

Apache Spark

(many slides derived from DataBricks workshop materials
under CC BY-NC-ND 4.0 license)



The City College
of New York



NYU

Center for Urban
Science + Progress

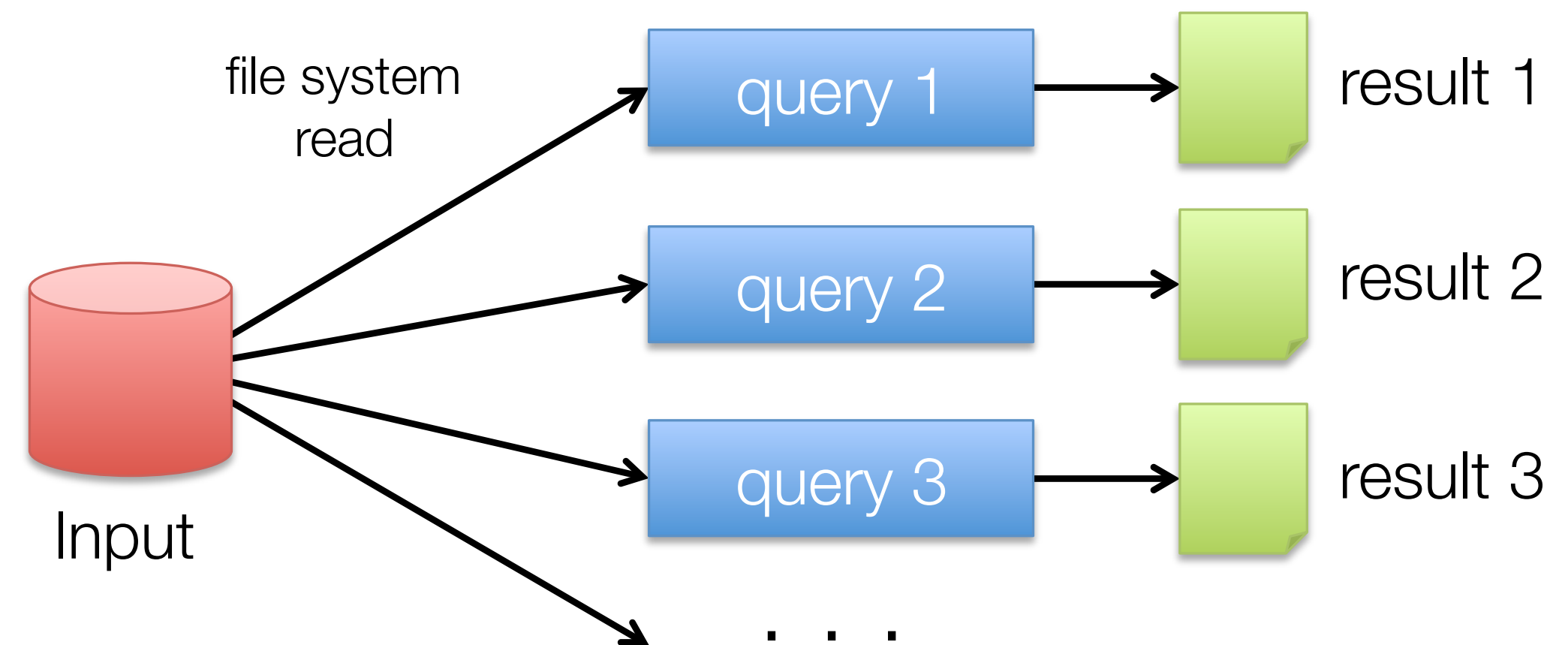
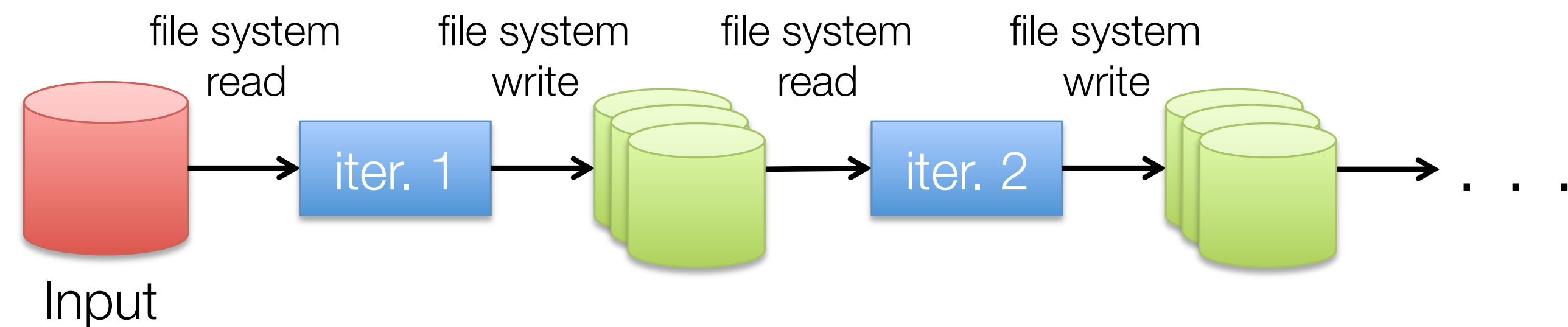
Hadoop is tedious. What do we want?

- Uniformly write all codes in Python (to transform data as well as “gluing” codes together)
 - Run in a notebook, passing data with Python’s native types
- Code should run on our local machines as well as a big cluster
- Easy to chain multiple jobs
 - Without wrapping around Map and Reduce
- ???

Apache Spark!

Apache Spark — Fixing MapReduce

- 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



Commonly spend 90% of time doing I/O

Apache Spark

- Favor memory more than disks (performance is #1)
 - Why does Hadoop need to write to disk so much?
- Handle batch, interactive, and real-time data within the same framework
- Improve performance and increase the flexibility in building dataflows
- Supports Python data model natively (along with Java and Scala)
- More general data constructs (beyond Map/Reduce)
- Open-source: active number of contributions surpassed Hadoop in 2013

Key distinctions of Spark vs. MapReduce

- A generalized execution engine
 - Using the same engine for various data constructs and use cases
- Lazy evaluation of lineage graph
 - Only execute dependencies when needed
- Use of Functional Language
- Lower overhead
- Less expensive shuffles

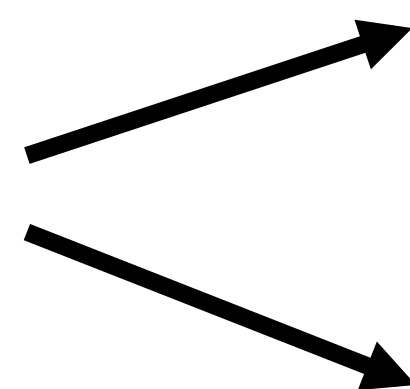
Benefits for Users

- Same engine can be used for parsing, training and querying data

Separate engines



Spark

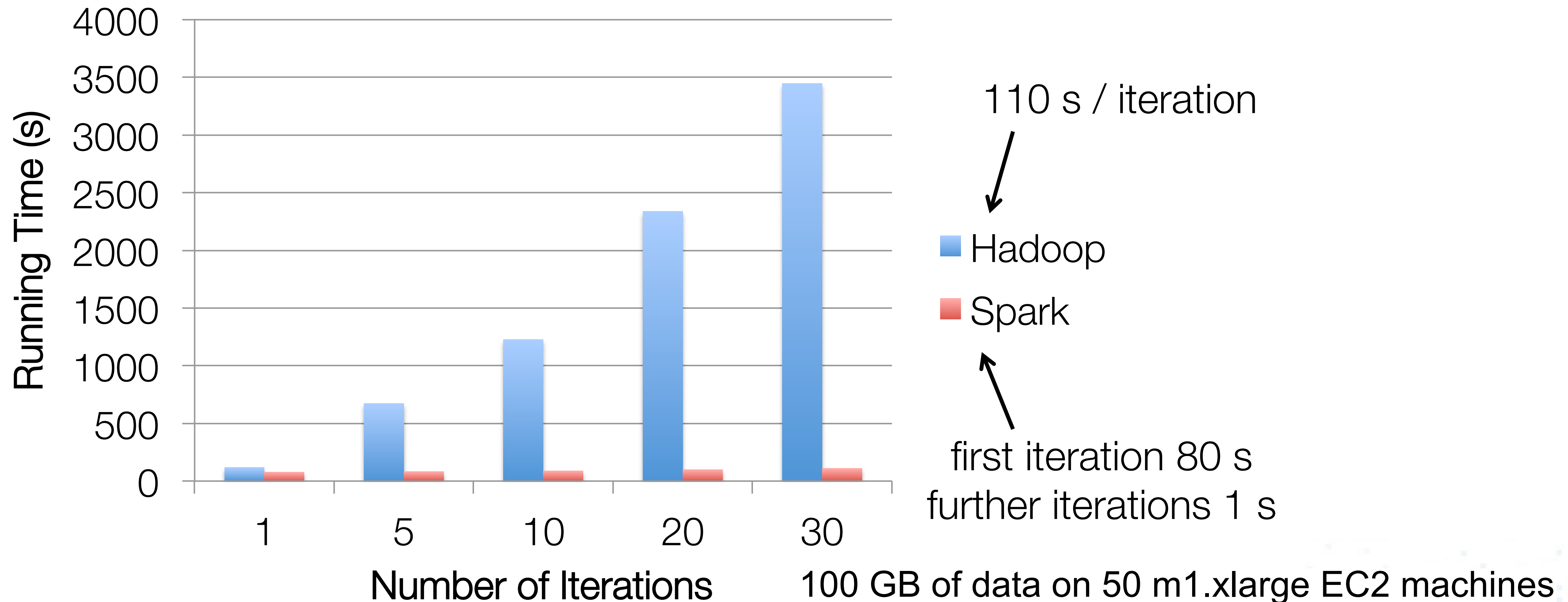


DFS

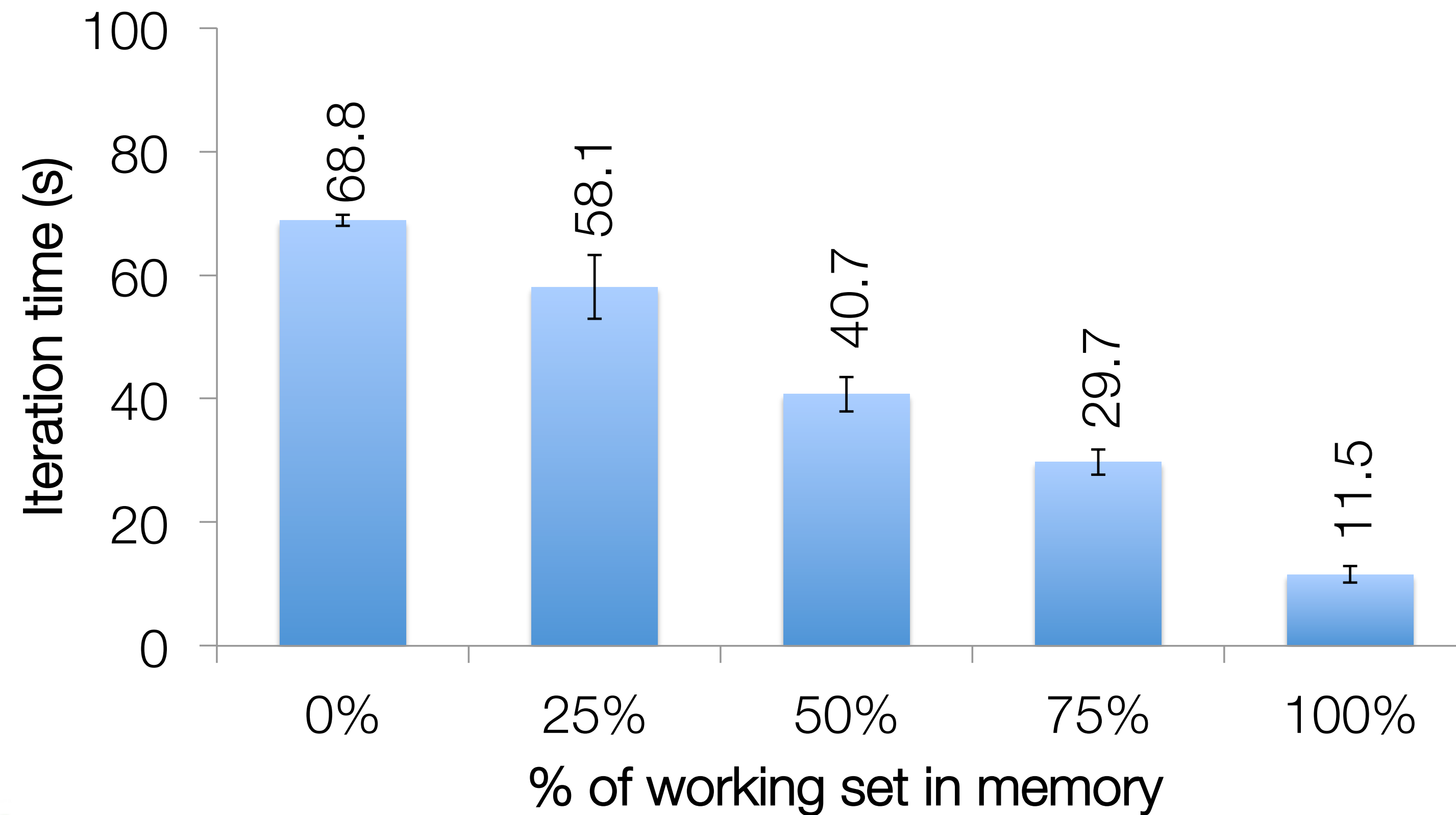
Raw Performance vs. MapReduce

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Logistic Regression: memory caching



Behavior with less RAM

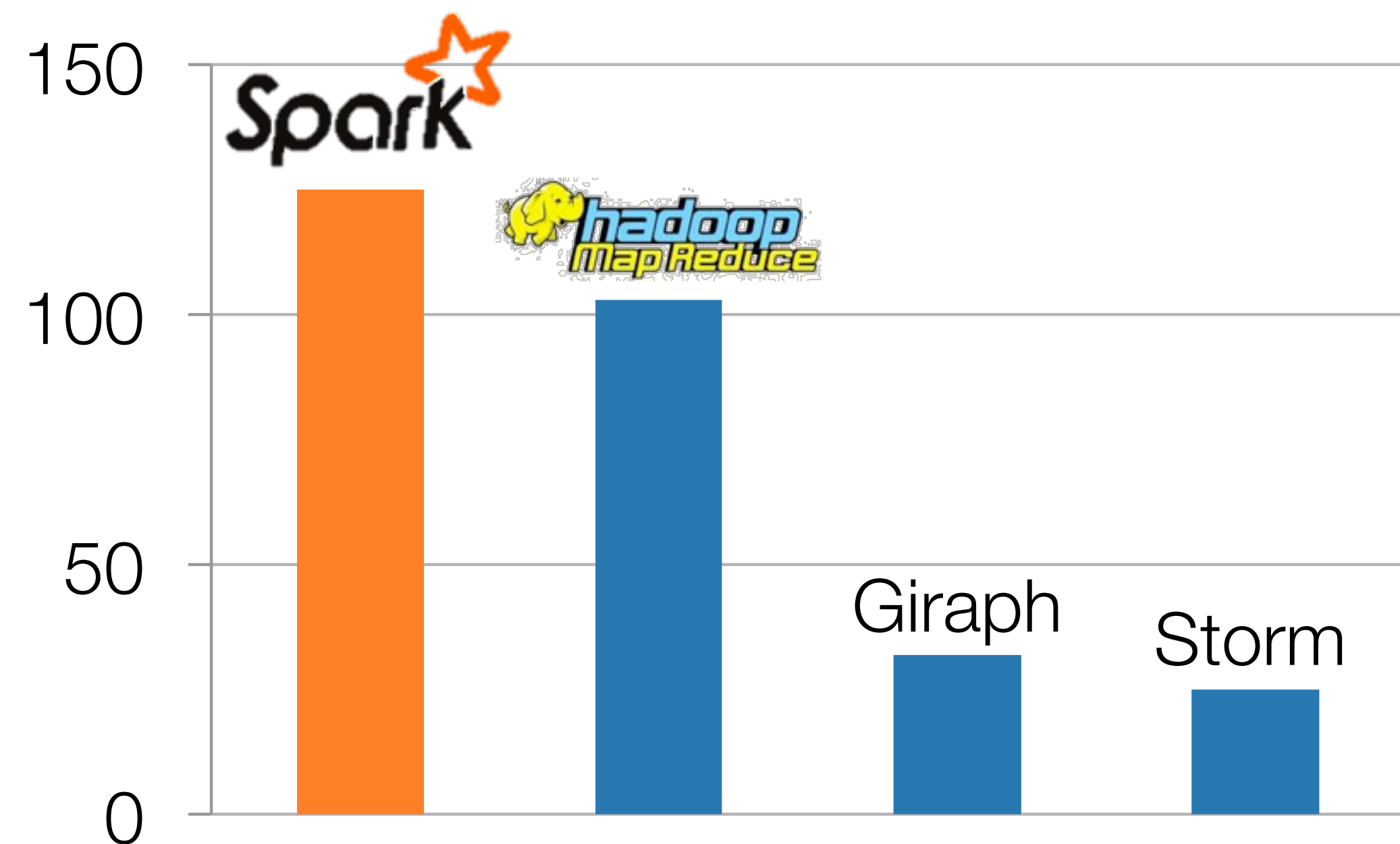


Spark Community

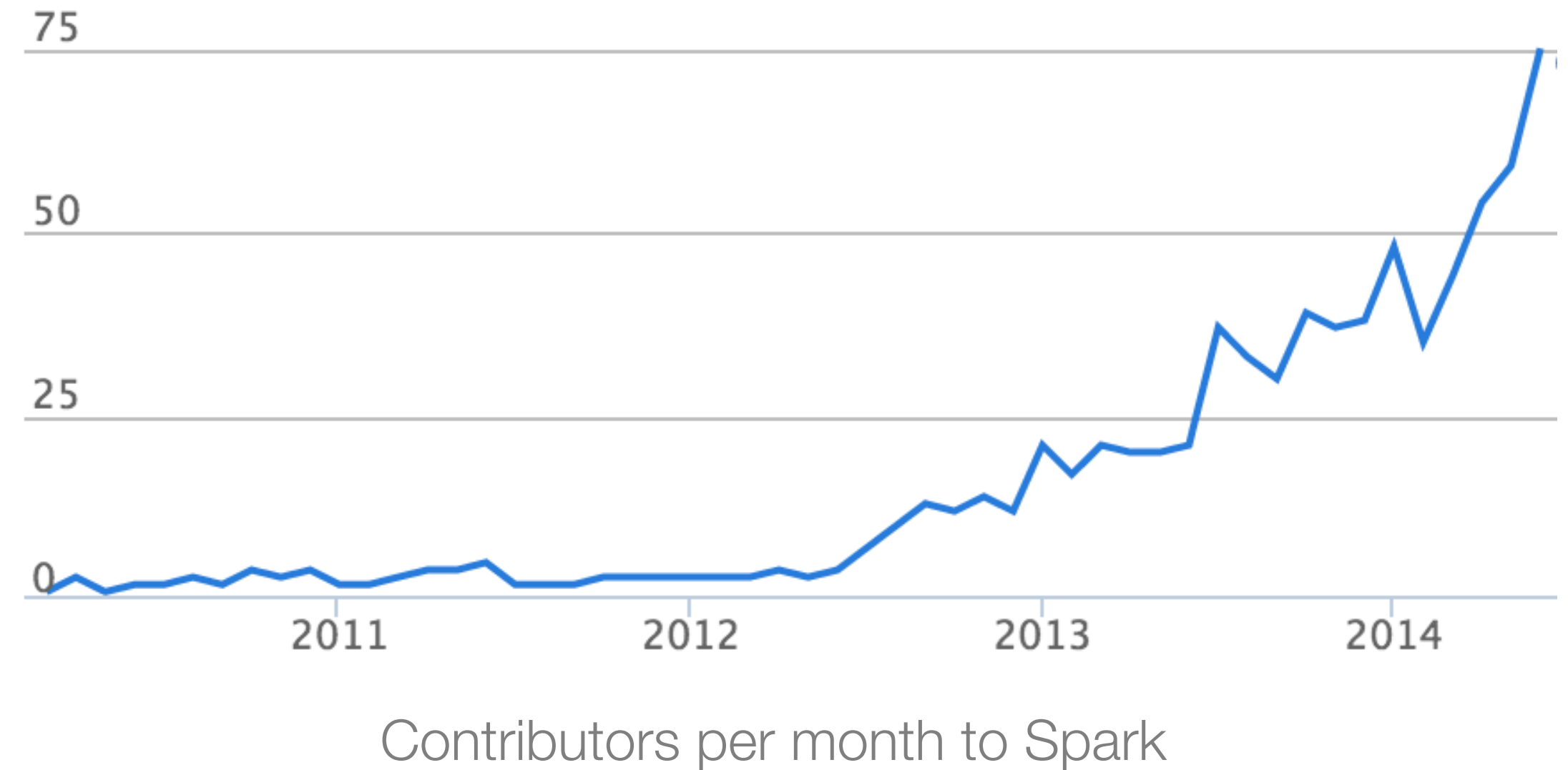
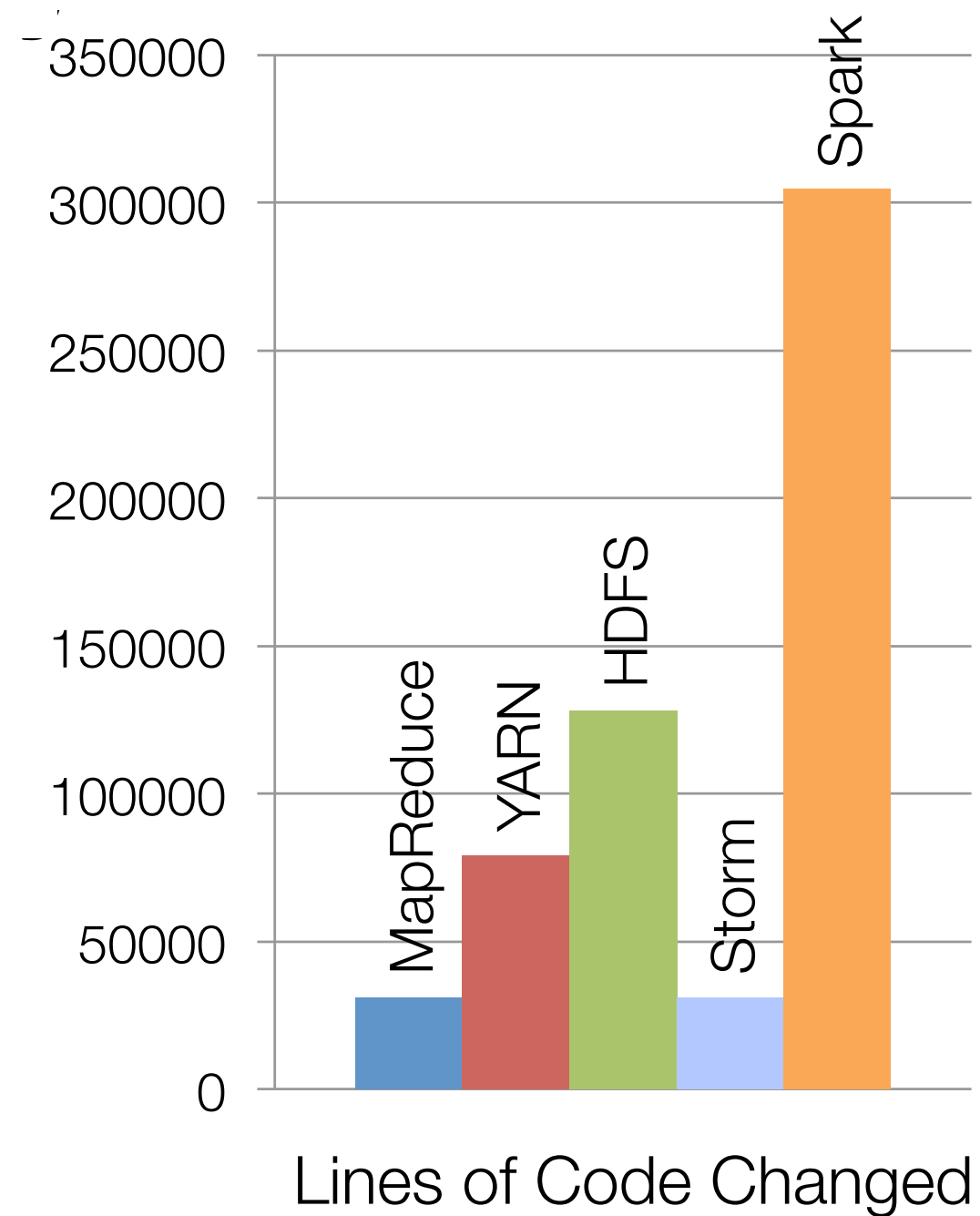
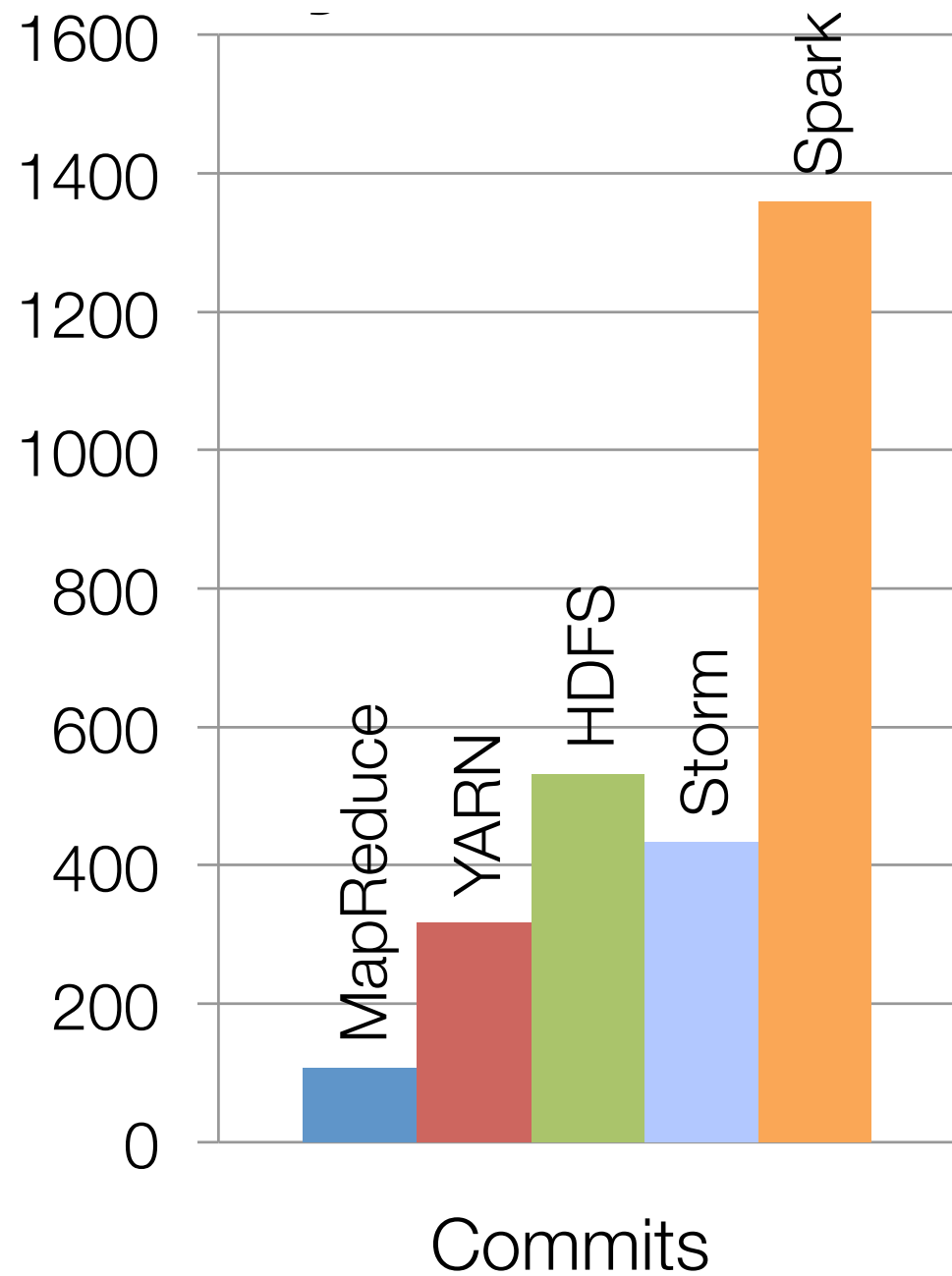
- Most active open source community in big data 200+ developers, 50+ companies contributing



Contributors in past year



Project Activity & Continuing Growth



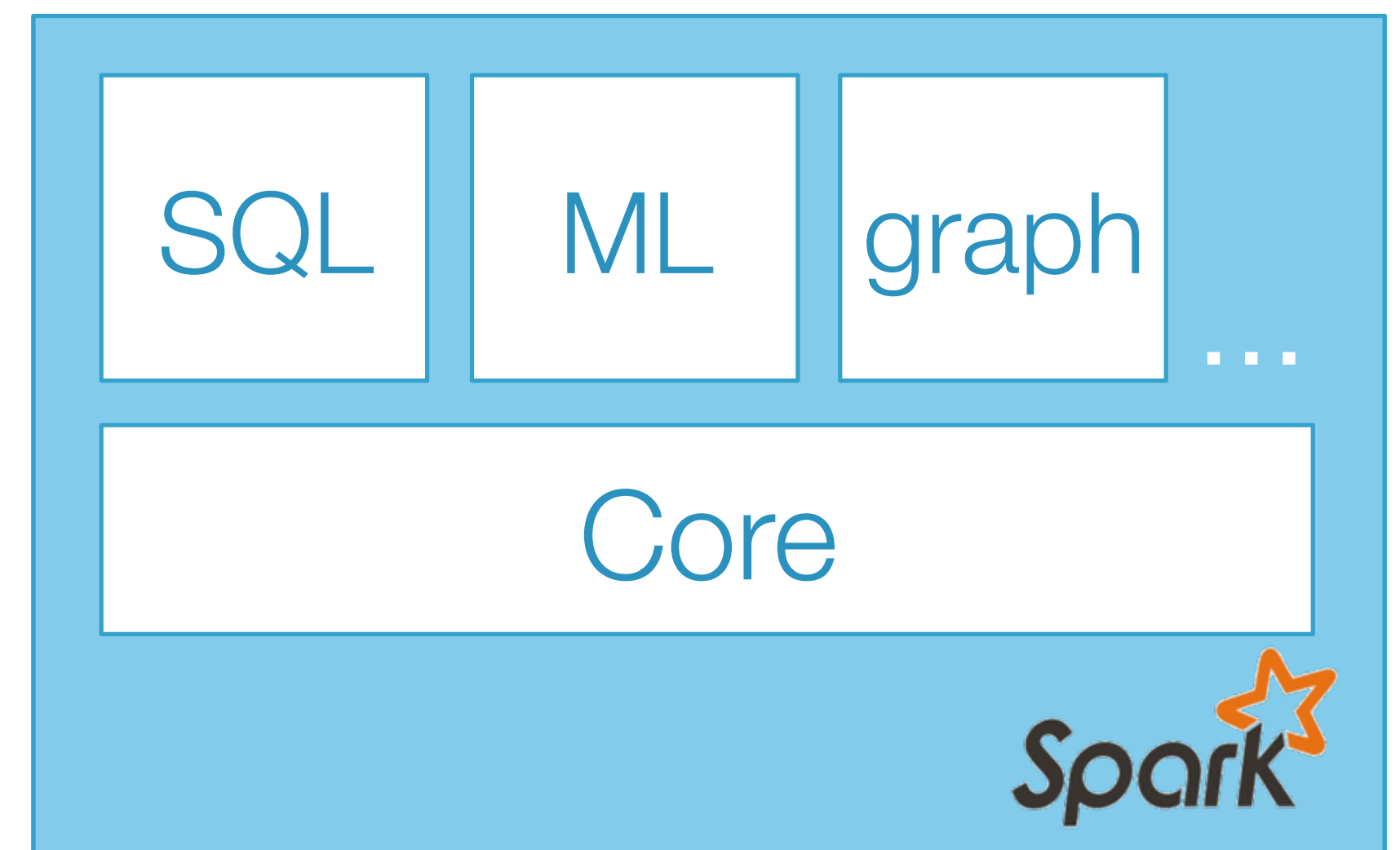
Activity in past 6 months

Standard Library included with Spark

Python Scala Java R

Big data apps lack libraries
of common algorithms

Spark's generality + support
for multiple languages make
suitable to offer this



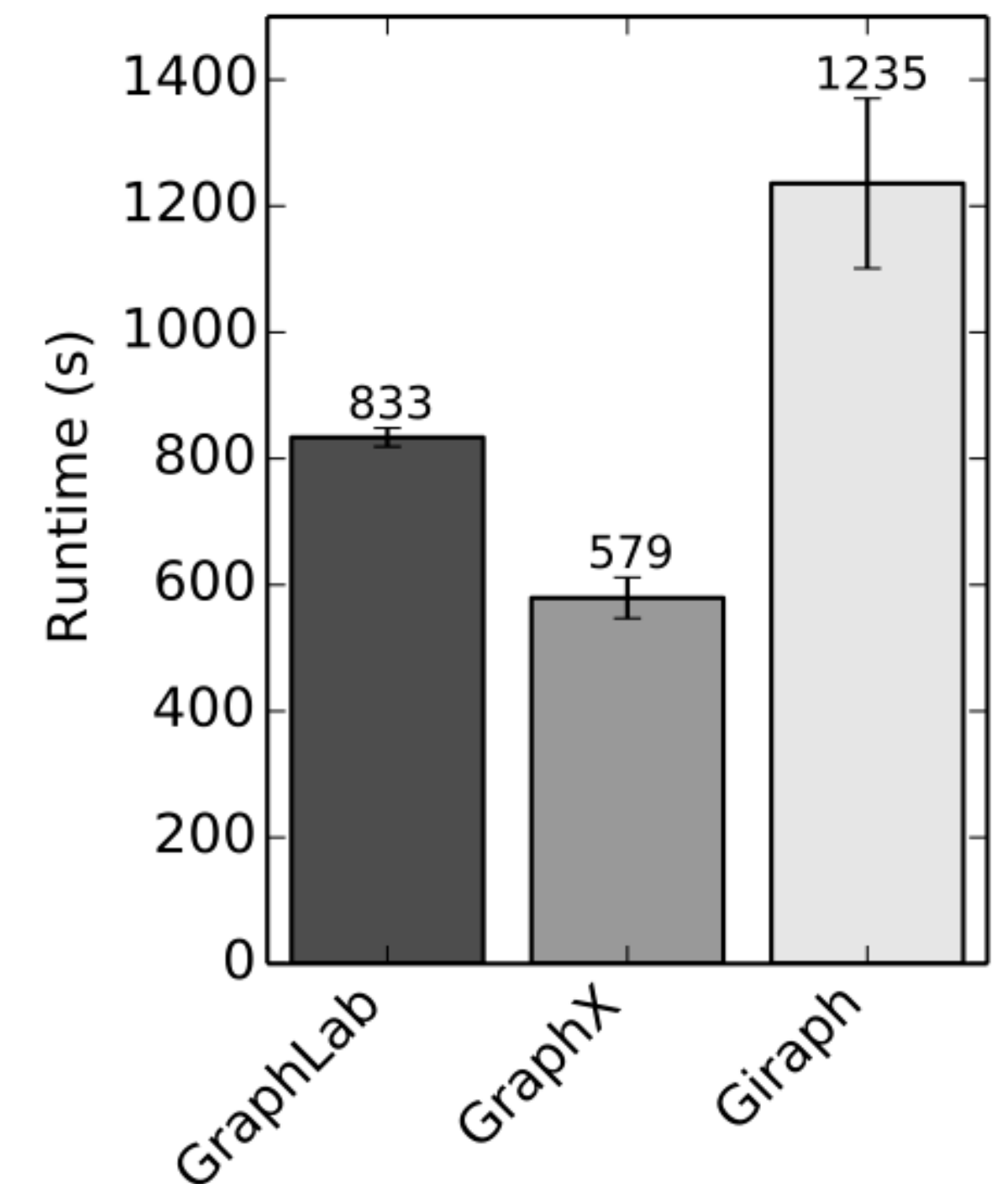
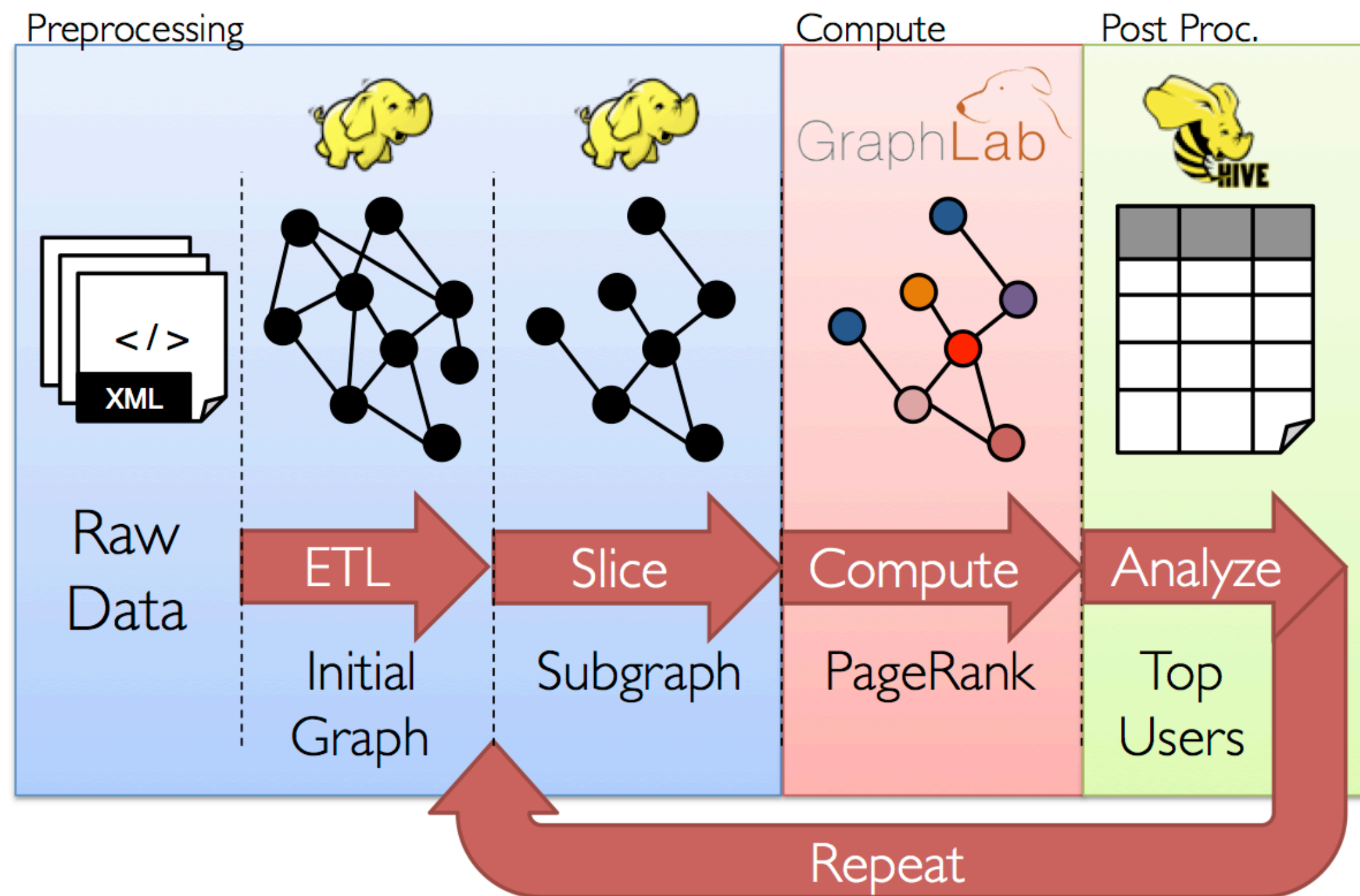
MLLib on Spark

For full list: <http://spark.apache.org/docs/latest/mllib-guide.html>

- 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** (Bisecting & Streaming), Gaussian Mixture, Power iteration clustering (PIC), Latent Dirichlet allocation (LDA)
- Decomposition: SVD, **PCA**

GraphX on Spark

- Build graph using RDDs of nodes and edges
- Large library of graph and graph-parallel algorithms
- High Performance



Spark SQL

- Enables loading & querying structured data in Spark

From Hive:

```
c = HiveContext(sc)
rows = c.sql("select text, year from hivetable")
rows.filter(lambda r: r.year > 2013).collect()
```

From JSON:

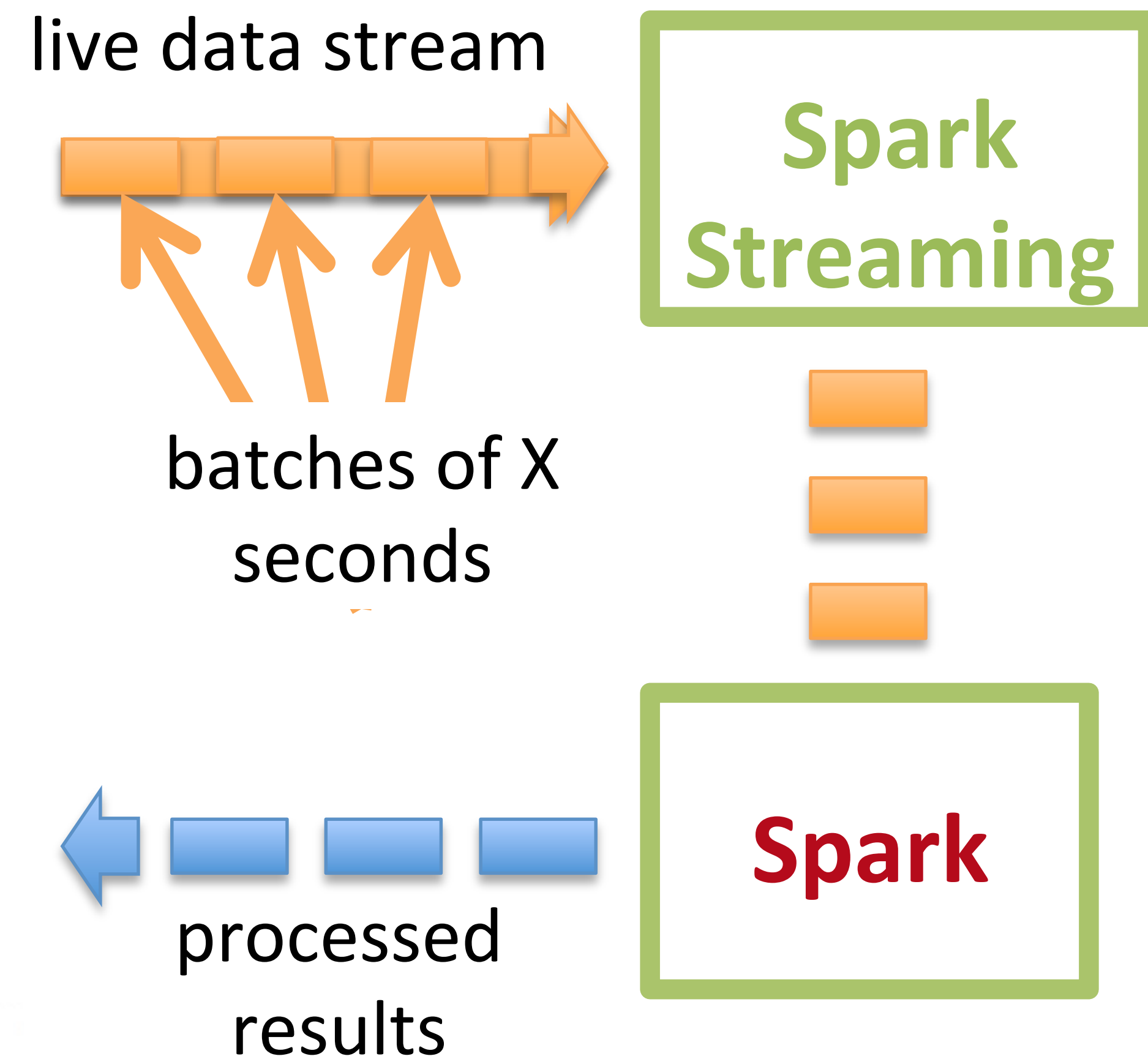
```
c.jsonFile("tweets.json").registerAsTable("tweets")
c.sql("select text, user.name from tweets")
```

tweets.json

```
{ "text": "hi",
  "user": {
    "name": "matei",
    "id": 123
  }
}
```

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

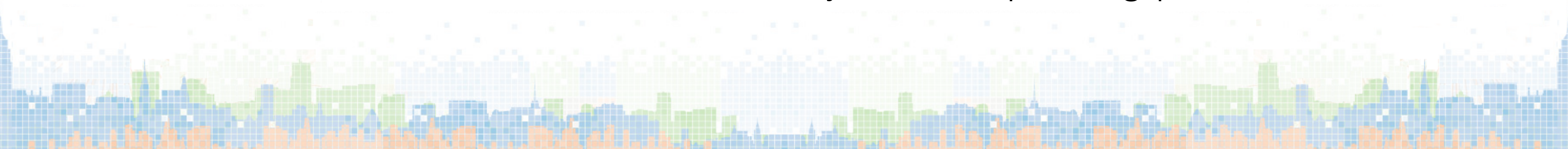


Next — Spark under the hood



Resilient Distributed Datasets (RDDs)

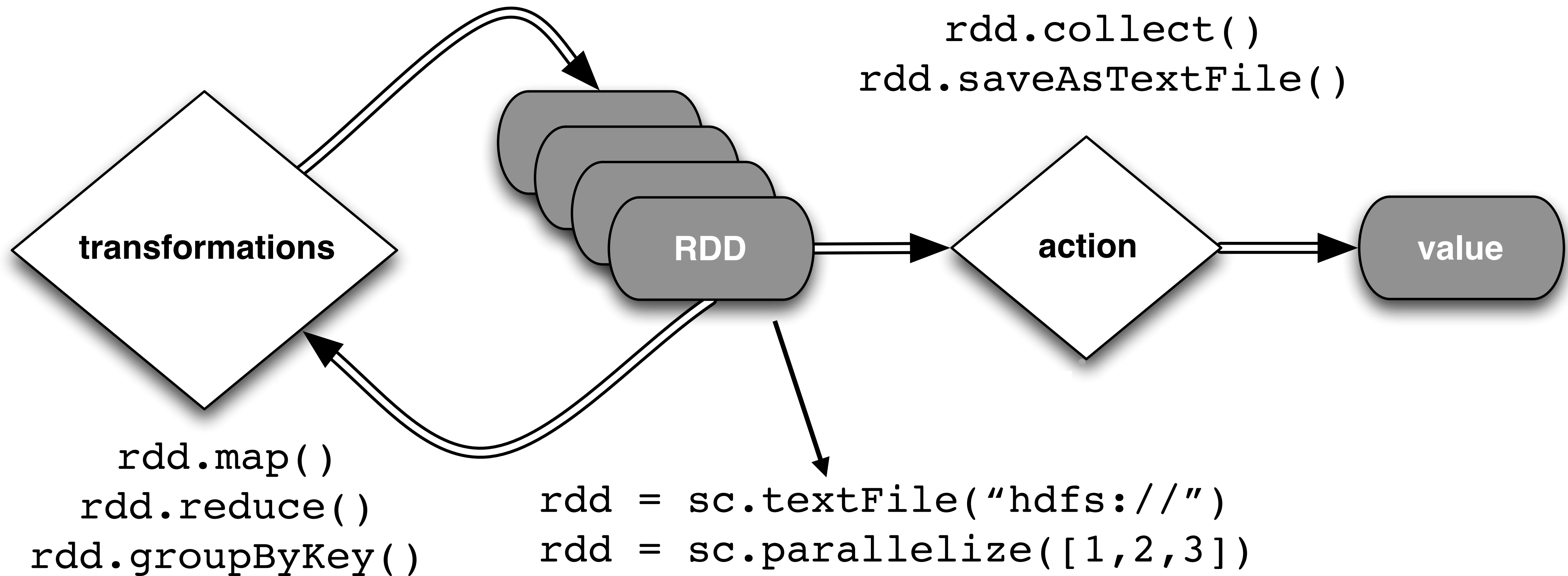
- The primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel
- Immutable collections of objects, spread across a cluster
 - user-controlled partitioning & storage
- RDDs can be automatically rebuilt on failure using lineage information
- Most important:
 - Contents can be cached in memory, thus, improving performance



Resilient Distributed Datasets (RDDs)

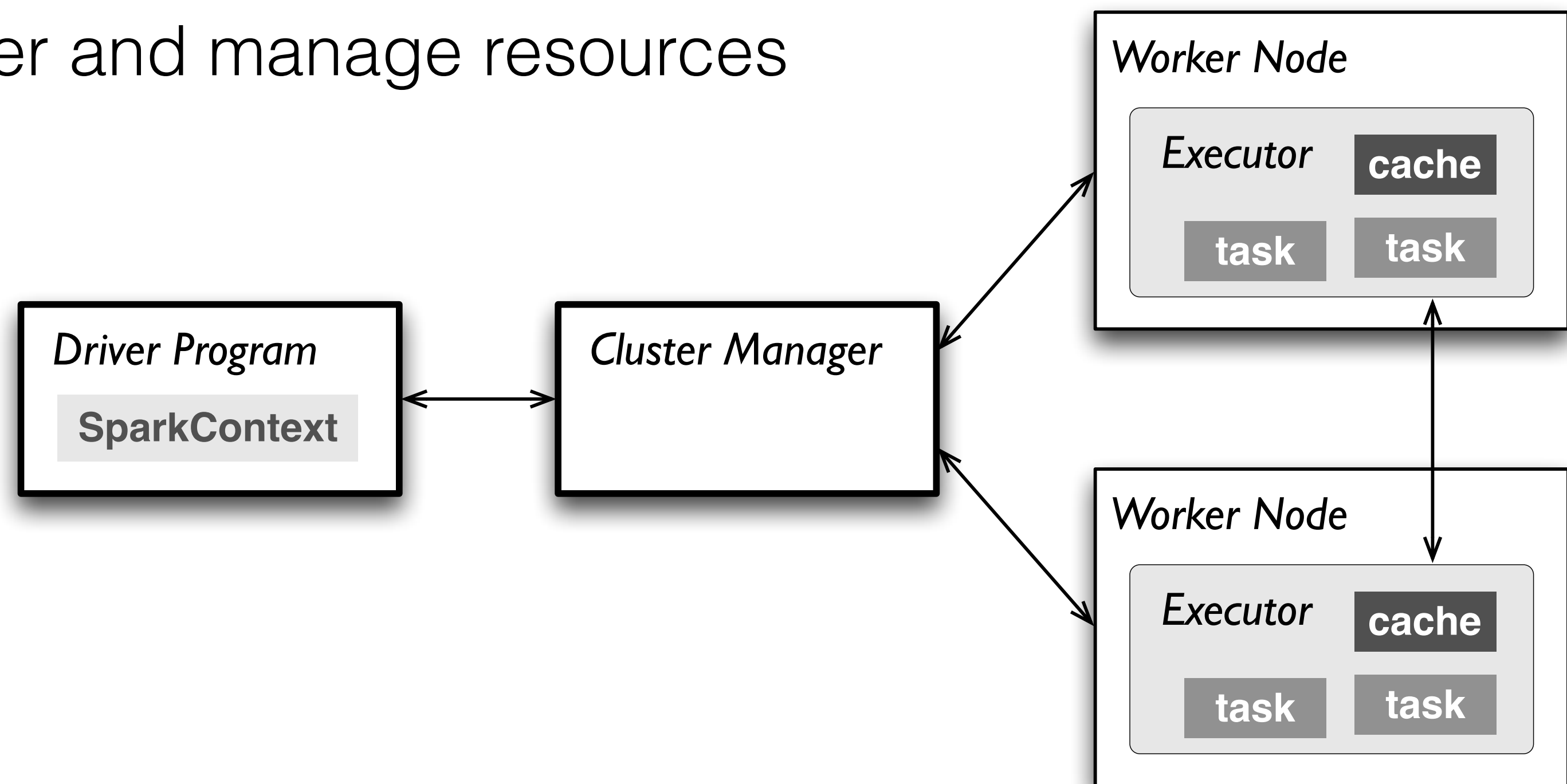
- Can be constructed from:
 - Parallelized containers (e.g. list)
 - Hadoop datasets: files on HDFS, Amazon S3, etc.
 - Transformations (map, filter, etc.) of another RDD
- Two types of operations can be done on RDDs
 - Transformation — results are RDDs
 - Actions — results are immediate data values or files on disks

General Dataflow of Spark



Job Execution in Spark

- Driver (master) and Worker nodes
- SparkContext — how to access cluster and manage resources
 - Acquiring executors
 - Send app code to executors
 - Python's objects are pickled
- Executors — run tasks
- Driver — control flow
 - post-processing of results



WordCount in Spark (Python)

- 4 lines of codes that can be tested on our local machine and runs on the cluster (by replacing the fileinput)

```
from operator import add
f = sc.textFile("README.md")
wc = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add)
wc.saveAsTextFile("wc_out")
```

closures



Understanding Closures

<http://spark.apache.org/docs/latest/programming-guide.html#understanding-closures-a-nameclosureslinka>

- To ensure maximum compatibility when running in cluster mode, closure must be *self-contained* — no global variables (except Spark's defined)!

```
counter = 0
rdd = sc.parallelize(data)

# Wrong: Don't do this!!
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)

print("Counter value: ", counter)
```

RDD's Transformations

- Transformations create a new RDD from an existing one
 - Chains of higher order functions!
- All transformations in Spark are lazy: they do not compute their results right away – instead they remember the transformations applied to some base dataset
 - optimize the required calculations
 - recover from lost data partitions
- Works with native Python data structures:
 - Key/Value pairs — just use a tuple!
 - OKAY to *yield* a Python's dictionary (instead of converting it to string)

RDD's Transformations

<i>transformation</i>	<i>description</i>
map (<i>func</i>)	return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
filter (<i>func</i>)	return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true
flatMap (<i>func</i>)	similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item)
sample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator <i>seed</i>
union (<i>otherDataset</i>)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct ([<i>numTasks</i>]))	return a new dataset that contains the distinct elements of the source dataset

RDD's Transformations — Key/Value Pairs

<i>transformation</i>	<i>description</i>
groupByKey ([<i>numTasks</i>])	when called on a dataset of (K , V) pairs, returns a dataset of (K , $\text{Seq}[V]$) pairs
reduceByKey (<i>func</i> , [<i>numTasks</i>])	when called on a dataset of (K , V) pairs, returns a dataset of (K , V) pairs where the values for each key are aggregated using the given reduce function
sortByKey ([<i>ascending</i>], [<i>numTasks</i>])	when called on a dataset of (K , V) pairs where K implements <code>Ordered</code> , returns a dataset of (K , V) pairs sorted by keys in ascending or descending order, as specified in the boolean <code>ascending</code> argument
join (<i>otherDataset</i> , [<i>numTasks</i>])	when called on datasets of type (K , V) and (K , W), returns a dataset of (K , (V , W)) pairs with all pairs of elements for each key
cogroup (<i>otherDataset</i> , [<i>numTasks</i>])	when called on datasets of type (K , V) and (K , W), returns a dataset of (K , $\text{Seq}[V]$, $\text{Seq}[W]$) tuples — also called <code>groupWith</code>
cartesian (<i>otherDataset</i>)	when called on datasets of types T and U , returns a dataset of (T , U) pairs (all pairs of elements)

map vs. flatMap vs. mapPartitions

- *map* — one to one mapping
 - Python's *map()*
- *flatMap* — one to many (none included) mapping
 - `mapreduce.py` — `mr.run's mapper()`
- *mapPartitions* — many to many mapping — **the most efficient!**
 - Hadoop Streaming *mapper()*
 - `rdd.mapPartitions(mapper).groupByKey().map(reducer)`

RDD's Actions

<i>action</i>	<i>description</i>
reduce (<i>func</i>)	aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect ()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count ()	return the number of elements in the dataset
first ()	return the first element of the dataset – similar to <i>take(1)</i>
take (<i>n</i>)	return an array with the first <i>n</i> elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, using the given random number generator seed

RDD's Actions

<i>action</i>	<i>description</i>
saveAsTextFile (<i>path</i>)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file
saveAsSequenceFile (<i>path</i>)	write the elements of the dataset as a Hadoop <code>SequenceFile</code> in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's <code>Writable</code> interface or are implicitly convertible to <code>Writable</code> (Spark includes conversions for basic types like <code>Int</code> , <code>Double</code> , <code>String</code> , etc).
countByKey ()	only available on RDDs of type <code>(K, V)</code> . Returns a <code>'Map'</code> of <code>(K, Int)</code> pairs with the count of each key
foreach (<i>func</i>)	run a function <i>func</i> on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD's Actions

- An action applied on an RDD trigger the “*lazy*” transformation of that RDD — with potential re-computations
- RDDs can be persistent, aka *cached*, to avoid these expansive recomputations
 - Only necessary if to run multiple actions/transformations on similar RDDs

```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).cache()
w.reduceByKey(add).collect()
```


Broadcast Variables

- Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- For example, to give every node a copy of a large input dataset efficiently
- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

```
broadcastVar = sc.broadcast(list(range(1, 4)))  
broadcastVar.value
```

Accumulators

- Accumulators are variables that can only be “added” to through an associative operation
- Used to implement counters and sums, efficiently in parallel
- Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types
- Only the driver program can read an accumulator’s value, not the tasks

```
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x

rdd.foreach(f)

accum.value
```


Spark SQL — DataFrame



Challenges with Functional API

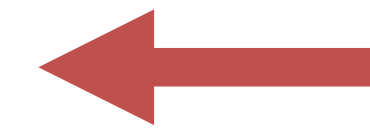
- Looks high-level, but hides many semantics of computation from engine
 - Functions passed in are arbitrary blocks of code
 - Data stored is arbitrary Java/Python objects
- Users can mix APIs in suboptimal ways
- Need a more efficient way to deal with structured data — like *pandas*



Suboptimal Execution Plan

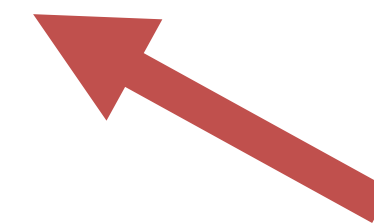
```
pairs = data.map(lambda word: (word, 1))
```

```
groups = pairs.groupByKey()
```



Materializes all groups
as lists of integers

```
groups.map(lambda k_vs: (k_vs[0], sum(k_vs[1])))
```

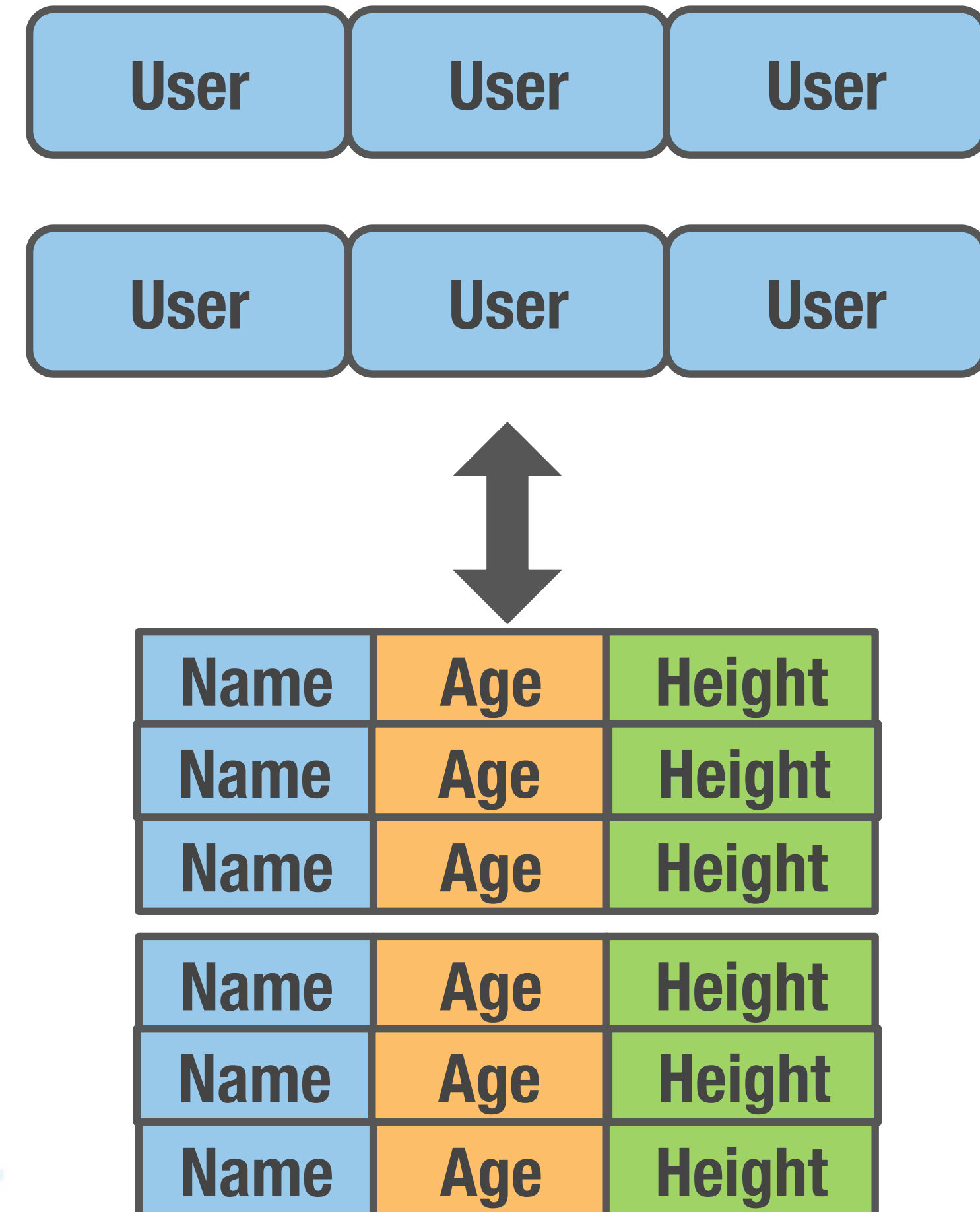


Then promptly
aggregates them



Spark SQL on DataFrame

- Efficient library for working with structured data
 - Two interfaces: DataFrames for columnar storage, SQL for data analysts and external applications
 - Optimized computation and storage underneath
- DataFrame: similar to *Pandas*, RDD + schema
- SparkSQL: DataFrame + SQL



DataFrame API

- DataFrames hold rows with a known **schema** and offer relational operations through a DSL
- Basic data structure for ML
- Easy to convert between RDDs and DataFrame

```
c = HiveContext() # or just sqlContext
users = c.sql("select * from users")

ma_users = users[users.state == "MA"]
ma_users.count()
ma_users.groupBy("name").avg("age")
ma_users.map(lambda row: row.user.toUpperCase())
```

Expression AST

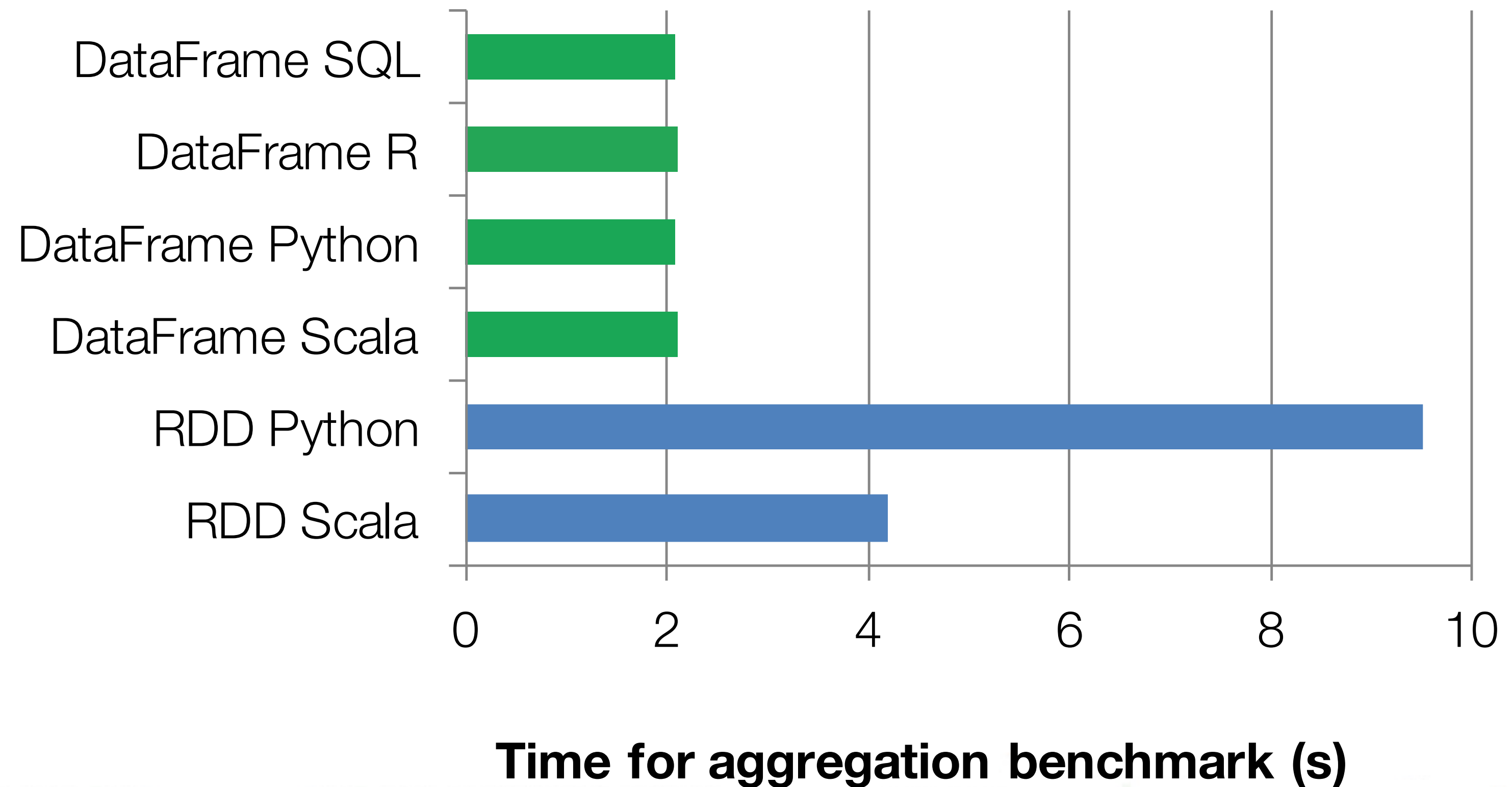
What DataFrame Enables

- Compact binary representation
 - Columnar, compressed cache; rows for processing
- Optimization across operators (join reordering, predicate pushdown, etc)
- Runtime code generation



SparkSQL Performance

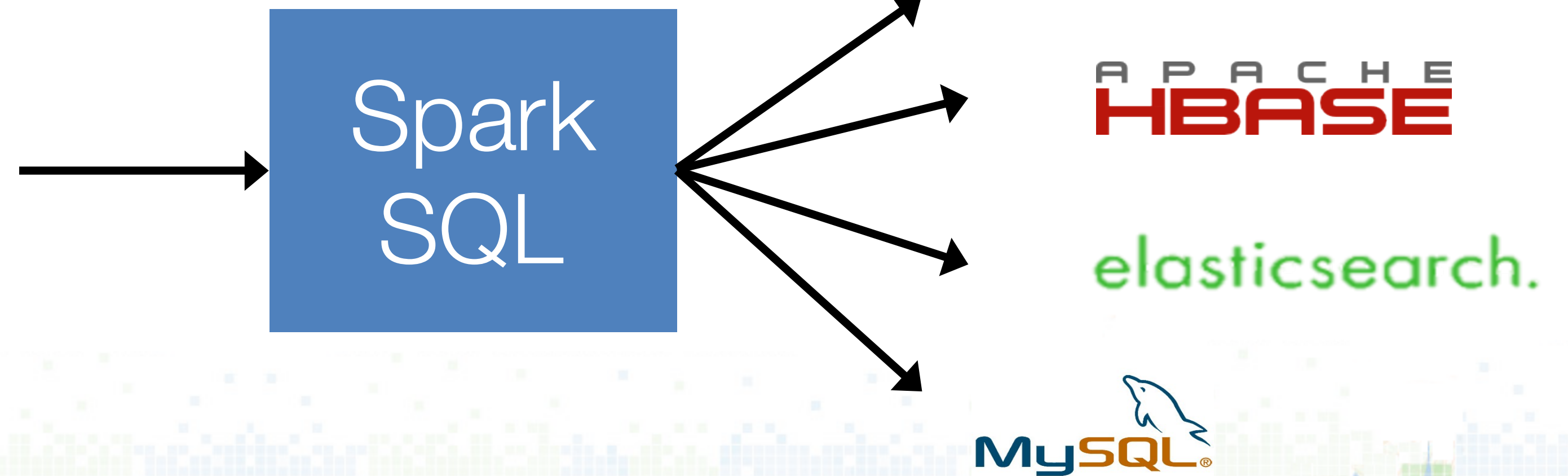
- ~5 times faster, thanks to
 - overhead
 - bad materialization
 - suboptimal parallelism



Data Sources

- Uniform way to access structured data
 - Apps can migrate across Hive, Cassandra, JSON, ...
 - Rich semantics allows query pushdown into data sources

```
users[users.age > 20]  
select * from users
```



Example: find the median using *less* code

```
lines = sc.textFile('citibike.csv')
minutes = lines.map(lambda x: (int(x.split(',')[2])/60,))
df = sqlContext.createDataFrame(minutes, ('minutes',)).cache()
df.registerTempTable('records')
sqlContext.sql('select percentile(minutes, 0.5) from records').collect()
```


DataFrame Construction

- Read directly from file/data source (using inferSchema ~ read twice)

```
df = spark.read.load('citibike.csv', format='csv',  
                    header=True, inferSchema=False)
```

```
df.dtypes
```

```
[('cartodb_id', 'string'),  
 ('the_geom', 'string'),  
 ('tripduration', 'string'),  
 ('starttime', 'string'),  
 ('stoptime', 'string'),  
 ('start_station_id', 'string'),  
 ('start_station_name', 'string'),
```


DataFrame Construction

- Read directly from file/data source (using inferSchema ~ read twice)

```
df = spark.read.load('citibike.csv', format='csv',  
                    header=True, inferSchema=True)
```

```
df.dtypes
```

```
[('cartodb_id', 'int'),  
 ('the_geom', 'string'),  
 ('tripduration', 'int'),  
 ('starttime', 'string'),  
 ('stoptime', 'string'),  
 ('start_station_id', 'int'),  
 ('start_station_name', 'string'),
```

DataFrame Construction

- Read directly from file/data source in JSON strings:

```
df = spark.read.load('twitter_1k.jsonl', format='json')
```

```
hashtags = df.select('entities.hashtags.text')
```

- Expected format: JSON Lines (<http://jsonlines.org/>)
 - A text file
 - Each line is a complete JSON string
 - All 'end of line' characters must be removed from JSON strings

DataFrame Construction

- From an RDD of Row objects with a schema inferred from reflection

```
def parseCSV(idx, part):  
    if idx==0:  
        part.next()  
    for p in csv.reader(part):  
        yield Row(tripduration=int(p[2]),  
                  starttime=p[3],  
                  start_station_name=p[6])  
  
rows = sc.textFile('citibike.csv') \  
        .mapPartitionsWithIndex(parseCSV)  
df = sqlContext.createDataFrame(rows)  
df.dtypes  
[('start_station_name', 'string'),  
 ('starttime', 'string'),  
 ('tripduration', 'bigint')]
```

DataFrame Construction

- From an RDD of tuples with a schema inferred programmatically

```
def parseCSV(idx, part):  
    if idx==0:  
        part.next()  
    for p in csv.reader(part):  
        yield (int(p[2]), p[3], p[6])  
  
rows = sc.textFile('citibike.csv').mapPartitionsWithIndex(parseCSV)  
schema = StructType([StructField('tripduration', IntegerType()),  
                      StructField('starttime', StringType()),  
                      StructField('start_station_name', StringType())])  
df = sqlContext.createDataFrame(rows, schema)  
df.dtypes  
[('tripduration', 'int'),  
 ('starttime', 'string'),  
 ('start_station_name', 'string')]
```


DataFrame Construction

- From an RDD of tuples with a schema inferred from reflection

```
def parseCSV(idx, part):  
    if idx==0:  
        part.next()  
    for p in csv.reader(part):  
        yield (int(p[2]), p[3], p[6])  
  
rows = sc.textFile('citibike.csv').mapPartitionsWithIndex(parseCSV)  
df = sqlContext.createDataFrame(rows,  
    ('tripduration', 'starttime', 'start_station_name'))
```

Spark SQL

- Can cache tables in memory
 - cache it (by calling `cache()`) if you're going to query multiple times
- Can read data stored in Hive
 - Ingest the data into Hive (instead of loading from file) if you're going to use it often
- Can quickly register UDFs (easier than using Pig):

```
registerFunction("countMatches", lambda (pattern, text):...)
sqlContext.sql("SELECT countMatches('a', text)...")
```



All Together Examples

```
from pyspark.sql import SQLContext, Row
sqlCtx = SQLContext(sc)

# Load a text file and convert each line to a dictionary
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the SchemaRDD as a table.
peopleTable = sqlCtx.inferSchema(people) peopleTable.registerTempTable("people")

# SQL can be run over SchemaRDDs that have been registered as a table
teenagers = sqlCtx.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
teenNames = teenagers.map(lambda p: "Name: " + p.name)
teenNames.collect()
```



Thanks!

