**Pig执行计划分析**

|  |  |
| --- | --- |
| **审核人** |  |
| **重要性** | 高 |
| **紧迫性** | 高 |
| **拟制人** | 彭锋 |
| **提交日期** | 2014-04-09 |

**修改记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **更新时间** | **变更内容** | **变更提出部门** | **变更理由** |
| 2014-04-09 |  |  | V1.0版 |
|  |  |  |  |

[1 概述 5](#_Toc385319134)

[2 名词解释 5](#_Toc385319135)

[3 Pig整体流程 5](#_Toc385319136)

[4 Loader/Store/Schema 5](#_Toc385319137)

[4.1 Loader概述 6](#_Toc385319138)

[4.2 PigStorage分析 6](#_Toc385319139)

[4.3 其他重要接口 7](#_Toc385319140)

[4.3.1 LoadMetadata 7](#_Toc385319141)

[4.3.2 LoadPushDown 7](#_Toc385319142)

[4.3.3 LoadCaster 7](#_Toc385319143)

[4.4 Store概述 7](#_Toc385319144)

[4.5 Schema结构 8](#_Toc385319145)

[4.6 待升级点 9](#_Toc385319146)

[5 从Pig Latin到Logical plan 9](#_Toc385319147)

[5.1 Parse过程 10](#_Toc385319148)

[5.2 Logical plan生成过程 11](#_Toc385319149)

[5.3 Logical plan结构 12](#_Toc385319150)

[6 Logical plan optimizer 14](#_Toc385319151)

[6.1 优化过程 14](#_Toc385319152)

[6.2 PartitionFilterOptimizer 15](#_Toc385319153)

[6.3 FilterLogicExpressionSimplifier 15](#_Toc385319154)

[6.4 SplitFilter 15](#_Toc385319155)

[6.5 PushUpFilter 16](#_Toc385319156)

[6.6 DuplicateForEachColumnRewrite 16](#_Toc385319157)

[6.7 FilterAboveForeach 16](#_Toc385319158)

[6.8 ImplicitSplitInserter 16](#_Toc385319159)

[6.9 MergeFilter 16](#_Toc385319160)

[6.10 PushDownForEachFlatten 16](#_Toc385319161)

[6.11 LimitOptimizer 16](#_Toc385319162)

[6.12 ColumnMapKeyPrune 16](#_Toc385319163)

[6.13 AddForEach 17](#_Toc385319164)

[6.14 MergeForEach 17](#_Toc385319165)

[6.15 GroupByConstParallelSetter 17](#_Toc385319166)

[6.16 其他优化 17](#_Toc385319167)

[6.16.1 Split&Multi-query execution 17](#_Toc385319168)

[6.16.2 Combiner 17](#_Toc385319169)

[6.16.3 Join 17](#_Toc385319170)

[7 从Logical plan到Physical plan. 18](#_Toc385319171)

[7.1 Physical plan生成过程 18](#_Toc385319172)

[7.2 Physical plan结构 20](#_Toc385319173)

[8 从Physical plan到Map-Reduce plan 22](#_Toc385319174)

[8.1 Map-Reduce plan结构 23](#_Toc385319175)

[9 Pig提供的实用工具 23](#_Toc385319176)

[9.1 Explain 23](#_Toc385319177)

[9.2 Illustrate 24](#_Toc385319178)

[10 其他 24](#_Toc385319179)

[10.1 Pigs Live Anywhere 24](#_Toc385319180)

[11 参考资料 24](#_Toc385319181)

# 概述

Pig是什么？本文的主要内容是什么？

本文的目的是什么？主要目的是借鉴Pig Latin on Hadoop，为设计Flare Latin on Spark提供参考

# 名词解释

|  |  |  |
| --- | --- | --- |
| 名词 | 解释 | 备注 |
| Pig Latin | Pig数据流处理语言 |  |
| Loader/Store | Pig用于加载和存储数据 |  |
| Schema | 加载数据时指定的数据格式 | Pig的数据类型分为标量和复杂类型，标量基本和Java基本数据类型一致，复杂类型包括Tuple（元组）、Map和Bag（元组的无序集合） |
| Relation | Pig操作的数据集合 | 即元组的集合，或者说就是一个Bag（更精确来说外部Bag，因为存在嵌套的内部Bag） |
| Logical plan | 逻辑执行计划 |  |
| Physical plan | 物理执行计划 |  |
| Optimizer | 优化器 | Pig现有的优化都是基于规则的 |
| DAG | Directed Acyclic Graph有向无环图 |  |

# Pig整体流程

这里来一张图吧

Parser🡪SementicChecks🡪Logic Optimizer🡪Logic To Physical Translator🡪Physical To MR Translator

# Loader/Store/Schema

Pig哲学之一——Pigs Eat Anything。Pig能够从不同数据源加载数据，能够处理不同格式的数据。Pig使用Loader/Store进行数据加载和存储，可选地使用Schema指定数据列名称和类型。如果加载数据时不指定Schema，数据列未命名，类型默认是字节数组(bytearray)，在后续操作中，Pig可以通过位置参数引用数据列，会根据在数据列上进行的操作进行自动类型转化。从性能和可读性考虑，最好在加载数据时指定Schema。

## Loader概述

Loader的基类是org.apache.pig.LoadFunc，规定了Loader需要实现的接口，并提供了一些默认实现。下图是Loader的继承体系，针对不同数据源，Pig实现了大量Loader，包括HBaseStorage和ParquestLoader等，能够处理列式存储。默认的Loader是PigStorage。

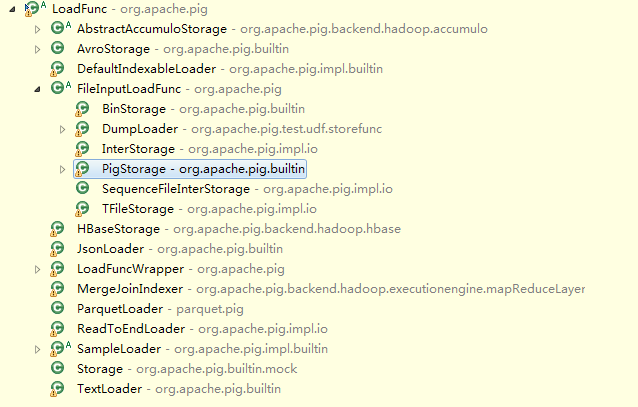


Figure 1

org.apache.pig.LoadFunc中的三个基本的方法决定了Where/What/How：

1. **public** **abstract** **void** setLocation(String location, Job job) **throws** IOException;指定加载位置。
2. **public** **abstract** InputFormat getInputFormat() **throws** IOException;指定数据源类型，使用HDFS的InputFormat处理不同数据源。
3. **public** LoadCaster getLoadCaster() **throws** IOException {

**return** **new** Utf8StorageConverter();

};如何处理数据从字节数组到实际类型的转化，默认使用Utf8StorageConverter

## PigStorage分析

1. 处理压缩格式，通过加载文件后缀加载不同的InputFormat：

@Override

**public** InputFormat getInputFormat() {

**if**(loadLocation.endsWith(".bz2") || loadLocation.endsWith(".bz")) {

**return** **new** Bzip2TextInputFormat();

} **else** {

**return** **new** PigTextInputFormat();

}

}

1. 读取数据：读取数据之前先调用**public** **void** prepareToWrite(RecordWriter writer) 设置InputFormat对应的RecordWriter，通过RecordWriter读取每行数据，再通过用户指定的分隔符处理每行文本，最终转换成元组。

@Override

**public** Tuple getNext() **throws** IOException

1. Schema处理，在getNext方法中，如果存在Schema，会对元组应用applySchema方法，给元组中的数据项指定名称和类型。

## 其他重要接口

通过实现其他一些接口，Loader能提供一些附加功能

### LoadMetadata

1. 通过getSchema方法自动加载Schema
2. 通过getPartitionKeys方法设置数据的分区键，把用户查询条件中的分区键通过setPartitionFilter直接传递给Loader，减少数据加载。参见HCatLoader中实现。

### LoadPushDown

在使用RCFile等基于列格式文件时，如果每次都加载所有列对性能影响较大。如果实现了LoadPushDown接口，优化器会将所需要用到的字段传递给pushProjection方法。

### LoadCaster

自定义字节数组到Schema中数据类型的转换，通过一系列方法能够自定义字节数组到到Pig的标量和复杂数据类型的转化。默认实现为Utf8StorageConverter，其中的复杂数据类型格式固定，比如元组格式为（），map格式为[]，bag为{}。

## Store概述

与org.apache.pig.LoadFunc对应，Pig中也存在org.apache.pig.StoreFunc抽象类。由于不少Loader（比如默认的PigStorage）同样实现了store功能，受Java单继承的限制，Pig提供了StoreFuncInterface接口。

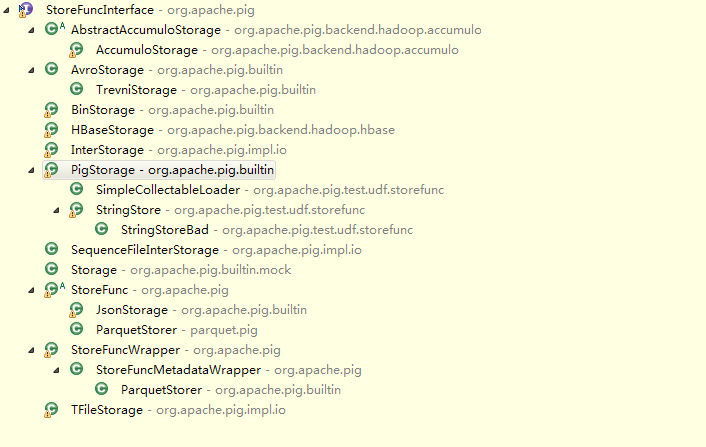


Figure 2

Store的实现与Loader对应，将实际输出操作委托给OutputFormat。值得注意的是，与LoadMetadata对应，Pig也提供了StoreMetadata接口用于处理元数据的存储。.

## Schema结构

Schema描述了一个数据集合每一行的列名称和数据类型，其中每一个列信息用FieldSchema表示。FieldSchema通常包括列名称、数据类型，如果列本身是bag的话，FieldSchema还会拥有自己的Schema。



Figure 3

## 待升级点

目前Pig针对逻辑执行计划的优化器都是基于规则的，如果要实现基于代价的优化，需要更多关于数据的统计信息，那么加载和存储数据应该是重要入口和出口。在LoadMetadata/StoreMetadata接口中已经存在getStatistics/setStatistics方法，不过现在在PigStorage中是空实现。

# 从Pig Latin到Logical plan

Pig基于Antlr进行语法解析，生成逻辑执行计划。逻辑执行计划基本上与Pig Latin中的操作步骤一一对应，以DAG形式排列。

以如下代码为例进行分析，包含了load、filter、join、group、foreach、count函数和stroe等常用操作。

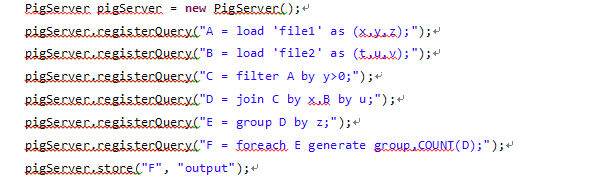


Figure 4

生成的逻辑执行计划如下图所示：



Figure 5

## Parse过程

QueryLexer.g和QueryParser.g分别是Pig Latin使用的词法文件和语法文件。

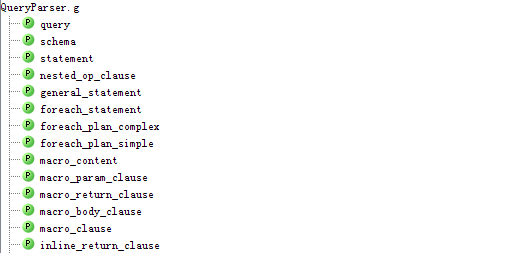


Figure 6

Pig不仅仅使用 Anltr生成的词法分析器和语法分析器，校验用户输入合法性（AstValidator）。还同时做了两件事情（具体细节不在此展开）

1. 在语法文件中嵌入动作，加入Java代码，对表达式做进一步处理。
2. 使用了Antlr 的抽象语法树语法，在语法分析的同时将用户输入转换成抽象语法树，如

foreach\_plan\_complex : LEFT\_CURLY nested\_blk RIGHT\_CURLY -> ^( FOREACH\_PLAN\_COMPLEX nested\_blk )

下图是JOIN语句语法规则的可视化表示

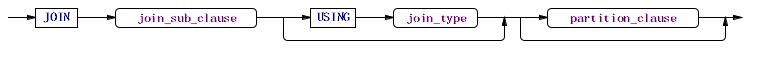


Figure 7

Parse时序图如下（省略了宏展开，用户REGISTER语句替换等细节，其中QueryLexer，QueryParser和AstValidator都是antlr生成的类）：



Figure 8

## Logical plan生成过程

[LogicalPlanGenerator.g](http://svn.apache.org/repos/asf/pig/trunk/src/org/apache/pig/parser/LogicalPlanGenerator.g)是一个树分析器文件，antlr生成LogicalPlanGenerator.java文件，实现org.antlr.runtime.tree.TreeParser接口，会对QueryParser.g对应的抽象语法树进行语义处理，用来生成逻辑执行计划（具体细节不在此展开）。

在本节开始图4中的运行Pig Latin程序中，每次调用pigServer.registerQuery方法，注册一个查询语句，Pig都会启动解析、验证步骤，然后调用LogicalPlanGenerator.query()方法，生成该条语句对应的逻辑执行子计划。直到调用pigServer.store方法，才会生成一个完整的逻辑执行计划，触发下一阶段操作。

以上过程信息，包括最后完成的逻辑执行计划都存储在PigServer的**private** Graph currDAG成员变量中，如果在更复杂的批量执行模式下，比如脚本里边调用其他脚本，需要队列来进行存储。

/\*

\* The data structure to support grunt shell operations.

\* The grunt shell can only work on one graph at a time.

\* If a script is contained inside another script, the grunt

\* shell first saves the current graph on the stack and works

\* on a new graph. After the nested script is done, the grunt

\* shell pops up the saved graph and continues working on it.

\*/

**protected** **final** Deque<Graph> graphs = **new** LinkedList<Graph>();

其中currDAG中存储的具体信息如下：lp代表完整的逻辑执行计划，operator可以理解为计划中的特定步骤，它们的关系和内部结构将在下一节介绍。

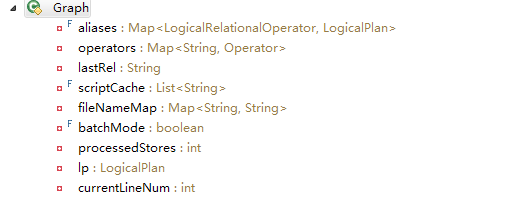


Figure 9

## Logical plan结构

下面是完整的逻辑执行计划，可以观察到自底而上，分别对应最初是的Load操作和最后的Store操作。

#-----------------------------------------------

# New Logical Plan:

#-----------------------------------------------

F: (Name: LOStore Schema: group#38:bytearray,#47:long)

|

|---F: (Name: LOForEach Schema: group#38:bytearray,#47:long)

| |

| (Name: LOGenerate[false,false] Schema: group#38:bytearray,#47:long)

| | |

| | group:(Name: Project Type: bytearray Uid: 38 Input: 0 Column: (\*))

| | |

| | (Name: UserFunc(org.apache.pig.builtin.COUNT) Type: long Uid: 47)

| | |

| | |---D:(Name: Project Type: bag Uid: 44 Input: 1 Column: (\*))

| |

| |---(Name: LOInnerLoad[0] Schema: group#38:bytearray)

| |

| |---D: (Name: LOInnerLoad[1] Schema: C::x#36:bytearray,C::y#37:bytearray,C::z#38:bytearray,B::t#41:bytearray,B::u#42:bytearray,B::v#43:bytearray)

|

|---E: (Name: LOCogroup Schema: group#38:bytearray,D#44:bag{#50:tuple(C::x#36:bytearray,C::y#37:bytearray,C::z#38:bytearray,B::t#41:bytearray,B::u#42:bytearray,B::v#43:bytearray)})

| |

| C::z:(Name: Project Type: bytearray Uid: 38 Input: 0 Column: 2)

|

|---D: (Name: LOJoin(HASH) Schema: C::x#36:bytearray,C::y#37:bytearray,C::z#38:bytearray,B::t#41:bytearray,B::u#42:bytearray,B::v#43:bytearray)

| |

| x:(Name: Project Type: bytearray Uid: 36 Input: 0 Column: 0)

| |

| u:(Name: Project Type: bytearray Uid: 42 Input: 1 Column: 1)

|

|---C: (Name: LOFilter Schema: x#36:bytearray,y#37:bytearray,z#38:bytearray)

| | |

| | (Name: GreaterThan Type: boolean Uid: 40)

| | |

| | |---(Name: Cast Type: int Uid: 37)

| | | |

| | | |---y:(Name: Project Type: bytearray Uid: 37 Input: 0 Column: 1)

| | |

| | |---(Name: Constant Type: int Uid: 39)

| |

| |---A: (Name: LOLoad Schema: x#36:bytearray,y#37:bytearray,z#38:bytearray)RequiredFields:null

|

|---B: (Name: LOLoad Schema: t#41:bytearray,u#42:bytearray,v#43:bytearray)RequiredFields:null

LogicalPlan继承自BaseOperatorPlan，如下面代码片段所示，LogicalPlan由一系列Operator组成，fromEdges和toEdges代表有向图的出边和入边，PlanEdge内部实际是一个MultiMap<Operator, Operator>数据结构。

**public** **abstract** **class** BaseOperatorPlan **implements** OperatorPlan {

**protected** List<Operator> ops;

**protected** PlanEdge fromEdges;

**protected** PlanEdge toEdges;

……

}

Operator对应逻辑执行计划中的具体操作步骤。Pig为每种操作都实现了相应的Operator。

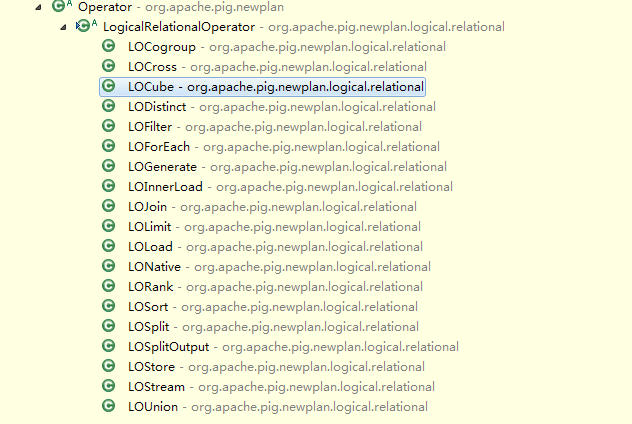


Figure 10

在Parse过程中，LogicalPlanBuilder负责构建Operator。Operator中包含名称、Schema、包装成LogicalExpressionPlan的运行参数等信息（以及requestedParallelism、mCustomPartitioner等用户自定义的Hadoop MapReduce运行配置信息），

# Logical plan optimizer

## 优化过程

Pig哲学之二——Pigs Are Domestic Animals。用户拥有足够的控制权。具体到逻辑执行计划的优化上，用户可以根据自己情况选择适合的优化规则（也可以理解为优化这块还大有潜力可挖）。

逻辑执行计划在编译成物理执行计划之前，会被LogicalPlanOptimizer处理，和一系列优化规则进行匹配，匹配上的优化规则会对原有执行计划进行变换，最终产生优化后的逻辑执行计划。整个过程如图所示：



Figure 11

下文逐个介绍每个优化规则。

## PartitionFilterOptimizer

将分区过滤条件下推到Loader（需要Loader支持，比如HCatLoader支持分区字段推送，详见4.3.1 LoadMetadata）

## FilterLogicExpressionSimplifier

简化filter语句中的逻辑条件表达式，其中规则较多，委托给LogicalExpressionProxy进行处理：常量计算，根据摩根定律变换and/or操作和使用DNF标准化逻辑公式等。

## SplitFilter

将filter语句中的条件分割，以便让他们分别下推。比如：

A = LOAD 'input1' as (a0, a1);

B = LOAD 'input2' as (b0, b1);

C = JOIN A by a0, B by b0;

D = FILTER C BY a1>0 and b1>0;

D中对a和b的过滤条件可以分割开，以便于这两个过滤条件能够分别被下推。

X = FILTER C BY a1>0;

D = FILTER X BY b1>0;

## PushUpFilter

将过滤条件下推（沿着数据流DAG图上推），减少数据传输量

## DuplicateForEachColumnRewrite

重写foreach语句中重复的字段？why用于udf？

## FilterAboveForeach

从foreach语句中移除与之前操作重复的过滤条件

## ImplicitSplitInserter

插入split语句，（原因详见下文“其他优化”中split部分）

## MergeFilter

在PushUpFilter之后，合并过滤条件，减少filter语句

## PushDownForEachFlatten

将foreach中的flatten往后放（沿着数据流DAG图下推），能够减少后续join等操作的数据量。因为如果flatten对bag操作，一条记录会生成多条记录，降低后续join操作的性能，优化后，会将flatten操作放在join操作之后。

## LimitOptimizer

Limit语句下推，尽早减少数据传输量。

## ColumnMapKeyPrune

裁剪不必要的加载列。需要Loader支持，否则，会在Load语句之后增加一个foreach语句来进行处理。

用到的辅助类：ColumnPruneVisitor和ColumnPruneHelper

## AddForEach

裁剪后续不会被用到的列，比如某个列在某个语句之后没有被用到，会在这条语句之后增加一个foreach语句进行裁剪。

## MergeForEach

将多个连续的foreach语句合并成一个

## GroupByConstParallelSetter

一些Group语句，比如group all，或者group的字段是常量，这种情况下，不能使用多个reduce并行，那么 parallel参数会被设置成1。

## 其他优化

### Split&Multi-query execution

在之前的例子中，严格来讲，执行计划都是树形结构，即多个操作最终产生一个结果，在显示使用Split语句或者逻辑执行计划被前文提到的ImplicitSplitInserter优化后，一个操作会产生多个输出结果。

SPLIT A INTO B IF ..., C IF ...

在这种情况下，Pig将A的数据同时通过两个管道生成B和C数据，即Multi-query execution。优点是只用加载一次数据A，付出的代价是会占用更多的内存，如果内存不够，数据需要写入磁盘的话，性能反而会降低。

### Combiner

先省略。。待补充

### Join

Pig支持多种join实现方式。同样因为Pig不支持基于成本优化器，所以控制权依然交给了用户，需要在在join语句中使用replicated/ merge/merge-sparse/skewed关键词选择不同的join策略。前三种Join实现只针对特定的场景，所以不在此展开。而在现实世界中，数据倾斜现象经常发生，所以介绍一下Pig针对数据倾斜所做的优化。

Join操作需要将key相同的数据放在一起处理，当出现数据倾斜时，某个节点处理大部分数据，会极大地拖慢整个任务的完成时间。Pig的解决思路是：先对数据进行取样，识别出数据倾斜严重的Key值，针对正常分布的数据，进行正常的Join操作，这部分数据倾斜的Key值则分发到多个节点上进行join操作。以上过程实现请参考Pig提供的SkewedPartitioner。

Pig的skewed join算法仅适用于参与join操作一方出现数据倾斜的场景。

# 从Logical plan到Physical plan.



Figure 12

## Physical plan生成过程

优化后的逻辑执行计划被[LogToPhyTranslationVisitor](http://svn.apache.org/repos/asf/pig/trunk/src/org/apache/pig/newplan/logical/relational/LogToPhyTranslationVisitor.java)处理，生成物理执行计划。

这是一个经典的Vistor设计模式应用场景。



Figure 13

其中，[LogToPhyTranslationVisitor](http://svn.apache.org/repos/asf/pig/trunk/src/org/apache/pig/newplan/logical/relational/LogToPhyTranslationVisitor.java)的visit()入口方法中，通过DependencyOrderWalker遍历处理逻辑执行计划中的每一个LogicalRelationalOperator。DependencyOrderWalker按照依赖顺序遍历DAG中节点，即保证当且仅当节点的所有前驱都被访问后，它才会被访问。核心逻辑如下，doAllPredecessors递归调用自己，将符合无前驱条件的节点添加到fifo队列中，最终实现的效果等效于将图拓扑排序后顺序访问。

**public** **void** walk(PlanVisitor visitor) **throws** FrontendException {

List<Operator> fifo = **new** ArrayList<Operator>();

Set<Operator> seen = **new** HashSet<Operator>();

List<Operator> leaves = plan.getSinks();

**if** (leaves == **null**) **return**;

**for** (Operator op : leaves) {

doAllPredecessors(op, seen, fifo);

}

**for** (Operator op: fifo) {

op.accept(visitor);

}

}

接下来，每个LogicalRelationalOperator又反过来调用[LogToPhyTranslationVisitor](http://svn.apache.org/repos/asf/pig/trunk/src/org/apache/pig/newplan/logical/relational/LogToPhyTranslationVisitor.java)相应的visit方法对自身进行处理，转化成PhysicalOperator。最终生成完整的逻辑执行计划。下图是[LogToPhyTranslationVisitor](http://svn.apache.org/repos/asf/pig/trunk/src/org/apache/pig/newplan/logical/relational/LogToPhyTranslationVisitor.java)中所有的visit operator方法。

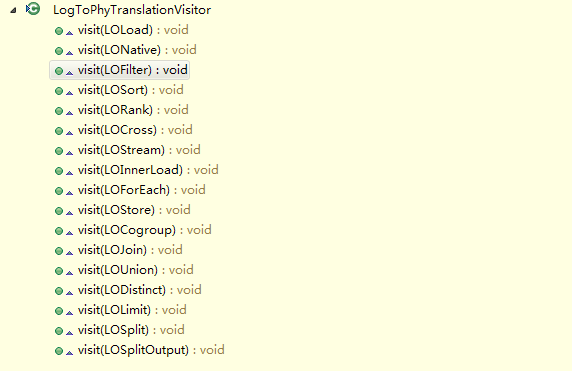


Figure 14

## Physical plan结构

下面是完整的物理执行计划。物理执行计划与逻辑执行计划结构类似，部分Operator一一对应，但存在几个明显区别：

1. 物理执行计划中包含了实际使用的Loader和Store，以及要操作的文件实际路径。
2. Group操作被分成了三部分：Local Rearrage，Global Rearrange和Package。（分别对应map-reduce中的map、shuffle和reduce）
3. 非replicate的join操作先被转换成CoGroup和Foreach操作，然后CoGroup操作与Group操作类似，也被转换为Local Rearrage，Global Rearrange和Package三步。

F: Store(output:org.apache.pig.builtin.PigStorage) - scope-28

|

|---F: New For Each(false,false)[bag] - scope-27

| |

| Project[bytearray][0] - scope-22

| |

| POUserFunc(org.apache.pig.builtin.COUNT)[long] - scope-25

| |

| |---Project[bag][1] - scope-24

|

|---E: Package[tuple]{bytearray} - scope-19

|

|---E: Global Rearrange[tuple] - scope-18

|

|---E: Local Rearrange[tuple]{bytearray}(false) - scope-20

| |

| Project[bytearray][2] - scope-21

|

|---D: New For Each(true,true)[tuple] - scope-17

| |

| Project[bag][1] - scope-15

| |

| Project[bag][2] - scope-16

|

|---D: Package[tuple]{bytearray} - scope-10

|

|---D: Global Rearrange[tuple] - scope-9

|

|---D: Local Rearrange[tuple]{bytearray}(false) - scope-11

| | |

| | Project[bytearray][0] - scope-12

| |

| |---C: Filter[bag] - scope-1

| | |

| | Greater Than[boolean] - scope-5

| | |

| | |---Cast[int] - scope-3

| | | |

| | | |---Project[bytearray][1] - scope-2

| | |

| | |---Constant(0) - scope-4

| |

| |---A: Load(file:///D:/Develop/projects/pig/file1:org.apache.pig.builtin.PigStorage) - scope-0

|

|---D: Local Rearrange[tuple]{bytearray}(false) - scope-13

| |

| Project[bytearray][1] - scope-14

|

|---B: Load(file:///D:/Develop/projects/pig/file2:org.apache.pig.builtin.PigStorage) - scope-6

PhysicalPlan类代表物理执行计划，继承自OperatorPlan。（继承时会使用PhysicalOperator替换下面代码片段中泛型参数E）

**public** **abstract** **class** OperatorPlan<E **extends** Operator> **implements** Iterable<E>, Serializable, Cloneable {

**protected** Map<E, OperatorKey> mOps;

**protected** Map<OperatorKey, E> mKeys;

**protected** MultiMap<E, E> mFromEdges;

**protected** MultiMap<E, E> mToEdges;

}

# 从Physical plan到Map-Reduce plan

入口类[MRCompiler](http://svn.apache.org/repos/asf/pig/trunk/src/org/apache/pig/backend/hadoop/executionengine/mapReduceLayer/MRCompiler.java)，待补充。

注：因为我们要实现的是Spark之上的执行计划，这部分参考意义不大，所以本部分仅供参考，不会深入分析。

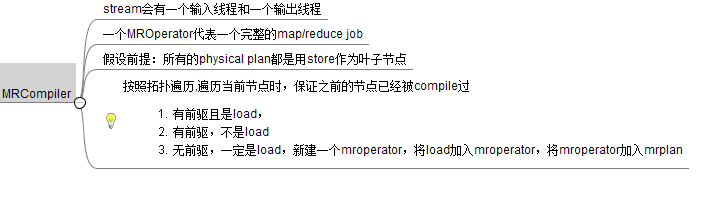




Figure 15

## Map-Reduce plan结构

MROperator类，待补充

# Pig提供的实用工具

## Explain

待补充，主要参考价值是dot部分。

## Illustrate

待补充，主要参考价值是自动生成适合执行计划的样本数据。

# 其他

## Pigs Live Anywhere

Pig哲学之三——Pigs Live Anywhere。理论上，Pig并不被限定运行在Hadoop框架上,有几个可以参考的实现和提议。

1. Pigen，Pig on Tez。<https://github.com/achalsoni81/pigeon>

架构图如下：

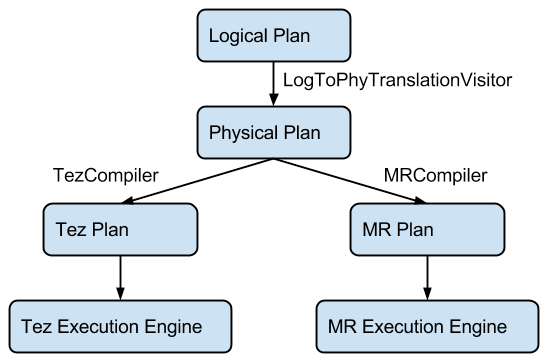


Figure 16

1. Pig的后端抽象层：<https://wiki.apache.org/pig/PigAbstractionLayer>。

目前已经实现了Pig Latin运行在Galago上。<http://www.galagosearch.org/>

# 参考资料

Pig官网：<http://pig.apache.org/>

Pig论文：Pig Latin paper at SIGMOD 2008