Apache Pig简介与实践

Apache Pig 是一个用来分析大数据集的平台，它由两部分 组成：一部分是用于表达数据分析程序的高 级脚本 语 言，另一部分是用于 评 估分析程序的基本工具。目前来看， Pig 主要用于离线数据的批量 处理应 用 场 景，但是随着 Pig 的发 展 处理数据的速度会不断地提升， 这 可能依 赖于 Pig 底 层 的 执行引擎。比如， Pig 通过指定 执 行模式，可以使用 Hadoop 的 MapReduce 计算引擎来 实现 数据 处 理，也可以使用基于 Tez 的计算引擎来 实现 （Tez 是为了绕开 MapReduce 多阶段 Job 写磁 盘 而设计 的 DAG 计算引擎，性能 应该 比 MapReduce 要快），看到 Pig 未来的 发 展路 线图，以后可能会基于 Storm 或 Spark 计算平台 实现 底 层计 算引擎，那样 速度会有极大地提升。

我 们 基于最新的 0.15.0 版本的 Pig （Hadoop 使用的是 2.2.0 版本），通过编 写一些例子脚本来 实 践 Pig 的 语 言特性。

Pig 安装与 执 行

Pig 安装非常 简单，只需要下 载 Pig 包，然后解 压缩 即可：

```
wget http://mirror.bit.edu.cn/apache/pig/pig-
0.15.0/pig-0.15.0.tar.gz
tarxvzf pig-0.15.0.tar.gz
sudo ln -s /usr/local/pig-0.15.0 /usr/local/pig
cd /usr/local/pig
bin/pig -x mapreduce
```

如果希望直接使用 pig 命令，可以修改 环 境 变 量文件 ~/.bashrc ，增加如下配置：

```
export PIG_HOME=/usr/local/pig
export PATH=$PATH:$PIG_HOME/bin
```

使 变 量配置生效：

```
. ~/.bashrc
```

Pig 支持如下 4 种 执行模式：

- 本地模式

本地模式主要是基于本地文件系 统，比较 适合 调试 脚本使用。 进 入本地模式 执行如下命令：

```
pig -x local
```

- Tez 本地模式

Tez 本地模式 类似于前面的本地模式，它使用 Tez 运行 时引擎，进 入 Tez 本地模式 执 行如下命令：

```
pig -x tez_local
```

不过该 模式 还处 于试验阶 段，不过 多累述。

- MapReduce 模式

MapReduce 模式基于 Hadoop ，数据存储 在 HDFS 上，它基于运行于 YARN 之上的 MapReduce 进 行 处 理。进 入 MapReduce 运行模式 执行如下命令：

```
pig -x mapreduce
```

一般，我 们 的数据都是存 储 在 HDFS 上的，使用 该 模式能 够 充分利用 Hadoop 集群的 计 算能力。

- Tez 模式

基于 Tez 模式执行，需要在安装 Hadoop 集群的时候，修改 Hadoop 配置文件 mapred-site.xml ，将属性 mapreduce.framework.name 的值设置为 yarn-tez 。进入 Tez 模式执行如下命令：

```
pig -x tez
```

有关 Tez 相关内容，可以查看 Apache Tez 官网介绍。

数据类型

Pig 的数据类型可以分为 2 类，分别为简单类型和复杂类型。简单类型包括：int 、long 、float 、double 、chararray 、bytearray 、boolean 、datetime 、biginteger 、bigdecimal 。复杂类型包括：tuple 、bag 、map 。

这里对特别的数据类型，解释说明一下：

chararray 相当于字符串 String ；bytearray 相当于字节数组；tuple 是一个有序的字段的集合，可以理解为元组，例如(3090018, ,Android?, 76) ；bag 是 tuple 的集合，例如 {(3090018, ,Android?, 76), (3090019, ,iOS?, 172)} ；map 是键值对的集合，例如 [name#Jeff Stone, age#28, healthy index#195.58] 。

基本操作符

- 算数操作符（Arithmetic Operators ）包括：+ 、- 、* 、/ 、% 、?: 、CASE WHEN THEN ELSE END 。
- 布尔操作符（Boolean Operators ）包括：AND 、OR 、IN 、NOT 。
- 类型转换操作符（Cast Operators ）：使用圆括号包含类型名，作用于一个字段，例如 (int)age 、(map[]) 、(chararray)COUNT($2) 、(tuple(chararray,int,map[]))name_age_scores 等等。
- 比较操作符（Comparison Operators ）包括：== 、!= 、< 、> 、<= 、>= 、matches 。其中，matches 比较操作符使用 Java 的 Pattern 进行匹配来比较，例如 user_name matches ,[a - n]{3,12}? 。
- 类型构造操作符（Type Construction Operators ）：可以创建复杂类型的数据，tuple 使用 () ，map 使用 [] ，bag 使用 {} ，例如 FOREACH users GENERATE (name, age, address) 。
- 解引用操作符（Dereference Operators ）：解引用主要是针对集合类型 tuple 、bag 、map ，从集合中拿到对应字段的值。比如对于 tuple ，定义类型 t=tuple(t1:int,t2:int,t3:int) ，则我要获取字段 t1 和 t3 的值，一种方式可以通过 t.t1 和 t.t3 得到，也可以通过 t.$0 和 t.$2 获取到。

关系操作符

操作符	语法	说明
ASSERT	ASSERT alias BY expression [, message];	断言：判定某个字段的值的条件为 true
COGROUP	alias = COGROUP alias { ALL | BY expression} [, alias ALL | BY expression …] [USING 'collected' | 'merge'] [PARTITION BY partitioner] [PARALLEL n];	数据分组，与 GROUP 相同，但是至多支持 127 个关系
CROSS	alias = CROSS alias, alias [, alias [PARTITION BY partitioner] [PARALLEL n];	笛卡尔积
CUBE	alias = CUBE alias BY { CUBE expression | ROLLUP expression }, [CUBE expression | ROLLUP	计算 CUBE ，支持 ROLLUP 操作

	expression] [PARALLEL n];	
DEFINE	DEFINE macro_name (param [, param ...]) RETURNS {void | alias [, alias ...]} { pig_latin_fragment }; DEFINE alias {function | [`command` [input] [output] [ship] [cache] [stderr]] };	定义宏（类似函数），能够重用脚本代码 为 UDF 或 streaming 设置别名
DISTINCT	alias = DISTINCT alias [PARTITION BY partitioner] [PARALLEL n];	去重操作，可以指定并行度（即Reducer 个数）
FILTER	alias = FILTER alias BY expression;	条件 过滤
FOREACH	alias = FOREACH alias GENERATE expression [AS schema] [expression [AS schema] … .]; alias = FOREACH nested_alias { alias = {nested_op | nested_exp}; [{alias = {nested_op | nested_exp}; …] GENERATE expression [AS schema] [expression [AS schema] … .] };	基于列 对数据 进行 转换
GROUP	alias = GROUP alias { ALL | BY expression} [, alias ALL | BY expression …] [USING 'collected' | 'merge'] [PARTITION BY partitioner] [PARALLEL n];	数据分 组操作： 只支持一个关系 PARALLEL 子句可以指定并行度（Reducer 个数）
IMPORT	I MPORT ,file -with- macro?;	导 入外部 Pig 脚本
JOIN	alias = JOIN alias BY {expression|?(,expression [, expression …],)?} (, alias BY {expression|?(,expression [, expression …],)?} …) [USING 'replicated' | 'skewed' | 'merge' | 'merge-sparse'] [PARTITION BY partitioner] [PARALLEL n]; alias = JOIN left-alias BY left-alias-column [LEFT|RIGHT|FULL] [OUTER], right-alias BY right-alias-column [USING 'replicated' | 'skewed' | 'merge'] [PARTITION BY partitioner] [PARALLEL n];	内连接 外连接
LIMIT	alias = LIMIT alias n;	输出结果集的 n 个记录
LOAD	LOAD ,data? [USING function] [AS schema];	从数据源加 载数据
MAPREDUCE	alias1 = MAPREDUCE ,mr.jar? STORE alias2 INTO ,inputLocation? USING storeFunc LOAD ,outputLocation? USING loadFunc AS schema	在 Pig 中执 行 MapReduce 程序，需要指定使用的 MapReduce 程序 JAR

	[`params, ... `];	文件
ORDER BY	alias = ORDER alias BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] } [PARALLEL n];	排序
RANK	alias = RANK alias [BY { * [ASC|DESC] | field_alias [ASC|DESC] [, field_alias [ASC|DESC] …] } [DENSE]];	排名操作：可能有排名相同的，即排名序号相同
SAMPLE	SAMPLE alias size;	用于采 样，size 范围[0, 1]
SPLIT	SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression …] [, alias OTHERWISE];	将一个大表，拆分成多个小表
STORE	STORE alias INTO ,directory? [USING function];	存储结 果到文件系 统：如果为指定 USING 子句，则使用默 认的 PigStorage() ，更多可以查看“ Load/Store 函数 ”。
STREAM	alias = STREAM alias [, alias …] THROUGH {`command` | cmd_alias } [AS schema] ;	将数据 发送到外部程序或者脚本
UNION	alias = UNION [ONSCHEMA] alias, alias [, alias …];	计 算并集

关系操作符示例

- ASSERT

```
live_user_ids = LOAD '/test/live_user_ids' USING
PigStorage() AS (udid: chararray);
ASSERT live_user_ids BY udid != null, 'udid MUST NOT
be NULL!';
```

上面断言表 live_user_ids 中的 udid 字段一定存在 值。

- GROUP

根据某个或某些字段 进行分 组，只根据一个字段 进 行分 组，比较简单 。如果想要根据两个字段分组，则 可以将两个字段构造成一个 tuple ，然后进行分 组。Pig 脚本如下所示：

```
events = LOAD
'/data/etl/hive_input/20150613/basis_event_2015061306-
r-00002'  USING PigStorage('\t');
projected_events = FOREACH events GENERATE $1 AS
(event_code: long), $2 AS (udid: chararray), $10 AS
(network: chararray), $34 AS (area_code: int); -- $1
是表 events 的第 2 个字段
uniq_events = DISTINCT projected_events;
```

```
uniq_events = FILTER uniq_events BY (event_code IS NOT
NULL) AND (udid IS NOT NULL) AND (network IS NOT NULL)
AND (area_code IS NOT NULL);
groupped = GROUP uniq_events BY (udid, event_code)
PARALLEL 2; --        指定使用  2 个 Reducer
selected10 = LIMIT groupped 10;
DUMP selected10;
```

从 HDFS 加载的文件中 执行投影操作，生成包含 event_code 、udid 、network 、area_code 这 4 个字段的一个表 projected_events ，接着执行去重、条件 过滤 操作，计算分 组的时 候基于 event_code 、udid 两个字段 进行分 组。

- FOREACH

FOREACH 操作可以 针对 一个数据集 进行迭代 处 理操作，生成一个新的数据集。它有 2 种使用方法，一种是 执 行投影操作， 选择 部分字段的数据，例如脚本：

```
provinces = LOAD '/test/provinces' USING
PigStorage(',') AS (country_id: int, province_id: int,
name: chararray);
compositekeyed_provinces = FOREACH provinces GENERATE
(CONCAT(CONCAT((chararray)country_id, '_'),
(chararray)province_id) AS pid, name);
```

这 里，将原数据集的两个主 键 字段的 值进 行拼接合并，作 为新表的一个字段。

另一种是，支持在 FOREACH 操作中使用代 码段，可以增加更复 杂 的处理逻辑 ，摘自官网的例子，例如脚本：

```
a = LOAD '/test/data' AS (url:chararray,
outlink:chararray);
DUMP a;
(www.ccc.com,www.hjk.com)
(www.ddd.com,www.xyz.org)
(www.aaa.com,www.cvn.org)
(www.www.com,www.kpt.net)
(www.www.com,www.xyz.org)
(www.ddd.com,www.xyz.org)
b = GROUP a BY url;
DUMP b;
(www.aaa.com,{(www.aaa.com,www.cvn.org)})
(www.ccc.com,{(www.ccc.com,www.hjk.com)})
(www.ddd.com,{(www.ddd.com,www.xyz.org),(www.ddd.com,w
ww.xyz.org)})
(www.www.com,{(www.www.com,www.kpt.net),(www.www.com,w
ww.xyz.org)})

result = FOREACH b {
filterda = FILTER a BY outlink == 'www.xyz.org'; --                过
滤掉 outlink       字段值为 'www.xyz.org'       的记录
filtered_outlinks =filterda.outlink;
```

```
filtered_outlinks = DISTINCT filtered_outlinks;  --                    对
outlink      集合进行去重
      GENERATE group, COUNT(filtered_outlinks); --
根据对表 a 进行分组得到 group ， 计算每个分 组中 outlink      的数量
};
DUMP result;
(www.aaa.com,0)
(www.ccc.com,0)
(www.ddd.com,1)
(www.www.com,1)
```

上面，表 b 的第一个字段 为 chararray 类 型的字段（存放域名字符串），第二个字段是一个 bag 类型的字段（存在当前域名的出 链 接，即<url, outlink> 的集合），最后统计 的 结 果是 给定的 url 的出 链 接的个数。

- FILTER

根据条件 进 行 过滤，相当于 SQL 中 WHERE 子句。示例脚本如下所示：

```
live_info = LOAD '/test/live_info' USING PigStorage()
AS (id: long, name: chararray);
newly_added_lives = FILTER live_info BY (id %
140000 >= 0) AND (name matches '[a-zA-Z0-9]{8, 32}' OR
name == 'test');
```

上面内容很好理解，不再累述。

- JOIN

JOIN 操作支持支持配置并行度，指定 Reducer 的数量。
表 连 接操作，支持内 连接和外 连 接，内连接脚本示例如下：

```
live = LOAD '/test/shiyj/pig/pig_live' USING
PigStorage('\t') AS (id: long, name: chararray);
program = LOAD '/test/shiyj/pig/pig_live_program'
USING PigStorage('\t') AS (id: long,name:
chararray,live_id: long,live_start: chararray,live_end:
chararray);
program_info = JOIN live BY id, program BY live_id;
DUMP program_info;
```

根据表 live 的 id 字段，表 program 的 live_id 字段 进 行内 连 接。
外 连 接的操作，官网 给出了 4 个例子，可以分 别 看一下。左外 连 接例子如下：

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A by $0 LEFT OUTER, B BY $0; --                表 A 和 B 的字段
n 进行左外 连接
```

全外 连接的例子如下所示：

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
C = JOIN A BY $0 FULL, B BY $0; --                使用 FULL 关键字
```

支持复制的左外 连 接（Replicated Join ），示例如下：

```
A = LOAD 'large';
B = LOAD 'tiny';
C= JOIN A BY $0 LEFT, B BY $0 USING 'replicated';
```

只有左外 连接支持 这种方式，实际上 Replicated Join 会在 MapReduce 的 Map 阶段把做左表进行复制，也就是 说做表 应该 是小表，能 够在内存中放得下，然后与右表 进行连接操作。还有一种使用 Skewed Join ，示例如下所示：

```
A = LOAD 'studenttab' as (name, age, gpa);
B = LOAD 'votertab' as (name, age, registration,
contribution);
C = JOIN A BY name FULL, B BY name USING 'skewed';
```

只有在 进行外 连接的两表的数据，明 显不对称，称为数据 倾斜，一个表很大，另一个表相 对小，但是内存中放不下， 这种情况可以使用 Skewed Join 操作。目前，Pig 支持基于两表的 Skewed Join 操作。

- DISTINCT

去重操作使用 DISTINCT ，比较简单，示例如下所示：

```
live_user_ids = LOAD '/test/live_user_ids' USING
PigStorage() AS (udid: chararray);
uniq_user_ids= DISTINCT live_user_ids PARALLEL 8;
```

DISTINCT 操作支持配置并行度，指定 Reducer 的数量。

- UNION

计算并集操作，使用 UNION 操作符，示例如下所示：

```
a = LOAD 'data' AS (a1:int,a2:int,a3:int);
b = LOAD 'data' AS (b1:int,b2:int);
u = UNION a, b;
```

计算并集，不要求两表的字段数一定相同。

- LIMIT

LIMIT 选择计 算结果的一部分，示例如下所示：

```
top100_user_ids = LIMIT live_user_ids 100;
```

- STORE

STORE 操作用来保存 计算结果，示例如下所示：

```
STORE play_users INTO '/test/shiyj/tmp.play_users'
USING PigStorage ('\t');
```

如果没有指定 USING 子句，则默认使用 PigStorage() 函数，另外 Pig 还支持如下的 Store/Load 函数：

```
BinStorage()
JsonLoader(['schema'])
JsonStorage()
PigDump()
PigStorage([field_delimiter] , ['options'])
TextLoader()
HBaseStorage('columns', ['options'])
```

AvroStorage(['schema|record name'], ['options'])
TrevniStorage(['schema|record name'], ['options'])
AccumuloStorage(['columns'[, 'options']])
OrcStorage(['options'])

具体使用方法，可以参考文档介　绍。

- CROSS

比较容易理解，摘自官网上的例子，如下所示：

```
--　加载表 a 数据
a = LOAD 'data1' AS (a1:int,a2:int,a3:int);
DUMP a;
(1,2,3)
(4,2,1)

--　加载表 b 数据
b = LOAD 'data2' AS (b1:int,b2:int);
DUMP b;
(2,4)
(8,9)
(1,3)

--　计算笛卡尔积，并输出结果
result = CROSS a, b;
DUMP result;
(1,2,3,2,4)
(1,2,3,8,9)
(1,2,3,1,3)
(4,2,1,2,4)
(4,2,1,8,9)
(4,2,1,1,3)
```

- CUBE

这个操作符功能比　较强大，如下所示：

```
users = LOAD '/test/shiyj/pig/pig_live_users' AS
(create_date,room_id,audio_id,udid);
groupped = COGROUP users BY
(create_date,room_id,audio_id);
groupped_count = FOREACH groupped {
uniq = DISTINCT users.udid;
    GENERATE group, COUNT(uniq);
};
STORE groupped_count INTO
'/test/shiyj/pig/groupped_count' USING
PigStorage('\t'); --　　将分组统计的结果保存到 HDFS

groupped_count = LOAD
'/test/shiyj/pig/groupped_count/part-r-*' AS (k:
```

```
    tuple(chararray, long, long), cnt: int); --                    加载前面保存
    的结果，进行 CUBE计算
    groupped_count = FOREACH groupped_count GENERATE k.$0,
    k.$1, k.$2, cnt;
    cubed_users = CUBE groupped_count BY CUBE($0, $1, $2);
    DUMP cubed_users;
```

我们可以看下官网文档的例子，简单比较容易理解：

（1）CUBE 操作

Pig 脚本内容，如下所示：

```
    salesinp = LOAD '/pig/data/salesdata' USING
    PigStorage(',') AS (product:chararray, year:int,
    region:chararray, state:chararray, city:chararray,
    sales:long);
    cubedinp = CUBE salesinp BY CUBE(product,year);
    result = FOREACH cubedinp GENERATE FLATTEN(group),
    SUM(cube.sales) AS totalsales;
    DUMP result;
```

如果输入数据为(car, 2012, midwest, ohio, columbus, 4000)，则上面脚本执行 CUBE 操作，结果输出内容如下所示：

```
    (car,2012,4000)
    (car,,4000)
    (,2012,4000)
    (,,4000)
```

上面针对产品（product）和年度（year）两个维度进行查询。

（2）ROLLUP 操作

CUBE 操作支持 ROLLUP（上卷操作），例如脚本内容：

```
    salesinp = LOAD '/pig/data/salesdata' USING
    PigStorage(',') AS (product:chararray, year:int,
    region:chararray, state:chararray, city:chararray,
    sales:long);
    rolledup = CUBE salesinp BY ROLLUP(region,state,city);
    result = FOREACH rolledup GENERATE FLATTEN(group),
    SUM(cube.sales) AS totalsales;
    DUMP result;
```

同样如果输入 tuple 值为 (car, 2012, midwest, ohio, columbus, 4000)，则 ROLLUP 操作结果如下所示：

```
    (midwest,ohio,columbus,4000)
    (midwest,ohio,,4000)
    (midwest,,,4000)
    (,,,4000)
```

上面只是根据 ROLLUP 表达式指定的维度执行 CUBE 操作。

（3）合并CUBE 和 ROLLUP 操作

还可以将 CUBE 操作和 ROLLUP 操作合并起来，例如执行脚本内容：

```
salesinp = LOAD '/pig/data/salesdata' USING
PigStorage(',') AS (product:chararray, year:int,
region:chararray, state:chararray, city:chararray,
sales:long);
cubed_and_rolled = CUBE salesinp BY CUBE(product,year),
ROLLUP(region, state, city);
result = FOREACH cubed_and_rolled GENERATE
FLATTEN(group), SUM(cube.sales) AS totalsales;
```

上面的 CUBE 操作等价于下面两中操作：

```
cubed_and_rolled = CUBE salesinp BY
CUBE(product,year,region, state, city);
--    或
cubed_and_rolled = CUBE salesinp BY
ROLLUP(product,year,region, state, city);
```

执行结果，如下所示：

```
(car,2012,midwest,ohio,columbus,4000)
(car,2012,midwest,ohio,,4000)
(car,2012,midwest,,,4000)
(car,2012,,,,4000)
(car,,midwest,ohio,columbus,4000)
(car,,midwest,ohio,,4000)
(car,,midwest,,,4000)
(car,,,,,4000)
(,2012,midwest,ohio,columbus,4000)
(,2012,midwest,ohio,,4000)
(,2012,midwest,,,4000)
(,2012,,,,4000)
(,,midwest,ohio,columbus,4000)
(,,midwest,ohio,,4000)
(,,midwest,,,4000)
(,,,,,4000)
```

- SAMPLE

对数据进行取样操作，脚本如下所示：

```
--    加载原始数据集，并 计算记录数
users = LOAD '/test/shiyj/pig/pig_live_users' AS
(create_date,room_id,audio_id,udid);
g_users = GROUP users ALL;
total_user_cnt = FOREACH g_users GENERATE COUNT(users);
DUMP total_user_cnt;

-- 15%  取样，计算取样记录数
sampled_users = SAMPLE users 0.15;
g_sampled_users = GROUP sampled_users ALL;
```

```
sampled_user_cnt = FOREACH g_sampled_users GENERATE
COUNT(sampled_users);
DUMP sampled_user_cnt;
```

- MAPREDUCE

MAPREDUCE 操作允 许在 Pig 脚本内部 执行 MapReduce 程序，示例脚本来自官网，如下所示：

```
A = LOAD 'WordcountInput.txt';
B = MAPREDUCE 'wordcount.jar' STORE A INTO 'inputDir'
LOAD 'outputDir'
   AS (word:chararray, count: int)
`org.myorg.WordCountinputDiroutputDir`;
```

如果直接写原生的 MapReduce 程序各个能解决 实际问题 ，可以将写好的程序打包，在 Pig 脚本中指定相关参数即可运行。

- SPLIT

将一个表拆分成多个表，可以按照 “水平拆分 ”的思想 进行操作，根据某些条件来生成新表。示例如下所示：

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int); --                    加载数据
到表 A
DUMP A;
(1,2,3)
(4,5,6)
(7,8,9)

SPLIT A INTO X IF f1<7, Y IF f2==5, Z IF (f3<6 OR
f3>6); -- A          表中满足条件 f1<7 的记录插入到表 X 中， 满足条件
f2==5 的记录插入到 T 表中， 满足条件 (f3<6 OR f3>6)       的记录插入到
Z 表中

DUMP X;
(1,2,3)
(4,5,6)

DUMP Y;
(4,5,6)

DUMP Z;
(1,2,3)
(7,8,9)
```

求值函数（Eval Functions ）

函数	语法	说明
AVG	AVG(expression)	计算某一个数字 类型的列的均 值，数字类型支持：int double ，bigdecimal ，biginteger ，bytearray
BagToString	BagToString(vals:bag [, delimiter:chararray])	将 bag 转换 成字符串，可以指定分隔符，适合拼接 b

CONCAT	CONCAT(expression, expression, [...expression])	字符串拼接
COUNT	COUNT(expression)	计算一个 bag 中元素的 总数，不含 NULL 值
COUNT_STAR	COUNT_STAR(expression)	计算一个 bag 中元素的 总数，包含 NULL 值
DIFF	DIFF (expression, expression)	比较一个 tuple 中的两个字段， 这两个字段都是 bag 在两个 bag 中不同的元素， 结果仍然是一个 bag
IsEmpty	IsEmpty(expression)	检查一个 map 或 bag 是否为空
MAX	MAX(expression)	计算最大值，支持数组类型:int，long，float ，doubl biginteger ，bytearray
MIN	MIN(expression)	计算最小值，支持数组类型:int，long，float ，doubl biginteger ，bytearray

PluckTuple	DEFINE pluck PluckTuple(expression1) DEFINE pluck PluckTuple(expression1,expression3) pluck(expression2)	允许定义一个字符串前 缀，然后过滤指定的列， 满足 开始，或者匹配 该正则表达式，下面是官网的例子：
		a = LOAD 'a' as (x, y);
		b = LOAD 'b' as (x, y);
		c = JOIN a by x, b by x; -- 表
		接，因为表 a 和 b 有相同的列名，所以 连
		缀“表名:: ”来区分
		DEFINE pluck PluckTuple('a::');
		- 定义前缀"a::" ，等价于 DEFINE plu
		PluckTuple('a::', true);
		d = FOREACH c GENERATE
		FLATTEN(pluck(*)); -- 包含前缀"a
		留
		DESCRIBE c;
		c: {a::x: bytearray,a::y:
		bytearray,b::x: bytearray,b::y:
		bytearray}
		DESCRIBE d;
		d: {plucked::a::x:
		bytearray,plucked::a::y: bytear
		DEFINE
		pluckNegativePluckTuple('a::','
		-- 定义前缀"a::" ，包含该前缀的过滤
		d = FOREACH c GENERATE
		FLATTEN(pluckNegative(*));
		DESCRIBE d; -- 结果中包含前缀"a::
		掉
		d: {plucked::b::x:
		bytearray,plucked::b::y: bytear

SIZE	SIZE(expression)	计算 Pig 指定数据类型的元素的数量，支持 类型:int

		double ， ,chararray ，bytearray 、tuple 、bag 、ma
SUBTRACT	SUBTRACT(expression, expression)	bag 操作符，用来 对两个 bag 做差集操作， 结果为： bag 中但不包含在第二个 bag 中的元素
SUM	SUM(expression)	求和操作，支持 类型:int，long，float，double ，bigd biginteger ，将bytearray 转换为 double 类型
TOKENIZE	TOKENIZE(expression [, 'field_delimiter'])	拆分一个字符串，得到一个 bag 结果集

数学函数

数学函数比 较简单，不再详细 描述，主要包括如下 20 个：

- ABS
- ACOS
- ASIN
- ATAN
- CBRT
- CEIL
- COS
- COSH
- EXP
- FLOOR
- LOG
- LOG10
- RANDOM
- ROUND
- ROUND_TO
- SIN
- SINH
- SQRT
- TAN
- TANH

具体使用可以 查看官方文档。

字符串函数

字符串函数非常常用，主要包括如下 20 个：

- ENDSWITH
- EqualsIgnoreCase
- INDEXOF
- LAST_INDEX_OF
- LCFIRST
- LOWER
- LTRIM
- REGEX_EXTRACT
- REGEX_EXTRACT_ALL
- REPLACE
- RTRIM
- SPRINTF
- STARTSWITH

STRSPLIT
STRSPLITTOBAG
SUBSTRING
TRIM
UCFIRST
UPPER
UniqueID

使用方法可以　查看文档。

日期 时间 函数

日期函数有下面　24 个，如下所示：

AddDuration
CurrentTime
DaysBetween
GetDay
GetHour
GetMilliSecond
GetMinute
GetMonth
GetSecond
GetWeek
GetWeekYear
GetYear
HoursBetween
MilliSecondsBetween
MinutesBetween
MonthsBetween
SecondsBetween
SubtractDuration
ToDate
ToMilliSeconds
ToString
ToUnixTime
WeeksBetween
YearsBetween

集合函数

集合函数主要是，将其他　类型的数据 转换为 集合 类型 tuple　、bag 、map ， 如下所示：

TOTUPLE
TOBAG
TOMAP
TOP

前面 3 个都是生成集合的函数，最后一个用来　　计 算一个集合中的　topN　个元素，可以指定是按照升序 / 降序得到的 结 果，语 法为 TOP(topN,column,relation)　　。

Hive UDF　　函数

在 Pig 中可以直接　调用 Hive　的 UDF，HiveUDAF　和 HiveUDTF　，语 法如下表所示：

函数	语 法	说 明
HiveUDF	HiveUDF(name[, constant	

	parameters])	DEFINE sin HiveUDF('sin'); - - 定义 HiveUDF ， 后面可以直接使用函数 sin a = LOAD 'student' AS (name:chararray, age:int, gpa:double); b = FOREACH a GENERATE sin(gpa); -- 使用函数 sin

HiveUDAF	HiveUDAF(name[, constant parameters])	DEFINE explode HiveUDTF('explode');
		a = LOAD 'mydata' AS (a0:{(b0:chararray)});
		b = FOREACH a GENERATE FLATTEN(explode(a0));

HiveUDTF	HiveUDTF(name[, constant parameters])	DEFINE avgHiveUDAF('avg');
		a = LOAD 'student' AS (name:chararray, age:int, gpa:double);
		b = GROUP a BY name;
		c = FOREACH b GENERATE group, AVG(a.age);

Pig UDF

Pig 也支持用 户 自定 义函数 UDF，而且支持使用多种 编程语 言来 实现 UDF，目前支持的 变成语 言包括：Java 、JavaScript 、Jython 、Ruby 、Groovy 、Python 。

以 Java 为例，可以通 过继 承自 类 org.apache.pig.EvalFunc 来实现 一个 UDF，将实现 的 UDF 泪打包后， Pig 安装目 录下的 CLASSPATH 下面，然后可以在 Pig 脚本中使用。例如，我们实现 的 UDF 类为 org.shirdrn.pig.udf.IPAddressConverterUDF ，用来根据 ip 代码（long 类型）,转换为对应 的点分十 进制的 IP 地址字符串，打包后 JAR 文件名称 为 iptool.jar ，则 可以在 Pig 脚本中 这样 使用：

```
REGISTER 'iptool.jar';
a = LOAD '/data/etl/$date_string/$event_file' AS
(event_code: long, udid: chararray, ip_code: long,
network: chararray);
b = FOREACH a GENERATE event_code, udid,
org.shirdrn.etl.pig.udf.IPAddressConverterUDF(ip_code);
DUMP b;
```

也可以 实现 一个自定 义的累加器（Accumulator ）或者过滤 器，或者其他一些功能，可以 实现相关的接口： Algebraic ，Accumulator ，FilterFunc ，LoadFunc ，StoreFunc ，具体可以参考相关文档或 资 料。

另外，也可以通 过 在 PiggyBank 在来 查找其它用 户 分享的 UDF，可以在 http://svn.apache.org/repos/asf/pig/trunk/contrib/piggybank 中找到。

相关 问题总结

- 运行 Pig 脚本，从外部向脚本 传递 参数

在实际使用中，我们经常需要从外部传递参数到 Pig 脚本中，例如，文件路径，或者日期时间，等等，可以直接使用 pig 命令的 -p 参数从外部传参，例如，有下面的 Pig 脚本 compute_event_user_count.pig ：

```
a = LOAD
'/data/etl/hive_input/20150613/basis_event_2015061305-
r-00001' USING PigStorage('\t');
DESCRIBE a;
b = FOREACH a GENERATE $1 AS event_code, $2 AS udid,
$4 AS install_id;
c = GROUP b BY (event_code, udid);
d = FOREACH c GENERATE group, COUNT($1);
DUMP d;
```

上面我们是直接将文件路径写死在脚本中，如果需要从外部传递输入文件、输出目录，则可以改写为：

```
a = LOAD '$input_file' USING PigStorage('\t');
DESCRIBE a;
b = FOREACH a GENERATE $1 AS event_code, $2 AS udid,
$4 AS install_id;
c = GROUP b BY (event_code, udid);
d = FOREACH c GENERATE group, COUNT($1);
STORE d INTO '$output_dir' USING PigStorage ('\t');
```

则可以执行 Pig 脚本，并传递参数：

```
bin/pig -p
input_file=/data/etl/hive_input/20150613/basis_event_2
015061305-r-00001 -p
output_dir=/test/shiyj/pig/example_output -x
mapreducecompute_event_user_count.pig
```

这样就可以实现从外部向 Pig 脚本传递参数，可以到 HDFS 上查看结果输出文件 /test/shiyj/pig/example_output/part-r-00000 。pig 命令更多选项，可以查看 pig 帮助命令：

```
bin/pig -h
```

- 运行 Pig 脚本出现异常“ Retrying connect to server: 0.0.0.0/0.0.0.0:10020 ”

实际应用中，我们几乎不可能将 Pig 安装到 Hadoop 集群的 NameNode 所在的节点，如果可以安装到 NameNode 节点上，基本不会报这个错误的。这个错误主要有是由于 Pig 没有安装在 NameNode 节点上，而是以外的其它节点上，它在执行计算过程中，需要与 MapReduce 的 JobHistoryServer 的 IPC 服务进行通信，所以在安装 Hadoop 时需要允许 JobHistoryServer 的 IPC 主机和端口被外部其它节点访问，只需要修改 etc/hadoop/mapreduce-site.xml 配置文件，增加如下配置即可：

```
<property>
<name>mapreduce.jobhistory.address</name>
<value>10.10.4.130:10020</value>
<description>MapReduce JobHistory Server IPC
host:port</description>
```

</property>

如果第一次安装 Hadoop 没有配置 该属性 mapreduce.jobhistory.address ，那么 Hadoop 集群的所有 节点上会使用默 认的配置 值为 0.0.0.0:10020 ，所以如果不在 NameNode 上安装 Pig 程序，导致 Pig 所在的 节点上安装的 Hadoop 的配置属性 mapreduce.jobhistory.address 使用默 认值，也就是 Pig 所在 节点 0.0.0.0:10020 ，所以 Pig 脚本就会 执 行过 程中与本机的 10020 端口通信， 显 然会失 败 的。

其实，如果已经在 NameNode 上启 动了 JobHistoryServer 进 程，只需要修改 mapreduce.jobhistory.address 的属性 值，然后同步到所有安装 Hadoop 文件的 节 点，包括 Pig 所在 节 点即可，不需要重启 NameNode 节 点上的 JobHistoryServer 进 程。如果没有在 NameNode 上启 动 JobHistoryServer 进程，执行如下命令启 动即可：

```
mr-jobhistory-daemon.sh start historyserver
```