# Deep Dive into Spark SQL's Catalyst Optimizer

April 13, 2015 (https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html) | by Michael Armbrust (https://databricks.com/blog/author/michael), Yin Huai (https://databricks.com/blog/author/yin-huai), Cheng Liang (https://databricks.com/blog/author/cheng-liang), Reynold Xin (https://databricks.com/blog/author/rxin) and Matei Zaharia (https://databricks.com/blog/author/matei) in **ENGINEERING BLOG (HTTPS://DATABRICKS.COM/BLOG/CATEGORY/ENGINEERING)**

(https://twitter.com/home?status=https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)
(https://www.linkedin.com/shareArticle?mini=true&url=https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html&title=Deep Dive into Spark SQL's Catalyst Optimizer&summary=&source=)
(https://www.facebook.com/sharer/sharer.php?u=https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)

Spark SQL is one of the newest and most technically involved components of Spark. It powers both SQL queries and the new DataFrame API (https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html). At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching (http://docs.scala-lang.org/tutorials/tour/pattern-matching.html) and quasiquotes (http://docs.scala-lang.org/overviews/quasiquotes/intro.html)) in a novel way to build an extensible query optimizer.

We recently published a paper (http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf) on Spark SQL that will appear in SIGMOD 2015 (http://www.sigmod2015.org/) (co-authored with Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, and Ali Ghodsi). In this blog post we are republishing a section in the paper that explains the internals of the Catalyst optimizer for broader consumption.

To implement Spark SQL, we designed a new extensible optimizer, Catalyst, based on functional programming constructs in Scala. Catalyst's extensible design had two purposes. First, we wanted to make it easy to add new optimization techniques and features to Spark SQL, especially for the purpose of tackling various problems we were seeing with big data (e.g., semistructured data and advanced analytics). Second, we wanted to enable external developers to extend the optimizer — for example, by adding data source specific rules that can push filtering or aggregation into external storage systems, or support for new data types. Catalyst supports both rule-based and cost-based optimization.

At its core, Catalyst contains a general library for representing trees and applying rules to manipulate them. On top of this framework, we have built libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, quasiquotes (http://docs.scala-lang.org/overviews/quasiquotes/intro.html), that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.

## Trees

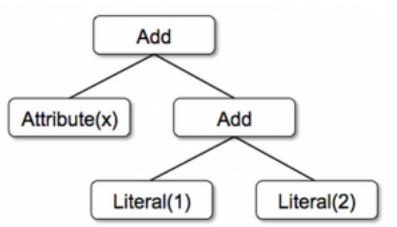
The main data type in Catalyst is a tree composed of node objects. Each node has a node type and zero or more children. New node types are defined in Scala as subclasses of the TreeNode class. These objects are immutable and can be manipulated using functional transformations, as discussed in the next subsection.

As a simple example, suppose we have the following three node classes for a very simple expression language:

- `Literal(value: Int)`: a constant value
- `Attribute(name: String)`: an attribute from an input row, e.g.,"x"
- `Add(left: TreeNode, right: TreeNode)`: sum of two expressions.

These classes can be used to build up trees; for example, the tree for the expression `x+(1+2)`, would be represented in Scala code as follows:

```
Add(Attribute(x), Add(Literal(1), Literal(2)))
```




(https://databricks.com/wp-content/uploads/2015/04/Screen-Shot-2015-04-12-at-8.11.11-AM.png)

## Rules

Trees can be manipulated using rules, which are functions from a tree to another tree. While a rule can run arbitrary code on its input tree (given that this tree is just a Scala object), the most common approach is to use a set of pattern matching functions that find and replace subtrees with a specific structure.

Pattern matching is a feature of many functional languages that allows extracting values from potentially nested structures of algebraic data types. In Catalyst, trees offer a transform method that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result. For example, we could implement a rule that folds Add operations between constants as follows:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

Applying this to the tree for `x+(1+2)` would yield the new tree `x+3`. The `case` keyword here is Scala's standard pattern matching syntax, and can be used to match on the type of an object as well as give names to extracted values (`c1` and `c2` here).

The pattern matching expression that is passed to transform is a partial function, meaning that it only needs to match to a subset of all possible input trees. Catalyst will tests which parts of a tree a given rule applies to, automatically skipping over and descending into subtrees that do not match. This ability means that rules only need to reason about the trees where a given optimization applies and not those that do not match. Thus, rules do not need to be modified as new types of operators are added to the system.

Rules (and Scala pattern matching in general) can match multiple patterns in the same transform call, making it very concise to implement multiple transformations at once:

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
  case Add(left, Literal(0)) => left
  case Add(Literal(0), right) => right
}
```

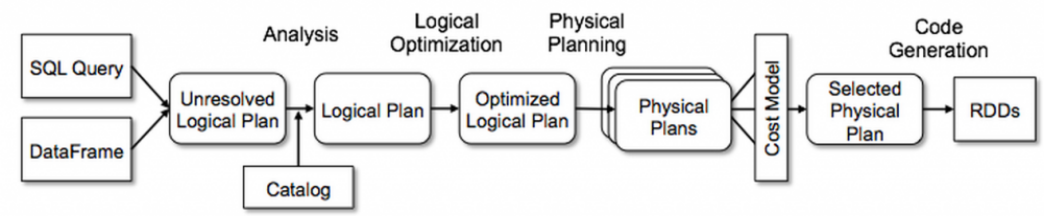
In practice, rules may need to execute multiple times to fully transform a tree. Catalyst groups rules into batches, and executes each batch until it reaches a fixed point, that is, until the tree stops changing after applying its rules. Running rules to fixed point means that each rule can be simple and self-contained, and yet still eventually have larger global effects on a tree. In the example above, repeated application would constant-fold larger trees, such as `(x+0)+(3+3)`. As another example, a first batch might analyze an expression to assign types to all of the attributes, while a second batch might use these types to do constant folding. After each batch, developers can also run sanity checks on the new tree (e.g., to see that all attributes were assigned types), often also written via recursive matching.

Finally, rule conditions and their bodies can contain arbitrary Scala code. This gives Catalyst more power than domain specific languages for optimizers, while keeping it concise for simple rules.

In our experience, functional transformations on immutable trees make the whole optimizer very easy to reason about and debug. They also enable parallelization in the optimizer, although we do not yet exploit this.

## Using Catalyst in Spark SQL

We use Catalyst's general tree transformation framework in four phases, as shown below: (1) analyzing a logical plan to resolve references, (2) logical plan optimization, (3) physical planning, and (4) code generation to compile parts of the query to Java bytecode. In the physical planning phase, Catalyst may generate multiple plans and compare them based on cost. All other phases are purely rule-based. Each phase uses different types of tree nodes; Catalyst includes libraries of nodes for expressions, data types, and logical and physical operators. We now describe each of these phases.




(https://databricks.com/wp-content/uploads/2015/04/Screen-Shot-2015-04-12-at-8.41.26-AM.png)

## Analysis

Spark SQL begins with a relation to be computed, either from an abstract syntax tree (AST) returned by a SQL parser, or from a DataFrame object constructed using the API. In both cases, the relation may contain unresolved attribute references or relations: for example, in the SQL query `SELECT col FROM sales`, the type of col, or even whether it is a valid column name, is not known until we look up the table sales. An attribute is called unresolved if we do not know its type or have not matched it to an input table (or an alias). Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources to resolve these attributes. It starts by building an "unresolved logical plan" tree with unbound attributes and data types, then applies rules that do the following:

- Looking up relations by name from the catalog.
- Mapping named attributes, such as col, to the input provided given operator's children.
- Determining which attributes refer to the same value to give them a unique ID (which later allows optimization of expressions such as `col = col`).
- Propagating and coercing types through expressions: for example, we cannot know the return type of `1 + col` until we have resolved col and possibly casted its subexpressions to a compatible types.

In total, the rules for the analyzer are about 1000 lines of code (https://github.com/apache/spark/blob/fedbfc7074dd6d38dc5301d66d1ca097bc2a21e0/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/analysis/Analyzer.scala).

## Logical Optimizations

The logical optimization phase applies standard rule-based optimizations to the logical plan. (Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.) These include constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules. In general, we have found it extremely simple to add rules for a wide variety of situations. For example, when we added the fixed-precision DECIMAL type to Spark SQL, we wanted to optimize aggregations such as sums and averages on DECIMALs with small precisions; it took 12 lines of code to write a rule that finds such decimals in SUM and AVG expressions, and casts them to unscaled 64-bit LONGs, does the aggregation on that, then converts the result back. A simplified version of this rule that only optimizes SUM expressions is reproduced below:

```
object DecimalAggregates extends Rule[LogicalPlan] {

  /** Maximum number of decimal digits in a Long */

  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {

    plan transformAllExpressions {

      case Sum(e @ DecimalType.Expression(prec, scale))

          if prec + 10 <= MAX_LONG_DIGITS =>

        MakeDecimal(Sum(UnscaledValue(e)), prec + 10, scale) }

}
```

As another example, a 12-line rule optimizes LIKE expressions with simple regular expressions into String.startsWith or String.contains calls. The freedom to use arbitrary Scala code in rules made these kinds of optimizations, which go beyond pattern-matching the structure of a subtree, easy to express.

In total, the logical optimization rules are 800 lines of code (https://github.com/apache/spark/blob/fedbfc7074dd6d38dc5301d66d1ca097bc2a21e0/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala).

## Physical Planning

In the physical planning phase, Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine. It then selects a plan using a cost model. At the moment, cost-based optimization is only used to select join algorithms: for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark. The framework supports broader use of cost-based optimization, however, as costs can be estimated recursively for a whole tree using a rule. We thus intend to implement richer cost-based optimization in the future.

The physical planner also performs rule-based physical optimizations, such as pipelining projections or filters into one Spark map operation. In addition, it can push operations from the logical plan into data sources that support predicate or projection pushdown. We will describe the API for these data sources in a later section.

In total, the physical planning rules are about 500 lines of code (https://github.com/apache/spark/blob/fedbfc7074dd6d38dc5301d66d1ca097bc2a21e0/sql/core/src/main/scala/org/apache/spark/sql/execution/SparkStrategies.scala).

## Code Generation

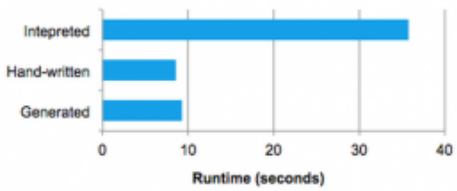
The final phase of query optimization involves generating Java bytecode to run on each machine. Because Spark SQL often operates on in-memory datasets, where processing is CPU-bound, we wanted to support code generation to speed up execution. Nonetheless, code generation engines are often complicated to build, amounting essentially to a compiler. Catalyst relies on a special feature of the Scala language, quasiquotes, to make code generation simpler. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. We use Catalyst to transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

As a simple example, consider the Add, Attribute and Literal tree nodes introduced in Section 4.2, which allowed us to write expressions such as `(x+y)+1`. Without code generation, such expressions would have to be interpreted for each row of data, by walking down a tree of Add, Attribute and Literal nodes. This introduces large amounts of branches and virtual function calls that slow down execution. With code generation, we can write a function to translate a specific expression tree to a Scala AST as follows:

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) => q"${compile(left)} + ${compile(right)}"
}
```

The strings beginning with `q` are quasiquotes, meaning that although they look like strings, they are parsed by the Scala compiler at compile time and represent ASTs for the code within. Quasiquotes can have variables or other ASTs spliced into them, indicated using `$` notation. For example, `Literal(1)` would become the Scala AST for 1, while `Attribute("x")` becomes `row.get("x")`. In the end, a tree like `Add(Literal(1), Attribute("x"))` becomes an AST for a Scala expression like `1+row.get("x")`.

Quasiquotes are type-checked at compile time to ensure that only appropriate ASTs or literals are substituted in, making them significantly more useable than string concatenation, and they result directly in a Scala AST instead of running the Scala parser at runtime. Moreover, they are highly composable, as the code generation rule for each node does not need to know how the trees returned by its children are constructed. Finally, the resulting code is further optimized by the Scala compiler in case there are expression-level optimizations that Catalyst missed. The following figure shows that quasiquotes let us generate code with performance similar to hand-tuned programs.




We have found quasiquotes very straightforward to use for code generation, and we observed that even new contributors to Spark SQL could quickly add rules for new types of expressions. Quasiquotes also work well with our goal of running on native Java objects: when accessing fields from these objects, we can code-generate a direct access to the required field, instead of having to copy the object into a Spark SQL Row and use the Row's accessor methods. Finally, it was straightforward to combine code-generated evalua- tion with interpreted evaluation for expressions we do not yet generate code for, since the Scala code we compile can directly call into our expression interpreter.

In total, Catalyst's code generator is about 700 lines of code (https://github.com/apache/spark/blob/fedbfc7074dd6d38dc5301d66d1ca097bc2a21e0/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/expressions/codegen/CodeGenerator.scala).

This blog post covered the internals of Spark SQL's Catalyst optimizer. It's novel, simple design has enabled the Spark community to rapidly prototype, implement, and extend the engine. You can read through rest of the paper here (http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf). If you are attending SIGMOD this year, please drop by our session!

You can also find more information about Spark SQL from the following:

- Spark SQL and DataFrame Programming Guide (http://spark.apache.org/docs/latest/sql-programming-guide.html) from Apache Spark
- Data Source API in Spark (http://www.slideshare.net/databricks/yin-huai-20150325meetupwithdemos) presentation by Yin Huai
- Introducing DataFrames in Spark for Large Scale Data Science (http://www.slideshare.net/databricks/introducing-dataframes-in-spark-for-large-scale-data-science) by Reynold Xin