**Hive源码分析（八）-Hive SQL转化为MapReduce执行计划深度解析**

2018年07月18日 11:28:24

**目录**

1. MapReduce实现基本SQL操作的原理

1.1 Join的实现原理

1.2 Group By的实现原理

1.3 Distinct的实现原理

2. SQL转化为MapReduce的过程

2.1 Phase1 SQL词法，语法解析

2.2 Phase2 SQL基本组成单元QueryBlock

2.3 Phase3 逻辑操作符Operator和逻辑操作树Operator Tree

2.4 Phase4 逻辑层优化器

2.5 Phase5 OperatorTree生成MapReduce Job的过程

2.6 Phase6 物理层优化器

3. CommonJoinResolver优化器

3.1 MapJoinResolver优化器

3.2 Hive SQL编译过程的设计

4. 社区发展方向

5. 参考

文章主体源于美团网技术陈纯大作，值得拥有。原文在美团技术团队中已被删除。梳理了其中一些关键内容。并添加了一些自己的理解。

其中一部分图来自Recruit Technologies的slice：Internal Hive，一些看不懂的可以去参考一下原文（日文）。

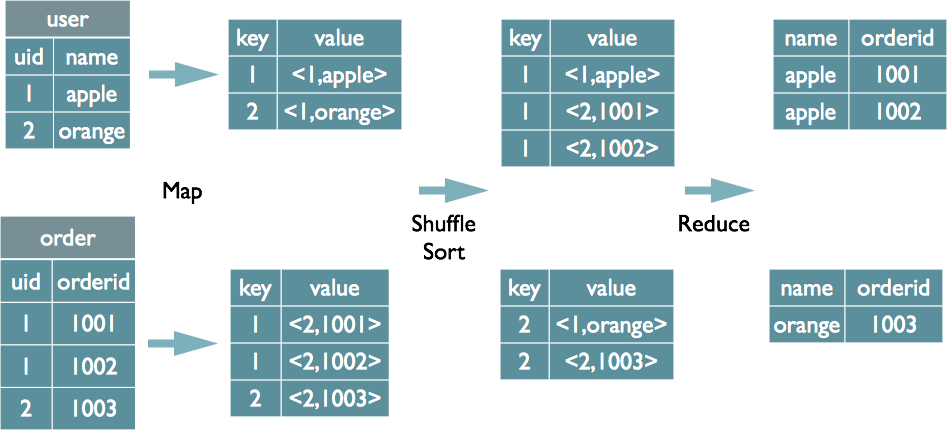
**1. MapReduce实现基本SQL操作的原理**

详细讲解SQL编译为MapReduce之前，我们先来看看MapReduce框架实现SQL基本操作的原理

**1.1 Join的实现原理**

select u.name, o.orderid from order o join user u on o.uid = u.uid;

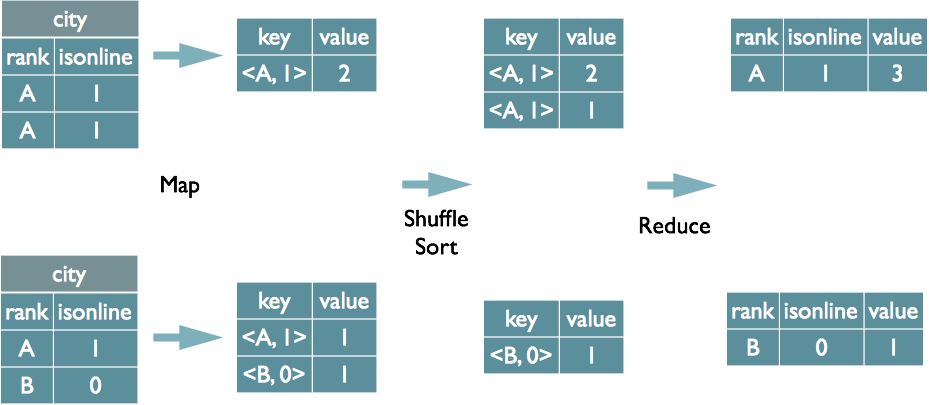
在map的输出value中为不同表的数据打上tag标记，在reduce阶段根据tag判断数据来源。MapReduce的过程如下（这里只是说明最基本的Join的实现，还有其他的实现方式）



**1.2 Group By的实现原理**

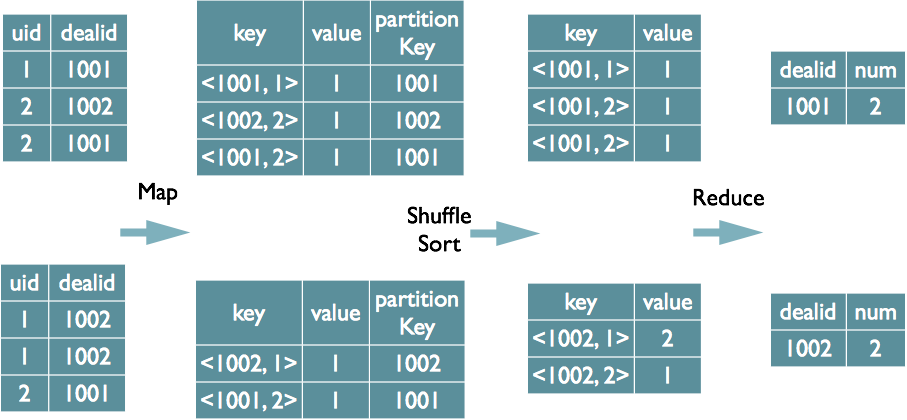
select rank, isonline, count(\*) from city group by rank, isonline;

将GroupBy的字段组合为map的输出key值，利用MapReduce的排序，在reduce阶段保存LastKey区分不同的key。MapReduce的过程如下（当然这里只是说明Reduce端的非Hash聚合过程）



**1.3 Distinct的实现原理**

select dealid, count(distinct uid) num from order group by dealid;

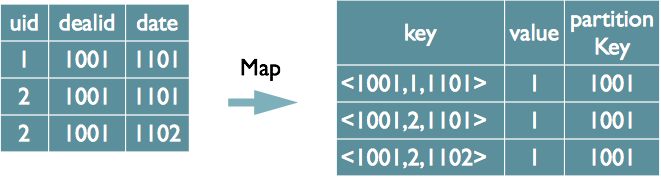


如果有多个distinct字段呢，如下面的SQL

select dealid, count(distinct uid), count(distinct date) from order group by dealid;

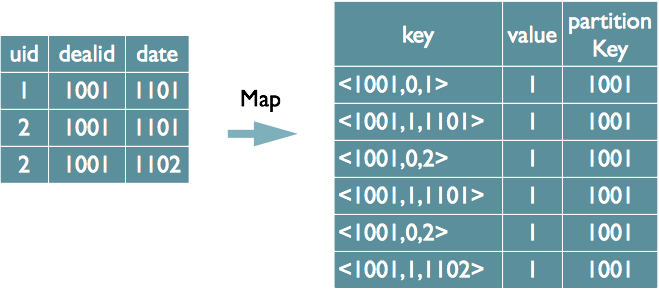
实现方式有两种：

（1）如果仍然按照上面一个distinct字段的方法，即下图这种实现方式，无法跟据uid和date分别排序，也就无法通过LastKey去重，仍然需要在reduce阶段在内存中通过Hash去重



（2）第二种实现方式，可以对所有的distinct字段编号，每行数据生成n行数据，那么相同字段就会分别排序，这时只需要在reduce阶段记录LastKey即可去重。

这种实现方式很好的利用了MapReduce的排序，节省了reduce阶段去重的内存消耗，但是缺点是增加了shuffle的数据量。需要注意的是，在生成reduce value时，除第一个distinct字段所在行需要保留value值，其余distinct数据行value字段均可为空。



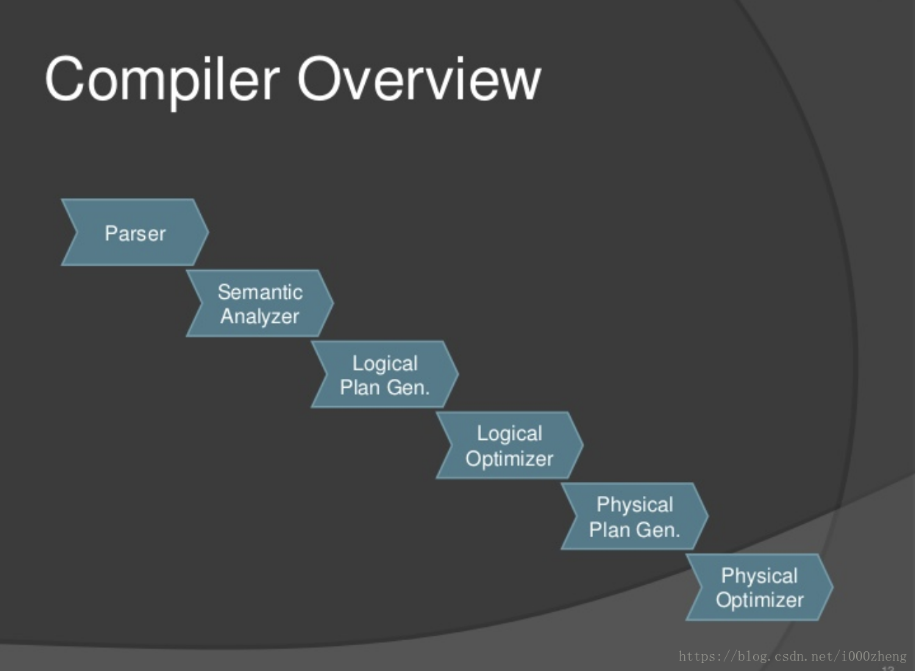
**2. SQL转化为MapReduce的过程**

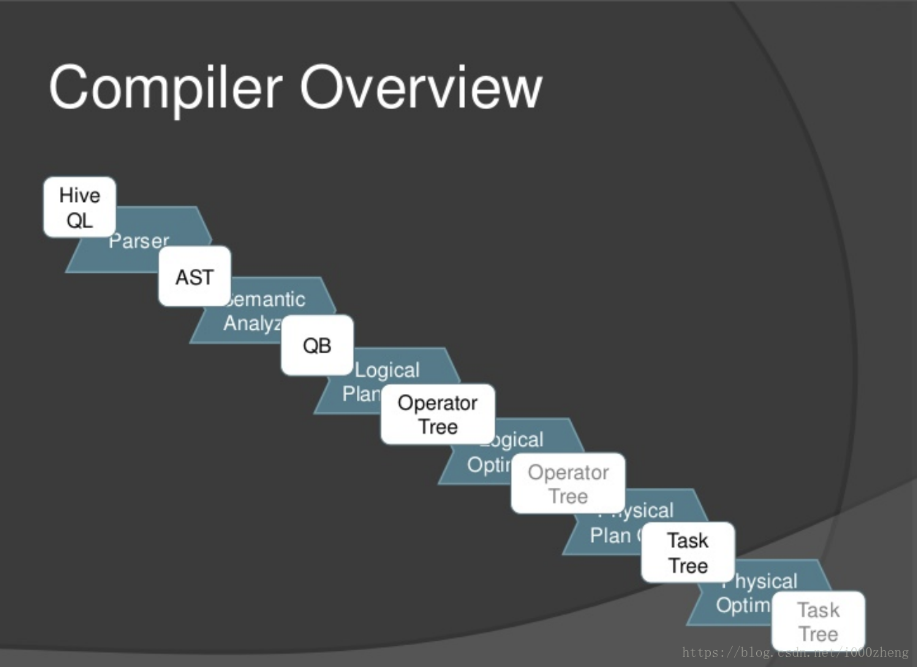
    了解了MapReduce实现SQL基本操作之后，我们来看看Hive是如何将SQL转化为MapReduce任务的，整个编译过程分为六个阶段：

1. Antlr定义SQL的语法规则，完成SQL词法，语法解析，将SQL转化为**抽象语法树AST Tree**
2. 遍历AST Tree，抽象出查询的基本组成单元**QueryBlock**
3. 遍历QueryBlock，翻译为执行操作树OperatorTree,
4. 逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量
5. 遍历OperatorTree，翻译为MapReduce任务
6. 物理层优化器进行MapReduce任务的变换，生成最终的执行计划

在slice中，将着六个阶段表示为：

1. 解析：将HIVEQL解析为AST。
2. 语义解析：将AST翻译为QB。
3. 生成逻辑计划：将QB转化为OperatorTree
4. 优化逻辑计划
5. 生成物理执行计划：将OperatorTree转化为TaskTree
6. 优化物理计划





**2.1 Phase1 SQL词法，语法解析**

**Antlr**

Hive使用Antlr实现SQL的词法和语法解析。Antlr是一种语言识别的工具，可以用来构造领域语言。  
这里不详细介绍Antlr，只需要了解使用Antlr构造特定的语言只需要编写一个语法文件，定义词法和语法替换规则即可，Antlr完成了词法分析、语法分析、语义分析、中间代码生成的过程。

**抽象语法树AST Tree（看不懂概念没关系，重点看guo）**

经过词法和语法解析后，如果需要对表达式做进一步的处理，使用 Antlr 的抽象语法树语法Abstract Syntax Tree，在语法分析的同时将输入语句转换成抽象语法树，后续在遍历语法树时完成进一步的处理。

下面的一段语法是Hive SQL中SelectStatement的语法规则，从中可以看出，SelectStatement包含select, from, where, groupby, having, orderby等子句。  
（在下面的语法规则中，箭头表示对于原语句的改写，改写后会加入一些特殊词标示特定语法，比如TOK\_QUERY标示一个查询块）

1. selectStatement
2. :
3. selectClause
4. fromClause
5. whereClause?
6. groupByClause?
7. havingClause?
8. orderByClause?
9. clusterByClause?
10. distributeByClause?
11. sortByClause?
12. limitClause? -> ^(TOK\_QUERY fromClause ^(TOK\_INSERT ^(TOK\_DESTINATION ^(TOK\_DIR TOK\_TMP\_FILE))
13. selectClause whereClause? groupByClause? havingClause? orderByClause? clusterByClause?
14. distributeByClause? sortByClause? limitClause?))
15. ;

为了详细说明SQL翻译为MapReduce的过程，这里以一条简单的SQL为例，SQL中包含一个子查询，最终将数据写入到一张表中。

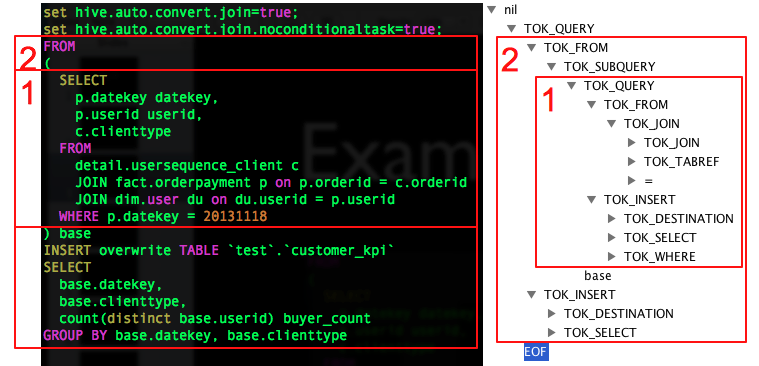
1. FROM
2. (
3. SELECT
4. p.datekey datekey,
5. p.userid userid,
6. c.clienttype
7. FROM
8. detail.usersequence\_client c
9. JOIN fact.orderpayment p ON p.orderid = c.orderid
10. JOIN default.user du ON du.userid = p.userid
11. WHERE p.datekey = 20131118
12. ) base
13. INSERT OVERWRITE TABLE `test`.`customer\_kpi`
14. SELECT
15. base.datekey,
16. base.clienttype,
17. count(distinct base.userid) buyer\_count
18. GROUP BY base.datekey, base.clienttype

**SQL生成AST Tree（重点）**

Antlr对Hive SQL解析的代码如下，HiveLexerX，HiveParser分别是Antlr对语法文件Hive.g编译后自动生成的词法解析和语法解析类，在这两个类中进行复杂的解析。

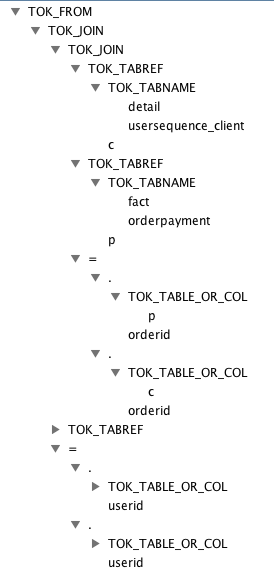
1. *// command 为hiveQL*
2. HiveLexerX lexer = new HiveLexerX(new ANTLRNoCaseStringStream(command)); *//词法解析，忽略关键词的大小写*
3. TokenRewriteStream tokens = new TokenRewriteStream(lexer);
4. if (ctx != null) {
5. ctx.setTokenRewriteStream(tokens);
6. }
7. HiveParser parser = new HiveParser(tokens); *//语法解析*
8. parser.setTreeAdaptor(adaptor);
9. HiveParser.statement\_return r = null;
10. try {
11. r = parser.statement(); *//转化为AST Tree*
12. } catch (RecognitionException e) {
13. e.printStackTrace();
14. throw new ParseException(parser.errors);
15. }

最终生成的AST Tree如下图右侧（使用Antlr Works生成，Antlr Works是Antlr提供的编写语法文件的编辑器），图中只是展开了骨架的几个节点，没有完全展开。  
子查询1/2，分别对应右侧第1/2两个部分。

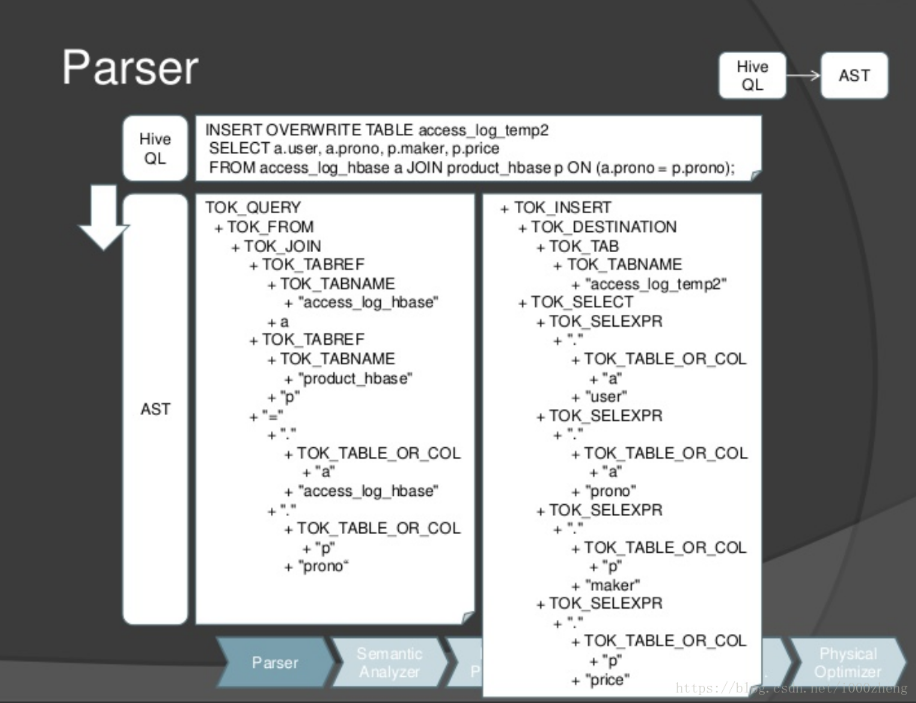


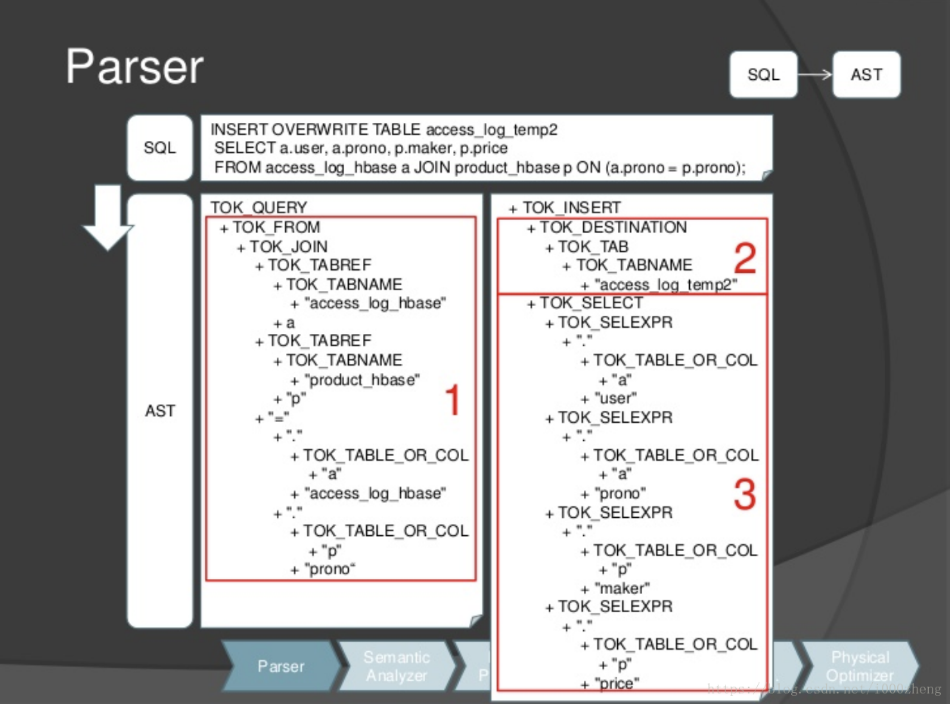
这里注意一下内层子查询也会生成一个TOK\_DESTINATION节点。请看上面SelectStatement的语法规则，这个节点是在语法改写中特意增加了的一个节点。原因是Hive中所有查询的数据均会保存在HDFS临时的文件中，无论是中间的子查询还是查询最终的结果，Insert语句最终会将数据写入表所在的HDFS目录下。

详细来看，将内存子查询的from子句展开后，得到如下AST Tree，每个表生成一个TOK\_TABREF节点，Join条件生成一个“=”节点。其他SQL部分类似，不一一详述。



看一个slice里面的例子：





**2.2 Phase2 SQL基本组成单元QueryBlock**

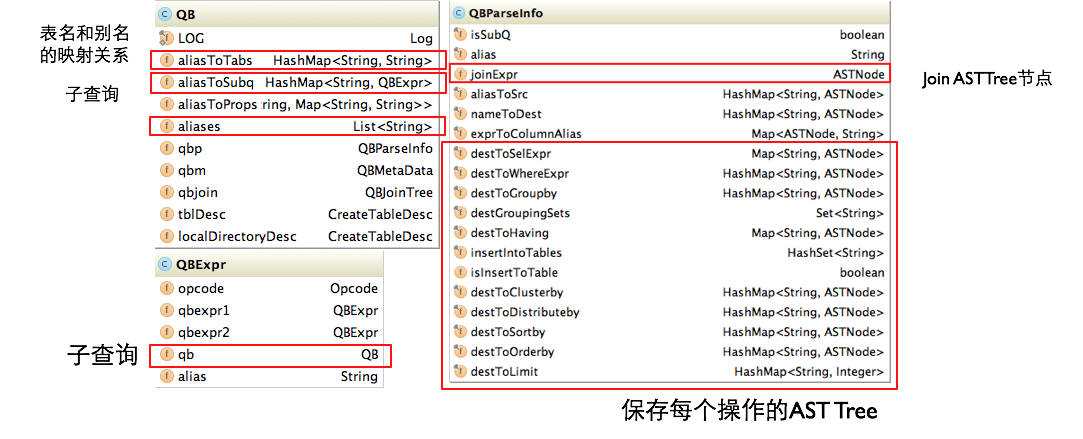
 AST Tree仍然非常复杂，不够结构化，不方便直接翻译为MapReduce程序，AST Tree转化为QueryBlock就是将SQL进一部抽象和结构化

**QueryBlock（非重点）**

QueryBlock是一条SQL最基本的组成单元，包括三个部分：输入源，计算过程，输出。简单来讲一个QueryBlock就是一个子查询。

下图为Hive中QueryBlock相关对象的类图，解释图中几个重要的属性

* QB.aliasToSubq（表示QB类的aliasToSubq属性）保存子查询的QB对象，aliasToSubq的key值是子查询的别名
* QB.qbp即QBParseInfo保存一个基本SQL单元中的各个操作部分的AST Tree结构，QBParseInfo.nameToDest这个HashMap保存查询单元的输出，key的形式是inclause-i（由于Hive支持Multi Insert语句，所以可能有多个输出），value是对应的ASTNode节点，即TOK\_DESTINATION节点。类QBParseInfo其余HashMap属性分别保存输出和各个操作的ASTNode节点的对应关系。
* QBParseInfo.JoinExpr保存TOK\_JOIN节点。QB.QBJoinTree是对Join语法树的结构化。
* QB.qbm保存每个输入表的元信息，比如表在HDFS上的路径，保存表数据的文件格式等。
* QBExpr这个对象是为了表示Union操作。

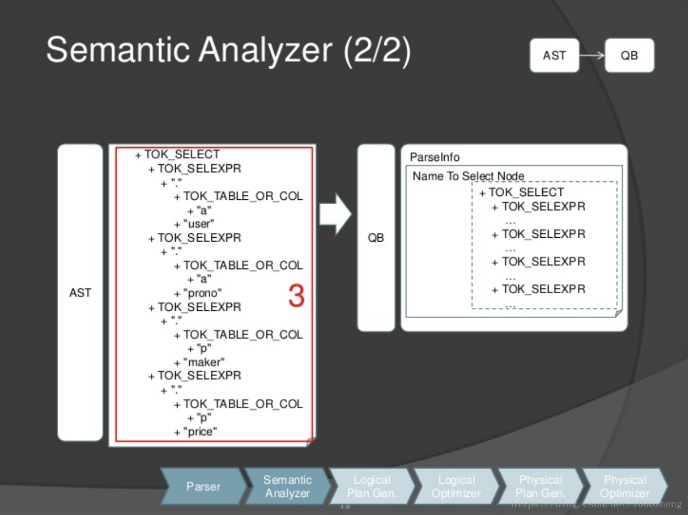
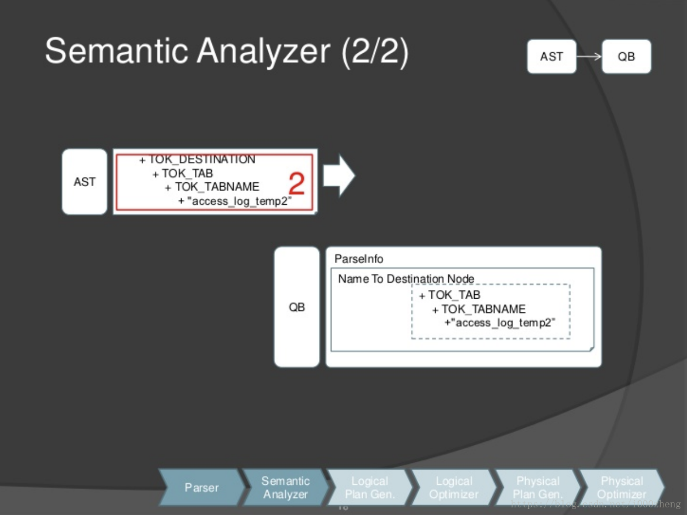
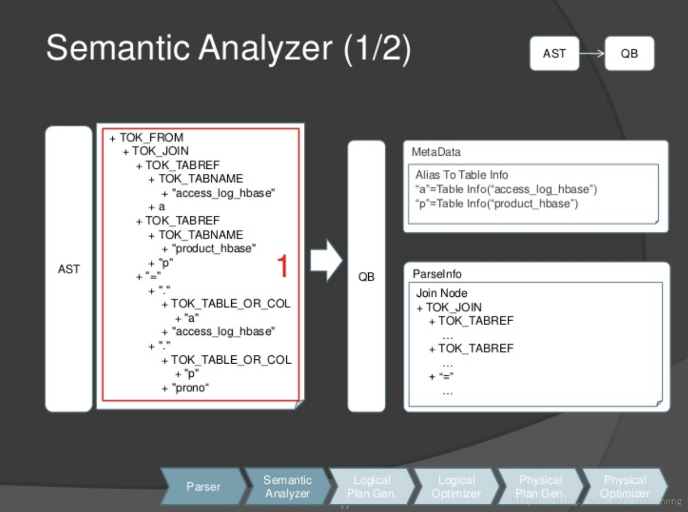


**AST Tree生成QueryBlock（重点）**

AST Tree生成QueryBlock的过程是一个递归的过程，先序遍历AST Tree，遇到不同的Token节点，保存到相应的属性中，主要包含以下几个过程：

* TOK\_QUERY => 创建QB对象，循环递归子节点
* TOK\_FROM => 将表名语法部分保存到QB对象的aliasToTabs等属性中
* TOK\_INSERT => 循环递归子节点
* TOK\_DESTINATION => 将输出目标的语法部分保存在QBParseInfo对象的nameToDest属性中
* TOK\_SELECT => 分别将查询表达式的语法部分保存在destToSelExpr、destToAggregationExprs、destToDistinctFuncExprs三个属性中
* TOK\_WHERE => 将Where部分的语法保存在QBParseInfo对象的destToWhereExpr属性中

看一下slice中的例子：



**2.3 Phase3 逻辑操作符Operator和逻辑操作树Operator Tree**

**Operator**

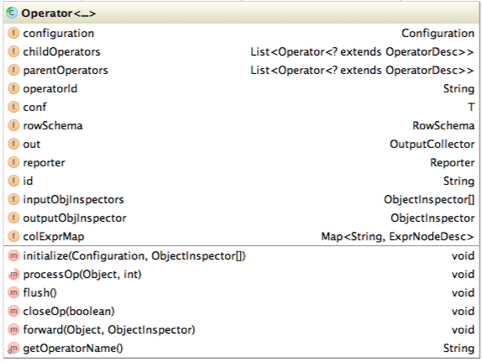
Hive最终生成的MapReduce任务，Map阶段和Reduce阶段均由OperatorTree组成。逻辑操作符，就是在Map阶段或者Reduce阶段完成单一特定的操作。基本的操作符包括TableScanOperator，SelectOperator，FilterOperator，JoinOperator，GroupByOperator，ReduceSinkOperator。

从名字就能猜出各个操作符完成的功能，TableScanOperator从MapReduce框架的Map接口原始输入表的数据，控制扫描表的数据行数，标记是从原表中取数据。JoinOperator完成Join操作。FilterOperator完成过滤操作。ReduceSinkOperator将Map端的字段组合序列化为Reduce Key/value, Partition Key，只可能出现在Map阶段，同时也标志着Hive生成的MapReduce程序中Map阶段的结束。

Operator在Map Reduce阶段之间的数据传递都是一个流式的过程。每一个Operator对一行数据完成操作后之后将数据传递给childOperator计算。

Operator类的主要属性和方法如下

* RowSchema表示Operator的输出字段
* InputObjInspector outputObjInspector解析输入和输出字段
* processOp接收父Operator传递的数据，forward将处理好的数据传递给子Operator处理
* Hive每一行数据经过一个Operator处理之后，会对字段重新编号，colExprMap记录每个表达式经过当前Operator处理前后的名称对应关系，在下一个阶段逻辑优化阶段用来回溯字段名
* 由 于Hive的MapReduce程序是一个动态的程序，即不确定一个MapReduce Job会进行什么运算，可能是Join，也可能是GroupBy，所以Operator将所有运行时需要的参数保存在OperatorDesc 中，OperatorDesc在提交任务前序列化到HDFS上，在MapReduce任务执行前从HDFS读取并反序列化。Map阶段 OperatorTree在HDFS上的位置在Job.getConf(“hive.exec.plan”)  
  + “/map.xml”



**QueryBlock生成Operator Tree**

QueryBlock生成Operator Tree就是遍历上一个过程中生成的QB和QBParseInfo对象的保存语法的属性，包含如下几个步骤：

* QB.aliasToSubq => 有子查询，递归调用
* QB.aliasToTabs => TableScanOperator
* QBParseInfo.joinExpr => QBJoinTree => ReduceSinkOperator + JoinOperator
* QBParseInfo.destToWhereExpr => FilterOperator
* QBParseInfo.destToGroupby => ReduceSinkOperator + GroupByOperator
* QBParseInfo.destToOrderby => ReduceSinkOperator + ExtractOperator

**2.4 Phase4 逻辑层优化器**

大部分逻辑层优化器通过变换OperatorTree，合并操作符，达到减少MapReduce Job（② ），减少shuffle数据量的目的（①）。

| **名称** | **作用** |
| --- | --- |
| ② SimpleFetchOptimizer | 优化没有GroupBy表达式的聚合查询 |
| ② MapJoinProcessor | MapJoin，需要SQL中提供hint，0.11版本已不用 |
| ② BucketMapJoinOptimizer | BucketMapJoin |
| ② GroupByOptimizer | Map端聚合 |
| ① ReduceSinkDeDuplication | 合并线性的OperatorTree中partition/sort key相同的reduce |
| ① PredicatePushDown | 谓词前置 |
| ① CorrelationOptimizer | 利用查询中的相关性，合并有相关性的Job，HIVE-2206 |
| ColumnPruner | 字段剪枝 |

表格中①的优化器均是一个Job干尽可能多的事情/合并。②的都是减少shuffle数据量，甚至不做Reduce。CorrelationOptimizer优化器非常复杂，都能利用查询中的相关性，合并有相关性的Job，参考 Hive Correlation Optimizer。

注：具体的每个优化器优化了什么，没看太明白。其中：PredicatePushDown看slice是把一个查询的谓词（LKIE、BETWEEN、IS NULL、IS NOT NULL、IN、EXISTS）提到他在OperatorTree中的前置查询里。以提前减少数据量。

**2.5 Phase5 OperatorTree生成MapReduce Job的过程**

OperatorTree转化为MapReduce Job的过程分为下面几个阶段

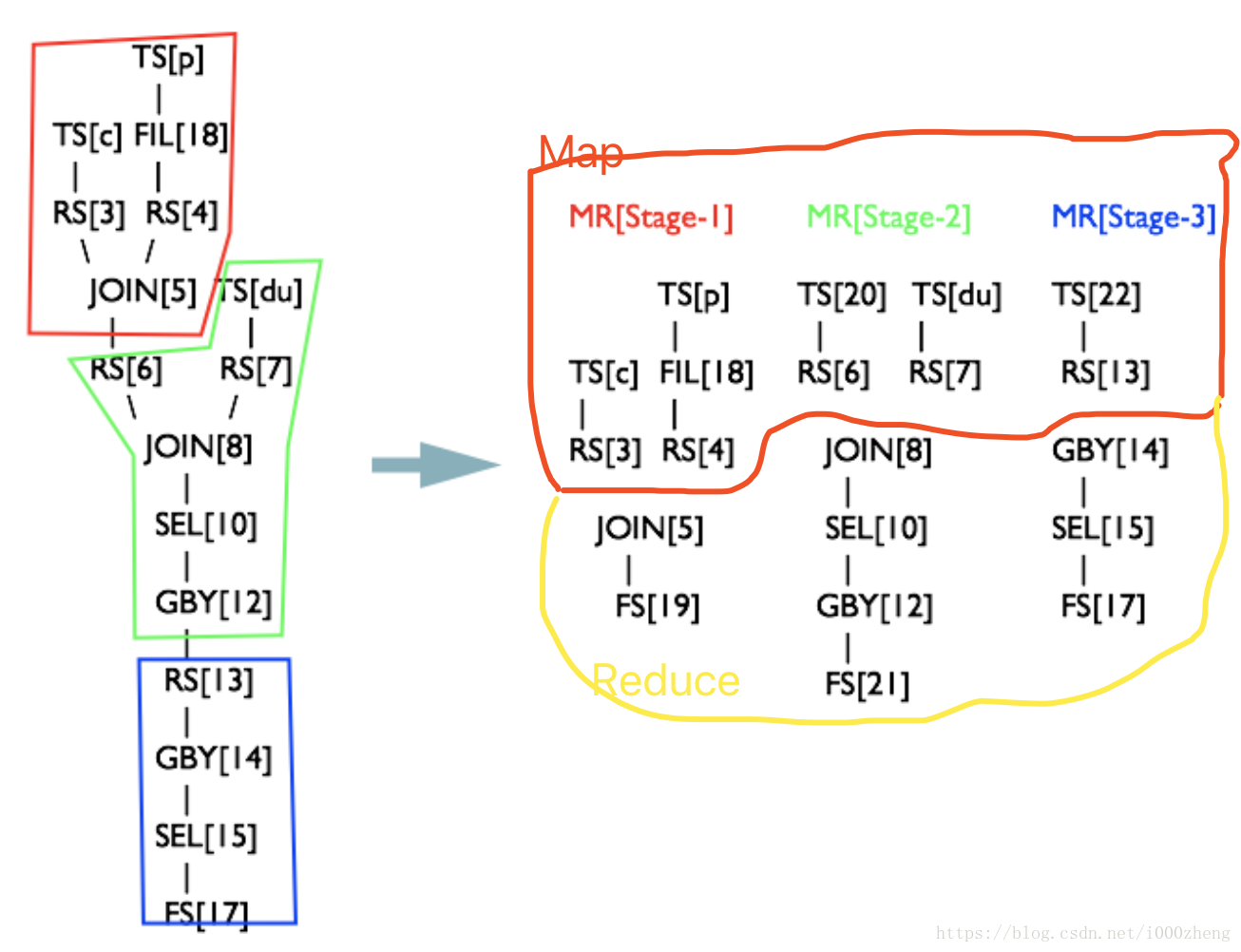
1. 对输出表生成MoveTask
2. 从OperatorTree的其中一个根节点向下深度优先遍历
3. ReduceSinkOperator标示Map/Reduce的界限，多个Job间的界限
4. 遍历其他根节点，遇过碰到JoinOperator合并MapReduceTask
5. 生成StatTask更新元数据
6. 剪断Map与Reduce间的Operator的关系

这一段太复杂。再次学习之后 再作补充。

大致思想是：从operatorTree的根节点开始遍历。把各个根节点入队，然后遇到符合一定规则的组合，就划分Map/Reduce的界限，或者划分Job的界限。有好几个规则（Rules）。

例如：遇到TS（QB.aliasToTabs），就生成一个MapWork，再遇到RS（ReduceSinkOperator），就从TS到RS划分为一个MapWork，从该RS到下一个RS前，划分为一个ReduceWork。下一个RS又生成一个MapWork。

下图是一个从OperatorTree到MapReduceJob的示例。FS[19]和FS[21]是Stage间生成的中间文件。

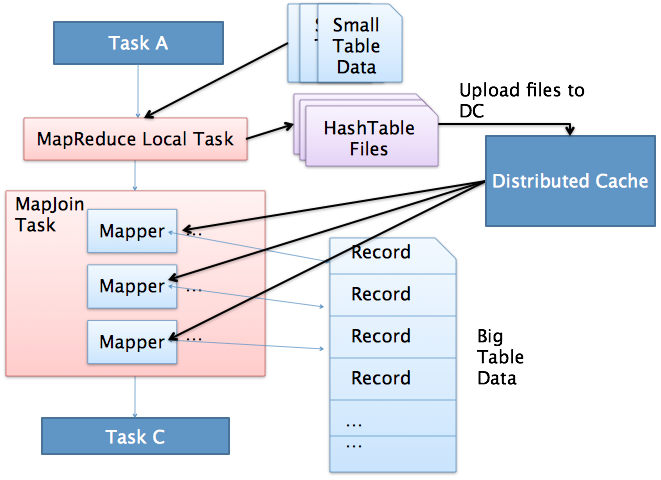


**2.6 Phase6 物理层优化器**

这里不详细介绍每个优化器的原理，单独介绍一下MapJoin的优化器

| **名称** | **作用** |
| --- | --- |
| Vectorizer | HIVE-4160，将在0.13中发布 |
| SortMergeJoinResolver | 与bucket配合，类似于归并排序 |
| SamplingOptimizer | 并行order by优化器，在0.12中发布 |
| CommonJoinResolver + MapJoinResolver | MapJoin优化器 |

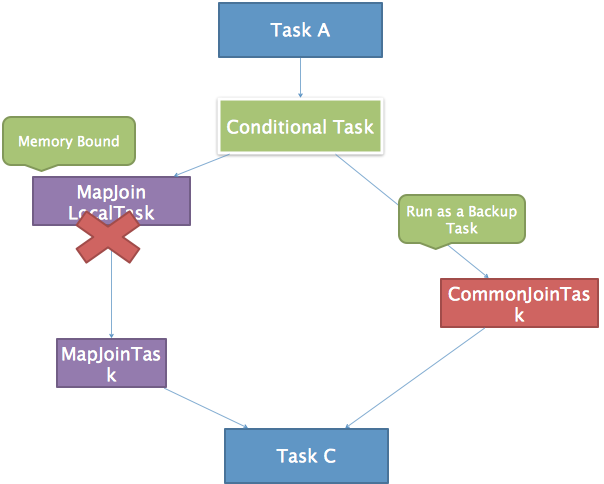
**MapJoin原理**



MapJoin简单说就是在Map阶段将小表读入内存，顺序扫描大表完成Join。

上图是Hive MapJoin的原理图，出自Facebook工程师Liyin Tang的一篇介绍Join优化的slice，从图中可以看出MapJoin分为两个阶段：

1. 通过MapReduce Local Task，将小表读入内存，生成HashTableFiles上传至Distributed Cache中，这里会对HashTableFiles进行压缩。
2. MapReduce Job在Map阶段，每个Mapper从Distributed Cache读取HashTableFiles到内存中，顺序扫描大表，在Map阶段直接进行Join，将数据传递给下一个MapReduce任务。

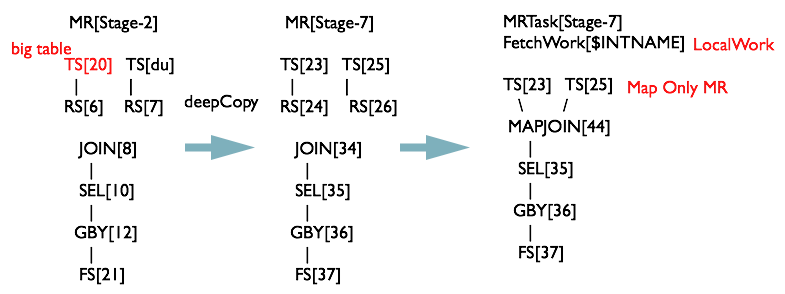


**3. CommonJoinResolver优化器**

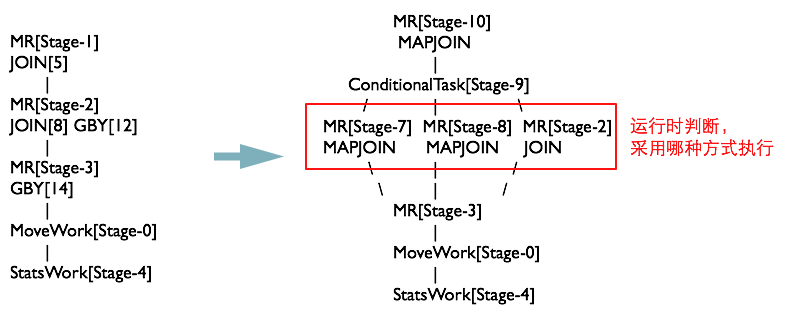
CommonJoinResolver优化器就是将CommonJoin转化为MapJoin，转化过程如下

1. 深度优先遍历Task Tree
2. 找到JoinOperator，判断左右表数据量大小
3. 对与小表 + 大表 => MapJoinTask，对于小/大表 + 中间表 => ConditionalTask

遍历上一个阶段生成的MapReduce任务，发现MapReduceTask[Stage-2] JOIN[8]中有一张表为临时表，先对Stage-2进行深度拷贝（由于需要保留原始执行计划为Backup Plan，所以这里将执行计划拷贝了一份），生成一个MapJoinOperator替代JoinOperator，然后生成一个MapReduceLocalWork读取小表生成HashTableFiles上传至DistributedCache中。

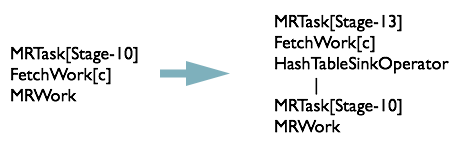


MapReduceTask经过变换后的执行计划如下图所示

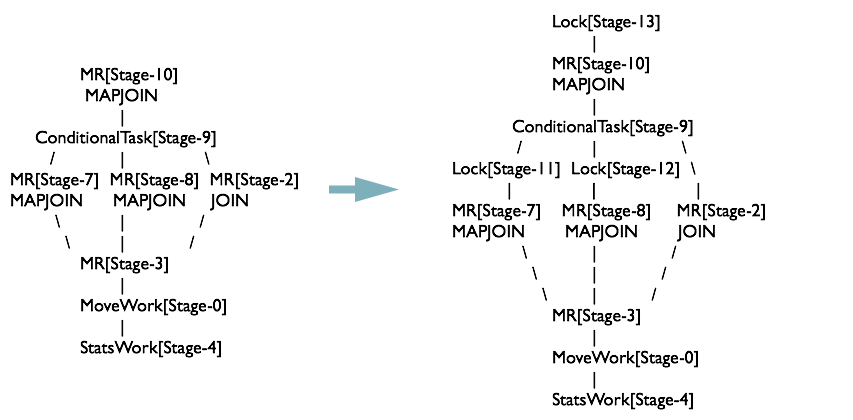


#### MapJoinResolver优化器

MapJoinResolver优化器遍历Task Tree，将所有有local work的MapReduceTask拆成两个Task



最终MapJoinResolver处理完之后，执行计划如下图所示



### Hive SQL编译过程的设计

从上述整个SQL编译的过程，可以看出编译过程的设计有几个优点值得学习和借鉴

* 使用Antlr开源软件定义语法规则，大大简化了词法和语法的编译解析过程，仅仅需要维护一份语法文件即可。
* 整体思路很清晰，分阶段的设计使整个编译过程代码容易维护，使得后续各种优化器方便的以可插拔的方式开关，譬如Hive 0.13最新的特性Vectorization和对Tez引擎的支持都是可插拔的。
* 每个Operator只完成单一的功能，简化了整个MapReduce程序。

### 社区发展方向

Hive依然在迅速的发展中，为了提升Hive的性能，hortonworks公司主导的Stinger计划提出了一系列对Hive的改进，比较重要的改进有：

* Vectorization - 使Hive从单行单行处理数据改为批量处理方式，大大提升了指令流水线和缓存的利用率
* Hive on Tez - 将Hive底层的MapReduce计算框架替换为Tez计算框架。Tez不仅可以支持多Reduce阶段的任务MRR，还可以一次性提交执行计划，因而能更好的分配资源。
* Cost Based Optimizer - 使Hive能够自动选择最优的Join顺序，提高查询速度
* Implement insert, update, and delete in Hive with full ACID support - 支持表按主键的增量更新

我们也将跟进社区的发展，结合自身的业务需要，提升Hive型ETL流程的性能

### 参考

Antlr: http://www.antlr.org/  
Wiki Antlr介绍: http://en.wikipedia.org/wiki/ANTLR  
Hive Wiki: https://cwiki.apache.org/confluence/display/Hive/Home  
HiveSQL编译过程: http://www.slideshare.net/recruitcojp/internal-hive  
Join Optimization in Hive: Join Strategies in Hive from the 2011 Hadoop Summit (Liyin Tang, Namit Jain)

Hive Design Docs: https://cwiki.apache.org/confluence/display/Hive/DesignDocs

转载自：http://tech.meituan.com/hive-sql-to-mapreduce.html

详细可参考: https://blog.csdn.net/u010738184/article/details/70893161