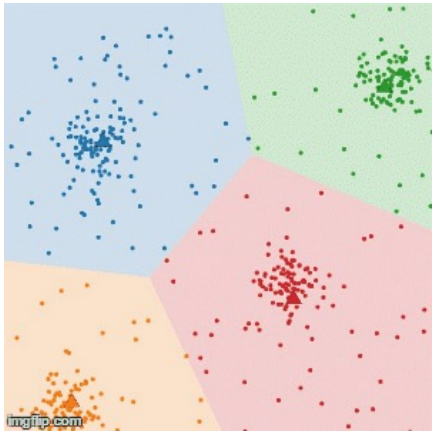
# CS 441: Engineering Distributed Objects For Cloud Computing

## *Lecture 15: Apache Spark: A Cloud Engine For Big Data Processing*

Instructor: Dr. Mark Grechanik
University of Illinois at Chicago

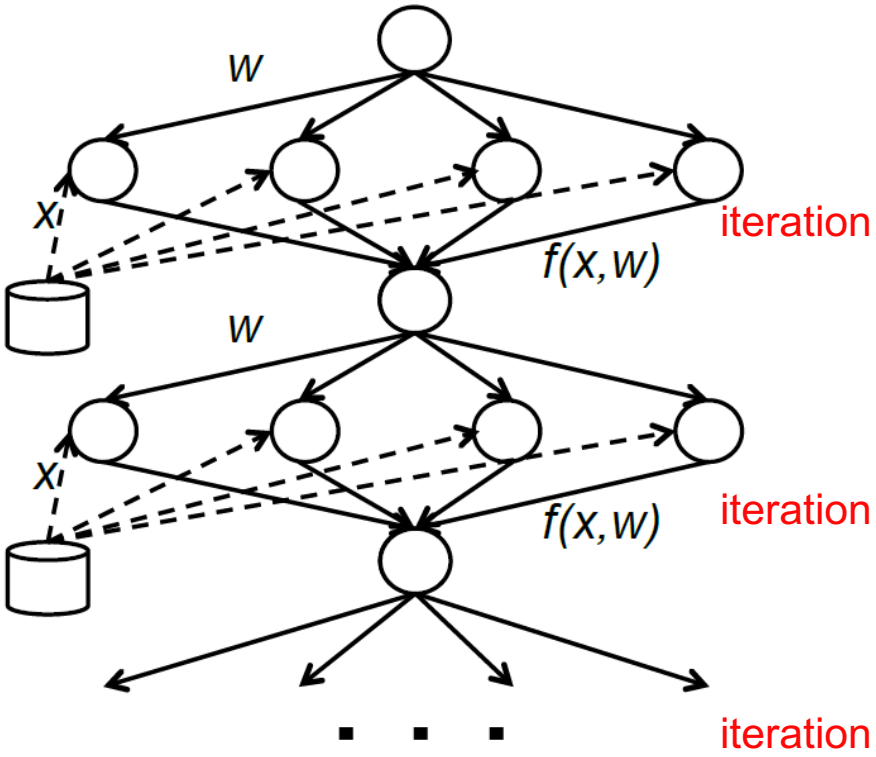
# K-Means Clustering

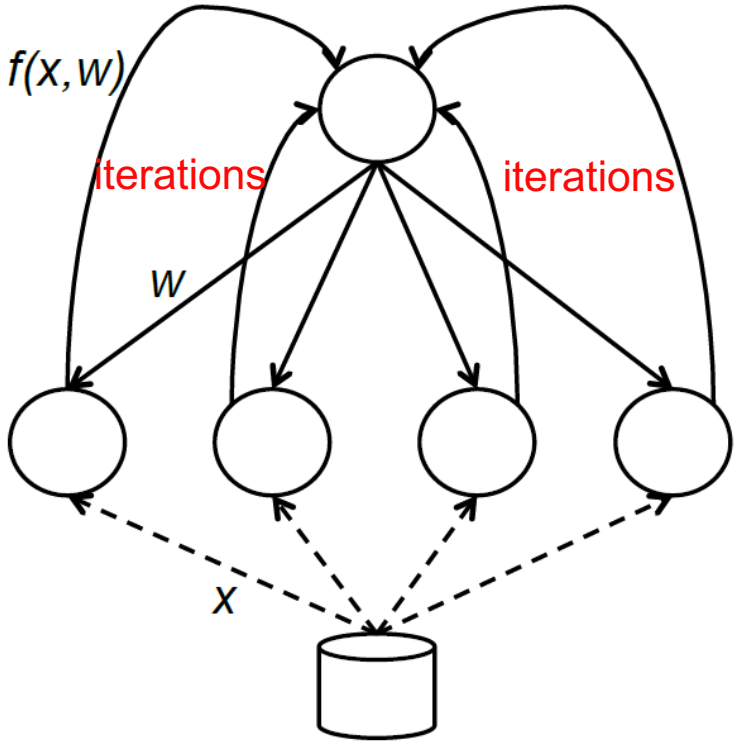


1. Choose the number of clusters, K
2. Compute the centroids c1, c2, ..... ck randomly
3. Repeat
    1. for each data point p:
        1. find the nearest centroid(c1, c2 ... ck)
        2. assign the point to that cluster
    2. for each cluster j = 1..k
        1. compute new centroid = mean of all points assigned to that cluster
4. Until the convergence measure is reached or until the predefined number of iterations is reached

**How will you implement this algorithm in map/reduce?**

# Example of Logistic Regression Task




**Map/Reduce** **Spark**

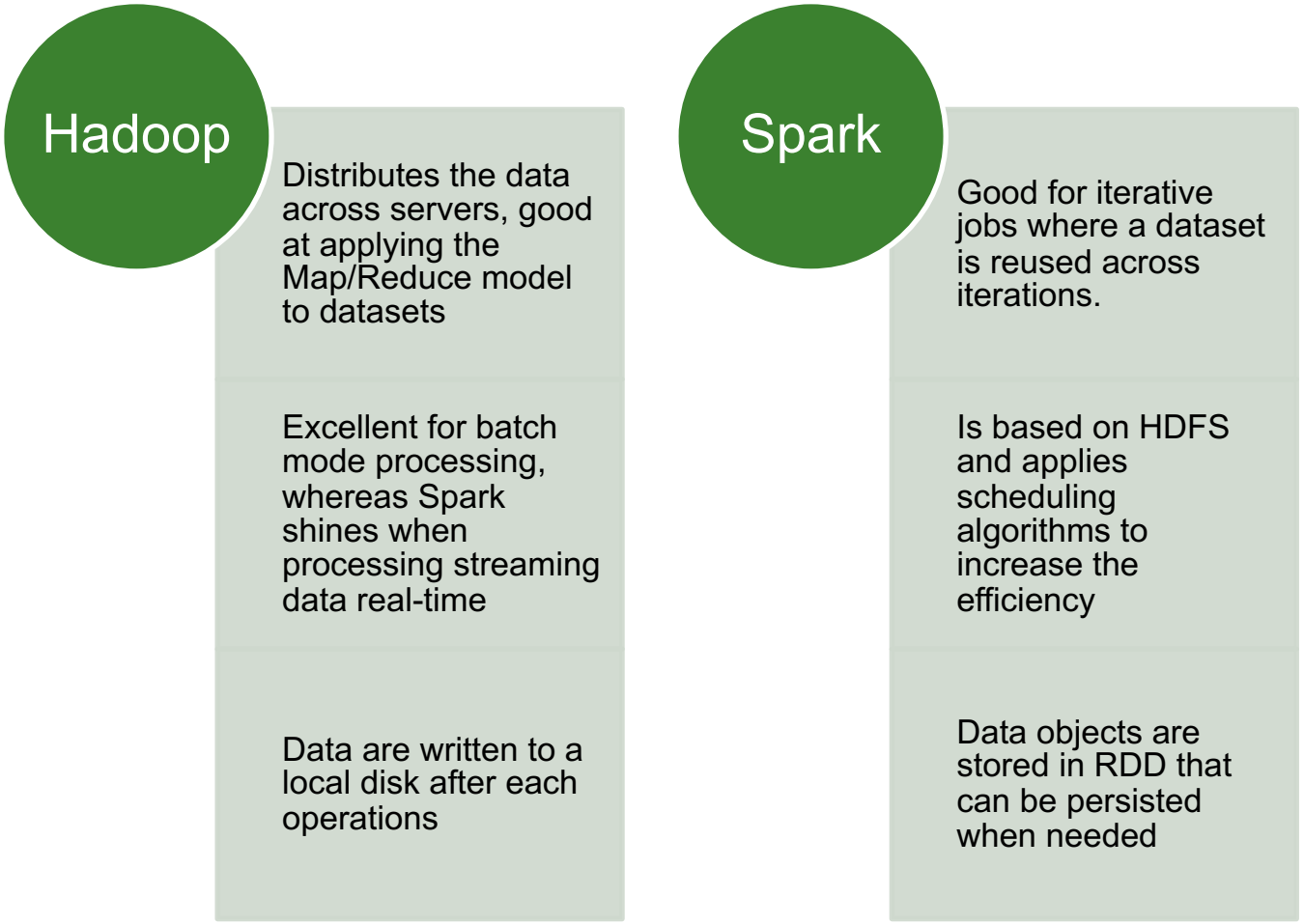
# What Sparked Spark?

Spark extends MapReduce with a data-sharing abstraction called Resilient Distributed Datasets (RDDs) and parallel operations of RDDs.

With RDD, Spark unifies multiple processing mechanisms, e.g, SQL, streaming, machine learning, graph processing - workloads.

With Spark, MapReduce can implement various distributed computations for different workloads using uniform APIs.

# Hadoop vs Spark



# Preview of the Programming Model

- The key programming abstraction in Spark is RDDs, which are fault-tolerant collections of objects partitioned across a cluster that can be manipulated in parallel.
- Users create RDDs by applying operations called "transformations" (such as map, filter, and groupBy) to their data.

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(s => s.startsWith("ERROR")) //ephemeral
val count = errors.map( _ => 1).reduce( _+_ )
println("Total errors: "+count())

errors.persist() //errors.cache()
errors.filter(s => s.contains("MySQL")).count()
// Fetch back the time fields of errors that
// mention PHP, assuming time is field #3:
errors.filter(s => s.contains("PHP")).map(line => line.split('\t')(3)).collect()
```

# RDDs

## RDD is a read-only collection of objects

- Partitioned across a set of machines.
- Can be easily rebuilt if a partition is lost.
- Each RDD is represented as a Scala object in Spark.

## Parallel operations can be performed on RDDs

- reduce combines dataset elements using an associative function.
- collect sends all elements to the driver program.
- foreach passes each element through a user-provided function.
- cache operation leaves the dataset lazy.
- save operation evaluated the dataset and writes it to a distributed filesystem.

## Shared variables for supporting usage patterns

- Broadcast variable objects wrap values and ensure that these values are copied to each worker once.
- Accumulator variables are used by workers that add values to using associative operations and that only the driver program can read.

# Constructing RDDs

- From a file in a shared file system, such as the Hadoop Distributed File System (HDFS)
- By dividing a Scala collection into a number of slices that will be sent to multiple nodes.
- By transforming an existing RDD using monadic operations like flatMap
- By changing the persistence of RDDs, which by default are lazy and ephemeral

# Some Optimizations Are Possible

- Each RDD represents a “logical plan” to compute a dataset, but Spark waits until certain output operations, such as count, to launch a computation.
- The engine can do some simple query optimization, such as pipelining operations.
  - Spark will pipeline reading lines from the HDFS file with applying the filter and computing a running count, so that it never needs to materialize the intermediate lines and errors results.
- Spark does not “understand” the structure of the data in RDDs or the semantics of user functions.

# Recalling flatMap and foldLeft

- **def flatMap[A,B](f: A => T[B]): T[B]**
- val strings = Seq("1", "2", "foo", "3", "bar", " 4", "5")
- def g(v:Int) = List(v-1, v, v+1)
- val list = strings.flatMap(toInt).flatMap(_ =>g(_))
- list: List[Int] = List(0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6)
- **def foldLeft[A,B](z: B)(f: (B, A) => B): B**
- val count = list.foldLeft(0)((sum,_) => sum + 1)
- val avg = list.foldLeft(0.0)(_+_)/list.foldLeft(0)((r,c)=> r+1)
- val last = list.foldLeft(list.head)((_, c) => c)
- val noduplicates = list.foldLeft(List[Int]()) {
  (r,c) =>if (r.contains(c)) r else c :: r }.reverse

# More Spark Examples

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "YOUR_SPARK_HOME/README.md" // Should be some
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    sc.stop()
  }
}
```

```scala
import org.apache.spark._
import org.apache.spark.rdd.RDD

object BasicAvg {
  def main(args: Array[String]) {
    val master = args.length match {
      case x: Int if x > 0 => args(0)
      case _ => "local"
    }
    val sc = new SparkContext(master, "BasicAvg", System.getenv("SPARK_HOME"))
    val input = sc.parallelize(List(1,2,3,4))
    val result = computeAvg(input)
    val avg = result._1 / result._2.toFloat
    println(result)
  }
  def computeAvg(input: RDD[Int]) = {
    input.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1),
      (x,y) => (x._1 + y._1, x._2 + y._2))
  }
}
```

https://github.com/databricks/learning-spark

https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples

# Spark's Internals

- Spark is built on top of Mesos, a "cluster operating system" that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster.
- The core of Spark is the implementation of resilient distributed datasets (RDDs).
- When a parallel operation is invoked on a dataset, Spark creates a task to process each partition of the dataset and sends these tasks to worker nodes using delay scheduling.
- Each RDD object implements the same simple interface, which consists of three operations: getPartitions, getIterator(partition), getPreferredLocations(partition)
- Once launched on a worker, each task calls getIterator to start reading its partition.
- Shipping tasks to workers requires shipping closures to them—both the closures used to define a distributed dataset, and closures passed to operations such as reduce. Since Scala closures are Java objects and can be serialized using Java serialization; this is a feature of Scala that makes it relatively straightforward to send a computation to another machine.
- Broadcast variables and accumulators, are implemented using classes with custom serialization formats.
- Homework: read how Spark is integrated with the Scala interpreter.

# Delay Scheduling

Achieve locality and fairness in cluster scheduling

- Many jobs are short and they get stuck in queues behind long-running jobs
- Queueing policies assign jobs to nodes that may not have necessary data

Relax queuing policy for a short period of time to achieve data locality: waiting is worth it if E(waittime) < cost to run non-locally

- Short waiting times are enough to get nearly 100% locality
- Scan jobs using some order and pick one that is permitted
  - If wait < T1, only allow node-local tasks
  - If T1 < wait < T2, allow rack-local
  - If wait > T2, allow off-rack

# More Insight Into Delayed Scheduling

- The cost of the expected wait time should be less that the cost of running non-local jobs
- We model task arrivals using the Poisson process
- Expected response time gain, E[gain] = (1-exp(-w/t))(d-t)
  - w is the wait time
  - t is the expected time to get local scheduling opportunity
  - d >= average task length is the delay from running a nonlocal job
  - t = (average task length)/(file replication x tasks per node)
- Sufficient fraction of tasks should be short, and
- There should be many locations where tasks can run efficiently with blocks replicated across nodes.

# Spark Is Built on Mesos

## Multiplex a cluster among multiple frameworks

- Improve utilization, share access to large datasets
- Replicating these datasets across clusters is costly
- Partitioning clusters for one-to-one partition/framework or allocating a set of VMs to each framework do not achieve high utilization and efficient data sharing

## Mismatch between the allocation granularities of these solutions and of existing frameworks.

- Many frameworks, such as Hadoop and Dryad, employ a fine-grained resource sharing model, where nodes are subdivided into “slots” and jobs are composed of short tasks that are matched to slots.
- Many frameworks are developed independently, and there is no way to perform fine-grained sharing across frameworks, making it difficult to share clusters and data efficiently between them.
- Longer tasks make it difficult to achieve high utilization across clusters.

## Mesos is a thin resource-sharing layer

- Enables fine-grained sharing across diverse cluster computing frameworks by giving them a common interface for accessing cluster resources.
- Support current and future frameworks that have different scheduling needs.
- Scheduler must scale to clusters of tens of thousands of nodes running hundreds of jobs with millions of tasks.
- Mesos is fault-tolerant and highly available.

# Mesos Abstraction

- Resource offer abstraction encapsulates a bundle of resources that a framework can allocate on a cluster node to run tasks.
- Mesos decides how many resources to offer each framework, based on an organizational policy such as fair sharing, while frameworks decide which resources to accept and which tasks to run on them.
- Resource offers are simple and efficient to implement, allowing Mesos to be highly scalable and robust to failures.

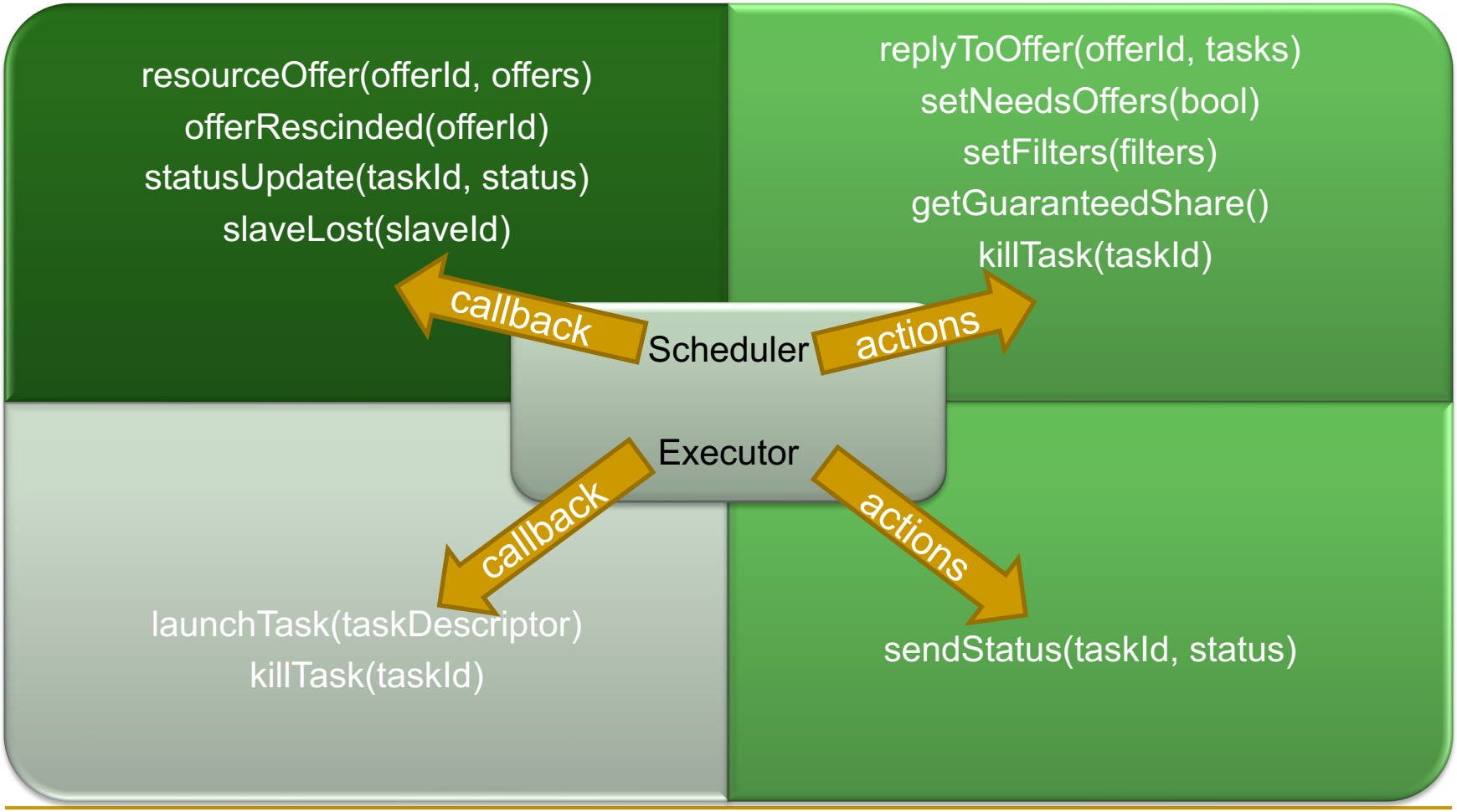
# Mesos Design Philosophy

Define a minimal interface that enables efficient resource sharing across frameworks

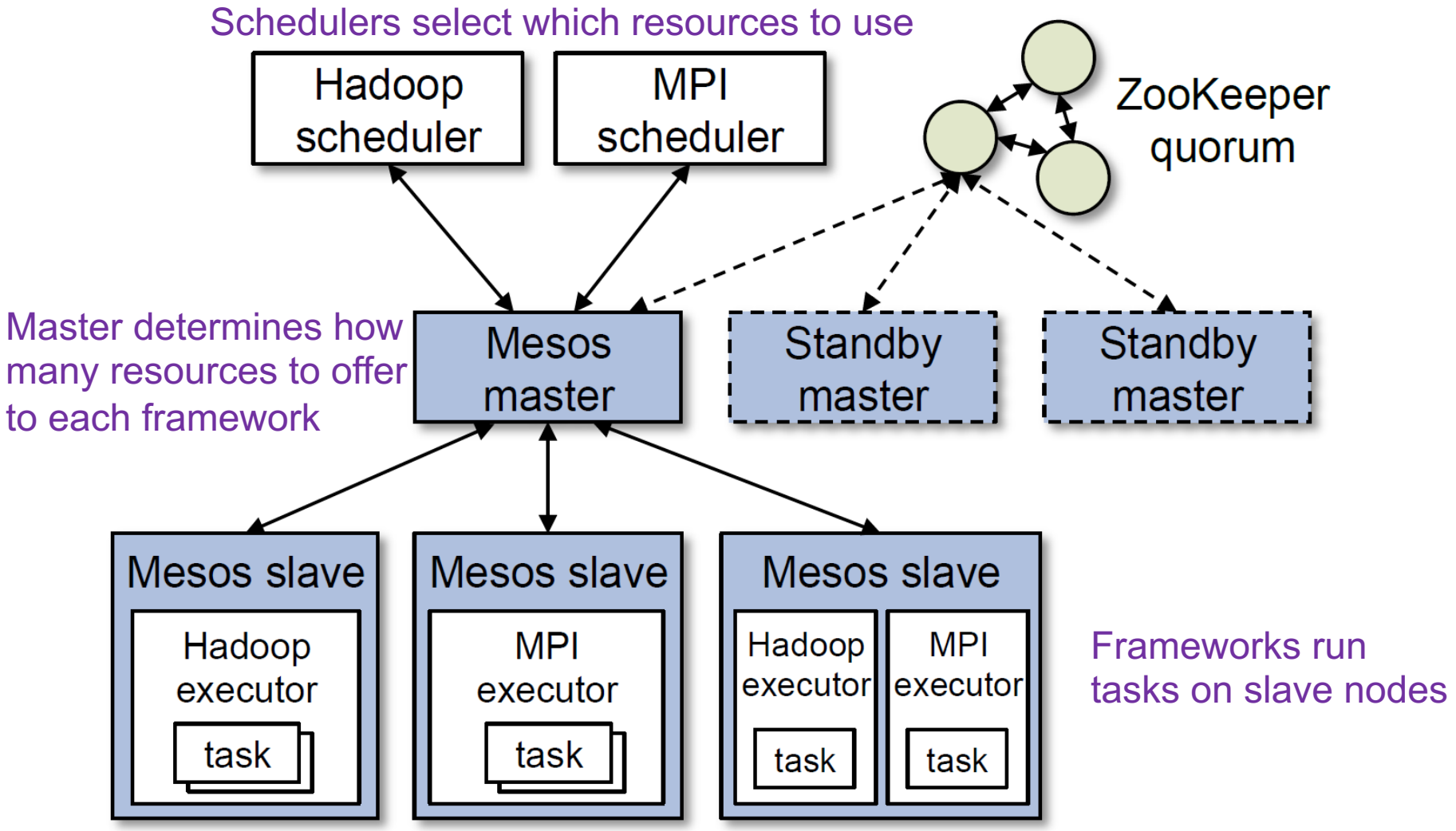
Push control of task scheduling and execution to the frameworks

Frameworks can implement diverse approaches to various problems in the cluster and evolve independently

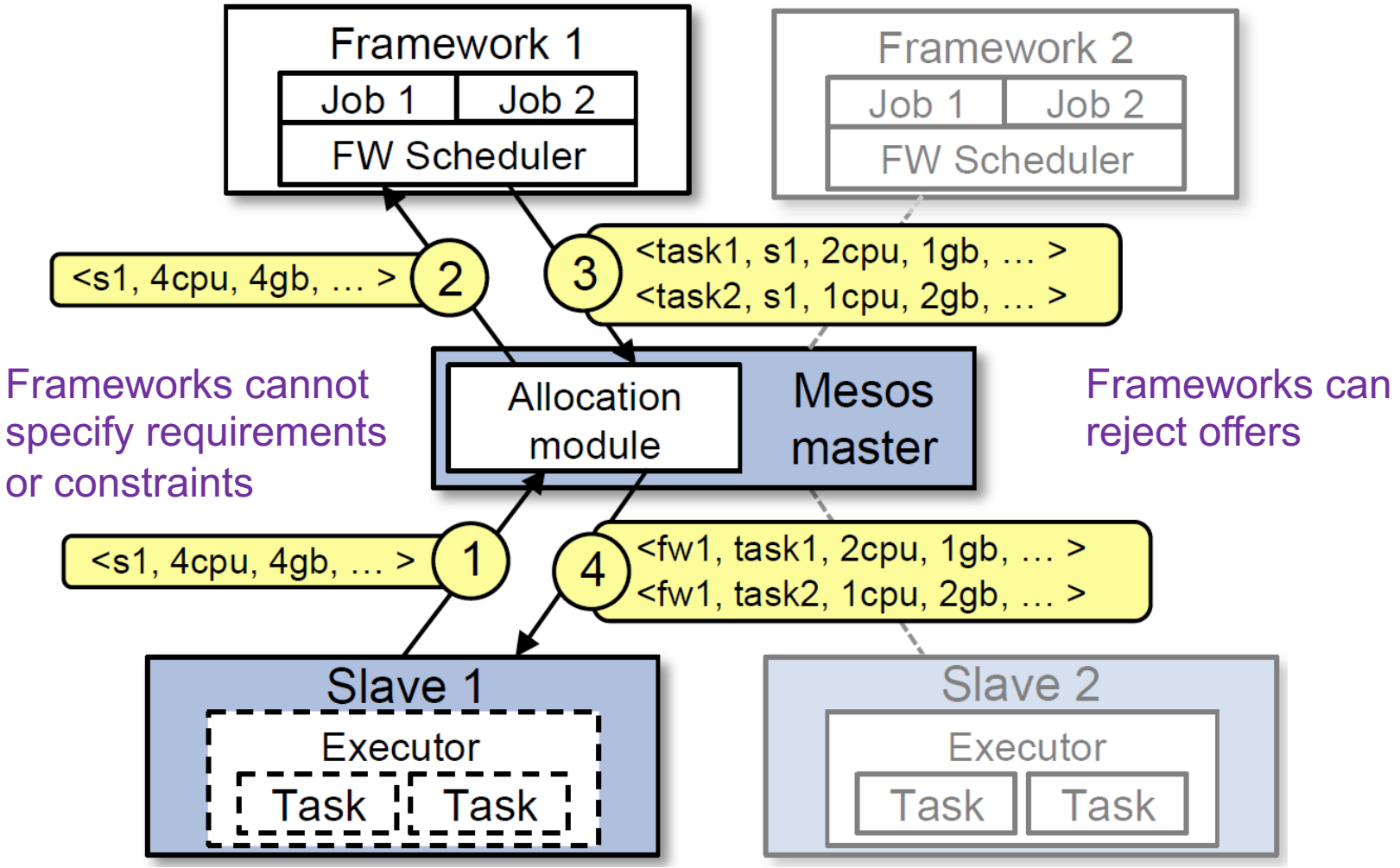
# Mesos API


resourceOffer(offerId, offers)
offerRescinded(offerId)
statusUpdate(taskId, status)
slaveLost(slaveId)
replyToOffer(offerId, tasks)
setNeedsOffers(bool)
setFilters(filters)
getGuaranteedShare()
killTask(taskId)
callback
Scheduler
actions
Executor
callback
actions
launchTask(taskDescriptor)
killTask(taskId)
sendStatus(taskId, status)

# Architecture of Mesos



# Resource Offer Example



# Scaling Resource Offers

- Because some frameworks will always reject certain resources, Mesos lets them short-circuit the rejection process and avoid communication by providing filters to the master.
  - only offer nodes from list L
  - only offer nodes with at least R resources free
- Because a framework may take time to respond to an offer, Mesos counts resources offered to a framework towards its allocation of the cluster.
- If a framework has not responded to an offer for a sufficiently long time, Mesos rescinds the offer and re-offers the resources to other frameworks.

# Mesos Implementation

## 10,000LOC of C++

- The system runs on Linux, Solaris and OS X, and supports frameworks written in C++, Java, and Python
- C++ library is used called libprocess that provides
- an actor-based programming model using efficient asynchronous I/O mechanisms
- ZooKeeper is used for electing a leader

## Four frameworks are ported to run on Mesos

- Hadoop, Torque, MPICH2 of the MPI
- Spark

# Spark SQL

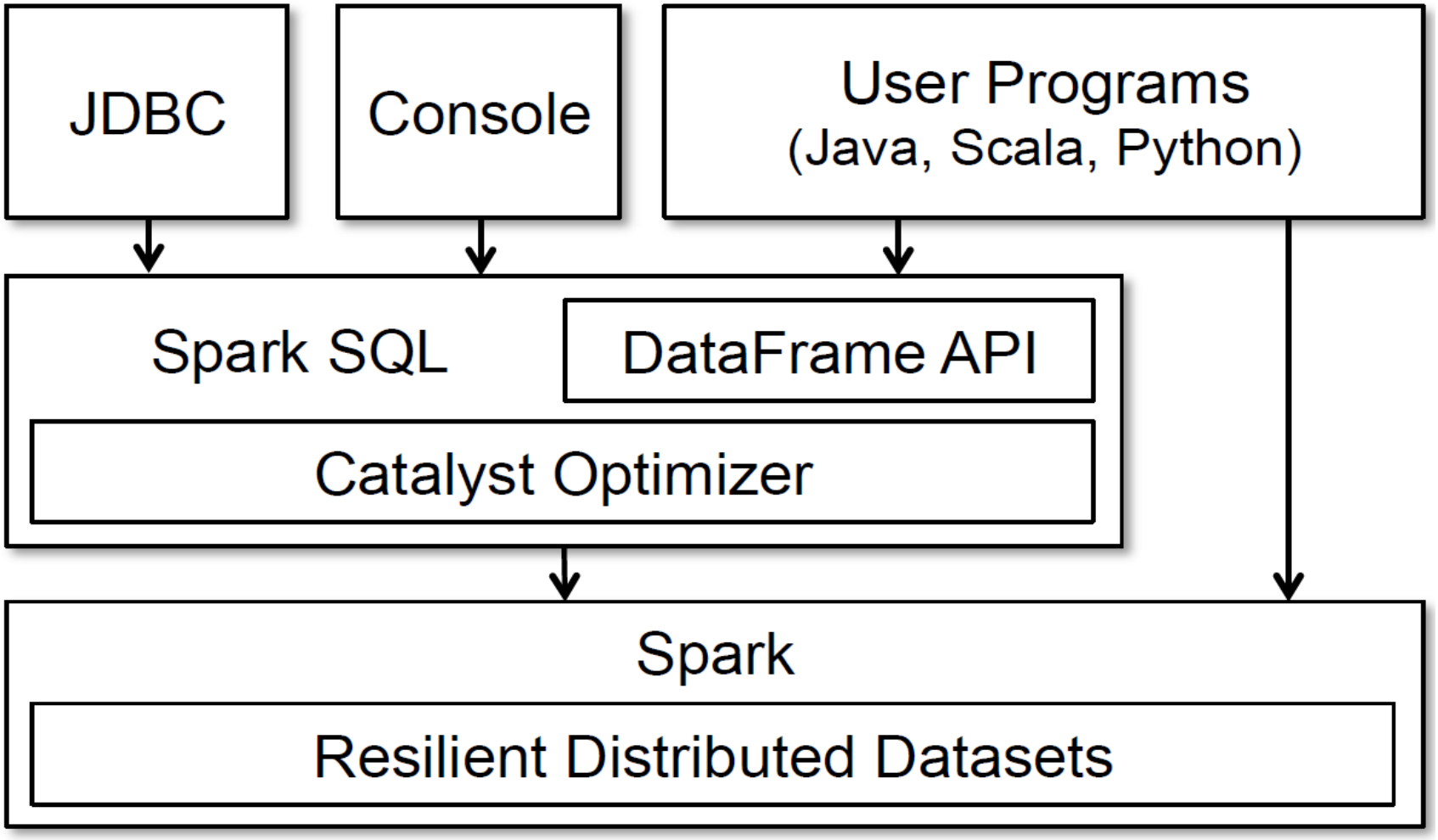
Integrate relational processing with Spark functional programming

- SQL users can call Spark analytics libraries
- Spark programmers can access relational processing of datasets through DataFrame API

Evolution of the core Spark API

- a large Internet company uses Spark SQL to build data pipelines and run queries on an 8000-node cluster with over 100 PB of data
- Each individual query regularly operates on tens of terabytes.

# Interfaces to Spark SQL

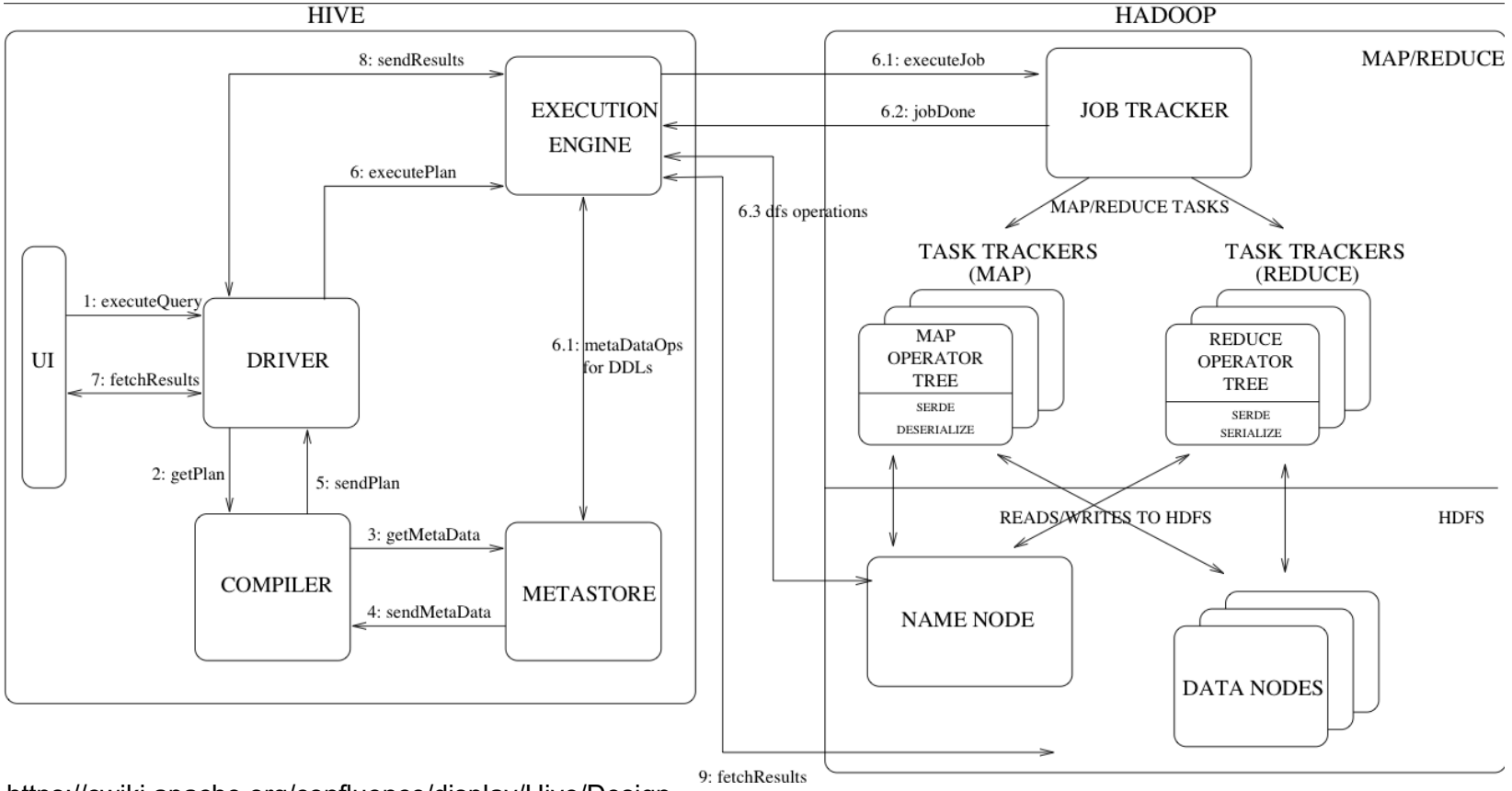


# DataFrame Examples

- DataFrame is an abstraction for a distributed collection of rows with the same schema
- ctx = new HiveContext()
  **users** = ctx.table("users")
  **young** = users.where(users("age") < 21)
  println(young.count())
- **employees**
  .join(dept , **employees**("deptId") === dept("id"))
  .where(**employees**("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
- **users**.where(**users**("age") < 21).registerTempTable("young")
  ctx.sql("SELECT count(*), avg(age) FROM young")

# Apache Hive

Spark is integrated with Hive, a data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis




https://cwiki.apache.org/confluence/display/Hive/Design

# Catalyst Optimizer

Catalyst contains a general library for representing trees and applying rules to manipulate them

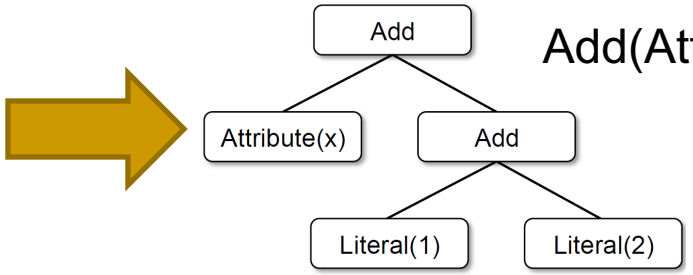
- libraries are built for relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, and physical planning

Code is generated automatically and parts of queries are compiled to Java bytecode

- Scala feature, quasiquotes is used to make it easy to generate code at runtime from composable expressions
- https://infoscience.epfl.ch/record/185242

# Rules and Transformations

x + (1 + 2)



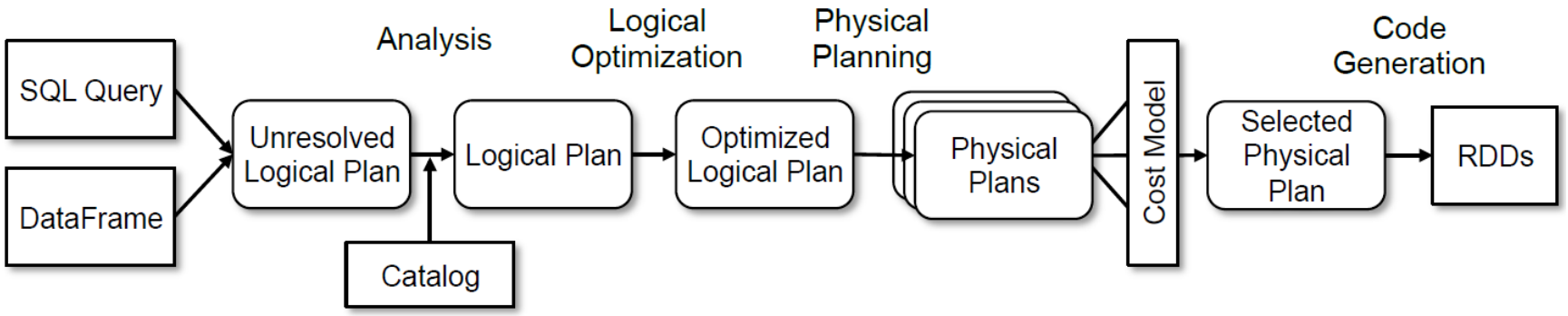

Add(Attribute(x), Add(Literal(1), Literal(2)))

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

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.

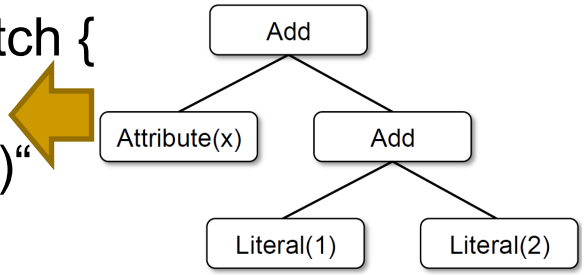
# Query Planning in Spark SQL

- Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources to resolve attributes.
- 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.
- Catalyst transforms 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.



# Code Generation in Spark

- With quasiquotes, generate code with performance similar to hand-tuned programs
  - 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)}“ }




- 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.

# Conclusions

- Spark is widely used in different industries for fast parallel data processing
- Fine-tuning Spark is very difficult and it requires deep understanding of Spark's internals and the data dependencies
- Often, using a few powerful machines with Hadoop does the job that would require more effort with Spark
- Spark requires a lot of memory and computing resources, and it is great for processing streaming data in iterative computations
- and… that's all, folks!

# Reading

- **Mandatory**
  - Option 1: Textbook, chapter 13
- **Mandatory for Option 2**
  - http://cacm.acm.org/magazines/2016/11/209116-apache-spark/fulltext
  - Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10). USENIX Association, Berkeley, CA, USA, 10-10.
  - Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11). USENIX Association, Berkeley, CA, USA, 295-308.
- **Optional**
  - Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). ACM, New York, NY, USA, 1383-1394. DOI: http://dx.doi.org/10.1145/2723372.2742797
  - Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems* (EuroSys '10). ACM, New York, NY, USA, 265-278. DOI=http://dx.doi.org/10.1145/1755913.1755940
  - https://en.wikipedia.org/wiki/Apache_Spark