Catalyst

A Functional Query Optimization Framework

Michael Armbrust

amplab databricks™ SHARK

What is Query Optimization?

SQL is a *declarative* language:

Queries express ***what*** data to retrieve,

not how to retrieve it.

The database is free to pick the ‘best’ execution strategy through a process known as optimization

Naïve Query Planning

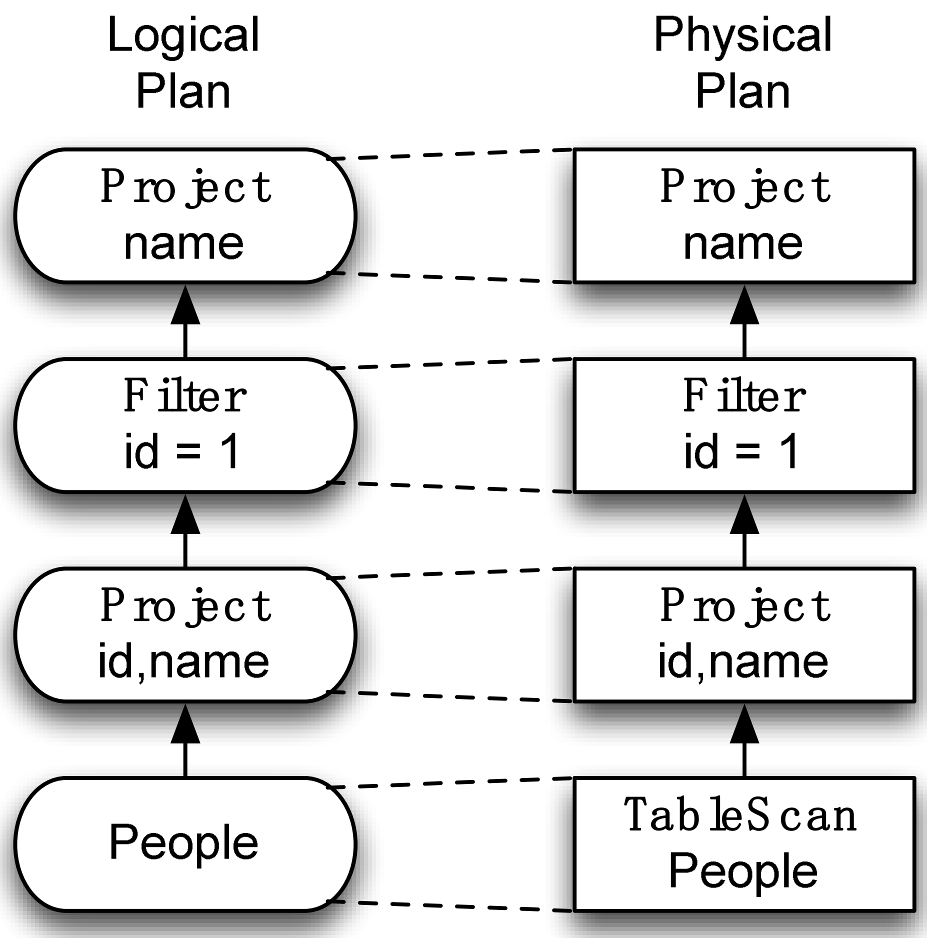
SELECT name

FROM (

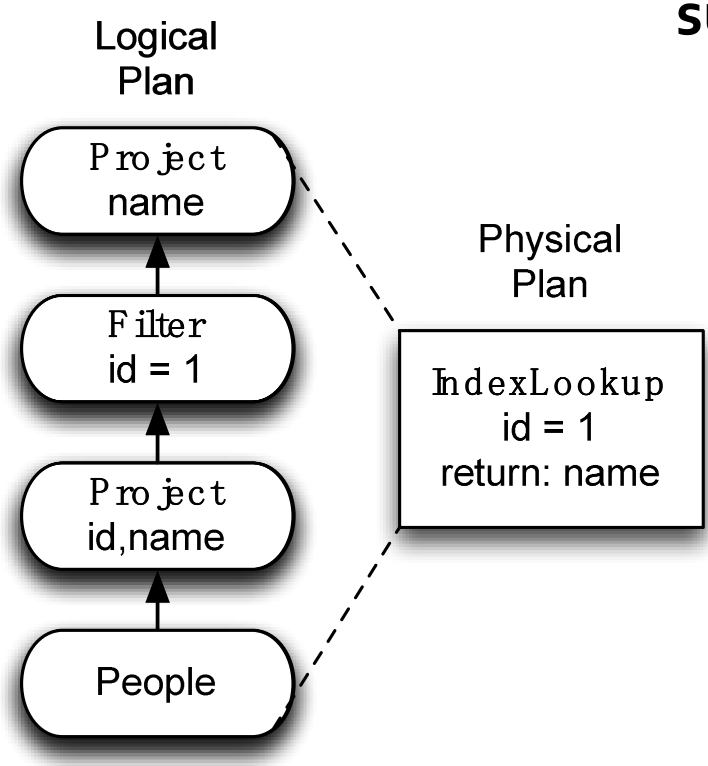
  SELECT id, name

  FROM People) p

WHERE p.id = 1



Optimized Execution

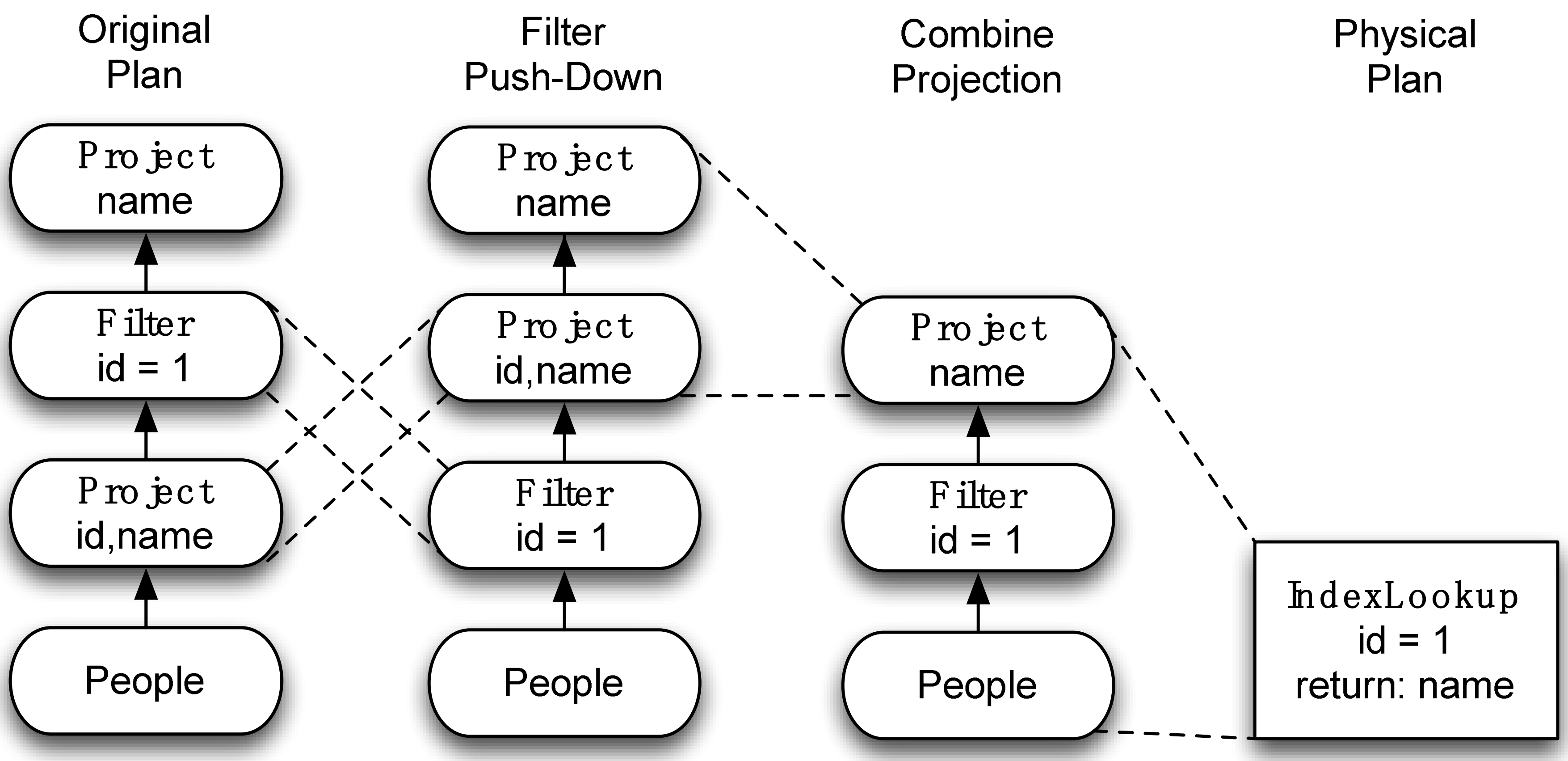


Writing imperative code to optimize such patterns generally is hard.

Instead write simple rules:

- Each rule makes one small change
- Run rules many rules together to fixed point.

Optimizing with Rules



Prior Work: Optimizer Generators

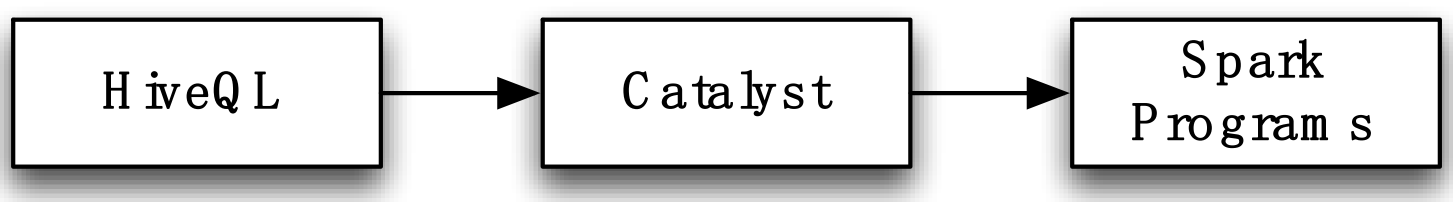
Volcano / Cascades:

- Create a custom language for expressing rules that rewrite trees of relational operators.
- Build a compiler that generates executable code for these rules.

Cons: Developers need to learn this custom language. Language might not be powerful enough.

Introducing Catalyst

Idea: Use high-level language features from a modern functional language (scala) to build an optimizer generator.



TreeNode Library

Easily transformable trees of operators

- Standard collection functionality (foreach, map, etc)
- Transform function – recursive modification of trees that match a specified pattern
- Debugging support

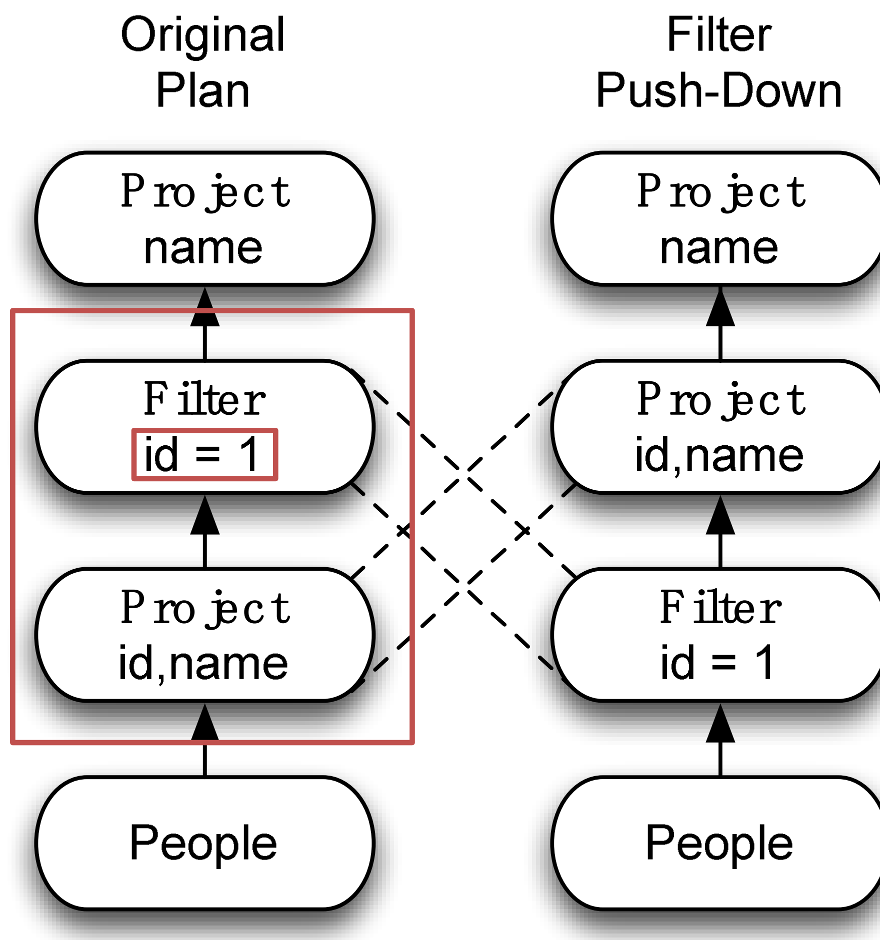
Tree Transformations

Expressed as PartialFunction[TreeType,TreeType] (i.e. a function that can be applied to some subset of all trees and that returns a new tree.)

1. If the function **does apply** to a given operator, that operator is **replaced** with the result.
2. When the function **does not apply** to a given operator, that operator is **left unchanged**.
3. The transformation is also **applied recursively** to the operators children.

Writing Rules as Tree Transformations

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.



Writing Rules as Tree Transformations

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Writing Rules as Tree Transformations

Tree

Partial Function

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Writing Rules as Tree Transformations

Find Filter on Project

```
val newPlan = queryPlan transform {
  case f @ Filter(_, p @ Project(_, grandChild))
    if(f.references subsetOf grandChild.output) =>
  p.copy(child = f.copy(child = grandChild)
}
```

Writing Rules as Tree Transformations

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Check that the filter can be evaluated without the result of the project.

Writing Rules as Tree Transformations

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

If so, switch the order.

Writing Rules as Tree Transformations

Pattern Matching

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Writing Rules as Tree Transformations

Collections library

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Writing Rules as Tree Transformations

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

Copy Constructors

Prototype Status

Prototype Shark without using the HIVE optimizer.

- 1 Developer, 1 month
- Able to plan SELECT, WHERE, GROUP BY, ORDER BY, COUNT, SUM, AVERAGE, COUNT DISTINCT.
  - Catalyst plans global aggregates better than current Shark (Using Spark Accumulator variables)
- ~700 Lines of code for planning + execution using Spark
- Working on integrating BlinkDB (approximate queries using sampling).

Conclusion

Catalyst is a framework for optimizing trees of relational operators:

- TreeNode library adds collection / transformation methods to tree nodes.
- Library for representing common logical relational operators and expressions.
- Machinery for executing batches of transformations to fixed point / enumerating possible physical plans for a given logical plan

Questions?

Expression Library

- Contains common expressions (+, -, etc.) and aggregates (SUM, COUNT, etc)
- Operators can reason functionally about data type and nullability.
- Assigns GUIDS to all attributes in a plan to allow reasoning about where values are produced even after transformations have destroyed scoping information.

Cost Models: Selecting the Cheapest Plan

- Many possible physical plans for most queries.
- Optimizer needs to decide which to use.
- Cost Model – Estimate cost of execution based on data statistics.

TableScan
People

$$|People|$$

IndexScan
name = 'michael'

$$\frac{|People|}{\#UniqueValuesInColumnName}$$

IndexLookup
id = 1
return: name

$$1$$