Understanding Query Planning

Janani Ravi
CO-FOUNDER, LOONYCORN

www.loonycorn.com

Overview

Query planning in Spark

Tungsten and Catalyst

Broadcast variables and accumulators

UDFs in query planning

Project Tungsten

Changes Starting Spark 2.0

Easier

Unifying Datasets and DataFrames, SQL support

Faster

Optimize like a compiler, not a DBMS

Project Tungsten

Umbrella project, launched in April 2015, to make changes to Apache Spark's execution engine that focuses on substantially improving the efficiency of memory and CPU.

Backdrop

In early 2015...

- Disks are getting faster: SSDs
- Networks are getting faster: 10 Gbps links
- CPU is now the bottleneck

Backdrop

But...

- Spark was optimizing for IO or network communication
- Java makes heavy use of virtual functions
- JVM garbage collection costs were heavy

Volcano Iterator Model

Used before Project Tungsten

Classic query evaluation strategy

Each query consisted of operators

Each operator implemented an interface

Volcano Iterator Model

Design Choice	Implication
Heavy use of interfaces	Lots of virtual function dispatches
Query represented as complex tree of function calls	Compiler features such as pipelining, prefetching, loop unrolling become hard
Interface has single method next()	Each tuple placed on call stack (memory, not CPU register)

Project Tungsten

Memory management: Go beyond JVM object model and GC

Cache-aware computation: Refine algorithms and data structures

Code generation: Learn from modern compilers

From Volcano to Tungsten

Volcano in Spark 1.x

Classic DBMS strategy, optimizes IO and network access

Rely on Java objects and JVM garbage collection

Hard to leverage modern compiler optimizations

Lots of virtual function dispatches

Tungsten in Spark 2.x

Focus on CPU optimization; disk and network no longer the bottleneck

Take control of object creation and garbage collection

Heavily leverage loop unrolling, 1 CPU instruction for multiple tuples

Entirely avoid virtual function calls

Performance Improvements

Comparison of time per row, on 1 billion records on single thread

Primitive	Spark 1.6	Spark 2.0	Speedup Factor
filter	15ns	1.1ns	13.6
sum w/o group	14ns	0.9ns	15.6
sum w/ group	79ns	10.7ns	7.4
hash join	115ns	4.0ns	28.8
sort (8-bit)	620ns	5.3ns	117.0
sort (64-bit)	620ns	40ns	15.5
sort-merge-join	750ns	700ns	1.1

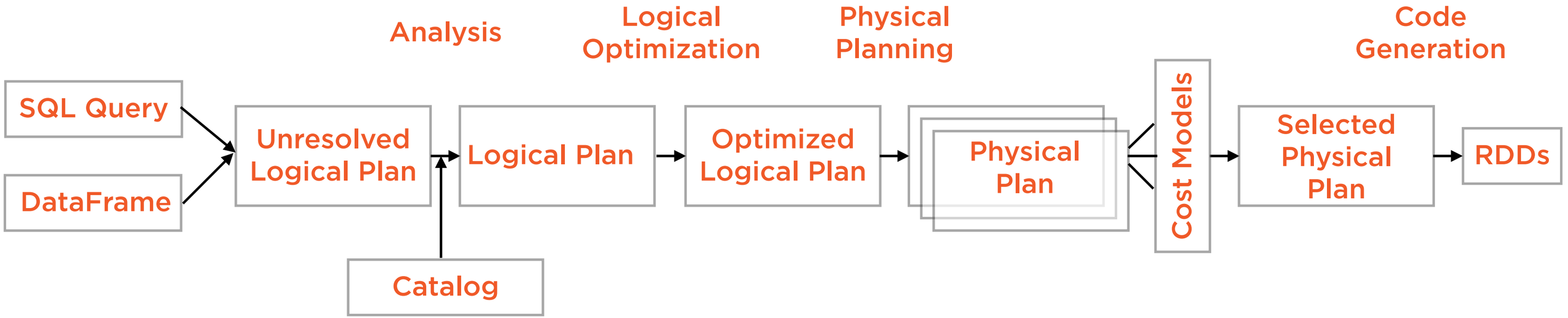
Source: https://databricks.com/blog/2016/07/26/introducing-apache-spark-2-0.html

Catalyst Optimizer

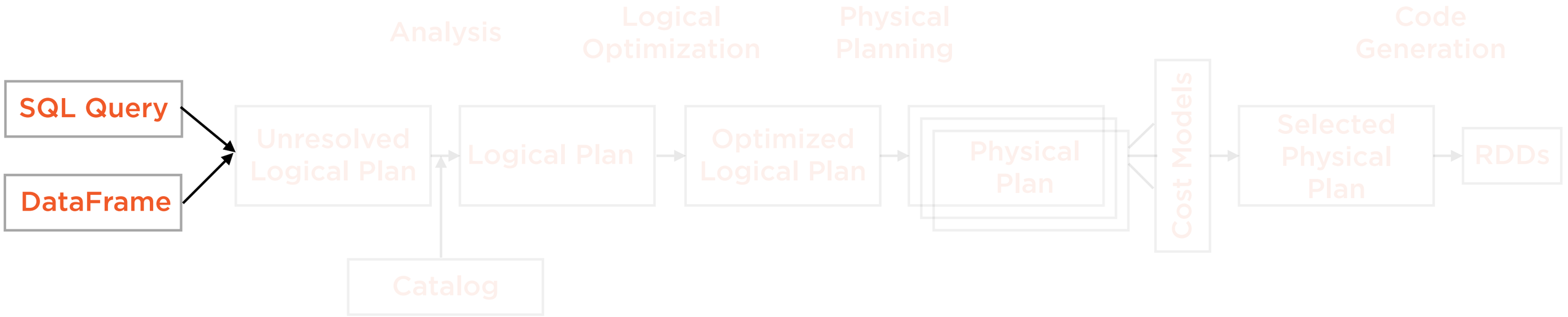
Catalyst Optimizer

Optimization engine that powers Spark SQL (as well as DataFrame API) since 2015.

Catalyst Optimizer

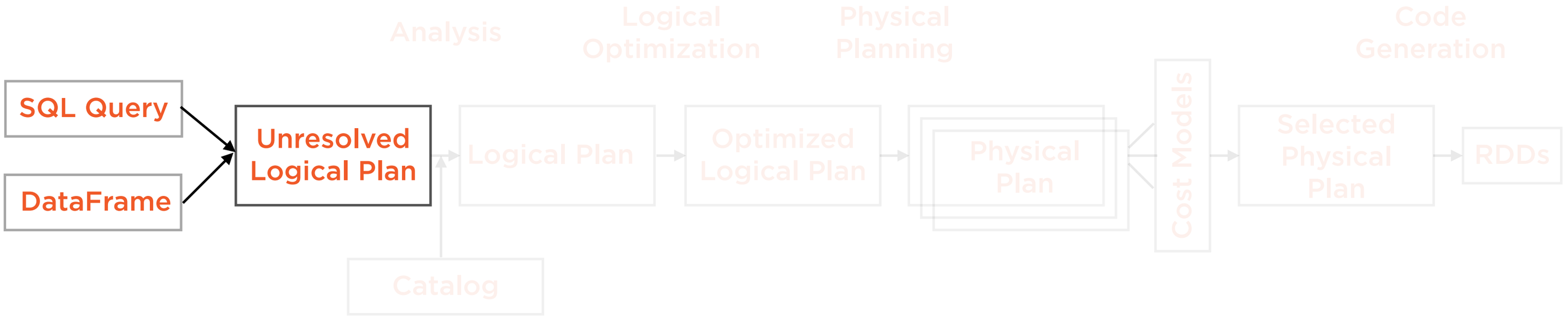


Catalyst Optimizer



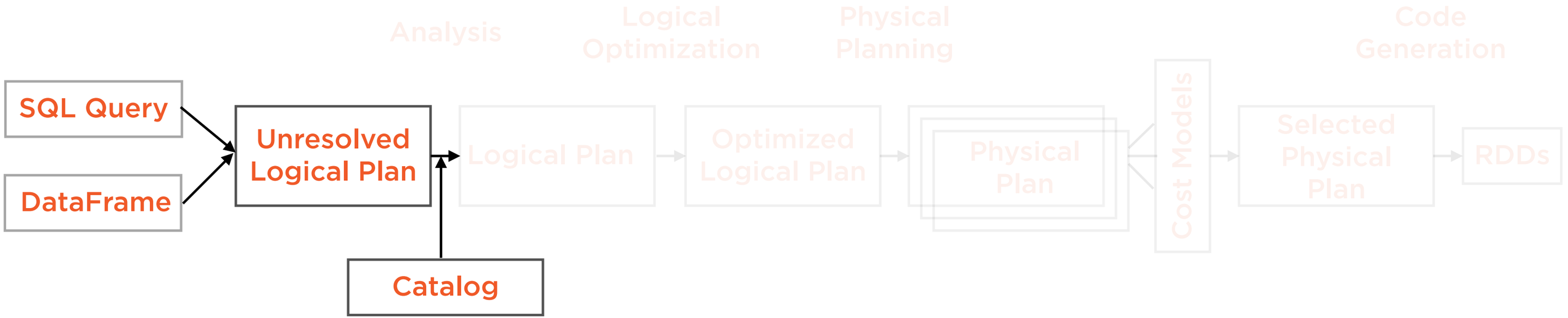
Relations to be processed

Catalyst Optimizer



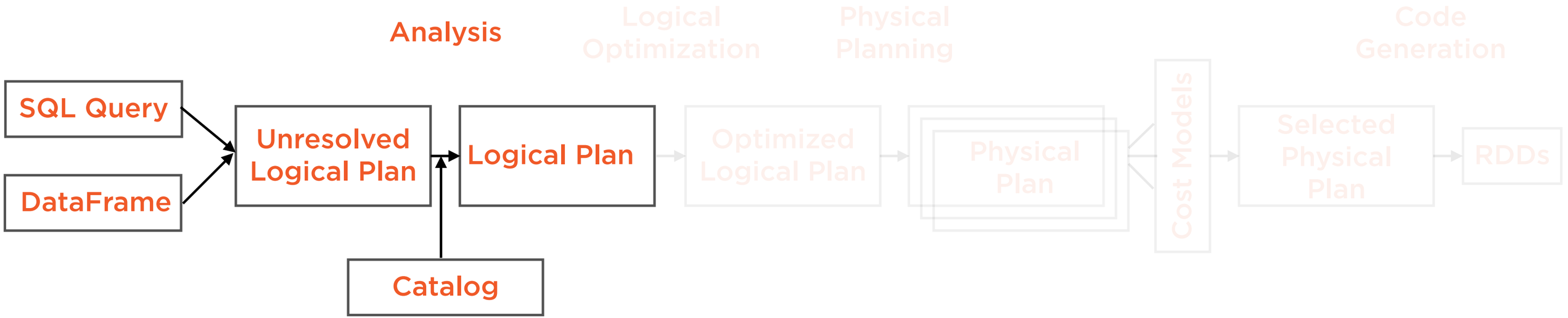
Unresolved as column types and existence yet to be ascertained

Catalyst Optimizer



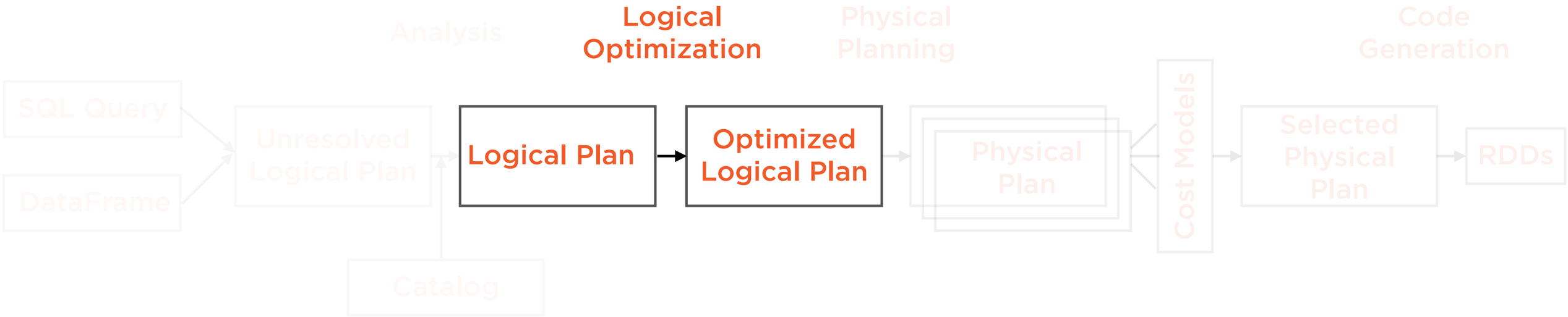
Catalog tracks tables in all data sources to resolve plan

Catalyst Optimizer



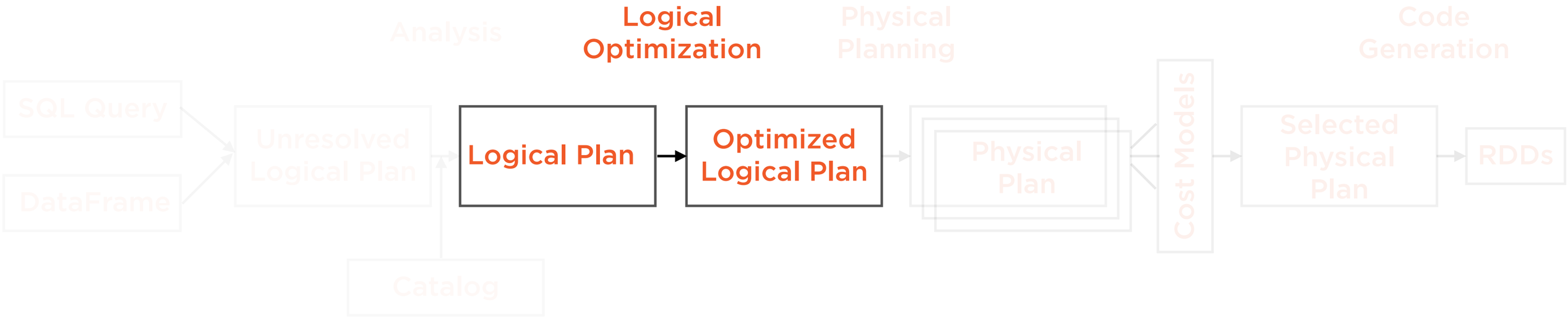
Output of the Analysis phase is a logical plan

Catalyst Optimizer



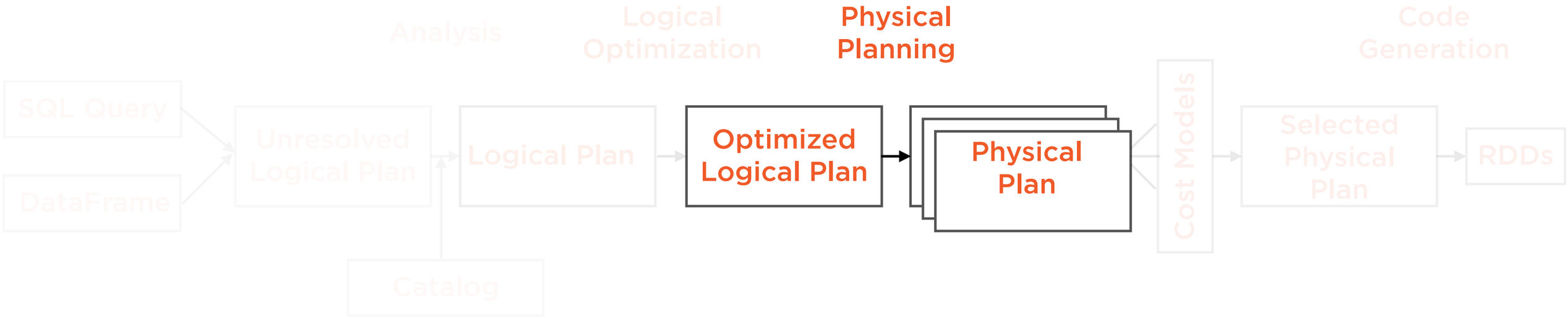
Predicate pushdown, projection pruning, null propagation, expression simplification...

Catalyst Optimizer



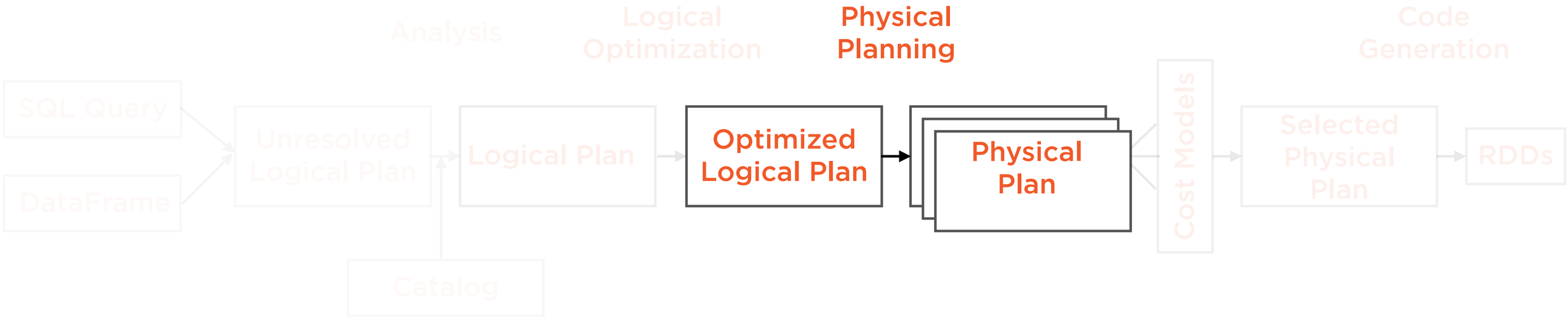
Generate various such logical plans, then pick the lowest-cost (optimized) logical plan

Catalyst Optimizer



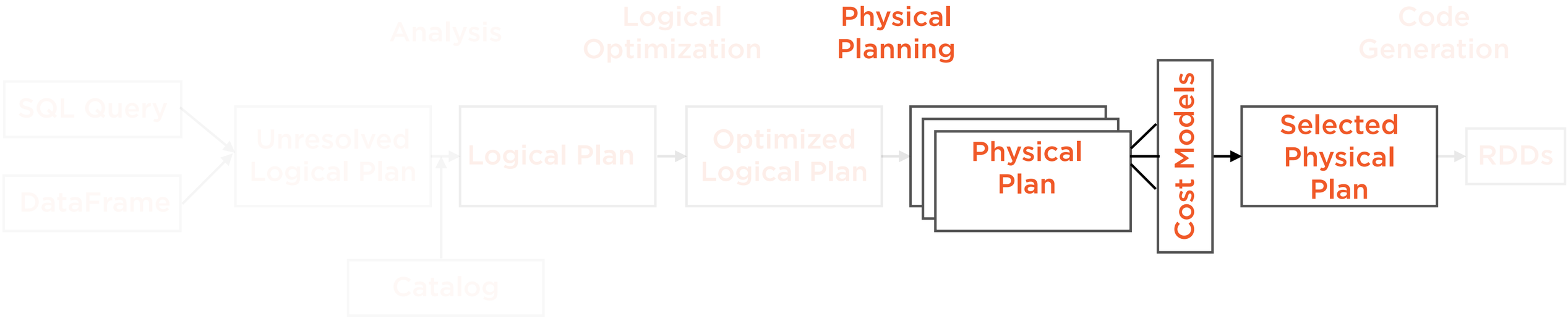
Generate different alternative physical plans for this optimized logical plan

Catalyst Optimizer



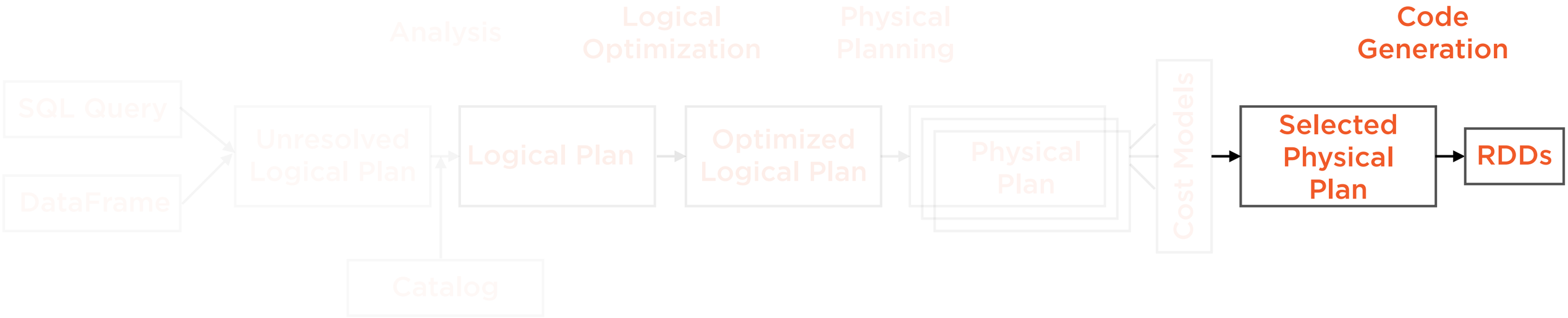
Here, Catalyst interfaces with the Spark execution engine (Tungsten)

Catalyst Optimizer



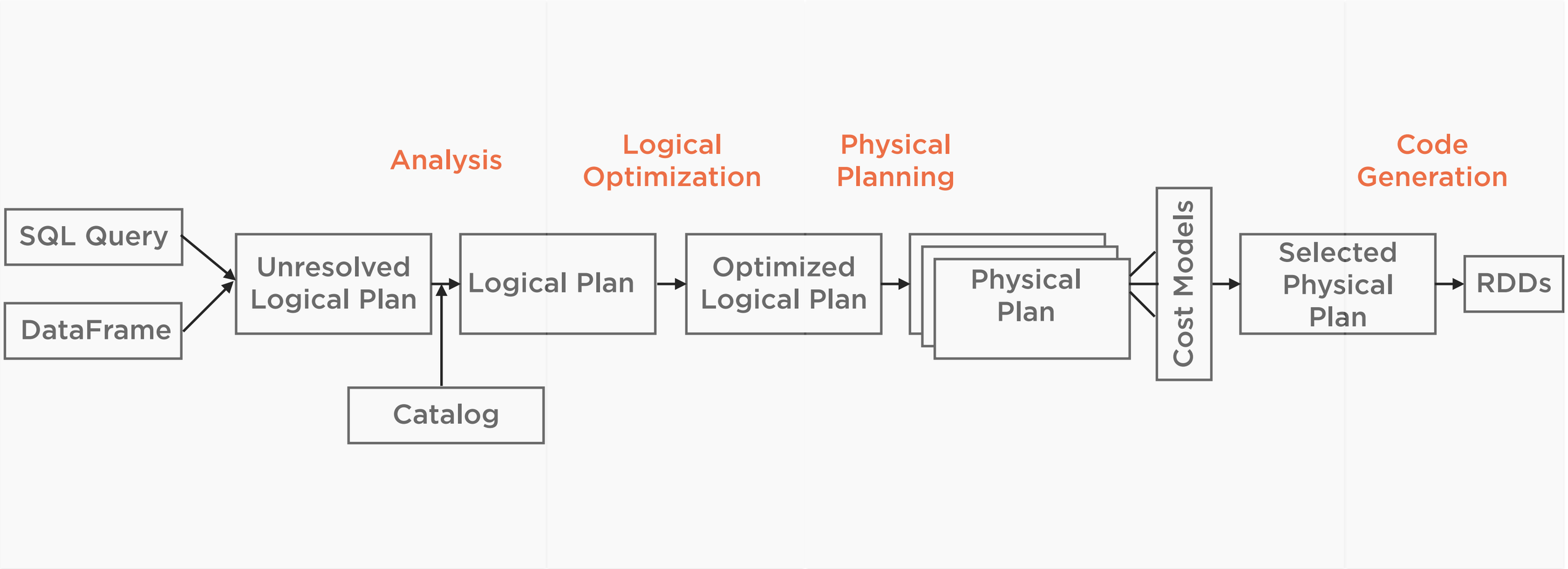
Apply cost models to find the best physical plan

Catalyst Optimizer



Generate Java bytecode to run on each machine

Catalyst Optimizer

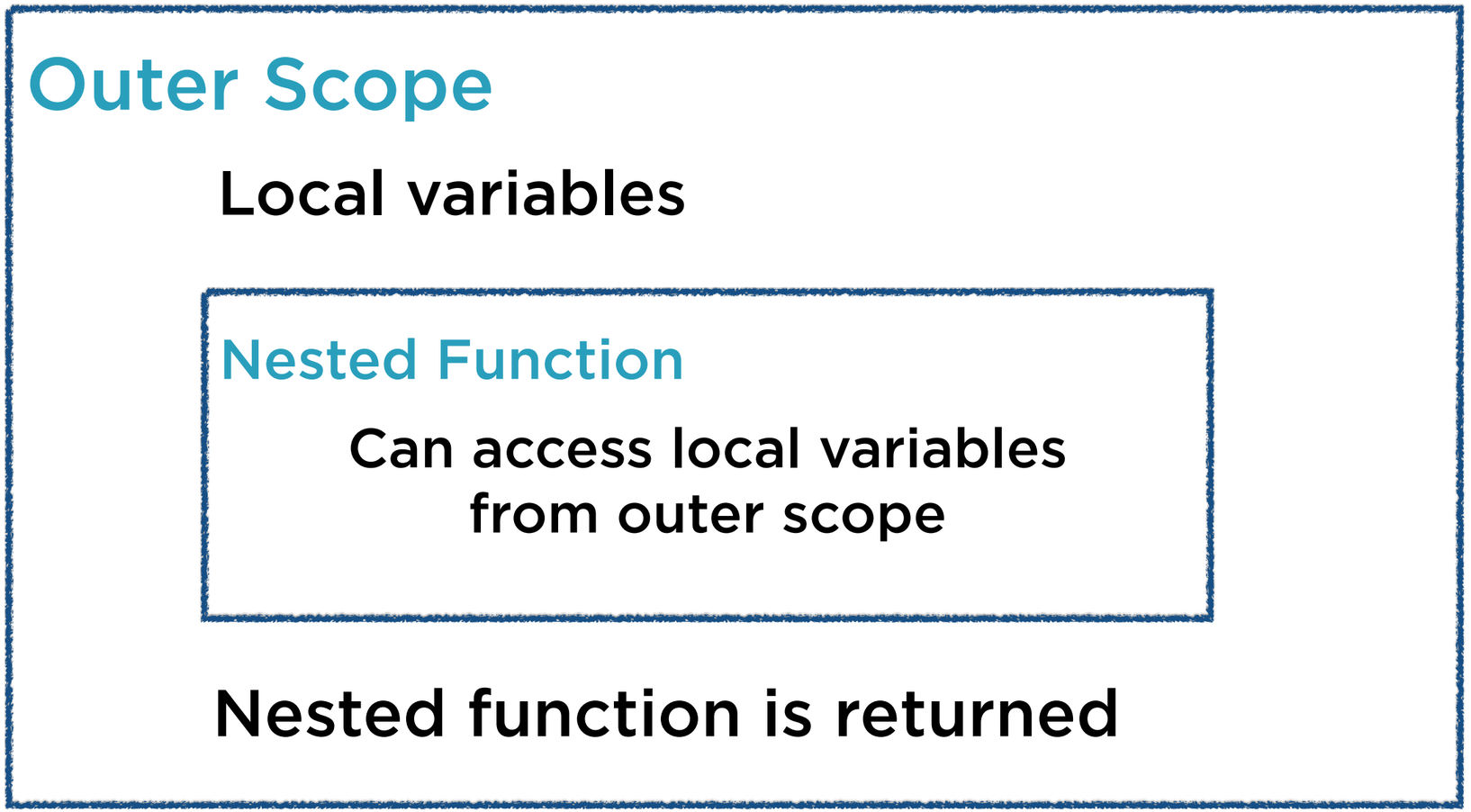


Four phases of query optimization and execution

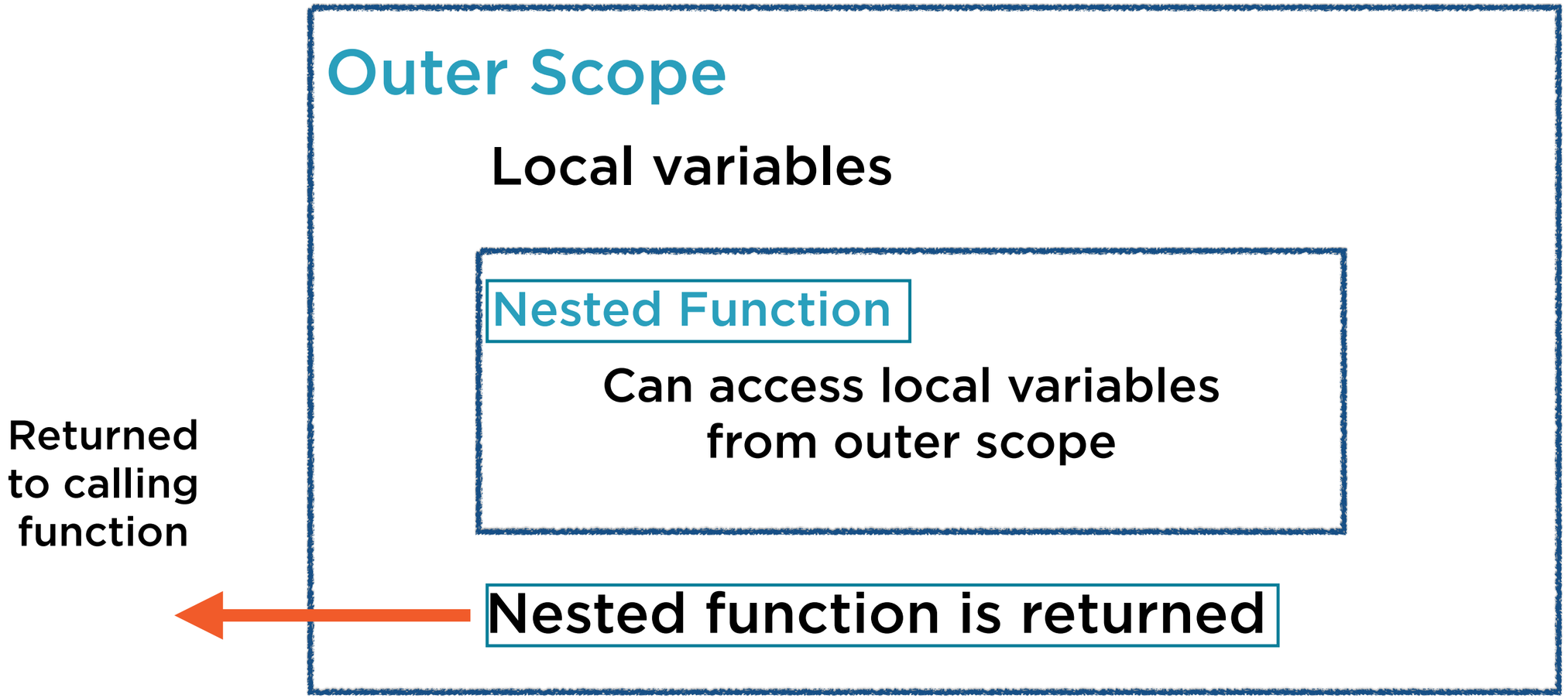
Accumulators and Broadcast Variables

Spark is written in Scala, and heavily utilizes closures

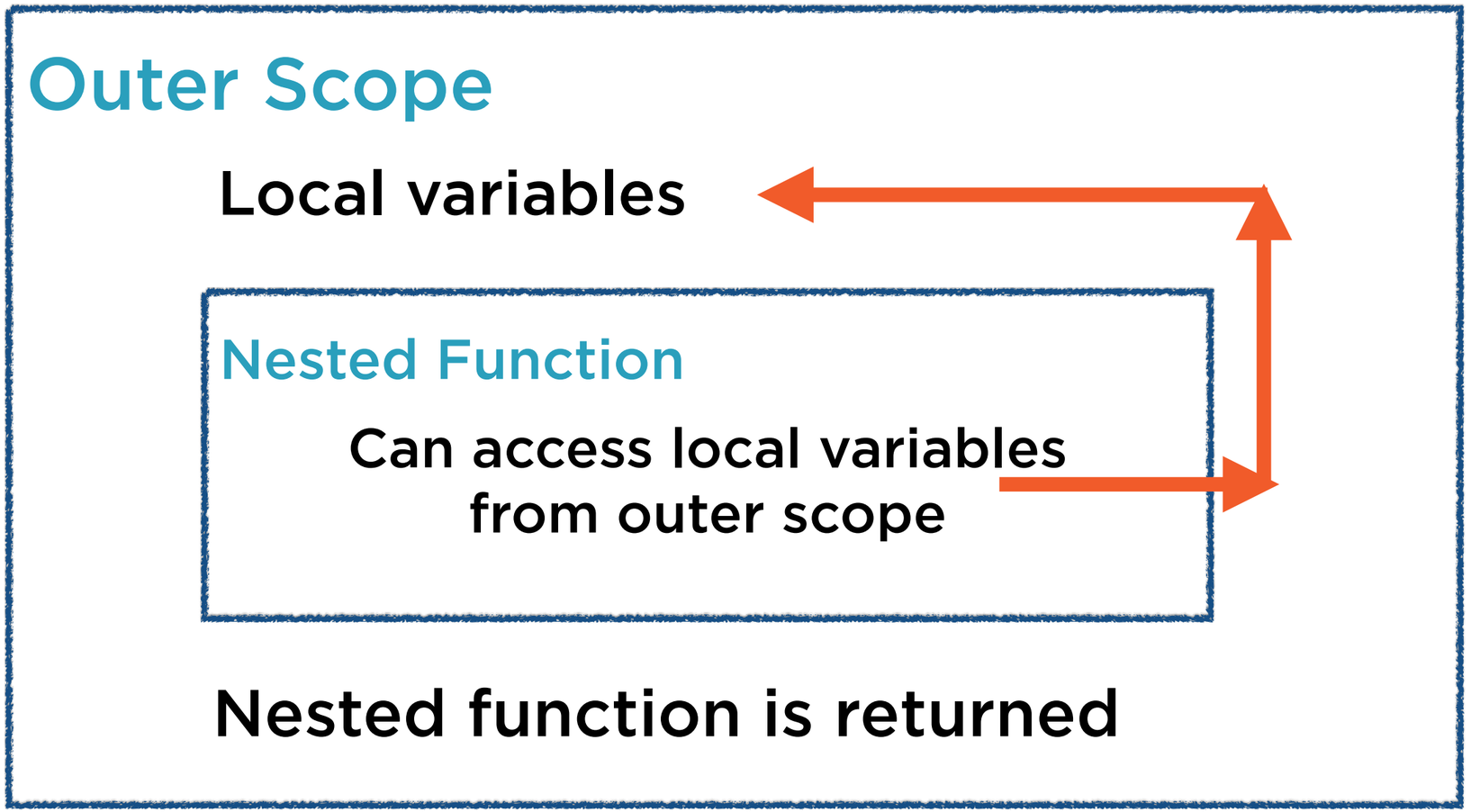
Closures


Outer Scope
Local variables
Nested Function
Can access local variables
from outer scope
Nested function is returned

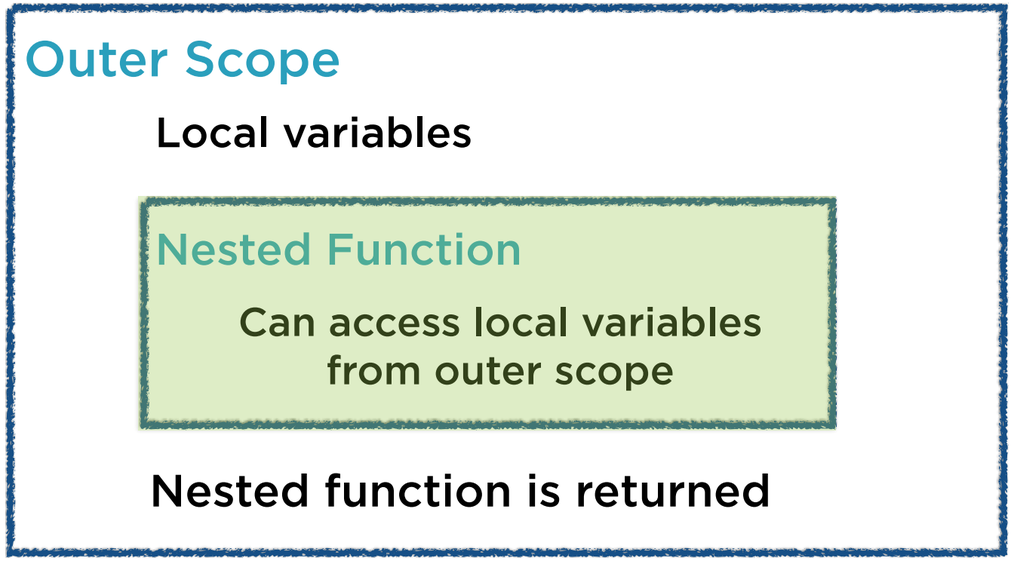
Closures


Outer Scope
Local variables
Nested Function
Can access local variables
from outer scope
Nested function is returned
Returned
to calling
function

Closures

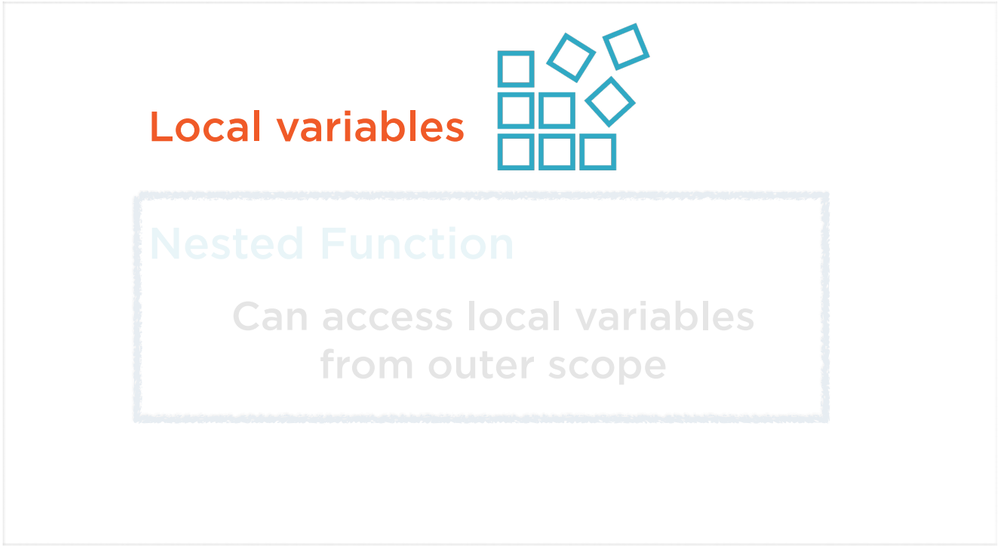

Outer Scope
Local variables
Nested Function
Can access local variables
from outer scope
Nested function is returned

Closures



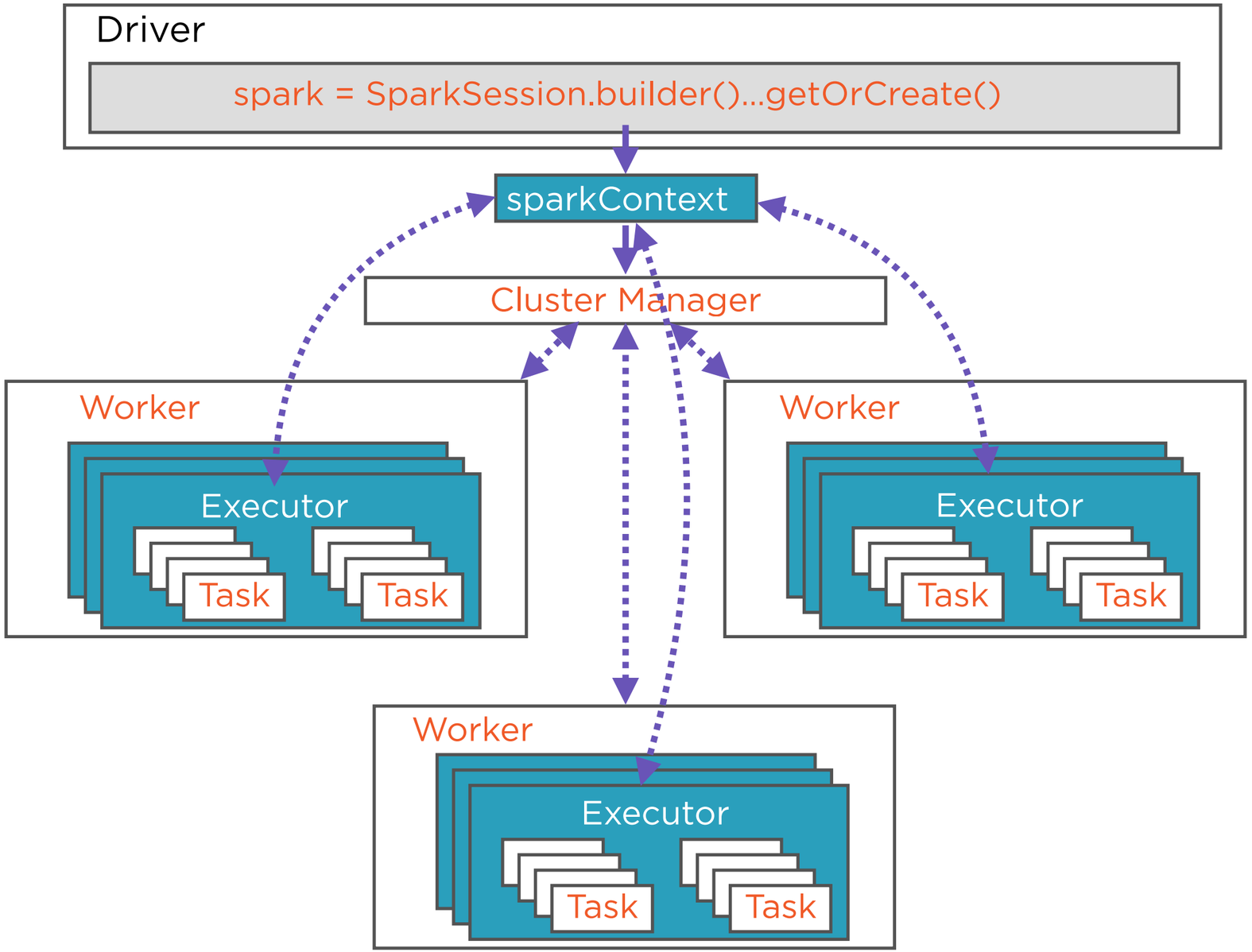
The returned nested function is called a closure

Closures



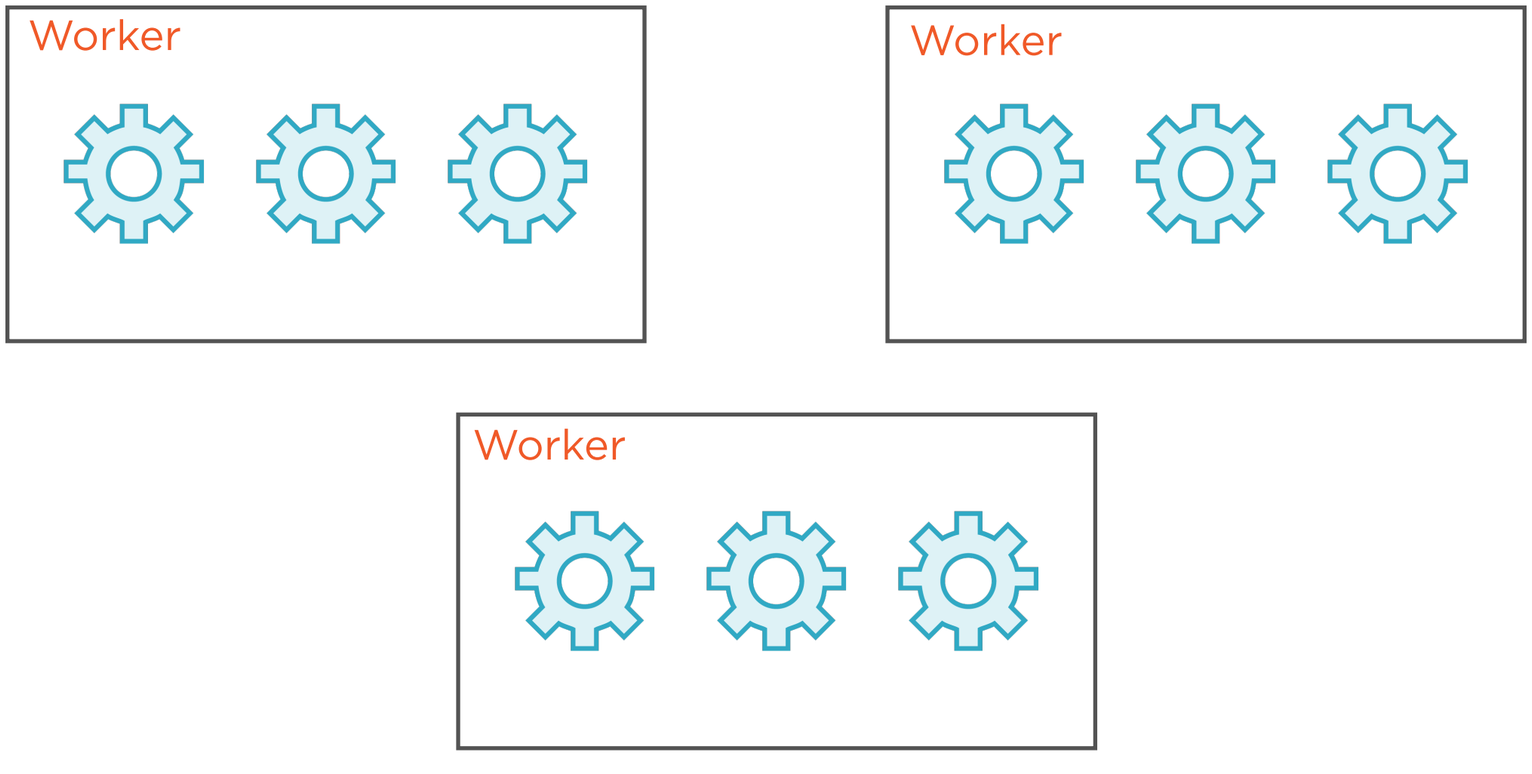
The closure retains its copies of local variables
- even after the outer scope ceases to exist

Spark 2.x Architecture



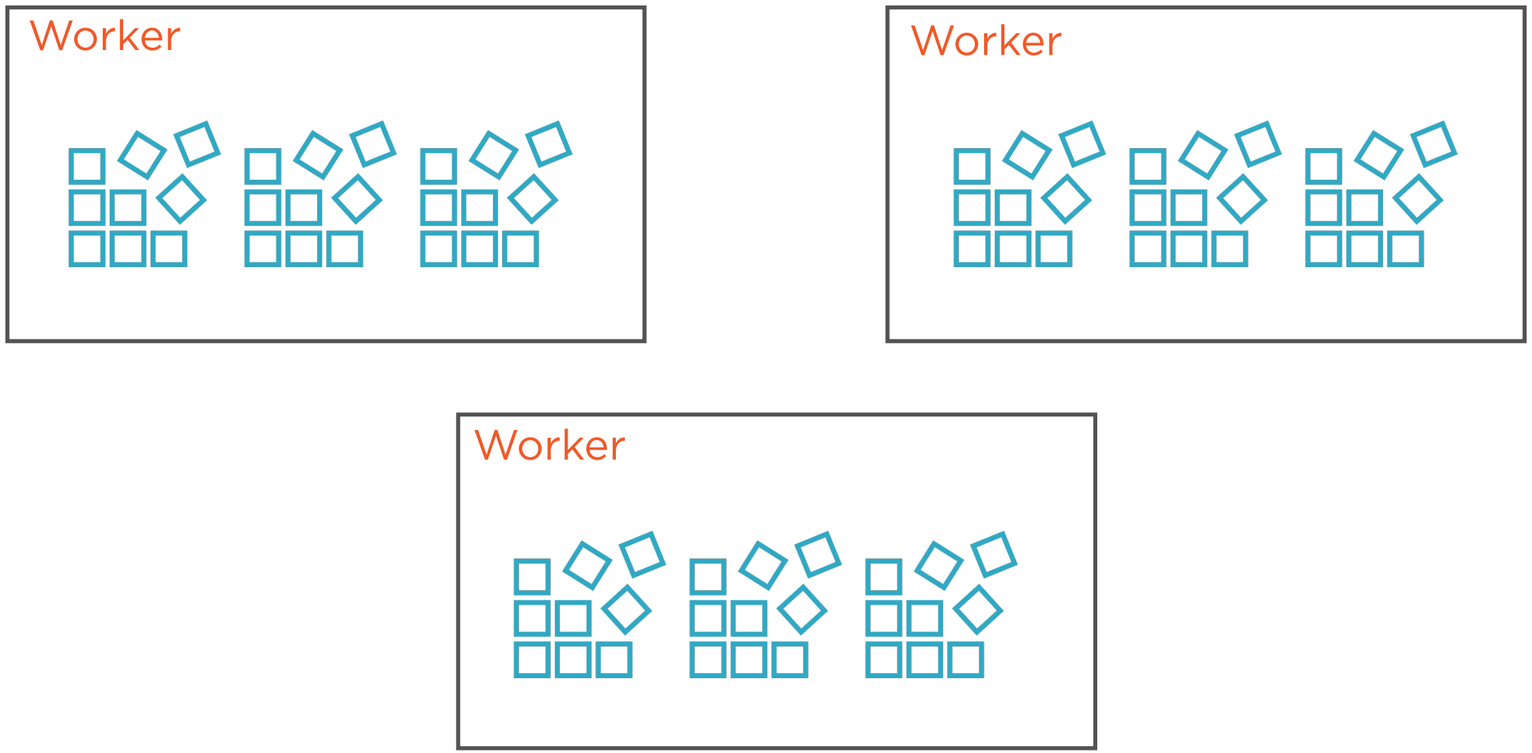
Spark 2.x Architecture

Tasks which run on individual workers are closures

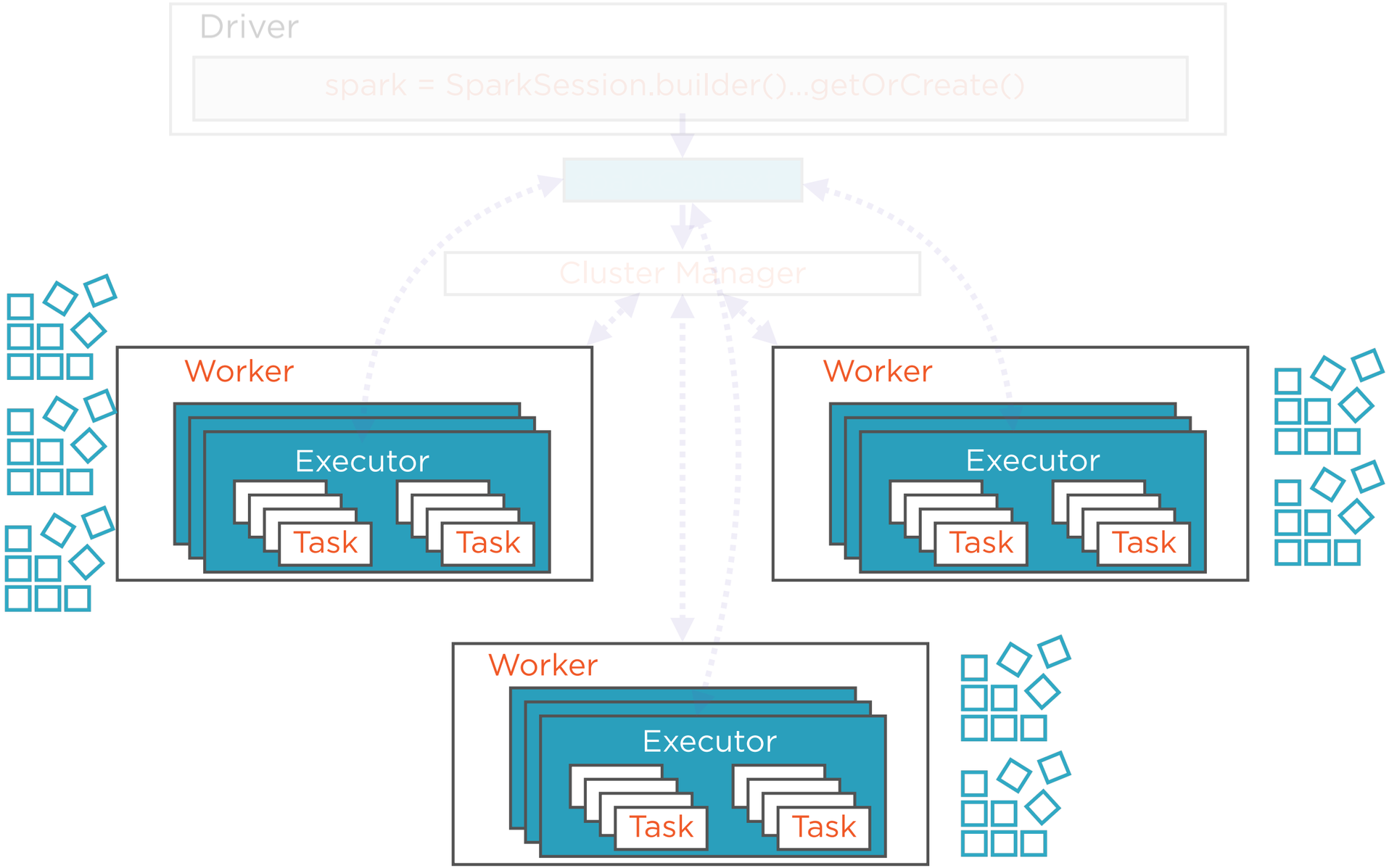


Spark 2.x Architecture

Every task will contain a copy of the variables that it works on



One Copy Per Task



Closures

By default functions in Spark carry around copies of all variables

Each function copies each variable to each node it runs on

No updates sent back to master

Closures

1 copy per task

- Lots of copies passed around

Shuffling ~ further cost hit

All copying from master

- No copying from 1 task to another

Closures

Need shared variables across tasks?

- Broadcast variables
- Accumulators

Shared Variables

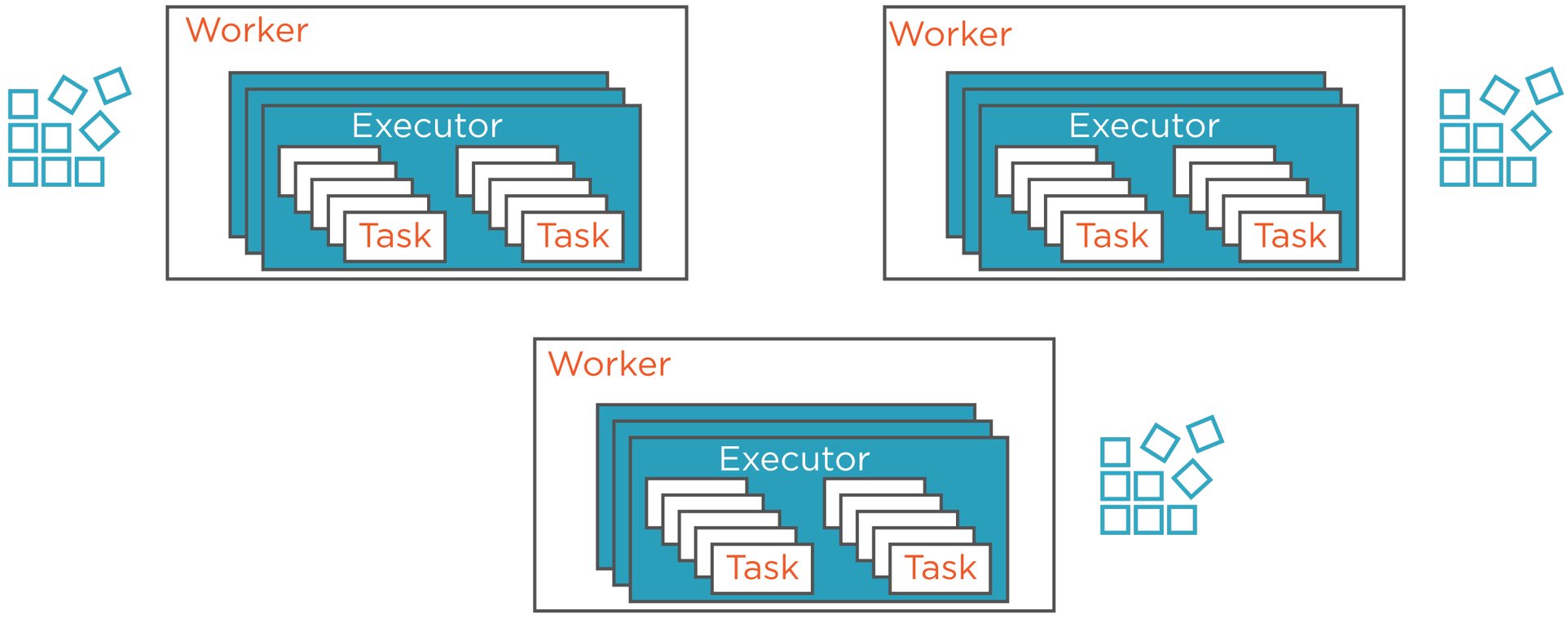
Broadcast Variables

Only 1 read-only copy per node
(not 1 copy per task)

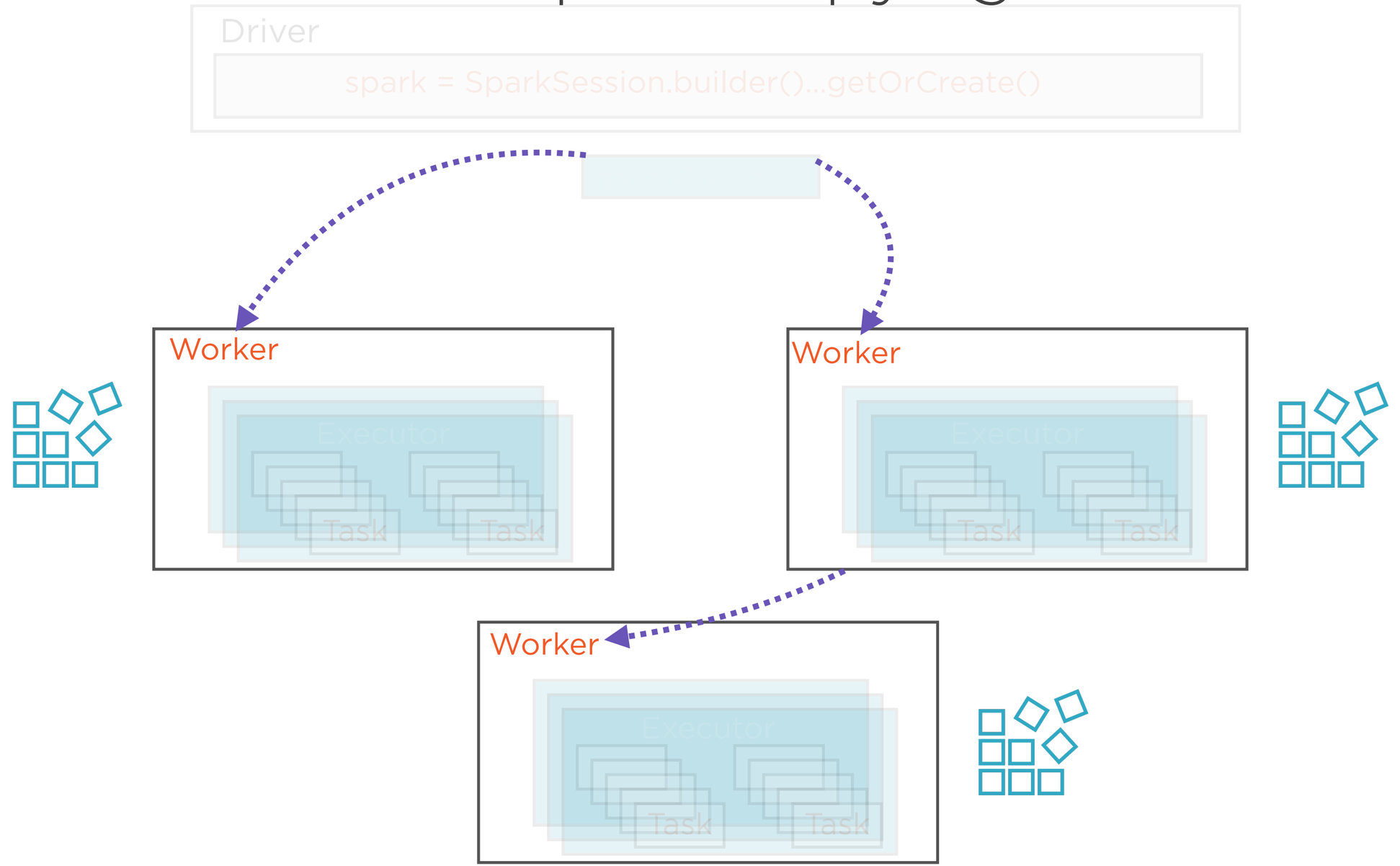
Accumulators

Broadcast to workers but can be
modified by adding to it

One Copy Per Node



Peer-to-peer Copying Too



Broadcast Variables

Will be cached in-memory on each node

So, can be large, but not too large

Broadcast Variables

Use whenever tasks across stages need same data

Share dataset with all nodes

- training data in ML
- static lookup tables

Shared Variables

Broadcast Variables

Only 1 read-only copy per node
(not 1 copy per task)

Accumulators

Broadcast to workers but can be
modified by adding to it

Accumulator Variables

Read-write shared variables

Added associatively and commutatively

Commutativity: A + B = B + A

Associativity: A + (B+C) = (A+B) + C

Accumulator Variables

Counters or sums

Workers can only modify state

Only the driver program can read state

Demo

Broadcast variables

Demo

Accumulators

UDFs in Spark

UDFs

User Defined Functions that can be registered with a Spark Session and invoked on DataFrames just like standard Spark operations.

Using UDFs Correctly

UDFs are not optimized by Spark

- Can lead to significant performance penalties

Error and null handling are responsibility of user

Demo

User-defined functions in Spark DataFrames

Demo

User-defined functions in Spark SQL

Summary

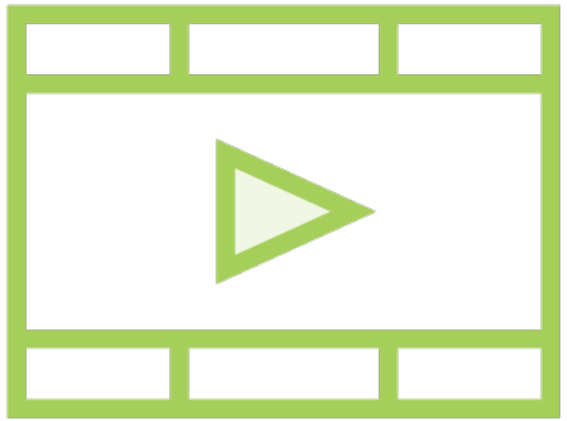
Query planning in Spark

Tungsten and Catalyst

Broadcast variables and accumulators

UDFs in query planning

Related Courses



Exploring the Apache Spark Structured Streaming API

Modeling Streaming Data for Processing with Apache Beam