**Distributed Computing**

|  |  |
| --- | --- |
| **审核人** |  |
| **重要性** | 中 |
| **紧迫性** | 中 |
| **拟制人** | 张包峰 |
| **提交日期** |  |

**作者：张包峰**

(版权所有,翻版必究)

**修改记录**

|  |  |  |
| --- | --- | --- |
| **更新时间** | **变更内容** | **变更理由** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

[**作者：张包峰** 1](#_Toc394396285)

[1.1 FlumeJava 4](#_Toc394396286)

[1.1.1 核心抽象和基本原语 4](#_Toc394396287)

[1.1.2 衍生原语(Derived Operations) 4](#_Toc394396288)

[1.1.3 延迟分析(Deffered Evaluation) 5](#_Toc394396289)

[1.1.4 PObjects 5](#_Toc394396290)

[1.1.5 优化器 5](#_Toc394396291)

[1.1.6 Executor 8](#_Toc394396292)

[1.2 Tenzing 9](#_Toc394396293)

[1.2.1 背景 9](#_Toc394396294)

[1.2.2 架构概述 9](#_Toc394396295)

[1.2.3 SQL特性 9](#_Toc394396296)

[1.2.4 性能 10](#_Toc394396297)

[1.2.5 LLVM query engine 11](#_Toc394396298)

[1.3 DryadLINQ 11](#_Toc394396299)

[1.4 Sawzall 11](#_Toc394396300)

[1.5 Dremel 11](#_Toc394396301)

[1.6 SCOPE 11](#_Toc394396302)

[1.7 HadoopDB 11](#_Toc394396303)

## FlumeJava

原始的MapReduce，分Map，Shuffle，Reduce。Map里包括shards。Shuffle理解为groupByKey的事情。Reduce里包括Combiner，可以定义Sharder来控制key怎么和Reducer worker对应起来。

### 核心抽象和基本原语

PCollection<T>是一个不可变的bag，可以是有序的(Sequence)，也可以是无序的(Collection)。PCollection可以来自于内存里的Java PCollection对象，也可以读取自文件。

PTable<K, V>，可以看成PCollection<Pair<K, V>>，不可变无序multi-map。

第一个原语是parallelDo()，把PCollection<T>变成新的PCollection<S>，处理方式定义在DoFn<T, S>里。emitFn是call-back，传给用户的process(…)，使用emitFn.emit(outElem)发射出去。parallelDo()可以在map或reduce中使用，DoFn不应该使用闭包外全局的变量，(inline function)纯操作自己的inputs。

第二个原语是groupByKey()，把PTable<K, V>转变成PTable<K, Collection<V>>，

第三个原语是combineValues()，接收input为PTable<K, Collection<V>>和一个V的符合结合律的方法，返回PTable<K, V>。

第四个原语是flatten()，接收一个PCollection<T>的list，返回一个PCollection<T>，

### 衍生原语(Derived Operations)

count()，接收PCollection<T>，返回PTable<T, Integer>

实现方式为parallelDo()，groupByKey()和combineValues()

join()，接收PTable<K, V1>，PTable<K, V2>，返回PTable<K, Tuple2<Collection<V1>, Collection<V2>>

实现方式为，第一步，使用parallelDo()把每个input PTable<K, Vi>变成通用的PTable<K, TaggedUnion2<V1, V2>>；第二步使用flattern来combine tables；第三步，使用groupByKey()作用于被扁平过了tables，产生PTable<K, Collection<TaggedUnion2<V1, V2>>>

top()，接收比较函数和N，

实现方式为parallelDo()，groupByKey()和combineValues()

### 延迟分析(Deffered Evaluation)

PCollection对象有两种状态，defferred或materialized。

FlumeJava.run()真正触发execution plan的物化/执行。

### PObjects

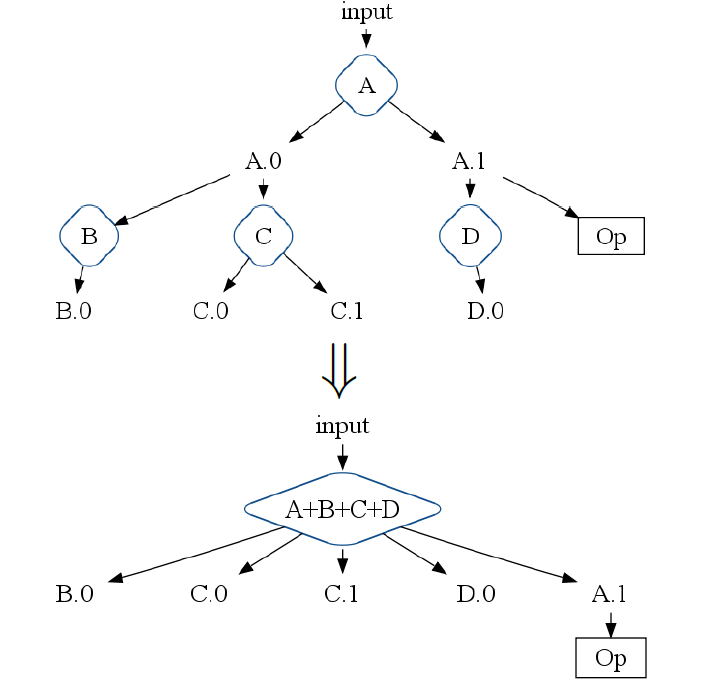
PObject<T>用于存储Java对象，物化过了之后可以使用getValue()方法获得PObject的值。有点像Future。

operate()方法

### 优化器

**parallelDo Fusion(融合)**

Producer-Consumer and Sibling Fusion，如下图



大致是说，ABCD这几种由同一份input产生的parallelDo，可以融合起来在一个parallelDo，即A+B+C+D，里处理。一些中间结果也可以不要。

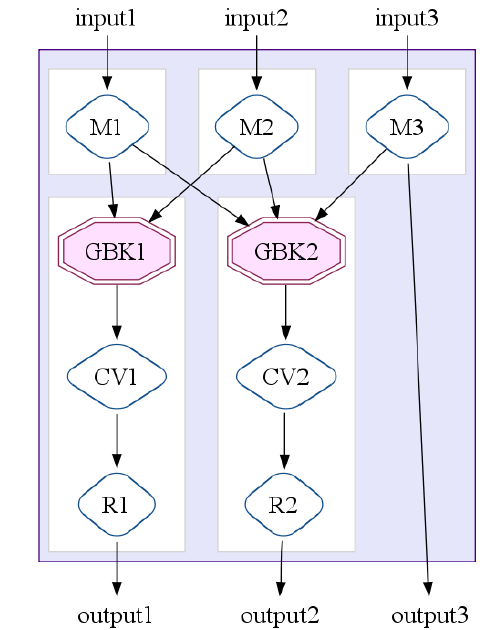
**MapShuffleCombineReduce(MSCR) Operation**

FlumeJava优化器的核心在于把ParallelDo，GroupByKey，CombineValues和Flattern的组合转换成一个个单个的MapReduce。

MSCR是一个中间层的操作，有M个input channels(每个可以进行map操作)，有R个Reduce channels(每个可以进行shuffle，或combine，或reduce操作)。单个input channal m，接收PCollection<Tm>作为输入，执行R路output输出的ParallelDo “map”操作，产生R个PTable<Kr, Vs> outputs。每个output channel r flatterns它的M个inputs，然后

1. 进行一次GroupByKey的“shuffle”，或CombineValues的“combine”，或Or-output的ParallelDo “reduce”，然后把结果写出到Or-output PCollections
2. 把inputs直接写出为outputs

前者这样的output channel称为”Grouping” channel，后者称为”pass-through” channel。”pass-through” channel允许map的output成为一个MSCR操作的输出。



每个MSCR操作可以用一个MapReduce完成。它让MapReduce更加通用，体现在：

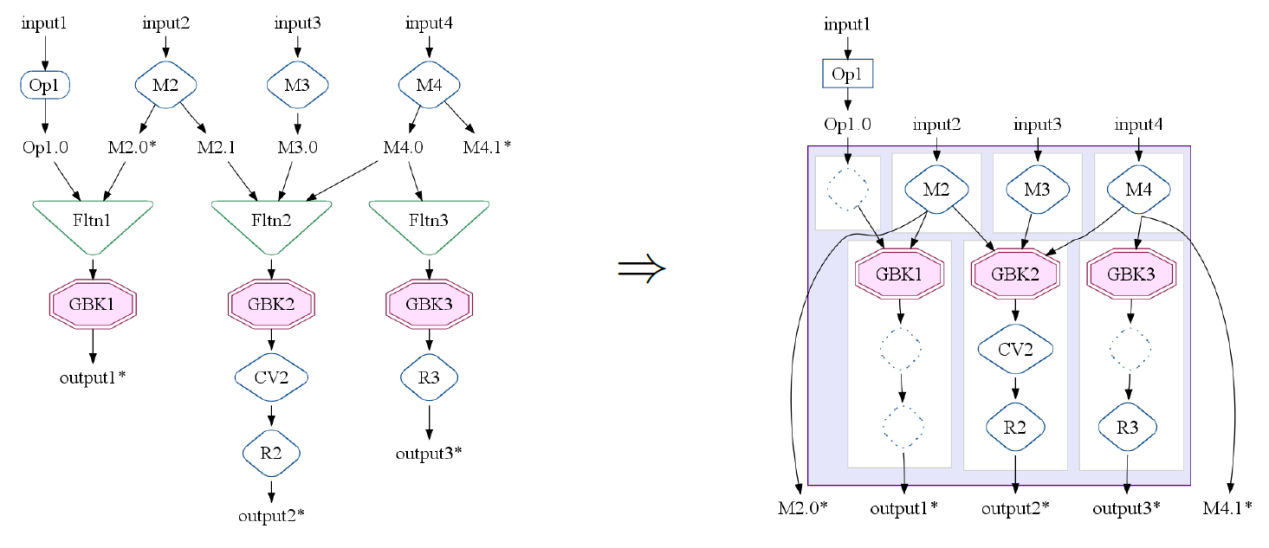
* 允许多个reducers和combiners；
* 允许每个reducer产生多个outputs；
* 消除了每个reducer必须以相同的key为input来产出output的约束；
* 允许pass-through形式的outputs。

所以MSCR是优化器里很好的一个中间操作目标。

**MSCR Fusion**

MSCR操作产生于一些相关的GroupByKey操作集合，相关的GroupByKey操作是指产生于相同的input(如Flattern操作)，或被同一个parallelDo操作制造出来的input。

这部分比较晦涩难懂啊，但是是理解核心

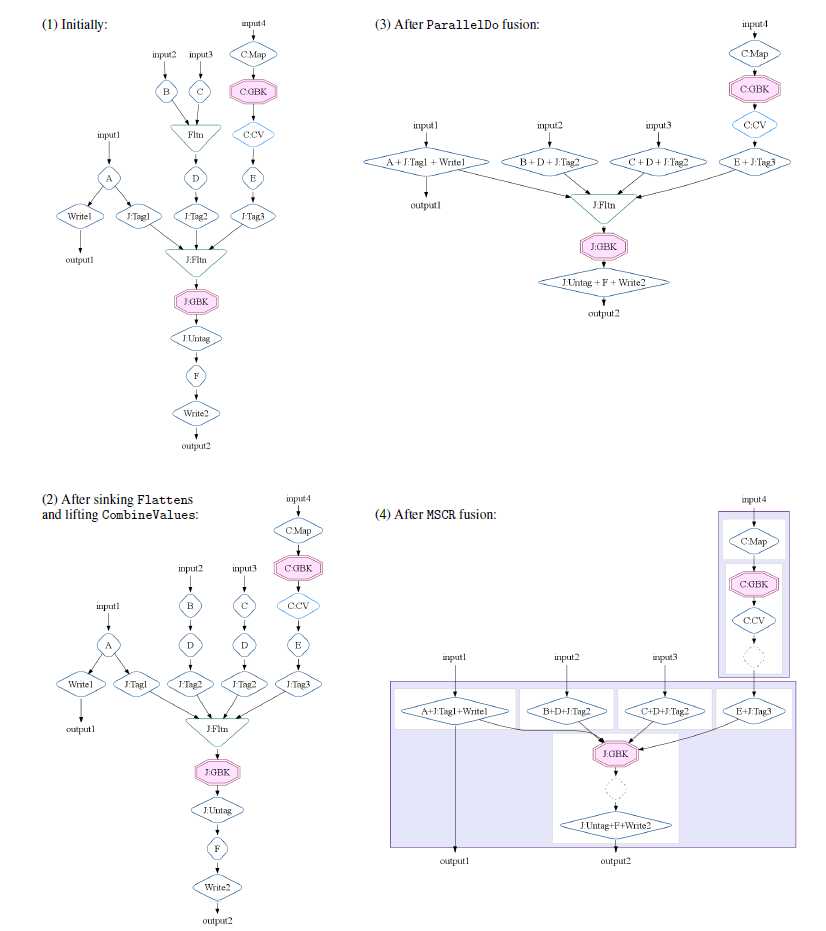


**全局优化策略**

优化要达到的效果是最后的执行计划里包含尽可能少的又高效的MSCR操作。

1. Sink Flatterns。把扁平操作下沉，如h(f(a)+f(b)) => h(f(a))+h(f(b))，即分配律，然后又能和parallelDo的融合特性结合起来，如(h*o*f)(a)+(h*o*g)(b)
2. Lift CombineValues。如果CombineValues紧跟着GroupByKey操作。
3. Insert fusion blocks。如果俩GroupByKey操作是由生产者-消费者的ParallelDo chain连起来的，ParallelDo要在GroupByKey里做上调和下移。
4. Fuse ParallelDos。
5. Fuse MSCRs。

针对这几个策略的实施，后面举了个例子并且描绘了具体的执行图，非常帮助理解



**优化的不足和未来工作**

优化器没有分析用户写的方法，比如估算input和output数据量大小。

也没有修改用户的代码来做优化。

需要做一些分析避免运算的重复，及去除不必要或不合理的groupByKey。

### Executor

优化完了之后是执行。目前支持的是batch的模式提交作业。

在执行方面，FlumeJava会做方便用户开发、debug，自动创建删除文件，自动识别数据量大小调整执行并行度和改变执行模式(remote)等等事情。

## Tenzing

### 背景

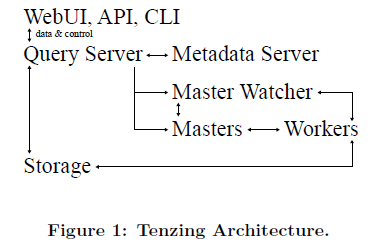
异构数据的适配及数据可扩展性，资源可扩展性，廉价机器，SQL查询。

### 架构概述

Worker pool，query server，client interfaces，metadata server

Worker线程是long-running的，worker pool包括master节点和worker节点，以及一个master watcher。

Query server把query解析，优化之后传给master执行，优化包括一些基本的规则优化和基于成本的优化。



### SQL特性

SQL语法主要是SQL92，并增加了一些高级分析，增强的部分也是适合MR完全可并行化的。

内嵌了Sawzall语法，支持用户写的和Sawzall自带的方法。

在映射和过滤方面，

常量表达式在compile阶段就计算；

谓词是常数或者查找是常数范围，那么可以直接下推给Bigtable这样的数据源；

如果谓词不包含复杂的UDF，数据源是mysql这样的数据库，那么也下推给数据源；

如果是分区的，那么也会跳过其他分区数据；

根据ColumnIO的meta信息，跳过范围外的columns数据；

如果下面数据源是面向列的，那么Tenzing会跳过无关列。

聚合函数方面，

支持 sum，count，min，max，distinct，count distinct，

还支持统计型的corr，covar，stddev

在实现上加了一些额外优化，如在mapreduce里实现基于hash table的聚合。

Join方面，

Tenzing支持跨数据源的各种join方式。

Broadcast joins，基于成本的优化器会发现足够小的table，广播到内存里，让mapper和reducer可以直接内存访问到。还有几个细节的优化点。

Remote lookup joins，如Bigtable支持基于index的key查找，那么就可以实时。

Distributed sort-merge joins。

Distributer hash joins，适合两表都放不了内存，且一张表比另一张大许多，且在join的key上都没有索引的情况。具体实现和优化细节，见论文伪代码及说明，很清晰。

分析函数，

类似PostgreSQL/Oracle，支持rank，sum，min，max，lead，lag，ntile。

OLAP扩展，

支持rollup，cube。

集合操作，

支持标准sql集合操作，如union，union all，minus，minus all。

嵌套查询和子查询，

会优化mr个数

处理结构化数据，

根据pb协议，有的sql支持有的不行

Views，

支持create views

DML，

支持批模式下的insert，update，delete，不具备ACID性质。

DDL，

支持create table，drop table，rename table，generate statistics，grant，revoke。

Table valued functions，

Data formats，

### 性能

性能目标是能与传统MPP数据库系统比较。

对mapreduce的优化和增强，

Work pool的设计，目的是减小latency，具体master watcher，master pool和worker pool的指责。

Streaming & In-memory chaining，MRs之间的数据衔接用流的方式；前一个mapper和后一个reducer可以在同一个进程里。

避免sort，一些hash join，hash聚合要shuffle，不要sort，增加了关闭sort开关。

块shuffle，原本基于row的shuffle是为了sort，在不需要sort的情况下，大约1M块的shuffle会高效3X倍。

本地执行，根据数据量大小(低于128M)决定可以本地执行。

### LLVM query engine

一代引擎把sql翻译为Sawzall code

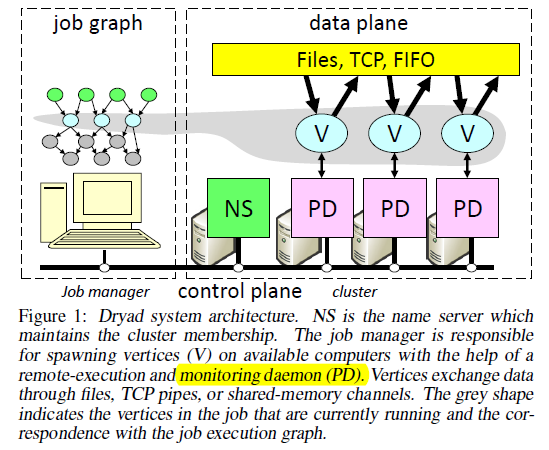
二代引擎是使用Dremel的sql表达式分析引擎

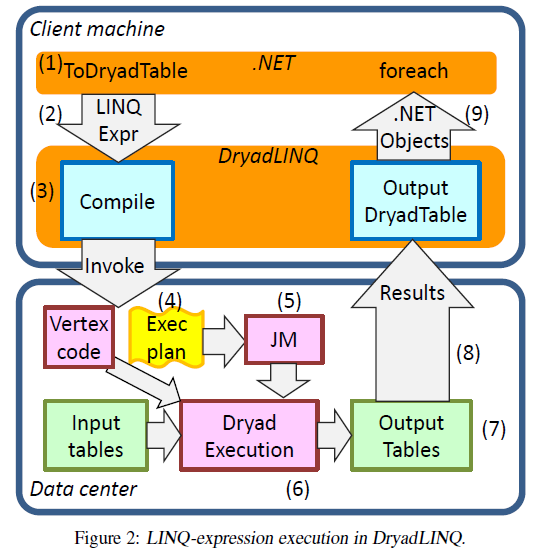
三代引擎尝试使用LLVM based row和vector based column

指出了一些优缺点，相信native code generation引擎是未来方向。

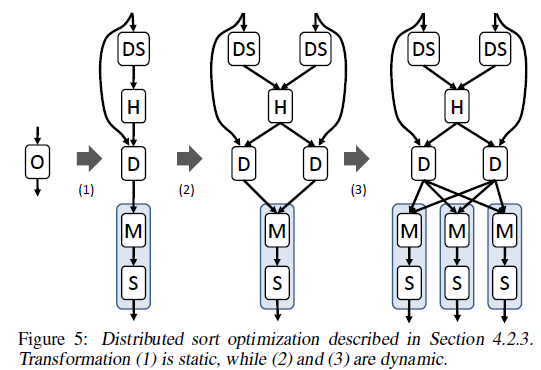
## DryadLINQ

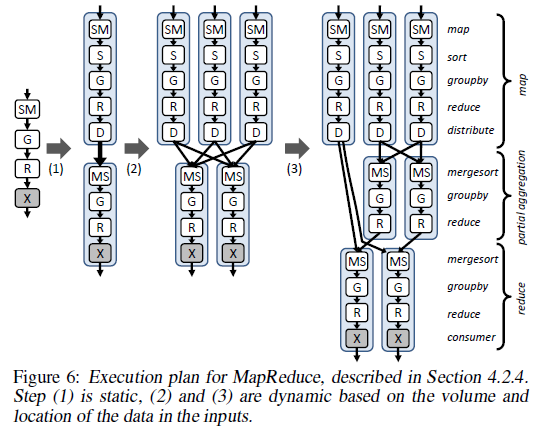
### 设计





### 优化





## Sawzall

## Dremel

<http://www.yankay.com/google-dremel-rationale/>

### 设计

根据Google公开的论文《Dremel: Interactive Analysis of WebScaleDatasets》可以看到Dremel的设计原理。还有一些测试报告。论文写于2006年，公开于2010年，Google在处理大数据方面，果真有得天独厚的优势。下面的内容，很大部分来自这篇论文。

随着Hadoop的流行，大规模的数据分析系统已经越来越普及。数据分析师需要一个能将数据“玩转”的交互式系统。如此，就可以非常方便快捷的浏览数据，建立分析模型。Dremel系统有下面几个主要的特点：

**Dremel是一个大规模系统。**在一个PB级别的数据集上面，将任务缩短到秒级，无疑需要大量的并发。磁盘的顺序读速度在100MB/S上下，那么在1S内处理1TB数据，意味着至少需要有1万个磁盘的并发读! Google一向是用廉价机器办大事的好手。但是机器越多，出问题概率越大，如此大的集群规模，需要有足够的容错考虑，保证整个分析的速度不被集群中的个别慢(坏)节点影响。

**Dremel是MR交互式查询能力不足的补充。**和MapReduce一样，Dremel也需要和数据运行在一起，将计算移动到数据上面。所以它需要GFS这样的文件系统作为存储层。在设计之初，Dremel并非是MapReduce的替代品，它只是可以执行非常快的分析，在使用的时候，常常用它来处理MapReduce的结果集或者用来建立分析原型。

**Dremel的数据模型是嵌套(nested)的。**互联网数据常常是非关系型的。Dremel还需要有一个灵活的数据模型，这个数据模型至关重要。Dremel支持一个嵌套(nested)的数据模型，类似于Json。而传统的关系模型，由于不可避免的有大量的Join操作，在处理如此大规模的数据的时候，往往是有心无力的。

**Dremel中的数据是用列式存储的。**使用列式存储，分析的时候，可以只扫描需要的那部分数据的时候，减少CPU和磁盘的访问量。同时列式存储是压缩友好的，使用压缩，可以综合CPU和磁盘，发挥最大的效能。对于关系型数据，如果使用列式存储，我们都很有经验。但是对于嵌套(nested)的结构，Dremel也可以用列存储，非常值得我们学习。

**Dremel结合了Web搜索 和并行DBMS的技术。**首先，他借鉴了Web搜索中的“查询树”的概念，将一个相对巨大复杂的查询，分割成较小较简单的查询。大事化小，小事化了，能并发的在大量节点上跑。其次，和并行DBMS类似，Dremel可以提供了一个SQL-like的接口，就像Hive和Pig那样。

## HadoopDB

<http://db.cs.yale.edu/hadoopdb/hadoopdb.html>