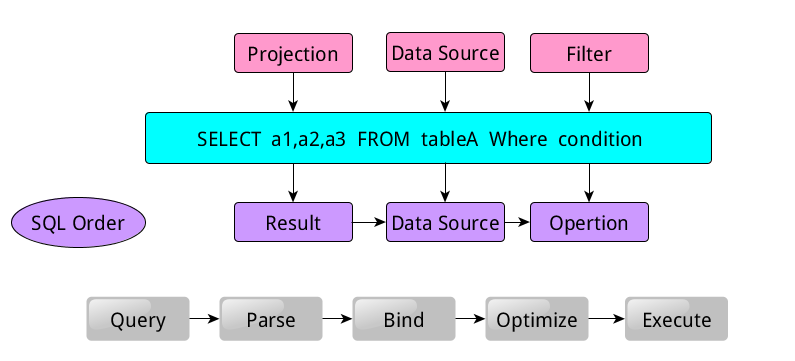
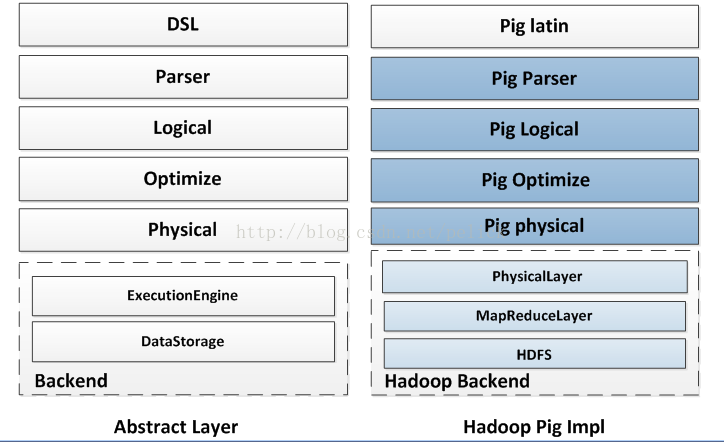
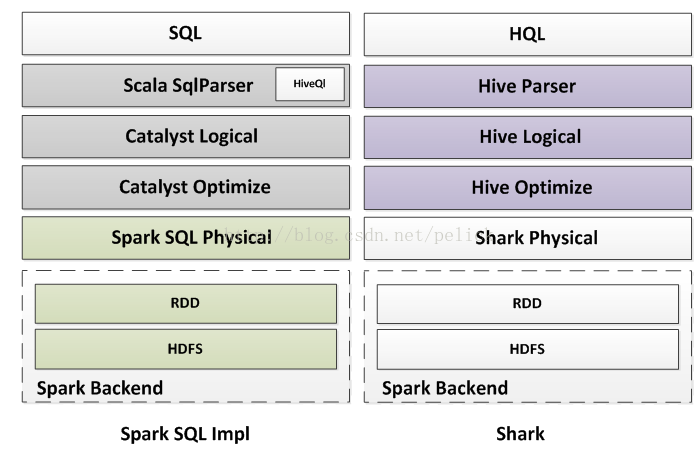
Spark SQL总结

# SQL执行过程



解析（select,where..）->绑定（列，表，试图…是否存在）->优化->执行





# Spark SQL

### 切入点SparkSession

val spark **=** SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

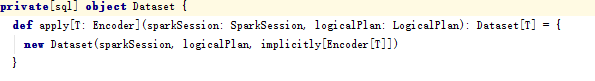
.getOrCreate()

SparkSession代替了之前版本的SQLContext和HiveContext，将其整合在了一起。作为了spark sql新的切入点。

### 两种操作方式

1. DataFrame/Dataset API

Spark2.0 将DataFrame 和 Dataset的api进行了统一（现在DataFrame就是Dataset[row]的别名）



Eg：employees

.join(dept,employees(“deptID”)===dept(“id”))

.where(employees(“gender”)===female)

.groupBy(dept(“id),dept(“name”))

.agg(count(“name”))

其中employees和dept都是Dataset，可以看到，Dataset存储的是结构化的数据，可以根据属性名来get值。

1. Query Language

通过sql语句来进行数据操作。

users.where(users(“age”)<21)

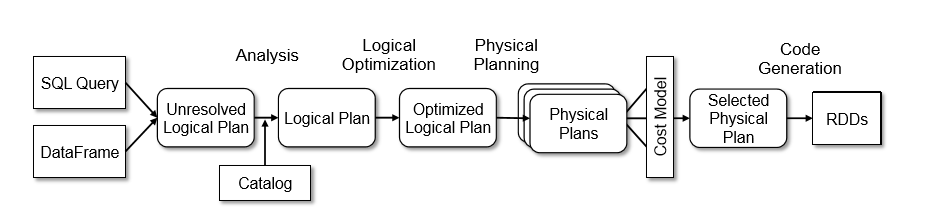
.createOrReplaceTempView(“young”)

Sparksession.sql(“SELECT COUNT(\*),AVG(age) FROM young”)

要使用sql语句查询，就必须先将dataset注册成临时表（视图）

这里要提到一点，Dataset API和SQL语句方式都是lazy的，两者都是先生成逻辑执行计划，然后优化在生成物理计划执行。

### Catalyst优化器



#### Expression

在SQL语句中，除了SELECT FROM等关键字以外，其他大部分元素都可以理解为Expression，比如SELECT sum(a), a，其中sum(a)和a都为Expression；

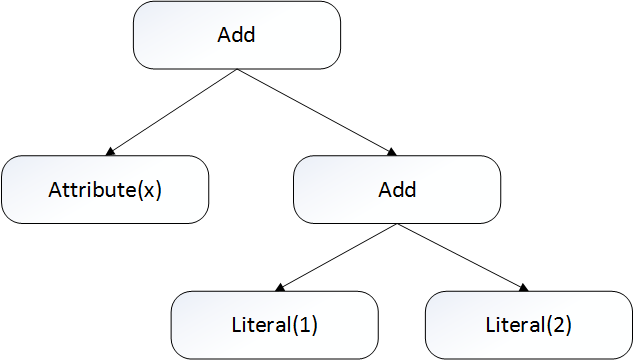
从SQL语句功能来说。Expression的功能可以划分为Projection，Predicate，Ordering，Aggregate；其中

Projection功能就是input数据进行加工，输出为InternalRow；

Predicate而是对输入的InternalRow进行加工输出为Bool；

Ordering和Aggregate则是针对Sortby/Groupby专用的两类表达式。Expression是一个Tree结构。

Eg：X+(1+2)



#### Attribute

Attribute直译为属性，在SQL中，可以简单理解为输入的Table中的字段，Attribute通过Name字段来进行命名。SQL语句通过Parse生成AST以后，SQL语句中的每个字段都会解析为UnresolvedAttribute。

比如SELECT a中的a就表示为UnresolvedAttribute("a")，还有一个特殊的UnresolvedAttribute，既为SQL语句中的＊，它表示为Star，属于UnresolvedAttribute类型的子类。

之后的Analysis过程会将unresolvedAttribute 变成resolvedAttribute

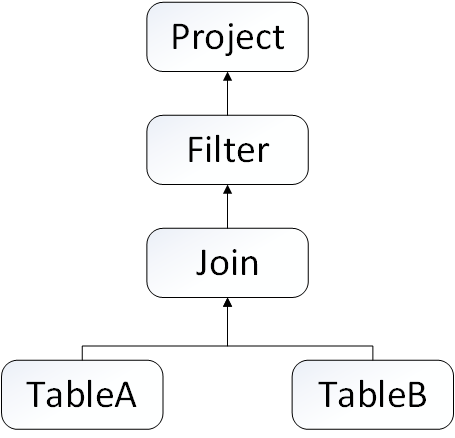
#### Plan

在SQL语句中，除了SELECT FROM等关键字以外，其他大部分元素都可以理解为Expression，单Expression只是一些Eval功能函数或者代码片段，需要一个东西来串联这些片段，这个东西就是Plan。

在Catalyst中，Plan是一个统称，它包括逻辑计划（logicalPlan）和物理计划（SparkPlan）

Plan(logicalPlan,physicalPlan)表现形式都是Tree，节点之间的关系可以理解为一种操作次序，比如Plan叶子节点表示从磁盘读取DB文件，而Root节点表示最终数据的输出；

Eg：Select project from tableA,tableB where filter



plan可以细分成三种类型的Node（逻辑计划和物理计划都有自己独自的三种节点）：

* UnaryNode 一元节点，即只有一个子节点。如Limit、Filter操作
* BinaryNode 二元节点，即有左右子节点的二叉节点。如Jion、Union操作
* LeafNode 叶子节点，没有子节点的节点。如ExistingRdd

#### TreeNode

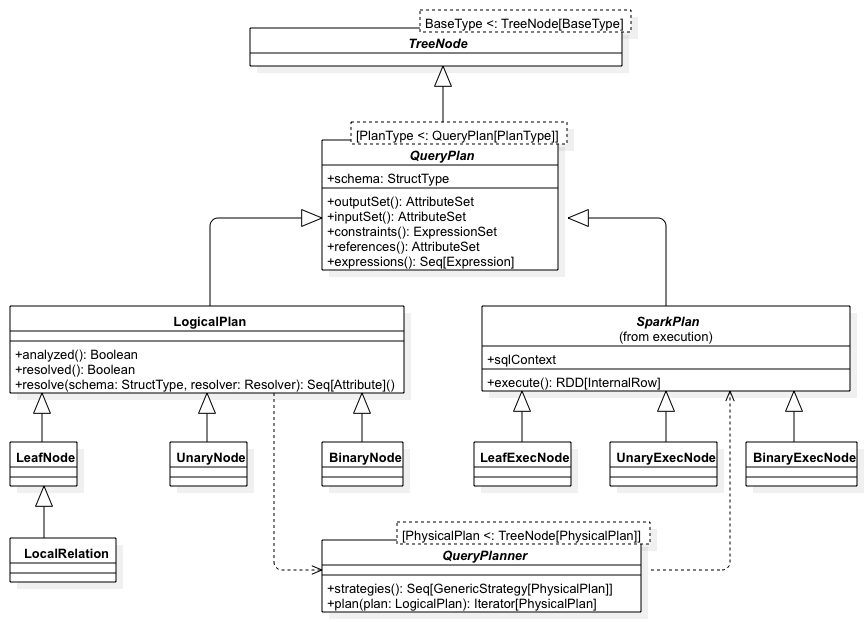
TreeNode是Catalyst执行计划表示的数据结构，是一个树结构，具备一些scala collection的操作能力和树遍历能力。这棵树一直在内存里维护，不会dump到磁盘以某种格式的文件存在，且无论在映射逻辑执行计划阶段还是优化逻辑执行计划阶段，树的修改是以替换已有节点的方式进行的。

TreeNode，内部带一个children: Seq[BaseType]表示孩子节点，具备foreach、map、collect等针对节点操作的方法，以及transformDown(默认，前序遍历)、transformUp这样的遍历树上节点，对匹配节点实施变化的方法。

提供UnaryNode,BinaryNode, LeafNode三种trait，即非叶子节点允许有一个或两个子节点。

TreeNode提供的是范型。

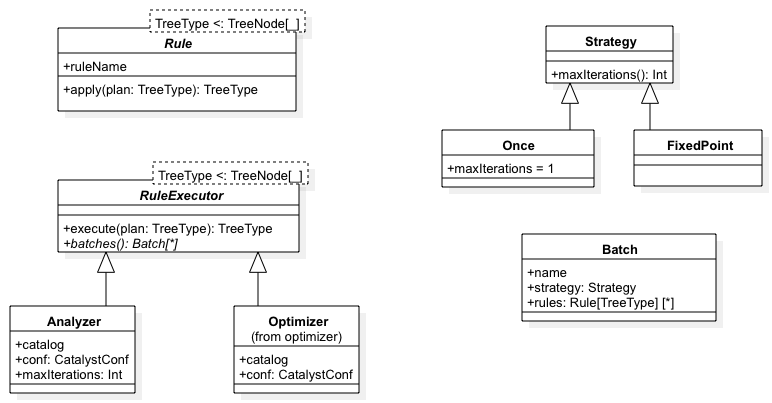
TreeNode有两个子类继承体系，QueryPlan和Expression。QueryPlan下面是逻辑（logicalPlan）和物理执行计划（sparkPlan）两个体系，前者在Catalyst里有详细实现，后者需要在系统自己实现。



#### Rules

顾名思义就是一种规则

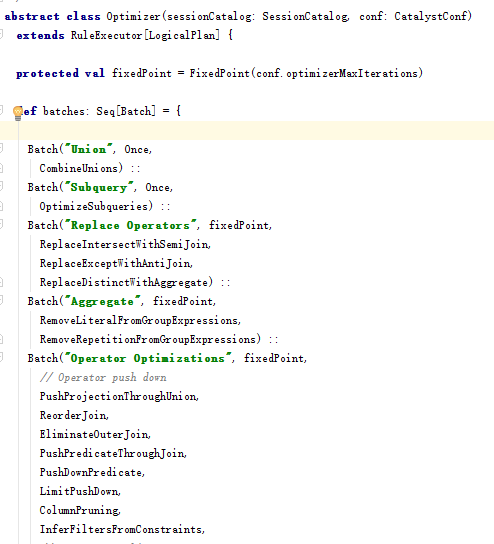
Rule是一个抽象类，拥有一个名字，默认为类名。具体的rule子类需要实现apply方法。



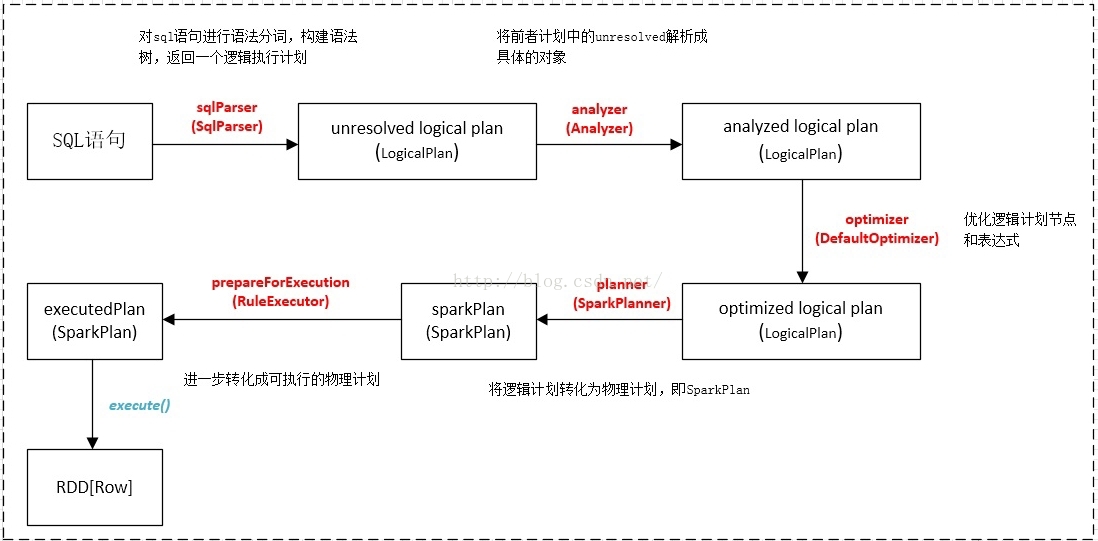
无论是对Expression进行操作还是对Plan进行操作，Catalyst将他们都抽象为Rule，它的apply函数通过传入一个TreeType类型的元素，并输出一个TreeType类型的元素。

另外可以将一组Rule组合为一个Batch，同时指定一个执行strategy，只执行一次或者是运行到树稳定为止（多次），

Catalyst在使用这些rule或batch时是将batch封装在RuleExecutor中，从而通过RuleExecutor将该组Rule的可执行接口提供给外部使用，比如Optimizer和Analyzer



#### Catalyst解析过程



1. SQL语句经过SqlParse解析（分词+语法规则）成一颗语法树（UnresolvedLogicalPlan）；
2. 使用analyzer结合catalog和functionRegistry进行绑定表的Schema和函数信息，生成resolvedLogicalPlan；
3. 使用optimizer对resolvedLogicalPlan进行优化，生成optimizedLogicalPlan；
4. 使用SparkPlanner将LogicalPlan转换成PhysicalPlan；
5. 使用prepareForExecution()将PhysicalPlan转换成可执行物理计划；
6. 使用execute()执行可执行PhysicalPlan；
7. 生成Dateset。

### 一个例子

#### SQL Text

select destinationURL,sum(adRevenue) as total

from uservisits

group by destinationURL

order by total desc

#### Parser

这个阶段主要进行sql语句的解析，将字符串转化成一个语法树，spark2.0中的catalyst采用了anltr语法分析器

================= Parsed Logical Plan ====================

'Sort ['total DESC], true

+- 'Aggregate ['destinationURL], ['destinationURL, 'sum('adRevenue) AS total#75]

+- 'UnresolvedRelation `uservisits`

================= Parsed Logical Plan ====================

可以看到这一步结束后，树形结构已经建立完毕，但是这些属性名称并不明确，我们将其命名为unresolveed logicalPlan

#### Analyzer

借助catalog类，它是一个记录表信息的类，专门提供给Analyzer用。这个阶段主要是进行：

在catalog中查找表名，对树中出现的属性（列名）进行判断和绑定，然后分配给它们一个唯一的id。这些处理策略都是通过一条条batch或rule来执行的

============ Analyzed Logical Plan ==============================

destinationURL: string, total: double

Sort [total#75 DESC], true

+- Aggregate [destinationURL#55], [destinationURL#55, sum(adRevenue#57) AS total#75]

+- SubqueryAlias uservisits

+- LogicalRDD [sourceIPAddr#54, destinationURL#55, visitDate#56L, adRevenue#57, UserAgent#58, cCode#59, lCode#60, sKeyword#61, avgTimeOnSite#62L]

============= Analyzed Logical Plan =============================

可以看到这一步结束后，unresolved属性都变成了resolved，我们将其称为resolved logicalPlan 或者analyzed logicalPlan

#### Optimizer

这一阶段主要是对logicalPlan进行优化，sql优化分为2种：

基于规则的优化

基于成本的优化（估算运行成本来选择代价最小计划，主要用在join中）

这一阶段我们使用的是基于规则的优化。

也是通过batch和rule来实现。Optimizer定义了大量的规则来对逻辑计划进行优化。

1. BooleanSimplification: 简化Boolean表达式
2. NullPropagation 对NULL常量参与表达式计算进行优化
3. ColumnPruning 字段剪枝
4. CombineLimits：Limit操作合并
5. PushDownPredicate 对于Filter操作，原则上它处于越底层越好，他可以显著减小后面计算的数据量。

<https://github.com/ColZer/DigAndBuried/blob/master/spark/spark-catalyst-optimizer.md>

===================== Optimized Logical Plan =====================

Sort [total#75 DESC], true

+- Aggregate [destinationURL#55], [destinationURL#55, sum(adRevenue#57) AS total#75]

+- Project [destinationURL#55, adRevenue#57]

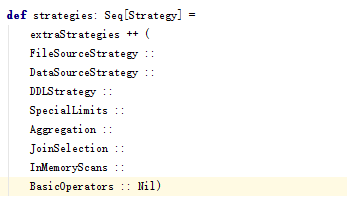
+- LogicalRDD [sourceIPAddr#54, destinationURL#55, visitDate#56L, adRevenue#57, UserAgent#58, cCode#59, lCode#60, sKeyword#61, avgTimeOnSite#62L]

===================== Optimized Logical Plan =====================

#### Planner

这一阶段将logicalPlan转化成physicalPlan。由QueryPlanner类处理，输入一个logicalPlan，返回多个physical Plan，然后从这多个physicalPlan中，选择最好的一个physical plan（但是这个还未实现，现阶段只是选择第一个计划执行）

这一部分的精髓是SparkStrategies，它定义了许多的策略



=======================Physical Plan ==========================

\*Sort [total#75 DESC], true, 0

+- Exchange rangepartitioning(total#75 DESC, 200)

+-\*HashAggregate(keys=[destinationURL#55],functions=[sum(adRevenue#57)], output=[destinationURL#55, total#75])

+- Exchange hashpartitioning(destinationURL#55, 200)

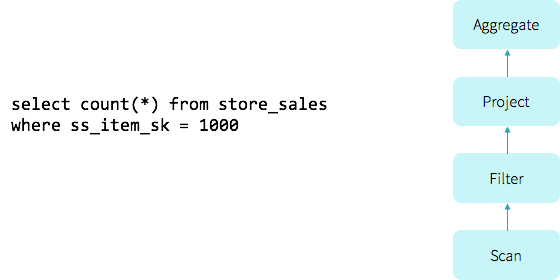
+-\*HashAggregate(keys=[destinationURL#55], unctions=[partial\_sum(adRevenue#57)], output=[destinationURL#55, sum#84])

+- \*Project [destinationURL#55, adRevenue#57]

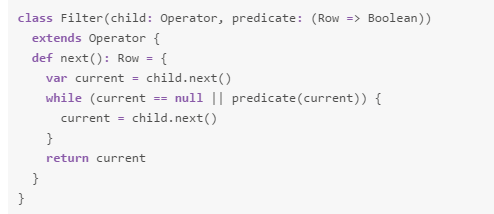
+-Scan ExistingRDD[sourceIPAddr#54,destinationURL#55,visitDate#56L,adRevenue#57,UserAgent#58,cCode#59,lCode#60,sKeyword#61,avgTimeOnSite#62L]

=======================Physical Plan ==========================

#### Code generation



为了计算这个查询，旧版本的Spark(1.x)会利用基于迭代模型的经典查询评估策略(通常被称为Volcano model)。在这个模型中，一个查询由多个算子(operators)组成，每个算子都提供了next()接口，该接口每次只返回一个元组(tuple)给嵌套树中的下一个算子。比如上面查询中的Filter算子大致可以翻译成下面的代码：



手写代码



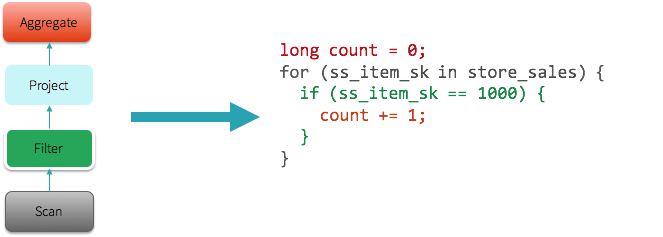
这样做实现看起来比较优雅和简洁，生成的代码也有一定的封装性，但是慢。

whole-stage code generation

在spark2.0中，采用了第二代Tungsten engine，它想通过在运行期间优化那些拖慢整个查询的代码放到一个单独的函数中，消除虚拟函数的调用以及利用cpu寄存器来存放那些中间数据。这种技术叫做whole-stage code generation。

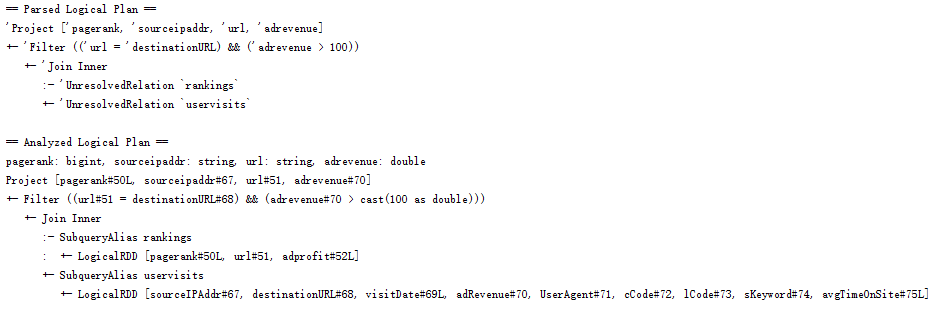
我们的目标是使用整段代码生成使得Spark计算引擎可以实现手写代码的性能，并且提供通用的功能。而不是在运行时依赖算子来处理数据，这些算子在运行时生成代码，如果可能的话将所有的查询片段组成到单个函数中，后面我们仅需要运行生成的代码。

比如对于上面的查询可以作为单个阶段，Spark可以产生以下的JVM字节码(这里展示的是Java代码)。复杂的查询将会产生多个阶段，这种情况下Spark将会产生多个不同的函数。



SQL Text一些测试

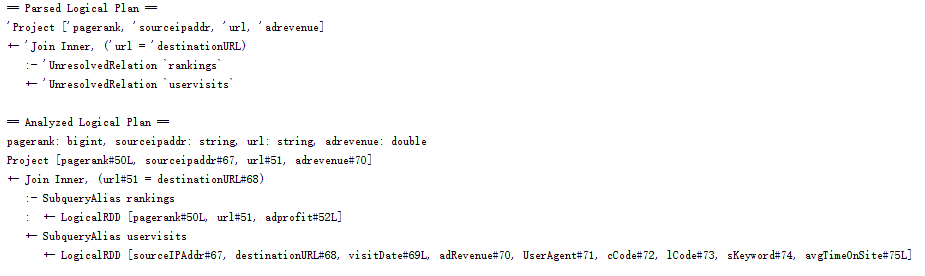
select pagerank,sourceipaddr,url,adrevenue  
from rankings,uservisits  
where url=destinationURL and adrevenue > 100

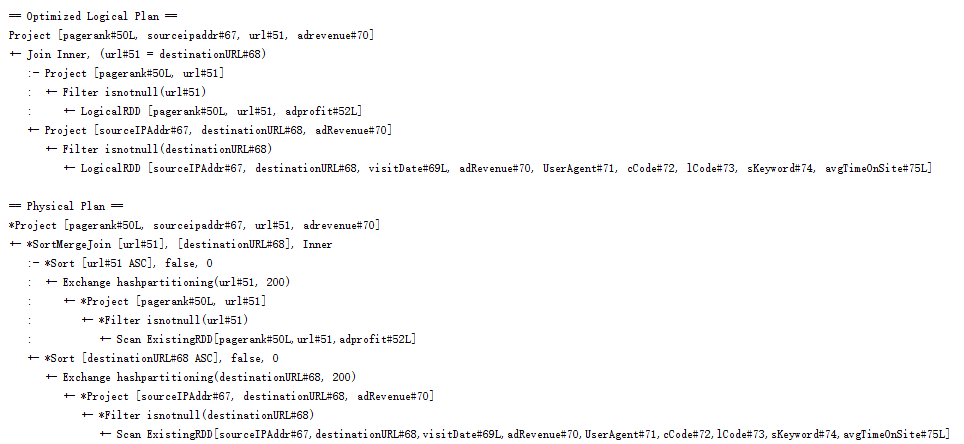




SQL Text

**select pagerank,sourceipaddr,url,adrevenue**  
**from rankings**  
**INNER JOIN uservisits**  
**ON url=destinationURL**





SQL Text（join顺序与where不对应时）

**Select \* from j1,j3,j2 where j2.id = j3.id and j1.id = j2.id**

