# Optimizer

Optimizer处理的是LogicalPlan对象

Optimizer的batches如下：  
**object** Optimizer **extends** RuleExecutor[LogicalPlan] {

**val** batches =

Batch("ConstantFolding", Once,

ConstantFolding, // 可静态分析的常量表达式

BooleanSimplification, // 布尔表达式提前短路

SimplifyFilters, // 简化过滤操作(false, true, null)

SimplifyCasts) :: // 简化转换(对象所属类已经是Cast目标类)

Batch("Filter Pushdown", Once,

CombineFilters, // 相邻(上下级)Filter操作合并

PushPredicateThroughProject, // 映射操作中的Filter谓词下推

PushPredicateThroughInnerJoin) :: Nil // inner join操作谓词下推

}

以上是4.1号的版本。

## ConstantFolding

把可以静态分析出结果的表达式替换成Literal表达式。

**object** ConstantFolding **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** q: LogicalPlan => q transformExpressionsDown {

// Skip redundant folding of literals.

**case** l: Literal => l

**case** e **if** e.foldable => Literal(e.apply(**null**), e.dataType)

}

}

}

Literal能处理的类型包括Int, Long, Double, Float, Byte, Short, String, Boolean, null。这些类型分别对应的是Catalyst框架的DataType，包括IntegerType, LongType, DoubleType, FloatType, ByteType, ShortType, StringType, BooleanType, NullType。

普通的Literal是不可变的，还有一个可变的MutalLiteral类，有update方法可以改变里面的value

## BooleanSimplification

提前短路可以短路的布尔表达式

**object** BooleanSimplification **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** q: LogicalPlan => q transformExpressionsUp {

**case** and @ And(left, right) =>

(left, right) **match** {

**case** (Literal(**true**, BooleanType), r) => r

**case** (l, Literal(**true**, BooleanType)) => l

**case** (Literal(**false**, BooleanType), \_) => Literal(**false**)

**case** (\_, Literal(**false**, BooleanType)) => Literal(**false**)

**case** (\_, \_) => and

}

**case** or @ Or(left, right) =>

(left, right) **match** {

**case** (Literal(**true**, BooleanType), \_) => Literal(**true**)

**case** (\_, Literal(**true**, BooleanType)) => Literal(**true**)

**case** (Literal(**false**, BooleanType), r) => r

**case** (l, Literal(**false**, BooleanType)) => l

**case** (\_, \_) => or

}

}

}

}

## SimplifyFilters

提前处理可以被判断的过滤操作

**object** SimplifyFilters **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** Filter(Literal(**true**, BooleanType), child) =>

child

**case** Filter(Literal(**null**, \_), child) =>

LocalRelation(child.output)

**case** Filter(Literal(**false**, BooleanType), child) =>

LocalRelation(child.output)

}

}

## SimplifyCasts

把已经是目标类的Cast表达式替换掉

**object** SimplifyCasts **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {

**case** Cast(e, dataType) **if** e.dataType == dataType => e

}

}

## CombineFilters

相邻都是过滤操作的话，把两个过滤操作合起来。相邻指的是上下两级。

**object** CombineFilters **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** ff @ Filter(fc, nf @ Filter(nc, grandChild)) => Filter(And(nc, fc), grandChild)

}

}

## PushPredicateThroughProject

把Project操作中的过滤操作下推。这一步里顺带做了别名转换的操作（认为开销不大的前提下）。

**object** PushPredicateThroughProject **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** filter @ Filter(condition, project @ Project(fields, grandChild)) =>

**val** sourceAliases = fields.collect { **case** a @ Alias(c, \_) =>

(a.toAttribute: Attribute) -> c

}.toMap // 把fields中的别名属性都取出来

project.copy(child = filter.copy( // 生成新的Filter操作

replaceAlias(condition, sourceAliases), // condition中有别名的替换掉

grandChild))

}

**def** replaceAlias(condition: Expression, sourceAliases: Map[Attribute, Expression]): Expression = {

condition transform {

**case** a: AttributeReference => sourceAliases.getOrElse(a, a)

}

}

}

## PushPredicateThroughInnerJoin

先找到Filter操作，若Filter操作里面是一次inner join，那么先把Filter条件和inner join条件先全部取出来，

然后把只涉及到左侧或右侧的过滤操作下推到join外部，把剩下来不能下推的条件放到join操作的condition里。

**object** PushPredicateThroughInnerJoin **extends** Rule[LogicalPlan] **with** PredicateHelper {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** f @ Filter(filterCondition, Join(left, right, Inner, joinCondition)) =>

// 这一步是把过滤条件和join条件里的condition都提取出来

**val** allConditions = splitConjunctivePredicates(filterCondition) ++

joinCondition.map(splitConjunctivePredicates).getOrElse(Nil)

// 把参考属性都属于右侧输出属性的condition挑选到rightCondition里

**val** (rightConditions, leftOrJoinConditions) =

allConditions.partition(\_.references subsetOf right.outputSet)

// 同理，把剩余condition里面，参考属性都属于左侧输出属性的condition挑选到

// leftCondition里，剩余的就属于joinCondition

**val** (leftConditions, joinConditions) =

leftOrJoinConditions.partition(\_.references subsetOf left.outputSet)

// 生成新的left和right：先把condition里的操作用AND折叠起来，然后将该折叠后的表达式和// 原始的left/right logical plan合起来生成新的Filter操作，即新的Filter logical plan

// 这样就做到了把过滤条件中的谓词下推到了left/right里，即本次inner join的“外部”

**val** newLeft = leftConditions.reduceLeftOption(And).map(Filter(\_, left)).getOrElse(left)

**val** newRight = rightConditions.reduceLeftOption(And).map(Filter(\_, right)).getOrElse(right)

Join(newLeft, newRight, Inner, joinConditions.reduceLeftOption(And))

}

}

Join操作(LogicalPlan的Binary)

**case** **class** Join(

left: LogicalPlan,

right: LogicalPlan,

joinType: JoinType,

condition: Option[Expression]) **extends** BinaryNode {

**def** references = condition.map(\_.references).getOrElse(Set.empty)

**def** output = left.output ++ right.output

}

Filter操作(LogicalPlan的Unary)

**case** **class** Filter(condition: Expression, child: LogicalPlan) **extends** UnaryNode {

**def** output = child.output

**def** references = condition.references

}

reduceLeftOption逻辑是这样的：

**def** reduceLeftOption[B >: A](op: (B, A) => B): Option[B] =

**if** (isEmpty) None **else** Some(reduceLeft(op))

reduceLeft(op)的结果是*op( op( ... op(x\_1, x\_2) ..., x\_{n-1}), x\_n)*

谓词助手这个trait，负责把And操作里的condition分离开，返回表达式Seq

**trait** PredicateHelper {

**def** splitConjunctivePredicates(condition: Expression): Seq[Expression] = condition **match** {

**case** And(cond1, cond2) => splitConjunctivePredicates(cond1) ++ splitConjunctivePredicates(cond2)

**case** other => other :: Nil

}

}

# Example

case class Person(name: String, age: Int)

case class Num(v1: Int, v2: Int)

## case one

**SELECT** people.age, num.v1, num.v2

**FROM**

people

**JOIN** num

**ON** people.age > 20 **and** num.v1 > 0

**WHERE** num.v2 < 50

== Query Plan ==

**Project** [age#1:1,v1#2:2,v2#3:3]

**CartesianProduct**

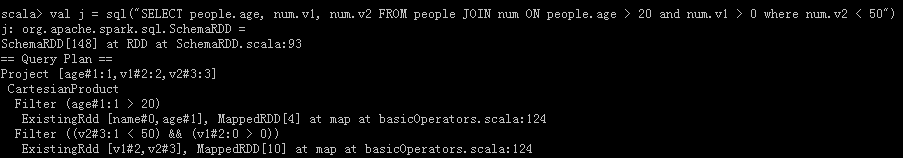
**Filter** (age#1:1 > 20)

**ExistingRdd** [name#0,age#1], MappedRDD[4] at map at basicOperators.scala:124

**Filter** ((v2#3:1 < 50) && (v1#2:0 > 0))

**ExistingRdd** [v1#2,v2#3], MappedRDD[10] at map at basicOperators.scala:124

分析：where条件 num.v2 < 50 下推到Join里



## case two

**SELECT** people.age, 1+2

**FROM**

people

**JOIN** num

**ON** people.name<>’abc’ **and** num.v1 > 0

**WHERE** num.v2 < 50

== Query Plan ==

**Project** [age#1:1,3 AS c1#14]

**CartesianProduct**

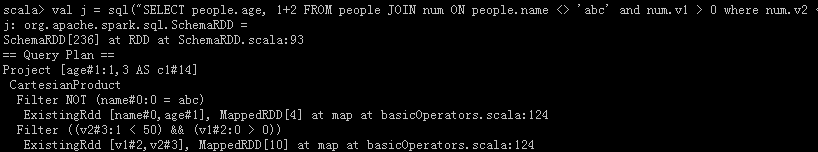
**Filter** **NOT** (name#0:0 = abc)

**ExistingRdd** [name#0,age#1], MappedRDD[4] at map at basicOperators.scala:124

**Filter** ((v2#3:1 < 50) && (v1#2:0 > 0))

**ExistingRdd** [v1#2,v2#3], MappedRDD[10] at map at basicOperators.scala:124

分析：1+2 被提前常量折叠，并被取了一个别名



## case three

**SELECT** people.age, 1+2

**FROM**

people

**JOIN** num

**ON** unknow.column<>’abc’ **and** num.v1 > 0

**WHERE** num.v2 < 50

== Query Plan ==

**Project** ['people.age,3 AS c1#13]

**Filter** (('num.v2 < 50) && NOT ('unknow.column = abc))

**CartesianProduct**

**ExistingRdd** [name#0,age#1], MappedRDD[4] at map at basicOperators.scala:124

**Filter** (v1#2:0 > 0)

**ExistingRdd** [v1#2,v2#3], MappedRDD[10] at map at basicOperators.scala:124

分析：疑惑为什么 num.v2 < 50 不下推

