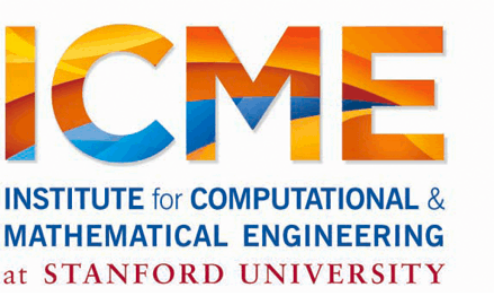
Advanced Data Science on Spark

Reza Zadeh



@Reza_Zadeh | http://reza-zadeh.com

Data Science Problem

Data growing faster than processing speeds

Only solution is to parallelize on large clusters

» Wide use in both enterprises and web industry

How do we program these things?

Use a Cluster

Convex Optimization

Matrix Factorization

Machine Learning

Numerical Linear Algebra

Large Graph analysis

Streaming and online algorithms

Following lectures on http://stanford.edu/~rezab/dao

Slides at http://stanford.edu/~rezab/slides/sparksummit2015

Outline

Data Flow Engines and Spark

The Three Dimensions of Machine Learning

Built-in Libraries

MLlib + {Streaming, GraphX, SQL}

Future of MLlib

Traditional Network Programming

Message-passing between nodes (e.g. MPI)

Very difficult to do at scale:

- How to split problem across nodes?
  - Must consider network & data locality
- How to deal with failures? (inevitable at scale)
- Even worse: stragglers (node not failed, but slow)
- Ethernet networking not fast
- Have to write programs for each machine

Rarely used in commodity datacenters

Disk vs Memory

L1 cache reference:	0.5 ns
L2 cache reference:	7 ns
Mutex lock/unlock:	100 ns
Main memory reference:	100 ns
Disk seek:	10,000,000 ns

Network vs Local

Send 2K bytes over 1 Gbps network:	20,000 ns
Read 1 MB sequentially from memory:	250,000 ns
Round trip within same datacenter:	500,000 ns
Read 1 MB sequentially from network:	10,000,000 ns
Read 1 MB sequentially from disk:	30,000,000 ns
Send packet CA->Netherlands->CA:	150,000,000 ns

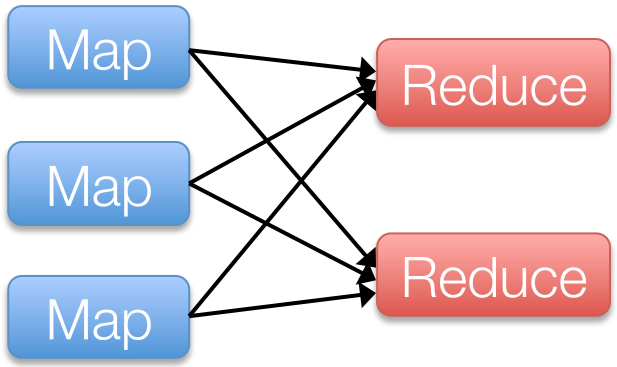
Data Flow Models

Restrict the programming interface so that the system can do more automatically

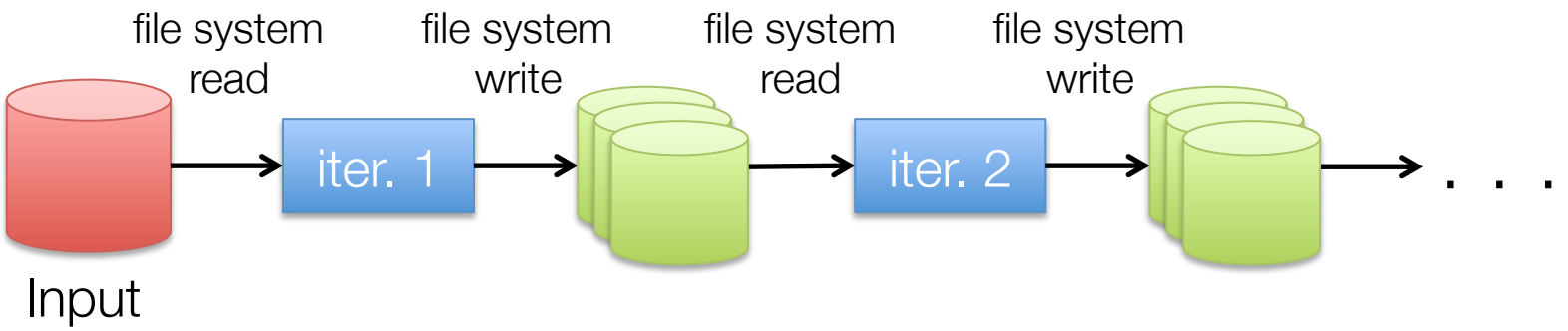
Express jobs as graphs of high-level operators

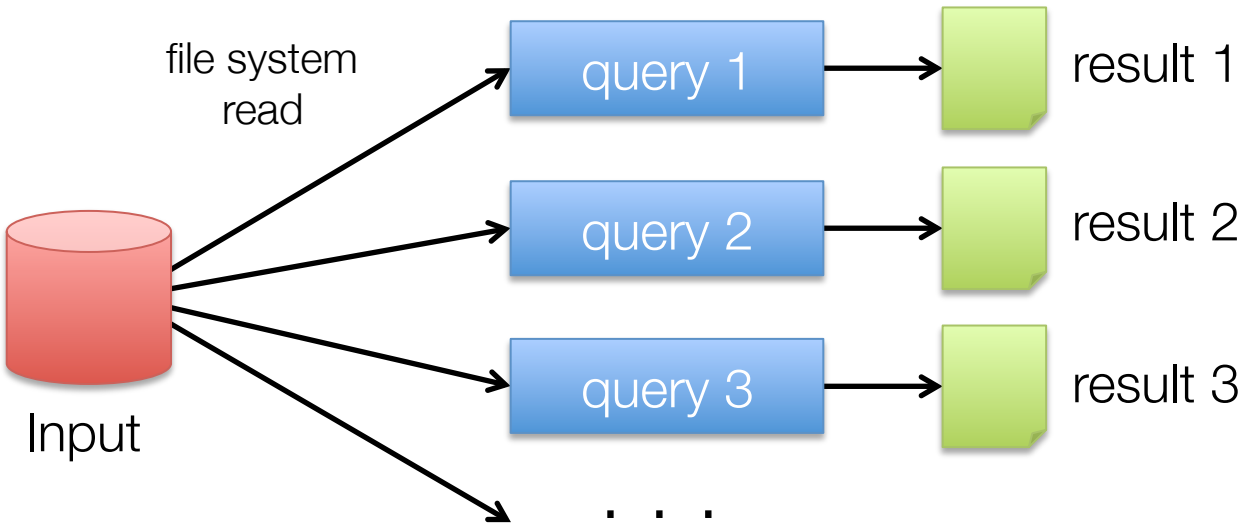
- » System picks how to split each operator into tasks and where to run each task
- » Run parts twice fault recovery

Biggest example: MapReduce



Example: Iterative Apps





Commonly spend 90% of time doing I/O

MapReduce evolved

MapReduce is great at one-pass computation, but inefficient for *multi-pass* algorithms

No efficient primitives for data sharing

- » State between steps goes to distributed file system
- » Slow due to replication & disk storage

Verdict

MapReduce algorithms research doesn't go to waste, it just gets sped up and easier to use

Still useful to study as an algorithmic framework, silly to use directly

Spark Computing Engine

Extends a programming language with a distributed collection data-structure

- » “Resilient distributed datasets” (RDD)

Open source at Apache

- » Most active community in big data, with 50+ companies contributing

Clean APIs in Java, Scala, Python

Community: SparkR, being released in 1.4!

Key Idea

Resilient Distributed Datasets (RDDs)

- » Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
- » Built via parallel transformations (map, filter, …)
- » The world only lets you make make RDDs such that they can be:

Automatically rebuilt on failure

Resilient Distributed Datasets (RDDs)

Main idea: Resilient Distributed Datasets

» Immutable collections of objects, spread across cluster

» Statically typed: RDD[T] has objects of type T

```
val sc = new SparkContext()
val lines = sc.textFile("log.txt")   // RDD[String]

// Transform using standard collection operations
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split('\t')(2))

messages.saveAsTextFile("errors.txt")
```

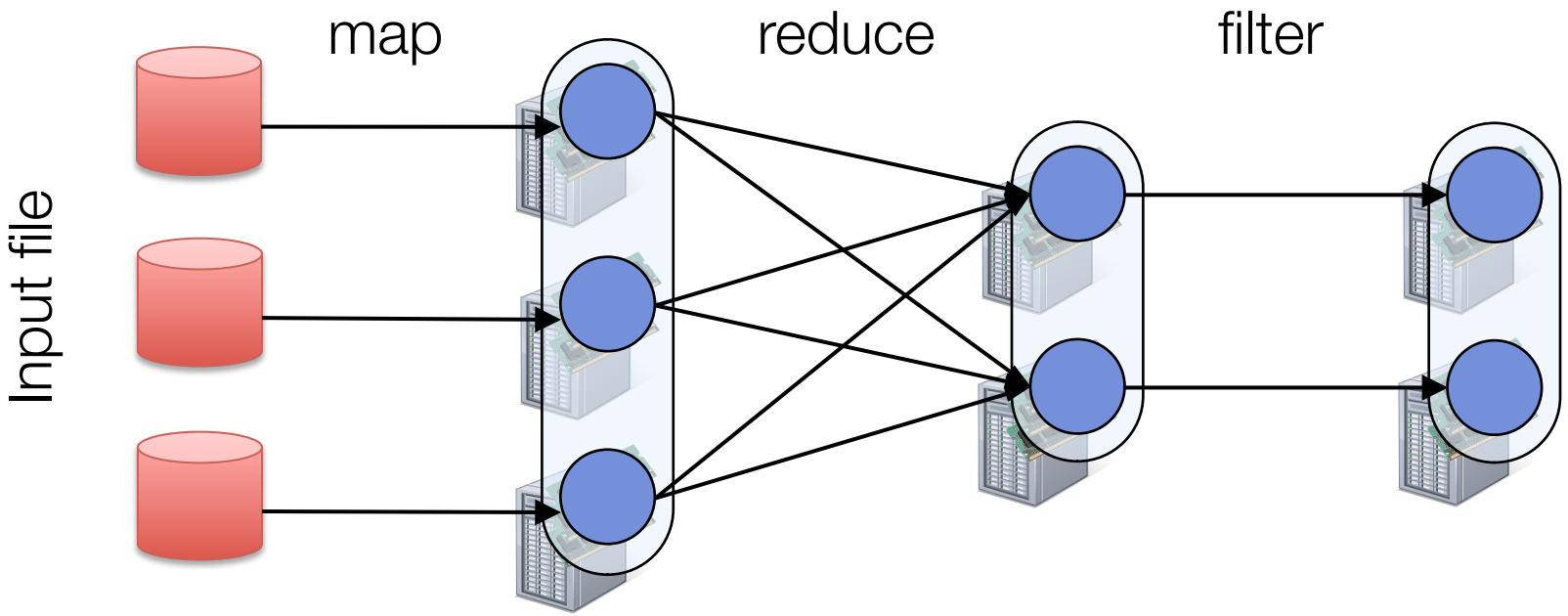
→ lazily evaluated

→ kicks off a computation

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

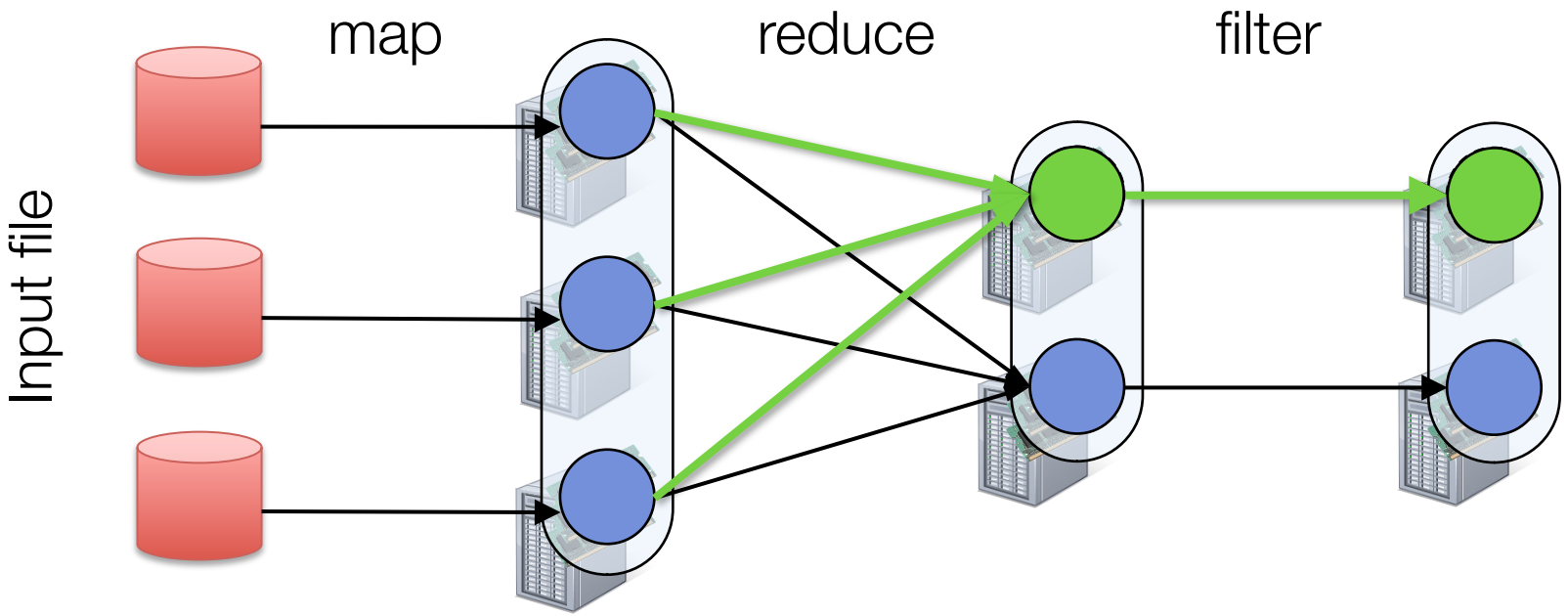
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Fault Tolerance

RDDs track *lineage* info to rebuild lost data

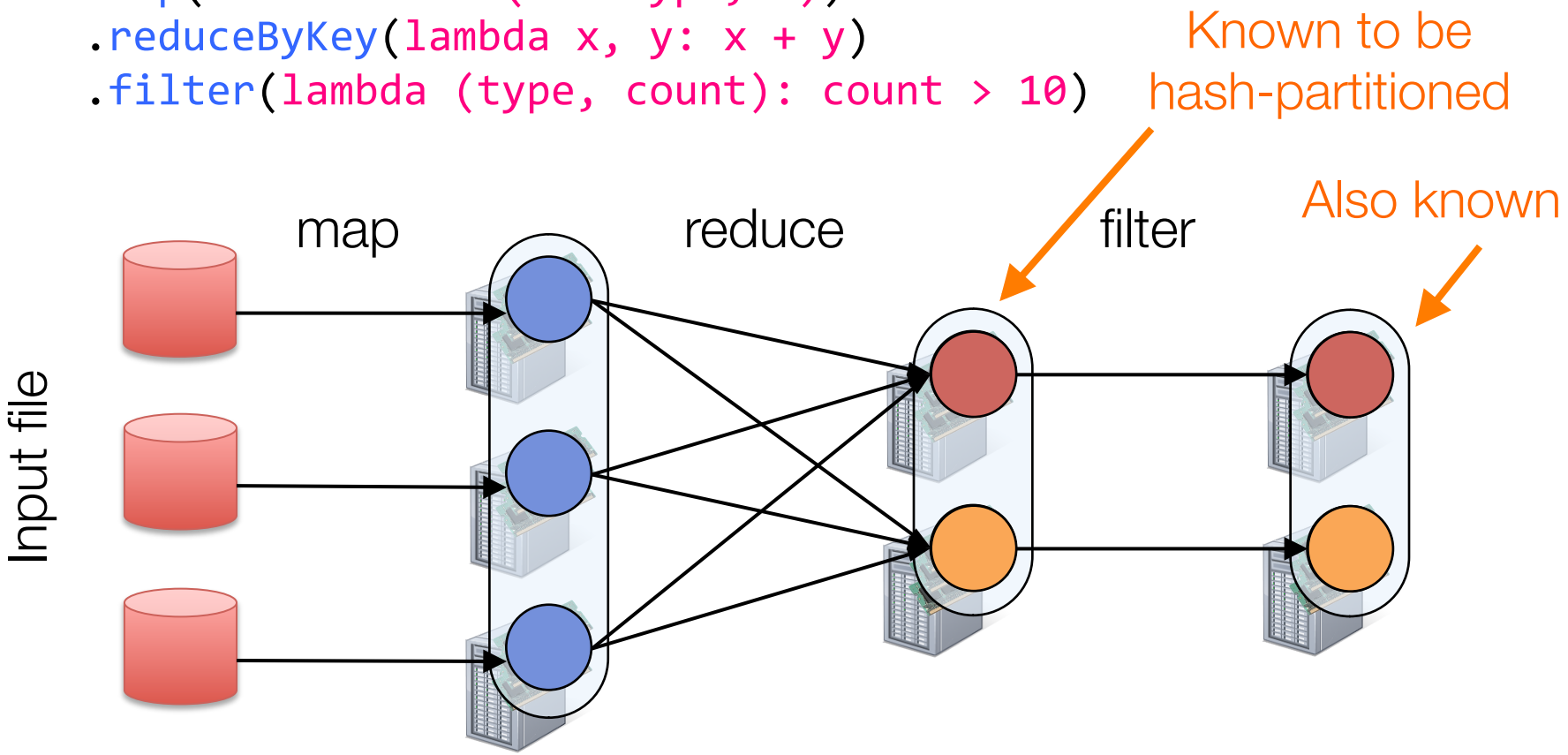
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Partitioning

RDDs know their partitioning functions

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



MLlib: Available algorithms

classification: logistic regression, linear SVM, naïve Bayes, least squares, classification tree

regression: generalized linear models (GLMs), regression tree

collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

clustering: k-means||

decomposition: SVD, PCA

optimization: stochastic gradient descent, L-BFGS

The Three Dimensions

ML Objectives

Almost all machine learning objectives are optimized using this update

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

w is a vector of dimension d
we're trying to find the best w via optimization

Scaling

1) Data size

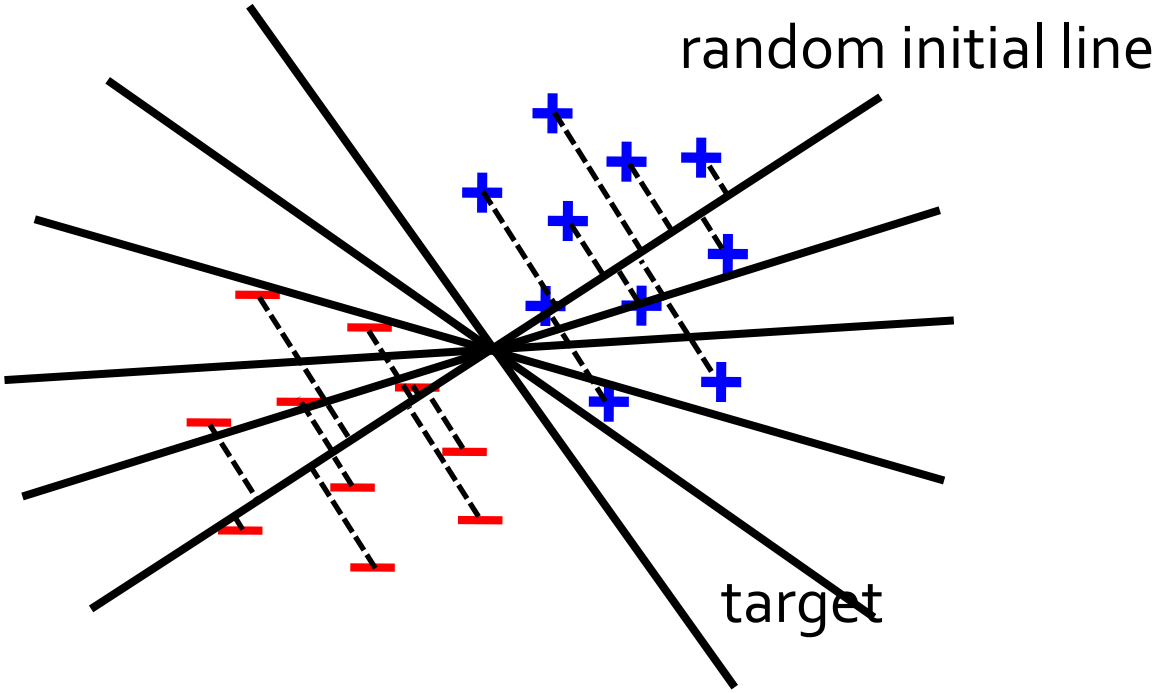
$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

2) Number of models

3) Model size

Logistic Regression

Goal: find best line separating two sets of points



Data Scaling

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

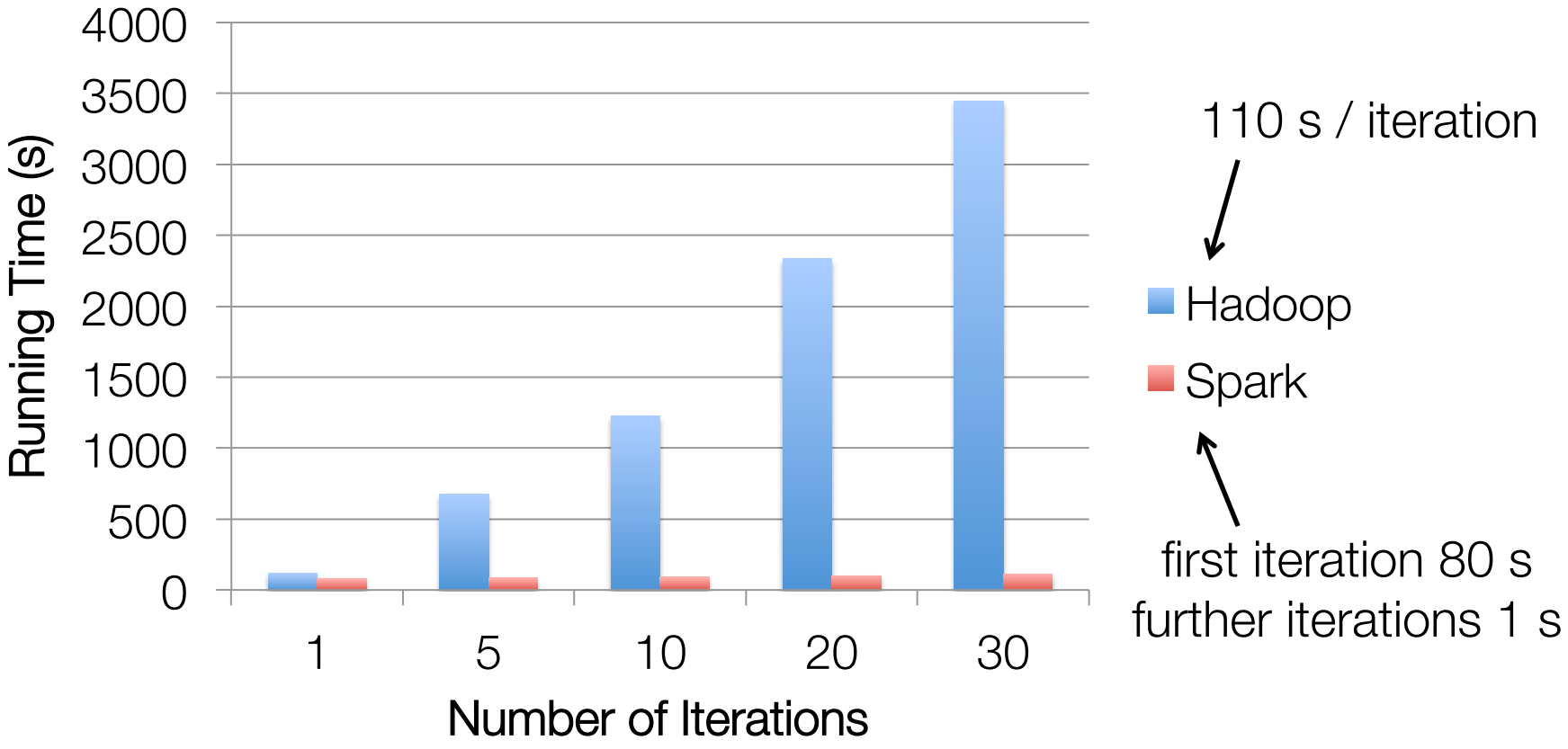
```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= alpha * gradient
}
```

Separable Updates

Can be generalized for

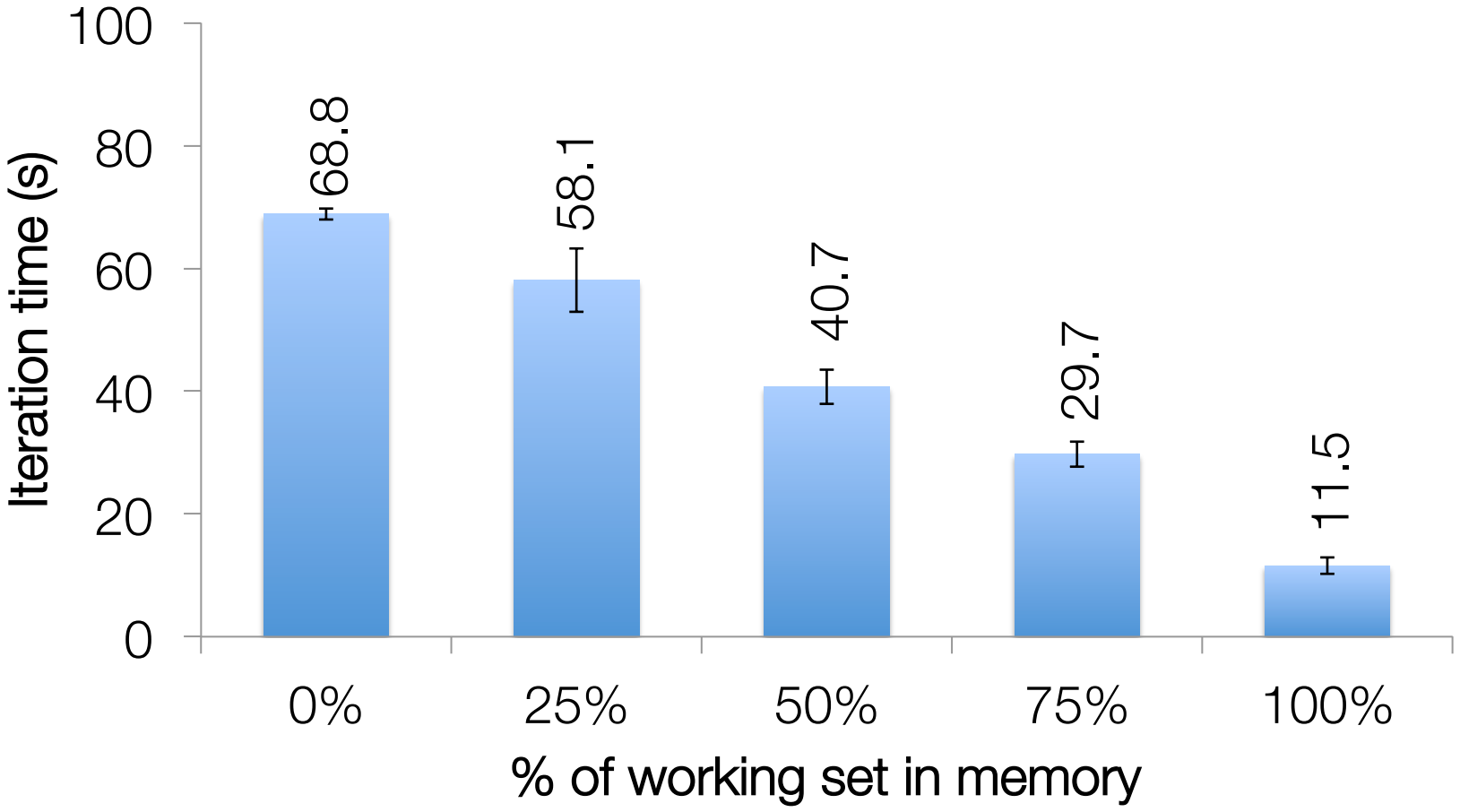
- Unconstrained optimization
- Smooth or non-smooth
- LBFGS, Conjugate Gradient, Accelerated Gradient methods, …

Logistic Regression Results



100 GB of data on 50 m1.xlarge EC2 machines

Behavior with Less RAM



Lots of little models

Is embarrassingly parallel

Most of the work should be handled by data flow paradigm

ML pipelines does this

Hyper-parameter Tuning

```
// Build a parameter grid.
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 20, 40))
  .addGrid(lr.regParam, Array(0.01, 0.1, 1.0))
  .build()

// Set up cross-validation.
val cv = new CrossValidator()
  .setNumFolds(3)
  .setEstimator(pipeline)
  .setEstimatorParamMaps(paramGrid)
  .setEvaluator(new BinaryClassificationEvaluator)

// Fit a model with cross-validation.
val cvModel = cv.fit(trainingDataset)
```

Model Scaling

Linear models only need to compute the dot product of each example with model

Use a BlockMatrix to store data, use joins to compute dot products

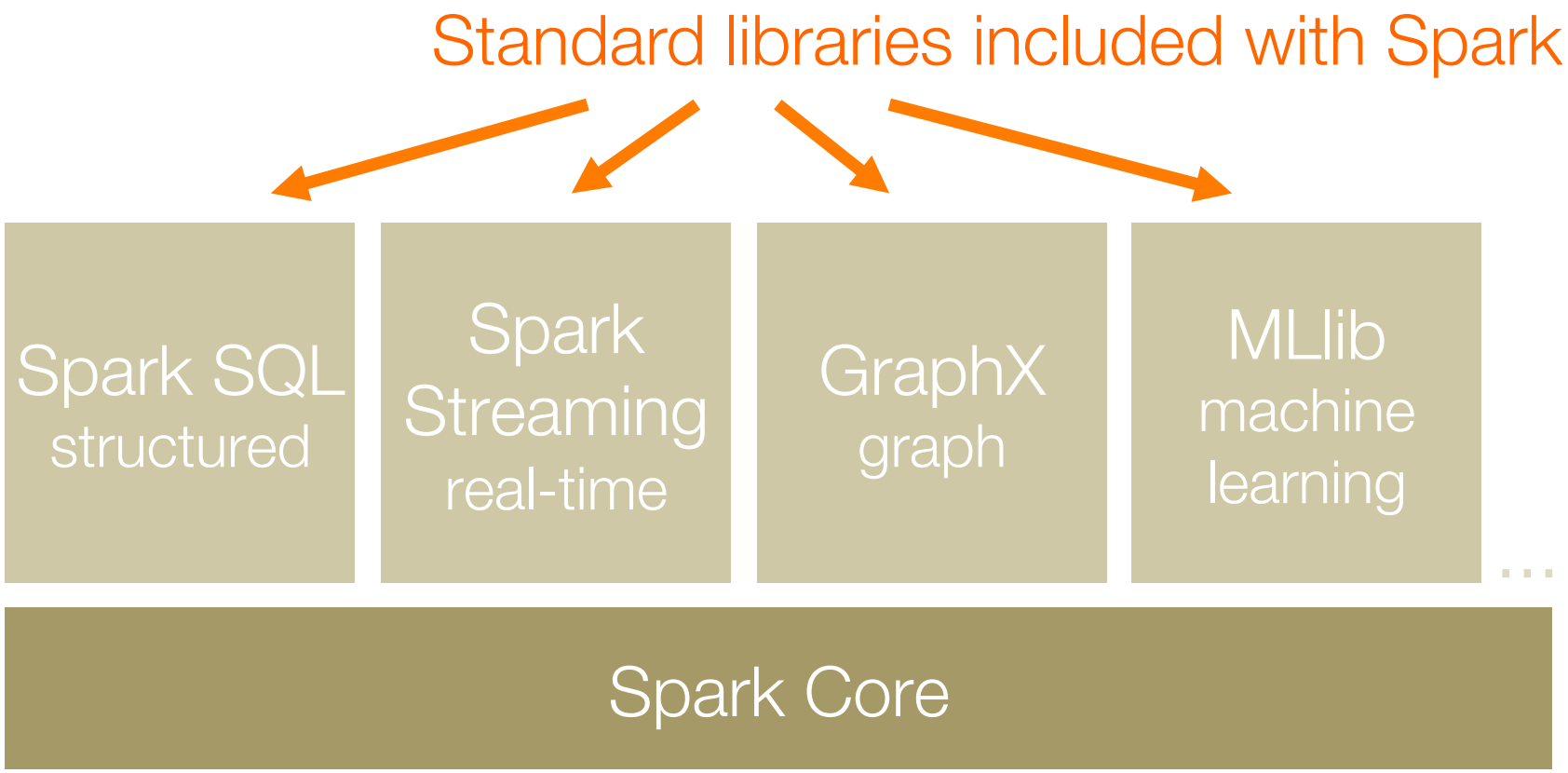
Coming in 1.5

Model Scaling

Data joined with model (weight):

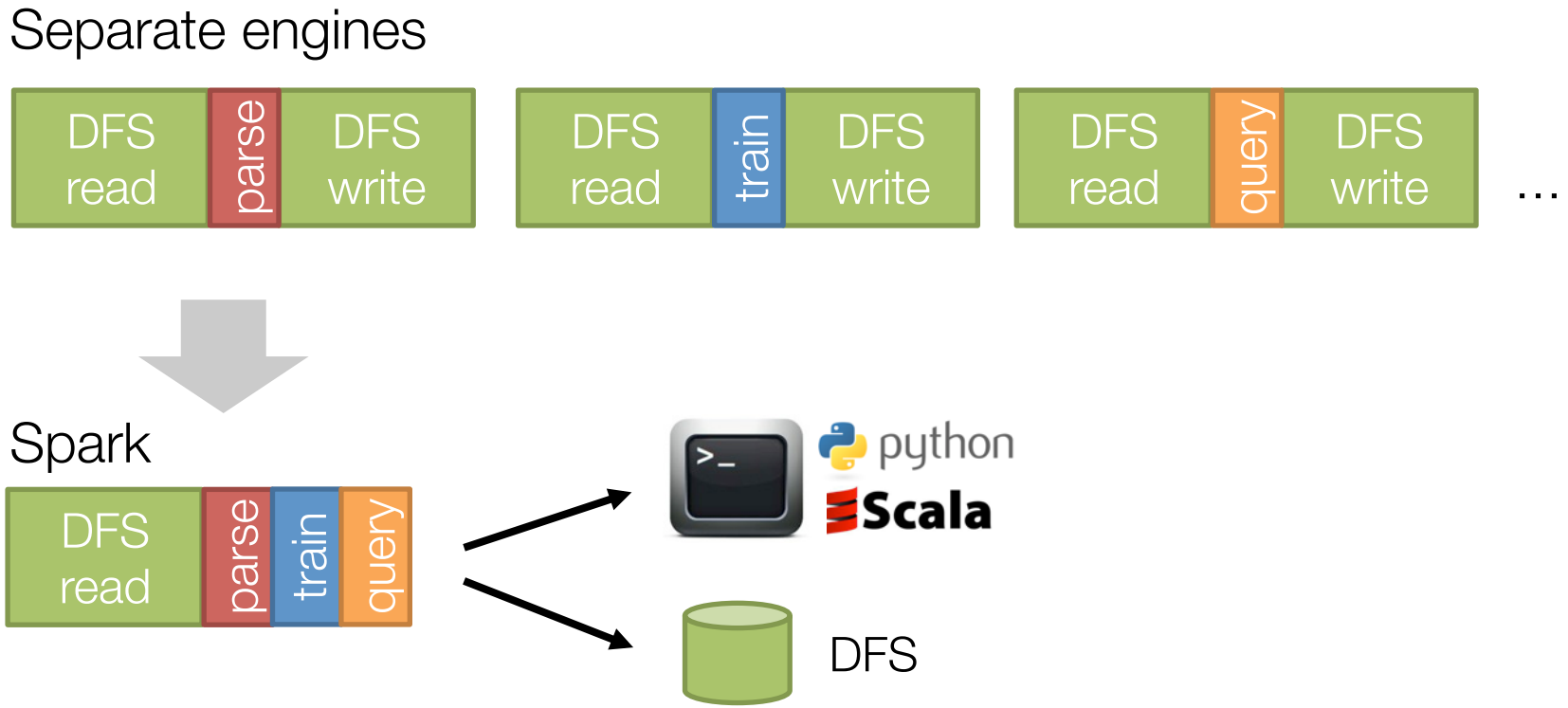
Built-in libraries

A General Platform



Benefit for Users

Same engine performs data extraction, model training and interactive queries



Machine Learning Library (MLlib)

```
points = context.sql(“select latitude, longitude from tweets”)
model = KMeans.train(points, 10)
```

70+ contributors
in past year

MLlib algorithms

classification: logistic regression, linear SVM, naïve Bayes, classification tree

regression: generalized linear models (GLMs), regression tree

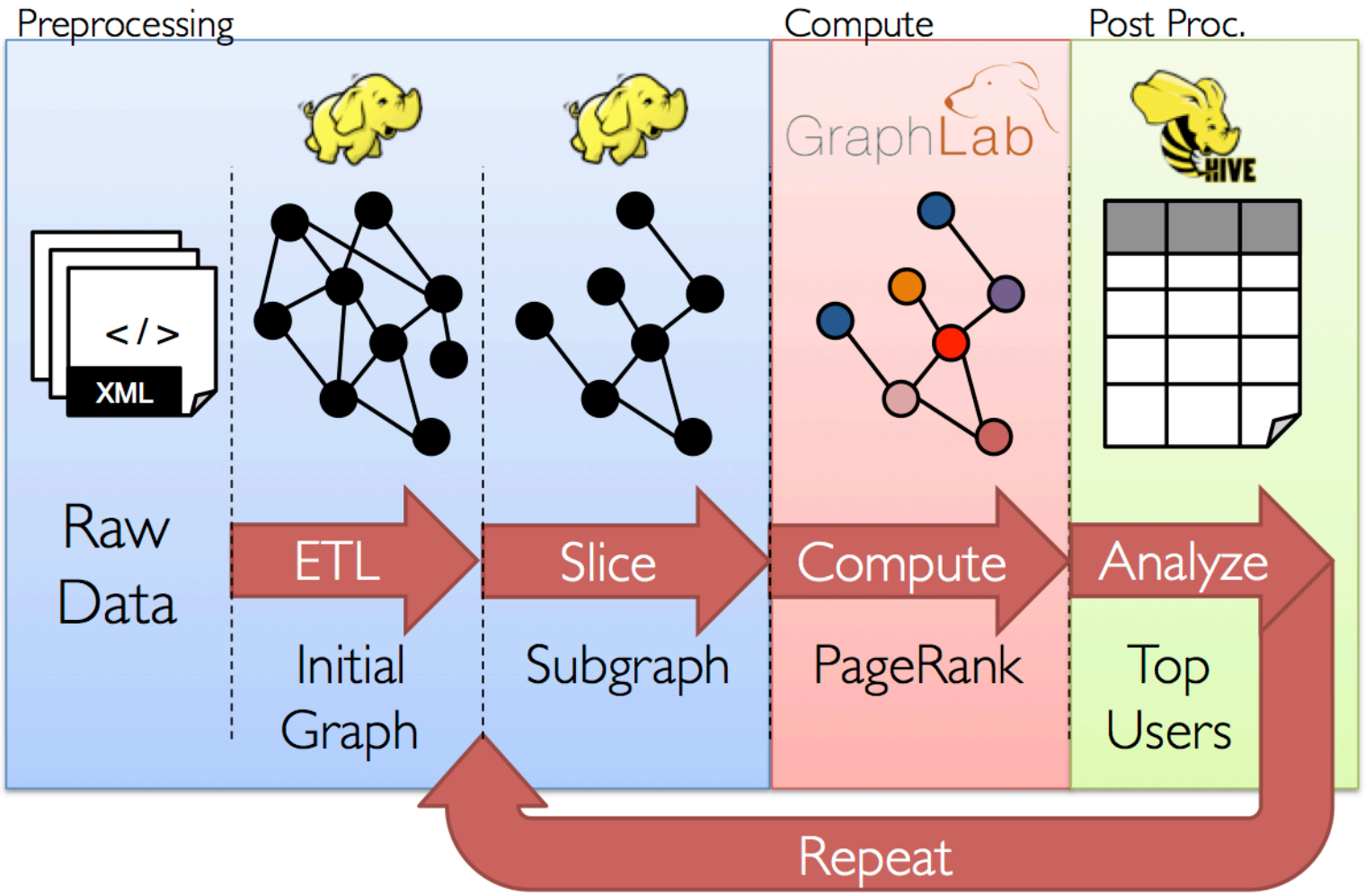
collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

clustering: k-means||

decomposition: SVD, PCA

optimization: stochastic gradient descent, L-BFGS

GraphX


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

GraphX

General graph processing library

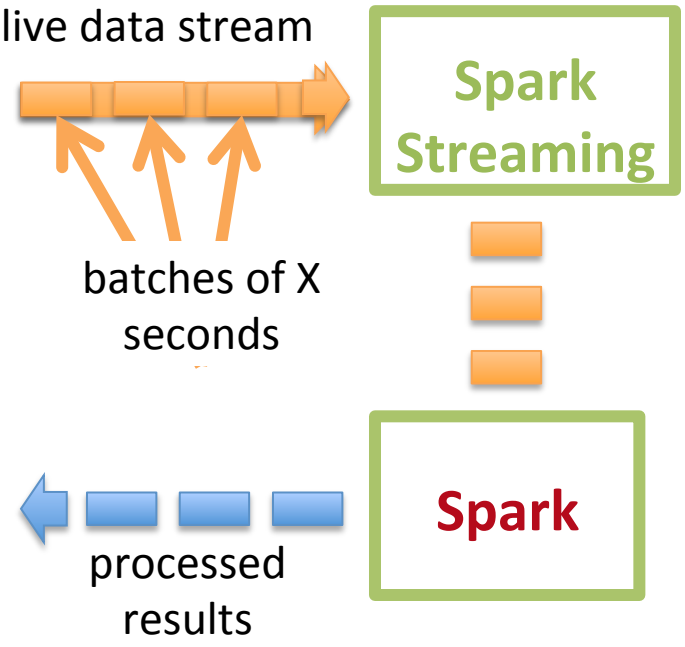
Build graph using RDDs of nodes and edges

Run standard algorithms such as PageRank

Spark Streaming

Run a streaming computation as a **series of very small, deterministic batch jobs**

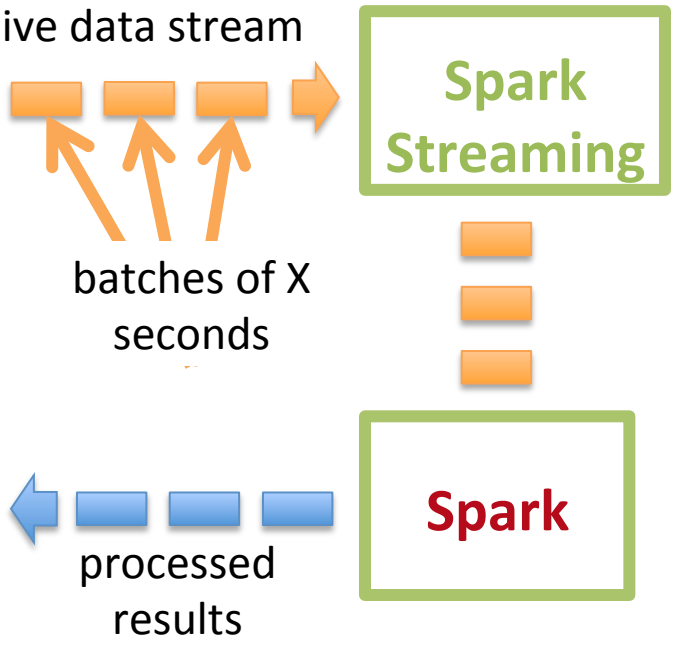
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Spark Streaming

Run a streaming computation as a **series of very small, deterministic batch jobs**

- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



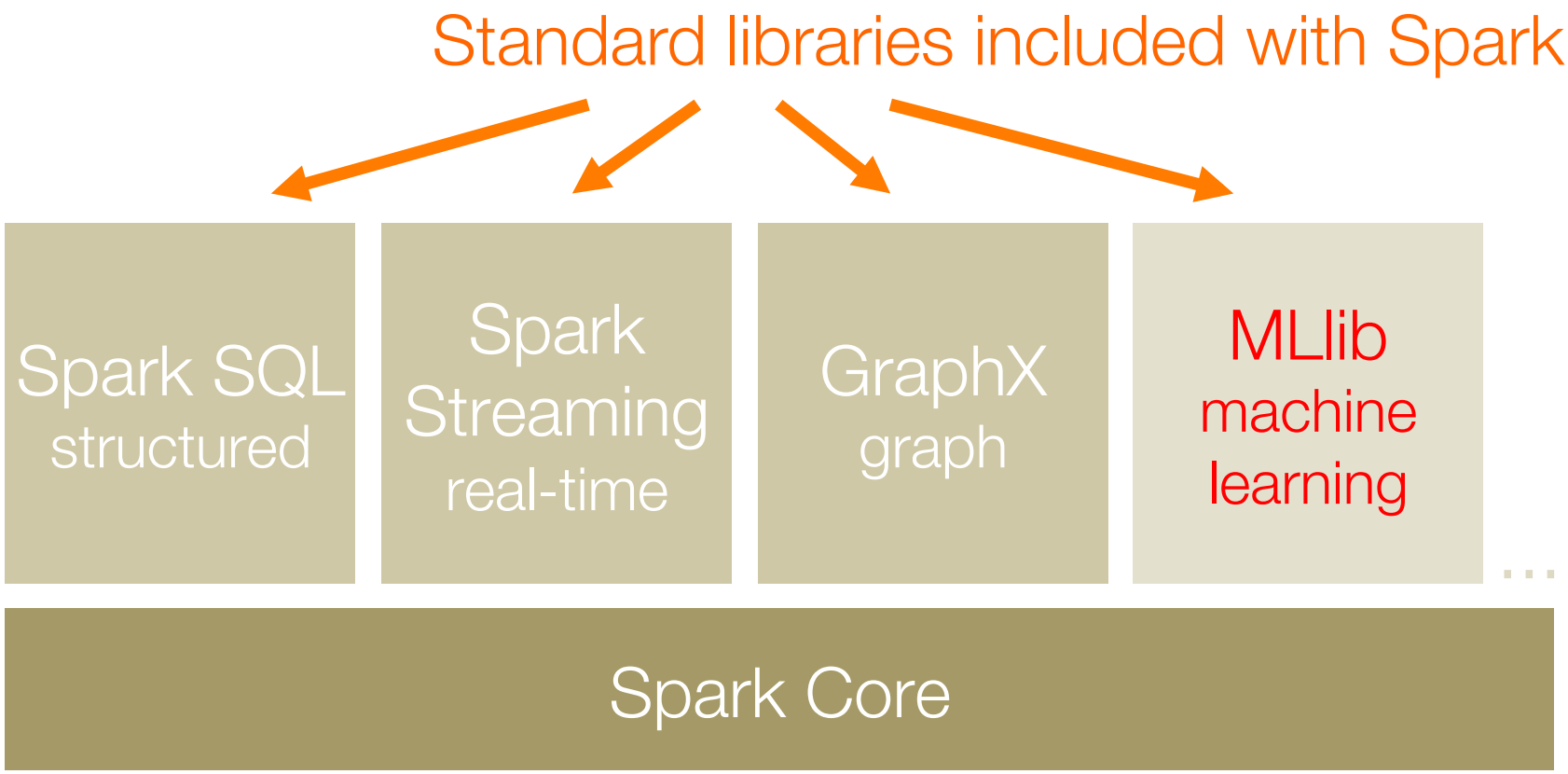
Spark SQL

```
// Run SQL statements
val teenagers = context.sql(
  "SELECT name FROM people WHERE age >= 13 AND age <= 19")


// The results of SQL queries are RDDs of Row objects
val names = teenagers.map(t => "Name: " + t(0)).collect()
```

MLlib + {Streaming, GraphX, SQL}

A General Platform



MLlib + Streaming

As of Spark 1.1, you can train linear models in a streaming fashion, k-means as of 1.2

Model weights are updated via SGD, thus amenable to streaming

More work needed for decision trees

MLlib + SQL

```
df = context.sql(“select latitude, longitude from tweets”)
model = pipeline.fit(df)
```

DataFrames in Spark 1.3! (March 2015)

Powerful coupled with new pipeline API

MLlib + GraphX

```
// assemble link graph
val graph = Graph(pages, links)
val pageRank: RDD[(Long, Double)] = graph.staticPageRank(10).vertices

// load page labels (spam or not) and content features
val labelAndFeatures: RDD[(Long, (Double, Seq((Int, Double))))] = ...
val training: RDD[LabeledPoint] =
  labelAndFeatures.join(pageRank).map {
    case (id, ((label, features), pageRank)) =>
      LabeledPoint(label, Vectors.sparse(features ++ (1000, pageRank))
}

// train a spam detector using logistic regression
val model = LogisticRegressionWithSGD.train(training)
```

Future of MLlib

Goals

Tighter integration with DataFrame and spark.ml API

Accelerated gradient methods & Optimization interface

Model export: PMML (current export exists in Spark 1.3, but not PMML, which lacks distributed models)

Scaling: Model scaling