Catalyst Query Optimization Framework

- About this Document
- Overview
- Architecture
  - TreeNode Library
  - Logical Plan Representation
  - Expressions
    - Standard Expressions
    - Named Expressions
  - Rules
    - Example Rules
      - Pushing down filters
      - Splitting Predicates
  - Planning & Execution Strategies

About this Document

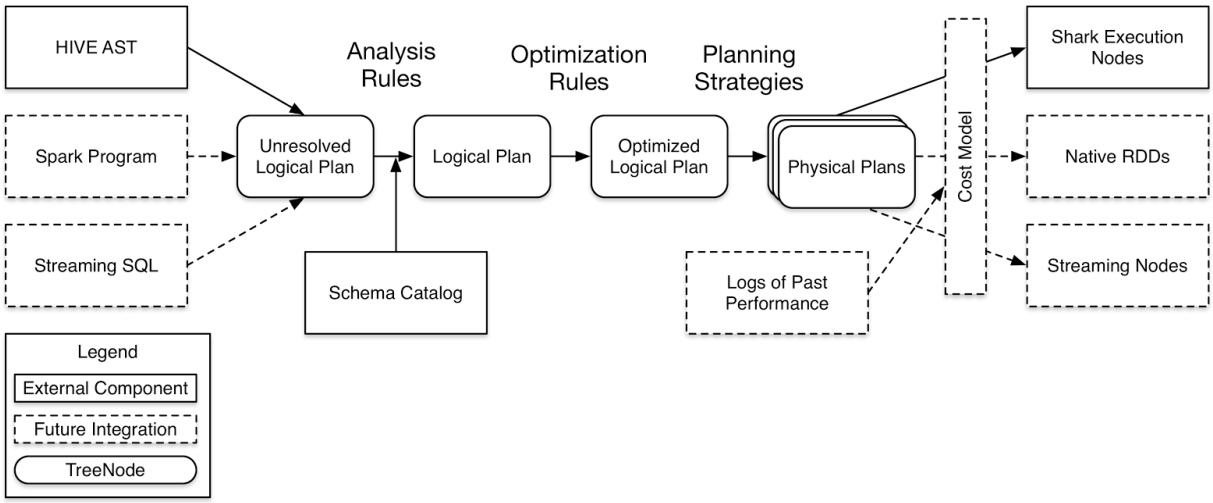
This document describes an initial design for catalyst, a framework for performing optimizations on graphs of relational dataflow operators. This framework will be initially be used by Spark SQL. Moving forward, the framework should provide both the ability to rapidly and concisely write new optimizations for Spark SQL and other dataflow engines, as well as easily reason about their correctness.

Overview

Catalyst is intended to serve two major purposes. First, it provides a toolbox for describing and manipulating trees of data flow operators. Second, it should provide a library of common relational constructs and rewrites that can be extended to support a specific workload / execution system.

Github: https://github.com/marmbrus/catalyst

Architecture



TreeNode Library

A library for easily manipulating trees of operators. Operators that extend TreeNode are granted the following interface:

- Scala collection like methods (foreach, map, flatMap, collect, etc)
- transform - accepts a partial function that is used to generate a new tree. When the partial function can be applied to a given tree segment, that segment is replaced with the result. After attempting to apply the partial function to a given node, the transform function recursively attempts to apply the function to that node's children.
- debugging support - pretty printing, easy splicing of trees, etc.

Logical Plan Representation

Represents the high level structure of the dataflow. Intermediate operations should be as generic (implementation agnostic) as possible. Leaf operators can be written to interface with many different data sources.

Each node in a logical query plan presents the following interface:

- output schema - the set of attributes and their types produced by this operator
- references - a list of attributes that are required by this operator

Expressions

The expressions library contains an extensible set of common expression classes each of which present the following interface:

- data type
- nullability

The expression library should both make it easy to identify certain patterns, for example, expressions that can be constant folded. It should also hide the details of naming / scoping

by providing an easy mechanism for determining which child of a given operator produces a given value.

Standard Expressions

A standard library of expressions (e.g., add, subtract), aggregates (e.g., SUM, COUNT), and other computations (e.g. UDFs). Each type expression is capable of determining its output schema as a function of its children's output schema.

Named Expressions

Some expression are named and thus can be referenced by other operators in the dataflow graph. The two types of named expressions are AttributeReferences and Aliases. AttributeReferences refer to parts of the input tuple for a given operator and form the leaves of all expression trees. Aliases assign a name to intermediate computations. For example, in the SQL statement "SELECT a+b AS c FROM ...", the expressions 'a' and 'b' would be represented by AttributeReferences and 'c' would be represented by an Alias.

During analysis, all named expressions are assigned a globally unique expression id, which can be used for equality comparisons. While the original names are kept around for debugging purposes, they should never be used to check attribute equality, as plan transformations can result in the introduction of naming ambiguity. For example, consider a plan that contains subqueries, both of which are reading from the same table. If an optimization removes the subqueries, scoping information would be destroyed, eliminating the ability to reason about which subquery produced a given attribute.

The expression ids can be constructed in such a way that it is possible to reason about the provenance of attributes through operations such as set operations (e.g., UNION) or outer joins, while still preserving the correct semantics regarding nullability. The details of this mechanism, however, are out of the scope for of document.

Rules

The bulk of the work done by the optimizer is expressed as Rules, which take in a query plan and produce a query plan. Catalyst will also provide a RuleExecutor class, which takes collections of rules and applies them to a given query plan until fixed point is reached. In addition to automating the application of rules, the RuleExecutor also provides support for debugging by optionally displaying transformations and ensuring plan invariants are satisfied (see Checks).

Example Rules

Pushing down filters

Below is the code for a rule that moves a filter beneath a projection if the filter references only attributes that are not produced by by the projection itself.

```
1 object FilterPushdown extends LogicalRule {
```

```
2   def apply(plan: LogicalPlan): LogicalPlan = plan transform {
3     case f @ Filter(_, p @ Project(_, projectChild))
4       if(f.references subsetOf projectChild.output) =>
5         p.copy(child = f.copy(child = projectChild))
6 }
```

Specifically, the case statement (Line 3) locates any instances in the given query plan where a filter is present on top of a projection. The guard (Line 4) ensures the that transformation only occurs when the filter can be evaluated using only the output of the original child of the project. If the above conditions are met, the order of the operators is reversed using the copy constructor of the two nodes (Line 5).

Splitting Predicates

Next, consider a rule that takes a single filter containing multiple conjunctive predicates (i.e. predicates connected by an AND), and breaks it into two filters. Simple rules such as this one can be combined with other rules that manipulate filters (e.g. the above rule) to result in more complicated emergent optimizations.

```
object SplitConjunctivePredicates extends LogicalRule {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case Filter(And(c1, c1), child) =>
      Filter(c1, Filter(c2, child))
  }
}
```

Planning & Execution Strategies

Execution strategies are objects that can generate a list of physical plans for a segment of a given logical plan. For example, a developer might create a strategy that identifies a set of filters on top of a join that could be executed by a single HashJoin operation. The planning framework is responsible linking these fragments together to produce full execution plans. The set of plans can then be explored within the given planning budget, choosing the cheapest physical plan for execution. Currently this choses only a single plan based on a heuristic ordering of possible planning strategies.