inpath '/home/hadoop/ziliao/test.txt' overwrite into table tb_test partition (name='XXX');
--查询分区，SELECT信息详见SELECT语句部分
select * from tb_test where name='XXX';
--显示分区
show partitions tb_test;
--从数据源表对分区插入数据，INSERT信息详见INSERT语句部分
insert overwrite table tb_test partition(name='XXX') select id from userinfo where name='XXX';
--删除分区，ALTER信息详见ALTER TABLE语句部分
alter table tb_test drop partition (name='XXX')
--添加分区，ALTER信息详见ALTER TABLE语句部分
alter table tb_test add if not exists partition (name='AAA') location '/user/hadoop/input/name=AAA';
--修改分区位置，ALTER信息详见ALTER TABLE语句部分
alter table tb_test partition (name='AAA') set location '/user/hadoop/input/name=AAA_new';
--修改分区值，ALTER信息详见ALTER TABLE语句部分
alter table tb_test partition (name='AAA') rename to partition (name='BBB');
--创建分桶表，须设置reduce tasks的数量(mapred.reduce.tasks)等于建表时分桶的数量，并且需要将分桶参数(hive.enforce.bucketing)设置为true
set mapred.reduce.tasks=3;
set hive.enforce.bucketing=true;
create table buc_test (id int, name string) clustered by(id) into 3 buckets row format delimited fields terminated by '\t';
--从数据源表插入数据至分桶表，INSERT信息详见INSERT语句部分
insert overwrite table buc_test select * from userinfo;
```

- **ALTER TABLE语句**

  ALTER TABLE语句可用来修改已有表的结构和属性，语法如下：

```
ALTER TABLE name RENAME TO new_name

ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name new_name new_type
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])

ALTER TABLE name { ADD | DROP } PARTITION (partition_spec)

ALTER TABLE name [PARTITION (partition_spec)]
  SET { FILEFORMAT format
  | LOCATION 'hdfs_path_of_directory'
  | TBLPROPERTIES (table_properties)
  | SERDEPROPERTIES (serde_properties) }

--参数具体如下
new_name ::= [new_database.]new_table_name
col_spec ::= col_name type_name
partition_spec ::= partition_col=partition_value
table_properties ::= 'name'='value'[, 'name'='value' ...]
serde_properties ::= 'name'='value'[, 'name'='value' ...]
```

  示例：

```
--表添加一列
alter table tb_person add columns (new_col int);
--添加一列并增加列字段注释
alter table tb_person add columns (new_col2 int comment 'a comment');
--修改列：原有表结构为create table test_change (a int, b int, c int);
--列a名字改为a1
alter table test_change change a a1 int;
--列a名字改为a1，类型改为string，放置在列b后，新表结构为：b int，a1 string，c int
alter table test_change change a a1 string after b;
--列b名字改为b1，放置在第一列，新表结构为：b1 int，a1 string，c int
alter table test_change change b b1 int first;
--修改表属性
--内部表转外部表
alter table table_name set tblproperties ('EXTERNAL'='TRUE');
--外部表转内部表
alter table table_name set tblproperties ('EXTERNAL'='FALSE');
--更改表名
alter table tb_stu rename to tb_stu_new;
--删除分区
alter table tb_test drop partition (name='XXX')
--添加分区
alter table tb_test add if not exists partition (name='AAA') location '/user/hadoop/input/name=AAA';
--修改分区位置 NameNode IP地址加端口加HDFS路径
alter table tb_test partition (name='AAA') set location 'hdfs://172.16.1.130:8020/user/hadoop/input/name=AAA_new';
--修改分区值
alter table tb_test partition (name='AAA') rename to partition (name='BBB');
```

- **DROP TABLE语句**

  DROP TABLE语句可用来删除已有表，语法如下：

```
DROP TABLE [IF EXISTS] [db_name.]table_name
```

  示例：

```
drop table tb_test;
--对于内部表，DROP TABLE操作会把元数据和数据文件删除掉，对于外部表，只是删除元数据，如果要删除数据，需要在HDFS上删除数据文件
dfs –rmr table_path
```

- **CREATE VIEW语句**

  视图可看做是一个复杂查询语句的缩略速记，Inceptor只支持逻辑视图，并不支持物理视图
  因此建立视图可以在MySQL元数据库中看到创建的视图表，但是在HDFS上并没有目录存储真正的视图数据值
  视图创建的同时也确定了视图的架构，即使之后基本表发生改变(如添加一列)，这一变化也将不会在视图的架构中体现
  如果基本表被删除或以不兼容的方式被修改,则该视图的查询无效。视图是只读的,不能用于LOAD/INSERT/ALTER
  创建视图的语法如下：

```
CREATE VIEW view_name [(column_list)]
  AS select_statement
```

  示例：

```
create view teacher_classsum as select teacher, count(classname) from classinfo group by teacher;
```

- **DROP VIEW语句**

  删除视图的语法如下：

```
DROP VIEW [database_name.]view_name
```

  示例：

```
drop view teacher_classnum;
```

数据类型

数据类型的定义多出现于CREATE TABLE和ALTER TABLE语句中，Inceptor支持的数据类型分两类，原生类型(primitive_type)和复杂类型

(data_type)

原生类型(primitive_type)：

- TINYINT
- SMALLINT
- INT
- BIGINT
- BOOLEAN
- FLOAT
- DOUBLE
- STRING
- INTEGER
- DATE
- DATETIME
- INTERVAL
- TIMESTAMP
- VARCHAR
- VARCHAR2
- BINARY
- DECIMAL
- DECIMAL(num, num)
- DEC(num, num)
- NUMERIC(num, num)
- NUMBER(num, num)

复杂类型(data_type)：

- arrays：ARRAY<data_type>
- maps：MAP<primitive_type, data_type>
- structs：STRUCT<col_name : data_type [COMMENT col_comment], ...>
- union：UNIONTYPE<data_type, data_type, ...>

Data Manipulation Language(DML)

Inceptor的DML语句实现了对表中数据的修改，包括 INSERT语句， 以及LOAD DATA语句。

- **INSERT语句**

INSERT语句可用于将已有表中数据导入其他表，分区或目录中，语法如下：

```
INSERT { INTO | OVERWRITE } TABLE tablename [PARTITION (partition_col1[=partition_value1][, partition_col2[=partition_value2] ...])] SELECT select_statement FROM from_statement

FROM from_statement
INSERT { INTO | OVERWRITE } TABLE tablename1 [PARTITION (partition_col1[=partition_value1][, partition_col2[=partition_value2] ...])] SELECT select_statement1 [INSERT { INTO | OVERWRITE } TABLE tablename2 [PARTITION (partition_col3[=partition_value1][, partition_col4[=partition_value2] ...])] SELECT select_statement2 ... ]

INSERT OVERWRITE [LOCAL] DIRECTORY path SELECT select_statement FROM from_statement
```

注解如下：
若使用INTO关键字，插入的数据将不覆盖添加到表中
若使用OVERWRITE关键字，插入的数据将添加到表中并覆盖表中原有数据
可对表插入，也可对分区插入数据，当对分区插入数据时，OVERWRITE关键字只作用于指定分区而不是全表
可只对一个表插入数据，也可同时对多个表插入数据
除了将数据插入表，也可将数据插入指定目录，目录既可以是本地目录也可以是HDFS上的目录
当向目录中插入数据时，LOCAL关键字代表插入本地目录，不加则代表插入HDFS上的目录

示例：

```
--将查询数据输出至HDFS目录
insert overwrite directory '/tmp/hdfs_out' select id,name from userinfo;
--将查询数据输出至本地目录
insert overwrite local directory '/tmp/local_out' select id,name from userinfo;
--将一个表的统计结果插入另一个表中
from invites a insert overwrite table events select a.bar, count(1) where a.foo > 0 group by a.bar;
insert overwrite table events select a.bar, count(1) from invites a where a.foo > 0 group by a.bar;
--多表插入
from userinfo insert overwrite table mutill select id,name insert overwrite table mutil2 select count(distinct id),name group by name;
```

- **LOAD DATA语句**

LOAD DATA语句可用于将数据导入Inceptor内部表，数据来源路径既可以为本地路径也可以为HDFS路径

语句会将指定路径的数据移动到Inceptor的数据仓库(/inceptor1/user/hive/warehouse)内，若删除内部表，数据仓库里的相应数据将被一并移除
语法如下：

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

注解如下：
若使用OVERWRITE关键字，导入的数据将添加到表中并覆盖表中原有数据，若不使用则不覆盖
可对表，也可对分区导入数据，当对分区导入数据时，OVERWRITE关键字只作用于指定分区而不是全表
若使用LOCAL关键字，则数据来源路径为本地路径，否则为HDFS路径

示例：

```
--加载本地数据入表
load data local inpath '/home/hadoop/ziliao/stu.txt' overwrite into table tb_person;
--加载HDFS数据，同时给定分区信息
load data inpath '/user/myname/stu.txt' overwrite into table tb_stu partition (ds='2008-08-15');
```

SELECT语句

SELECT语句检索一张或多张表执行查询，并返回包含多行/列的结果集。基本的SELECT语句语法如下：

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_name
[WHERE where_condition]
[GROUP BY col_list [HAVING condition]]
[ORDER BY col_list]
[LIMIT number]
```

注解如下：
WHERE条件可用来过滤结果集，支持常用比较运算符和IN,EXISTS,NOT EXISTS语句
对于SELECT语句中其它常用的关键词，下面将一一叙述。

- **WITH AS**
WITH AS在SELECT语句中为可选项，放置在SELECT语句之前，表示将AS后面查询语句的结果起一个别名，以方便后面的语句使用它。此举既可以起到优化sql的作用，又可以提高sql的可读性，语法如下：

```
—针对一个别名
WITH alias AS subquery main_query

—针对多个别名(本例三个)
WITH
  alias1 AS subquery1,
  alias2 AS subquery2,
  alias3 AS subquery3
main_query
```

示例：

```
--定义两个别名方便之后引用
with t1 as (select * from table1), t2 as (select * from table2) insert into table tbl select * from t1 union all select * from t2;
--内层子查询和外层子查询各定义一个别名
with t1 as (select * from table1) (with t2 as (select * from table2) select * from t2) union all select * from t1;
```

注：Inceptor尚未支持WITH AS的递归调用

- **DISTINCT**
DISTINCT关键词对SELECT语句的结果进行过滤以去掉所有重复项，相应的不过滤的关键词为ALL，默认值为ALL，示例如下：

```
--DISTINCT一列
select distinct c_birth_country from customer;
--DISTINCT多列时，针对所有列的组合去重
select distinct c_salutation, c_last_name from customer;
--DISTINCT也可以与COUNT等聚合函数合用
select count(distinct c_birth_country) from customer;
select count(distinct c_salutation, c_last_name) from customer;
```

- **GROUP BY**
GROUP BY语句可根据特定字段的不同值将所有行组织成不同的组，通常与聚合函数连用，对于没有参与聚合处理的列，一定要将他们放在GROUP BY的语句中。示例如下：

```
select
  ss_item_sk as item,
  count(ss_item_sk) as times_purchased,
  sum(ss_quantity) as total_quantity_purchased
from store_sales
  group by ss_item_sk;
--由于GROUP BY后不能再加where条件，因此要对聚合后的结果进行过滤，我们要使用HAVING语句对聚合函数作用的结果进行过滤
select
  ss_item_sk as item,
  count(ss_item_sk) as times_Purchased,
  sum(ss_quantity) as total_quantity_purchased
from store_sales
  group by ss_item_sk
  having times_purchased >= 100；
```

- **ORDER BY**
  ORDER BY关键词用于根据一列或多列的值将查询结果排序，对于分布式系统，ORDER BY语句会开销较大
  这是因为在排序之前，所有的结果集必须要被转移到同一节点上，而在排序完成前，将没有结果逐条输出
  因此较之没有ORDER BY的查询，含有ORDER BY的会慢些
  语法如下：

```
ORDER BY col1 [, col2 ...] [ASC | DESC]
```

  注解如下：
  ASC代表升序，DESC代表降序，默认为ASC升序

  示例：

```
--由于ORDER BY需要保证全局有序，因此执行order by的时候只能启动单个reduce，如果排序的结果集过大，那么执行时间会非常漫长
select * from test order by id;
--指定limit数量，以减少执行时间
select * from test order by id limit 10;
```

- **LIMIT**
  LIMIT关键词用于取出SELECT语句执行结果的前几条，常使用于以下场景
  1.返回查询结果的top N条记录，例如销售记录最高的10条，常与ORDER BY语句连用
  2.对没有ORDER BY的语句，可以用来展示查询结果的样本值
  3.防止因查询结果过多而导致的各种问题
  示例如下：

```
select x from numbers limit 100;
```

- **UNION**
  UNION关键词将多个语句的查询结果结合起来，包含去重(DISTINCT)和不去重(ALL)两种模式
  默认值DISTINCT模式，因此UNION等价于UNION DISTINCT
  由于去重对大量结果集是代价很高的举动，实际生产通常采用UNION ALL
  示例如下：

```
--去重
select * from (select x from few_ints union select x from few_ints) as t1；
--不去重
select x from few_ints union all select x from few_ints;
```

- **JOIN**
  连接(JOIN)在SELECT语句中有着举足轻重的地位，连接将两个表中在共同数据项上相互匹配的那些行合并起来
  Inceptor支持的连接有内连接(INNER JOIN)，隐式连接(IMPLICIT JOIN)，左外连接(LEFT OUTER JOIN)，右外连接(RIGHT OUTER JOIN)，全外连接(FULL OUTER JOIN)，左半连接(LEFT SEMI JOIN)和交叉连接(CROSS JOIN)
  语法如下：

```
--内连接，左外连接，右外连接，全外连接，左半连接
SELECT select_list FROM
  table_or_subquery1 [INNER] JOIN table_or_subquery2 |
  table_or_subquery1 {LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]} JOIN table_or_subquery2 |
  table_or_subquery1 LEFT SEMI JOIN table_or_subquery2
    [ ON col1 = col2 [AND col3 = col4 ...] |
      USING (col1 [, col2 ...]) ]
  [other_join_clause ...]
[ WHERE where_clauses ]

--隐式连接，等价于内连接
SELECT select_list FROM
  table_or_subquery1, table_or_subquery2 [, table_or_subquery3 ...]
  [other_join_clause ...]
WHERE
    col1 = col2 [AND col3 = col4 ...]

--交叉连接，对两个表做笛卡尔积，等价于不提供ON条件的内连接
SELECT select_list FROM
  table_or_subquery1 CROSS JOIN table_or_subquery2
  [other_join_clause ...]
[ WHERE where_clauses ]
```

下面详解内连接，左外连接，右外连接，全外连接和左半连接

内连接

```
--内连接使用ON条件指定每个表的列并根据其值匹配两个表中的行，其中INNER关键词为可选
select userinfo.*, choice.* from userinfo join choice on userinfo.id=choice.userid;
```

左外连接

```
--左外连接的结果集包括左表的所有行，而不仅仅是内连接所匹配的行。如果左表的某行在右表中没有匹配行，则在结果集中右表的相应列为空值
select userinfo.*, choice.* from userinfo left outer join choice on userinfo.id=choice.userid;
```

右外连接

```
--右外连接的结果集包括右表的所有行，而不仅仅是内连接所匹配的行。如果右表的某行在左表中没有匹配行，则在结果集中左表的相应列为空值
select userinfo.*, choice.* from userinfo right outer join choice on userinfo.id=choice.userid;
```

全外连接

```
--全外连接的结果集包括两表的所有行，而不仅仅是内连接所匹配的行。如果右(左)表的某行在左(右)表中没有匹配行，则在结果集中左(右)表的相应列为空值
select userinfo.*, choice.* from userinfo full outer join choice on userinfo.id=choice.userid;
```

左半连接

```
--左半连接的结果集只包括左表的特定行，这些行必须根据ON语句在右表中存在匹配的行。无论左边某行在右表中有多少匹配行，结果集中只返回一次该行
select userinfo.* from userinfo left semi join choice on userinfo.id=choice.userid;
```

内置运算符

Inceptor的内置运算符包括三类： 关系运算符，算术运算符，以及逻辑运算符。

- **关系运算符**

运算符	类型	说明
A = B	所有原生类型	如果A与B相等，返回TRUE，否则返回FALSE，注意不能使用==
A <> B	所有原生类型	如果A不等于B返回TRUE，否则返回FALSE，如果A或B值为NULL，结果返回NULL
A < B	所有原生类型	如果A小于B返回TRUE，否则返回FALSE，如果A或B值为NULL，结果返回NULL
A <= B	所有原生类型	如果A小于等于B返回TRUE，否则返回FALSE，如果A或B值为NULL，结果返回NULL
A > B	所有原生类型	如果A大于B返回TRUE，否则返回FALSE，如果A或B值为NULL，结果返回NULL
A >= B	所有原生类型	如果A大于等于B返回TRUE，否则返回FALSE，如果A或B值为NULL，结果返回NULL
A IS NULL	所有类型	如果A值为NULL，返回TRUE，否则返回FALSE
A IS NOT NULL	所有类型	如果A值不为NULL，返回TRUE，否则返回FALSE
A LIKE B	字符串	如果A或B值为NULL，结果返回NULL，字符串A与B通过sql进行匹配，如果相符返回TRUE，不符返回FALSE，B字符串中的“_”代表任意字符，“%”则代表多个任意字符。例如：(‘foobar’ like ‘foo’)返回FALSE，(‘foobar’ like ‘foo_ _ _’或者‘foobar’ like ‘foo%’)返回TURE
A RLIKE B	字符串	如果A或B值为NULL，结果返回NULL，字符串A与B通过java进行匹配，如果相符返回TRUE，不符返回FALSE。例如：(‘foobar’ rlike ‘foo’) 返回FALSE，(‘foobar’ rlike ‘^f.*r$’) 返回TRUE
A REGEXP B	字符串	与RLIKE相同

- **算术运算符**

运算符	类型	说明
A + B	所有数字类型	A和B相加，结果与操作数值有共同类型。例如每一个整数是一个浮点数，浮点数包含整数，所以一个浮点数和一个整数相加结果也是一个浮点数
A – B	所有数字类型	A和B相减，结果与操作数值有共同类型
A * B	所有数字类型	A和B相乘，结果与操作数值有共同类型。需要说明的是，如果乘法造成溢出，将选择更高的类型
A / B	所有数字类型	A和B相除，结果是一个double(双精度)类型的结果
A % B	所有数字类型	A除以B取余，结果与操作数值有共同类型
A & B	所有数字类型	运算符查看两个参数的二进制表示法的值，并执行按位“与AND”操作。两个表达式的对应位均为1时，结果的该位为1，否则，结果的该位为0
A | B	所有数字类型	运算符查看两个参数的二进制表示法的值，并执行按位“或OR”操作。只要任一表达式的对应位为1，结果的该位为1，否则，结果的该位为0
A ^ B	所有数字类型	运算符查看两个参数的二进制表示法的值，并执行按位“异或XOR”操作。当且仅当只有一个表达式的对应位上为1，结果的该位才为1，否则，结果的该位为0
~A	所有数字类型	运算符查看两个参数的二进制表示法的值，并执行按位“非NOT”操作

- **逻辑运算符**

运算符	类型	说明
A AND B	布尔值	A和B同时为TRUE时，返回TRUE，否则FALSE，如果A或B值为NULL，返回NULL
A && B	布尔值	与A AND B相同
A OR B	布尔值	A或B为TRUE，或两者同时为TRUE返回TRUE，否则FALSE，如果A和B值同时为NULL，返回NULL
A | B	布尔值	与A OR B相同
NOT A	布尔值	如果A为NULL或FALSE的时候返回TURE，否则返回FALSE
! A	布尔值	与NOT A相同

内置函数

Inceptor的内置函数包括以下种类： 数学函数， 收集函数， 类型转换函数， 日期函数， 条件函数， 字符串函数， 聚合函数， 表生成函数， 复杂类型构造函数， 以及对复杂类型操作函数。

- **数学函数**

返回类型	函数	说明
BIGINT	round(double a)	四舍五入
DOUBLE	round(double a, int d)	小数部分d位之后数字四舍五入，例如round(21.263,2)，返回21.26
BIGINT	floor(double a)	对给定数进行向下舍入最接近的整数，例如floor(21.2)，返回21
BIGINT	ceil(double a), ceiling(double a)	对给定数进行向上进入最接近的整数，例如ceil(21.2)，返回22
double	rand(), rand(int seed)	返回大于等于0且小于1的平均分布随机数(依重新计算而变)
double	exp(double a)	返回e的a次方
double	ln(double a)	返回a的自然对数
double	log10(double a)	返回a的以10为底的对数
double	log2(double a)	返回a的以2为底的对数
double	log(double base, double a)	返回a的以给定底数为底的对数
double	pow(double a, double p) power(double a, double p)	返回a的p次幂
double	sqrt(double a)	返回a的平方根
string	bin(BIGINT a)	整数转换为二进制格式，参考：http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_hex
string	hex(BIGINT a) hex(string a)	整数或字符串转换为十六进制格式，参考：http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_hex
string	unhex(string a)	十六进制字符串转换为由数字表示的字符串
string	conv(BIGINT num, int from_base, int to_base)	将指定数值，由原来的度量体系转换为指定的体系，例如conv('a',16,2)，返回'1010'，参考：http://dev.mysql.com/doc/refman/5.0/en/mathematical-functions.html#function_conv
double	abs(double a)	取绝对值
int double	pmod(int a, int b) pmod(double a, double b)	返回a除以b的余数的绝对值
double	sin(double a)	返回a的正弦值
double	asin(double a)	如果a在[-1,1]区间内，返回a的反正弦值，否则返回NULL
double	cos(double a)	返回余弦
double	acos(double a)	如果a在[-1,1]区间内，返回a的反余弦值，否则返回NULL
int double	positive(int a) positive(double a)	返回a的绝对值
int double	negative(int a) negative(double a)	返回A的绝对值的相反数

- **收集函数**

返回类型	函数	说明
int	size(Map<K,V>)	返回的Map类型的元素数量
int	size(Array<T>)	返回数组类型的元素数量

- **类型转换函数**

返回类型	函数	说明
指定转换到的类型	cast(expr as <type>)	类型转换，例如将字符"1"转换为整数：cast('1' as bigint)，如果转换失败返回NULL

- **日期函数**

返回类型	函数	说明
string	from_unixtime(bigint unixtime[, string format])	UNIX_TIMESTAMP参数表示返回一个值‘YYYY- MM – DD HH：MM：SS’或YYYYMMDDHHMMSS.uuuuuu格式，这取决于是否是在一个字符串或数字语境中使用的功能，该值表示在当前的时区
bigint	unix_timestamp()	如果不带参数的调用，返回从‘1970-01–01 00:00:00’到现在的UTC秒数)，为无符号整数
bigint	unix_timestamp(string date)	指定日期参数调用unix_timestamp，返回从‘1970-01–01 00:00:00’到指定日期的UTC秒数，为无符号整数
bigint	unix_timestamp(string date, string pattern)	指定时间输入格式，返回从‘1970-01–01 00:00:00’到指定日期的UTC秒数，为无符号整数：unix_timestamp(‘2009-03-20’, ‘yyyy-MM-dd’) = 1237532400
string	to_date(string timestamp)	返回时间中的年月日：to_date(“1970-01-01 00:00:00″) = “1970-01-01″
string	to_dates(string date)	给定一个日期date，返回一个天数(0年以来的天数)
int	year(string date)	返回指定时间的年份，范围在1000到9999，或为0
int	month(string date)	返回指定时间的月份，范围为1至12月，或为0
int	day(string date) dayofmonth(string date)	返回指定时间的日期
int	hour(string date)	返回指定时间的小时，范围为0到23
int	minute(string date)	返回指定时间的分钟，范围为0到59
int	second(string date)	返回指定时间的秒，范围为0到59
int	weekofyear(string date)	返回指定日期所在一年中的星期号，范围为0到53。
int	datediff(string enddate, string startdate)	返回两个时间参数的日期之差
int	date_add(string startdate, int days)	返回在给定时间基础上加上指定的时间段
int	date_sub(string startdate, int days)	返回在给定时间基础上减去指定的时间段

- **条件函数**

返回类型	函数	说明
T	if(booleantestCondition, T valueTrue, T valueFalseOrNull)	判断是否满足条件，如果满足返回一个值，如果不满足则返回另一个值
T	COALESCE(T v1, T v2,...)	返回一组数据中，第一个不为NULL的值，如果均为NULL,返回NULL
T	CASE a WHEN b THEN c [WHEN d THEN e]* [ELSE f] END	当a=b时,返回c；当a=d时，返回e，否则返回f
T	CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END	当条件a为TRUE时返回b,当条件c为TRUE时返回d，否则返回e

- **字符串函数**

返回类型	函数	说明
int	length(string A)	返回字符串的长度
string	reverse(string A)	返回倒序字符串
string	concat(string A, string B…)	连接多个字符串，合并为一个字符串，可以接受任意数量的输入字符串
string	concat_ws(string SEP, string A, string B…)	链接多个字符串，字符串之间以指定的分隔符SEP分开
string	substr(string A, int start) substring(string A, int start)	从文本字符串中指定的起始位置后的字符
string	substr(string A, int start, intlen) substring(string A, int start, int len)	从文本字符串中指定的位置指定长度的字符
string	upper(string A) ucase(string A)	将文本字符串转换成字母全部大写形式
string	lower(string A) lcase(string A)	将文本字符串转换成字母全部小写形式
string	trim(string A)	删除字符串两端的空格，字符之间的空格保留
string	ltrim(string A)	删除字符串左边的空格，其他的空格保留
string	rtrim(string A)	删除字符串右边的空格，其他的空格保留
string	regexp_replace(string A, string B, string C)	字符串A中的B字符被C字符替代
string	regexp_extract(string subject, string pattern, int index)	通过下标返回正则表达式指定的部分，regexp_extract(‘foothebar’, ‘foo(.*?)(bar)’, 2)返回‘bar.’
string	parse_url(string urlString, string partToExtract [, string keyToExtract])	返回URL指定的部 分，parse_url(‘http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1’, ‘HOST’)返回‘facebook.com’
string	space(int n)	返回指定数量的空格
string	repeat(string str, int n)	重复n次字符串
int	ascii(string str)	返回字符串中首字符的数字值
string	lpad(string str, intlen, string pad)	返回指定长度的字符串，给定字符串长度小于指定长度时，由指定字符从左侧填补
string	rpad(string str, intlen, string pad)	返回指定长度的字符串，给定字符串长度小于指定长度时，由指定字符从右侧填补
array	split(string str, string pat)	将字符串转换为数组
int	find_in_set(string str, string strList)	返回字符串str第一次在strlist出现的位置，如果任一参数为NULL，返回NULL；如果第一个参数包含逗号，返回0
array<array<string>>	sentences(string str, string lang, string locale)	将自然语句的每一个句子拆分为含有单词的数组，每个单词间以逗号分隔，最后返回句子的数组，lang和locale参数可选，例如sentences(‘Hello there! How are you?’)返回((“Hello”, “there”), (“How”, “are”, “you”))
array<struct<string,double>>	ngrams(array<array<string>>, int N, int K, int pf)	从一系列句子中，返回top-K N-grams，参数pf可选
array<struct<string,double>>	context_ngrams(array<array<string>>, array<string>, int K, int pf)	从一系列句子中，返回top-K contextual N-grams，参数pf可选

- **聚合函数**

返回类型	函数	说明
bigint	count(*), count(expr), count(DISTINCT expr1[, expr2,...])	返回记录条数
double	sum(col), sum(DISTINCT col)	求和
double	avg(col), avg(DISTINCT col)	求平均值
double	min(col)	返回指定列中最小值
double	max(col)	返回指定列中最大值
double	var_pop(col)	返回指定列的方差
double	var_samp(col)	返回指定列的样本方差
double	stddev_pop(col)	返回指定列的偏差
double	stddev_samp(col)	返回指定列的样本偏差
double	covar_pop(col1, col2)	两列数值协方差
double	covar_samp(col1, col2)	两列数值样本协方差
double	corr(col1, col2)	返回两列数值的相关系数
double	percentile(BIGINT col, p)	返回数值区域的百分比数值点，0<=P<=1，否则返回NULL，不支持浮点型数值
array	percentile(BIGINT col, array(p1 [, p2]...))	返回数值区域的一组百分比值分别对应的数值点，0<=P<=1，否则返回NULL，不支持浮点型数值
double	percentile_approx(DOUBLE col, p [, B])	返回数值区域的百分比数值点的近似值，0<=P<=1，否则返回NULL，支持浮点型数值。参数B设置精确度，越高的B值精确度越高，缺省值为10000。当列中distinct值的数目小于B时，返回准确值
array	percentile_approx(col, array(p~1,, [, p,,2_]…) [, B])	与上个方法相似，除了接受与返回一组而不是一个百分比数值点
array<struct{‘x’,‘y’}>	histogram_numeric(col, b)	计算一列的histogram，使用b non-uniformly spaced bins，返回长度为d的数组，存有双组分的结构类型，双组分分别表示bin的中心和高度
array	collect_set(col)	返回无重复记录

- **表生成函数**

  普通(非聚合)函数输入一行得到一行输出，表生成函数输入一行得到多行输出

返回类型	函数	说明
多行	explode(ARRAY)	返回多行，每一行包含数组中的一个值
多行	explode(MAP)	返回多行，每一行包含两列，分别对应MAP中一个entry的key值和value值
tuple	json_tuple(jsonStr, k1, k2, ...)	输入一个JSON string和一系列key值，返回value的tuple，由于可以一次输入多个key值，这是get_json_object的高效版
tuple	parse_url_tuple(url, p1, p2, ...)	与parse_url()相似，但可一次调用即得到URL的多个部分，可用的part值有：HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY:<KEY>

  使用表生成函数有一些限制：
  1.SELECT中不可包含其他表达式：不支持SELECT pageid, explode(adid_list) AS myCol...
  2.表生成函数不能套叠：不支持SELECT explode(explode(adid_list)) AS myCol...
  3.不能使用GROUP BY/CLUSTER BY/DISTRIBUTE BY/SORT BY：不支持SELECT explode(adid_list) AS myCol...GROUP BY myCol

- **复杂类型构造函数**

函数	输入类型	说明
map	(key1, value1, key2, value2,...)	通过指定的键/值对，创建一个map
struct	(val1, val2, val3, ...)	通过指定的字段值，创建一个结构，结构字段名称将为COL1，COL2，...
named_struct	(name1, val1, name2, val2,...)	通过指定的字段值，创建一个结构，结构字段名称将为指定的name1，name2，...
array	(val1, val2, ...)	通过指定的元素，创建一个数组

- **对复杂类型操作函数**

函数	输入类型	说明
A[n]	A是一个数组，n为int型	返回数组A的第n个元素，第一个元素的索引为0，如果A数组为['foo','bar']，则A[0]返回‘foo’，A[1]返回‘bar’
M[key]	M是Map<K, V>，key为关键值类型	返回关键key值对应的value值，例如map M为{‘f’ -> ‘foo’, ‘b’ -> ‘bar’, ‘all’ -> ‘foobar’}，则M[‘all’]返回‘foobar’
S.x	S为结构类型	返回结构S的的x字符串存储值，如foobar为{int foo, int bar}，那么foobar.foo返回的foo中存储的int

用户自定义函数

Inceptor支持用户自定义函数，本节将首先举例说明如何自定义函数并应用在Inceptor里，之后将列举Inceptor自增的UDF

- **创建自定义函数(UDF)**
  自定义UDF时需要注意以下几点：
  1.自定义UDF需要继承org.apache.hadoop.hive.ql.exec.UDF
  2.需要实现evaluate函数
  3.evaluate函数支持重载

  以下代码实现了两个数求和的UDF，evaluate函数代表两个整型数据相加或两个浮点型数据相加

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

public final class Add extends UDF {
  public Integer evaluate(Integer a, Integer b) {
    if (null == a || null == b) {
      return null;
    }
    return a + b;
  }

  public Double evaluate(Double a, Double b) {
    if (a == null || b == null)
      return null;
    return a + b;
  }
}
```

  实际使用时，需要执行以下几步：
  1.程序打包放到目标机器上进入Inceptor客户端，添加jar包：add jar [jar_file_path];
  2.创建临时函数：CREATE TEMPORARY FUNCTION my_add AS 'inceptor.udf.Add';
  3.使用临时函数：SELECT my_add(scores.math, scores.art) FROM scores;
  4.使用完成后销毁临时函数：DROP TEMPORARY FUNCTION my_add;

- **创建自定义聚合函数(UDAF)**
  自定义UDAF时需要注意以下几点：
  1.自定义UDAF需要继承org.apache.hadoop.hive.ql.exec.UDAF
  2.内部类Evaluator需要实现UDAFEvaluator接口：Evaluator需要实现init、iterate、terminatePartial、merge、terminate这几个函数
  2.1.init函数实现接口UDAFEvaluator的init函数
  2.2.iterate接收传入的参数，并进行内部的轮转，其返回类型为boolean
  2.3.terminatePartial无参数，其为iterate函数轮转结束后，返回轮转数据，terminatePartial类似于hadoop的Combiner
  2.4.merge接收terminatePartial的返回结果，进行数据merge操作，其返回类型为boolean
  2.5.terminate返回最终的聚集函数结果

  以下代码实现了一个求平均数的UDAF

```
package inceptor.udaf;
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

public class Avg extends UDAF {
  public static class AvgState {
    private long mCount;
    private double mSum;
  }

  public static class AvgEvaluator implements UDAFEvaluator {
    AvgState state;

    public AvgEvaluator() {
      super();
      state = new AvgState();
      init();
    }

    public void init() {
      state.mSum = 0;
      state.mCount = 0;
    }

    public boolean iterate(Double o) {
      if (o != null) {
        state.mSum += o;
        state.mCount++;
      }
      return true;
    }

    public AvgState terminatePartial() {
      return state.mCount == 0 ? null : state;
    }

    public boolean merge(AvgState o) {
      if (o != null) {
        state.mCount += o.mCount;
        state.mSum += o.mSum;
      }
      return true;
    }

    public Double terminate() {
      return state.mCount == 0 ? null : Double.valueOf(state.mSum
          / state.mCount);
    }
  }
}
```

实际使用时，需要执行以下几步：
1.程序打包放到目标机器上进入Inceptor客户端，添加jar包：add jar [jar_file_path];
2.创建临时函数：CREATE TEMPORARY FUNCTION my_avg AS 'inceptor.udaf.Avg';
3.使用临时函数：SELECT scores.art, my_avg(scores.math) FROM scores GROUP BY scores.art;
4.使用完成后销毁临时函数：DROP TEMPORARY FUNCTION my_avg;

下面列举Inceptor自增的UDF

- **trunc**

```
trunc(date, format)：支持string类型的date字段按format返回特定的日期string
trunc(number,num_digits)：支持数字类型的按给定数值进行裁剪
```

- **to_char**

```
to_char(date, pattern)：把yyyy-MM-dd HH:mm:ss形式的string转化为输入pattern形式的string
to_char(datetime, pattern)：把yyyy-MM-dd形式的string转化为输入pattern形式的string
to_char(number [,format])：把数值转化为string，基本类型（byte, short, int, bigint, float, double）转string或按照一定format转string
```

- **translate**

```
translate('input_string','from_string','to_string')
```

translate方法作用于input string（第一个参数）的字符上，规则为出现在from_string中的字符用to_string中对应位置的字符来替换。如果to_string的长度比from_string短，那么from_string中多出来的位置的字符会在output中被移除。

translate('abcdef', 'adc', '19')：返回 '1b9ef'，'1'代替'a','9'代替'd'，'c'被移除。

- **nvl**

nvl(eExpr1, eExpr2)：eExpr1为空时设为eExpr2的值，仅支持基本类型（包括string）

- **nvl2**

nvl2(eExpr1, eExpr2, eExpr3)：eExpr1不为空时设置为eExpr2的值，为空时设置为eExpr3的值，仅支持基本类型（包括string）

- **dense_rank**

dense_rank(hash_key, order_by_col1, order_by_col2 ...)：dense_rank与rank的区别是dense_rank的排序号会持续+1递增的，不会由于部分字段相同排序号而跳跃，即按照1、2、2、2、3、4、5排序
select key1,key2,value,dense_rank(1,value) as rank from (select key1,key2,value from test_table distribute by hash(key1) sort by key1,key2,value) t;

- **greatest**

greatest(value1, value2, value3, ...)：字段属性必须为PRIMITIVE，且需要是相同的TypeInfo，或者能转成CommonBase

- **group_max**

select key1,key2,value,group_max(key1,key2) as rank from (select key1,key2,value from test_table sort by key1) t;

NOTE：按照key1 group by之后对key2取其中的最大值，此方法不属于聚合函数，所以是逐行处理的。由于每个group内当前行记录只是基于其之前所有行的统计结果（而不是扫描group内所有行），所以每个group内前面部分行记录的最大值可能不是group真正的最大值：

```
1   12   string1   12
1   2    string1   12
1   11   string1   12
1   12   string2   12
2   NULL string1   12
2   -21  string1   -21
2   23   string1   23
2   4    string2   23
2   14   string2   23
3   32   string2   32
3   6    null      32
4   8              8
4   42   string2   42
5   54   string2   54
5   52   string1   54
5   NULL value5    54
6   12   502       12
6   64   string2   64
```

类似的函数还有group_min和group_sum

- **instr**

instr(str, substr)：查找子字符串index

- **lag**

lag(hash_key,column[,offset[,default]])：用法和group_max等类似依赖于sort_by。此函数和oracle原生语法相同，取出当前行前面第N行的数据，其不支持取出后N行数据，暂时也没有其他类似方法提供该功能

注意事项如下：
1.column与default值需要是同一字段类型
2.offset只能为自然数，其默认值为1，即取前一行的字段数据，如果没有则为NULL，如果设为负数将全为NULL

范例：

select key1,key2,value,lag(key1,key2) as rank from (select key1,key2,value from test_table distribute by key1 sort by key1,value) t

```
1   11   string1   NULL
1   2    string1   11
1   12   string1   2
1   12   string2   12
2   NULL  string1   NULL
2   -21   string1   NULL
2   23   string1   -21
2   14   string2   23
2   4    string2   14
3   6    null      NULL
3   32   string2   6
4   8              NULL
4   42   string2   8
5   52   string1   NULL
5   54   string2   52
5   NULL  value5    54
6   12   502       NULL
6   64   string2   12
```

- **lnnvl**

```
lnnvl(condition)：当条件为true时返回false
```

- **rank**

```
rank(hash_key, order_by_col1, order_by_col2 ...)：与dense_rank类似，但是其会以1、2、2、2、5、6、7排序
```

- **row_number**

```
row_number(hashkey)
select key1,key2,value,row_number(key1) as rank from (select key1,key2,value from test_table distribute by key1 sort by key1) t;
```

此函数同样不属于聚合函数，可看做rank的一个变种，给出1、2、3、4的标准行号，因此不需要比较部分字段是否相同以停留排序号。其依然依赖于distribute by和sort by之后的group结果，查询结果如下：

```
1   12   string1   1
1   2    string1   2
1   11   string1   3
1   12   string2   4
2   NULL  string1   1
2   -21   string1   2
2   23   string1   3
2   14   string2   4
2   4    string2   5
3   32   string2   1
3   6    null      2
4   8              1
4   42   string2   2
5   54   string2   1
5   52   string1   2
5   NULL  value5    3
6   64   string2   1
6   12   502       2
```

- **to_number**

```
to_number(value, format_mask)：string类型转换为number，按指定的format
```

NOTE：1.value必须为string类型
2.无参数时转为整数
3.format_mask必须为double或者字符串（字符串必须为带“$”且value也带“$”，否则抛出异常并返回NULL结果，但可以执行完成）
4.format_mask必须全由9组成，否则将返回空
5.当format_mask为double时，不会影响其结果的小数点位数（最多给整数加上.0）

- **chr**

```
chr(number_code)：只能传入int，转为String.valueof((char) number_code)，字符编码格式估计为内置的unicode
```

- **str_to_date**

```
str_to_date(dateText,pattern)：将输入的string类型的date按照pattern转为string类型的date，但是一定要以“yyyy-MM-dd HH:mm:ss”格式返回最终结果
```

- **date_format**

  date_format(dateText,pattern)：将输入的string类型的date按照pattern转为string类型的date，以pattern格式返回最终结果，这是其与str_to_date的区别

创建列式存储

Inceptor实现了分布式内存列式存储(Holodesk)，Inceptor可以将数据从磁盘中加载进内存，可以基于内存中进行运算。内存数据表的数据源可由内部表或外部表提供。

- 内部表的创建由 CREATE TABLE 来完成，之后可由 LOAD DATA LOCAL INPATH [数据源路径] 来导入本地文件系统中的数据，如果不用LOCAL关键字，也可导入HDFS中的数据：

  ```
  -- 创建表并指定字段的分隔符
  create table 表名 (列1 列1类型，列2 列2类型...) row format delimited fields terminated by ',';
  -- 导入数据
  load data local inpath '路径' overwrite into table 表名；
  ```

  若创建分区表，可在上述指令基础上加入 partitioned by（分区字段，数据类型） 关键字，加载数据时按照 partition（分区字段=分区值k） 关键字为第K个分区插入值。

- 外部表的创建由 CREATE EXTERNAL TABLE 来完成，在建表的同时指定一个指向实际数据的路径（LOCATION）：

  ```
  -- 创建表并指向实际数据路径
  create external table 表名 (列1 列1类型，列2 列2类型...) row format delimited fields terminated by ',' location '数据实际路径';
  ```

下面可由数据源表创建Inceptor内存数据表：

- 创建内存数据表并由数据源表添加数据

  ```
  -- [内存数据库表]为用户自定义的表名，与Inceptor中已有的表名区分开即可。
  CREATE TABLE [内存数据库表] TBLPROPERTIES("cache"="ram" ,"cache.checkpoint"="false",<"filters"=[过滤器类型]>,[其它表属性]) AS
  SELECT [列1,列2,.....]
  FROM [数据源表]
  WHERE [条件]
  DISRIBUTE BY|GROUP BY|CLUSTER BY [条件]
  ```

  cache=ram表示该表存放在内存中，同时可支持的选项有SSD，Memory等，当有服务器上配置有SSD设备时可以指定cache=SSD，Inceptor会自动利用SSD对SQL加速。
  cache.checkpoint：是否对内存数据表做CheckPoint，可选值有true和false，该值表示在创建内存表时，是否需要同步将数据放入HDFS中。
  这样在未来运行计算任务时，如果集群中有机器当机时，则当机机器上存放的内存数据可以从HDFS中直接读取恢复，否则当机机器中内存的数据会通过重新计算的方式恢复，恢复时间上慢于从HDFS中恢复。
  filters（过滤器）：为可选参数，如果加载进入内存的数据分布满足某些特性，则可以在该表创建时指定对应的过滤器，在某些应用场景下，SQL语句的执行性能会有明显提升，下文会详细解释过滤器的创建和使用规则。

- 由数据源表向已存在的内存数据表添加数据

  ```
  -- [内存数据库表]为已在内存中构建好的一张内存数据库表
  INSERT INTO TABLE [内存数据库表] SELECT [列1,列2,.....]
  FROM [数据源表]
  WHERE [条件]
  DISRIBUTE BY|GROUP BY|CLUSTER BY [条件]
  ```

实例-创建Inceptor内存分区分桶表：

- 创建多重动态分区表，需要保证作为数据源的外表也是分好区的，无论是硬盘表还是内存表，创建后查询均可以使用/*+combine(字段)*/ hint

  ```
  --先创建分区外表作为数据源
  create external table par_ext (date string) partitioned by (id int, value string) location '/user/test/par_ext';
  set hive.exec.dynamic.partition=true;
  set hive.exec.dynamic.partition.mode=nonstrick;
  set hive.exec.max.dynamic.partitions.pernode=10000;
  set hive.exec.max.dynamic.partitions=10000;
  insert overwrite table par_ext partition(id,value) select date,id,value from simple_table;
  --创建多重动态分区内存表
  create table par_mem (date string) partitioned by (id int, value string) tblproperties('cache'='ram');
  insert overwrite table par_mem partition(id, value) select date, id, value from par_ext;
  ```

- 创建动态分区+分桶表，需要保证作为数据源的外表也是分好区和桶的，无论是硬盘表还是内存表，创建后查询均可以使用/*+combine(字段)*/ hint

```
--先创建分区分桶外表作为数据源
set mapred.reduce.tasks=7;
set hive.enforce.bucketing=true;
create external table parbuc_ext (date string, id int) partitioned by (value string) clustered by (id) into 7 buckets location '/user/test/parbuc_ext';
set hive.exec.dynamic.partition=true;
set hive.exec.dynamic.partition.mode=nonstrick;
set hive.exec.max.dynamic.partitions.pernode=10000;
set hive.exec.max.dynamic.partitions=10000;
insert overwrite table parbuc_ext partition(value) select date,id,value from simple_table;
--创建动态分区+分桶内存表
set mapred.reduce.tasks=7;
create table parbuc_mem (date string, id int) partitioned by (value string) clustered by (id) into 7 buckets tblproperties('cache'='ram');
insert overwrite table parbuc_mem partition(value) select date, id, value from parbuc_ext;
```

- 创建分桶表，作为数据源的外表无需一定分好桶，内存表创建后查询可以使用/*+combine(字段)*/ hint

```
--创建分桶内存表
set mapred.reduce.tasks=7;
create table buc_mem (date string, id int, value string) clustered by (id) into 7 buckets tblproperties('cache'='ram');
insert overwrite table buc_mem select date, id, value from simple_table;
```

高级功能

本小节叙述了几个Inceptor内存数据库高级功能：MAPJOIN，在内存表中设置过滤器（FILTER）以及利用过滤器优化SQL性能。

MAPJOIN

在Inceptor中，如果遇到A（大表）JOIN B（小表)的情况，可以使用SQL的HINT `/*+MAPJOIN(B)*/` 完成本次JOIN操作。在实际的案例中小表一搬指的是小于100M的数据表。

使用MAPJOIN加速的原理是：MAPJOIN会把小表B全部读入内存中，在map阶段直接拿大表A的数据和内存中的表数据做匹配，即在map中进行了join操作，省去了reduce的运行效率会高很多。

```
-- A表和B表为任意Inceptor表，B为小表，可以为内存表或者是Inceptor中存放在磁盘上的表
SELECT /*+MAPJOIN(B)*/ *
FROM A JOIN B
ON A.JOINKEY=B.JOINKEY
WHERE [条件]
```

在内存表中设置过滤器（FILTER）

过滤器是建立在**内存表**的基础上，用来标识内存表中数据分布特性的组件。当GROUP BY或者JOIN的键值包含了已经按特定规则分布过的字段时，Inceptor会用过滤器来减少数据检索的范围以自动优化计算过程，避免在运算时进行数据迁移，从而极大提升GROUP BY和JOIN的性能。

Inceptor已经实现了三种FILTER：

- 单值过滤器（Unique Value Filter）：标识在一个数据BLOCK（MAP任务的数据切割单元）中，对应列（COLUMN）中所有的值相等。
- 哈希桶过滤器（Hash Bucket Filter）：标识在一个数据BLOCK（MAP任务的数据切割单元）中，对应列（COLUMN）中所有的值的哈希值对桶大小取模后值相等。
- 布隆过滤器（Bloom Filter）：对一个数据BLOCK（MAP任务的数据切割单元）中创建布隆过滤器，可以对全表中所有的列创建布隆过滤器。

内存表中过滤器的特性来源有两种:数据源中数据分布满足过滤器特性以及数据在加载进入内存表中的同时重新分布。

数据源中数据分布满足过滤器特性

- 单值过滤器（Unique Value Filter）

```
--创建Inceptor分区表
CREATE TABLE A (其它字段) PARTITIONED BY (分区字段)

--向刚创建的Inceptor分区表中按分区导入数据，使其数据分布满足单值过滤器特性
--由于是对分区插入值，下面语句中若将INTO改为OVERWRITE，覆盖的只是分区内的值，因此也可以实现对表按分区插入值
FROM [数据源表]
INSERT INTO TABLE A PARTITION(分区字段='分区值1')
   SELECT 其它字段 WHERE 分区字段='分区值1'
INSERT INTO TABLE A PARTITION(分区字段='分区值2')
   SELECT 其它字段 WHERE 分区字段='分区值2'
INSERT INTO TABLE A PARTITION(分区字段='分区值3')
   SELECT 其它字段 WHERE 分区字段='分区值3'
.....

--将分区表作为数据源加载进内存表
CREATE TABLE C TBLPROPERTIES("cache"="ram","filters"="uniquevalue:分区字段") AS
   SELECT 字段 FROM A
```

- 哈希桶过滤器（Hash Bucket Filter）

```
--设置桶数量，如果不设置，默认有一个桶
set mapred.reduce.tasks = 桶数量

--创建Inceptor分桶表
CREATE TABLE A (字段) CLUSTERED BY (分桶字段) INTO 桶数量 BUCKETS

--向刚创建的Inceptor分桶表中导入数据，使其数据分布满足哈希桶过滤器特性
INSERT OVERWRITE TABLE A SELECT 字段 FROM 数据源表 DISTRIBUTE BY 分桶字段

--将分桶表作为数据源加载进内存表
CREATE TABLE C TBLPROPERTIES("cache"="ram","filters"="hashbucket(桶数量):分桶字段") AS
   SELECT 字段 FROM A
```

数据在加载进入内存表中的同时重新分布

- 单值过滤器（Unique Value Filter）

```
--创建内存表，并从数据源直接加载一个分区的数据进内存
--此处若只创建内存表而不一并添加第一个分区数据，需要在tblproperties里面再加一条"columnar.store"="false"，否则用insert插入值值后查询会报错
CREATE TABLE C TBLPROPERTIES("cache"="ram","filters"="uniquevalue:分区字段") AS
SELECT 字段 FROM 数据源表 WHERE 分区字段=分区值1

--向内存表中加载第二个分区的数据
INSERT INTO TABLE C SELECT 字段 FROM 数据源表 WHERE 分区字段=分区值2
INSERT INTO TABLE C SELECT 字段 FROM 数据源表 WHERE 分区字段=分区值3
.....
```

- 哈希桶过滤器（Hash Bucket Filter）

```
--设置桶数量，如果不设置，默认有一个桶
set mapred.reduce.tasks = [桶数量]
--从Inceptor数据源表直接加载数据进内存
CREATE TABLE C TBLPROPERTIES("cache"="ram","filters"="hashbucket(桶数量):分桶字段") AS
   SELECT 字段 FROM 数据源表 DISTRIBUTE BY 分桶字段
```

利用过滤器优化SQL性能

- 利用过滤器优化扫描性能

```
--C表为已经创建好过滤器的内存表，优化器会根据不同的过滤器和操作符的类型（等值，非等值）进行优化，缩小数据搜索范围
SELECT [字段] FROM C WHERE [创建好FILTER的字段] [操作符] [值]
```

- 利用过滤器优化GROUP BY性能

```
--C表为已经创建好过滤器的内存表，并且在字段KEY上设置了单值或哈希桶过滤器
--优化与聚合字段顺序无关
SELECT /*+combine(KEY)*/ [字段或聚合函数]
FROM C
WHERE [条件]
GROUP BY KEY,[其余聚合字段]
```

- 利用过滤器优化JOIN性能

```
--C表为已经创建好过滤器的内存表并且在字段KEY_C上设置了哈希桶过滤器
--D表为已经创建好过滤器的内存表并且在字段KEY_D上设置了哈希桶过滤器
SELECT /*+combine(KEY_C, KEY_D)*/ [字段]
FROM C JOIN D
ON C.KEY_C = D.KEY_D AND [其他JOIN条件]
WHERE [条件]
```

- 利用过滤器优化子查询性能

```
--C表为已经创建好过滤器的内存表并且在字段KEY_C上设置了哈希桶过滤器，当子查询中不改变KEY_C的值分布情况时，KEY_C的分布会一直保持下去，那么在进行下一步的GROUP BY/JOIN时，Inceptor依然不会进行数据迁移
SELECT[字段]FROM
   (SELECT /*+combine(KEY_C)*/ KEY_C, [其它聚合字段], [聚合函数] FROM C GROUP BY KEY_C,[其它聚合字段]) C_1
   JOIN
   (SELECT KEY_C, [其它聚合字段], [聚合函数] FROM (SELECT /*+combine(KEY_C)*/ KEY_C, [其它聚合字段], [聚合函数] FROM C GROUP BY KEY_C,[其它聚合字段]) C_2 GROUP BY KEY_C, [其它聚合字段]) C_3
ON C_1.KEY_C = C_3.KEY_C
```

Inceptor for Hyperbase

Inceptor支持创建与Hyperbase表相关联的外表，利用索引技术加快SQL运行速度。

SQL语法

```
--[inceptor_table]为用户自定义的表名，与Inceptor中已有的表名区分开即可。
set hyperbase.reader=true;
SELECT /*+struct_index(1)*/ [列1,列2,.....] FROM [inceptor_table]<WHERE [条件]><GROUP BY [列1,……]><HAVING [条件]>
SELECT /*+struct_index(1)*/ [列1,列2,.....]FROM [inceptor_table] A JOIN [数据源表] B ON [条件] <WHERE [条件]>
```

`set hyperbase.reader=true;`表示使用Hyperbase新特性，加快数据的读取速度。
`/*+struct_index(1)*/`表示当Hyperbase中主键在Inceptor表中被映射为结构体时，使用结构体的第一个字段作为切分依据，即感知语意的切分。当主键未被映射为结构体时，此值不需要设置（默认取1），表示将主键整体作为索引字段。

注：当`hyperbase.reader`设置为`true`时，建议对对应的Hyperbase表进行flush操作，可通过Java或Hbase Shell操作：

```
HBaseAdmin hAdmin = new HBaseAdmin(config);
try{
  hAdmin.flush(tableName);
  hAdmin.close();
}catch (InterruptedException e) {
  e.printStackTrace();
}
```

亦可在Hbase Shell中设置：

```
flush '[hbase_table_name]'
```

使用举例

首先创建一张Inceptor表关联Hyperbase中的表。

```
--[inceptor_table]为用户自定义的表名，与Inceptor中已有的表名区分开即可。
--[hyperbase_table]为Hyperbase的表名。
CREATE EXTERNAL TABLE [inceptor_table] (key struct<key1:string,key2:string,key3:string>, value string) ROW FORMAT DELIMITED COLLECTION ITEMS TERMINATED
BY '~' STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH SERDEPROPERTIES('hbase.columns.mapping'=':key,f:q')
TBLPROPERTIES('hbase.table.name='[hyperbase_table]');
```

Inceptor为Hyperbase中的表hyperbase_table创建了一张外表
hbase.columns.mapping表示Inceptor表中key、value分别与hyperbase_table的key、f:q相对应。其中，hyperbase_table的主键在Inceptor表中被映射成一个结构体struct(使用~进行分割)，分为三段(key1,key2,key3)，hyperbase_table的f:q对应Inceptor表中的value字段。
hbase.table.name表示hyperbase中的表名。

Group by

```
SELECT /*+struct_index(1)*/ * FROM [table] GROUP BY key.key1 <WHERE [条件]>;
```

当使用Group by操作时，需要指定切分方式。此处struct_index内部使用参数设置为1。

Join

```
SELECT /*+struct_index(2)*/ * FROM [table] A JOIN [table] B on A.key.key1=B.key.key1 and A.key.key2=B.key.key2 <WHERE [条件]>;
```

当使用Join操作时，也可以指定切分方式。此处struct_index可以设置为1或2，根据数据量以及数据特性可以起到优化效果。

Sqoop

Sqoop是一个用来将Hadoop和关系型数据库（RDBMS）中的数据相互转移的工具，本章节介绍了将一个关系型数据库（例如: MySQL，Oracle等）中的数据导入到HDFS中的方法。

从RDBMS导入数据到HDFS

- **连接数据库服务器**
  从数据库导入数据至HDFS，须要用`--connect`指定连接串，此连接串与URL相似，同时需要指定安全连接所用的用户名和密码：

  ```
  #连接至mysql数据库
  sqoop import --connect jdbc:mysql://[数据库IP]:[PORT]/[数据库SID] --username 用户名 --password 密码
  #连接至oracle数据库
  sqoop import --connect jdbc:oracle:thin:@[数据库IP]:[PORT]:[数据库SID] --username 用户名 --password 密码
  ```

  参数解释及注意事项：

  - [数据库IP]:[PORT]：数据库所处地址URL，如果要在分布式集群中使用Sqoop, 那么不要使用主机名作为URL
  - [数据库SID]：数据库实例名
  - 用户名：要使用大写，否则会报错

  Sqoop支持多个数据库，但有时需要启动相应的driver以连接RDBMS数据库，相应的驱动jar包可在`/usr/lib/sqoop/lib`目录下找到。使用Sqoop的`--driver`命令来连接至相应的数据库。

- **导入数据**
  使用`--table`来选择导入RDBMS表，表的名字要大写，否则会报错。Sqoop默认导入所选的表中所有的列，并在HDFS中按顺序排列。

  可以用`--columns`来实现导入部分列，每一列的名字用逗号隔开。

  可以用`--where`来实现导入某些行。

  用`--target-dir`指定HDFS目标存储路径。

  在HDFS中数据默认存储为textfile格式，可以用`--fields-terminated-by`指定分隔符，建议采用chr(1)作为分隔符，chr(1)表示ASCII码01，SOH(start of headline)标题开始。该字符没有意义在RDBMS中基本不会出现，出现了也属于脏数据。

  可以用`-m 1`来进行单任务导入；当需要并行导入数据，需要用`--split-by`来指定切分列，通常选用数据range比较离散的列作为切分列以加速数据导入。

  ```
  #单任务导入
  sqoop import --connect [连接串] --username 用户名 --password 密码 --table [RDBMS表名] --columns "列1[, 列2[,...]]" --where [条件] --target-dir [HDFS路径] --fields-terminated-by '分割符' --m 1
  #并行导入
  sqoop import --connect [连接串] --username 用户名 --password 密码 --table [RDBMS表名] --columns "列1[, 列2[,...]]" --where [条件] --target-dir [HDFS路径] --fields-terminated-by '分割符' --m 4 --split-by [拆分字段]
  ```

- **SQL清洗**
  在某些特定情况下，脏数据会出现。因此在导入之前需要对数据进行预清洗，可使用`--query`来实现清洗并导入（即使不进行数据预清洗，也可使用--query进行数据导入）

  ```
  sqoop import --connect [连接串] --username 用户名 --password 密码 --target-dir [HDFS路径] --query [query_SQL] --fields-terminated-by '分割符' --m 1
  ```

  [query_SQL]的格式通常为"select $QUERY from [表名] where [条件]"，其中`QUERY`中包含了数据清洗的语句，下面举例详叙如何生成`QUERY`语句：

```
#转换清洗函数叫做dt(Dynamic Translation)，初始的清洗语句为空
dt_res=""
#转换清洗函数dt：对于非时间类型字段都需要做一次预清洗，将NULL转为空，将无意义字符chr(1)转为空，换行符chr(10)转为空格，回车字符chr(13)转为空格；时间类型字段由sqoop自行转换
#实际使用中，运行指令如下：dt [column1_hadoop] [column1_RDB] [column2_RDB] ...
#这表示将原RDB中的一个或多个columns清洗后合并在一起并对应到hadoop的一个column，实际应用中，通常建议的操作方法是将所有RDB的columns都对应到hadoop的一个column，之后再用分隔符"\\01"（即chr(1)）建外表
function dt{
  ALIAS=$1
  shift
  dt_res="("
  for var in $@
  do
    isdate=false
    for dtime in $DATE_TIME
    do
      if [ "$dtime" == "$var" ]; then
        isdate=true
        break
      fi
    done
    if [ "$isdate" == "false" ]; then
      CASEWHEN="CASE WHEN $var IS NOT NULL THEN REPLACE(REPLACE(REPLACE(CAST($var AS VARCHAR(6000)),chr(1),''),chr(13),' '),chr(10),' ') ELSE ''
END "
    else
      CASEWHEN="$var "
    fi
    if [ "$dt_res" == "(" ]; then
      dt_res=$dt_res"$CASEWHEN"
    else
      dt_res=$dt_res"||chr(1)||$CASEWHEN"
    fi
  done
  dt_res=$dt_res") as $ALIAS"
}
#初始清洗QUERY为空
QUERY=""
#执行清洗函数dt并将结果放入dt_res，下面语句是将所有RDB的columns都对应到hadoop的一个column，因此只需要执行一次dt
dt [column1_hadoop] [column1_RDB] [column2_RDB] … [columnN_RDB]
QUERY=$QUERY$dt_res
#若需要其它对应，需要执行多次dt的时候，每次执行dt后可用下列语句将结果append到QUERY
QUERY=$QUERY","$dt_res
```

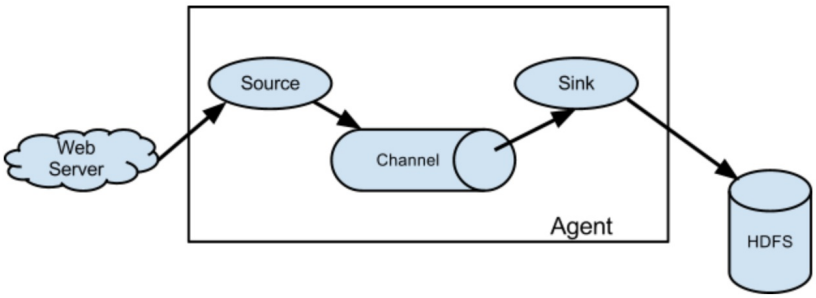
Flume

Flume提供了收集，聚集和转移海量日志文件的连接器。同时，Flume使用ZooKeeper来确保配置的一致性和可用性。

Flume具有如下特点：

- 高可用性：Flume提供了端到端的可靠传输盒本地数据存储选项。
- 可管理性：通过ZooKeeper保证配置的可用性并使用多个master来管理所有节点。
- 拓展性：用户可用java语言实现新的方法。

本处使用的Flume版本为Flume1.2+，即Flume-ng，它的抽象结构可用下图表示：



其中：
Event：传输的单位数据流
Agent：一个JVM进程，管理从外部数据源进入的event至下一站，每个agent由source，channel，sink三部分组成。Flume-ng可以由多个agent组成。
Source：要接受的数据来源
Sink：对应于每个source，是flume-ng的结果路径
Channel：Source与Sink之间的通道，flume-ng通过channel来传输数据。

搭建 Flume-ng

搭建agent

Flume agent的配置储存在本地配置文件。一个配置文件中可以储存多个agent的信息，此文件中包括了每一个agent的source，sink和channel的属性以及他们的连接方式。

以下为一个配置文件的基本格式：

```
# 定义agent的source，sink和channels
[Agent].sources = [Source]
[Agent].sinks = [Sink]
[Agent].channels = [Channel1] [Channel2]
# 设置source的channel
[Agent].sources.[Source].channels = [Channel1] [Channel2] ...
# 设置sink的channel
[Agent].sinks.[Sink].channel = [Channel1]
```

配置单个组件

每一个组件（source，sink，channel）都有名称（name），类型（type），以及特定的属性。

比如，一个 Avro源需要hostname（或者IP 地址），以及接收数据的端口号； 一个内存通道（memory channel）可以包含最大的队列大小（“capacity”）； 而一个HDFS sink需要文件文件系统的URL，创建文件路径，以及创建频率（“hdfs.rollInterval”）等。

以下为单个组件的基本格式：

```
#source的配置属性
[Agent].sources.[Source].[someProperty] = [someValue]
#channels的配置属性
[Agent].channel.[Channel].[someProperty] = [someValue]
#sinks的配置属性
[Agent].sources.[Sink].[someProperty] = [someValue]
```

启动 agent

agent由flume安装目录中bin/下的flume-ng脚本启动，用户需要指定agent名称，配置目录，以及配置文件

```
$ bin/flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties
```

TranswarpR 指南

本章节介绍了TranswarpR的相关配置及使用，内容包括：简介，RStudio的配置及使用，TranswarpR组件使用说明，TranswarpR算法使用案例以及函数参考手册。

简介

- R简介

  R和Matlab很类似，除了包含一种广为使用的统计分析语言之外，还提供了强大的绘图、操作环境等功能。R的使用过程和Matlab也很类似，除了提供在命令行下交互操作的模式外，用户还可以将一些R程序或命令写到文本文件里，称为R脚本，编译连续执行。

- Rstudio简介

  RStudio是R的一种强大而便捷的IDE，把常用的窗口都整合在一起，是一个集成化的图形化开发环境，除了在各种系统中有桌面版本，还可以以云服务的形式构建在Linux服务器上，通过远程网页登陆访问。

- TranswarpR简介

  TranswarpR是Transwarp Data Hub中的一个组件，集成了RStudio并且还提供加载好了并行化后台并行化执行引擎，并将R脚本的编写、编译、跟踪执行以及中间变量查看和绘图集于一体，为用户提供了一个强大的R的操作环境。在TranswarpR中，用户除了可以自行编写R的程序脚本、调用开源版本R提供的数千个包和函数之外，还可以直接调用TranswarpR实现的私有的并行化机器学习算法库。TranswarpR目前实现的并行化机器学习算法已经提供了常用的分类、聚类、回归等功能。我们还在根据用户的具体需求在产品开发的中进一步实现更多的并行化算法。

RStudio的配置及使用

Rstudio配置

RStudio有配置文件，但是默认情况下，这个文件没有创建，如果我们要修改默认配置，需要先创建这个文件：

```
touch /etc/rstudio/rserver.conf
```

- 配置网络端口号

  默认的端口是8787，如果我们想改成其他端口，可以在/etc/rstudio/rserver.conf文件中添加并修改参数：

  ```
  www-port=800
  ```

  800是我们修改后的端口号。通过以下命令重启RStudio server，重启后配置生效。

  ```
  rstudio-server restart
  ```

- 其他常用配置

  rsession-ld-library-path=/opt/local/lib:/opt/local/someapp/lib 指定额外的库地址

  auth-required-user-group=rstudio_users 限制可登陆R用户

  rsession-memory-limit-mb=4000 限制使用的最大内存

  rsession-stack-limit-mb=10 限制最大的栈大小

  rsession-process-limit=100 限制最多进程数

  session-timeout-minutes=30 进程超时时间

  r-libs-user=~/R/packages 设置默认的R包

  limit-file-upload-size-mb=100 设置最大的上传文件大小

  r-cran-repos=http://cran.case.edu/ 设置默认的CRAN

Rstudio使用

- Rstudio server的访问

  通过web进行访问，默认情况下RStudio Server的端口是8787。所以，我们可以使用地址http://[server-ip]:8787进行访问，输入linux用户账号进行登录。RStudio不允许系统用户登录，即不允许id小于500的用户登录，可通过在终端输入id命令查看当前用户的id。

- Rstudio server的常用操作

  RStudio-server的启动：

```
rstudio-server start
```

RStudio-server的停止：

```
rstudio-server stop
```

RStudio-server的重启：

```
rstudio-server restart
```

恢复服务可以使用：

```
rstudio-server online
```

- 管理RStudio-server进程

  列出目前正在运行的RStudio-server进程号：

  ```
  rstudio-server active-sessions
  ```

  挂起某一个正在运行的RStudio-server进程号：

  ```
  rstudio-server suspend-session
  ```

  挂起所有正在运行的RStudio-server进程号：

  ```
  rstudio-server suspend-all
  ```

  强行挂起正在运行Rsession的进程：

  ```
  rstudio-server force-suspend-session
  rstudio-server force-suspend-all
  ```

TranswarpR组件使用说明

- **RHDFS**

  在RStudio Server中加载并初始化rhdfs：

  ```
  library(rhdfs)
  hdfs.init()
  ```

- **RHive**

  1.在HDFS根目录下创建/rhive目录，并赋予该目录全权限：

  ```
  sudo –u hdfs hadoop fs –mkdir /rhive
  sudo –u hdfs hadoop fs –chmod 777 /rhive
  ```

  2.在RStudio Server中加载并初始化rhive，并连接到hive server：

```
library(RHive)
rhive.init()
rhive.connect("hive-server-ip","hive-thrift-port")   ##hive-thrift-port默认值10000
str(d)      ##查看查询结果
rhive.close()   ##关闭RHive接口
```

- **TranswarpR**

  TranswarpR是Transwarp Data Hub中提供在R语言的运行环境中调用Inceptor并行执行引擎的组件。它有如下功能：
  1.提供R Shell和Rstudio图形化开发环境
  2.提供常见的基于TranswarpR实现的并行化的机器学习算法库

  TranswarpR有可以运行单机多线程并行版和集群分布式版本，数据可以存放在本地文件系统也可以存放在HDFS文件系统（TDH支持HDFS1.0和2.0）

  使用TranswarpR时，首先需要进行初始化：

```
sc <- TranswarpR.init()
```

  然后可以直接调用TranswarpR的各种并行化算法：

```
> trainFile<-textFile(sc, "hdfs://data.txt")
> txkmeans(trainFile,2,0.5, ",",2)
…
```

- **配置YARN的用户穿透**

  1.登陆管理界面（http://server-ip:8180），进入YARN的配置界面，将属性`yarn.nodemanager.container-executor.class`的值改为`org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor`

  2.在每个节点上添加与Rstudio Server登陆用户相同的用户。如用test用户登陆Rsudio，则同时需要在集群中其他节点上也创建test用户

  3.在集群每个节点的/etc/yarn1/conf/目录下执行以下命令，创建配置文件：

```
$ cd /etc/yarn1/conf/
$ touch container-executor.cfg
```

  container-executor.cfg文件内容如下：

```
yarn.nodemanager.linux-container-executor.group=yarn
min.user.id=0
banned.users=nobody
```

  4.修改集群中所有节点fastdisk所在目录的访问权限：

  进入Inceptor的配置界面，查看属性`inceptor.fastdisk.dir的值`，执行以下命令修改集群中所有节点fastdisk所在目录的访问权限

```
$ pdsh -S -w transwarp-node[1-5] chmod 777 [inceptor.fastdisk.dir的值]
```

  5.修改集群中所有节点localdisk所在目录的访问权限：

  进入Inceptor的配置界面，查看属性`inceptor.local.dirs的值`，执行以下命令修改集群中所有节点localdisk所在目录的访问权限

```
$ pdsh -S -w transwarp-node[1-5] chmod 777 [inceptor.local.dirs的值]
```

  6.修改集群中所有节点ngmr相关目录的权限：

```
$ pdsh -S -w transwarp-node[1-5] chmod 777 /usr/lib/ngmr/work
$ pdsh -S -w transwarp-node[1-5] chmod 777 /usr/lib/ngmr/logs
```

TranswarpR算法使用案例

本章节描述了一个TranswarpR算法使用案例：对某银行用户数据进行KMeans聚类分析：

- **Step 1：创建用户**

  Rstudio Server的登陆用户名为linux用户，可使用已经存在的Linux用户登陆也可通过以下指令创建新用户Ruser并为其设置密码，进行登陆：

```
adduser Ruser
passwd Ruser
```

- **Step 2：访问Rstudio Server**

  通过inceptor角色界面确定Inceptor Transwarp RStudio角色所在节点，通过http://:8787进行访问，也可以单击蓝色RStudio UI直接访问

- **Step 3：查询分析**

  执行以下SQL从hive中查询银行用户数据包括开卡时间、子女数、学历、用卡年限，并对查询结果进行kmeans聚类分析：

```
sc <- TranswarpR.init("kmeans", "GreedyHeterogeneousMode", "1", "0.4")
sql <- "select custage,familyno,toplevel,custyear from bank_client"
txKmeansSQL(sql, 4, iter.max =10, nstart = 1, sep=" ", partsNum = 20)
```

- **Step 4：查看分析结果**

  待算法运行结束，由console可查看对银行用户数据进行聚类后的结果。

对于复杂数据，通过SQL对数据进行清洗后执行kmeans聚类分析：

- **Step 1：执行以下SQL对hive中demo_pl表进行数据清洗**

  其中通过decode将location列中值为'SH', 'BJ','NJ', 'GZ', 'WH'的记录分别替换为1,2,3,4,5,其它值替换为0；
  通过case when语句将age列中小于20的记录替换为1，大于20小于40的记录替换为2，大于40小于60的记录替换为3，大于60小于80的记录替换为4，大于80的记录替换为5；
  通过decode将education列中值为'SEN', 'UND', 'GRA'的记录分别替换为1,2,3，其它记录的值替换为0；
  通过decode将gender列中值为'F', 'M'的记录分别替换为1,2，其它记录的值替换为0

```
sc <- TranswarpR.init("kmeans", "GreedyHeterogeneousMode", "1", "0.4")
sql<- "select uid, decode(location, 'SH', 1, 'BJ', 2, 'NJ', 3, 'GZ', 4, 'WH', 5, 0),
        case when age < 20 then 1
        when 20 <= age < 40 then 2
        when 40 <= age < 60 then 3
        when 60 <= age < 80 then 4
        when age >= 80 then 5 end,
        decode(education, 'SEN', 1, 'UND', 2, 'GRA', 3 , 0),
        decode(gender, 'F', 1, 'M', 2, 0)
        from demo_pl"
```

- **Step 2：对清洗后的数据进行kmeans聚类分析**

```
txKmeansSQL(sql, centers=4, iter.max =10, nstart = 1, sep=" ", partsNum = 20)
```

- **Step 3：查看分析结果**

  待算法运行结束，由console可查看对银行用户数据进行聚类后的结果。

函数参考手册

TranswarpR.init

- **Description**

  初始化一个SharkContext

- **Usage**

```
TranswarpR.init(appName = "TranswarpR",
scheduler_mode = "BasicHeterogeneousMode", mem_core_ratio = "1",
core_use_percent = "0.2", exec_num = "3", core_num = "1",
mem_size = "1024", sparkHome = Sys.getenv("SPARK_HOME"))
```

- **Arguments**

`appName：`在集群中注册的application name，默认值为TranswarpR

`scheduler_mode：`提供四种模式，包括BasicMode，BasicHeterogeneousMode，GreedyMode和GreedyHeterogeneousMode。不同的模式代表向YARN申请资源的不同策略。
Basic模式需要直接指定executor的个数，Greedy模式是每个nodemanger上起一个executor。
集群同构的模式下可以使用BasicMode或者GreedyMode，这两个模式下需要直接指定每个executor占用的core的个数以及内存的大小。
集群异构的模式下可以使用BasicHeterogeneousMode或者GreedyHeterogeneousMode，这两个模式是通过指定每个executor占用的cpu core占总core数的百分比分配core，通过mem和core的比值，决定内存的大小。
集群同构时建议使用BasicMode，集群异构时建议使用BasicHeterogeneousMode，默认值是BasicHeterogeneousMode 。

`mem_core_ratio：`此参数用于BasicHeterogeneousMode或者GreedyHeterogeneousMode。指内存和cpu的比值，内存单位是GB，比如有1个cpu，比值是2，则对应2G内存，默认值为1

`core_use_percent：`此参数用于BasicHeterogeneousMode或者GreedyHeterogeneousMode。指每个Executor使用的core占机器总core的比例，默认值为0.2

`exec_num：`此参数用于BasicMode或是GreedyMode。指提交一个application时启动executor的数目，默认值为3

`core_num：`此参数用于BasicMode或是GreedyMode。指每个executor使用的core的数目，默认值为1

`mem_size：`此参数用于BasicMode或是GreedyMode。指每个executor使用的内存数，默认值为1024M

`sparkHome：`SparkHome地址，默认从环境变量中获取

- **Example**

```
## Not run:
sc <- TranswarpR.init("TranswarpR ", "GreedyHeterogeneousMode", "1", "0.1")
## End(Not run)
```

txKmeans

- **Description**

  该方法通过对输入文件中的训练数据进行学习，生成一个kmeans聚类模型

- **Usage**

```
txKmeans(inputFilePath, centers, iter.max = .Machine$integer.max, nstart = 1, sep, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  `inputFilePath：`数据文件的路径

  `centers：`初始化聚类中心的数目

  `iter.max：`最大迭代次数

  `nstart：`随机选取初始中心的次数.根据nstart的值多次初始化数据中心，并选取最优值.默认值为1

  `sep：`样本数据分隔符

  `partsNum：`数据并行化的分区数，默认值为2

  `outputHDFSPath：`预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  `kPoints：`聚类中心，以矩阵形式返回

  `label：`样本聚类后的类标，以整数向量形式返回

- **Example**

```
## Not run:
txKmeans("hdfs://dataset/kmeans_data.txt",2,10,1," ",4)
## End(Not run)
```

txKmeansSQL

- **Description**

  该方法通过对输入文件中的训练数据进行学习，生成一个kmeans聚类模型

- **Usage**

```
txKmeansSQL(sql, centers, iter.max = .Machine$integer.max, nstart = 1, sep,  partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  `sql：` 抽取数据的SQL语句

  `iter.max：` 最大迭代次数

  `centers：` 初始化聚类中心的数目

  `nstart：` 随机选取初始中心的次数.根据nstart的值多次初始化数据中心，并选取最优值.默认值为1

  `sep：` 样本数据分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

  `outputHDFSPath：` 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  `kPoints：` 聚类中心，以矩阵形式返回

  `label：` 样本聚类后的类标，以整数向量形式返回

- **Example**

```
## Not run:
txKmeansSQL(sql,2,10,1," ",4)
## End(Not run)
```

txLogReg

- **Description**

  该方法将对输入数据进行训练，返回一个逻辑回归模型

- **Usage**

```
txLogReg(trainFile, sep, iter.max = .Machine$integer.max, partsNum = 2)
```

- **Arguments**

  `trainFile：` 训练样本的路径

  `iters.Max：` 最大迭代次数

  `sep：` 样本数据分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

- **Value**

  `intercept：` 模型的截距

  `weight：` 逻辑回归系数

- **Example**

```
## Not run:
txLogReg("hdfs://dataset/breast_cancer_scale.txt"," ",50,10)
## End(Not run)
```

txLogRegSQL

- **Description**

  该方法将对sql查询的数据进行训练，返回一个逻辑回归模型

- **Usage**

```
txLogRegSQL(sql, sep, iter.max = .Machine$integer.max, partsNum = 2)
```

- **Arguments**

  `sql：` 抽取数据的SQL语句

`iters.Max：`最大迭代次数

`sep：`样本数据分隔符

`partsNum：`数据并行化的分区数，默认值为2

- **Value**

`intercept：`模型的截距

`weight：`逻辑回归系数

- **Example**

```
## Not run:
txLogRegSQL("select * from logit_train"," ",50,10)
## End(Not run)
```

txLogRegPredict

- **Description**

该方法将txLogReg或者txLogRegSQL训练所得的模型对预测数据进行预测

- **Usage**

```
txLogRegPredict(testFile, sep, model, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

`testFile：`测试数据的文件路径

`model：`由txLogReg或者txLogRegSQL训练所得的模型

`sep：`样本数据分隔符

`partsNum：`数据并行化的分区数，默认值为2

`outputHDFSPath：`预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

`label：`预测类标值

- **Example**

```
## Not run:
a<-txLogReg("hdfs://dataset/breast_cancer_scale.txt"," ",10,10)
txLogRegPredict("hdfs://dataset/breast_cancer_scale.txt"," ",a$model,10)
## End(Not run)
```

txLogRegPredictSQL

- **Description**

该方法将根据txLogReg或者txLogRegSQL训练所得模型对SQL查询的数据进行预测

- **Usage**

```
txLogRegPredictSQL(sql, sep, model, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

`model：`由txLogReg或者txLogRegSQL训练所得的模型

`sql：`抽取数据的SQL语句

`sep：`样本数据分隔符

`partsNum：`数据并行化的分区数，默认值为2

`outputHDFSPath：`预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

`label：`预测类标值

- **Example**

```
## Not run:
a<-txLogRegSQL("select * from logit_train"," ",50,10)
txLogRegPredictSQL("select * from logit_train"," ",a$model,10)
## End(Not run)
```

txLinReg

- **Description**

  该方法将对输入文件的数据进行训练，返回一个线性回归模型

- **Usage**

```
txLinReg(trainFile, sep, partsNum = 2)
```

- **Arguments**

  `trainFile：` 输入数据的文件路径

  `sep：` 样本数据的分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

- **Value**

  `coefficients：` 线性回归系数，以矩阵形式返回

- **Example**

```
## Not run:
a<-txLinReg(formula=NULL,trainFile="hdfs://ha/user/Rtest/linReg/train/linReg_data.txt",sep=" ",partsNum=2)
## End(Not run)
```

txLinRegSQL

- **Description**

  该方法将对sql的查询结果进行训练，返回一个线性回归模型

- **Usage**

```
txLinRegSQL(sql, partsNum, sep)
```

- **Arguments**

  `sql：` 抽取数据的SQL语句

  `sep：` 样本数据的分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

- **Value**

  `coefficients：` 线性回归系数，以矩阵形式返回

- **Example**

```
## Not run:
a<-txLinRegSQL(sql="select * from linear_train",partsNum=10,sep=" " )
## End(Not run)
```

txLinRegPredict

- **Description**

  该方法将根据由txLinReg 或者 txLinRegSQL方法训练出来的模型，对输入数据进行预测

- **Usage**

```
txLinRegPredict(model, predictFile, sep, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  model： 由txLinReg或者txLinRegSQL方法训练所得的模型

  predictFile： 包含测试数据的文件路径

  sep： 样本数据分隔符

  partsNum： 数据并行化的分区数，默认值为2

  outputHDFSPath： 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  value： 样本预测值，以向量形式返回

- **Example**

```
## Not run:
model<-txLinReg("hdfs://dataset/breast_cancer_scale.txt",50,10," ",4)
txLinRegPredict(model,"hdfs://dataset/breast_cancer_scale_predict.txt",sep=" ",partsNum=2 )
## End(Not run)
```

txLinRegPredictSQL

- **Description**

  该方法将根据由txLinReg 或者 txLinRegSQL方法训练出来的模型，对输入数据进行预测

- **Usage**

```
txLinRegPredictSQL(model, sql, sep, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  model： 由txLinReg或者txLinRegSQL方法训练所得的模型

  sql： 抽取数据的SQL语句

  sep： 样本数据分隔符

  partsNum： 数据并行化的分区数，默认值为2

  outputHDFSPath： 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  value： 样本预测值，以向量形式返回

- **Example**

```
## Not run:
model<-txLinReg("hdfs://dataset/breast_cancer_scale.txt",50,10," ",4)
txLinRegPredictSQL(model,"hdfs://dataset/breast_cancer_scale_predict.txt",sep=" ",partsNum=2 )
## End(Not run)
```

txNavieBayes

- **Description**

  该方法将对输入数据进行训练生成一个朴素贝叶斯模型

- **Usage**

```
txNavieBayes(trainFile, sep, lambda, partsNum = 2)
```

- **Arguments**

  trainFile： 输入数据的文件路径

  sep： 样本数据分隔符

  lambda： 正则化参数

  partsNum： 数据并行化的分区数，默认值为2

- **Value**

`Pi：` 贝叶斯模型的参数，Pi=log p(C)

`Theta：` 贝叶斯模型的参数，Theta=log p(Fi/C)

- **Example**

```
## Not run:
txNavieBayes("hdfs://dataset/sample_naive_bayes_data.txt"," ",1.0,4)
## End(Not run)
```

txNavieBayesSQL

- **Description**

  该方法将SQL查询的数据进行训练生成一个朴素贝叶斯模型

- **Usage**

```
txNavieBayesSQL(sql, sep, lambda, partsNum = 2)
```

- **Arguments**

  `sql：` 抽取数据的SQL语句

  `sep：` 样本数据分隔符

  `lambda：` 正则化参数，一般为1或者2

  `partsNum：` 数据并行化的分区数，默认值为2

- **Value**

  `Pi：` 贝叶斯模型的参数，Pi=log p(C)

  `Theta：` 贝叶斯模型的参数，Theta=log p(Fi/C)

- **Example**

```
## Not run:
txNavieBayesSQL("select * from nb_train"," ",1.0,4)
## End(Not run)
```

txNavieBayesPredict

- **Description**

  该方法将根据txNavieBayes或者txNavieBayesSQL训练所得的模型对输入数据进行预测

- **Usage**

```
txNavieBayesPredict(testFile, sep, model, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  `textFile：` 输入数据的文件路径

  `model：` 由txNavieBayes 或者 txNavieBayesSQL训练所得的模型

  `lambda：` 正则化参数，使用1或者2

  `sep：` 样本数据分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

  `outputHDFSPath：` 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  `label：` 预测类标值

- **Example**

```
## Not run:
a<-txNavieBayes("hdfs://dataset/sample_naive_bayes_data.txt"," ",1.0,4)
txNavieBayesPredict("hdfs://dataset/sample_naive_bayes_data_predict.txt"," ",a$model,4)
## End(Not run)
```

txNavieBayesPredictSQL

- **Description**

  该方法将根据txNavieBayes或者txNavieBayesSQL训练所得的模型对SQL查询的数据进行预测

- **Usage**

```
txNavieBayesPredictSQL(sql, sep, model, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  `model：` 由txNavieBayes 或者 txNavieBayesSQL训练所得的模型

  `sql：` 抽取数据的SQL语句

  `sep：` 样本数据分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

  `lambda：` 正则化参数，使用1或者2

- **Value**

  `label：` 预测类标值

- **Example**

```
## Not run:
a<-txNavieBayesSQL("select * from nb_train"," ",1.0,4)
txNavieBayesPredictSQL("seledt * from bn_predict"," ",a$model,4)
## End(Not run)
```

txSVM

- **Description**

  该方法将对输入数据进行训练，返回一个SVM分类模型

- **Usage**

```
txSVM(trainFile, sep, iter.max = .Machine$integer.max, partsNum = 2)
```

- **Arguments**

  `trainFile：` 输入数据的文件路径

  `sep：` 样本数据分隔符

  `iters.Max：` 最大迭代次数

  `partsNum：` 数据并行化处理的分区数，默认值为2

- **Value**

  `intercept：` 模型的截距

  `weight：` SVM分类模型系数

- **Example**

```
## Not run:
txSVM("hdfs://dataset/breast_cancer_scale.txt"," ",10,4)
## End(Not run)
```

txSVMSQL

- **Description**

  该方法将对SQL查询的结果进行训练，返回一个SVM分类模型

- **Usage**

```
txSVMSQL(sql, sep, iter.max = .Machine$integer.max, partsNum = 2)
```

- **Arguments**

  `sql：` 抽取数据的SQL语句

  `iters.Max：` 最大迭代次数

  `partsNum：` 数据并行化的分区数，默认值为2

  `sep：` 样本数据分隔符

- **Value**

  `intercept：` 模型的截距

  `weight：` SVM分类模型系数

- **Example**

```
## Not run:
txSVM("select * from svm_train"," ",10,4)
## End(Not run)
```

txSVMPredict

- **Description**

  该方法将根据txSVM或者txSVMSQL训练所得的模型对输入数据进行预测

- **Usage**

```
txSVMPredict(testFile, model, sep, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

  `textFile：` 输入数据的文件路径

  `model：` 由txSVM或者txSVMSQL训练所得的模型

  `sep：` 样本数据分隔符

  `partsNum：` 数据并行化的分区数，默认值为2

  `outputHDFSPath：` 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  `label：` 预测类标值

- **Example**

```
## Not run:
a<-txSVM("hdfs://dataset/svm_train.txt"," ",10,4)
txSVMPredict("hdfs://dataset/svm_test.txt",a$model,4,"hdfs:/user/test/result")
## End(Not run)
```

txSVMPredictSQL

- **Description**

  该方法将根据txSVM或者txSVMSQL训练所得的模型对sql查询的结果数据进行预测

- **Usage**

```
txSVMPredictSQL(sql, model, sep, partsNum = 2, outputHDFSPath = "")
```

- **Arguments**

model： 由txSVM或者txSVMSQL训练所得的模型

sql： 抽取数据的SQL语句

sep： 样本数据分隔符

partsNum： 数据并行化的分区数，默认值为2

outputHDFSPath： 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

label： 预测类标值

- **Example**

```
## Not run:
a<-txSVM("hdfs://dataset/svm_train.txt"," ",10,4)
txSVMPredictSQL("select * from svm_test",a$model,4,"hdfs:/user/test/result")
## End(Not run)
```

txRecommendation

- **Description**

协同过滤算法，对输入数据进行协同过滤

- **Usage**

```
txRecommendation(trainFile, sep, iter.max, rank = 10, lambda = 0.01, partsNum, outputHDFSPath)
```

- **Arguments**

trainFile： 输入数据的文件路径

sep： 样本数据的分隔符

iters.Max： 最大迭代次数

rank： 特征矩阵的秩，默认值为10

lambda： 正则化参数，默认为0.01

partsNum： 数据并行化的分区数，默认值为2

outputHDFSPath： 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

recom： 推荐结果，将直接保存在HDFS中

- **Example**

```
## Not run:
txRecommendation("hdfs://dataset/recom.txt"," ",50,10,0.01,20,"hdfs://user/test/result")
## End(Not run)
```

txRecommendationSQL

- **Description**

协同过滤算法，对sql查询的数据进行协同过滤

- **Usage**

```
txRecommendationSQL(sql, sep, iter.max, rank = 10, lambda = 0.01, partsNum, outputHDFSPath)
```

- **Arguments**

sql： 抽取数据的SQL语句

sep： 样本数据的分隔符

iters.Max： 最大迭代次数

rank： 特征矩阵的秩，默认值为10

lambda： 正则化参数，默认为0.01

`partsNum：` 数据并行化的分区数，默认值为2

`outputHDFSPath：` 预测结果保存在HDFS下的路径（如不设置，则只在屏幕输出）

- **Value**

  `recom：` 推荐结果，将直接保存在HDFS中

- **Example**

```
## Not run:
txRecommendationSQL("select * from recom"," ",50,10,0.01,20,"hdfs://user/test/result")
## End(Not run)
```

TranswarpR

Documentation for TranswarpR

rhdfs	Document and APIs for rhdfs
rmr2	Document and APIs for rmr2
rhbase	Document and APIs for rhbase
rhive	Document and APIs for rhive
sparkR	Document and APIs for sparkR

R and Hadoop Distributed Filesystem

Documentation for package ‘rhdfs’ version 1.0.8

- TranswarpR Home

Packages	
as.character.hdfsFH	Reading and Writing to Files on the HDFS
hdfs.cat	Reading Text Files from the HDFS
hdfs.chmod	File and Directory Manipulation Commands for the HDFS
hdfs.chown	File and Directory Manipulation Commands for the HDFS
hdfs.close	Reading and Writing to Files on the HDFS
hdfs.copy	File Copying Commands for the HDFS
hdfs.cp	File Copying Commands for the HDFS
hdfs.defaults	Retrieve and Set rhdfs Defaults
hdfs.del	File and Directory Manipulation Commands for the HDFS
hdfs.delete	File and Directory Manipulation Commands for the HDFS
hdfs.dircreate	File and Directory Manipulation Commands for the HDFS
hdfs.exists	File and Directory Manipulation Commands for the HDFS
hdfs.file	Reading and Writing to Files on the HDFS
hdfs.file.info	File and Directory Manipulation Commands for the HDFS
hdfs.flush	Reading and Writing to Files on the HDFS
hdfs.get	File Copying Commands for the HDFS
hdfs.init	Initialize the rhdfs Package
hdfs.line.reader	Reading Text Files from the HDFS
hdfs.ls	File and Directory Manipulation Commands for the HDFS
hdfs.mkdir	File and Directory Manipulation Commands for the HDFS
hdfs.move	File Copying Commands for the HDFS
hdfs.mv	File Copying Commands for the HDFS
hdfs.put	File Copying Commands for the HDFS
hdfs.read	Reading and Writing to Files on the HDFS
hdfs.read.text.file	Reading Text Files from the HDFS
hdfs.rename	File Copying Commands for the HDFS
hdfs.rm	File and Directory Manipulation Commands for the HDFS
hdfs.rmr	File and Directory Manipulation Commands for the HDFS
hdfs.seek	Reading and Writing to Files on the HDFS
hdfs.stat	File and Directory Manipulation Commands for the HDFS
hdfs.tell	Reading and Writing to Files on the HDFS
hdfs.write	Reading and Writing to Files on the HDFS
hdfslist.files	File and Directory Manipulation Commands for the HDFS
print.hdfsFH	Reading and Writing to Files on the HDFS
rhdfs	rhdfs: R and Hadoop Distributed File System

Documentation for package ‘rmr2’ version 2.3.0

- TranswarpR Home

Packages	
big.data.object	The big data object.
c.keyval	Create, project or concatenate key-value pairs
dfs.empty	Backend-independent file manipulation
dfs.mv	Backend-independent file manipulation
dfs.rmr	Backend-independent file manipulation
dfs.size	Backend-independent file manipulation
equijoin	Equijoins using map reduce
from.dfs	Read or write R objects from or to the file system
gather	Functions to split a file over several parts or to merge multiple parts into one
increment.counter	Set the status and define and increment counters for a Hadoop job
keys	Create, project or concatenate key-value pairs
keyval	Create, project or concatenate key-value pairs
make.input.format	Create combinations of settings for flexible IO
make.output.format	Create combinations of settings for flexible IO
mapreduce	MapReduce using Hadoop Streaming
rmr	A package to perform Map Reduce computations in R
rmr.options	Function to set and get package options
rmr.sample	Sample large data sets
rmr.str	Print a variable's content
scatter	Functions to split a file over several parts or to merge multiple parts into one
status	Set the status and define and increment counters for a Hadoop job
to.dfs	Read or write R objects from or to the file system
to.map	Create map and reduce functions from other functions
to.reduce	Create map and reduce functions from other functions
values	Create, project or concatenate key-value pairs
vsum	Fast small sums

R and HBase Connector

Documentation for package 'rhbase' version 1.2.0

- TranswarpR Home

Packages	
hb.compact.table	Initializing the HBase package and Managing Tables
hb.defaults	Initializing the HBase package and Managing Tables
hb.delete	Functions to Modify HBase Tables
hb.delete.table	Initializing the HBase package and Managing Tables
hb.describe.table	Initializing the HBase package and Managing Tables
hb.get	Functions to Modify HBase Tables
hb.get.data.frame	Functions to Modify HBase Tables
hb.init	Initializing the HBase package and Managing Tables
hb.insert	Functions to Modify HBase Tables
hb.insert.data.frame	Functions to Modify HBase Tables
hb.list.tables	Initializing the HBase package and Managing Tables
hb.new.table	Initializing the HBase package and Managing Tables
hb.regions.table	Initializing the HBase package and Managing Tables
hb.scan	Functions to Modify HBase Tables
hb.scan.ex	Functions to Modify HBase Tables
hb.set.table.mode	Initializing the HBase package and Managing Tables
rhbase	rhbase: Interact with HBase tables from R

Documentation for package ‘RHive’ version 0.0-7

- TranswarpR Home

Packages	
hiveAssign	Methods for the class
hiveClose	Methods for the class
hiveConnect	Methods for the class
hiveDescTable	Methods for the class
hiveExport	Methods for the class
hiveExportAll	Methods for the class
hiveListTables	Methods for the class
hiveLoadTable	Methods for the class
hiveQuery	Methods for the class
hiveRm	Methods for the class
RHive	rhive: R and Hive
rhive	rhive: R and Hive
rhive.aggregate	R Distributed aggregate function using HQL
rhive.assign	Export R function to Hive using functions in Package 'RHive'
rhive.basic	R Distributed basic statistic function using Hive
rhive.basic.by	R Distributed basic statistic function using Hive
rhive.basic.cut	R Distributed basic statistic function using Hive
rhive.basic.cut2	R Distributed basic statistic function using Hive
rhive.basic.merge	R Distributed basic statistic function using Hive
rhive.basic.mode	R Distributed basic statistic function using Hive
rhive.basic.range	R Distributed basic statistic function using Hive
rhive.basic.scale	R Distributed basic statistic function using Hive
rhive.basic.t.test	R Distributed basic statistic function using Hive
rhive.basic.xtabs	R Distributed basic statistic function using Hive
rhive.big.query	Execute HQL(Hive Query) in R, using functions in Package 'RHive'
rhive.block.sample	R Distributed basic statistic function using Hive
rhive.close	Manage connection to Hive using functions in Package 'RHive'
rhive.connect	Manage connection to Hive using functions in Package 'RHive'
rhive.desc.table	R functions to get informations of table from HIVE
rhive.drop.table	R functions to get informations of table from HIVE
rhive.env	Manage connection to Hive using functions in Package 'RHive'
rhive.exist.table	R functions to get informations of table from HIVE
rhive.export	Export R function to Hive using functions in Package 'RHive'
rhive.exportAll	Export R function to Hive using functions in Package 'RHive'
rhive.hdfs.cat	R functions to communicate with HDFS
rhive.hdfs.chgrp	R functions to communicate with HDFS
rhive.hdfs.chmod	R functions to communicate with HDFS
rhive.hdfs.chown	R functions to communicate with HDFS
rhive.hdfs.close	R functions to communicate with HDFS
rhive.hdfs.connect	R functions to communicate with HDFS
rhive.hdfs.du	R functions to communicate with HDFS

rhive.hdfs.exists	R functions to communicate with HDFS
rhive.hdfs.get	R functions to communicate with HDFS
rhive.hdfs.info	R functions to communicate with HDFS
rhive.hdfs.ls	R functions to communicate with HDFS
rhive.hdfs.mkdirs	R functions to communicate with HDFS
rhive.hdfs.put	R functions to communicate with HDFS
rhive.hdfs.rename	R functions to communicate with HDFS
rhive.hdfs.rm	R functions to communicate with HDFS
rhive.hdfs.tail	R functions to communicate with HDFS
rhive.init	Manage connection to Hive using functions in Package 'RHive'
rhive.list.tables	R functions to get informations of table from HIVE
rhive.load	R functions to communicate with HDFS
rhive.load.table	R functions to get informations of table from HIVE
rhive.load.table2	R functions to get informations of table from HIVE
rhive.mapapply	R Distributed apply function using HQL
rhive.mrapply	R Distributed apply function using HQL
rhive.napply	R Distributed apply function using HQL
rhive.query	Execute HQL(Hive Query) in R, using functions in Package 'RHive'
rhive.reduceapply	R Distributed apply function using HQL
rhive.rm	Export R function to Hive using functions in Package 'RHive'
rhive.sapply	R Distributed apply function using HQL
rhive.save	R functions to communicate with HDFS
rhive.script.export	Export R function to Hive using functions in Package 'RHive'
rhive.script.unexport	Export R function to Hive using functions in Package 'RHive'
rhive.size.table	R functions to get informations of table from HIVE
rhive.write.table	R functions to communicate with HDFS

Documentation for package 'SparkR' version 0.1

- TranswarpR Home

Packages	
Broadcast	S4 class that represents a Broadcast variable
broadcast	Broadcast a variable to all workers
Broadcast-class	S4 class that represents a Broadcast variable
cache	Persist an RDD
cache-method	Persist an RDD
collect	Collect elements of an RDD
collect-method	Collect elements of an RDD
collectPartition	Collect elements of an RDD
collectPartition-method	Collect elements of an RDD
combineByKey	Combine values by key
combineByKey-method	Combine values by key
count	Return the number of elements in the RDD.
count-method	Return the number of elements in the RDD.
flatMap	Flatten results after apply a function to all elements
flatMap-method	Flatten results after apply a function to all elements
groupByKey	Group values by key
groupByKey-method	Group values by key
hashCode	Compute the hashCode of an object
includePackage	Include this specified package on all workers
lapply-method	Apply a function to all elements
lapplyPartition	Apply a function to each partition of an RDD
lapplyPartition-method	Apply a function to each partition of an RDD
lapplyPartitionsWithIndex	Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.
lapplyPartitionsWithIndex-method	Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.
length-method	Return the number of elements in the RDD.
map	Apply a function to all elements
map-method	Apply a function to all elements
mapPartitionsWithIndex	Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.
mapPartitionsWithIndex-method	Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.
parallelize	Create an RDD from a homogeneous list or vector.
partitionBy	Partition an RDD by key
partitionBy-method	Partition an RDD by key
RDD	S4 class that represents an RDD
RDD-class	S4 class that represents an RDD
reduce	Reduce across elements of an RDD.
reduce-method	Reduce across elements of an RDD.
reduceByKey	Merge values by key
reduceByKey-method	Merge values by key

sampleRDD	Return an RDD that is a sampled subset of the given RDD.
sampleRDD,RDD	Return an RDD that is a sampled subset of the given RDD.
sampleRDD-method	Return an RDD that is a sampled subset of the given RDD.
setBroadcastValue	Internal function to set values of a broadcast variable.
sparkR.init	Initialize a new Spark Context.
take	Take elements from an RDD.
take-method	Take elements from an RDD.
takeSample	Return a list of the elements that are a sampled subset of the given RDD.
takeSample,RDD	Return a list of the elements that are a sampled subset of the given RDD.
takeSample-method	Return a list of the elements that are a sampled subset of the given RDD.
textFile	Create an RDD from a text file.
value	Broadcast a variable to all workers
value-method	Broadcast a variable to all workers