HIVE的DML语句执行简介

1. **执行入口简介**

对于一般语句的执行入口都是在Driver.run(String command)这个方法中，runInternal🡪compile🡪execute

**词法+语法解析**:调用antlr的解析类，生成一棵ast语法树



**语义解析：**以嵌套的方式解析出一个sql中各个数据项，调用SemanticAnalyzer.doPhase1,主要是遍历整个ast树结构，找出selett、dest、group等选项。

**逻辑执行计划：**生成Operator的ast执行树结构，一个个的operator（其中有属性表明其父Operator和childOperator） ，这一步生成的operator其实就已经是可以执行的一个个算子。

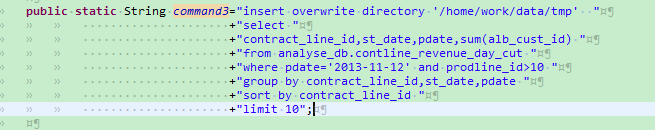
**Optimizer :** 在逻辑执行计划的基础上根据一些规则来重新逻辑执行计划，以达到更高的效率 。

**物理执行计划:**将优化后的逻辑执行计划，翻译成可在mr上执行的物理执行计划，就是将上面的一个个算子组合起来，然后调用，其输出是一个List<Task> rootTask 。

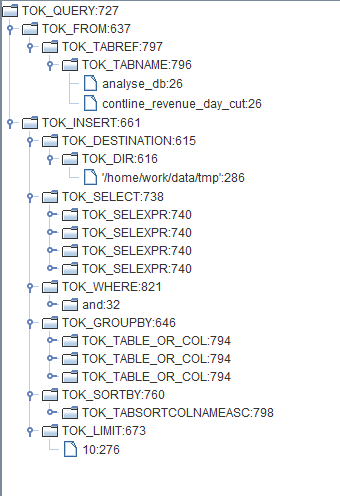
**rootTask的执行：**物理执行计划的输入是一堆的List<Task> rootTask（即最上层的task，没有前置依赖的），task中存储有自己的前置依赖和child节点，每个task有自己的execute方法，按照依赖关系一次调度起来就行了。

1. **词法+语法解析**

利用HiveParse（通过hive的语法.g文件生成的）分析sql语句，然后生成一颗有语义的AST树，以以下语句为例



其语法树的结构如下：



一个dml语句最终都翻译成为TOK\_QUERY类型的ast树，这棵树可以看成是个平行的结构，主要包含这几部分属性：

**TOK\_FROM**：就是数据来源，这个可以是一个的table或者subquery、join、union、lateralview(hive的udtf)这几种，其中后面的四种内部又是一个新的嵌套结构，嵌套一个新的 TOK\_QUERY，这个TOK\_QUERY也包含相同的这些项。

**TOK\_INSERT**：数据的去向，包含一堆平行的结构，*TOK\_DESTINATION(数据要insert的地方，如果只是简单的select，hive也会翻译出insert到一个tmpfile)、TOK\_SELECT(对于要插入的数据的select操作)/TOK\_SELECTDI(select distinct)、TOK\_WHERE(filter条件)、TOK\_GROUPBY(group by列)、TOK\_DISTRIBUTEBY(distributeby)、TOK\_SORTBY(sort)/TOK\_ORDERBY(order)、TOK\_HAVING(having条件)、TOK\_LIMIT(limit)*这些元素，还有一些不常用的没有列举。

简单的一个语义流程就是：从from的数据源取数据🡪where条件过滤—>group by/distribute by操作--->having条件过滤-🡪sort by/order by操作 🡪 limit 操作 🡪写到dest中（这个在hive中叫FileSinkOperator，简称FS）。

**只有from后面才会嵌套结构（subquery、join、union、lateralview），其它的可以看作是一个hql中平行的结构。**

 使用JTree将ast的结果封装了一下，可以查看相应的树形结构。

1. **语义解析**

BaseSemanticAnalyzer.analyze-🡪SemanticAnalyzer. analyzeInternal ,这个方法中其实做了语义解析、逻辑执行计划、optimizer、物理执行计划这些操作都是这这个方法中运行的。

一个hql语句唯一有可能有嵌套的部分就是from后面的东东（lateview也可以认为是from后面的），这种可以使用递归的方式处理，在最上层的hql中不必关系嵌套内的东东（事实上hive的处理对于每个最上层/嵌套结构最终对外暴露都是一个最上层operator，这个operator可以对外提供一行行的数据），其它都是平行的结构，语义解析的过程可以看作是寻找一个hql的这些部分，这个主要是在看SemanticAnalyzer.doPhase1()方法中做的 。

Insclause-$num的概念 ，因为hive支持这样的语句 ：

from    tab\_from

       insert   tab\_in1   select \*\*\*\*

       insert   tab\_in2   select \*\*\*\*

       insert   tab\_in3   select \*\*\*\*

所以在一次解析中有一个from table ，但是会有多组(insert\*\*\*select \*\*\* where\*\*\*group by\*\*\*\*),这个每组叫一个insclause，编号从0开始，依次增加，在*TOK\_INSERT*节点下是平行的，每组insclause都是先*TOK\_DESTINATION*节点，然后再跟着相应的其它节点，所以doPhase1方法才会出现一下这样的解析语句：



表示遇到了一个新的insclause，计数+1 。

解析的信息主要存储在QB以及其的一个属性QBParseInfo中 ，主要的解析信息如下：

1. case HiveParser.TOK\_SELECTDI：抽取insclause中的select distinct信息
2. case HiveParser.TOK\_SELECT:抽取select中的selExp信息（设置到QBParseInfo.destToSelExpr）、select中的agg函数信息（设置到QBParseInfo.destToAggregationExprs）、select节点中hint（mapjoin之类的,设置到QBParseInfo.hints中）信息并设置到QBParseInfo中相应的字段里面
3. case HiveParser.TOK\_WHERE：抽取where后面的filter信息，设置到QBParseInfo.destToWhereExpr中 。
4. case HiveParser.TOK\_DESTINATION：抽取这层hql的dest节点，存储到QBParseInfo.nameToDest中，由于每个dest是一组insclause的开始，所以case到这个会将insclause的计数器+1；
5. case HiveParser.TOK\_FROM：针对数据源节点的处理，可以分为几种：直接的table ref,设置到QB.aliasToTabs中，别名-🡪table映射 ；针对只查询subquery，针对这个子查询递归调用doPhase1，然后得到子查询的QBExpr(其有两个属性，一个是这个子查询的别名alias，一个是这个子查询的QB信息，针对于uninon操作有两个QB对象)；针对lateralView，这个暂时没有细看 ；针对join操作，由于无论几轮的join最终都会转成嵌套的jioin结构，所以会针对join的left、right分别处理，如果是table这table处理相同，如果是subquery这subquery相同 。
6. case HiveParser.TOK\_CLUSTERBY:处理cluster by，设置到QBParseInfo.destToClusterby中 。
7. HiveParser.TOK\_DISTRIBUTEBY:处理distribute by属性，同上
8. HiveParser.TOK\_SORTBY、HiveParser.TOK\_ORDERBY：同上，设置相应的操作属性，同样设置到QBParseInfo中。
9. HiveParser.TOK\_GROUPBY\*\*\*：设置group by节点的属性，也是设置到QBParseInfo中。
10. case HiveParser.TOK\_HAVING、HiveParser.TOK\_LIMIT：同上，设置每个insclause的这两个属性 。
11. 待补充

整个doPhase1就是设置这些属性的一个遍历 。

**getMetaData()，获取元数据**

获取source table的元数据，如果一个table实际上是一个view，将其重写为view的定义；

递归的为每个子查询中的源表获得元数据；

获取所有destination table/dir/local dir的元数据；

**关于字段级别的血缘关系的一点感想**

至此，整个hive的语义解析部分就完成了，在这个基础上是可以获取字段级别的血缘关系的，对于一个基本的select a1 ,a2,udf1(a3) from table/subquery alias where \*\*\* group by c5 sort/order \*\* limit 10 ,这样的语句，影响select 后面的列（如果要是有insert table的话，就把select后面的和table的schema对应起来），只有udf表达式、table、group by后面的字段 ，对于使用agg聚合函数的则group by后面的列比较关键，否则就是from 和where了 。

对于第一层sql，可以先计算出select后的每个列和alias的血缘关系，然后再对alias递归做相应的操作，然后再一步步的替换到原始的table 。

1. **逻辑执行计划**

通过SemanticAnalyzer.genPlan(QB qb)方法获得的执行计划，QB是上一步语义解析填充的对象，该对象返回的结果是逻辑执行计划最后一步的operator 。

**关于operator**

operator就是具体的操作，其中有可调用处理数据行的processOp(Object row, int tag)方法，已经可以被调用来处理数据的，每个operator包含**childOperators**、**parentOperators**，根据这些可以构造operator的DAG属性结构，由于每个operator会有多个parent，所以tag来标志哪个parent传来的数据，每个operator中还有一个**RowSchema**属性，表明这个operator产出的数据的schema，RowSchema不仅包含正常列还包含partitions列，在hive0.11后还支持虚拟列的概念，对于TableScan操作，有filename、fileoffset这些虚拟列可以用作一些特殊用途（比如过滤table数据文件夹中不需要的文件） ，每个operator都一个**OperatorDesc**，相当于这个operator的配置文件，由于operator不同，所以配置也需要相应自己的实现 ，里面保留着每个operator操作时候需要的信息。

由于operator是一个dag树的概念，每个operator通过processOp(Object row, int tag)方法来处理自己的数据，处理完以后调用forward(Object row, ObjectInspector rowInspector)，来将处理好的数据传递给子operator继续处理，其实forward中就是顺序调用child的processOp方法，tag根据自己在child的parent中的index来计算的，整个数据的处理就是这样的一个递归过程，对于suquery/lateview/join等子操作都是最终对外暴露一个operator，将数据传递给外层的，即外层是这个operator的子operator 。

整个SemanticAnalyzer.genPlan(QB qb)后，将operator的dag树的rootOperator存储在SemanticAnalyzer.topOps中，是整个DAG树的root operator ，从这几个开始可以遍历整个operator树 ，这里因为topOps需要存储整个sql包括subquery这里面的root operator，所以只能存储在全局变量中，而不能在每个QB中 。

**genplan的流程**

1. 生成from后面的数据源的operator，如果是table则直接生成TableScanOperator，是子查询则生成子查询的Operator ，这里面的operator是子查询的最后一个operator，为当前提供row数据的。
2. 对于PTF、winfun等流程暂未详细看 。
3. 针对join的处理，如果是from后面有join操作的话，则上一步中会生成两个operator，一个是join的left以及是join的right的，这一步会根据QB中的joinTree信息，将这两部分组合起来生成一个operator，作为提供数据源的operator，对于join的处理会涉及到join操作的合并，但是没有考虑mapjoin，mapjoin是在后面的逻辑计划optimizer中做的 。
4. genBodyPlan(),先生成where条件的filteroperator，然后在filter operator的基础生成group by的operator，处理group by时候会根据是否处理倾斜、是否mapside-join、是否合并多个insclause的group by等情况生成不同的group by operator，这个operator是上一步的child operator 。
5. 然后在group by operator之后，在genPostGroupByBodyPlan()方法中依次处理having（生成一个新的filterOperator）、Select操作、sort/order by、limit、FileSinkOperator操作，下一个为上一个的子节点 。
6. Demo如下 ：TS[0]-->FIL[1]-->SEL[2]-->GBY[3]-->RS[4]-->GBY[5]-->SEL[6]-->RS[7]-->EX[8]-->LIM[9]-->RS[10]-->EX[11]-->LIM[12]-->FS[13] ，数字为每个operator的id

**Operator查看工具**

1. 因为SemanticAnalyzer.topOps是private变量，所以需要改动一些相应类属性的访问权限才能使用
2. 注释掉物理执行计划的代码，生成物理执行计划的时候会破坏SemanticAnalyzer.topOps的树形结构



**常用operator以及作用（转自别人文档）**

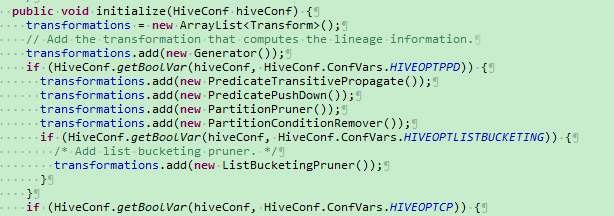
OperatorFactory是算子工厂，该工厂根据一个算子描述实例生成相应的算子实例。目前注册的算子及其描述包括以下这些。



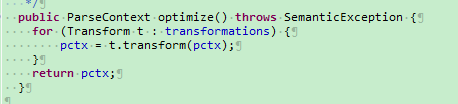


1. **Optimizer优化逻辑执行计划**

**Optimizer的初始化**

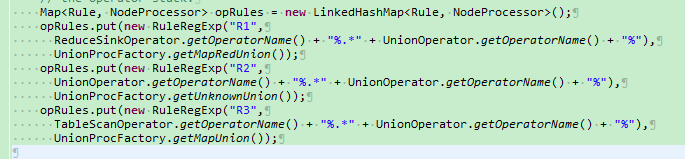


根据配置添加一大推的Transform实现进去（一个Transform相当于一个优化器）,优化operator时候，顺序调用Transform. Transform（pctx）就可以啦

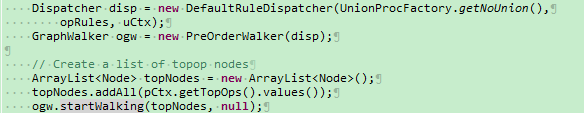


**对于一个Transform的执行流程**

对于每一个Transform其实都是在内部维护了一系列的规则Rule到优化器NodeProcessor的映射map，例如UnionProcessor：

****

**Rule:一个匹配的字符串+处理process**

****

GraphWalker是一个对于树结构的遍历器，这个实现为前序遍历器，Dispatcher disp中封装的是那些规则🡪优化处理器的Map ，方法调用：GraphWalker.startWalking—后续遍历执行计划树🡪针对每个节点调用Dispatcher. Dispatch方法 。

其中Dispatcher. dispatch(Node nd, Stack<Node> stack, Object... nodeOutputs)方法，是针对树上的某个节点进行处理（其实就是规则匹配，使用匹配到的优化器处理一遍），他的几个参数释义如下：

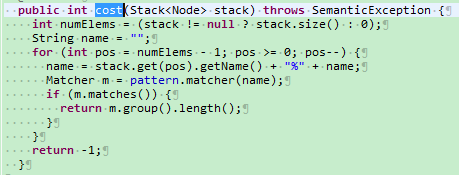
Node nd：待处理的节点

Stack<Node> stack:该节点的前置节点，因为是后序遍历，所以stack中是这个节点的一层层父节点 。

Object[]nodeOutputs:是每个子节点经过优化器处理后的结果，一般都是null ，以下是DefaultRuleDispatcher. Dispatch方法



对于Rule的cost方法，代码如下：



返回的值是匹配的字符串的长度，即将前置父节点按照”辈分”打成字符串，然后和每条规则Rule进行字符串匹配，然后返回匹配结果字符串长度，选择cost最大的即是选择匹配最长字符串的规则 。

操作的name类似于这样的字符串：

TS : TableScan Operator

FLT : filter 操作

**Hive中一些基于规则的optimizer**

1. Generator

处理字段以及表级别血缘关系的优化器，本身只是提取数据的

1. PredicateTransitivePropagate

针对的操作流:JOIN🡪RS🡪FIL ,join完后，写文件，然后再过滤

过滤条件提前到RS之前

1. PredicatePushDown

谓词下推

1. PartitionPruner

Partition裁剪，将partitions条件放在

1. PartitionConditionRemover

移除partitions条件

1. ListBucketingPruner

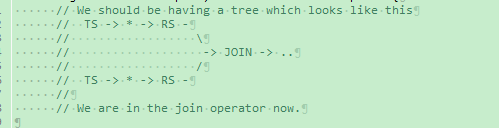
对于bucket的TS裁剪，不是很明白

1. ColumnPruner

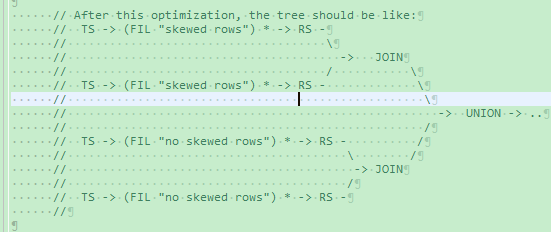
列裁剪 ，通过血缘关系，修改Operator中的RowResolver对象

1. SkewJoinOptimizer

针对数据倾斜的处理，将倾斜数据和非倾斜数据分开join，然后再union

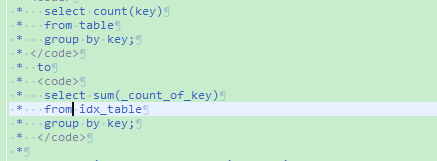


修改后



1. RewriteGBUsingIndex

针对有index的表，做的聚合函数优化，such as



1. GroupByOptimizer

针对sorted && bukect的表做group by 的优化 ，聚合可以完全在map端完成

1. SamplePruner

把Sample的smaple下放到TS的操作上

1. MapJoinProcessor

Join操作转换成mapjoin的操作

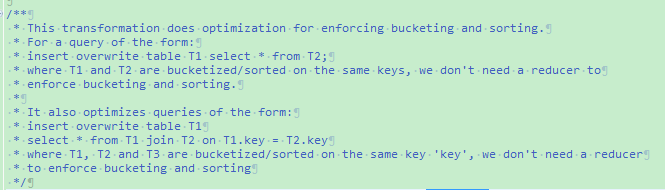
1. BucketMapJoinOptimizer

针对join表示bucket的表做的bucketmapjoin

1. SortedMergeBucketMapJoinOptimizer

Bucketsortmapjoin，不许用本地hashTable的操作了

1. BucketingSortingReduceSinkOptimizer



针对bucket sorting 的数据，节省一轮sort的reduce

1. UnionProcessor

只是识别是否union的两个subquery都是只需要在map就可以完成，这个只是存储这个信息，不对operator做任何操作 。

1. JoinReorder

针对join时候，小表放左边来做的优化，暂时hive只是通过标记的形式（貌似是使用stream的hint来标记的），后续这块可以是实现为基于cost的优化

1. ReduceSinkDeDuplication

如果两个RS是相同的，则合并这两个文件的吐出

1. NonBlockingOpDeDupProc



合并连续的sel/sel 以及filter/filter这样的操作

1. GlobalLimitOptimizer

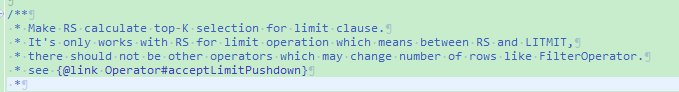
简单的将limit N的操作下推到具体的TS操作中

1. CorrelationOptimizer

一整篇优化的论文

<http://www.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-11-7.pdf>

1. LimitPushdownOptimizer



针对limit的下推操作，将limit N的操作转化为top-N的操作，可以在reduce、sort的基础上做优化，本质上应该是采用topN的堆算法 。

1. SimpleFetchOptimizer



就是针对直接取数的select \* from table 这样的，直接转化成本地的local的一个fetchTask

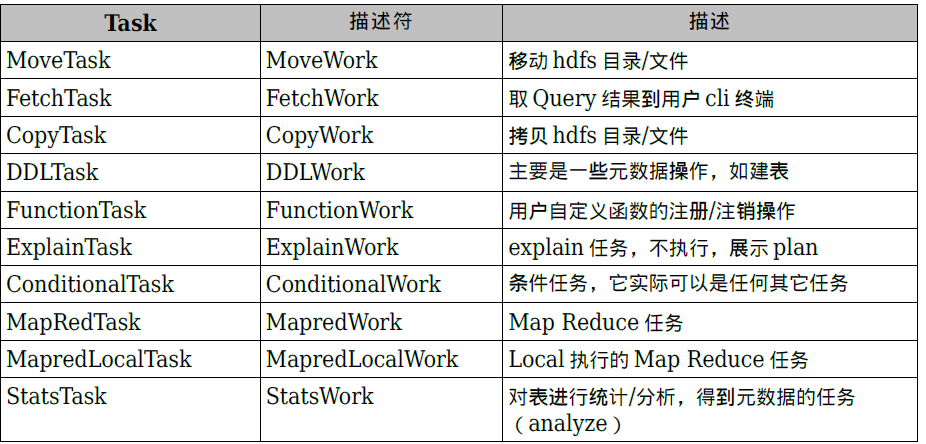
1. SimpleFetchAggregation

将最后的一些agg操作放到FetchTask中来做

1. **物理执行计划**

将以上的operator翻译成一个个的可执行的task，本部分暂未了解

常用的task



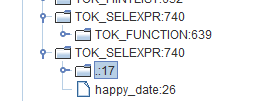
1. **rootTask的执行**

经过物理执行计划，会将所有的operator封装到一个个的task中，构成一个List<Task<? extends Serializable>> rootTasks ，所以root task的集合， task是和operator差不多的数据结构，每个task有自己的parentTask List以及childTask List，并且task有自己的executeTask()方法，可以运行 ，具体的执行过程在Driver.Execute()方法中，具体如下 ：

1. 初始化一个runnable的Queue<Task<? extends Serializable>> 和task和taskResult对应的map，将rootTasks加入到queue中。
2. 依次从queue中取出task，运行，运行成功后，看其子task是否满足运行条件（即所有parentTask都执行完毕），如果ok，则将相应的子task加入可运行的队列中 。
3. 重复步骤2，直到所有的task运行完毕 。
4. **Tips**

以上是整个hive运行的主流程：hql解析成语法树-语法解析-语义解析-逻辑执行计划-optimizer-物理执行计划—task执行 ，以下是看代码时候的一些笔记tips。

1. 关于TOK\_SELEXPR类型的节点



对于TOK\_SELEXPR来说，如果有别名的话，有两个子节点，第二个子节点是别名，第一个是表达式

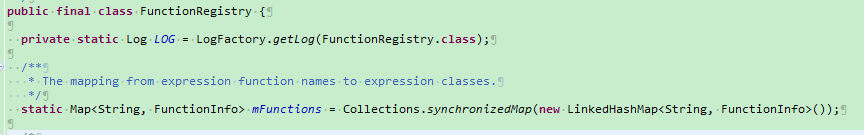
1. BaseSemanticAnalyzer.validate()  ，验证plan的结果是否是有效的
2. 关于一般的函数（udf/udaf）表达式

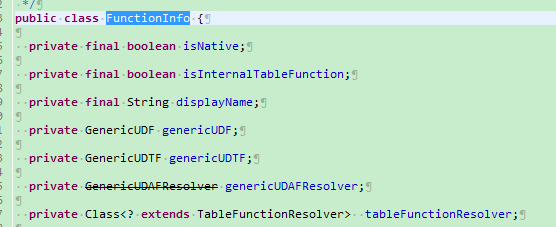
C:\Users\dingdongchao\AppData\Local\YNote\Data\dingdongchao@163.com\f3d061dfa72a42e888f560c4d0f7ec23\clipboard.png

child[0]位置上是函数名，1要计算的数值（列名称、嵌套的的fun等等）

1. 关于FunctionRegistry

     管理整个hive执行过程中所有的函数的，所以hive引入新的函数需要在代码中显示制定，以方便注册进来

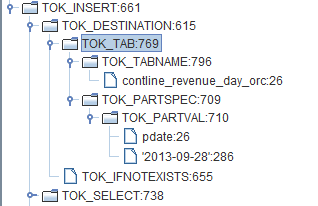
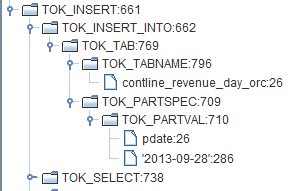




union的数据结构构成FunctionInfo信息，若是相应类型，则有值

1. TOK\_INSERT\_INTO 针对的是dest是table(partitions)的情况

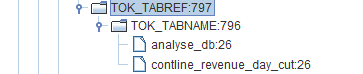
      TOK\_INSERT  的dest可以是directory 或者 local directory 或者  table(partition)

以上两个分别是insert into  和 insert overwrite  ，后一个支持if not exits

也可以看出，TOK\_TAB其实包含两部分信息，表名和分区信息

1. 直接对于table的引用结构



1. QBExpr  QB  QBParseInfo 依次为包含关系

           QBExpr qbexpr = new QBExpr(alias);

           QB qb1=qbexpr.getQB();

           QBParseInfo qbParse1=qb1.getParseInfo();

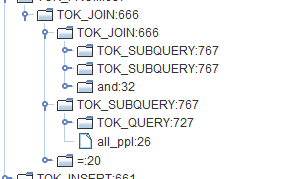
每个QBExpr中可以有两个QB，用于处理union的操作

每一个子查询TOK\_QUERY对应这一个QBExpr  ，可以看到QBExpr 的构造方法对应的就是TOK\_QUERY的别名，每个TOK\_QUERY必须有别名 。

1. 整个AST树以TOK\_QUERY为基本单位   ，所有的子查询都会重写为

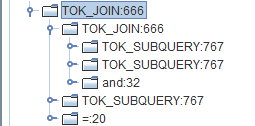
     insert into  tmp\_file   select \*\*\* from \*\*\*\* where  \*\*\*\*

1. 关于hql中的join



hql中的join的ast树这种嵌套的结构，a join b join c  join d 会翻译成：join (join (join(join a b) c) d) 这种嵌套的join形式

1. 对于join 操作



0  1 位置是两个表，2未知是on条件

对于多个join操作合并成left嵌套的操作

1. 关于hive的TOK\_QUERY语句

SELECT [/\*mapjoin(\*\*\*\*)\*/]

[ALL | DISTINCT] select\_expr, select\_expr, ...   
FROM table\_reference   
[WHERE where\_condition]   
[GROUP BY col\_list]   
[ CLUSTER BY col\_list | [DISTRIBUTE BY col\_list] [SORT BY col\_list] ]   
[LIMIT number]

转换以后就是  ：

FROM  table\_reference/tabjoin/tabunion

INSERT INTO   tabl[partspec] /local directory /directory

SELECT [/\*mapjoin(\*\*\*\*)\*/]

[ALL | DISTINCT] select\_expr, select\_expr, ...

[WHERE where\_condition]

[GROUP BY col\_list]

[ CLUSTER BY col\_list | [DISTRIBUTE BY col\_list] [SORT BY col\_list] ]  [ORDER BY col\_list] ]    
[LIMIT number]

包含的数据项是：from 、 insert 、select  、where 、 group by   、having、cluster、distribute 、sort by  、order by 、limit  这11种元素

insert overwrite directory '/home/work/data/tmp'

select

contract\_line\_id,st\_date,pdate,sum(alb\_cust\_id)

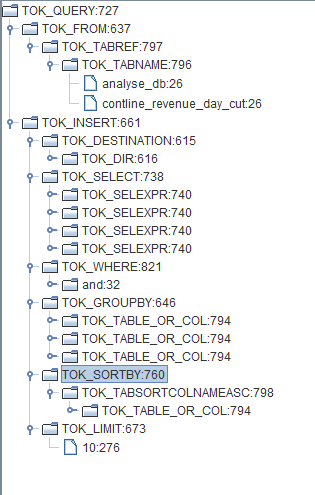
from analyse\_db.contline\_revenue\_day\_cut

where pdate='2013-11-12' and prodline\_id>10

group by contract\_line\_id,st\_date,pdate

sort by contract\_line\_id

limit 10



1. doPhase1中的case

        对于TOK\_QUERY的ast树进行遍历，一个tok\_query中的以上几个组成部分，对于子查询（所有的子查询都重写为一个新的tok\_query类型的ast树）等，都是递归再处理   。

       这对几种元素做的处理以及存储信息（主要是在QB和QBParseInfo中）：

         1） HiveParser.TOK\_FROM

      一层的sql ast中有且只有一个TOK\_FROM  类型的节点，但是TOK\_FROM 下面的可以是一个 子查询/union/table/join操作，在这个case语句中

        如果是table，则向QBParseInfo.QBParseInfo中put一个映射关系，key是table的别名，value是table的ast树

        如果是subquery，则向QB.aliasToSubq中设置一个key-value，key为子查询别名，value为该子查询的QBExpr

        如果是joi操作的话，则将join的left和right分别取出来，执行上面两个操作 ，两个中可能都有值或者是某一个中有两个值 。

        如果是lateview的话，\*\*\*\*\*\*\*\*\*\*

1. 阅读源代码的几个经验

        一定要单机debug

        对于太琐碎/不常用(例如ptf、lateview等)的分支可以先放弃

1. 解析中Clause的概念

       hive支持这样的语句

       from    tab\_from

       insert   tab\_in1   select \*\*\*\*

       insert   tab\_in2   select \*\*\*\*

       insert   tab\_in3   select \*\*\*\*

一个insert组就称为一个Clause ，其key  name一次为 insClause-0 ，insClause-1 ，insClause-2 ........

就算是正常insert \*\*\*   select \*\*\*\*   from  \*\*\*   ,也会给改写成上面的形式，只是只有一个clause ，name=insClause-0

1. TOK\_SELECTDI   （和 TOK\_SELECT 并列的）

        针对这种   select   distinct  a ,b,c  from   tab  (相当于select a,b,c from tab group by a,b,c  )

1. 针对于上面那种多个insert的操作

        将group  by  相同的找出来，在tabScan的自己点中添加自己的where过滤filter

1. OpParseContext

         每个operator的上下文，存储在全局的opParseCtx这个map中，key是相应的operator  ，主要是保存operator的输出Schema，RowResolver

1. 关于ExplainSemanticAnalyzer

其实是将ast树去掉   Explain节点，然后再

SemanticAnalyzerFactory.get(conf, (ASTNode) ast .getChild(0));得到真正的SemanticAnalyzer，然后再调用相应的analyzer方法，得到优化后的逻辑计划（记住此处不是物理执行计划），然后再打印

1. **待补充**