> 原文地址 https://tech.meituan.com/2014/02/12/hive-sql-to-mapreduce.html

Hive 是基于 Hadoop 的一个数据仓库系统，在各大公司都有广泛的应用。美团数据仓库也是基于 Hive 搭建，每天执行近万次的 Hive ETL 计算流程，负责每天数百 GB 的数据存储和分析。Hive 的稳定性和性能对我们的数据分析非常关键。

在几次升级 Hive 的过程中，我们遇到了一些大大小小的问题。通过向社区的咨询和自己的努力，在解决这些问题的同时我们对 Hive 将 SQL 编译为 MapReduce 的过程有了比较深入的理解。对这一过程的理解不仅帮助我们解决了一些 Hive 的 bug，也有利于我们优化 Hive SQL，提升我们对 Hive 的掌控力，同时有能力去定制一些需要的功能。

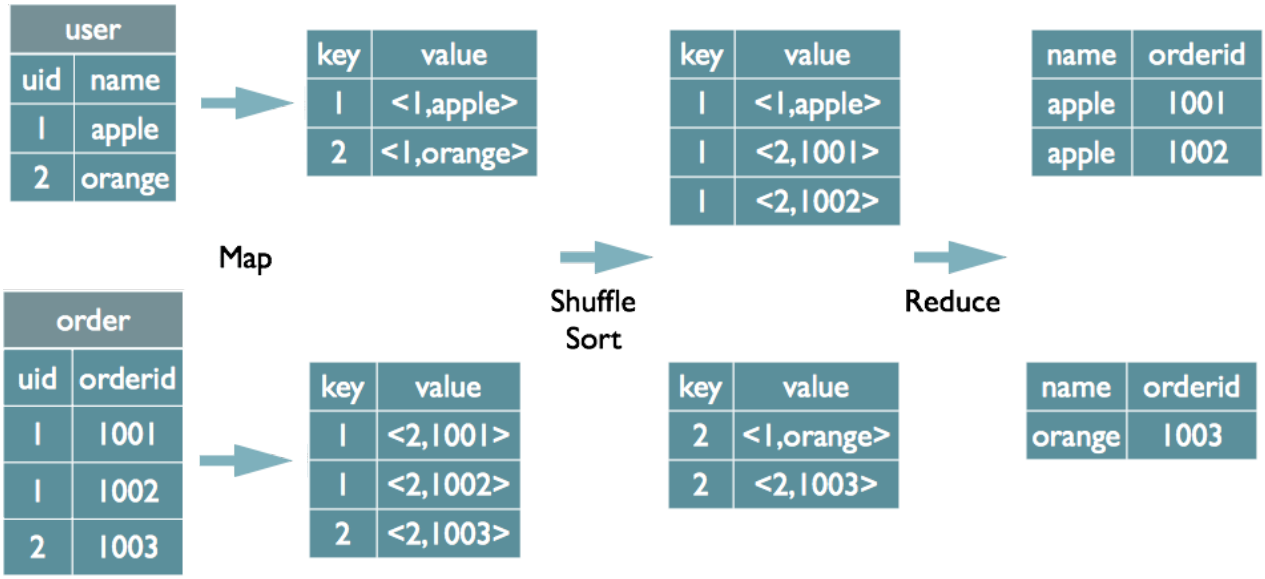
MapReduce 实现基本 SQL 操作的原理

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

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）

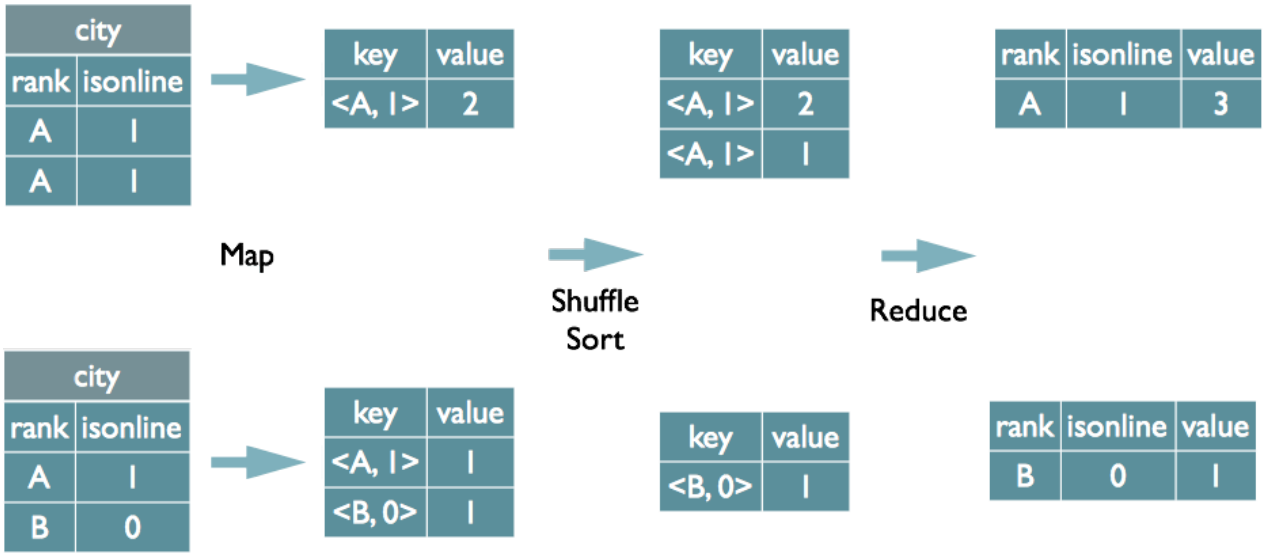


MapReduce CommonJoin 的实现

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）

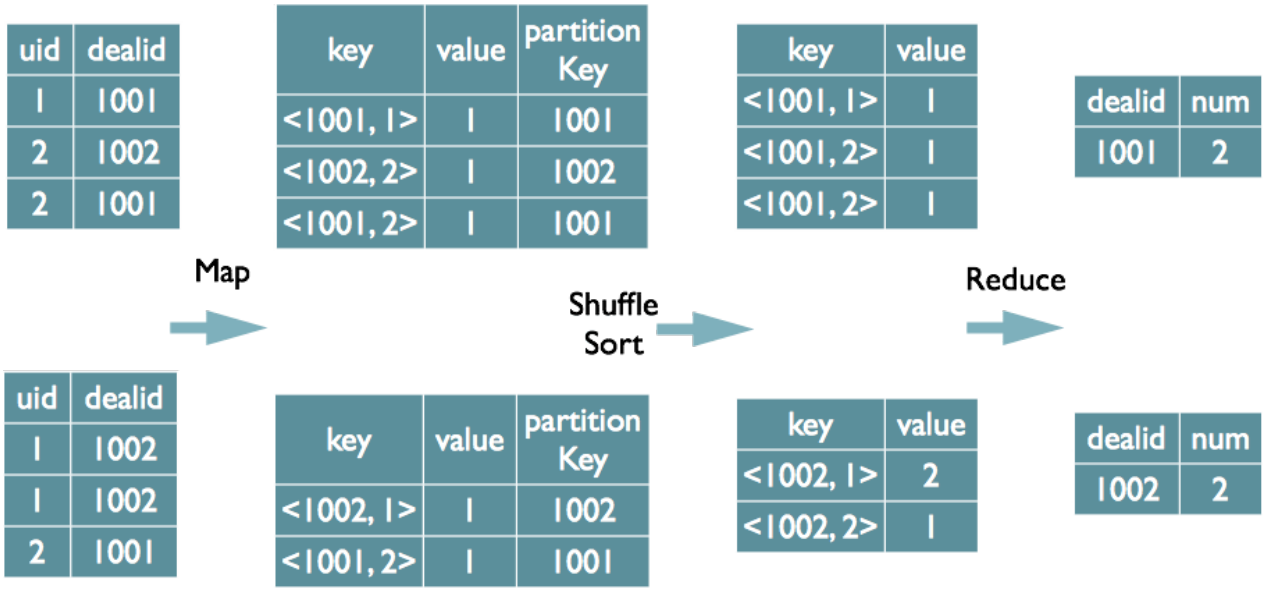


MapReduce Group By 的实现

Distinct 的实现原理

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

当只有一个 distinct 字段时，如果不考虑 Map 阶段的 Hash GroupBy，只需要将 GroupBy 字段和 Distinct 字段组合为 map 输出 key，利用 mapreduce 的排序，同时将 GroupBy 字段作为 reduce 的 key，在 reduce 阶段保存 LastKey 即可完成去重



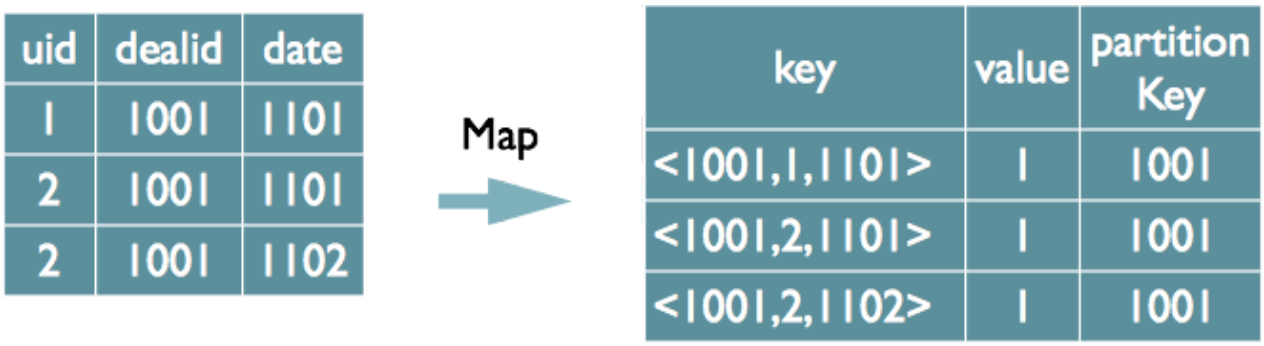
MapReduce Distinct 的实现

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

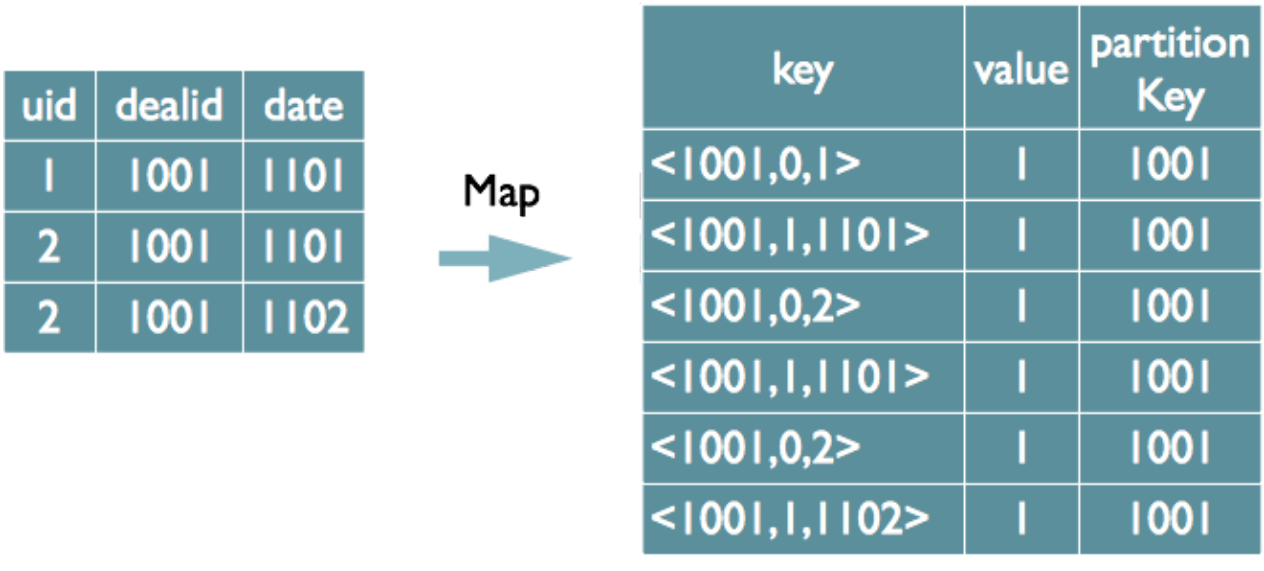


MapReduce Multi Distinct 的实现

（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。



MapReduce Multi Distinct 的实现

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

下面分别对这六个阶段进行介绍

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。

Hive 中语法规则的定义文件在 0.10 版本以前是 Hive.g 一个文件，随着语法规则越来越复杂，由语法规则生成的 Java 解析类可能超过 Java 类文件的最大上限，0.11 版本将 Hive.g 拆成了 5 个文件，词法规则 HiveLexer.g 和语法规则的 4 个文件 SelectClauseParser.g，FromClauseParser.g，IdentifiersParser.g，HiveParser.g。

抽象语法树 AST Tree

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）

```
selectStatement
    :
    selectClause
    fromClause
    whereClause?
    groupByClause?
    havingClause?
    orderByClause?
    clusterByClause?
    distributeByClause?
    sortByClause?
    limitClause? -> ^(TOK_QUERY fromClause ^(TOK_INSERT ^(TOK_DESTINATION
^(TOK_DIR TOK_TMP_FILE))
                        selectClause whereClause? groupByClause? havingClause?
orderByClause? clusterByClause?
                        distributeByClause? sortByClause? limitClause?))
    ;
```

样例 SQL

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

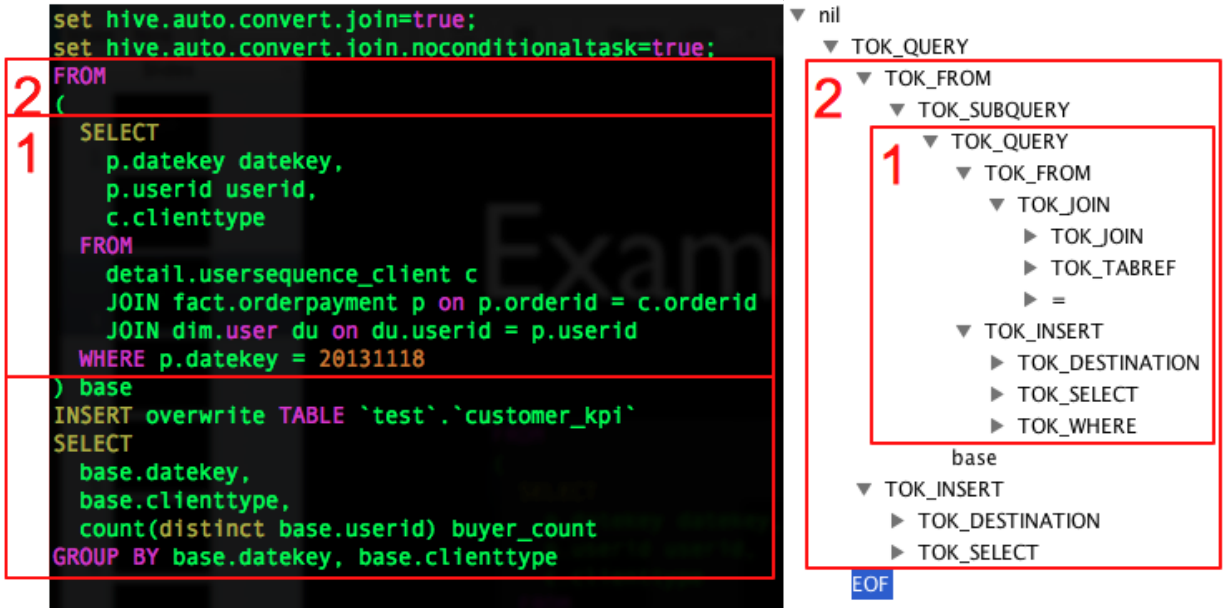
```
FROM
(
  SELECT
    p.datekey datekey,
    p.userid userid,
    c.clienttype
  FROM
    detail.usersequence_client c
    JOIN fact.orderpayment p ON p.orderid = c.orderid
    JOIN default.user du ON du.userid = p.userid
  WHERE p.datekey = 20131118
) base
INSERT OVERWRITE TABLE `test`.`customer_kpi`
SELECT
  base.datekey,
  base.clienttype,
  count(distinct base.userid) buyer_count
GROUP BY base.datekey, base.clienttype
```

SQL 生成 AST Tree

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。

```
HiveLexerX lexer = new HiveLexerX(new ANTLRNoCaseStringStream(command));
TokenRewriteStream tokens = new TokenRewriteStream(lexer);
if (ctx != null) {
  ctx.setTokenRewriteStream(tokens);
}
HiveParser parser = new HiveParser(tokens);
parser.setTreeAdaptor(adaptor);
HiveParser.statement_return r = null;
try {
  r = parser.statement();
} catch (RecognitionException e) {
  e.printStackTrace();
  throw new ParseException(parser.errors);
}
```

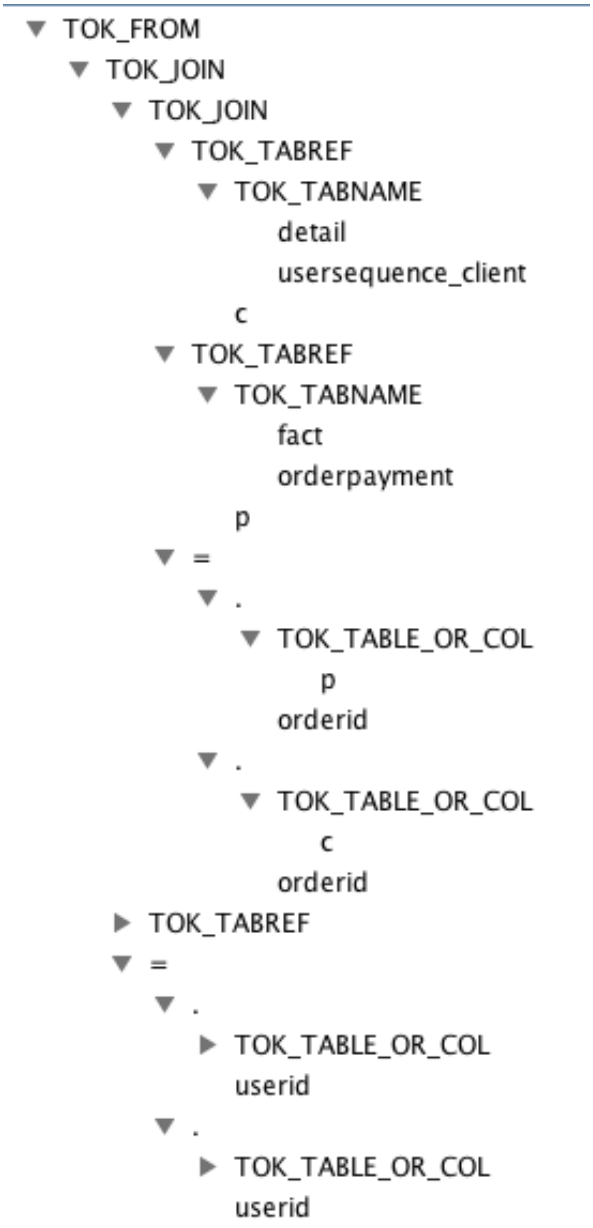
最终生成的 AST Tree 如下图右侧（使用 Antlr Works 生成，Antlr Works 是 Antlr 提供的编写语法文件的编辑器），图中只是展开了骨架的几个节点，没有完全展开。 子查询 1/2，分别对应右侧第 1/2 两个部分。



SQL 生成 AST Tree

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。



AST Tree

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

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 key 值是子查询的别名
- QB#qbp 即 QBParseInfo 保存一个基本 SQL 单元中的给个操作部分的 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。

QueryBlock

AST Tree 生成 QueryBlock

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

最终样例 SQL 生成两个 QB 对象，QB 对象的关系如下，QB1 是外层查询，QB2 是子查询

```
QB1

  \

   QB2
```

Phase3 逻辑操作符 Operator

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

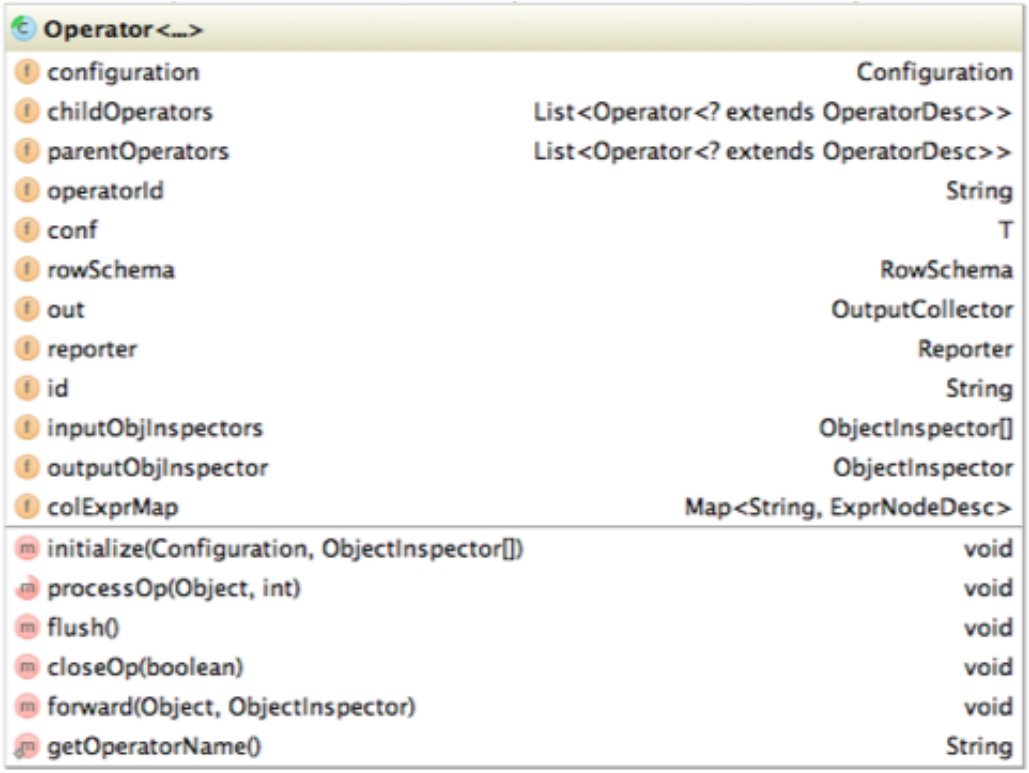
从名字就能猜出各个操作符完成的功能，TableScanOperator 从 MapReduce 框架的 Map 接口原始输入表的数据，控制扫描表的数据行数，标记是从原表中取数据。JoinOperator 完成 Join 操作。FilterOperator 完成过滤操作

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

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

由于 Join/GroupBy/OrderBy 均需要在 Reduce 阶段完成，所以在生成相应操作的 Operator 之前都会先生成一个 ReduceSinkOperator，将字段组合并序列化为 Reduce Key/value, Partition Key

接下来详细分析样例 SQL 生成 OperatorTree 的过程

先序遍历上一个阶段生成的 QB 对象

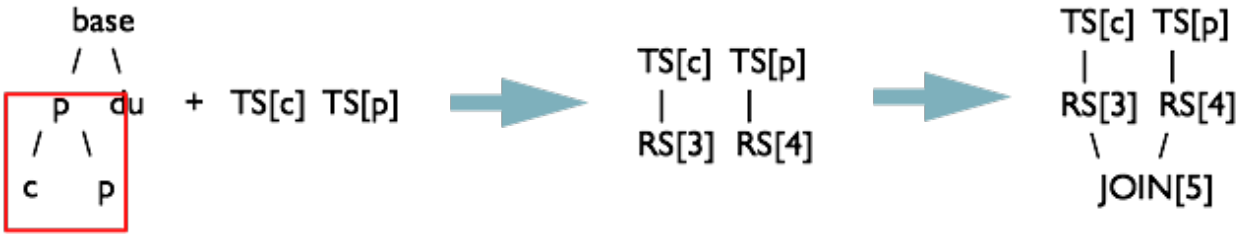
1. 首先根据子 QueryBlock `QB2#aliasToTabs {du=dim.user, c=detail.usersequence_client, p=fact.orderpayment}` 生成 TableScanOperator

```
TableScanOperator(“dim.user”) TS[0]
TableScanOperator(“detail.usersequence_client”) TS[1]
TableScanOperator(“fact.orderpayment”) TS[2]
```

2. 先序遍历 `QBParseInfo#joinExpr` 生成 `QBJoinTree`，类 `QBJoinTree` 也是一个树状结构，`QBJoinTree` 保存左右表的 ASTNode 和这个查询的别名，最终生成的查询树如下

```
   base
   /  \
  p    du
 /      \
c        p
```

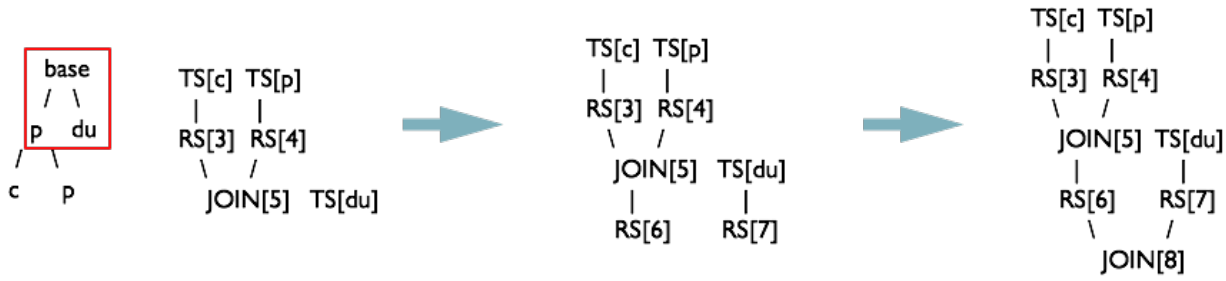
3. 前序遍历 `QBJoinTree`，先生成 `detail.usersequence_client` 和 `fact.orderpayment` 的 Join 操作树



Join to Operator

图中 TS=TableScanOperator RS=ReduceSinkOperator JOIN=JoinOperator

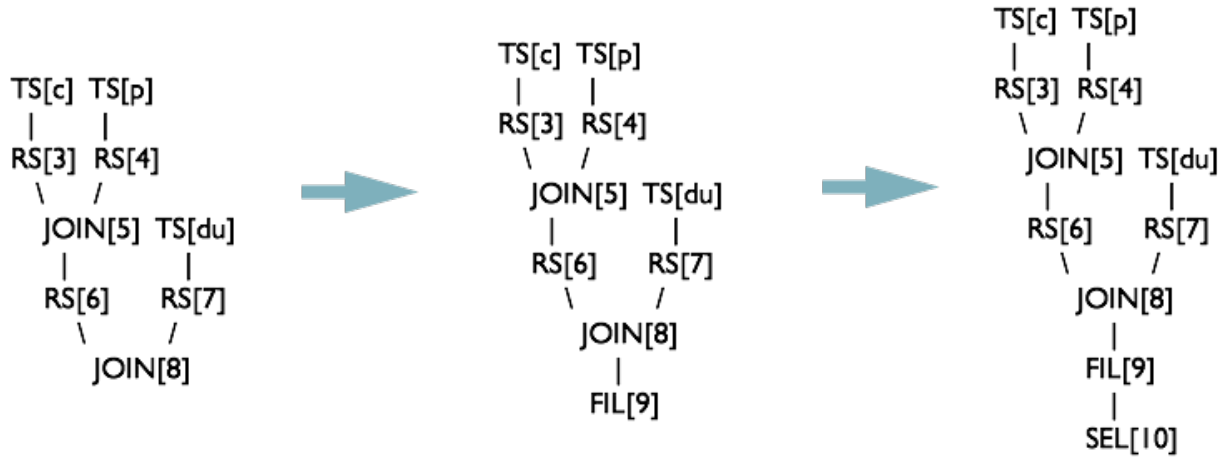
1. 生成中间表与 dim.user 的 Join 操作树



Join to Operator

1. 根据 QB2 `QBParseInfo#destToWhereExpr` 生成`FilterOperator`。此时 QB2 遍历完成。

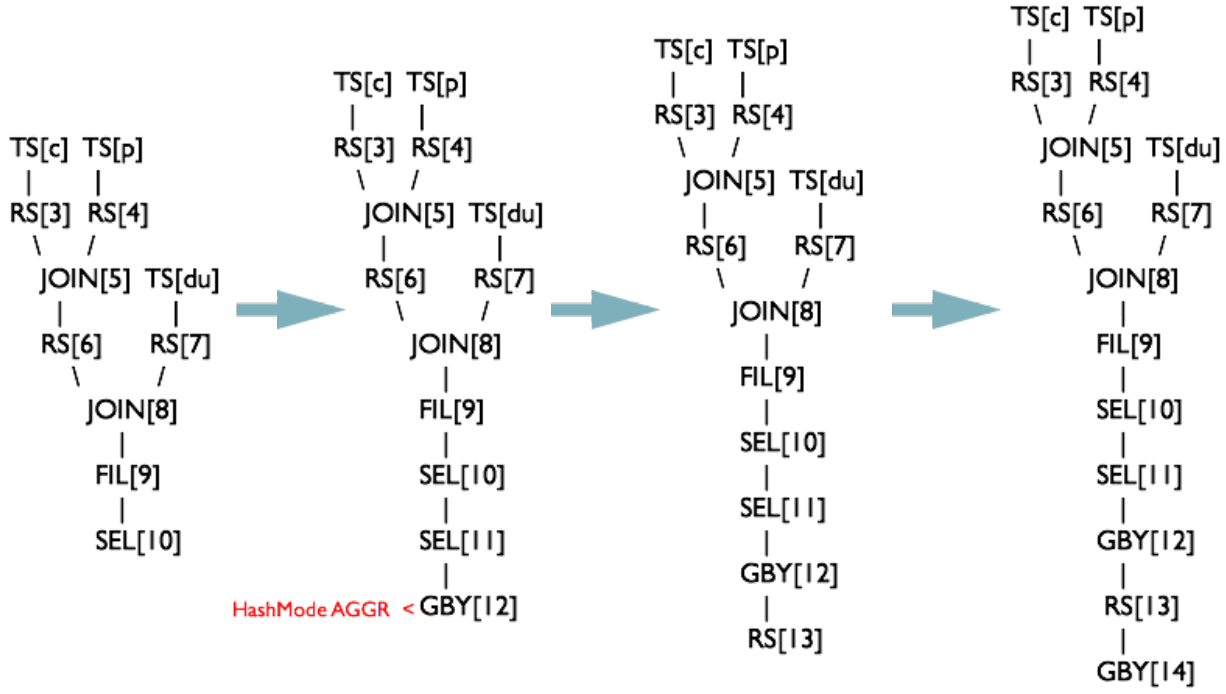
下图中 SelectOperator 在某些场景下会根据一些条件判断是否需要解析字段。



Where to Operator

图中 FIL= FilterOperator SEL= SelectOperator

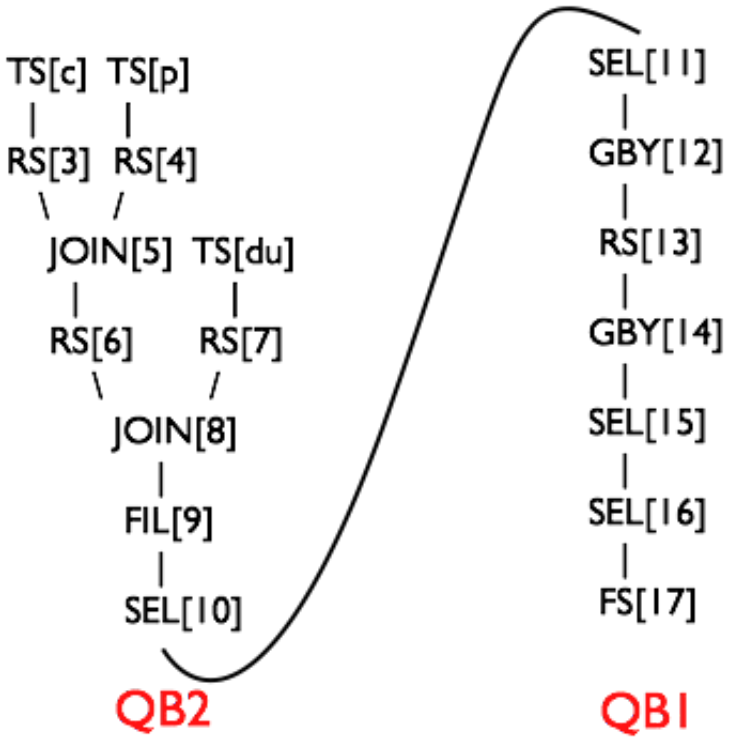
1. 根据 QB1 的 QBParseInfo#destToGroupby 生成 ReduceSinkOperator + GroupByOperator



GroupBy to Operator

图中 GBY= GroupByOperator GBY[12] 是 HASH 聚合，即在内存中通过 Hash 进行聚合运算

1. 最终都解析完后，会生成一个 FileSinkOperator，将数据写入 HDFS



FileSinkOperator

图中 FS=FileSinkOperator

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，减少 shuffle 数据量的目的。

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

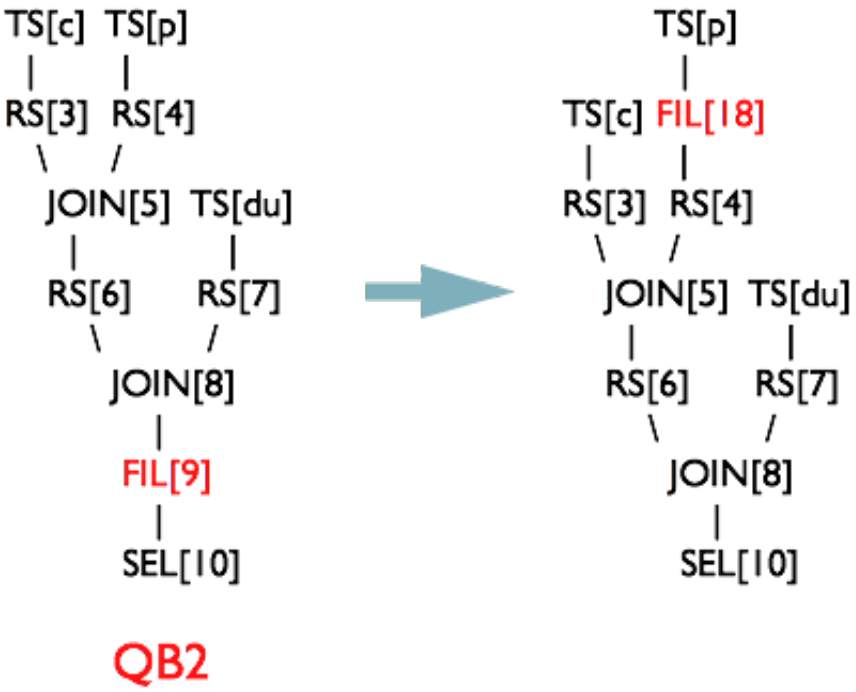
表格中①的优化器均是一个 Job 干尽可能多的事情 / 合并。②的都是减少 shuffle 数据量，甚至不做 Reduce。

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

对于样例 SQL，有两个优化器对其进行优化。下面分别介绍这两个优化器的作用，并补充一个优化器 ReduceSinkDeDuplication 的作用

PredicatePushDown 优化器

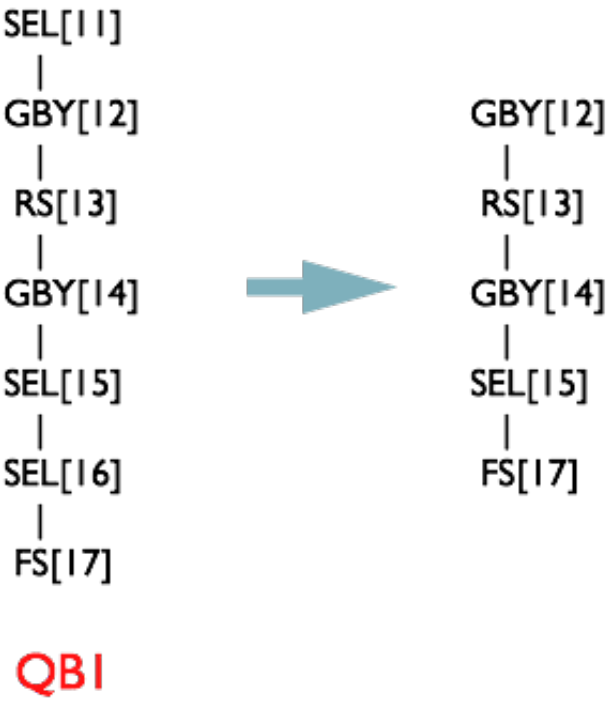
断言判断提前优化器将 OperatorTree 中的 FilterOperator 提前到 TableScanOperator 之后



PredicatePushDown

NonBlockingOpDeDupProc 优化器

`NonBlockingOpDeDupProc` 优化器合并 SEL-SEL 或者 FIL-FIL 为一个 Operator



NonBlockingOpDeDupProc

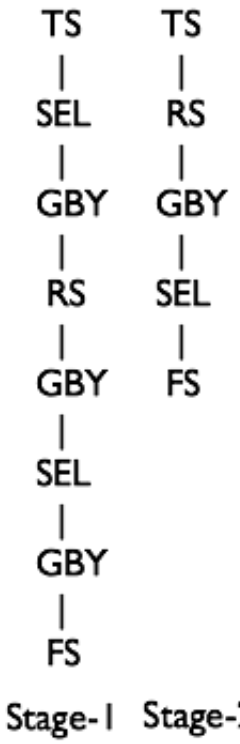
ReduceSinkDeDuplication 优化器

ReduceSinkDeDuplication 可以合并线性相连的两个 RS。实际上 CorrelationOptimizer 是 ReduceSinkDeDuplication 的超集，能合并线性和非线性的操作 RS，但是 Hive 先实现的 ReduceSinkDeDuplication

譬如下面这条 SQL 语句

```
from (select key, value from src group by key, value) s select s.key group by s.key;
```

经过前面几个阶段之后，会生成如下的 OperatorTree，两个 Tree 是相连的，这里没有画到一起



ReduceSinkDeDuplication

这时候遍历 OperatorTree 后能发现前前后两个 RS 输出的 Key 值和 PartitionKey 如下

	Key	**PartitionKey**
childRS	key	key
parentRS	key,value	key,value

ReduceSinkDeDuplication 优化器检测到：1. pRS Key 完全包含 cRS Key，且排序顺序一致；2. pRS PartitionKey 完全包含 cRS PartitionKey。符合优化条件，会对执行计划进行优化。

ReduceSinkDeDuplication 将 childRS 和 parentheRS 与 childRS 之间的 Operator 删掉，保留的 RS 的 Key 为 key,value 字段，PartitionKey 为 key 字段。合并后的 OperatorTree 如下：

ReduceSinkDeDuplication

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

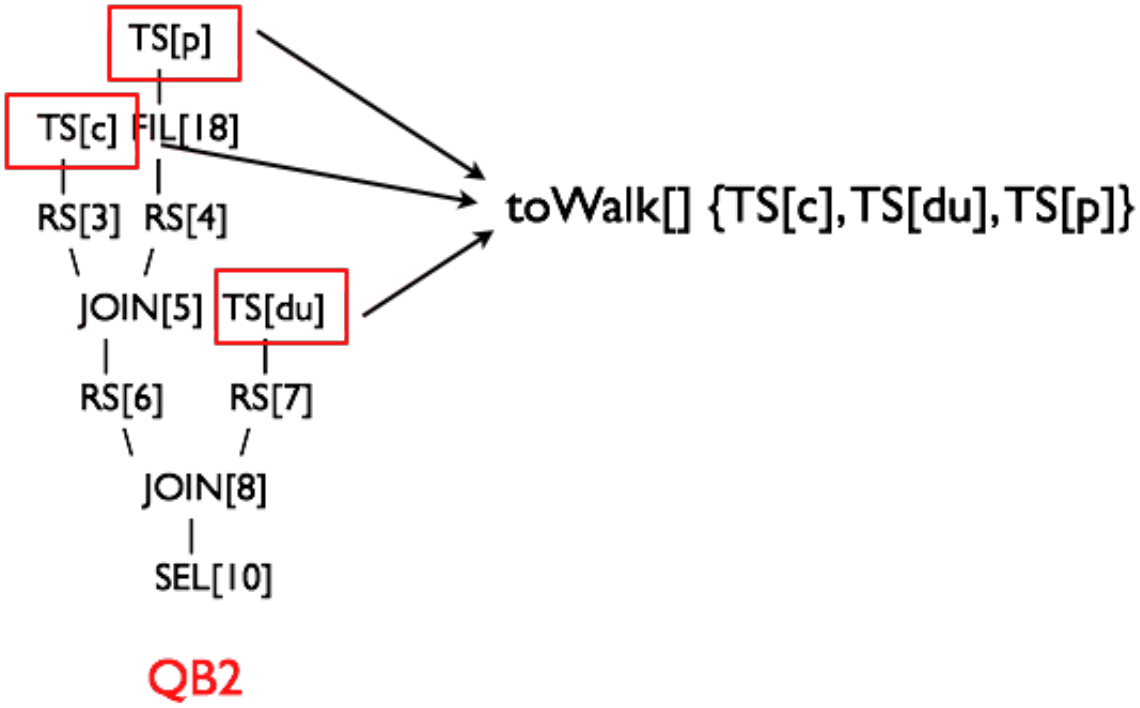
对输出表生成 MoveTask

由上一步 OperatorTree 只生成了一个 FileSinkOperator，直接生成一个 MoveTask，完成将最终生成的 HDFS 临时文件移动到目标表目录下

```
MoveTask[Stage-0]
Move Operator
```

开始遍历

将 OperatorTree 中的所有根节点保存在一个 toWalk 的数组中，循环取出数组中的元素（省略 QB1，未画出）



开始遍历

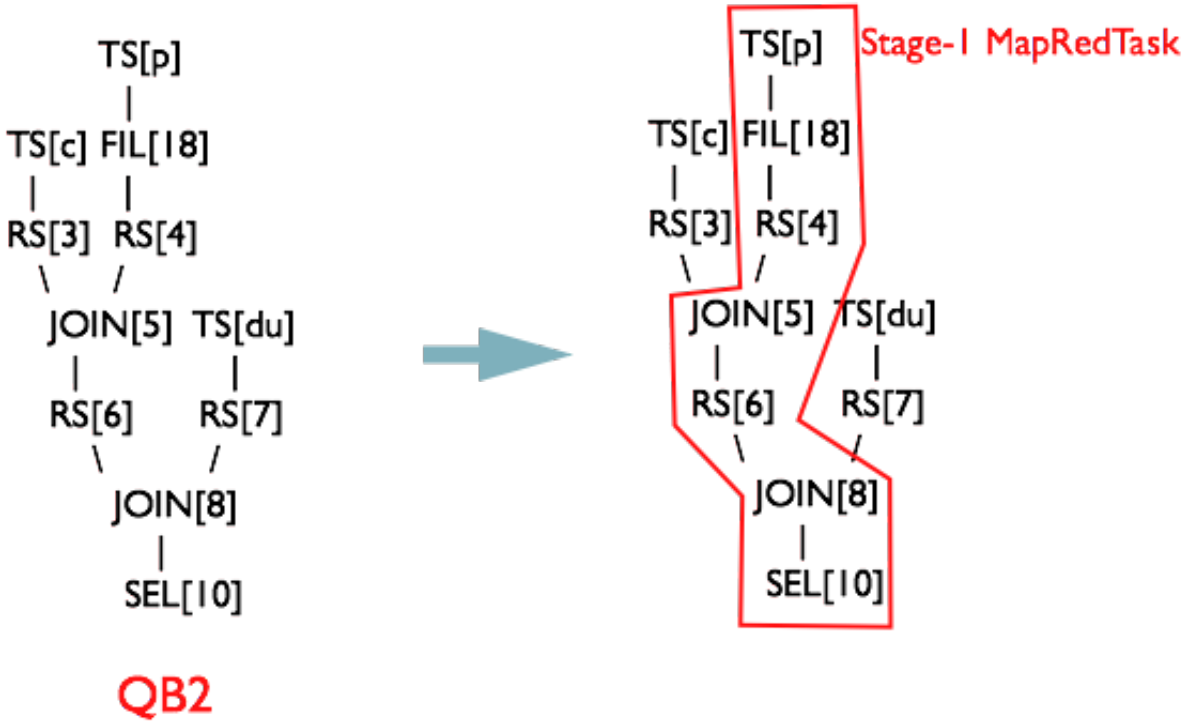
取出最后一个元素 TS[p] 放入栈 opStack{TS[p]} 中

Rule #1 TS% 生成 MapReduceTask 对象，确定 MapWork

发现栈中的元素符合下面规则 R1（这里用 python 代码简单表示）

```
"".join([t + "%" for t in opStack]) == "TS%"
```

生成一个`MapReduceTask[Stage-1]`对象，`MapReduceTask[Stage-1]`对象的`MapWork`属性保存 Operator 根节点的引用。由于 OperatorTree 之间之间的 Parent Child 关系，这个时候`MapReduceTask[Stage-1]`包含了以`TS[p]`为根的所有 Operator



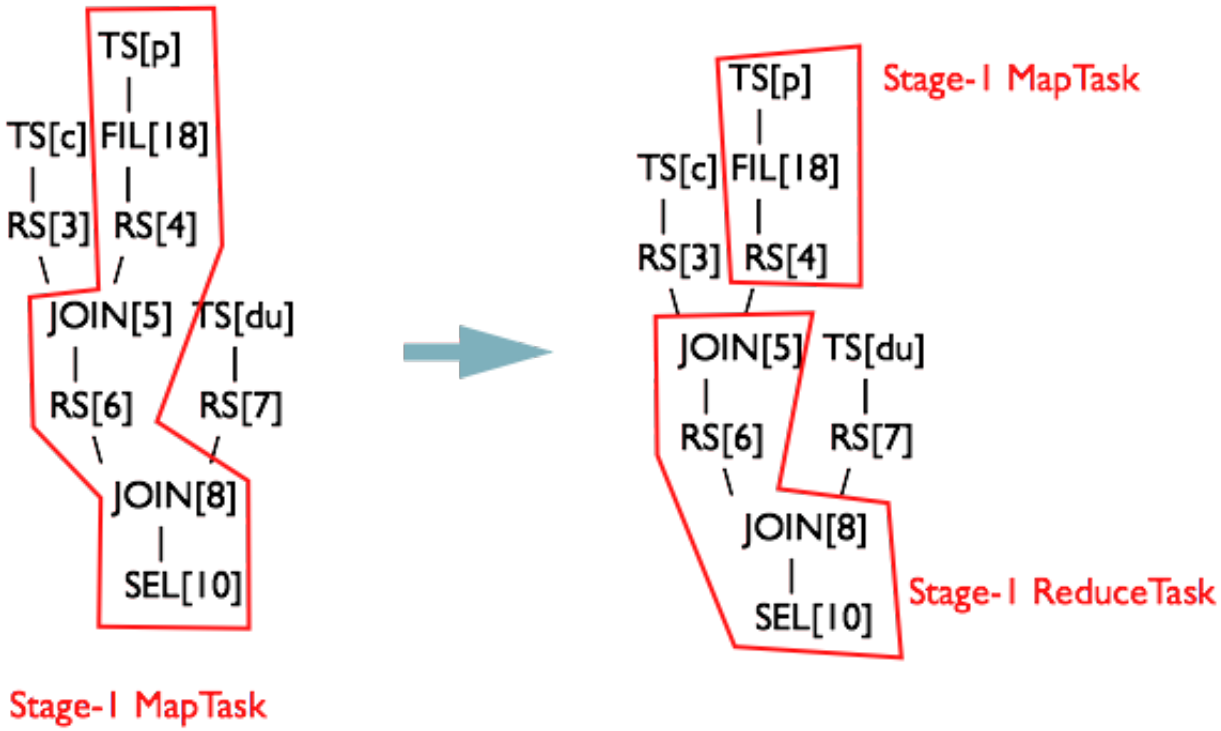
Stage-1 生成 Map 阶段

Rule #2 TS%.*RS% 确定 ReduceWork

继续遍历 TS[p] 的子 Operator，将子 Operator 存入栈 opStack 中 当第一个 RS 进栈后，即栈 opStack = {TS[p], FIL[18], RS[4]} 时，就会满足下面的规则 R2

```
"".join([t + "%" for t in opStack]) == "TS%.*RS%"
```

这时候在`MapReduceTask[Stage-1]`对象的`ReduceWork`属性保存`JOIN[5]`的引用



Stage-1 生成 Reduce 阶段

Rule #3 RS%.*RS% 生成新 MapReduceTask 对象，切分 MapReduceTask

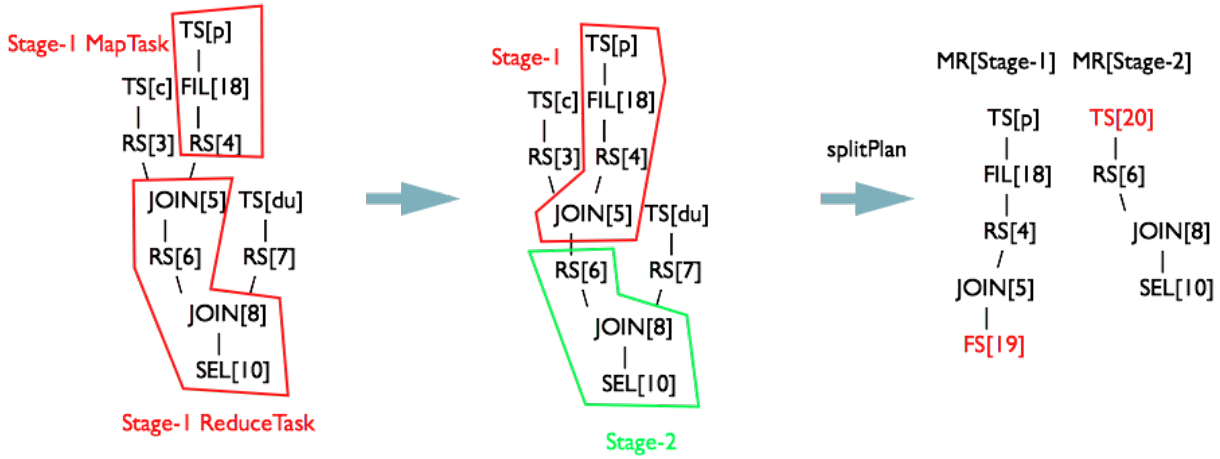
继续遍历 JOIN[5] 的子 Operator，将子 Operator 存入栈 opStack 中

当第二个 RS 放入栈时，即当栈`opStack = {TS[p], FIL[18], RS[4], JOIN[5], RS[6]}`时，就会满足下面的规则 R3

```
"".join([t + "%" for t in opStack]) == “RS%.*RS%” //循环遍历opStack的每一个后缀数组
```

这时候创建一个新的`MapReduceTask[Stage-2]`对象，将 OperatorTree 从`JOIN[5]`和`RS[6]`之间剪开，并为`JOIN[5]`生成一个子 Operator `FS[19]`，`RS[6]`生成一个`TS[20]`，`MapReduceTask[Stage-2]`对象的`MapWork`属性保存`TS[20]`的引用。

新生成的`FS[19]`将中间数据落地，存储在 HDFS 临时文件中。

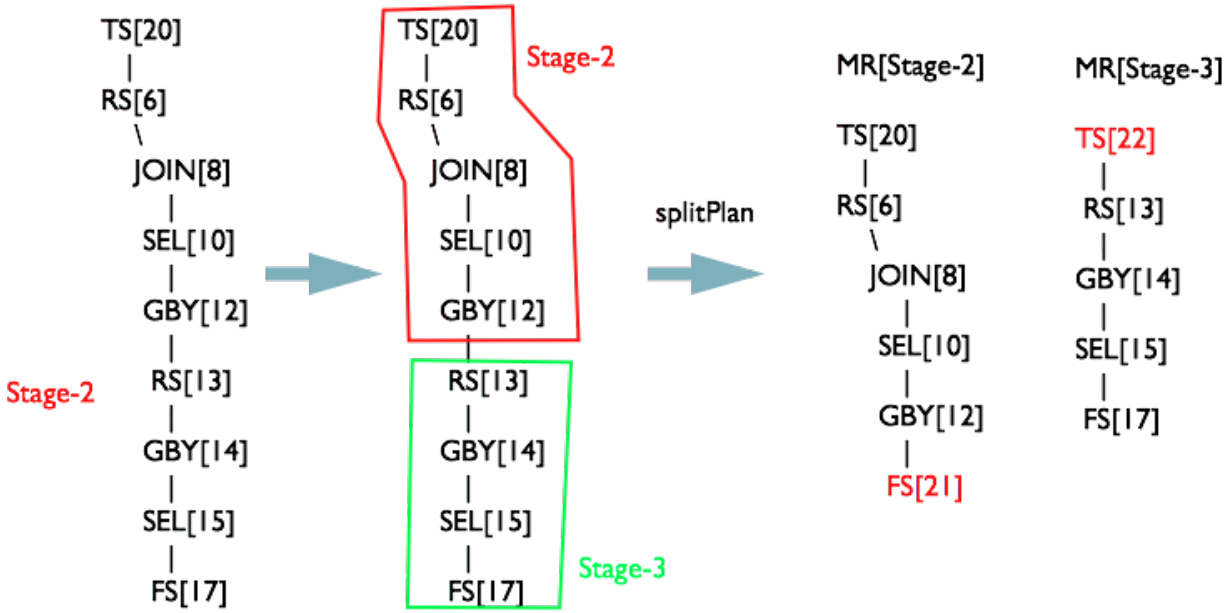


Stage-2

继续遍历 RS[6] 的子 Operator，将子 Operator 存入栈 opStack 中

当`opStack = {TS[p], FIL[18], RS[4], JOIN[5], RS[6], JOIN[8], SEL[10], GBY[12], RS[13]}`时，又会满足 R3 规则

同理生成`MapReduceTask[Stage-3]`对象，并切开 Stage-2 和 Stage-3 的 OperatorTree



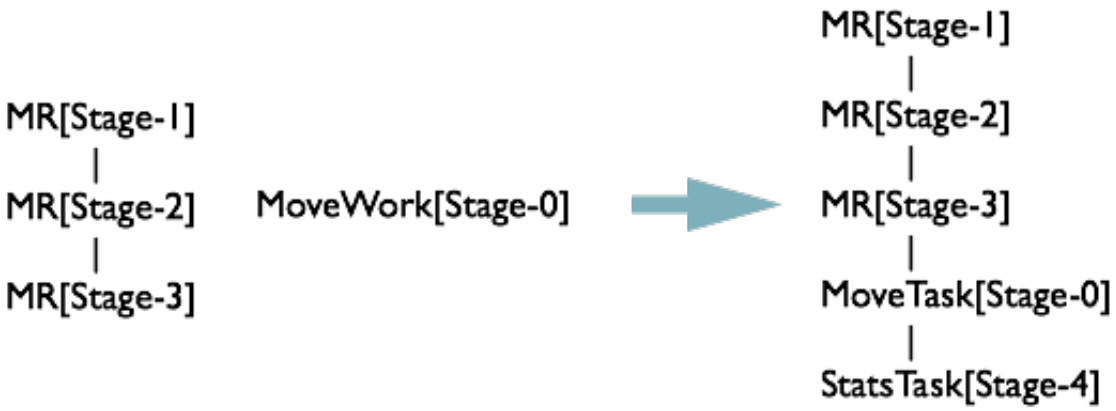
Stage-3

R4 FS% 连接 MapReduceTask 与 MoveTask

最终将所有子 Operator 存入栈中之后，`opStack = {TS[p], FIL[18], RS[4], JOIN[5], RS[6], JOIN[8], SEL[10], GBY[12], RS[13], GBY[14], SEL[15], FS[17]}` 满足规则 R4

```
"".join([t + "%" for t in opStack]) == “FS%”
```

这时候将`MoveTask`与`MapReduceTask[Stage-3]`连接起来，并生成一个`StatsTask`，修改表的元信息



MoveTask

合并 Stage

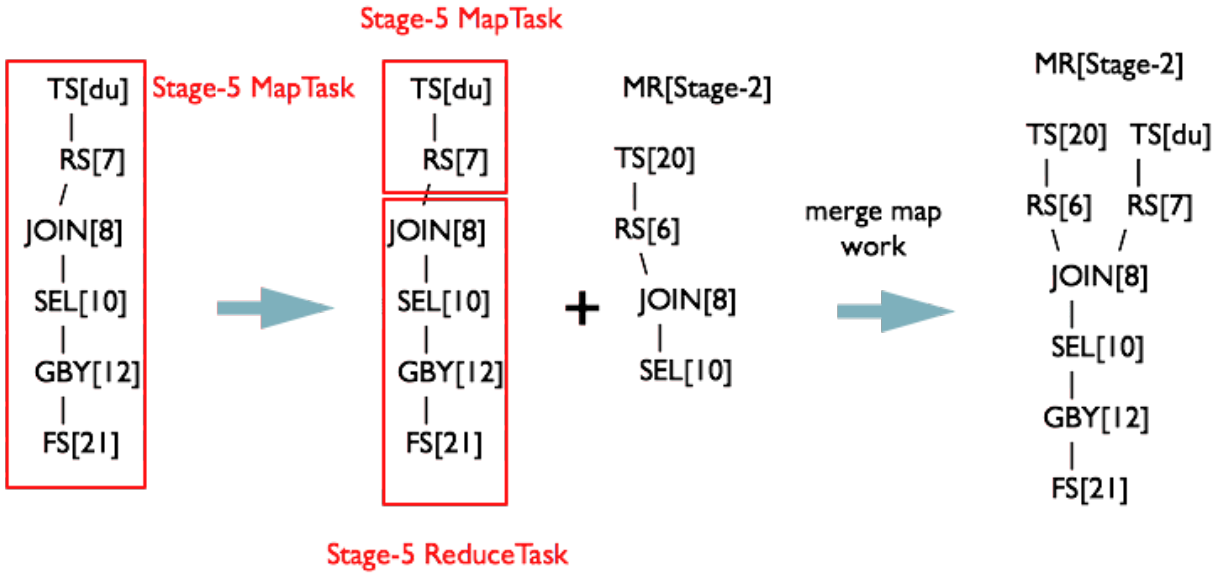
此时并没有结束，还有两个根节点没有遍历。

将 opStack 栈清空，将 toWalk 的第二个元素加入栈。会发现`opStack = {TS[du]}`继续满足 R1 TS%，生成`MapReduceTask[Stage-5]`

Stage-5

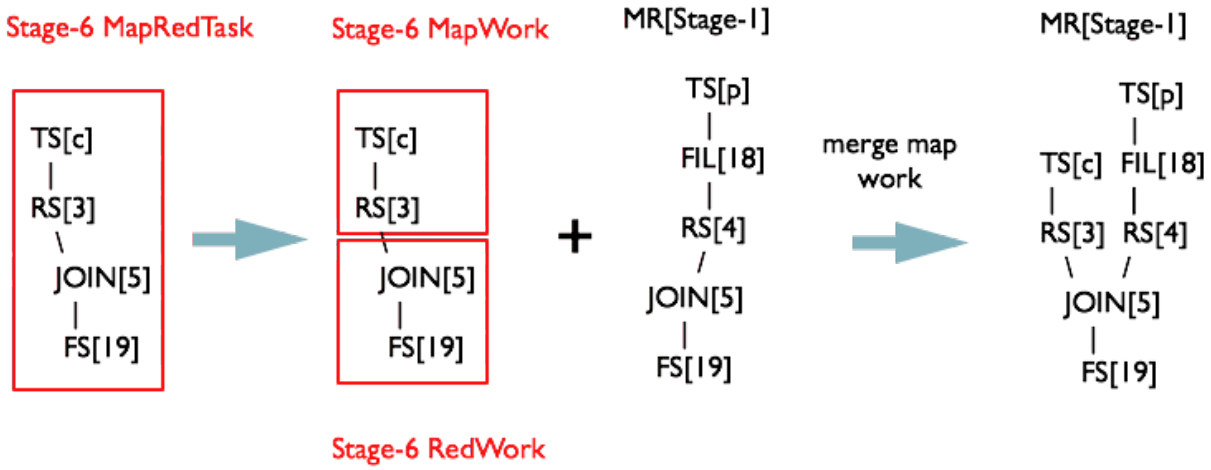
继续从`TS[du]`向下遍历，当`opStack={TS[du], RS[7]}`时，满足规则 R2 TS%.*RS%

此时将`JOIN[8]`保存为`MapReduceTask[Stage-5]`的`ReduceWork`时，发现在一个 Map 对象保存的 Operator 与 MapReduceWork 对象关系的`Map<Operator, MapReduceWork>`对象中发现，`JOIN[8]`已经存在。此时将`MapReduceTask[Stage-2]`和`MapReduceTask[Stage-5]`合并为一个 MapReduceTask



合并 Stage-2 和 Stage-5

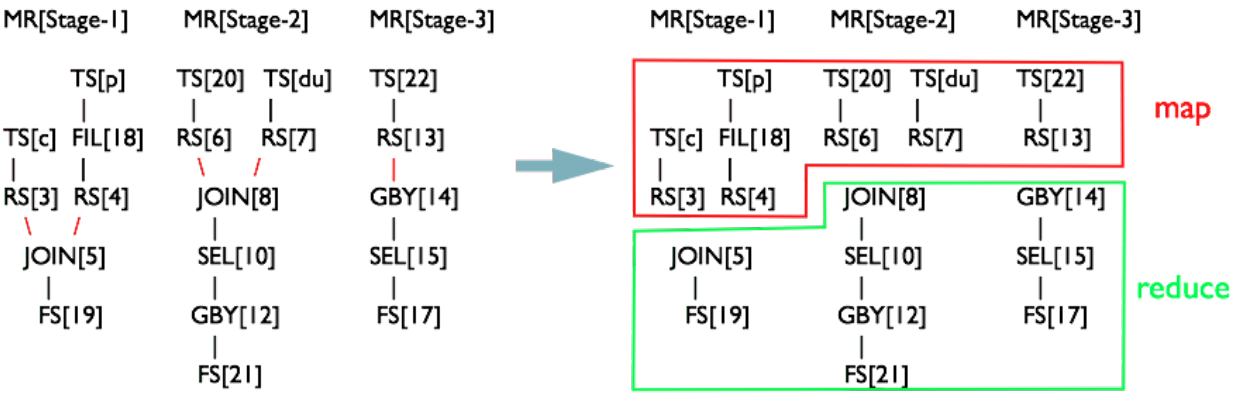
同理从最后一个根节点`TS[c]`开始遍历，也会对 MapReduceTask 进行合并



合并 Stage-1 和 Stage-6

切分 Map Reduce 阶段

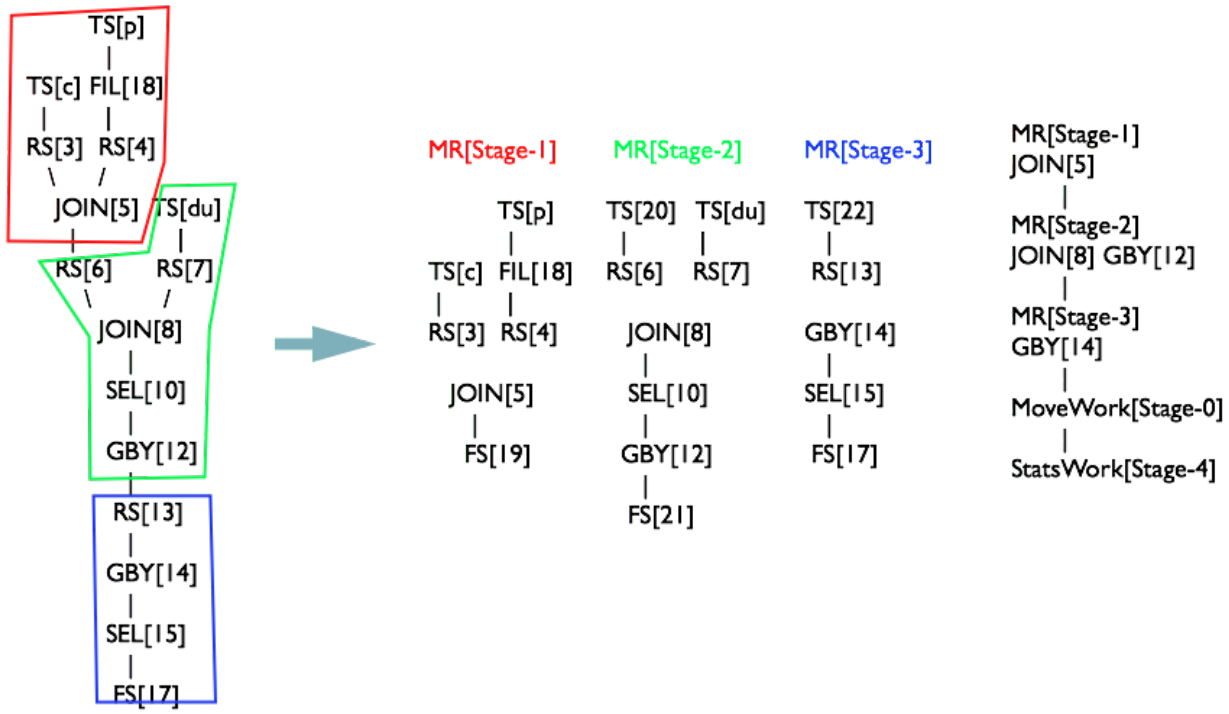
最后一个阶段，将 MapWork 和 ReduceWork 中的 OperatorTree 以 RS 为界限剪开



切分 Map Reduce 阶段

OperatorTree 生成 MapReduceTask 全貌

最终共生成 3 个 MapReduceTask，如下图



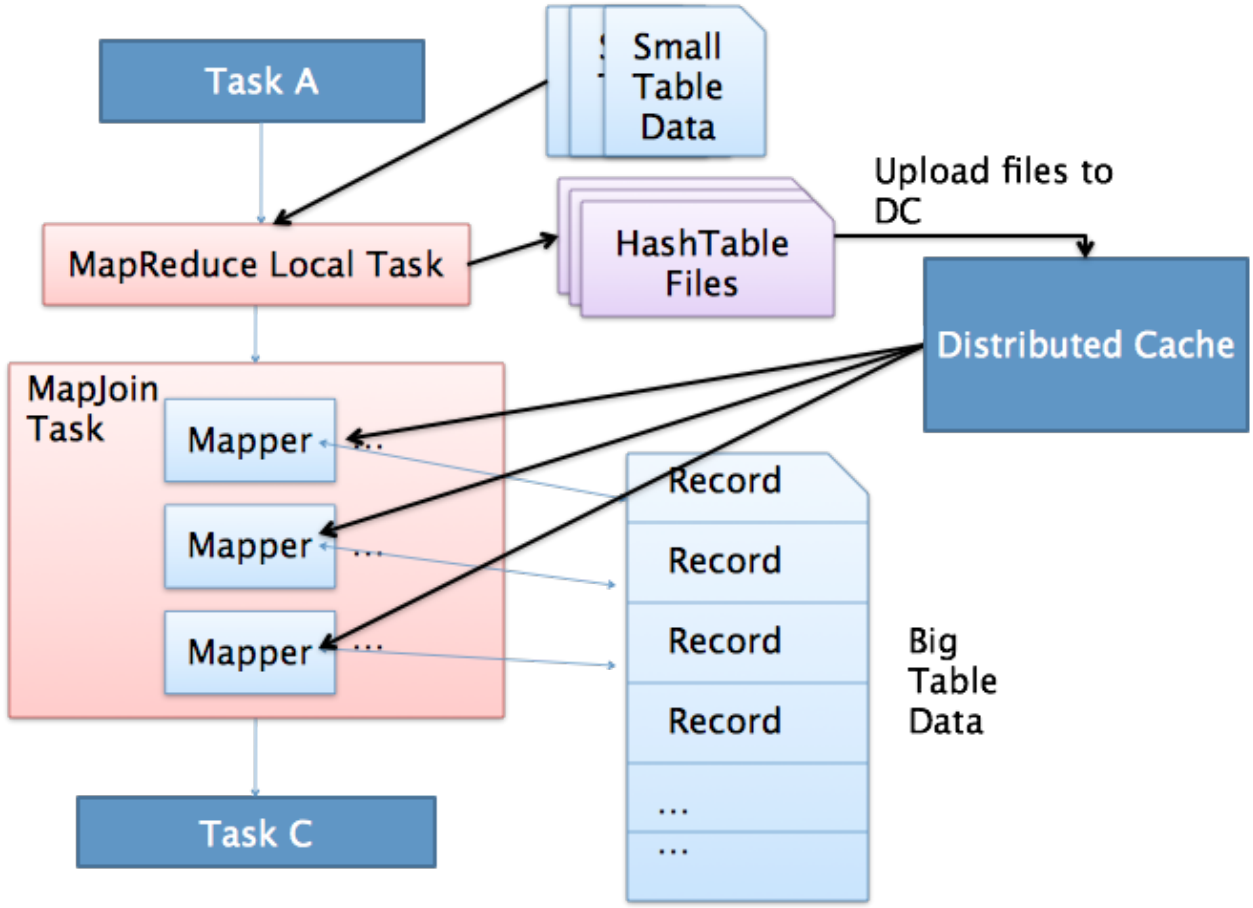
OperatorTree 生成 MapReduceTask 全貌

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

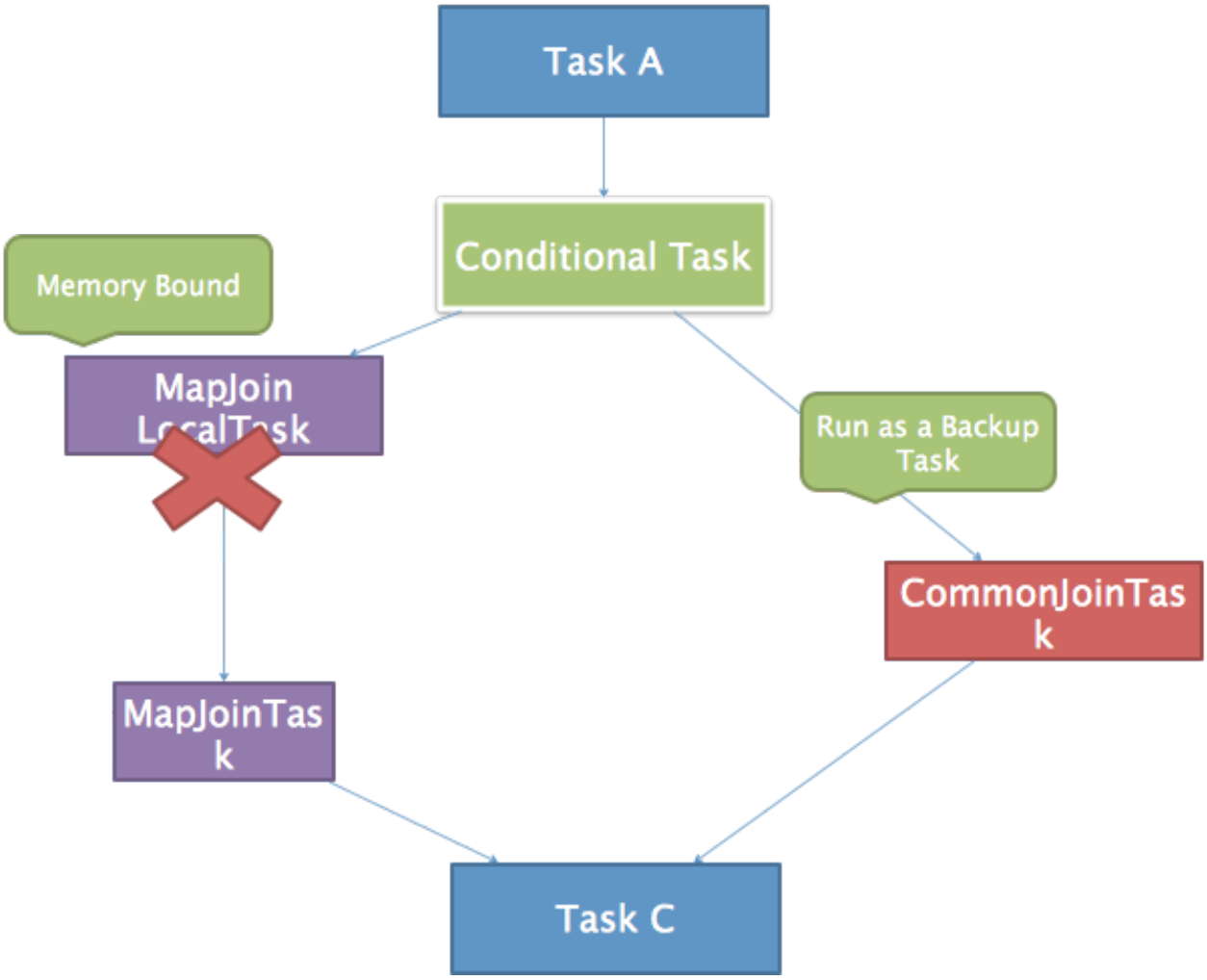


mapjoin 原理

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。



conditionaltask

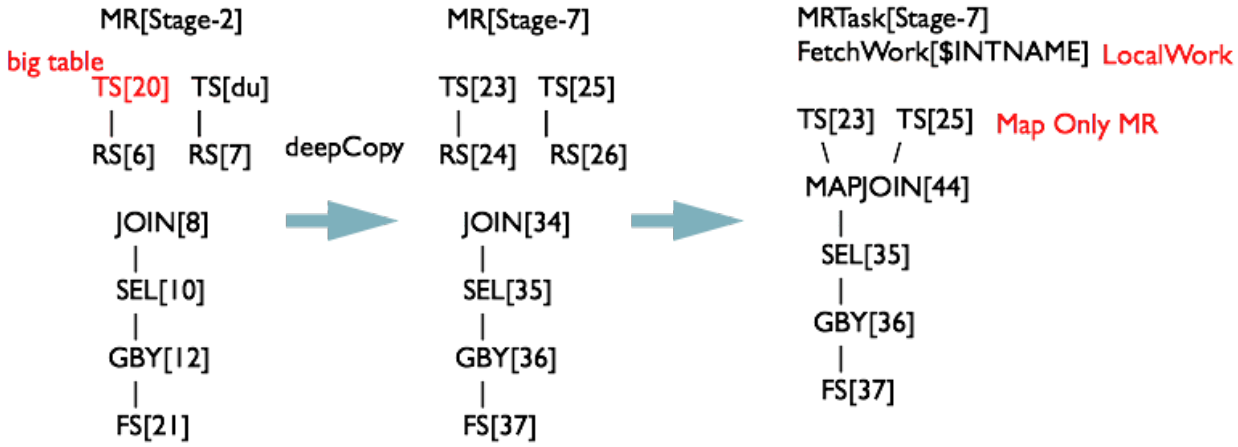
如果 Join 的两张表一张表是临时表，就会生成一个 ConditionalTask，在运行期间判断是否使用 MapJoin

CommonJoinResolver 优化器

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

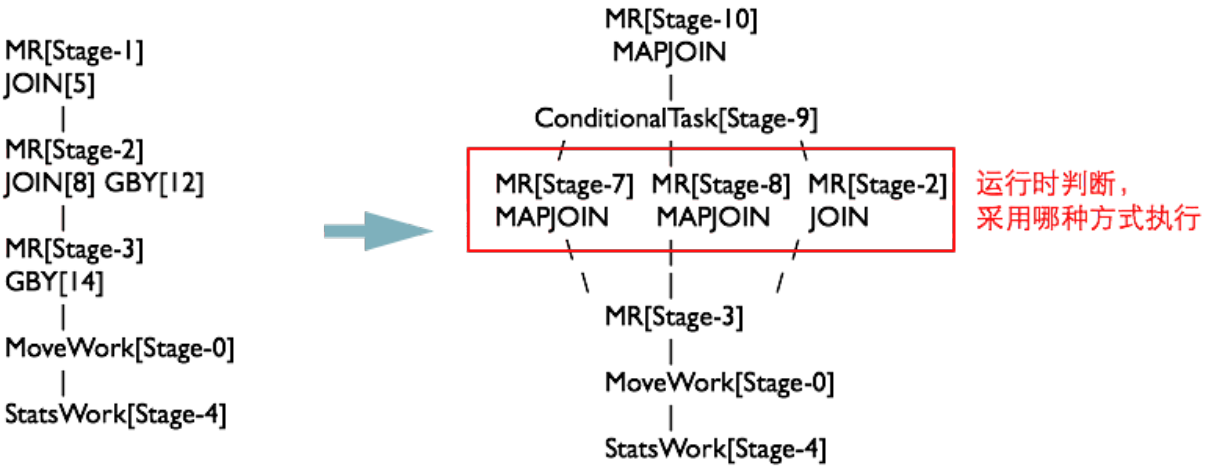
1. 深度优先遍历 Task Tree
2. 找到 JoinOperator，判断左右表数据量大小
3. 对与小表 + 大表 => MapJoinTask，对于小 / 大表 + 中间表 => ConditionalTask

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。



mapjoin 变换

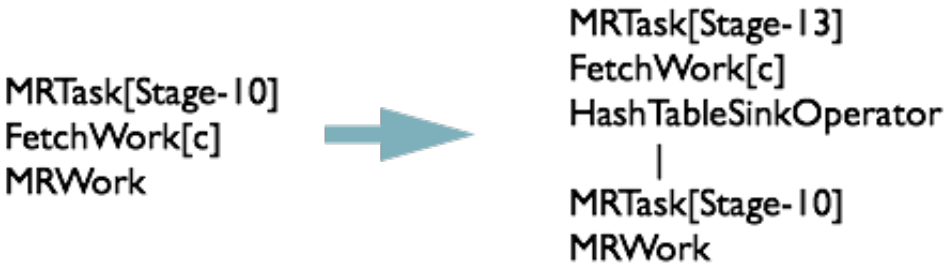
MapReduceTask 经过变换后的执行计划如下图所示



mapjoin 变换

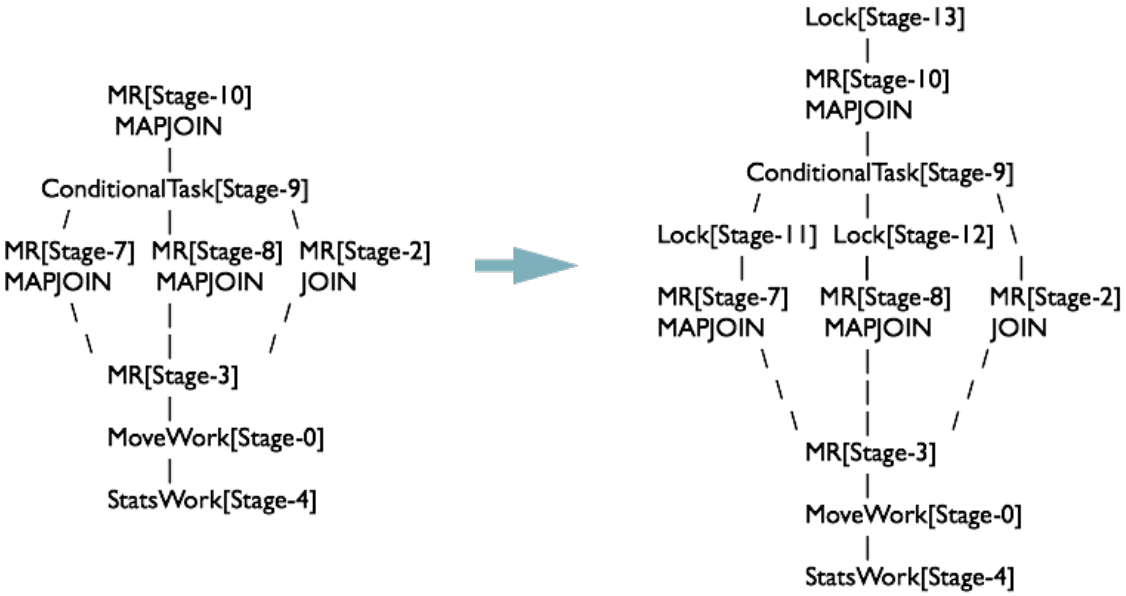
MapJoinResolver 优化器

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



MapJoinResolver

最终 MapJoinResolver 处理完之后，执行计划如下图所示



MapJoinResolver

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