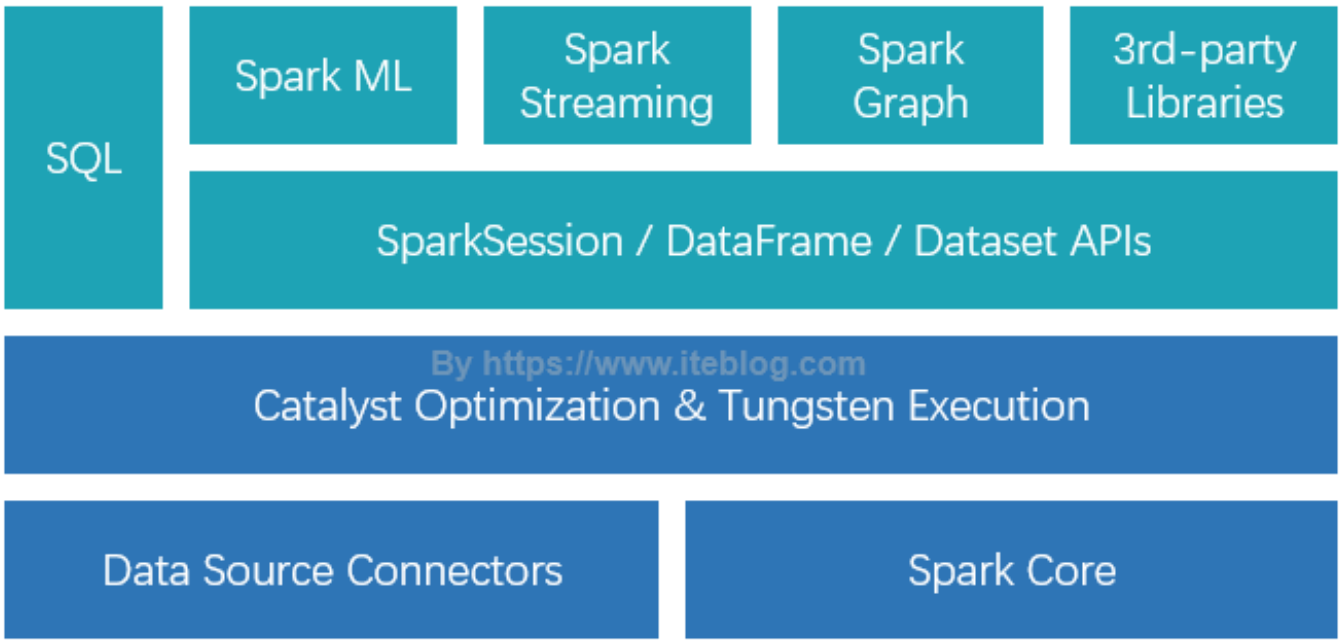
一条 SQL 在 Apache Spark 之旅（上）

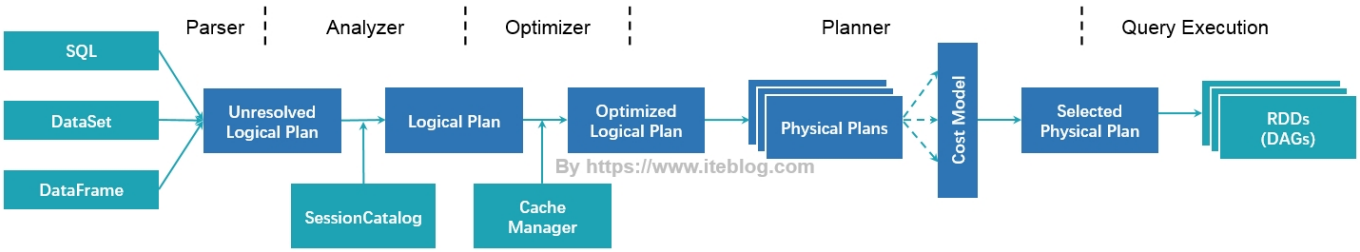
Spark SQL 是 Spark 众多组件中技术最复杂的组件之一，它同时支持 SQL 查询和 DataFrame DSL。通过引入了 SQL 的支持，大大降低了开发人员的学习和使用成本。目前，整个 SQL 、Spark ML、Spark Graph 以及 Structured Streaming 都是运行在 Catalyst Optimization & Tungsten Execution 之上的，如下图所示：



所以，正常的 SQL 执行先会经过 SQL Parser 解析 SQL，然后经过 Catalyst 优化器处理，最后到 Spark 执行。而 Catalyst 的过程又分为很多个过程，其中包括：

- Analysis：主要利用 Catalog 信息将 Unresolved Logical Plan 解析成 Analyzed logical plan；
- Logical Optimizations：利用一些 Rule（规则）将 Analyzed logical plan 解析成 Optimized Logical Plan；
- Physical Planning：前面的 logical plan 不能被 Spark 执行，而这个过程是把 logical plan 转换成多个 physical plans，然后利用代价模型（cost model）选择最佳的 physical plan；
- Code Generation：这个过程会把 SQL 查询生成 Java 字节码。

所以整个 SQL 的执行过程可以使用下图表示：



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

其中蓝色部分就是 Catalyst
优化器处理的部分，也是本文重点介绍的部分。下面我们以一条简单的 SQL 为例，从 High-level 角度介绍 一条 SQL 在 Spark 之旅。本文我们用到的 SQL 查询语句如下：

```
SELECT sum(v)
  FROM (
    SELECT
      t1.id,
  1 + 2 + t1.value AS v
  FROM t1 JOIN t2
    WHERE
  t1.id = t2.id AND
  t1.cid = 1 AND
  t1.did = t1.cid + 1 AND
    t2.id > 5) iteblog
```

SQL 解析阶段 - SparkSqlParser

为了能够在 Spark 中运行 SQL 查询，第一步肯定是需要解析这条 SQL。在 Spark 1.x 版本中，SQL 的解析有两种方法：

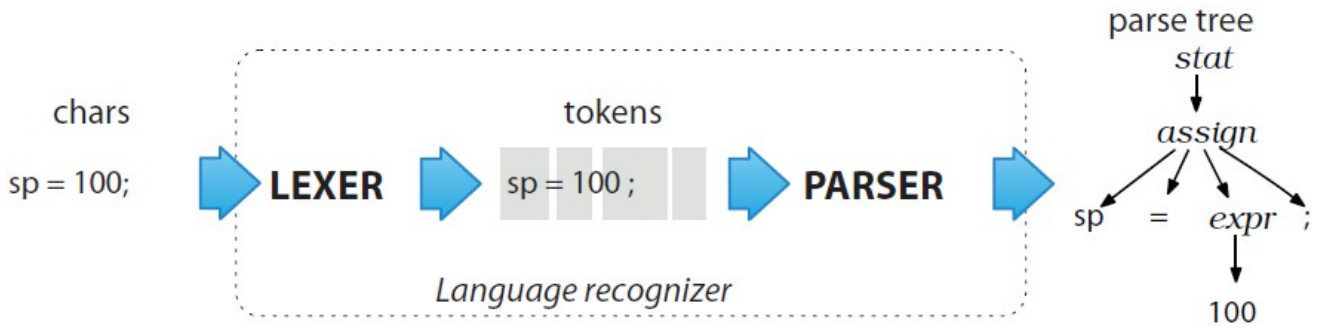
- 基于 Scala parser combinator 实现
- 基于 Hive 的 SQL 解析

可以通过 spark.sql.dialect 来设置。虽然 SQL 的解析引擎可以选择，但是这种方案有以下几个问题：Scala parser combinator 解析器有时候会给出错误信息，而且在定义语法中存在冲突不会发出警告；而 Hive SQL 解析引擎依赖于 Hive，这导致扩展性不好。

为了解决这个问题，从 Spark 2.0.0 版本开始引入了第三方语法解析器工具 ANTLR（详情参见 SPARK-12362），Antlr 是一款强大的语法生成器工具，可用于读取、处理、执行和翻译结构化的文本或二进制文件，是当前 Java 语言中使用最为广泛的语法生成器工具，我们常见的大数据 SQL 解析都用到了这个工具，包括 Hive、Cassandra、Phoenix、Pig 以及 presto

等。目前最新版本的 Spark 使用的是 ANTLR4，通过这个对 SQL 进行词法分析并构建语法树。

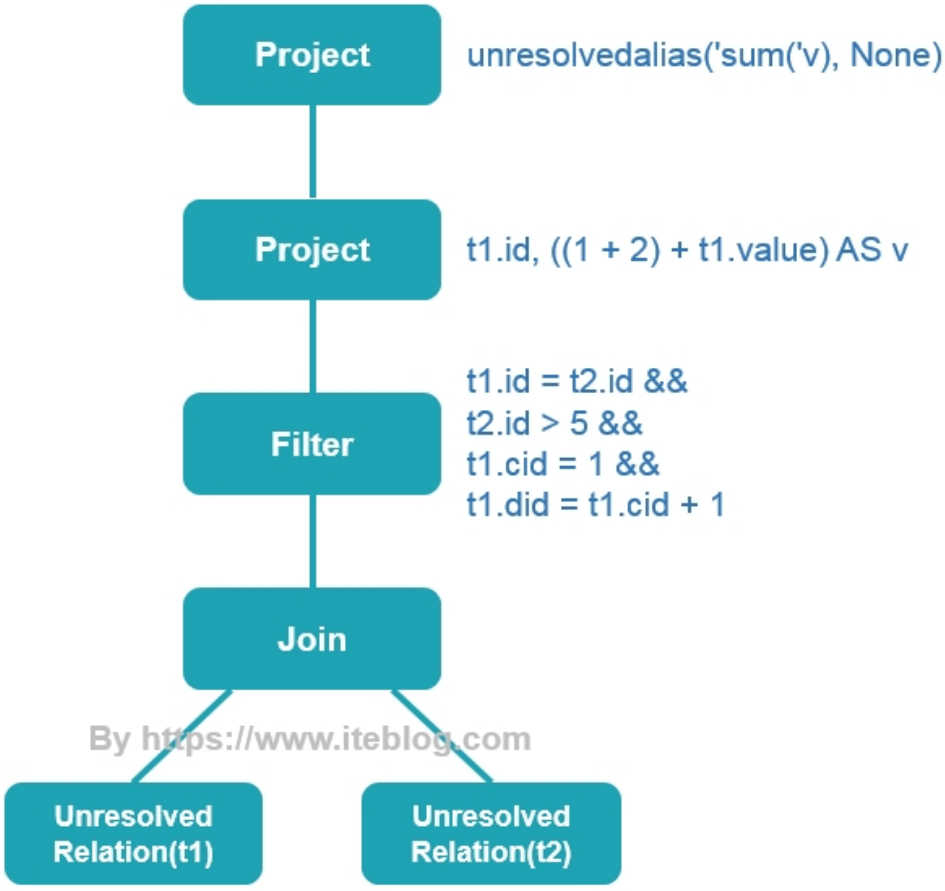
具体的，Spark 基于 presto 的语法文件定义了 Spark SQL 语法文件 SqlBase.g4（路径 spark-2.4.3₩sql₩catalyst₩src₩main₩antlr4₩org₩apache₩spark₩sql₩catalyst₩parser₩SqlBase.g4），这个文件定义了 Spark SQL 支持的 SQL
语法。如果我们需要自定义新的语法，需要在这个文件定义好相关语法。然后使用 ANTLR4 对 SqlBase.g4 文件自动解析生成几个 Java 类，其中就包含重要的词法分析器 SqlBaseLexer.java 和语法分析器SqlBaseParser.java。运行上面的 SQL 会使用 SqlBaseLexer
来解析关键词以及各种标识符等；然后使用 SqlBaseParser
来构建语法树。整个过程就类似于下图。



生成语法树之后，使用 AstBuilder 将语法树转换成 LogicalPlan，这个 LogicalPlan 也被称为 Unresolved LogicalPlan。解析后的逻辑计划如下：

```
== Parsed Logical Plan ==
'Project [unresolvedalias('sum('v), None)]
+- 'SubqueryAlias `iteblog`
   +- 'Project ['t1.id, ((1 + 2) + 't1.value) AS v#16]
      +- 'Filter ((('t1.id = 't2.id) && ('t1.cid = 1)) && (('t1.did = ('t1.cid + 1)) && ('t2.id > 5)))
         +- 'Join Inner
            :- 'UnresolvedRelation `t1`
            +- 'UnresolvedRelation `t2`
```

图片表示如下：



Unresolved LogicalPlan 是从下往上看的，t1 和 t2 两张表被生成了 UnresolvedRelation，过滤的条件、选择的列以及聚合字段都知道了，SQL 之旅的第一个过程就算完成了。

绑定逻辑计划阶段 - Analyzer

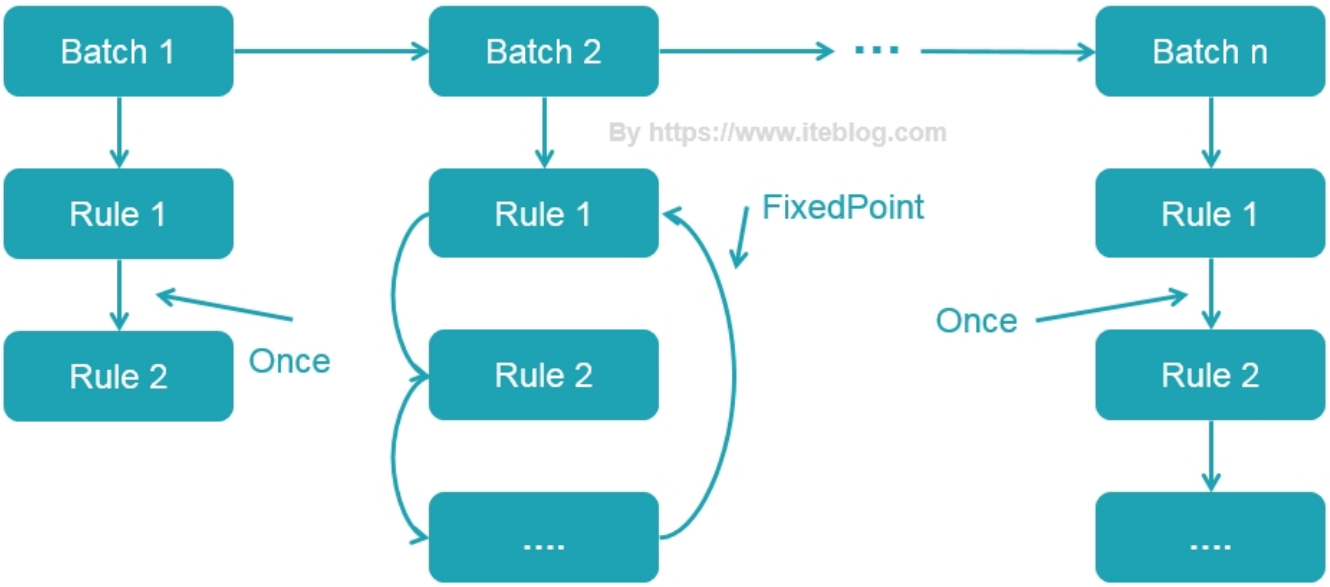
在 SQL 解析阶段生成了 Unresolved LogicalPlan，从上图可以看出逻辑算子树中包含了 UnresolvedRelation 和 unresolvedalias 等对象。Unresolved LogicalPlan 仅仅是一种数据结构，不包含任何数据信息，比如不知道数据源、数据类型，不同的列来自于哪张表等。Analyzer 阶段会使用事先定义好的 Rule 以及 SessionCatalog 等信息对 Unresolved LogicalPlan 进行 transform。SessionCatalog 主要用于各种函数资源信息和元数据信息（数据库、数据表、数据视图、数据分区与函数等）的统一管理。而Rule 是定义在 Analyzer 里面的，如下具体如下：

```
lazy val batches: Seq[Batch] = Seq(
    Batch("Hints", fixedPoint,
      new ResolveHints.ResolveBroadcastHints(conf),
      ResolveHints.ResolveCoalesceHints,
      ResolveHints.RemoveAllHints),
    Batch("Simple Sanity Check", Once,
      LookupFunctions),
```

```
Batch("Substitution", fixedPoint,
  CTESubstitution,
  WindowsSubstitution,
  EliminateUnions,
  new SubstituteUnresolvedOrdinals(conf)),
Batch("Resolution", fixedPoint,
  ResolveTableValuedFunctions ::    //解析表的函数
  ResolveRelations ::       //解析表或视图
  ResolveReferences ::       //解析列
  ResolveCreateNamedStruct ::
  ResolveDeserializer ::      //解析反序列化操作类
  ResolveNewInstance ::
  ResolveUpCast ::        //解析类型转换
  ResolveGroupingAnalytics ::
  ResolvePivot ::
  ResolveOrdinalInOrderByAndGroupBy ::
  ResolveAggAliasInGroupBy ::
  ResolveMissingReferences ::
  ExtractGenerator ::
  ResolveGenerate ::
  ResolveFunctions ::       //解析函数
  ResolveAliases ::        //解析表别名
  ResolveSubquery ::        //解析子查询
  ResolveSubqueryColumnAliases ::
  ResolveWindowOrder ::
  ResolveWindowFrame ::
  ResolveNaturalAndUsingJoin ::
  ResolveOutputRelation ::
  ExtractWindowExpressions ::
  GlobalAggregates ::
  ResolveAggregateFunctions ::
  TimeWindowing ::
  ResolveInlineTables(conf) ::
  ResolveHigherOrderFunctions(catalog) ::
  ResolveLambdaVariables(conf) ::
  ResolveTimeZone(conf) ::
  ResolveRandomSeed ::
  TypeCoercion.typeCoercionRules(conf) ++
  extendedResolutionRules : _*),
Batch("Post-Hoc Resolution", Once, postHocResolutionRules: _*),
Batch("View", Once,
  AliasViewChild(conf)),
Batch("Nondeterministic", Once,
  PullOutNondeterministic),
Batch("UDF", Once,
  HandleNullInputsForUDF),
```

```
    Batch("FixNullability", Once,
      FixNullability),
    Batch("Subquery", Once,
      UpdateOuterReferences),
    Batch("Cleanup", fixedPoint,
      CleanupAliases)
)
```

从上面代码可以看出，多个性质类似的 Rule 组成一个 Batch，比如上面名为 Hints 的 Batch就是由很多个 Hints Rule 组成；而多个 Batch 构成一个 batches。这些 batches 会由 RuleExecutor 执行，先按一个一个 Batch 顺序执行，然后对 Batch 里面的每个 Rule 顺序执行。每个 Batch 会之心一次（Once）或多次（FixedPoint，由 spark.sql.optimizer.maxIterations 参数决定），执行过程如下：

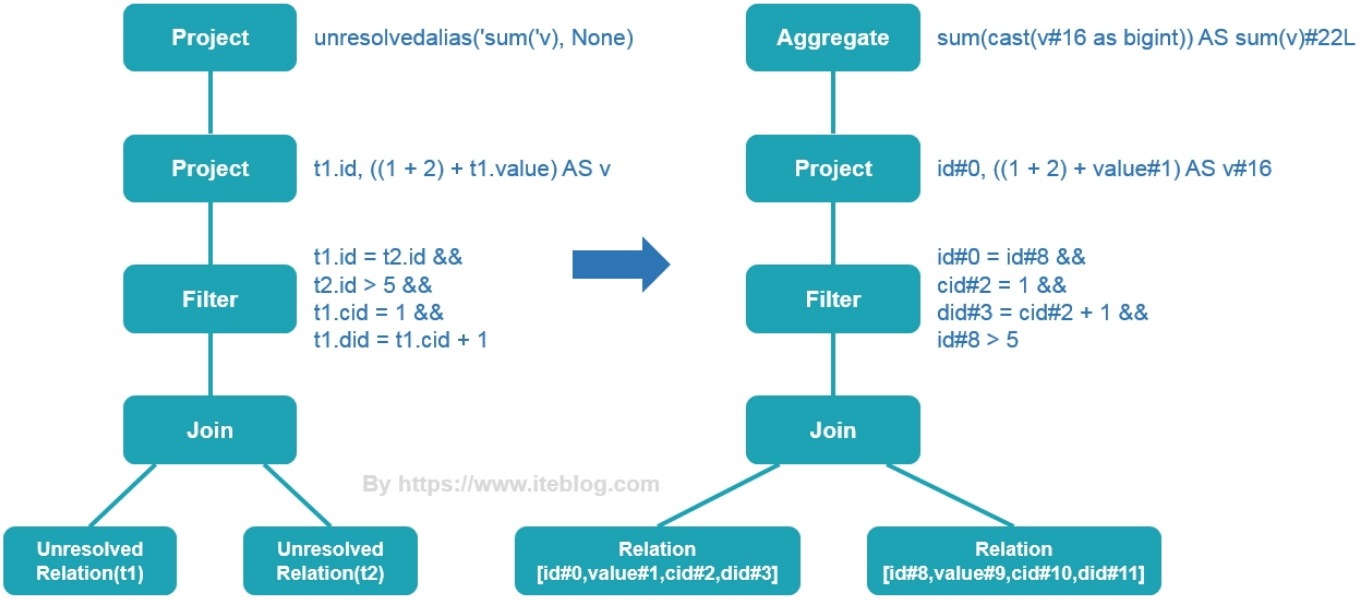


所以上面的 SQL 经过这个阶段生成的 Analyzed Logical Plan 如下：

```
== Analyzed Logical Plan ==
sum(v): bigint
Aggregate [sum(cast(v#16 as bigint)) AS sum(v)#22L]
+- SubqueryAlias `iteblog`
   +- Project [id#0, ((1 + 2) + value#1) AS v#16]
      +- Filter (((id#0 = id#8) && (cid#2 = 1)) && ((did#3 = (cid#2 + 1)) && (id#8 > 5)))
         +- Join Inner
            :- SubqueryAlias `t1`
            :  +- Relation[id#0,value#1,cid#2,did#3] csv
```

```
+- SubqueryAlias `t2`
  +- Relation[id#8,value#9,cid#10,did#11] csv
```

从上面的结果可以看出，t1 和 t2 表已经解析成带有 id、value、cid 以及 did 四个列的表，其中这个表的数据源来自于 csv 文件。而且每个列的位置和数据类型已经确定了，sum 被解析成 Aggregate 函数了。下面是从 Unresolved LogicalPlan 转换到 Analyzed Logical Plan 对比图。



到这里， Analyzed LogicalPlan 就完全生成了。由于篇幅的原因，剩余的 SQL 处理我将在下一篇文章进行介绍，包括逻辑计划优化、代码生成等东西，敬请关注。