Spark高级编程（III）

摘要

- Spark MLlib
- GraphX

摘要

- Spark MLlib
- GraphX

Spark MLlib

MLlib is Apache Spark's scalable machine learning library.

Ease of Use

Usable in Java, Scala, Python, and R.

MLlib fits into Spark's APIs and interoperates with NumPy in Python (as of Spark 0.9) and R libraries (as of Spark 1.5). You can use any Hadoop data source (e.g. HDFS, HBase, or local files), making it easy to plug into Hadoop workflows.

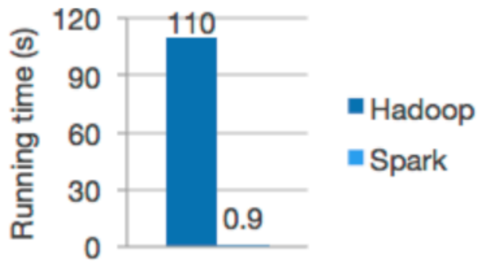
```
data = spark.read.format("libsvm")\
  .load("hdfs://...")

model = KMeans(k=10).fit(data)
```

Calling MLlib in Python

Performance

High-quality algorithms, 100x faster than MapReduce.

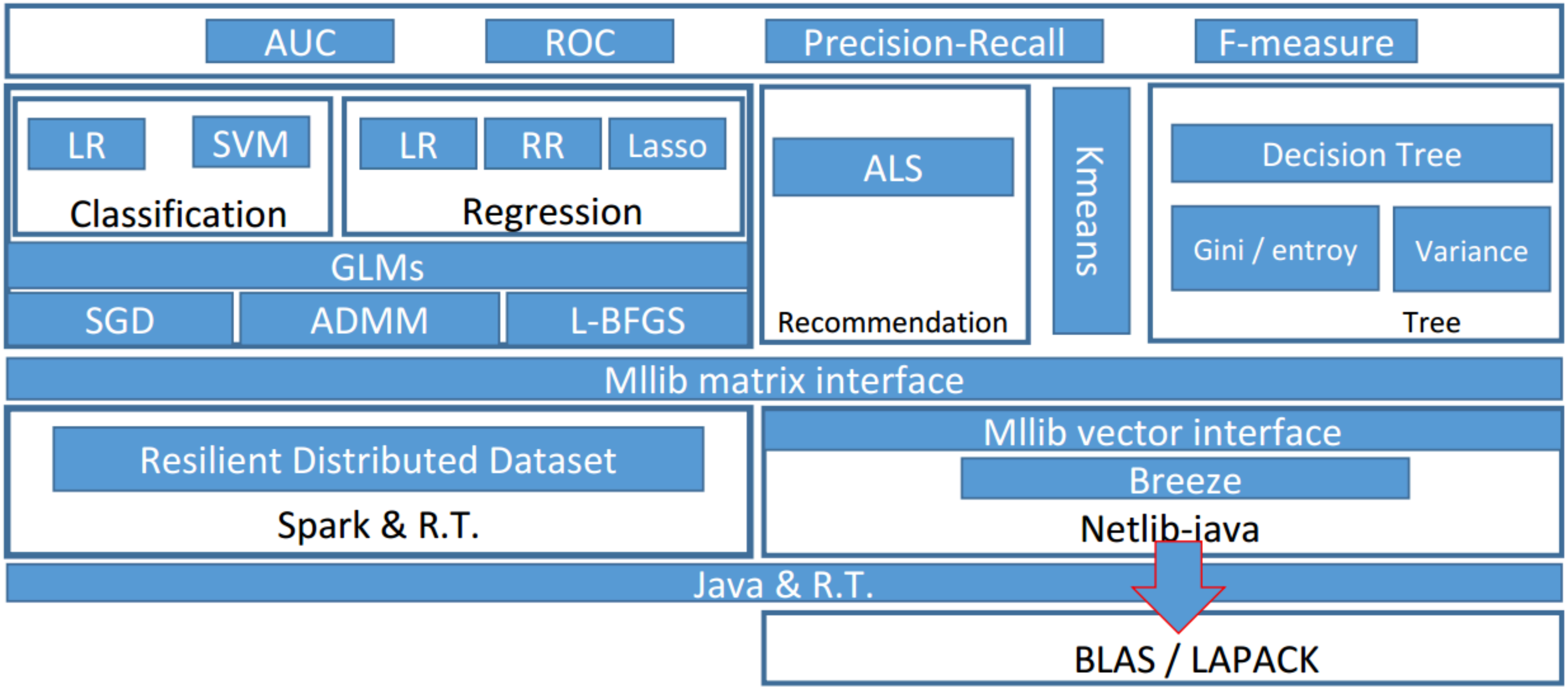
Spark excels at iterative computation, enabling MLlib to run fast. At the same time, we care about algorithmic performance: MLlib contains high-quality algorithms that leverage iteration, and can yield better results than the one-pass approximations sometimes used on MapReduce.



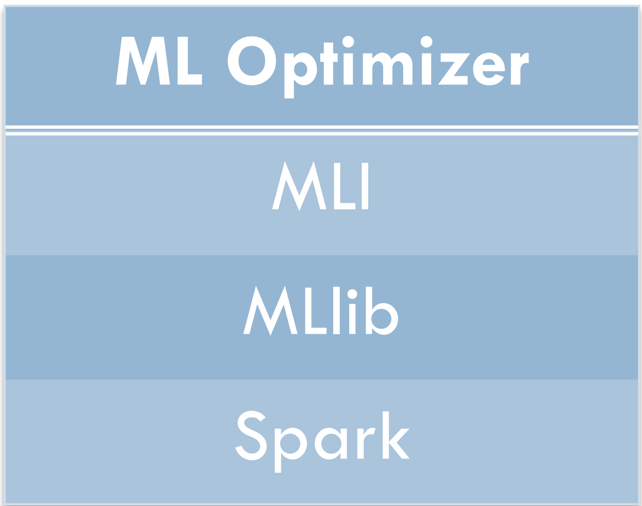
Logistic regression in Hadoop and Spark

架构

- spark.mllib.* // not deprecated
- spark.ml.* // MLlib DataFrame-based API



架构



MLBase的分层结构

- MLlib是常用机器学习算法的实现库
- MLI是进行特征抽取和高级ML编程抽象的算法实现的API
- ML Optimizer优化器会选择最合适的，已经实现好了的机器学习算法和相关参数

例子

- 训练分类器

```
//构造一个10行10列的数组
val data = Array.ofDim[Int](10,10)
for (i <- 0 until 10){
      for ( j <- 0 until 10){
          //给数组赋值随机数
          data(i)(j) = scala.util.Random.nextInt(100)
      }
      //取第2～10列数据（训练集的样本特征空间）
      x = data[, 2 to 10]
     //取第1列数据（样本相应的分类标签）
      y = data[, 1]
     //调用分类算法进行分类（MLBase自动选择优化方案）
      model = do_classify(y,x)
```

设计理念

- MLlib：把数据以RDD的形式表示，然后在分布式数据集上调用各种算法。引入一些数据类型（比如点和向量），给出一系列可供调用的函数的集合。
- MLlib只包含能够在集群上运行良好的并行算法
  - 特征提取，例如TF-IDF
  - 统计
  - 分类与回归：线性回归，逻辑回归，SVM，朴素贝叶斯，决策树与随机森林
  - 聚类
  - 协同过滤与推荐
  - 降维
  - 模型评估

MLlib数据类型

- 本地向量
- 标记点
- 本地矩阵
- 分布式矩阵
- 行矩阵
- 索引矩阵
- 三元组矩阵

本地向量

- 本地向量存储在单机上，由从0开始的Int型的索引和Double型的值组成，存储在单机上。
- MLlib支持两种类型的本地向量：密集向量和稀疏向量。密集向量的值由Double型的数据表示，而稀疏向量由两个并列的索引和值表示。

```
//导入MLlib
import org.apache.spark.mllib.linalg.{Vector, Vectors}
//创建（1.0, 0.0, 3.0）的密集向量
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
//通过指定非零向量的索引和值，创建(1.0, 0.0, 3.0)的数组类型的稀疏向量
val sv1: Vector = Vectors.sparse(3, Array(0,2), Array(1.0, 3.0))
//通过指定非零向量的索引和值，创建(1.0, 0.0, 3.0)的序列化的稀疏向量
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

标记点

- 标记点是由一个本地向量（密集或稀疏）和一个标签（Int型或Double型）组成。在MLlib中，标记点主要被应用于回归和分类这样的监督学习算法中。标签通常采用Int型或Double型的数据存储格式。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
//通过一个正相关的标签和一个密集的特征向量创建一个标记点
val  pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
//通过一个负向标签和一个稀疏特征向量创建一个标记点
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0,2), Array(1.0, 3.0)))
```

稀疏数据

- MLlib可以读取存储为LIBSVM格式的数据，其每一行代表一个带有标签的稀疏特征向量。格式如下：

  label index1:value1 index2:value2 ...

- 其中label是标签值，index是索引，其值从1开始递增。加载完成后，索引被转换为从0开始。
- 接口：MLUtils.loadLibSVMFile

  val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "data/MLlib/sample_libsvm_data.txt")

本地矩阵

- 本地矩阵是由（Int类型行索引，Int类型列索引，Double类型值）组成，存放在单机中。Mllib支持密集矩阵，密集矩阵的值以列优先方式存储在一个Double类型的数组中，矩阵如下：

$$\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{bmatrix} \qquad \begin{bmatrix} 9.0 & 0.0 \\ 0.0 & 8.0 \\ 0.0 & 6.0 \end{bmatrix}$$

- 这个3行2列的矩阵存储在一个一维数组[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]中。
- MLlib实现：DenseMatrix

```
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))
```

分布式矩阵

- 分布式矩阵由（Long类型行索引，Long类型列索引，Double类型值）组成，分布存储在一个或多个RDD中。因为要缓存矩阵的大小，所以分布式矩阵底层的RDD必须是确定的，选择正确的格式来存储巨大的分布式矩阵是非常重要的，否则会导致错误的出现。MLlib已实现了四种分布式矩阵：
  - 行矩阵 RowMatrix
  - 行索引矩阵 IndexedRowMatrix
  - 三元组矩阵 CoordinateMatrix
  - 块矩阵 BlockMatrix

MLlib的算法库

- 基本统计
  - 汇总统计，相关性统计，分层抽样，假设检验，随机数据生成，核密度估计
- 分类和回归
  - 线性模型（支持向量机SVM、逻辑回归、线性回归）
  - 朴素贝叶斯
  - 决策树，随机森林和梯度提升决策树（GBT）
- 协同过滤
  - 交替最小二乘法（ALS）
- 聚类
  - K-means，高斯混合，快速迭代聚类，三层贝叶斯概率模型，流式K-means

MLlib的算法库

- 降维
  - 奇异值分解（SVD）
  - 主成分分析（PCA）
- 频繁模式挖掘
  - FP-growth，关联规则，PrefixSpan
- 优化器
  - 随机梯度下降
  - 限制内存BFGS（L-BFGS）
- 特征值提取和转换，评价指标，PMML模型输出等算法实现

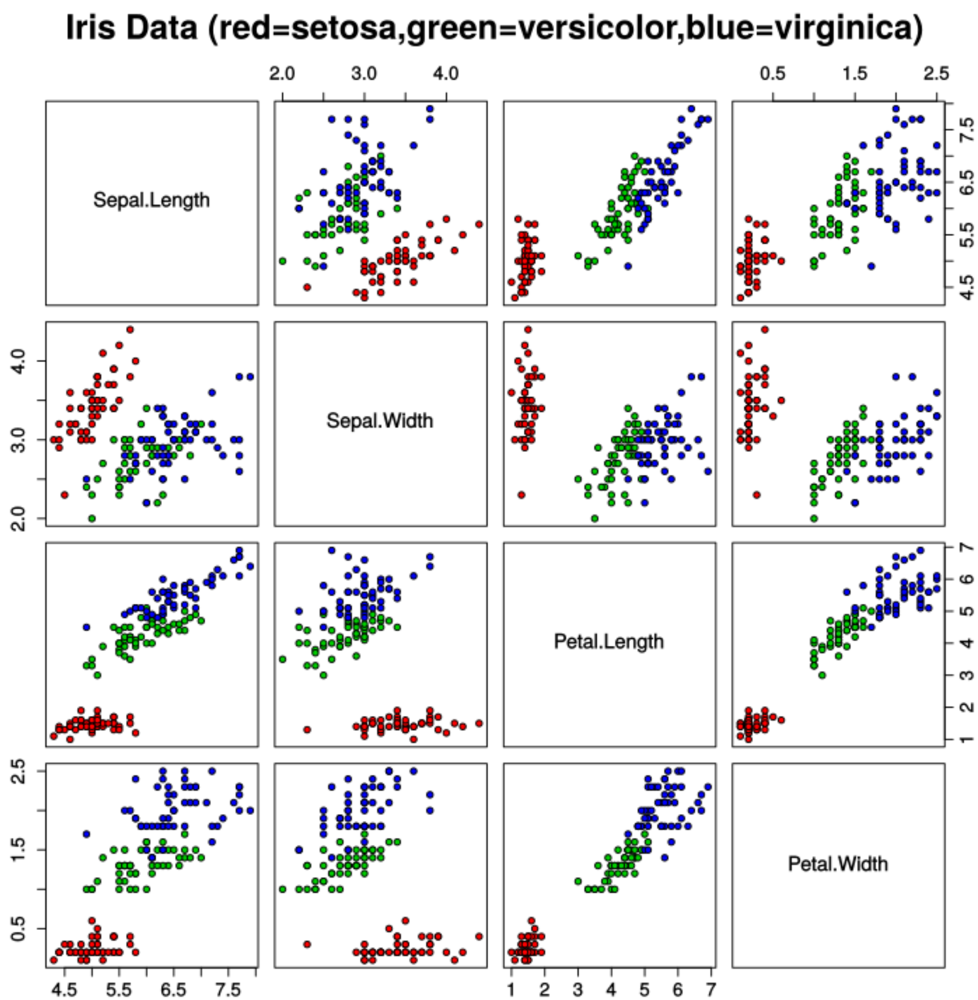
常见步骤

- 例如，如果要用MLlib来完成文本分类的任务，只需如下操作：
  - 首先用字符串RDD来表示你的消息
  - 运行MLlib的一个特征提取算法来把文本数据转换为数值特征，该操作会返回一个向量RDD
  - 对向量RDD调用分类算法（比如逻辑回归），这步会返回一个模型对象，可以使用该对象对新的数据点进行分类
  - 使用MLlib的评估函数在测试数据集上评估模型

再看K-Means


```
import org.apache.spark.mllib.clustering.{KMeans, KMeansModel}
import org.apache.spark.mllib.linalg.Vectors

val data = sc.textFile("data/mllib/kmeans_data.txt")
val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()
// Cluster the data into two classes using KMeans
val numClusters = 2
val numIterations = 20
val clusters = KMeans.train(parsedData, numClusters, numIterations)
// Evaluate clustering by computing Within Set Sum of Squared Errors
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
// Save and load model
clusters.save(sc, "target/org/apache/spark/KMeansExample/KMeansModel")
val sameModel = KMeansModel.load(sc, "target/org/apache/spark/KMeansExample/KMeansModel")
```

Iris数据集分类


Iris Data (red=setosa,green=versicolor,blue=virginica)
Sepal.Length
Sepal.Width
Petal.Length
Petal.Width
2.0 3.0 4.0
0.5 1.5 2.5
4.5 5.5 6.5 7.5
1 2 3 4 5 6 7
2.0 3.0 4.0
0.5 1.5 2.5
4.5 5.5 6.5 7.5
1 2 3 4 5 6 7

实验步骤：数据处理

- 首先需要将Iris-setosa，Iris-versicolour，Iris-virginica转化成0，1，2来表示。生成LabeledPoint类型RDD
  - 利用*loadLibSVMFile*接口从LibSVM格式的文件读取数据。当然首先需要把原始的数据文件转换成LibSVM格式，然后调用*loadLibSVMFile*接口就可以生成LabeledPoint类型的RDD。
  - 先用textFile 读取数据，然后对string类型的RDD调用map操作，转换成LabeledPoint类型的RDD。

实验步骤：数据处理

读取数据

```
val rdd: RDD[String] = sc.textFile(path)
```

转换得到LabeledPoint

```
var rddLp: RDD[LabeledPoint] = rdd.map( x => { val strings: Array[String] = x.split(",")
regression.LabeledPoint( strings(4) match { case "Iris-setosa" => 0.0 case "Iris-
versicolor" => 1.0 case "Iris-virginica" => 2.0 } , Vectors.dense( strings(0).toDouble,
strings(1).toDouble, strings(2).toDouble, strings(3).toDouble)) } )
```

分割数据集为训练集和测试集

```
val Array(trainData,testData): Array[RDD[LabeledPoint]] =
rddLp.randomSplit(Array(0.8,0.2))
```

实验步骤：训练模型及模型评估

- 选取朴素贝叶斯，决策树，随机森林，支持向量机，以及logistics回归共5种分类算法。采用留出法对建模结果评估，留出30%数据作为测试集，评估标准采用精度accuracy。
- 支持向量机（SVM），logistics回归是二分类的算法，由于本数据集有多个类别，所以可以利用多个二分类分类器来实现多分类目标。

参考代码：决策树

Scala

#构建模型

```
val decisonModel: DecisionTreeModel =
DecisionTree.trainClassifier(trainData,3, Map[Int, Int](),"gini",8,16)
```

得到测试集预测的结果

```
val result: RDD[(Double, Double)] = testData.map( x=> { val pre: Double =
decisonModel.predict(x.features) (x.label,pre) } )
val acc: Double = result.filter(x=>x._1==x._2).count().toDouble /result.count()
```

参考代码：朴素贝叶斯

Python

```
# 分割数据集为训练集和测试集
traindata,testdata = data.randomSplit([0.7,0.3])
# 朴素贝叶斯训练并评估
Bayesmodel = NaiveBayes.train(traindata,1.0)
predictionAndLabel_Bayes = testdata.map(lambda
        p :(Bayesmodel.predict(p.features),p.label))
accuracy= 1.0*predictionAndLabel_Bayes.filter(lambda p1:
        p1[0]==p1[1]).count()/testdata.count()
```

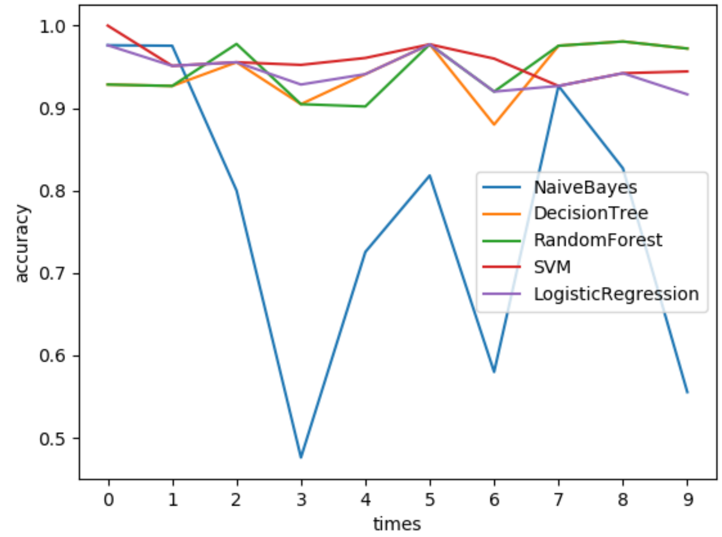
参考代码：SVM

Python

```
#用多个SVM分类器实现多分类
model1 = SVMWithSGD.train(train0_1, iterations=1000)
model2 =SVMWithSGD.train(train0_2,iterations=1000)
model3 = SVMWithSGD.train(train1_2,iterations=1000)
predictions1 = model1.predict(testdata.map(lambda x :x.features))
predictions2 = model2.predict(testdata.map(lambda x :x.features))
predictions3 = model3.predict(testdata.map(lambda x :x.features))
true_label = testdata.map(lambda x :x.label).collect()
label_list1=predictions1.collect() ;
label_list2=predictions2.collect();
label_list3=predictions3.collect()
#投票产生结果
predict_label =[]
account =0
```

参考代码：SVM


```
for index in range(len(true_label)):
    dictionary ={0.0:0,1.0:0,2.0:0}
    if label_list1[index] ==0:
        dictionary[0.0] +=1
    else: dictionary[1.0]+=1
    if label_list2[index] ==0:
        dictionary[0.0]+=1
    else: dictionary[2.0] +=1
    if label_list3[index] ==0:
        dictionary[2.0]+=1
    else: dictionary[1.0]+=1
    maxlabel = 0.0
for item in dictionary.keys():
    if dictionary[item]>dictionary[maxlabel]:
        maxlabel =item
    if maxlabel==true_label[index]:
        account+=1
    predict_label.append(maxlabel)
accuracy_SVM =1.0*account/len(true_label)
```



ML库

- Spark的ML库基于DataFrame提供高性能的API，帮助用户创建和优化实用的机器学习流水线（Pipeline），包括特征转换独有的Pipelines API。相比较Mllib，变化主要体现在：
  - 从机器学习的library开始转向构建一个机器学习工作流的系统。ML把整个机器学习的过程抽象成Pipeline，一个Pipeline由多个Stage组成，每个Stage由Transformer或者Estimator组成。
  - ML框架下所有的数据源都基于DataFrame，所有模型都基于Spark的数据类型表示，ML的API操作也从RDD向DataFrame全面转变。

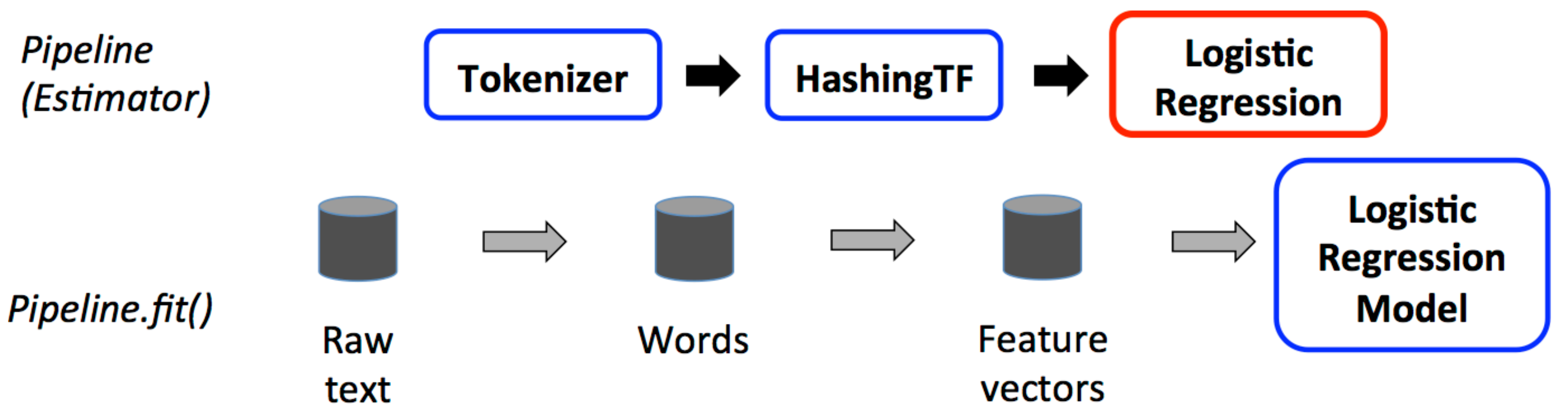
ML主要概念

- DataFrame：将Spark SQL的DataFrame作为一个ML数据集使用，支持多种数据类型。一个DataFrame可以有不同的列存储文本、特征向量、真实标签和预测。
- Transformer：实现一个DataFrame转换成另一个DataFrame的算法。实现transform()方法。
- Estimator：适配一个DataFrame，产生另一个Transformer的算法。实现fit()方法。
- Pipeline：指定连接多个Transformers和Estimators的ML工作流。
- Parameter：全部的Transformers和Estimators共享一个指定Parameter的通用API。

Pipeline

- 机器学习的流水线通常指运行一系列算法的过程，并从数据中学习。例如，一个简单的文本文档处理工作流程可能包括以下几个阶段：
  - 将每个文档的文本切分成单词；
  - 将每个文档单词转换成一个数值特征向量；
  - 使用特征向量和标签，学习一个预测模型。
- Spark ML代表一个作为流水线的工作流，由一系列流水线阶段组成，并以一个特定的顺序运行。
- 一个流水线被指定为一系列由Transformer或Estimator组成的阶段（Stage）。这些阶段按照顺序运行，输入的DataFrame在运行的每个阶段进行转换。

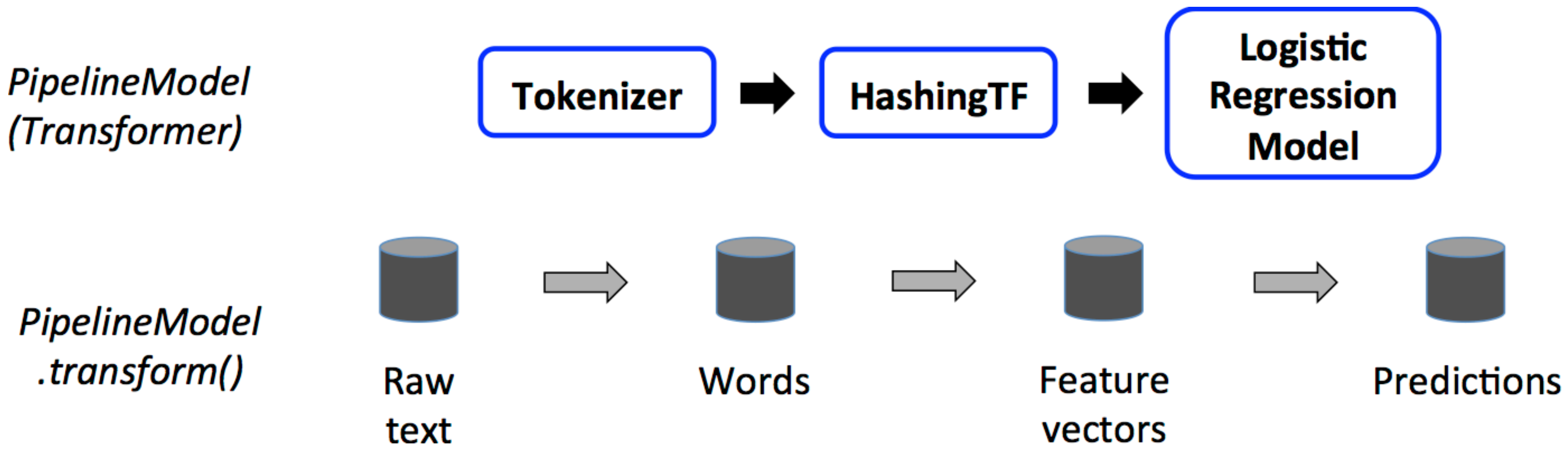
Pipeline



Training Time的Pipeline

流水线是一个Estimator，因此，在一个流水线的fit()方法运行之后，生成一个PipelineModel，该模型是一个Transformer。

Pipeline



Test Time的Pipeline

Pipeline和PipelineModel在实际运行Pipeline之前，使用DataFrame模式（schema）进行类型检查，该模式描述DataFrame中列的数据类型。

再看K-Means


```
import org.apache.spark.ml.clustering.Kmeans
val dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")
// Trains a k-means model.
val kmeans = new KMeans().setK(2).setSeed(1L)
val model = kmeans.fit(dataset)
// Make predictions
val predictions = model.transform(dataset)
// Evaluate clustering by computing Silhouette score
val evaluator = new ClusteringEvaluator()
val silhouette = evaluator.evaluate(predictions)
// Shows the result.
println("Cluster Centers: ")
model.clusterCenters.foreach(println)
```

再看鸢尾花


```
val df: DataFrame = sparkSession.read.format("csv").option("inferSchema",
"true").option("header","true").option("sep",",").load(path)
//特征工程
//将4个特征整合为一个特征向量
val assembler: VectorAssembler = new VectorAssembler().setInputCols(Array
("sepal_length","sepal_width","petal_length","petal_width")).setOutputCol("features")
val assmblerDf: DataFrame = assembler.transform(df)
//将类别型class转变为数值型
val stringIndex: StringIndexer = new StringIndexer().setInputCol("class").
setOutputCol("label")
val stingIndexModel: StringIndexerModel = stringIndex.fit(assmblerDf)
val indexDf: DataFrame = stingIndexModel.transform(assmblerDf)
//将数据切分成两部分，分别为训练数据集和测试数据集
val Array(trainData,testData): Array[Dataset[Row]] = indexDf.randomSplit
(Array(0.8,0.2))
```

再看鸢尾花

// 准备计算，设置特征列和标签列

```
val classifier: DecisionTreeClassifier = new DecisionTreeClassifier().setFeaturesCol
("features").setMaxBins(16).setImpurity("gini").setSeed(10)
```

```
val dtcModel: DecisionTreeClassificationModel = classifier.fit(trainData)
```

// 完成建模分析

```
val trainPre: DataFrame = dtcModel.transform(trainData)
```

// 预测分析

```
val testPre: DataFrame = dtcModel.transform(testData)
```

// 评估

```
val acc: Double = new MulticlassClassificationEvaluator().setMetricName
("accuracy").evaluate(testPre)
```

一般步骤

- Spark MLlib：
  - 加载数据
  - 把数据转换成所需的格式
  - 设置算法参数
  - 调用算法模型训练
  - 预测
  - 模型评估
- Spark ML：
  - 把整个机器学习过程抽象成Pipeline
  - 通过Transformer和Estimator构成的多个Stage完成Pipeline过程。

预测回头客

- 1. 导入需要的包
- 2. 读取训练数据
- 3. 构建模型
- 4. 评估模型

摘要

- Spark MLlib
- GraphX

GraphX

GraphX is Apache Spark's API for graphs and graph-parallel computation.

Flexibility

Seamlessly work with both graphs and collections.

GraphX unifies ETL, exploratory analysis, and iterative graph computation within a single system. You can view the same data as both graphs and collections, transform and join graphs with RDDs efficiently, and write custom iterative graph algorithms using the Pregel API.

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Using GraphX in Scala

Algorithms

Choose from a growing library of graph algorithms.

In addition to a highly flexible API, GraphX comes with a variety of graph algorithms, many of which were contributed by our users.

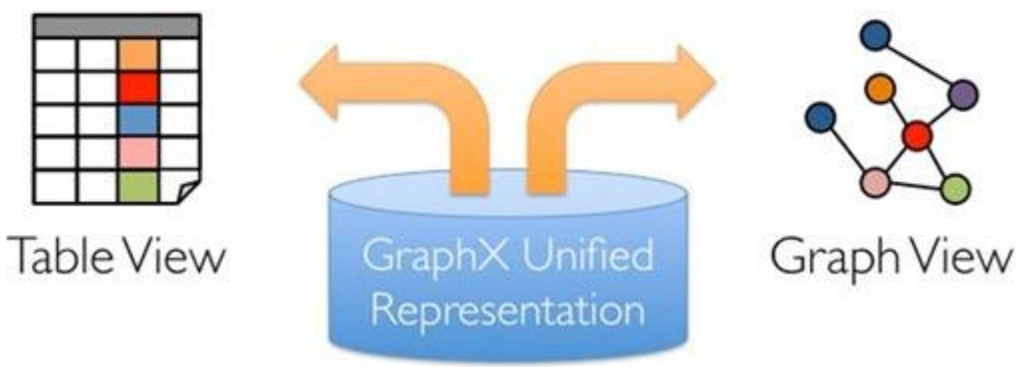
- PageRank
- Connected components
- Label propagation
- SVD++
- Strongly connected components
- Triangle count

GraphX

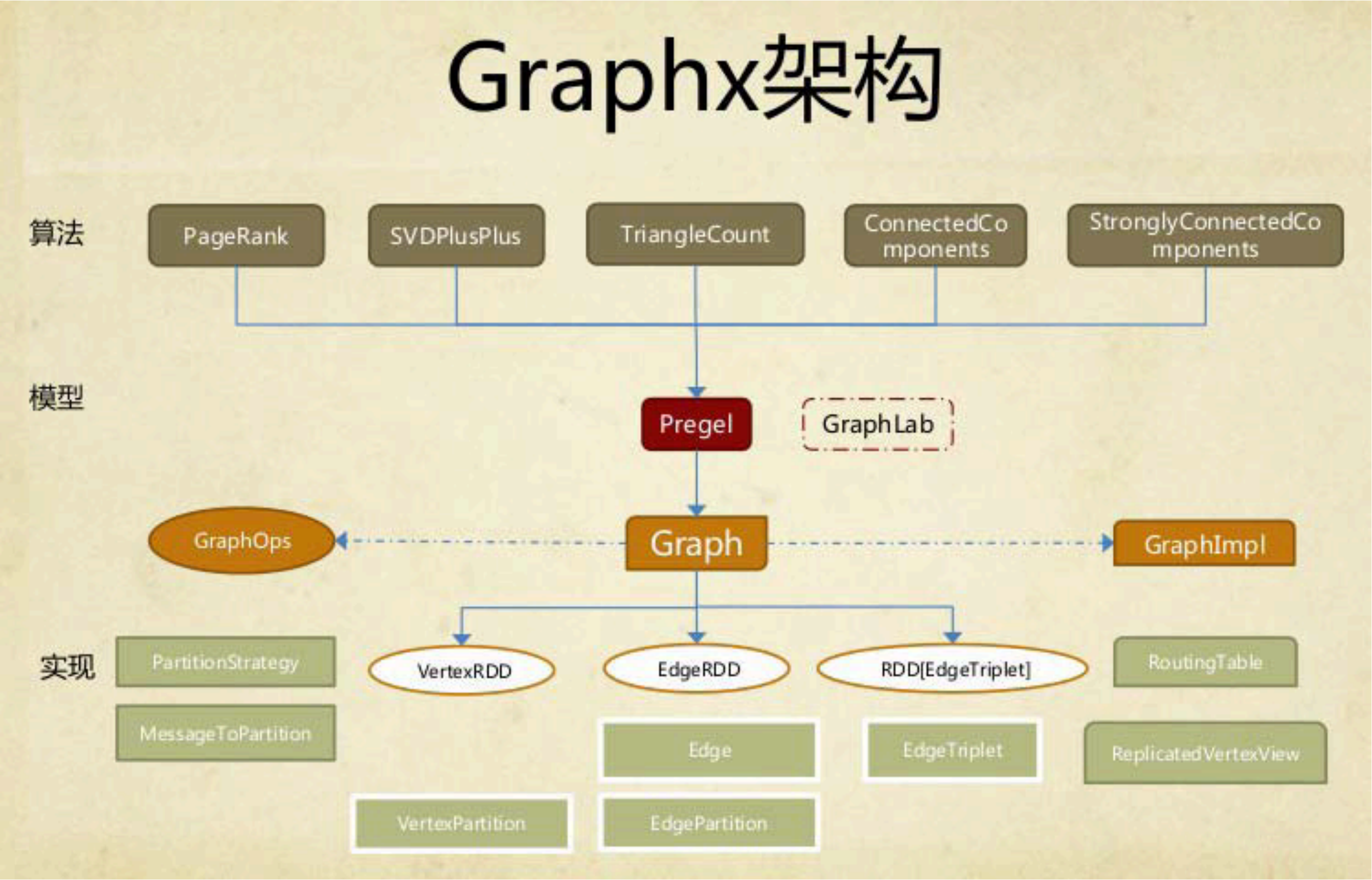
- GraphX是Spark中用于图和图并行计算的组件。
- GraphX通过扩展Spark RDD引入一个新的图抽象，一个将有效信息放在顶点和边的有向多重图。
- GraphX公开了一系列基本运算，以及一个优化后的Pregel API的变形。包括越来越多的图形计算和builder构造器，以简化图形分析任务。
- 在Spark之上提供了一站式解决方案，可以方便且高效地完成图计算的一整套流水作业。

GraphX核心抽象

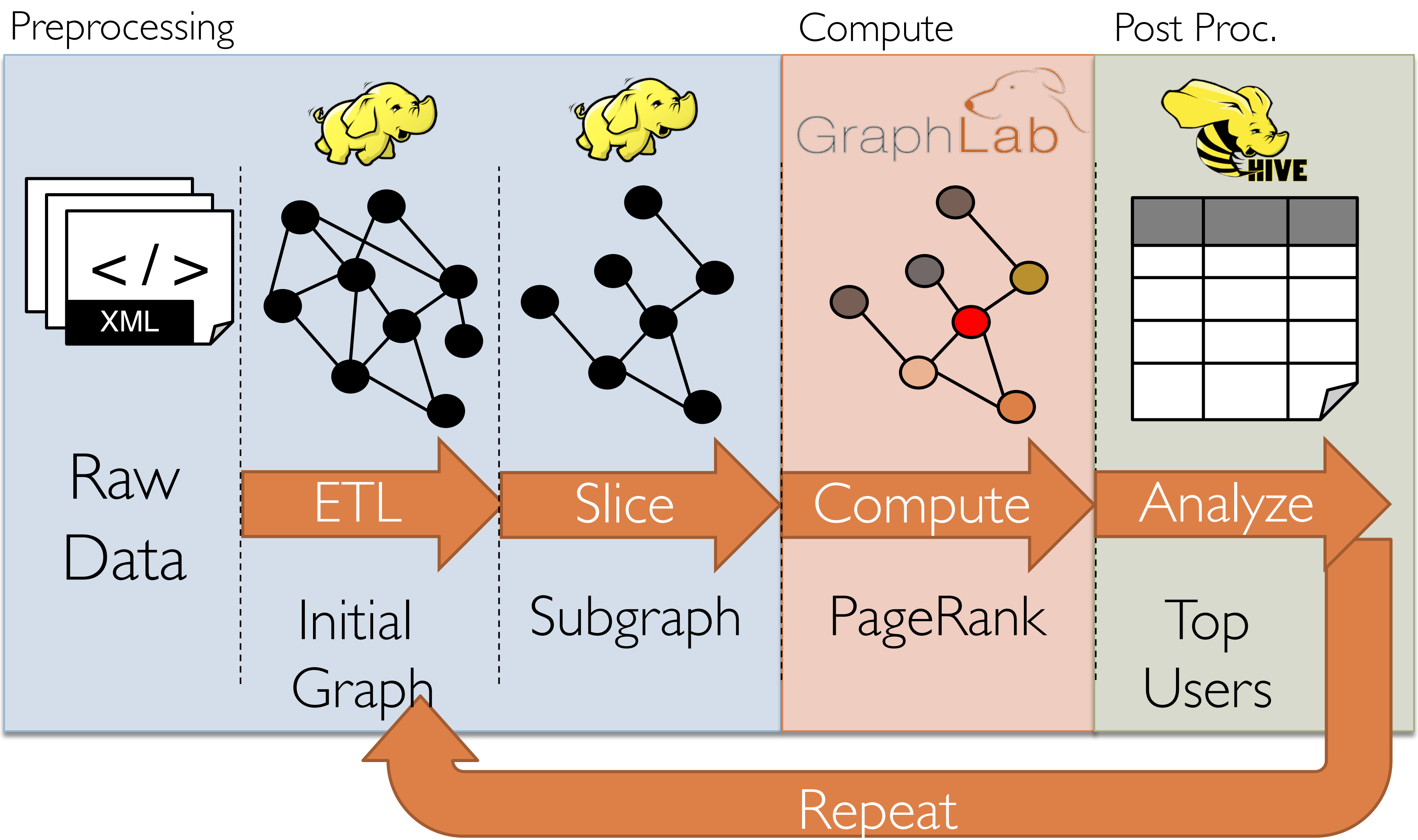
- 弹性分布式属性图（Resilient Distributed Property Graph），一种点和边都带属性的有向多重图。它扩展了Spark RDD的抽象，有Table和Graph两种视图，而只需要一份物理存储。两种视图都有自己独有的操作符，从而获得了灵活操作和执行效率。



GraphX框架


Graphx架构
算法
PageRank
SVDPlusPlus
TriangleCount
ConnectedComponents
StronglyConnectedComponents
模型
Pregel
GraphLab
GraphOps
Graph
GraphImpl
实现
PartitionStrategy
MessageToPartition
VertexRDD
EdgeRDD
RDD[EdgeTriplet]
RoutingTable
Edge
EdgeTriplet
ReplicatedVertexView
VertexPartition
EdgePartition

图处理流水线

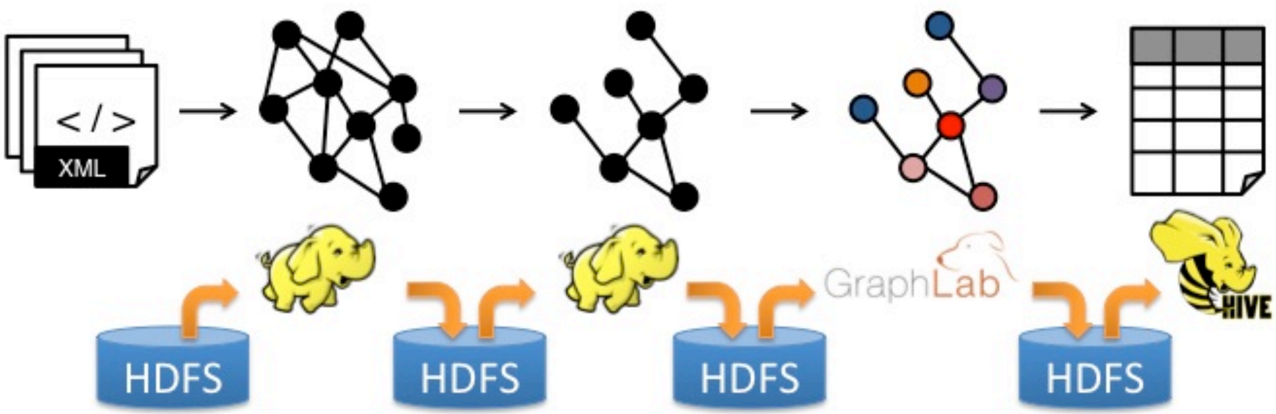


Preprocessing
Compute
Post Proc.
< / >
XML
GraphLab
HIVE
Raw Data
ETL
Slice
Compute
Analyze
Initial Graph
Subgraph
PageRank
Top Users
Repeat

图处理流水线

Inefficient

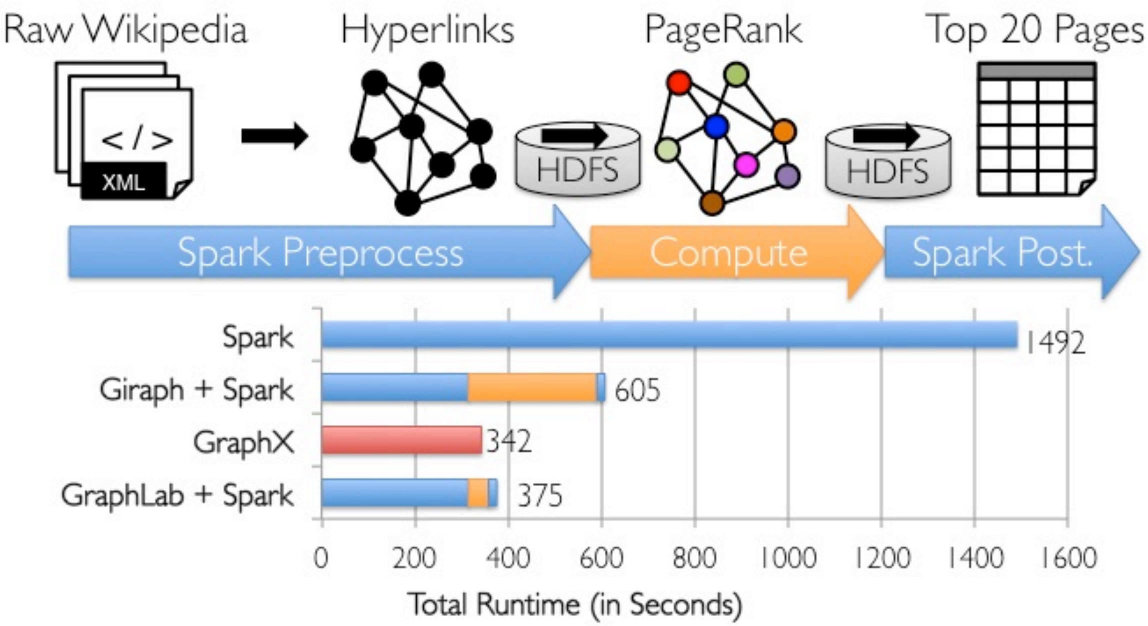
Extensive data movement and duplication across the network and file system



Limited reuse internal data-structures across stages

图处理流水线

A Small Pipeline in GraphX



Timed end-to-end GraphX is *faster* than GraphLab

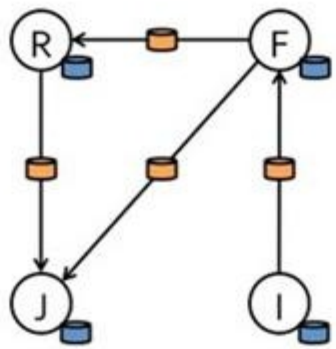
两种视图

- 对Graph视图的所有操作，最终都会转换成其关联的Table视图的RDD操作来完成。这样对一个图的计算，最终在逻辑上，等价于一系列RDD的转换过程。因此，Graph最终具备了RDD的3个关键特性：Immutable、Distributed和Fault-Tolerant，其中最关键的是Immutable（不变性）。逻辑上，所有图的转换和操作都产生了一个新图；物理上，GraphX会有一定程度的不变顶点和边的复用优化，对用户透明。

- 两种视图底层共用的物理数据，由RDD[VertexPartition]和RDD[EdgePartition]这两个RDD组成。点和边实际都不是以表Collection[tuple]的形式存储的，而是由VertexPartition/EdgePartition在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在RDD转换过程中是共用的，降低了计算和存储开销。

两种视图

Property Graph



Vertex Property Table

Id	Property (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Edge Property Table

SrcId	DstId	Property (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

- Table视图将图看成Vertex Property Table和Edge Property Table等的组合，这些Table继承了Spark RDD的API(filter, map等)。
- Graph视图上包括reverse/subgraph/mapV(E)/joinV(E)/mrTriplets等操作。

GraphX编程

- 属性图是一个用户定义顶点和边的有向多重图。
- 有向多重图是一个有向图，它可能有多个平行边共享相同的源顶点和目标顶点。
- 多重图支持并行边的能力简化了有多重关系的建模场景。每个顶点是由具有64位长度的唯一标识符（VertexID）作为主键。GraphX没有对顶点添加任何顺序的约束。同样，每条边具有相应的源顶点和目标顶点的标识符。
- 属性表的参数由顶点（VD）和边（ED）的类型决定。

GraphX编程


```scala
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

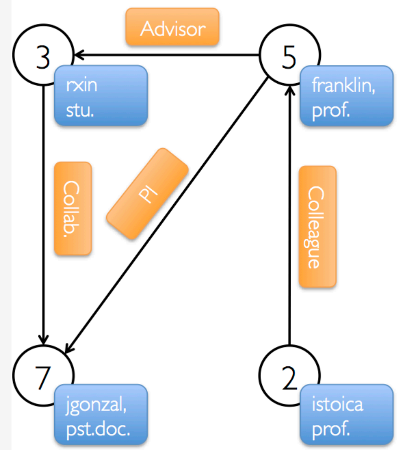
```scala
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

GraphX的图操作

- 构造图
  - 通过Graph Object构造
  - 通过Graph Builder构造

```scala
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

GraphX的图操作

- GraphLoader.edgeListFile提供了一种从磁盘上边的列表载入图的方式。

```scala
object GraphLoader {
  def edgeListFile(
      sc: SparkContext,
      path: String,
      canonicalOrientation: Boolean = false,
      minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}
```

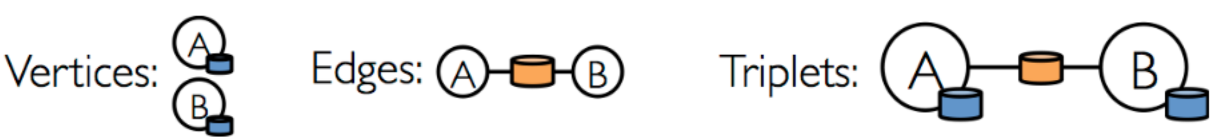
```scala
import org.apache.spark.graphx.GraphLoader

// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
```

GraphX的图操作


```
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

```
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```



GraphX的图操作

- 属性操作
- 转换操作
- 结构操作
- 关联操作
- 聚合操作
- 缓存操作

Table Operators

- Table (RDD) operators are inherited from Spark:

`map`	`reduce`	`sample`
`filter`	`count`	`take`
`groupBy`	`fold`	`first`
`sort`	`reduceByKey`	`partitionBy`
`union`	`groupByKey`	`mapWith`
`join`	`cogroup`	`pipe`
`leftOuterJoin`	`cross`	`save`
`rightOuterJoin`	`zip`	`...`

Graph Operators


```scala
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // Information about the Graph ===================================================================
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections =============================================================
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs ==================================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // Change the partitioning heuristic  ============================================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
  // Transform vertex and edge attributes ==========================================================
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
    : Graph[VD, ED2]
```

Graph Operators

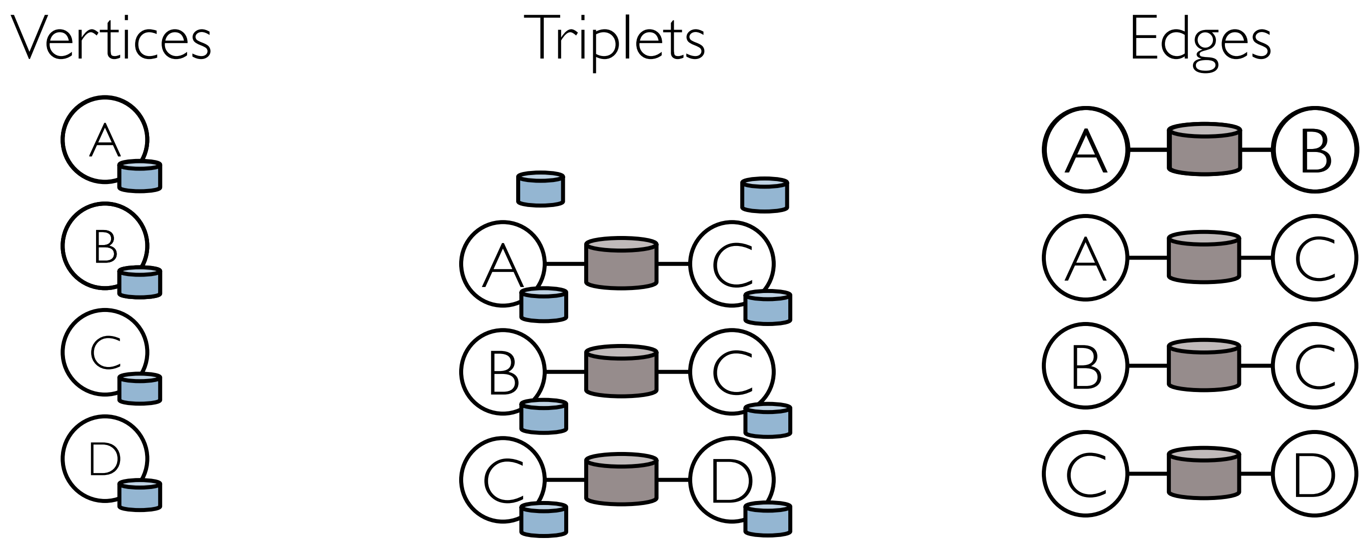

```scala
// Modify the graph structure ====================================================================
def reverse: Graph[VD, ED]
def subgraph(
    epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
    vpred: (VertexId, VD) => Boolean = ((v, d) => true))
  : Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
// Join RDDs with the graph ======================================================================
def joinVertices[U](table: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, U) => VD): Graph[VD, ED]
def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])
    (mapFunc: (VertexId, VD, Option[U]) => VD2)
  : Graph[VD2, ED]
// Aggregate information about adjacent triplets =================================================
def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId, VD)]]
def aggregateMessages[Msg: ClassTag](
    sendMsg: EdgeContext[VD, ED, Msg] => Unit,
    mergeMsg: (Msg, Msg) => Msg,
    tripletFields: TripletFields = TripletFields.All)
  : VertexRDD[A]
```

Graph Operators


```scala
  // Iterative graph-parallel computation ==========================================================
  def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
      vprog: (VertexId, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId,A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED]
  // Basic graph algorithms ========================================================================
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexId, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]
}
```

Triplets Join Vertices and Edges

- The ***triplets*** operator joins vertices and edges:




```scala
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

常用图算法

- PageRank算法
- 三角形计数算法
- 连接分量算法

PageRank算法

- GraphX自带PageRank的静态和动态实现，放在PageRank对象中。静态的PageRank运行固定数量的迭代，而动态的PageRank运行直到排名收敛。

```
import org.apache.spark.graphx.GraphLoader

// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

三角形计数算法

- 计算通过各顶点的三角形数目，从而提供集群的度。TriangleCount要求边的指向（srcId<dstId），并使用Graph.partitionBy分割图形。

```
import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

连接分量算法

- 连接分量算法标出了图中编号最低的顶点所连接的子集。

```
import org.apache.spark.graphx.GraphLoader

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

应用场景

- 图谱体检平台
- 多图合并工具
- 能量传播模型
- ……