Dive Into Catalyst

QCon Beijing 2015

Cheng Lian <lian@databricks.com>

databricks™

"In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation."

Ada Lovelace, 1843

What Is Catalyst?

Catalyst is a functional, extensible query optimizer used by Spark SQL.

- Leverages advanced FP language (Scala) features
- Contains a library for representing trees and applying rules on them

Trees

Tree is the main data structure used in Catalyst

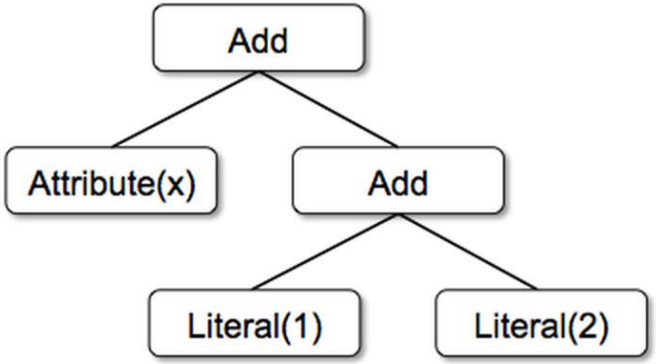
- A tree is composed of node objects
- A node has a node type and zero or more children
- Node types are defined in Scala as subclasses of the `TreeNode` class

Trees

Examples

- `Literal(value: Int)`
- `Attribute(name: String)`
- `Add(left: TreeNode, right: TreeNode)`

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



Rules

Rules are functions that transform trees

- Typically functional
- Leverage pattern matching
- Used together with
  - `TreeNode.transform` (synonym of `transformDown`)
  - `TreeNode.transformDown` (pre-order traversal)
  - `TreeNode.transformUp` (post-order traversal)

Rules

Examples

- Simple constant folding

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

Rules

Examples

- Simple constant folding

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

Pattern

Rules

Examples

- Simple constant folding

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

Transformation

Rules

Examples

- Simple constant folding

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

Rule

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

- Yeah, it’s not a typo, just wanna avoid saying dirty words all the time during the talk :^)

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

- Inspired by brainfuck optimization strategies by Mats Linander

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

- Goals

  Illustrate the power and conciseness of the Catalyst optimizer

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

- Goals

  As short as possible so that I can squeeze it into a single talk (292 loc)

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

- Non-goal

  Build a high performance practical compiler

Brainsuck

Brainsuck is an optimizing compiler of the Brainfuck programming language

- Contains
  - Parser, IR, and interpreter of the language
  - A simplified version of the Catalyst library, consists of the `TreeNode` class and the rule execution engine
  - A set of optimization rules

```scala
object MergeMoves extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transformUp {
    case Move(n, Move(m, next)) => if (n + m == 0) next else Move(n + m, next)
  }
}

object MergeAdds extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transformUp {
    case Add(n, Add (m, next)) => if (n + m == 0) next else Add(n + m, next)
  }
}

object Clears extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transformUp {
    case Loop(Add(_, Halt), next) => Clear(next)
  }
}

object Scans extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transformUp {
    case Loop(Move(n, Halt), next) => Scan(n, next)
  }
}

object MultisAndCopies extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transform {
    case Loop(Add(-1, MoveAddPairs(seq, offset, Move(n, Halt))), next) if n == -offset =>
      seq.foldRight(Clear(next): Instruction) {
        case ((distance, 1), code) => Copy(distance, code)
        case ((distance, increment), code) => Multi(distance, increment, code)
      }
  }
}

object MoveAddPairs {
  type ResultType = (List[(Int, Int)], Int, Instruction)

  def unapply(tree: Instruction): Option[ResultType] = {
    def loop(tree: Instruction, offset: Int): Option[ResultType] = tree match {
      case Move(n, Add(m, inner)) =>
        loop(inner, offset + n).map { case (seq, finalOffset, next) =>
          ((offset + n, m) :: seq, finalOffset, next)
        }
      case inner => Some((Nil, offset, inner))
    }
    loop(tree, 0)
  }
}
```

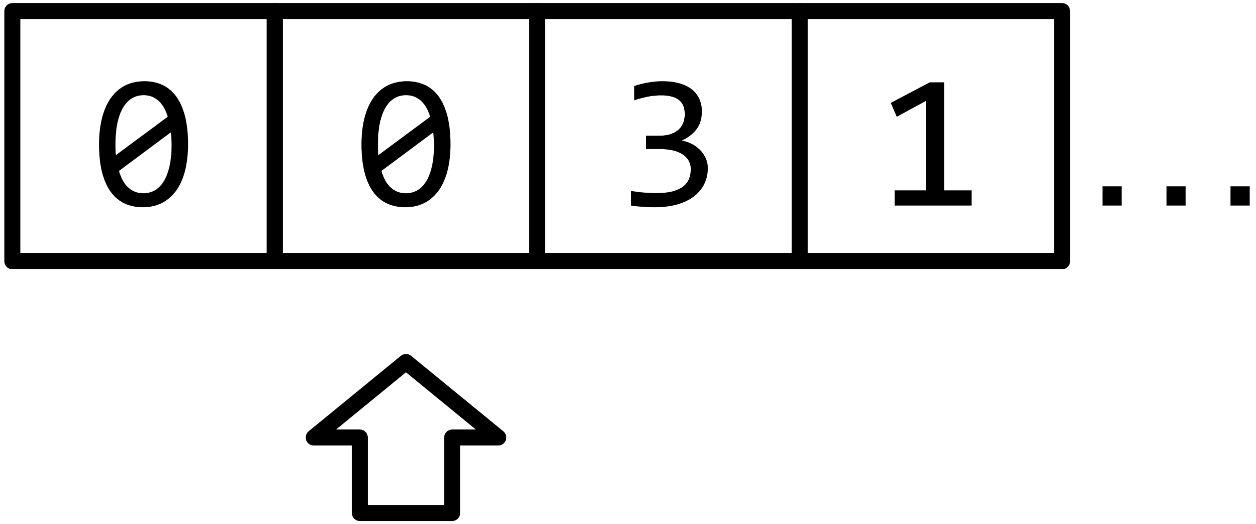
Optimization rules in Brainsuck

Optimization rules in bfoptimization by Mats Linander

databricks™

Machine Model

A dead simple simulation of Turing machine



Instruction Set

Instruction	Meaning
`(Initial)`	`char array[infinitely large size] = {0};` `char *ptr = array;`
`>`	`++ptr; // Move right`
`<`	`--ptr; // Move left`
`+`	`++*ptr; // Increment`
`-`	`--*ptr; // Decrement`
`.`	`putchar(*ptr); // Put a char`
`,`	`*ptr = getchar(); // Read a char`
`[`	`while (*ptr) { // Loop until *ptr is 0`
`]`	`} // End of the loop`

databricks™

Sample Program

```
++++++++[>++++[>++>+++>+++>
+<<<<-]>+>+>->>+[<]<-]>>.>
---.+++++++..+++.>>.<-.<.++
+.------.--------.>>+.>++.
```

databricks

Sample Program

```
++++++++[>++++[>++>+++>+++>
+<<<<-]>+>+>->>+[<]<-]>>.>
---.+++++++..+++.>>.<-.<.++
+.------.--------.>>+.>++.
```

Hello World!

Brainsuck Instruction Set

Instruction	Meaning
`Move(n)`	`ptr += n;`
`Add(n)`	`*ptr += n;`
`Loop(body)`	`while (*ptr) { body; }`
`Out`	`putchar(*ptr);`
`In`	`*ptr = getchar();`
`Scan(n)`	`while (*ptr) { ptr +=n; }`
`Clear`	`*ptr = 0`
`Copy(offset)`	`*(ptr + offset) = *ptr;`
`Multi(offset, n)`	`*(ptr + offset) += (*ptr) * n;`
`Halt`	`abort();`

Brainsuck Instruction Set

Instruction	Meaning
`Move(n)`	`ptr += n;`
`Add(n)`	`*ptr += n;`
`Loop(body)`	`while (*ptr) { body; }`
`Out`	`putchar(*ptr);`
`In`	`*ptr = getchar();`
`Scan(n)`	`while (*ptr) { ptr +=n; }`
`Clear`	`*ptr = 0`
`Copy(offset)`	`*(ptr + offset) = *ptr;`
`Multi(offset, n)`	`*(ptr + offset) += (*ptr) *`
`Halt`	`abort();`

For Optimization

TreeNode

Forms trees, and supports functional transformations

```scala
trait TreeNode[BaseType <: TreeNode[BaseType]] {
  self: BaseType =>

  def children: Seq[BaseType]

  protected def makeCopy(args: Seq[BaseType]): BaseType

  def transform(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformUp(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  ...
}
```

TreeNode

Forms trees, and supports functional transformations

```
trait TreeNode[BaseType <: TreeNode[BaseType]] {
  self: BaseType =>

  def children: Seq[BaseType]

  protected def makeCopy(args: Seq[BaseType]): BaseType

  def transform(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformUp(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  ...
}
```

TreeNode

Forms trees, and supports functional transformations

```
trait TreeNode[BaseType <: TreeNode[BaseType]] {
  self: BaseType =>

  def children: Seq[BaseType]

  protected def makeCopy(args: Seq[BaseType]): BaseType

  def transform(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformUp(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  ...
}
```

TreeNode

Forms trees, and supports functional transformations

```
trait TreeNode[BaseType <: TreeNode[BaseType]] {
  self: BaseType =>

  def children: Seq[BaseType]

  protected def makeCopy(args: Seq[BaseType]): BaseType

  def transform(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  def transformUp(rule: PartialFunction[BaseType, BaseType]): BaseType = ...

  ...
}
```

TreeNode

Helper classes for leaf nodes and unary nodes

```
trait LeafNode[BaseType <: TreeNode[BaseType]] extends TreeNode[BaseType] {
  self: BaseType =>
  override def children = Seq.empty[BaseType]
  override def makeCopy(args: Seq[BaseType]) = this
}

trait UnaryNode[BaseType <: TreeNode[BaseType]] extends TreeNode[BaseType] {
  self: BaseType =>
  def child: BaseType
  override def children = Seq(child)
}
```

IR

Represented as tree nodes

```
sealed trait Instruction extends TreeNode[Instruction] {
  def next: Instruction
  def run(machine: Machine): Unit
}

sealed trait LeafInstruction extends Instruction with LeafNode[Instruction] {
  self: Instruction =>
  def next: Instruction = this
}

sealed trait UnaryInstruction extends Instruction with UnaryNode[Instruction] {
  self: Instruction =>
  def next: Instruction = child
}
```

databricks™

IR

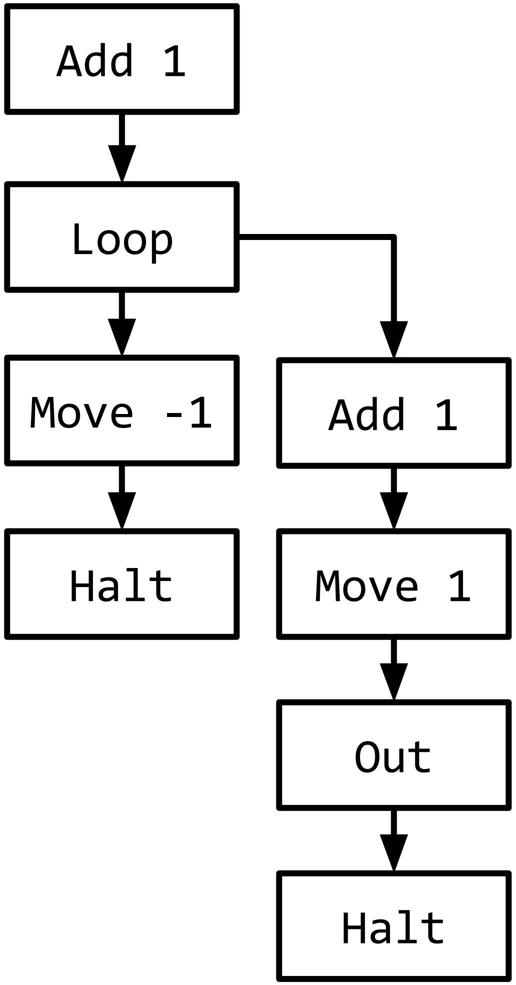
Examples

```scala
case object Halt extends LeafInstruction {
  override def run(machine: Machine) = ()
}

case class Add(n: Int, child: Instruction) extends UnaryInstruction {
  override protected def makeCopy(args: Seq[Instruction]) =
    copy(child = args.head)
  override def run(machine: Machine) = machine.value += n
}

case class Move(n: Int, child: Instruction) extends UnaryInstruction {
  override protected def makeCopy(args: Seq[Instruction]) =
    copy(child = args.head)
  override def run(machine: Machine) = machine.pointer += n
}
```

IR

Examples

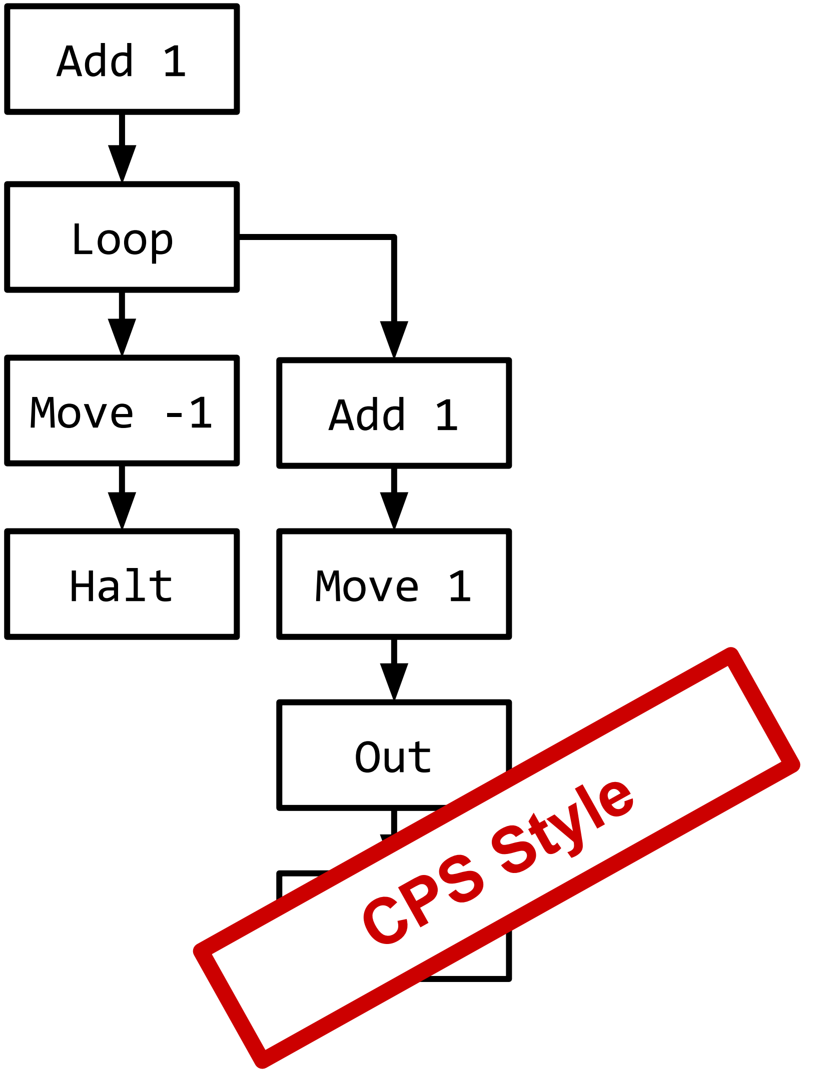
```
// +[<]+>.
Add(1,
  Loop(
    Move(-1,
      Halt),
    Add(1,
      Move(1,
        Out(
          Halt)))))
```



IR

Examples

```
// +[<]+>.
Add(1,
  Loop(
    Move(-1,
      Halt),
    Add(1,
      Move(1,
        Out(
          Halt)))))
```



Rule Executor

`Rules` are functions organized as `Batches`

```scala
trait Rule[BaseType <: TreeNode[BaseType]] {
  def apply(tree: BaseType): BaseType
}

case class Batch[BaseType <: TreeNode[BaseType]](
    name: String,
    rules: Seq[Rule[BaseType]],
    strategy: Strategy)
```

Rule Executor

Each `Batch` has an execution strategy

```
sealed trait Strategy {
  def maxIterations: Int
}
```

A `Batch` of rules can be repeatedly applied to a tree according to some `Strategy`

Rule Executor

Each `Batch` has an execution strategy

```
sealed trait Strategy {
  def maxIterations: Int
}
```

Apply once and only once

```
case object Once extends Strategy {
  val maxIterations = 1
}
```

Rule Executor

Each `Batch` has an execution strategy

```
sealed trait Strategy {
  def maxIterations: Int
}
```

Apply repeatedly until the tree doesn't change

```
final case class FixedPoint(maxIterations: Int) extends Strategy

case object FixedPoint {
  val Unlimited = FixedPoint(-1)
}
```

databricks™

Finally…
Optimizations!

Optimization 1: Merge Moves

Patterns:

- >>>>
- <<<<
- >><<<>>

Basic idea:

- Merge adjacent moves to save instructions

Optimization 1: Merge Moves

Examples:

- `Move(1, Move(1, next)` ⇒ `Move(2, next)`
- `Move(1, Move(-1, next)` ⇒ `next`

Optimization 1: Merge Moves

```
object MergeMoves extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transformUp {
    case Move(n, Move(m, next)) =>
      if (n + m == 0) next else Move(n + m, next)
  }
}
```

Strategy:

- `FixedPoint`

  We'd like to merge all adjacent moves until none can be found

Optimization 2: Merge Adds

Patterns

- ++++
- - - - -
- ++- - -++

Basic idea:

- Merge adjacent adds to save instructions

Optimization 2: Merge Adds

Examples:

- `Add(1, Add(1, next)) ⇒ Add(2, next)`
- `Add(1, Add(-1, next)) ⇒ next`

Optimization 2: Merge Adds

```
object MergeAdds extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transformUp {
    case Add(n, Add(m, next)) =>
      if (n + m == 0) next else Add(n + m, next)
  }
}
```

Strategy:

- `FixedPoint`

  We'd like to merge all adjacent adds until none can be merged

Optimization 3: Clears

Patterns

- `[-]`
- `[--]`
- `[+]`

Basic idea:

- These loops actually zero the current cell. Find and transform them to `Clears`.

Optimization 3: Clears

Examples:

- `Loop(Add(-1, Halt), next) ⇒ Clear(next)`
- `Loop(Add(2, Halt), next) ⇒ Clear(next)`

Optimization 3: Clears

```
object Clears extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transform {
    case Loop(Add(n, Halt), next) =>
      Clear(next)
  }
}
```

Strategy:

- `Once`

  After merging all adds, we can find all “clear” loops within a single run

Optimization 4: Scans

Patterns

- `[<]`
- `[<<]`
- `[>]`

Basic idea:

- These loops move the pointer to the most recent zero cell in one direction. Find and transform them to `Scans`.

Optimization 4: Scans

Examples:

```
- Loop(Move(-1, Halt), next) ⇒
  Scan(-1, next)
- Loop(Move(2, Halt), next) ⇒
  Scan(2, next)
```

Optimization 4: Scans

```scala
object Scans extends Rule[Instruction] {
  override def apply(tree: Instruction) = tree.transform {
    case Loop(Scan(n, Halt), next) =>
      Scan(n, next)
  }
}
```

Strategy:

- `Once`

  After merging all adds, we can find all “scan” loops within a single run

Optimization 5: Multiplications

Patterns:

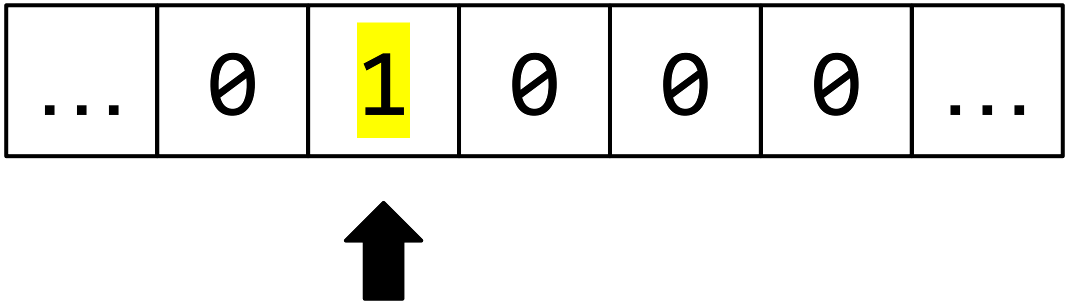
```
- [->>++<<]
  *(ptr + 2) += (*ptr) * 2;
- [->>++<<<--->]
  *(ptr + 2) += (*ptr) * 2;
  *(ptr - 1) += (*ptr) * (-3);
```

Optimization 5: Multiplications

...	0	2	0	0	0	...

```
[->>++<<]
```

Optimization 5: Multiplications




```
[->>++<<]
```

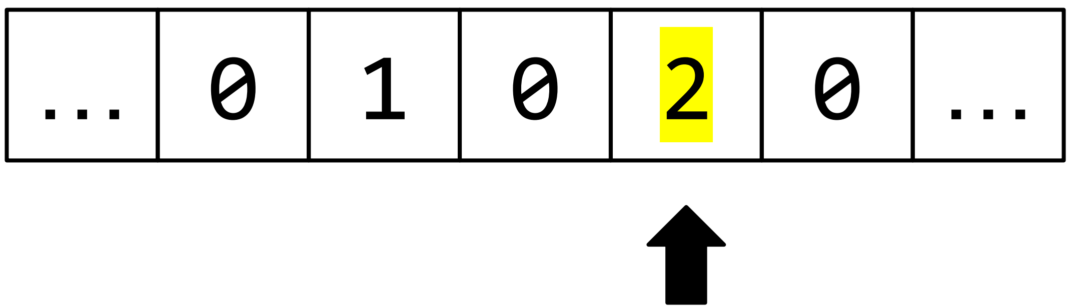
Optimization 5: Multiplications

...	0	1	0	0	0	...

```
[->>++<<]
```

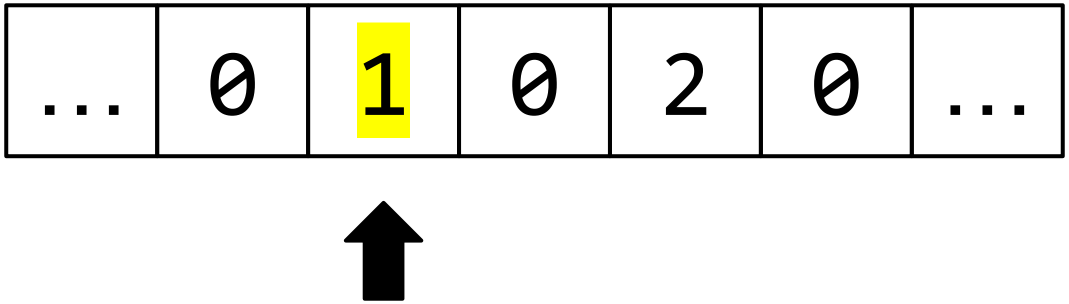
databricks™

Optimization 5: Multiplications




```
[->>++<<]
```

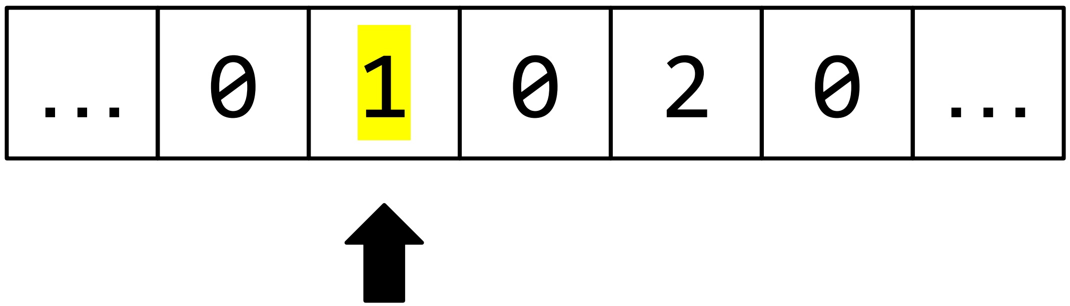
Optimization 5: Multiplications




```
[->>++<<]
```

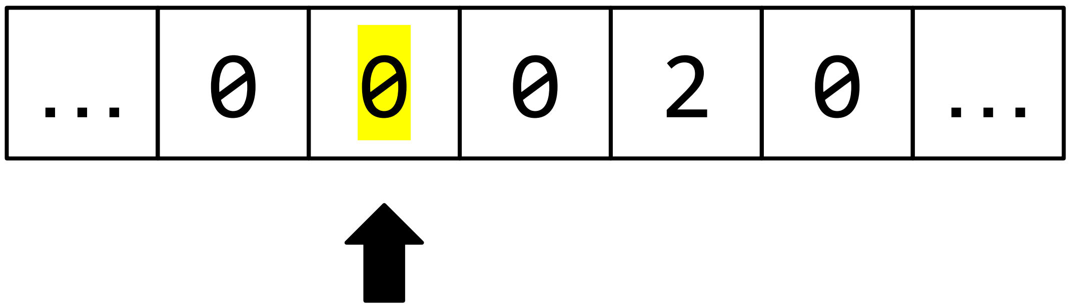
databricks™

Optimization 5: Multiplications



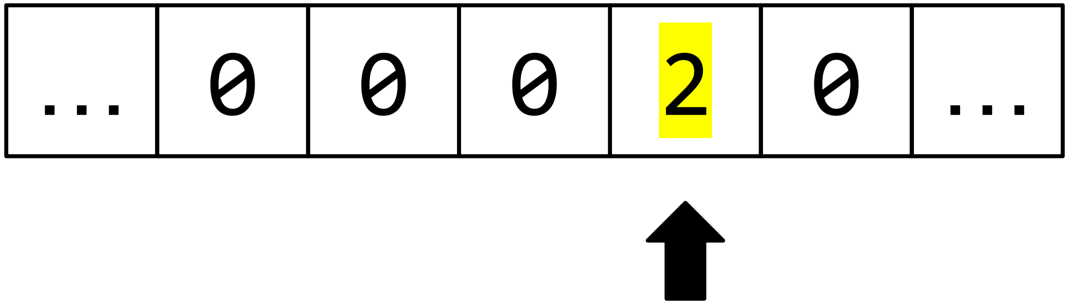

```
[->>++<<]
```

Optimization 5: Multiplications



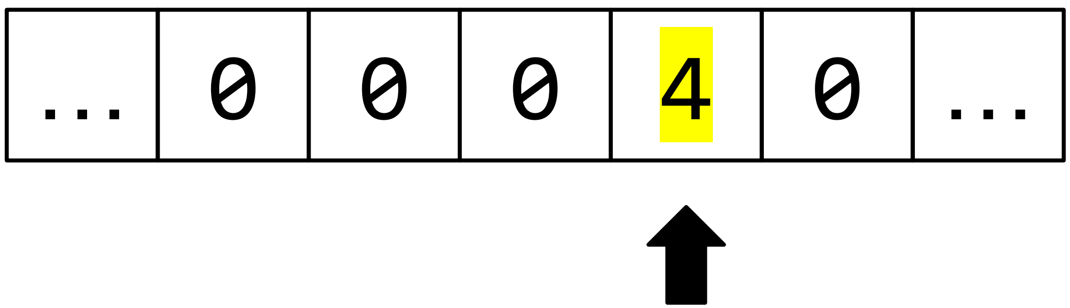

```
[->>++<<]
```

Optimization 5: Multiplications



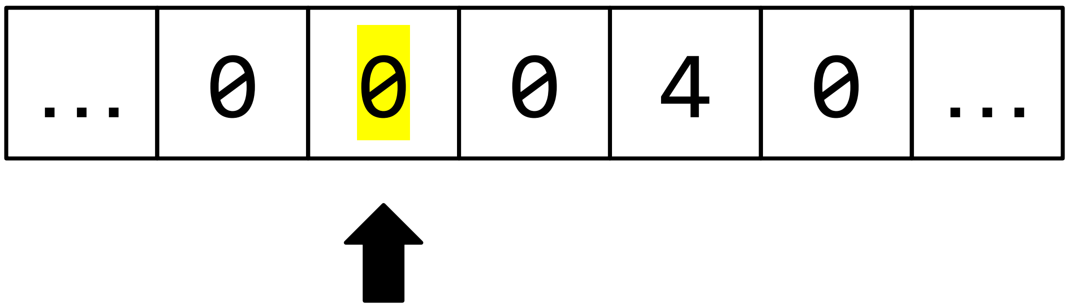

```
[->>++<<]
```

Optimization 5: Multiplications



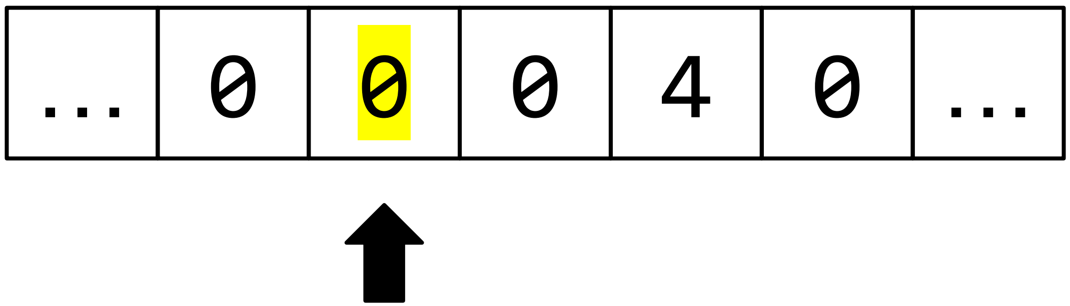

```
[->>++<<]
```

Optimization 5: Multiplications




```
[->>++<<]
```

Optimization 5: Multiplications




```
[->>++<<]
```

Optimization 5: Multiplications

Examples:

```
- Loop(
    Add(-1,
      Move(2,
        Add(2,
          Move(-2, Halt)))), next) ⇒
  Multi(2, 2,
    Clear(next))
```

Optimization 5: Multiplications

Examples:

```
- Loop(
    Add(-1,
      Move(2, Add(2,
        Move(-3, Add(-3,
          Move(1, Halt)))))), next) ⇒
  Multi(2, 2,
    Multi(-1, -3,
      Clear(next)))
```

Optimization 5: Multiplications

This pattern is relatively harder to recognize, since it has variable length:

```
tree.transform {
  case Loop(Add(-1, ???)) => ???
}
```

Optimization 5: Multiplications

What we want to extract from the pattern?

- `(offset, step)` pairs

  For generating corresponding `Multi` instructions sequences

- `finalOffset`

  To check whether the pointer is reset to the original cell at the end of the loop body

Optimization 5: Multiplications

Examples:

- `[->>++<<]`
  - Pairs: `(2,2) :: Nil`, `finalOffset: 2`
- `[->>++<---<]`
  - Pairs: `(2,2)::(1,-3)::Nil`, `finalOffset: 1`
- `[-<<<+++>-->>]`
  - Pairs: `(-3,3)::(-2,-2)::Nil`, `finalOffset: -2`

databricks™

Extractor Objects

In Scala, patterns can be defined as extractor objects with a method named `unapply`. This methods can be used to recognize arbitrarily complex patterns.

Optimization 5: Multiplications

An extractor object which recognizes move-add pairs and extracts offsets and steps:

```scala
object MoveAddPairs {
  type ResultType = (List[(Int, Int)], Int, Instruction)

  def unapply(tree: Instruction): Option[ResultType] = {
    def loop(tree: Instruction, offset: Int): Option[ResultType] = tree match {
      case Move(n, Add(m, inner)) =>
        loop(inner, offset + n).map { case (seq, finalOffset, next) =>
          ((offset + n, m) :: seq, finalOffset, next)
        }
      case inner => Some((Nil, offset, inner))
    }
    loop(tree, 0)
  }
}
```

databricks

Optimization 5: Multiplications

An extractor object which recognizes move-add pairs and extracts offsets and steps:

```
object MoveAddPairs {
  type ResultType = (List[(Int, Int)], Int, Instruction)

  def unapply(tree: Instruction): Option[ResultType] = {
    def loop(tree: Instruction, offset: Int): Option[ResultType] = tree match {
      case Move(n, Add(m, inner)) =>
        loop(inner, offset + n).map { case (seq, finalOffset, next) =>
          ((offset + n, m) :: seq, finalOffset, next)
        }
      case inner => Some((Nil, offset, inner))
    }
    loop(tree, 0)
  }
}
```

Optimization 5: Multiplications

```
object MultisAndCopies extends Rule[Instruction] {
  override def apply(tree: Instruction): Instruction = tree.transform {
    case Loop(Add(-1, MoveAddPairs(seq, offset, Move(n, Halt))), next)
        if n == -offset =>
      seq.foldRight(Clear(next): Instruction) {
        case ((distance, increment), code) => Multi(distance, increment, code)
      }
  }
}
```

Strategy:

- `Once`

  After merging moves and adds, we can find all multiplication loops within a single run

Optimization 6: Copies

Patterns:

- `[->>+<<]`
- `[->>+<<<+>]`

Basic idea:

- These are actually special forms of multiplication loops with 1 as multiplicand
- Replace `Multi` with a cheaper `Copy`

Optimization 6: Copies

Examples:

```
- Loop(
   Add(-1,
    Move(2, Add(1,
     Move(-3, Add(1,
      Move(1, Halt)))))), next) ⇒
  Copy(2,
   Copy(-1),
    Clear(next)))
```

Optimization 6: Copies

```
object MultisAndCopies extends Rule[Instruction] {
  override def apply(tree: Instruction): Instruction = tree.transform {
    case Loop(Add(-1, MoveAddPairs(seq, offset, Move(n, Halt))), next)
        if n == -offset =>
      seq.foldRight(Clear(next): Instruction) {
        case ((distance, 1), code) => Copy(distance, code)
        case ((distance, increment), code) => Multi(distance, increment, code)
      }
  }
}
```

Strategy:

- Once

  After merging moves and adds, we can find all copy loops within a single run

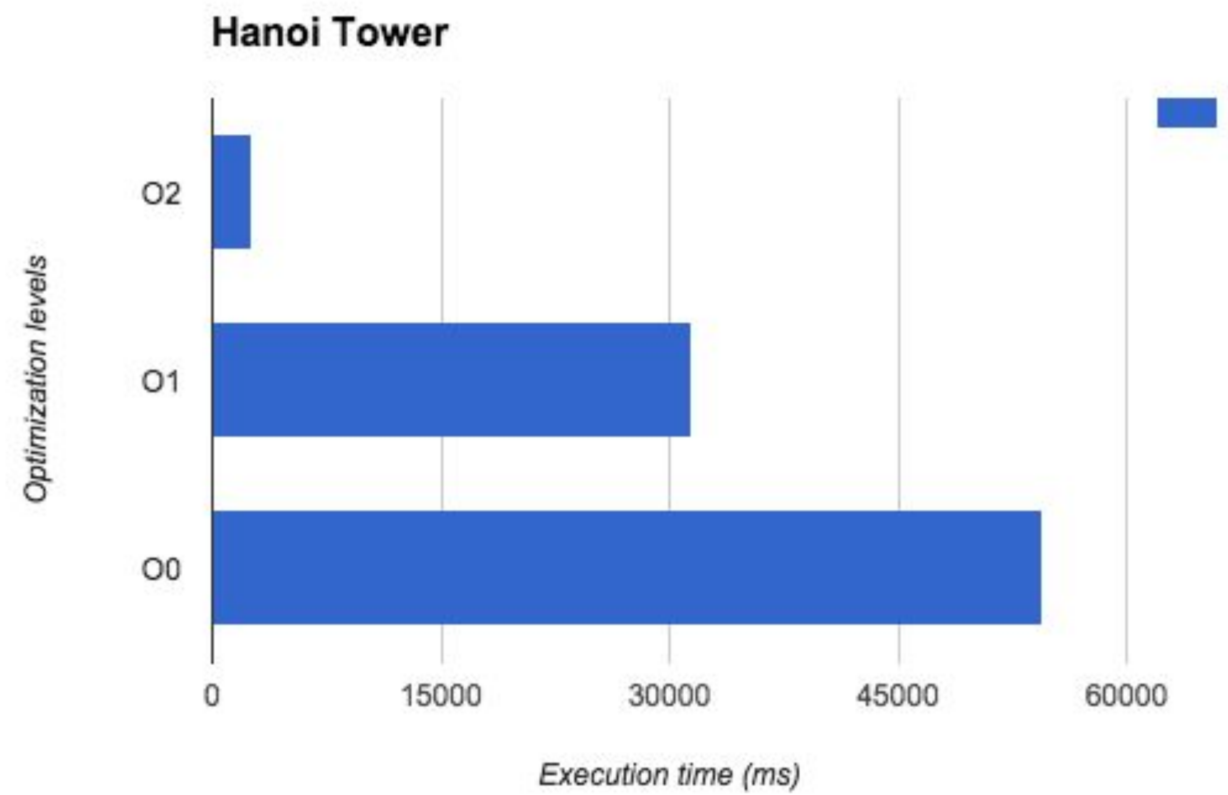
Composing the Optimizer

```
val optimizer = new Optimizer {
  override def batches = Seq(
    Batch(
      "Contraction",
      MergeMoves :: MergeAdds :: Nil,
      FixedPoint.Unlimited),

    Batch(
      "LoopSimplification",
      Clears :: Scans :: MultisAndCopies :: Nil,
      Once)
  ).take(optimizationLevel)
}
```

Demo: Hanoi Tower

```
                        Towers of Hanoi in Brainf*ck
            Written by Clifford Wolf <http://www.clifford.at/bfcpu/>



            xXXXXXXXXXx
  xXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx                     xXXXXXx
xXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx         xXXXXXXXXXXXXXXXXXXXXXXXXXx
-----------------------------------     -----------------------------------



                                    xXx
                              xXXXXXXXXXXXXXx
                            xXXXXXXXXXXXXXXXXXx
                          xXXXXXXXXXXXXXXXXXXXXXx
                    -----------------------------------
```

databricks™



Hanoi Tower
Optimization levels
O2
O1
O0
0
15000
30000
45000
60000
Execution time (ms)

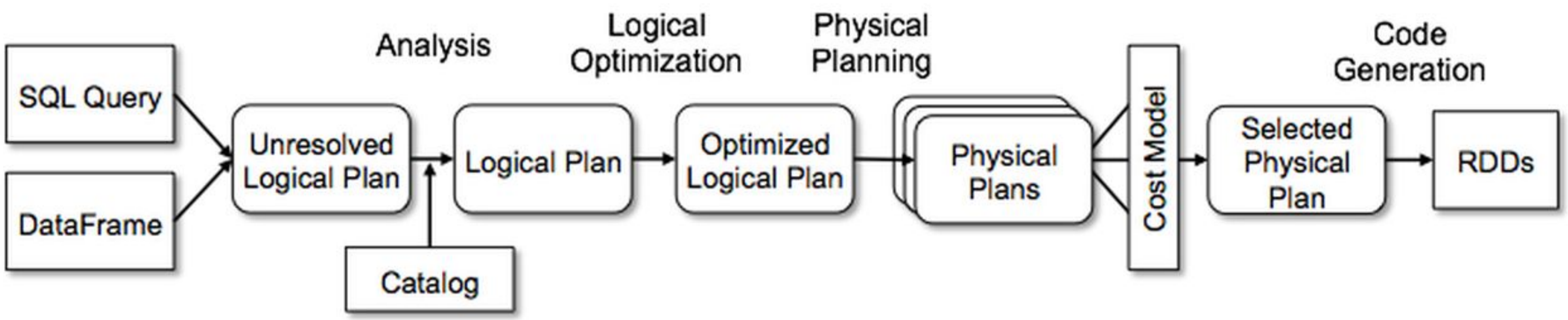
databricks

How does Spark SQL use Catalyst

The following structures are all represented as `TreeNode` objects in Spark SQL

- Expressions
- Unresolved logical plans
- Resolved logical plans
- Optimized logical plans
- Physical plans

How does Spark SQL use Catalyst

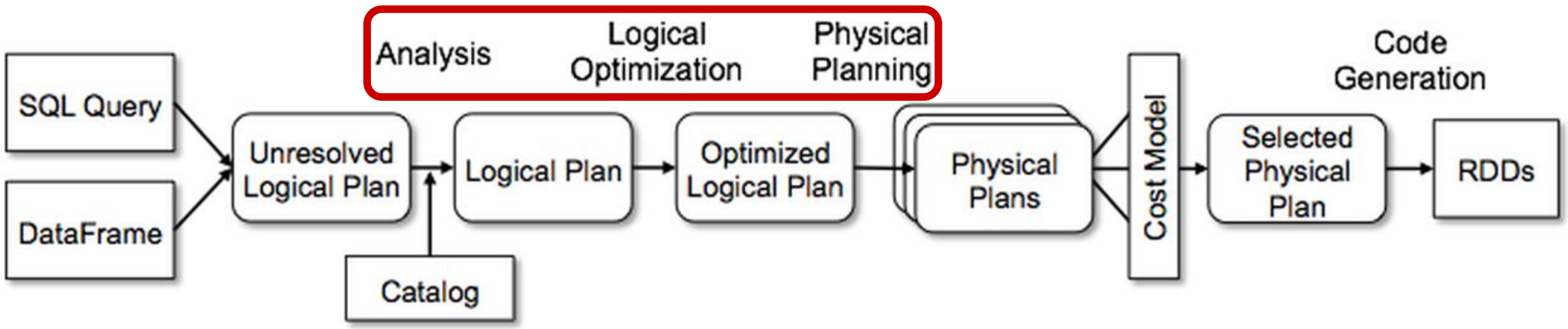


How does Spark SQL use Catalyst

The actual Catalyst library in Spark SQL is more complex than the one shown in this talk. It's used for the following purposes:

- Analysis (logical ⇒ logical)
- Logical optimization (logical ⇒ logical)
- Physical planning (logical ⇒ physical)

How does Spark SQL use Catalyst



References

- Spark SQL and Catalyst
  - Deep Dive into Spark SQL's Catalyst Optimizer (Databricks blog)
  - Spark SQL: Relational Data Processing in Spark (Accepted by SIGMOD15)
- Brainfuck language and optimization
  - Brainfuck optimization strategies
  - Optimizing brainfuck compiler
  - Wikipedia: Brainfuck
- Brainsuck code repository
  - https://github.com/liancheng/brainsuck

Thanks

Q & A

databricks™