Data-Intensive Distributed Computing

CS 451/651 431/631 (Winter 2018)

Mix of slides from:

- Reza Zadeh http://reza-zadeh.com
- Jimmy Lin's course at UWaterloo: http://lintool.github.io/bigdata-2018w/

So far in 416

Focused on distributed coordination

- Distributed algorithms: consensus, atomic commitment, mutual exclusion,…
- Distributed systems: CDN, DFS, BT, BChains, Kademlia, ..

What about programmability?

So far in 416

Focused on distributed coordination

- Distributed algorithms: consensus, atomic commitment, mutual exclusion,…
- Distributed systems: CDN, DFS, BT, BChains, Kademlia, ..

What about programmability?

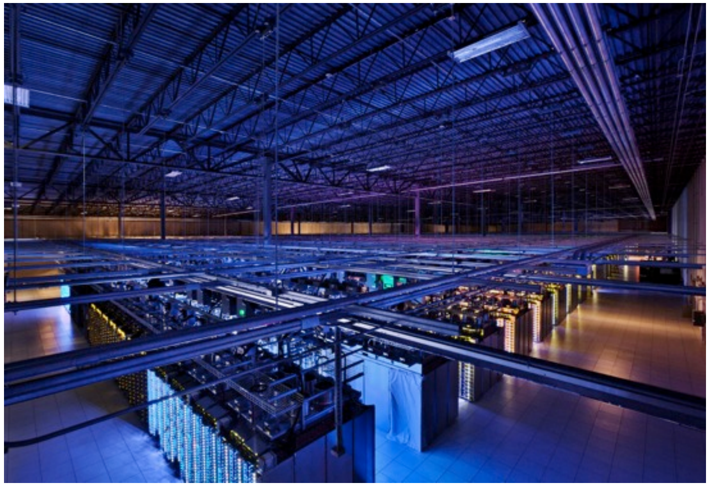
Well, there is RPC. What, is that not enough?

Reality check

Data growing faster than processing speeds

Only solution is to parallelize on large clusters

» Widely use in both enterprises and web industry



Reality check

Data growing faster than processing speeds

Only solution is to parallelize on large clusters

» Widely use in both enterprises and web industry

How do we let *regular (non 416) developers* program these things?

Processes 20 PB a day (2008)
Crawls 20B web pages a day (2012)
Search index is 100+ PB (5/2014)
Bigtable serves 2+ EB, 600M QPS (5/2014)

400B pages, 10+ PB (2/2014)

19 Hadoop clusters: 600 PB, 40k servers (9/2015)

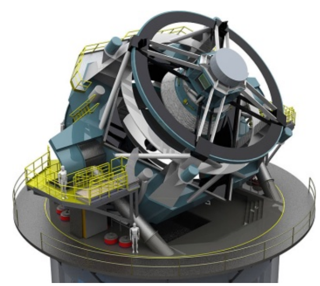
150 PB on 50k+ servers running 15k apps (6/2011)

Hadoop: 10K nodes, 150K cores, 150 PB (4/2014)

300 PB data in Hive + 600 TB/day (4/2014)

LHC: ~15 PB a year

S3: 2T objects, 1.1M request/second (4/2013)



LSST: 6-10 PB a year (~2020)

640K ought to be enough for anybody.

SKA: 0.3 – 1.5 EB per year (~2020)

How much data?


The datacenter *is* the computer!
Source: Google

Traditional Dist. computing

Message-passing between nodes: RPC, 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)

» Heterogeneity of nodes, their locations, complex env

» Have to write programs for each machine

Traditional Dist. computing

Message-passing between nodes: RPC, MPI, …

Very difficult to do at scale:

Rarely used in commodity datacenters

- » How to deal with failures? (inevitable at scale)
- » Even worse: stragglers (node not failed, but slow)
- » Heterogeneity of nodes, their locations, complex env
- » Have to write programs for each machine

Traditional Dist. computing

Message-passing between nodes: RPC, MPI, …

Very difficult to do at scale:

Rarely used in commodity datacenters

» How to deal with failures? (inevitable at scale)

Key question: *how do we let developers leverage distribution without having them build a distributed system per use case?*

The datacenter *is* the computer!

It's all about the right level of abstraction

Moving beyond the von Neumann architecture

What's the "instruction set" of the datacenter computer?

Hide system-level details from the developers

No more race conditions, lock contention, etc.

No need to explicitly worry about reliability, fault tolerance, etc.

Separating the *what* from the *how*

Developer specifies the computation that needs to be performed

Execution framework ("runtime") handles actual execution

MapReduce is the first instantiation of this idea… but not the last!

Data Flow Models

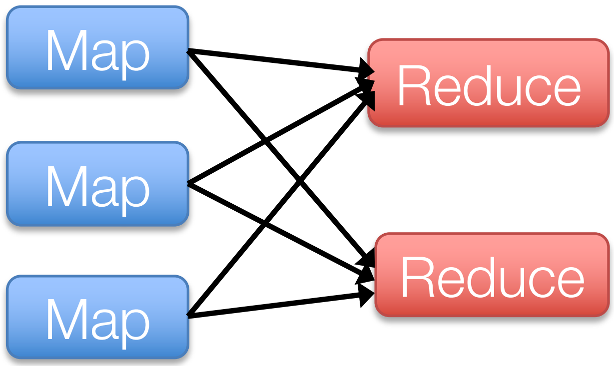
Restrict the programming interface so that the system can do more automatically

Express jobs as graphs of high-level operators

» System picks how to split each operator into tasks and where to run each task

» Re-run parts for fault recovery

Best example: MapReduce



Why Use a Data Flow Engine?

Ease of programming

- » High-level functions instead of message passing

Wide deployment

- » More common than MPI, especially “near” data

Scalability to huge commodity node clusters

- » Even HPC world is now concerned about resilience

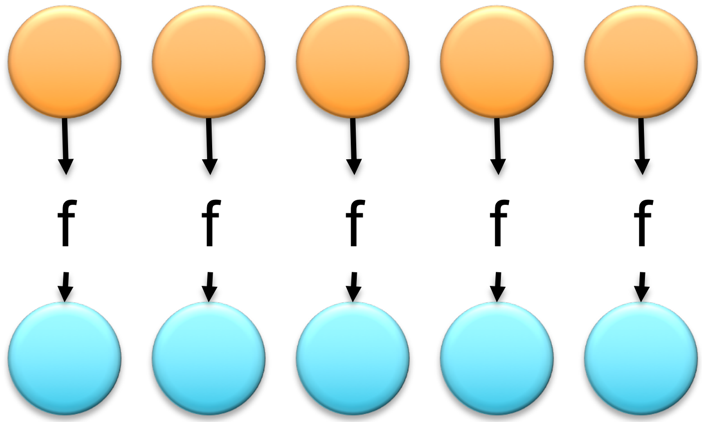
Examples: Pig, Hive, Storm, but initially publicized with MapReduce

Roots in Functional Programming

Simplest data-parallel abstraction

Process a large number of records: "do" something to each

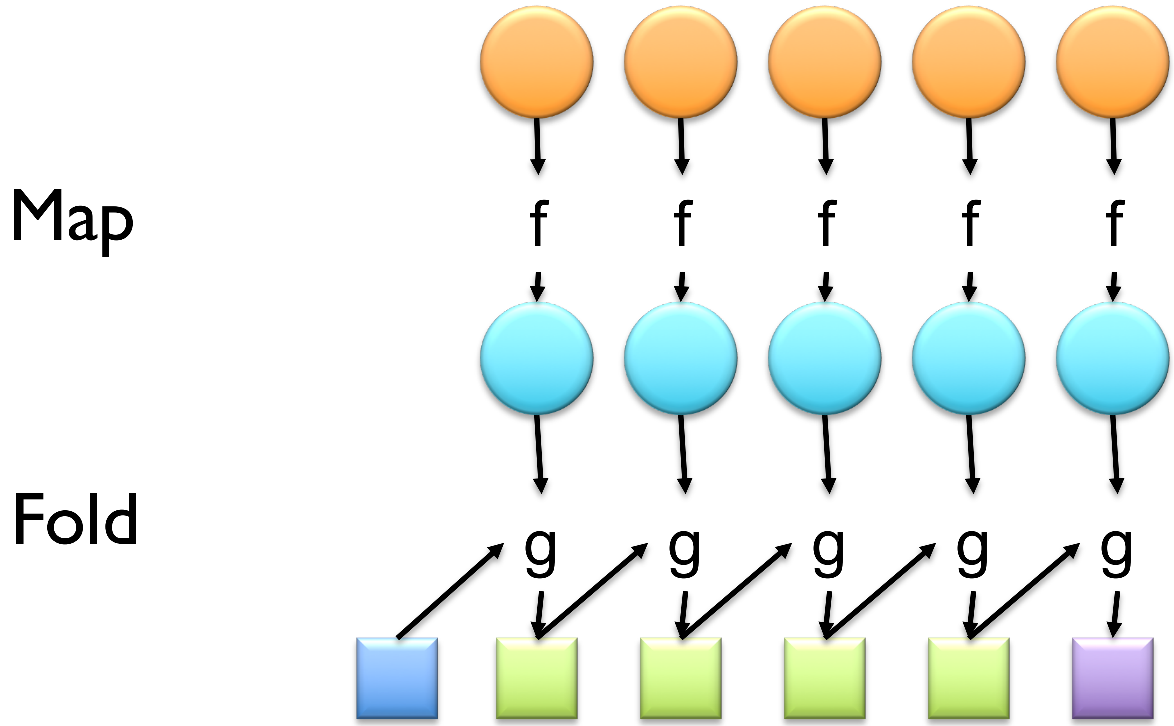
Map



We need something more for sharing partial results across records!

Roots in Functional Programming

Let's add in aggregation!



MapReduce = Functional programming + distributed computing!

A Data-Parallel Abstraction

Process a large number of records

Map "Do something" to each

Group intermediate results

"Aggregate" intermediate results Reduce

Write final results

Key idea: provide a functional abstraction for these two operations

MapReduce

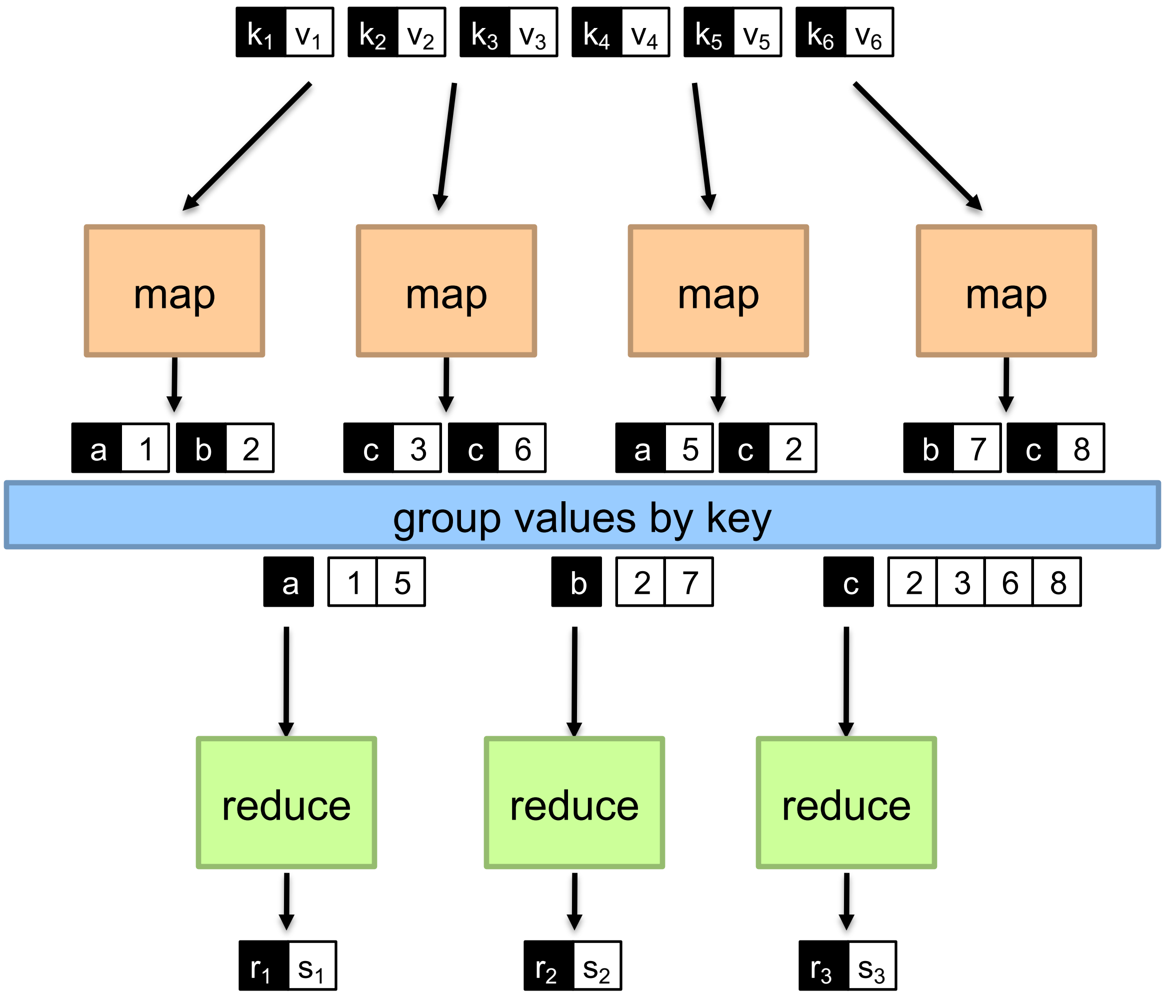
Programmer specifies two functions:

$$\texttt{map}\ (k_1, v_1) \rightarrow \mathrm{List}[(k_2, v_2)]$$

$$\texttt{reduce}\ (k_2, \mathrm{List}[v_2]) \rightarrow \mathrm{List}[(k_3, v_3)]$$

All values with the same key are sent to the same reducer

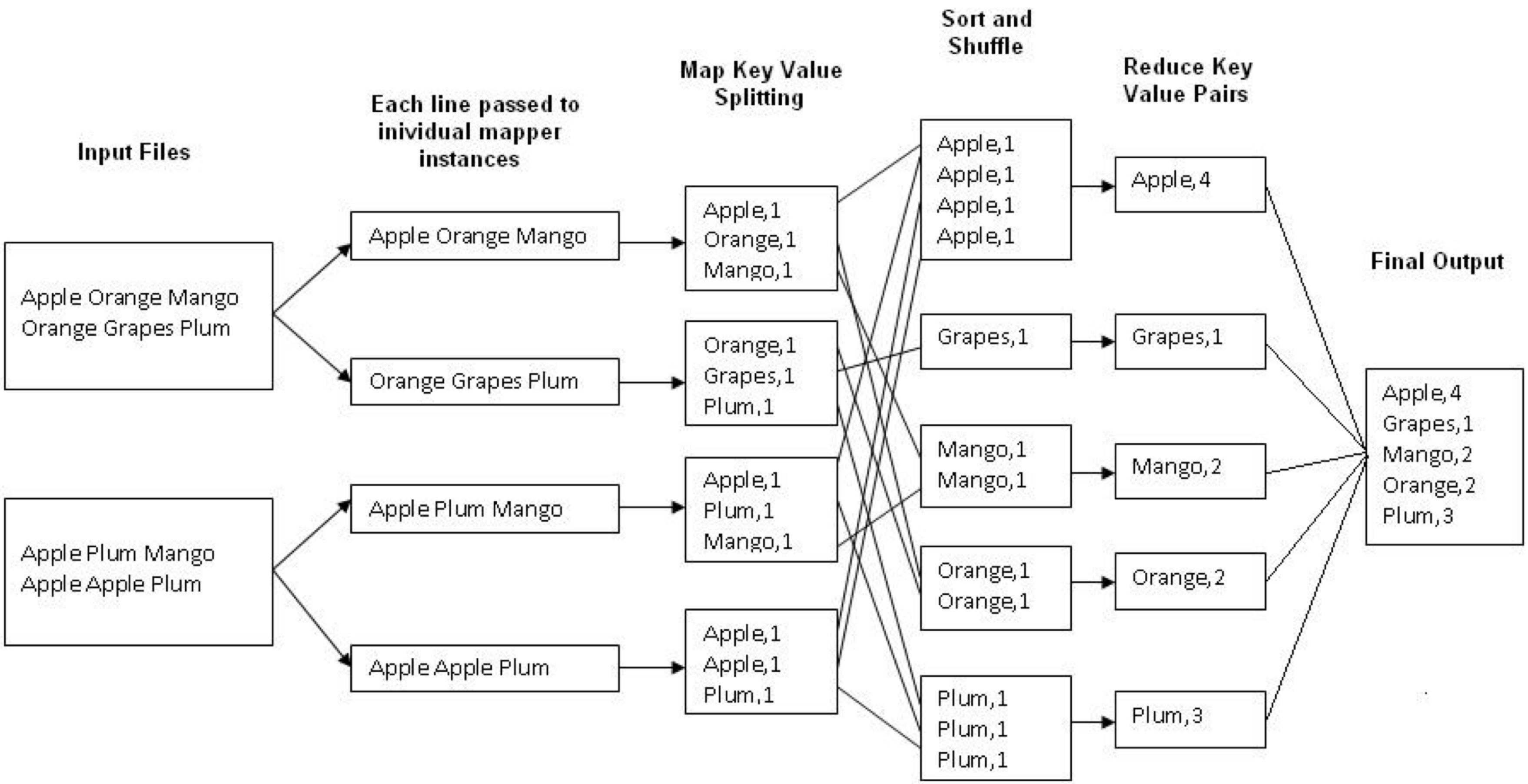
The execution framework handles everything else…


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
group values by key
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

"Hello World" MapReduce: Word Count

```
def map(key: Long, value: String) = {
  for (word <- tokenize(value)) {
    emit(word, 1)
  }
}


def reduce(key: String, values: Iterable[Int]) = {
  for (value <- values) {
    sum += value
  }
  emit(key, sum)
}
```

"Hello World" MapReduce: Word Count



MapReduce

Programmer specifies two functions:

$\texttt{map}\ (k_1, v_1) \rightarrow \texttt{List}[(k_2, v_2)]$

$\texttt{reduce}\ (k_2, \texttt{List}[v_2]) \rightarrow \texttt{List}[(k_3, v_3)]$

All values with the same key are sent to the same reducer

The execution framework handles everything else…

What's "everything else"?

MapReduce “Runtime”

Handles scheduling

Assigns workers to map and reduce tasks

Handles “data distribution”

Moves processes to data

Handles synchronization

Groups intermediate data

Handles errors and faults

Detects worker failures and restarts

Everything happens on top of a distributed FS (HDFS)

MapReduce Implementations

Google has a proprietary implementation in C++

Bindings in Java, Python

Hadoop provides an open-source implementation in Java

Development begun by Yahoo, later an Apache project

Used in production at Facebook, Twitter, LinkedIn, Netflix, …

Large and expanding software ecosystem

Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations

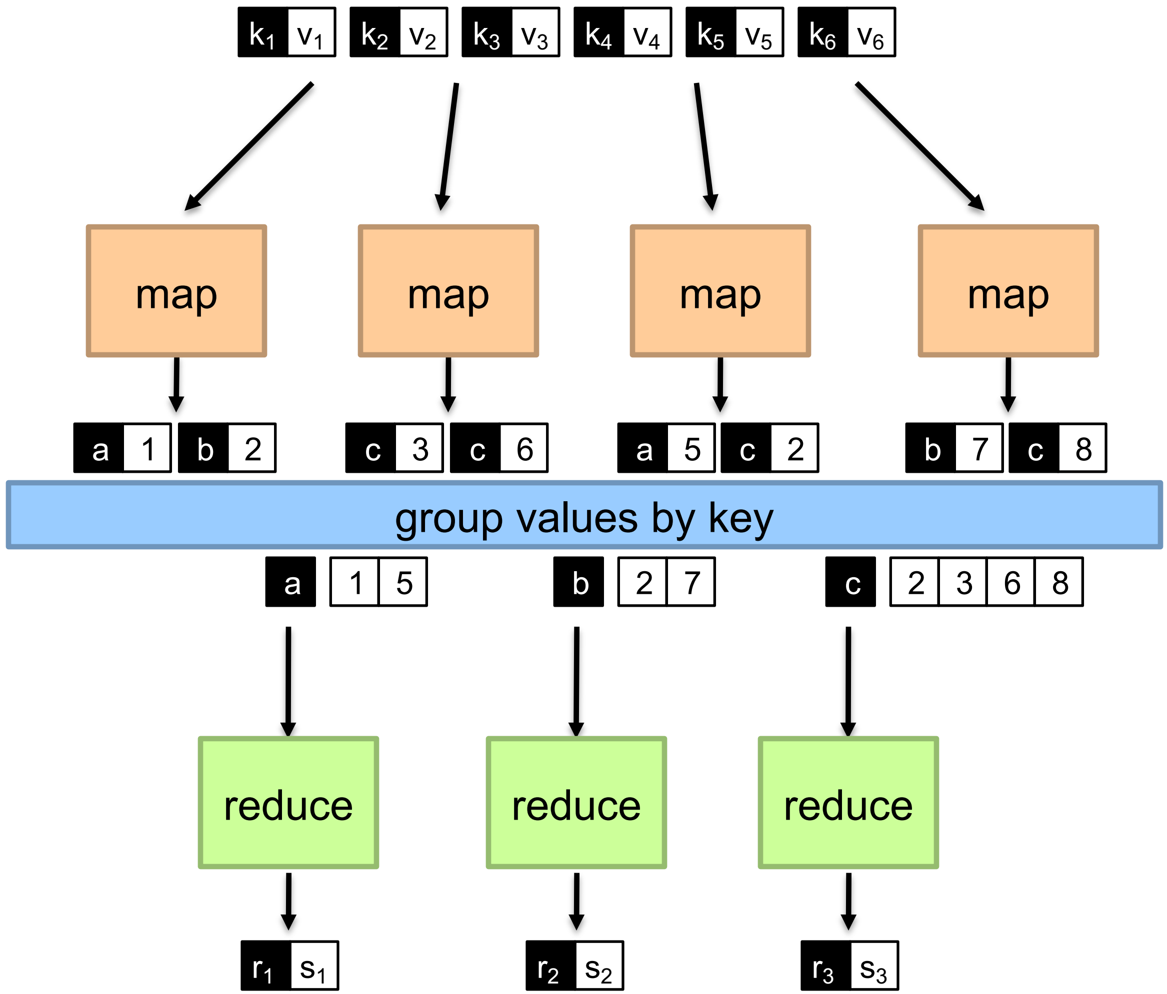
Limitations of MapReduce

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

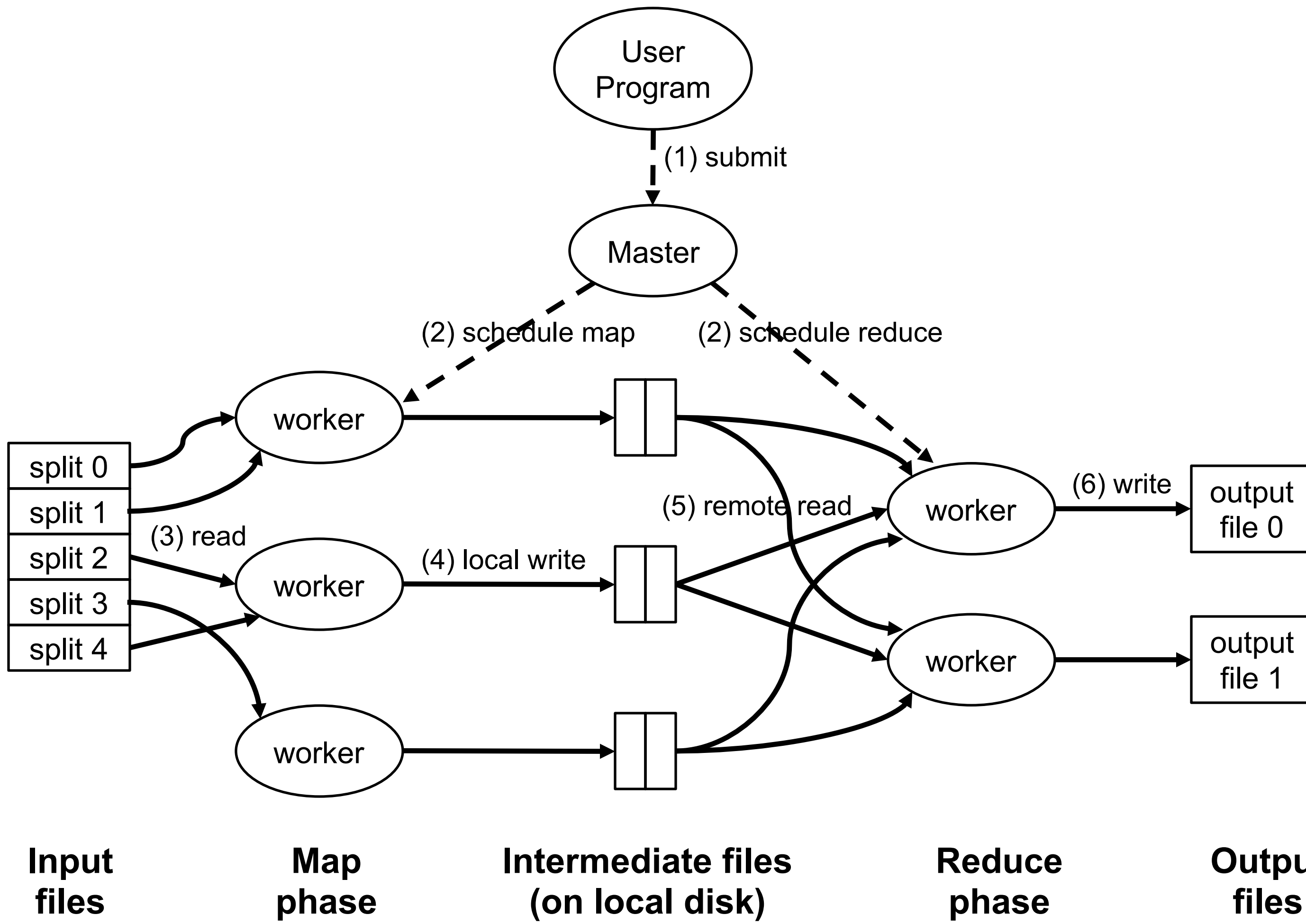
No efficient primitives for data sharing

- » State between steps goes to distributed file system
- » Slows down pipeline: replication & disk storage

Logical View

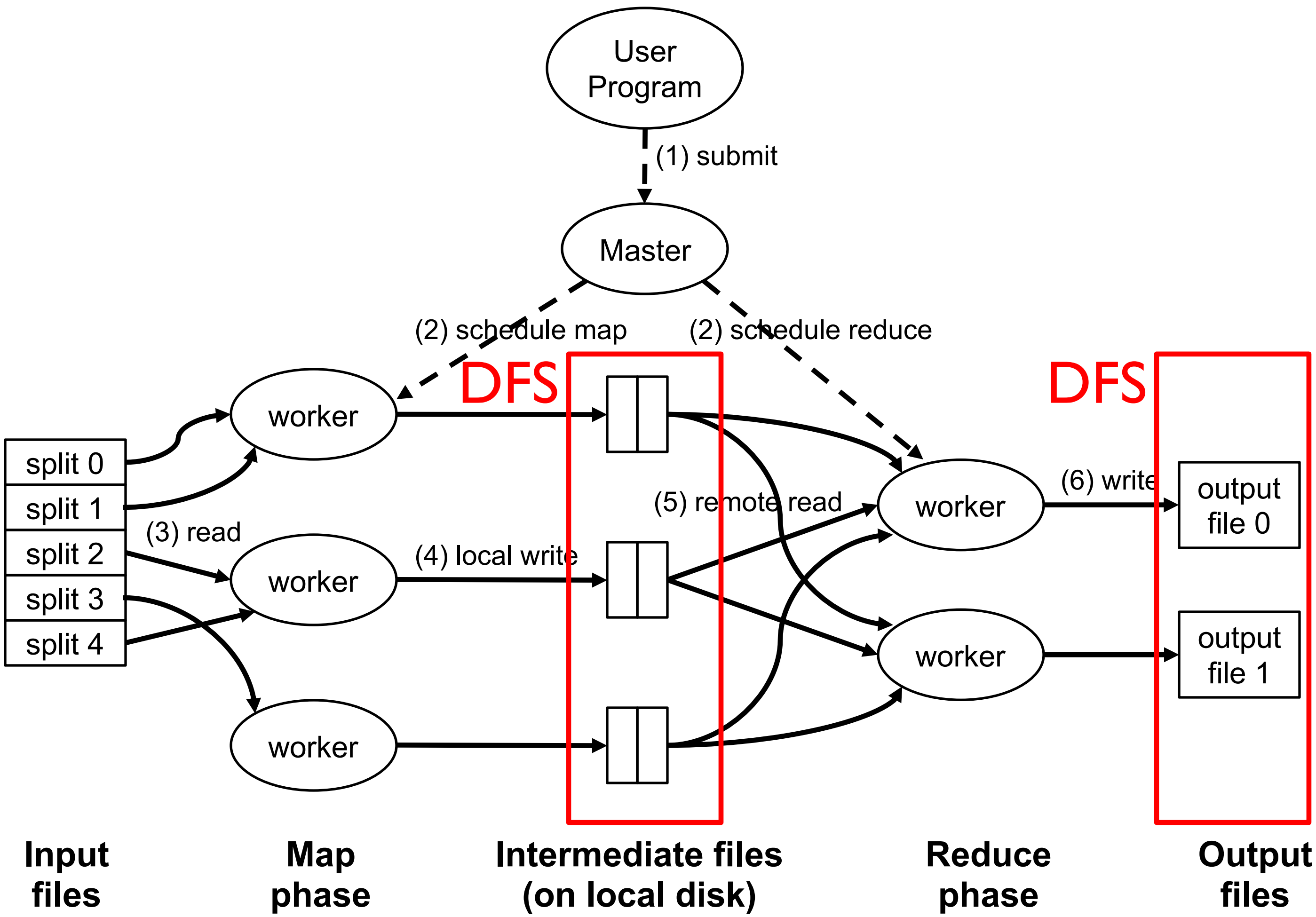


Physical View



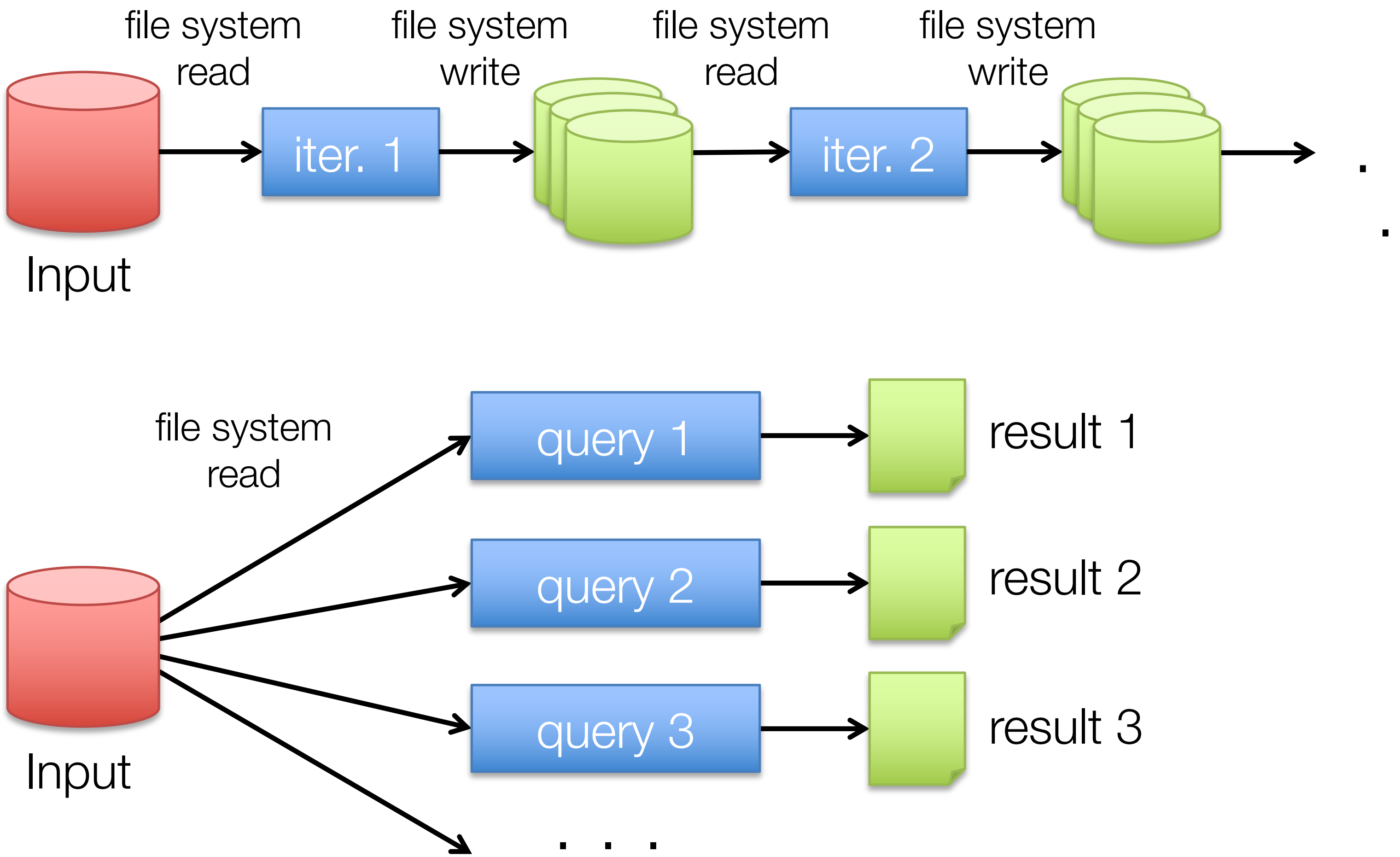
Adapted from (Dean and Ghemawat, OSDI 2004)

Physical View



Adapted from (Dean and Ghemawat, OSDI 2004)

Example: Iterative Apps

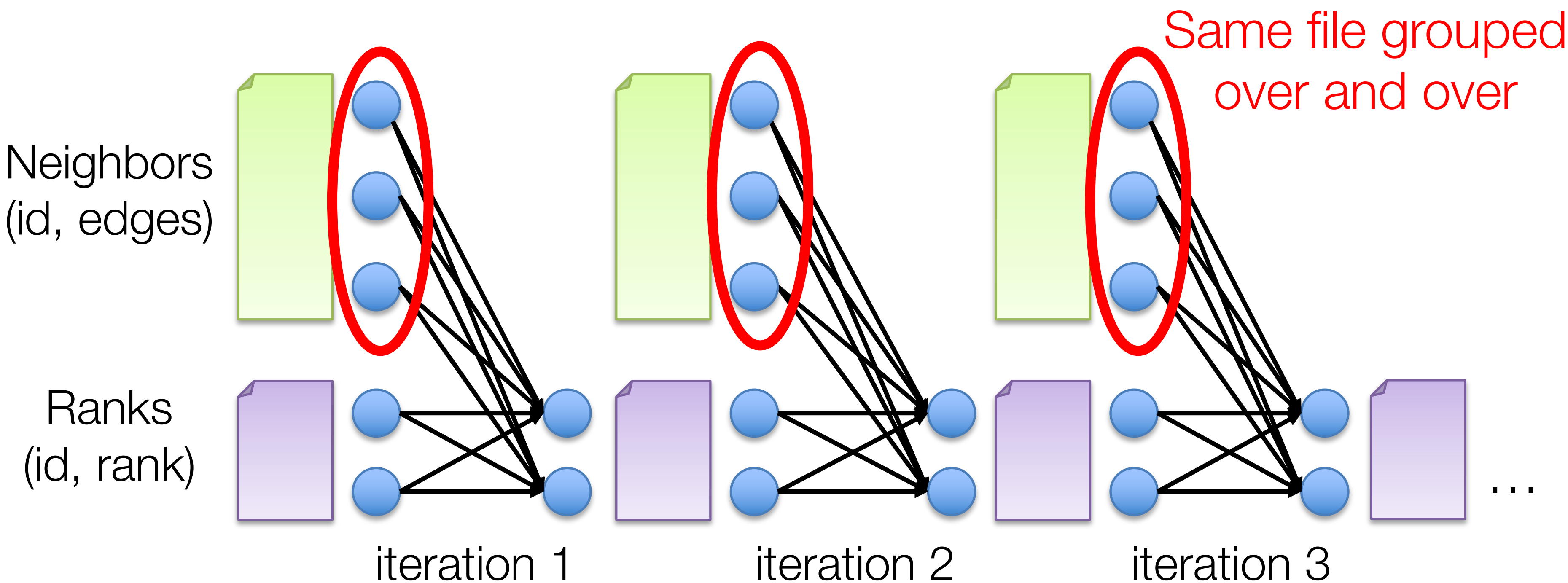


Commonly spend 90% of time doing I/O

Example: PageRank

Repeatedly multiply sparse matrix and vector

Requires repeatedly hashing together page adjacency lists and rank vector



MapReduce -> Spark

While MapReduce is simple, composing multiple M/R stages has a huge I/O cost: network + disk

Spark compute engine:

- Extends a **PL** with data-flow operators and in-memory distributed collection data-structure
  - » "Resilient distributed datasets" (RDD)

Spark

Answer to “What’s beyond MapReduce?”

Brief history:

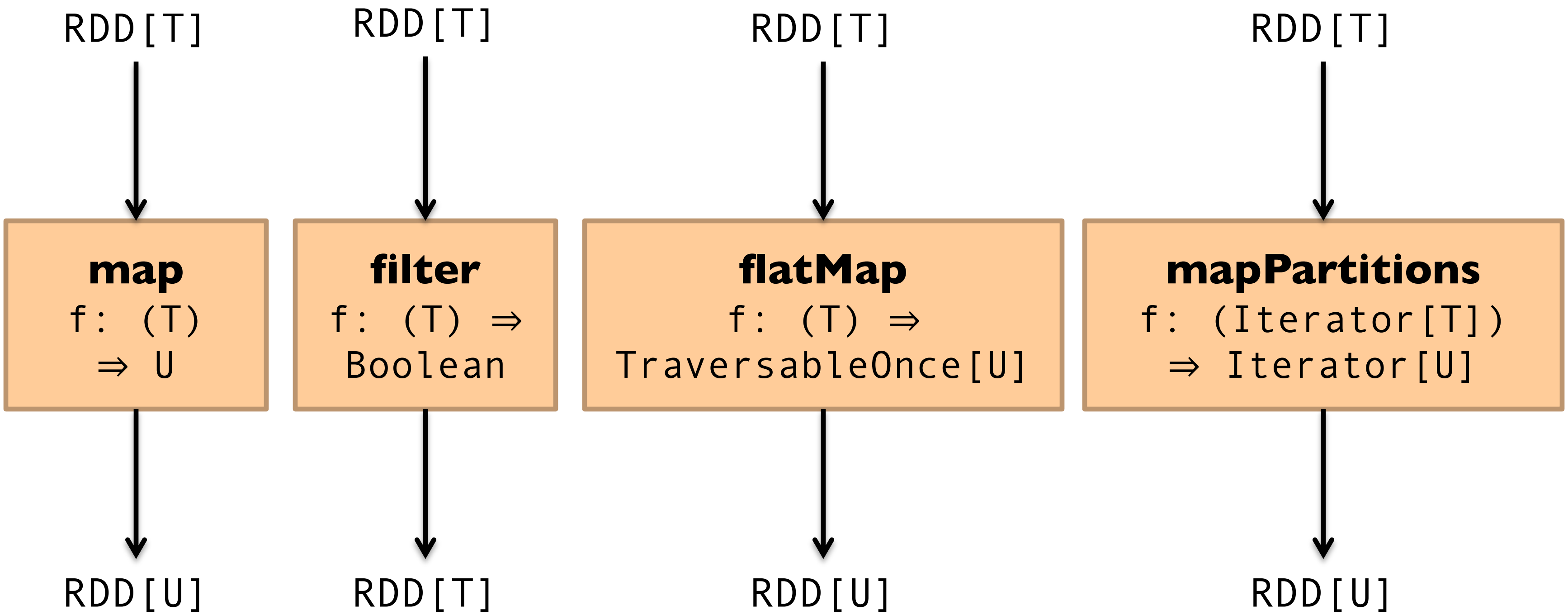
Developed at UC Berkeley AMPLab in 2009

Open-sourced in 2010

Became top-level Apache project in February 2014

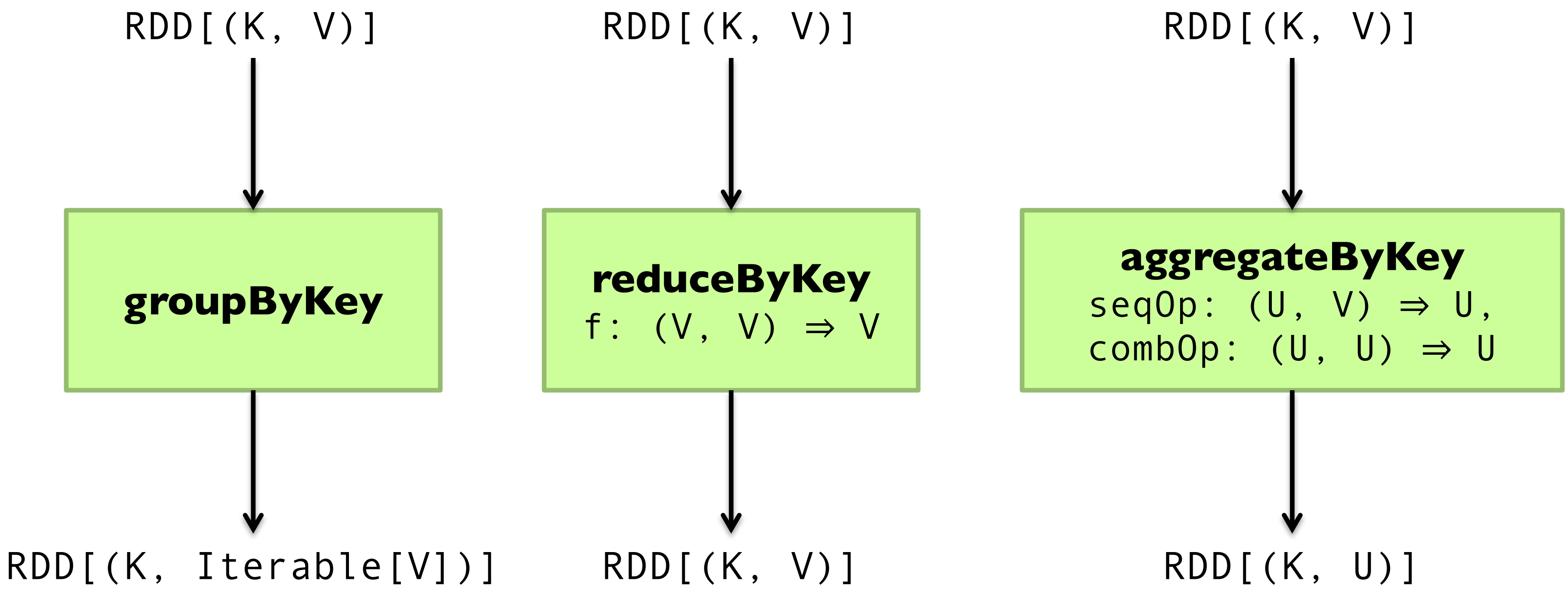
Commercial support provided by DataBricks

Map-like Operations



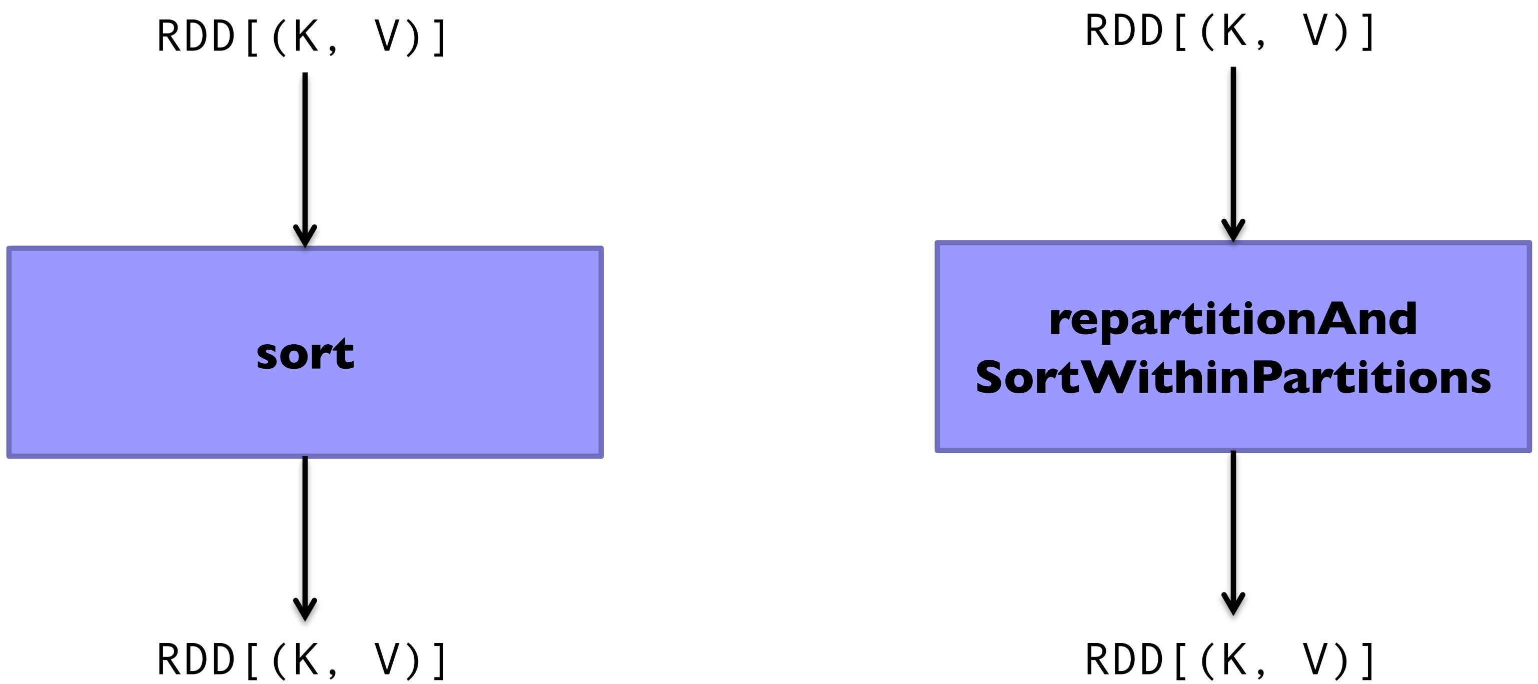
(Not meant to be exhaustive)

Reduce-like Operations



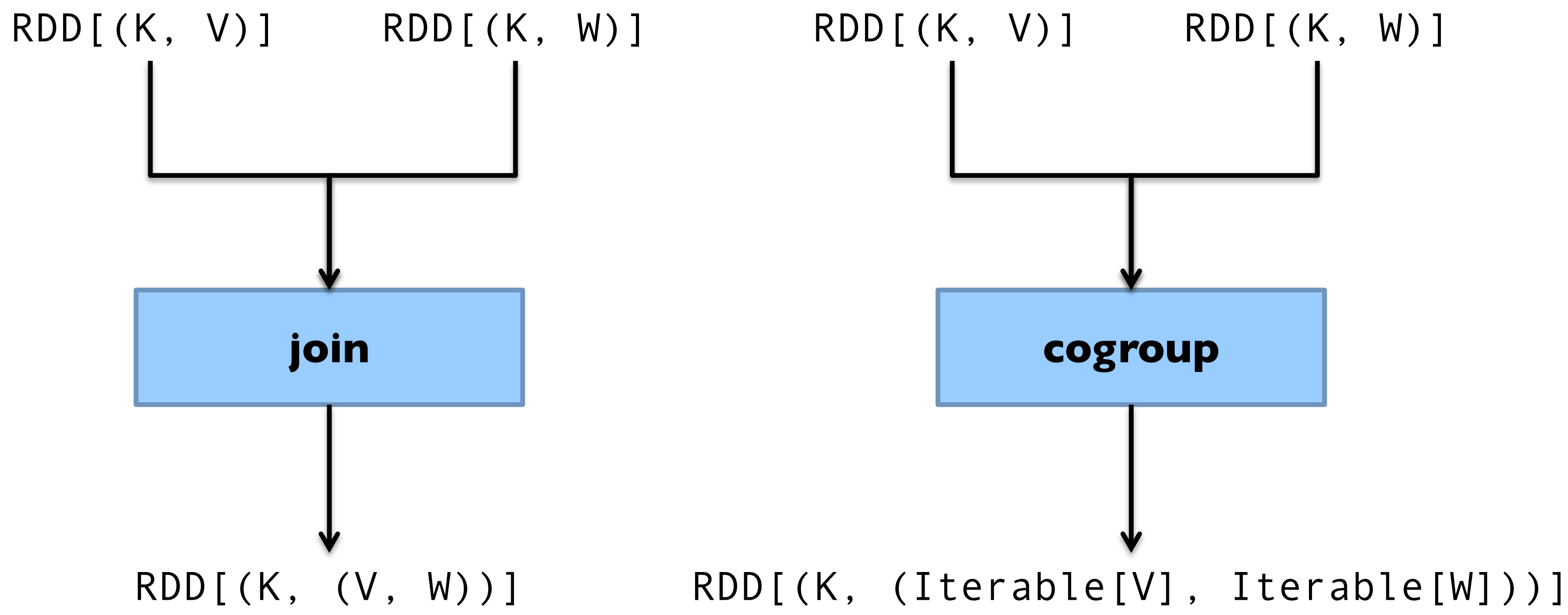
(Not meant to be exhaustive)

Sort Operations



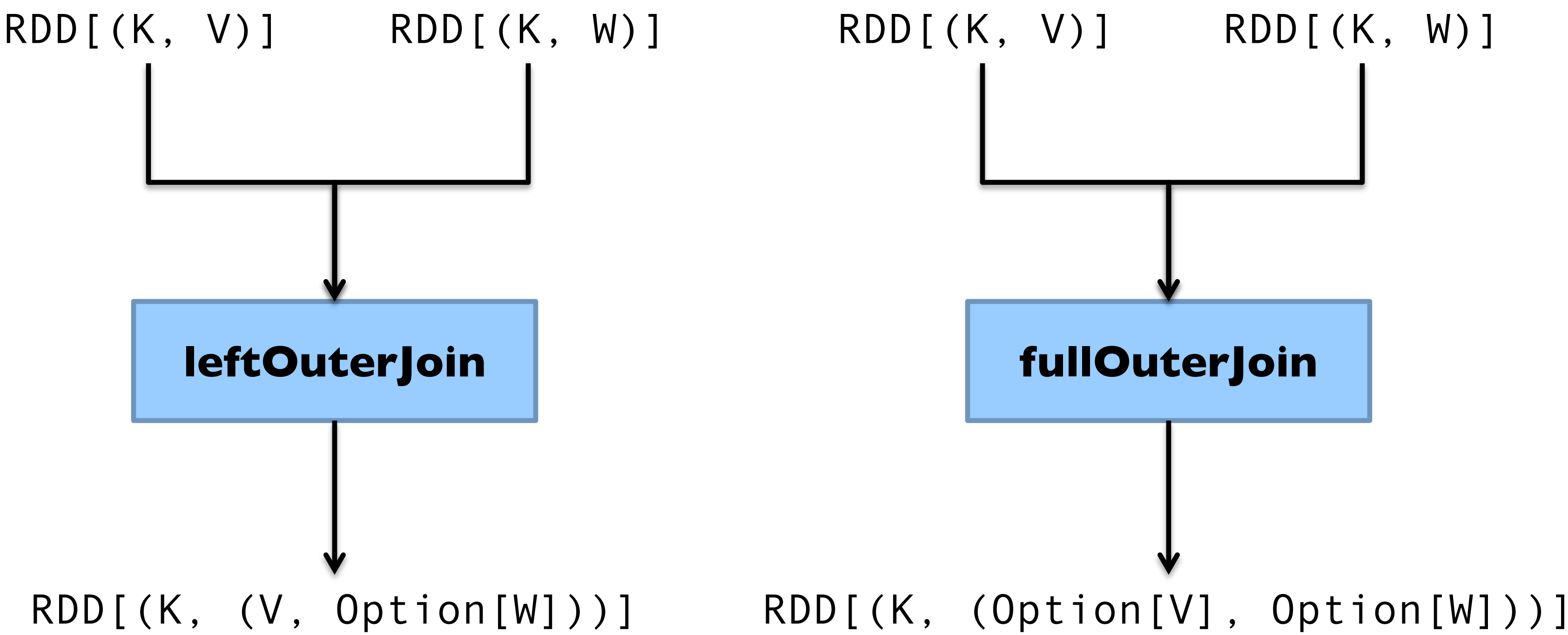
(Not meant to be exhaustive)

Join-like Operations



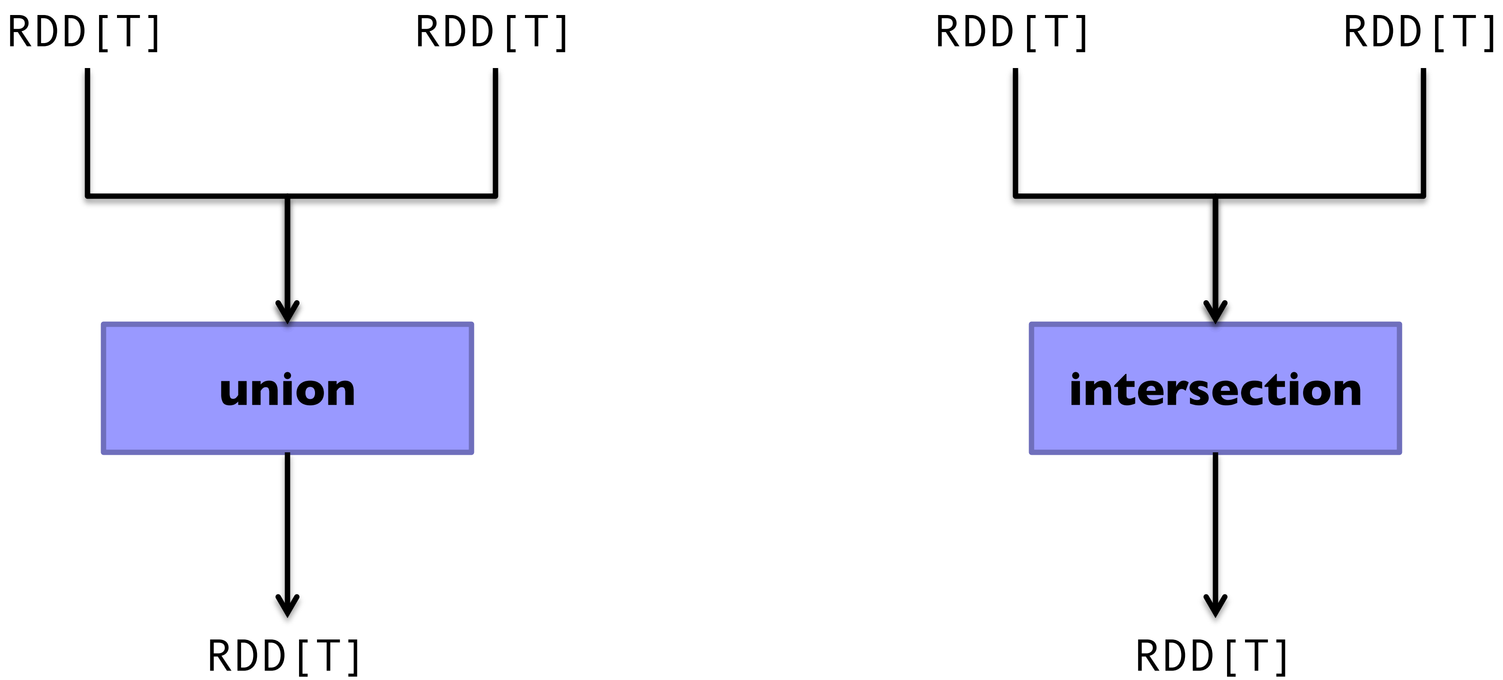
(Not meant to be exhaustive)

Join-like Operations



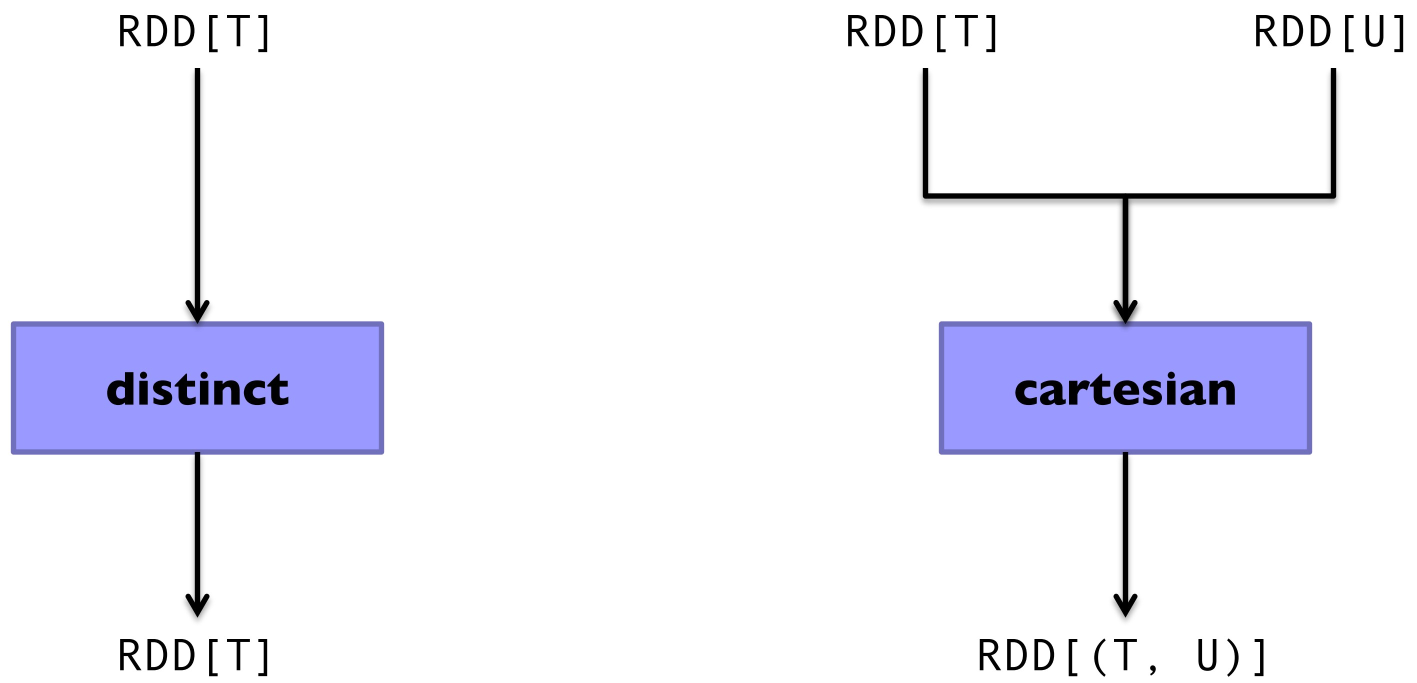
(Not meant to be exhaustive)

Set-ish Operations



(Not meant to be exhaustive)

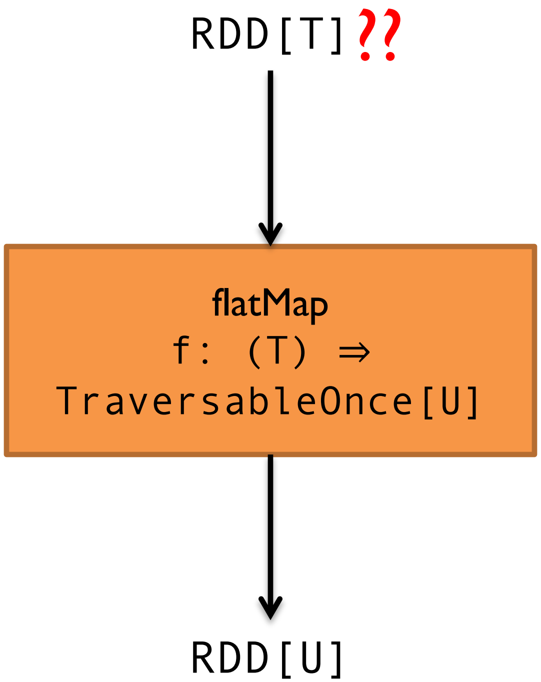
Set-ish Operations



(Not meant to be exhaustive)

Spark Word Count

```
val textFile = sc.textFile(args.input())

textFile
  .flatMap(line => tokenize(line))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
  .saveAsTextFile(args.output())
```



What's an RDD?

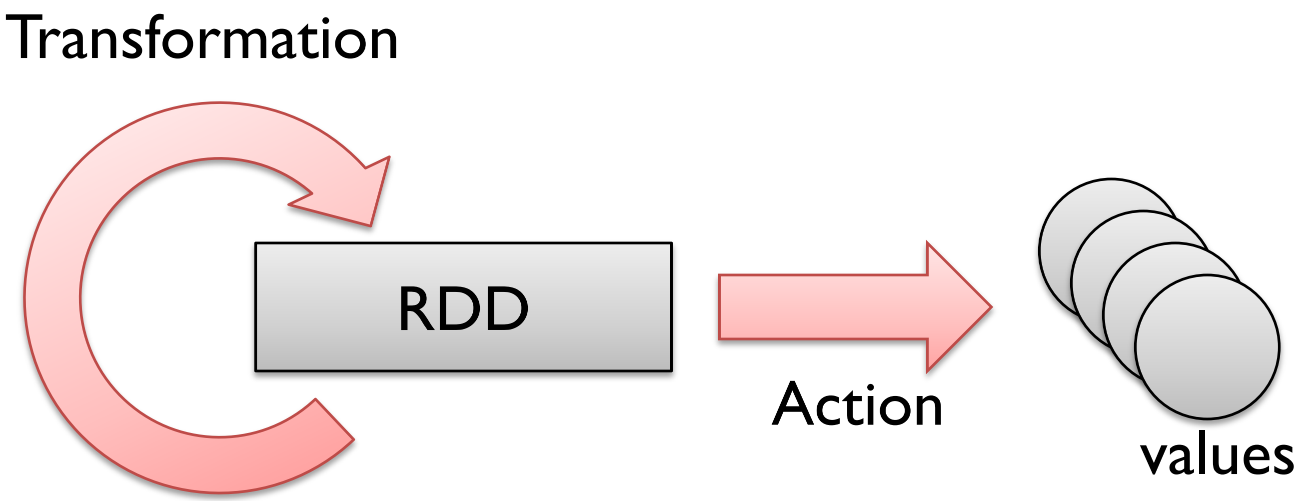
Resilient Distributed Dataset (RDD)

= immutable   = partitioned

» Immutable collections of objects, spread across cluster
» Statically typed: RDD[T] has objects of type T

Wait, so how do you actually do anything?
Developers define *transformations* on RDDs
Framework keeps track of lineage

RDD Lifecycle



Transformations are lazy:
Framework keeps track of lineage

Actions trigger actual execution

Spark Word Count

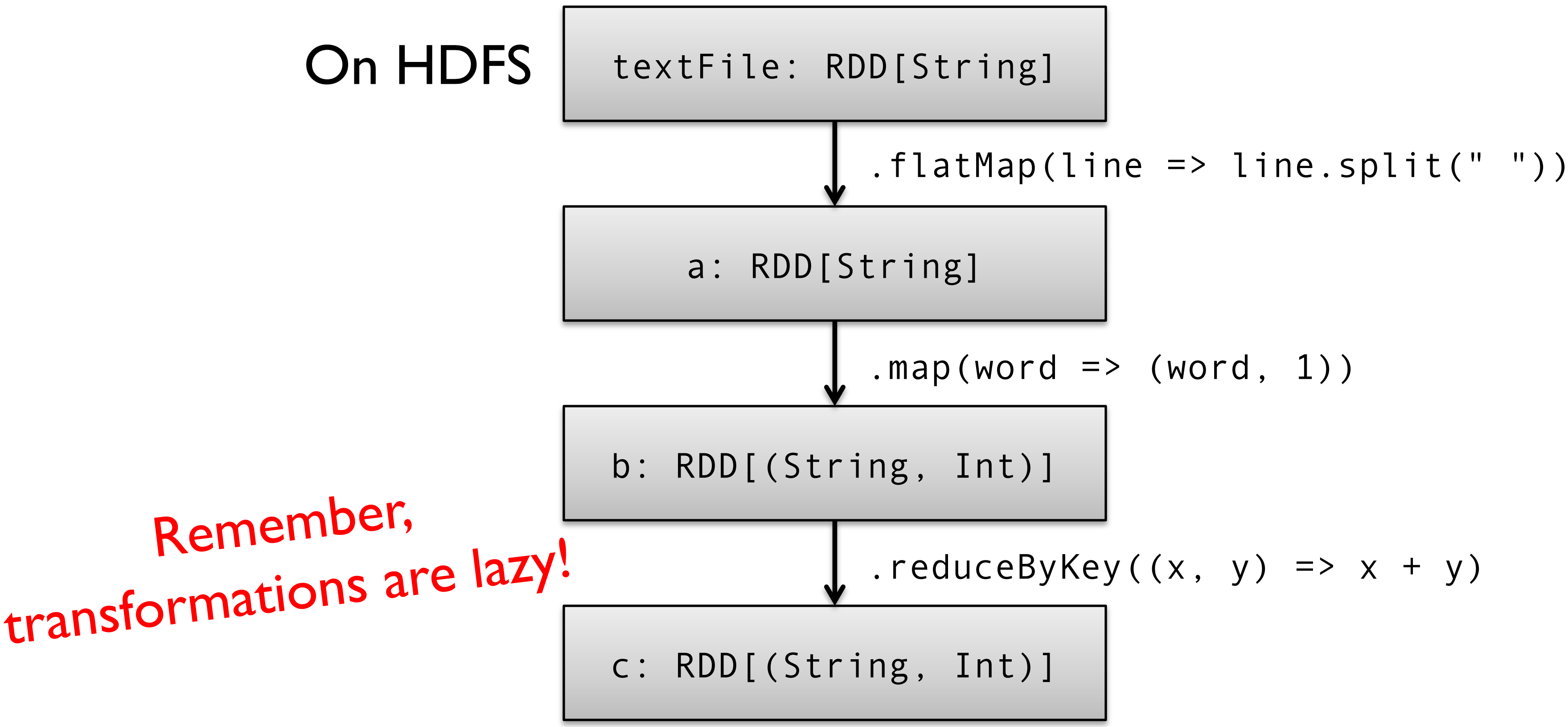
RDDs

```
val textFile = sc.textFile(args.input())

val a = textFile.flatMap(line => line.split(" "))
val b = a.map(word => (word, 1))
val c = b.reduceByKey((x, y) => x + y)

c.saveAsTextFile(args.output())
```

Action

Transformations

RDDs and Lineage



RDDs and Lineage

On HDFS

```
textFile: RDD[String]
    .flatMap(line => line.split(" "))
a: RDD[String]
    .map(word => (word, 1))
b: RDD[(String, Int)]
    .reduceByKey((x, y) => x + y)
c: RDD[(String, Int)]
    .saveAsTextFile(args.output())
```

Action!

Remember,
transformations are lazy!

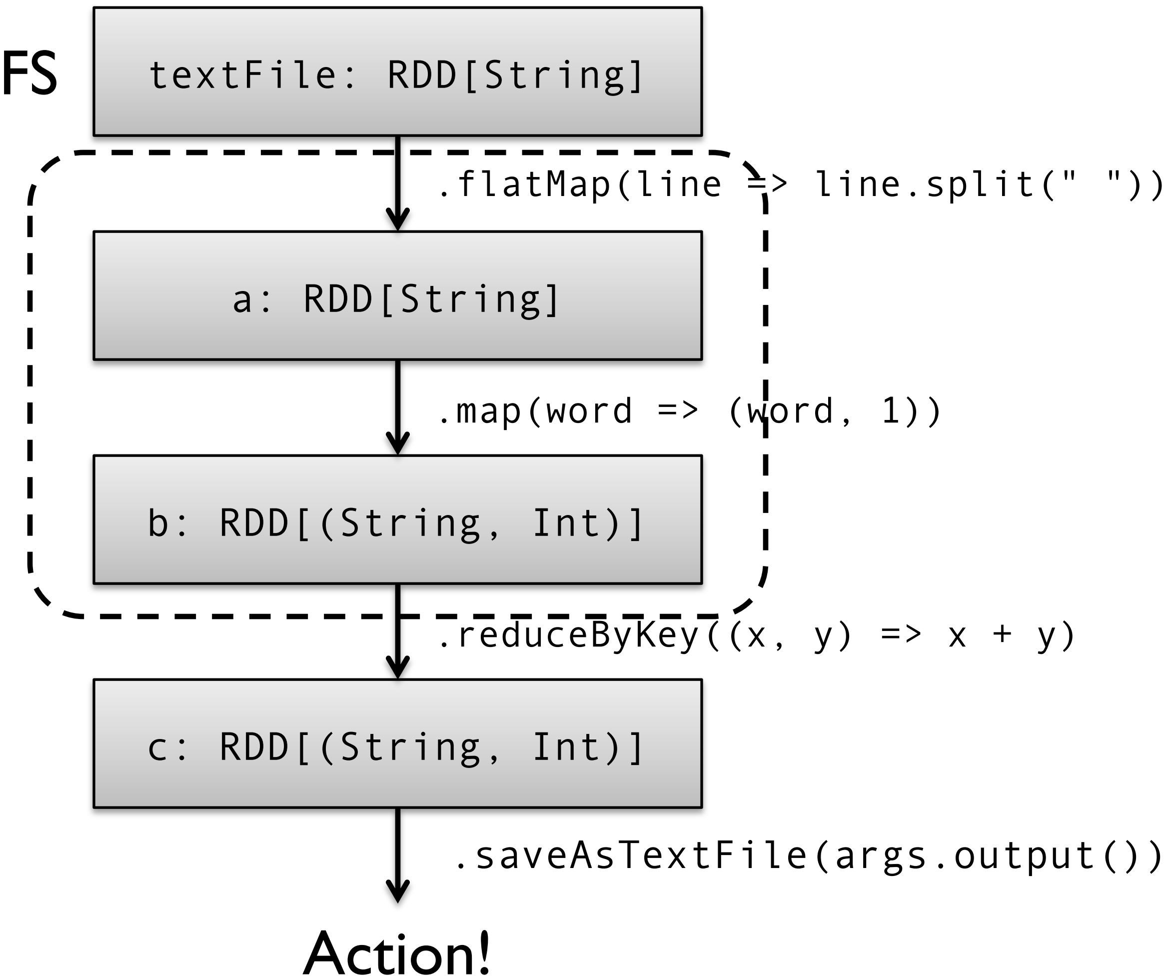
RDDs and Optimizations

Lazy evaluation creates optimization opportunities

On HDFS

RDDs don't need to be materialized!

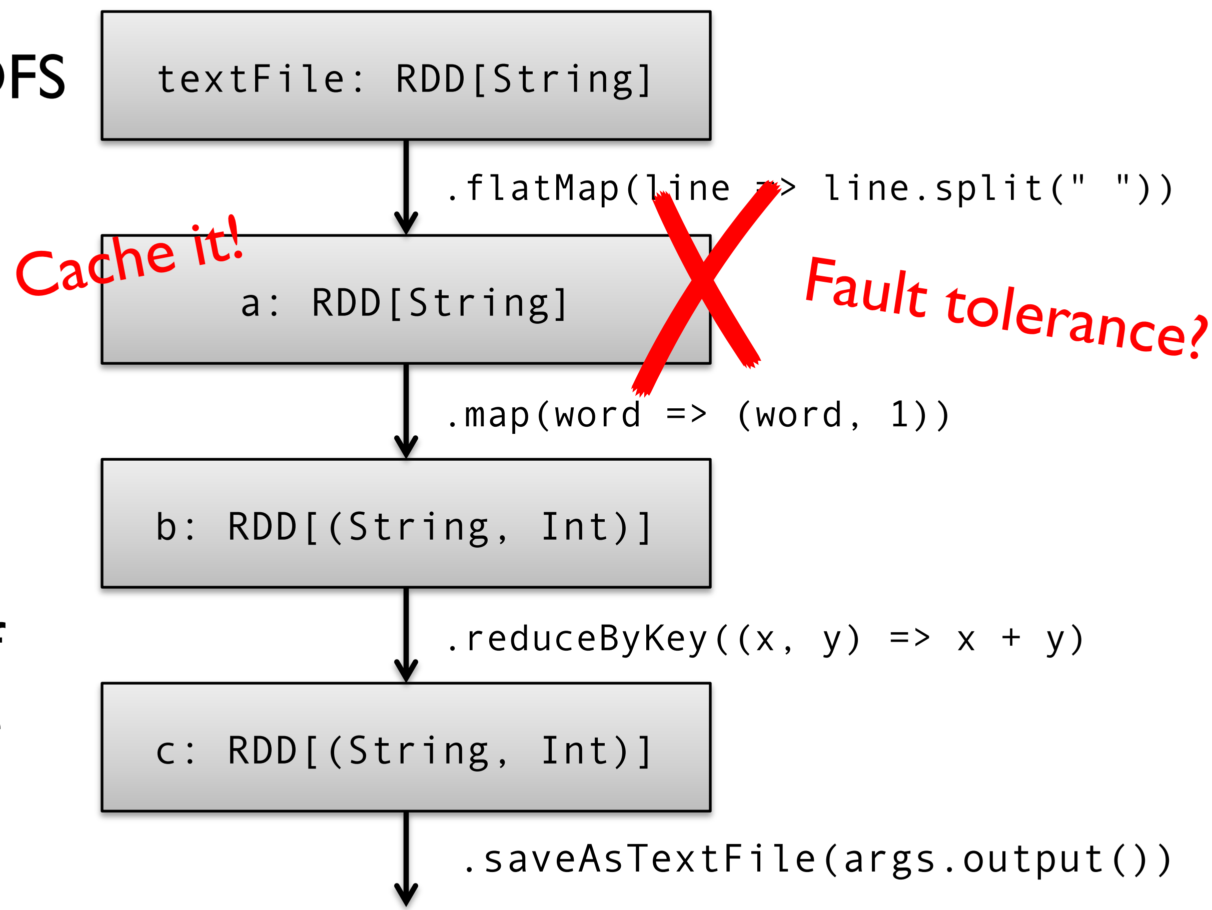
Optimize Map->Map



RDDs and Caching

RDDs can be materialized in memory (and on disk)!

On HDFS



Spark works even if the RDDs are *partially* cached!

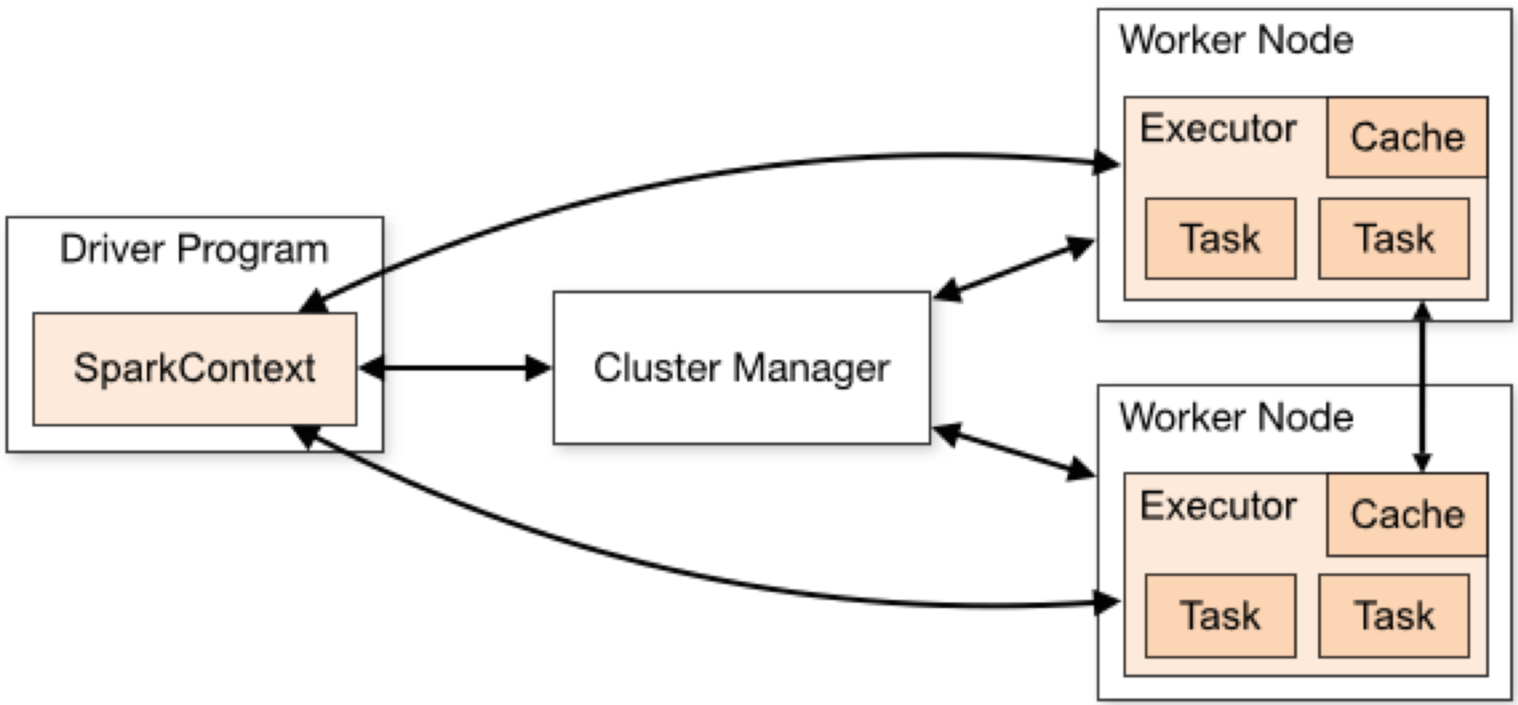
Action!

Resilient Distributed Datasets (RDDs)

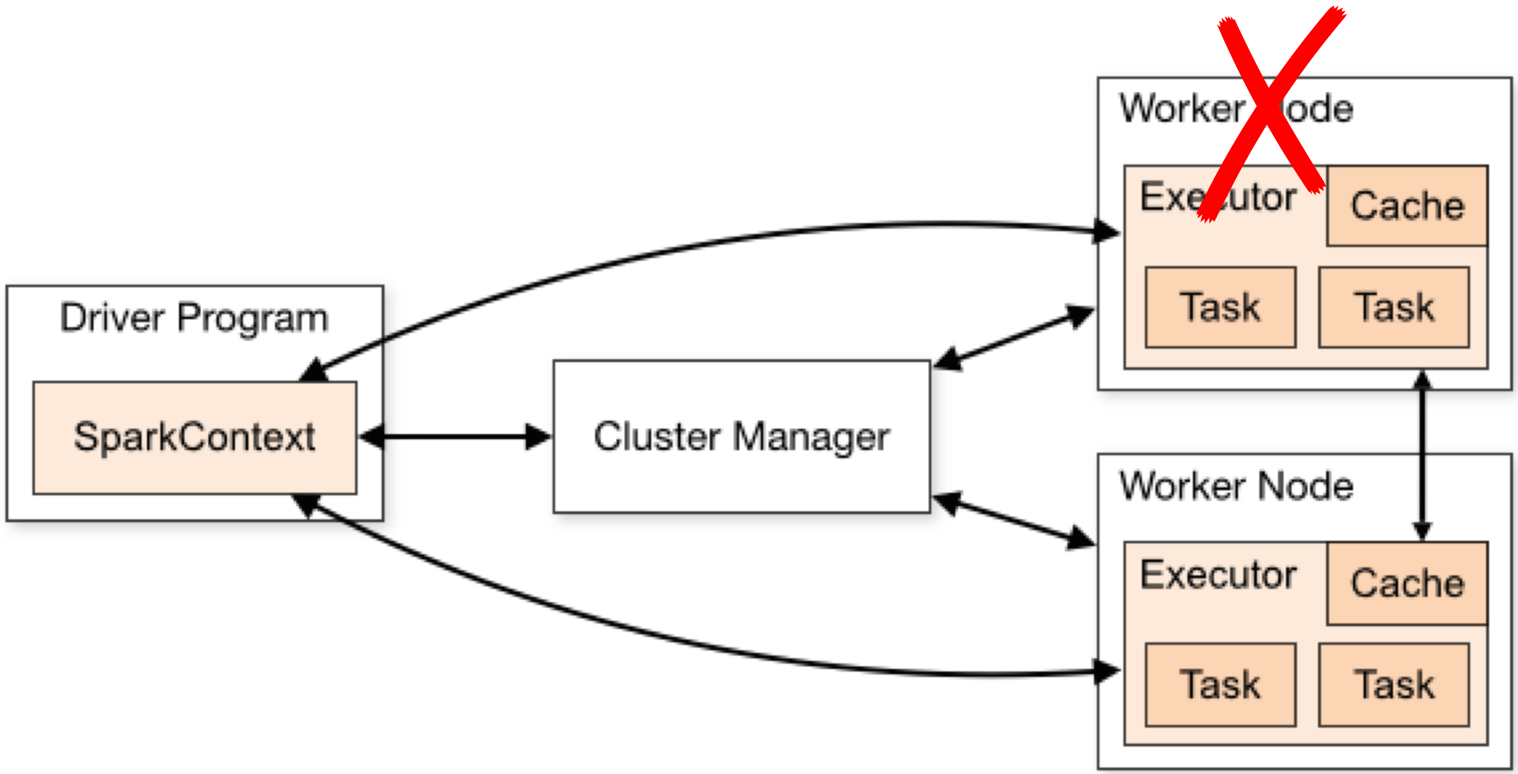
- Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, ...)
- Built via parallel transformations (map, filter, …)
- Only lets you make RDDs such that they can be:

Automatically rebuilt on **failure**

Spark Architecture



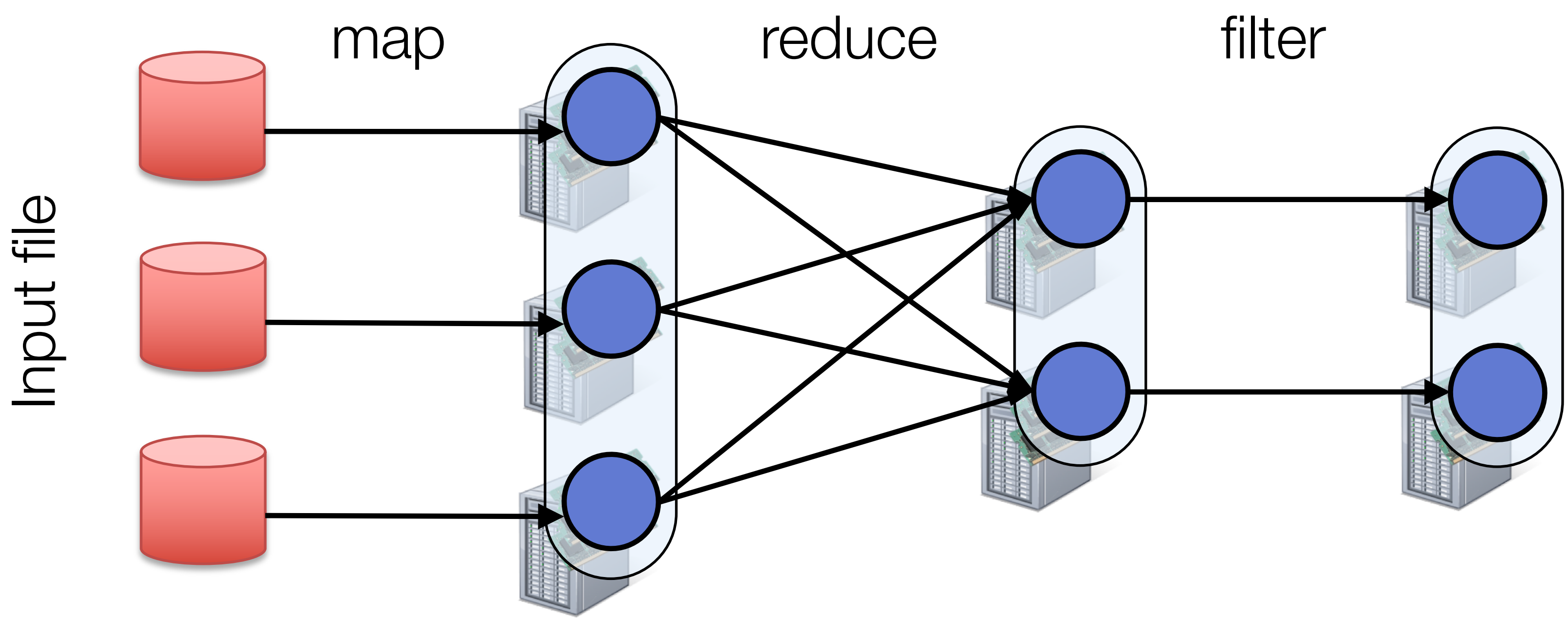
Spark Architecture



Fault Tolerance

RDDs track *lineage* info to rebuild lost data

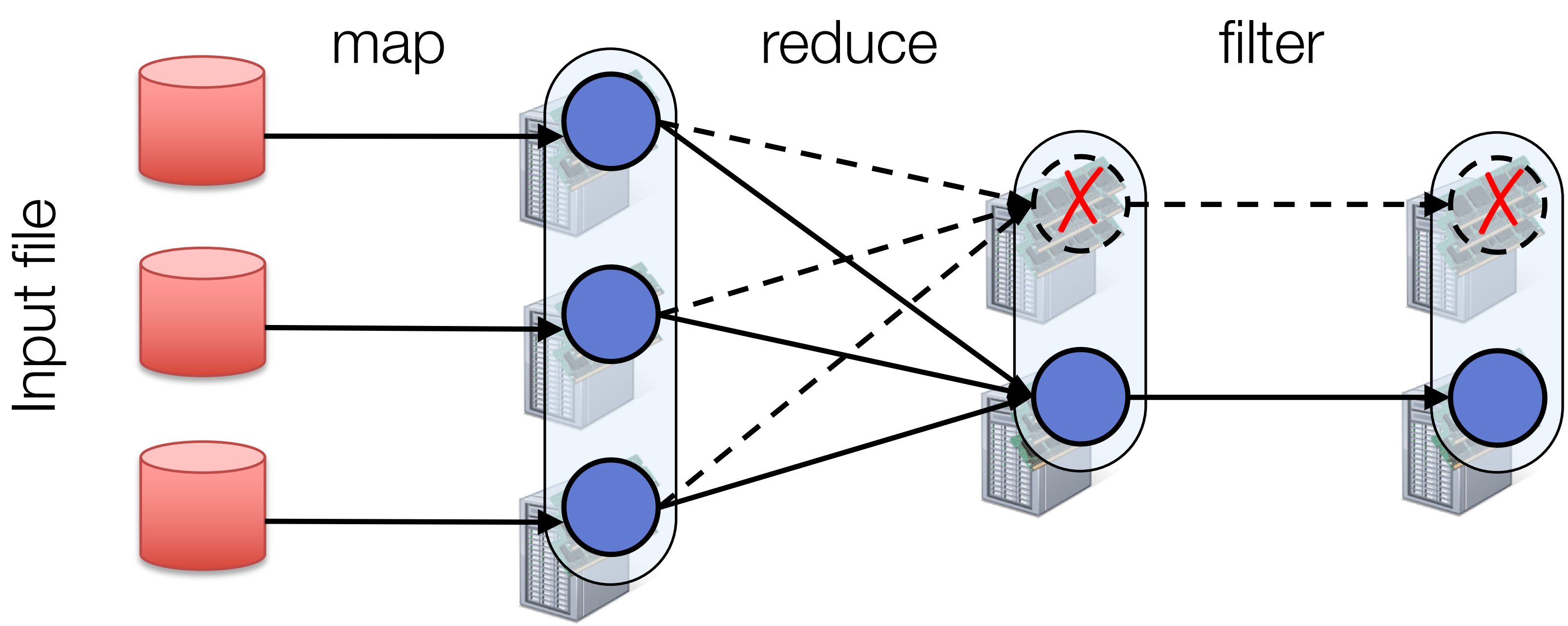
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Fault Tolerance

RDDs track *lineage* info to rebuild lost data

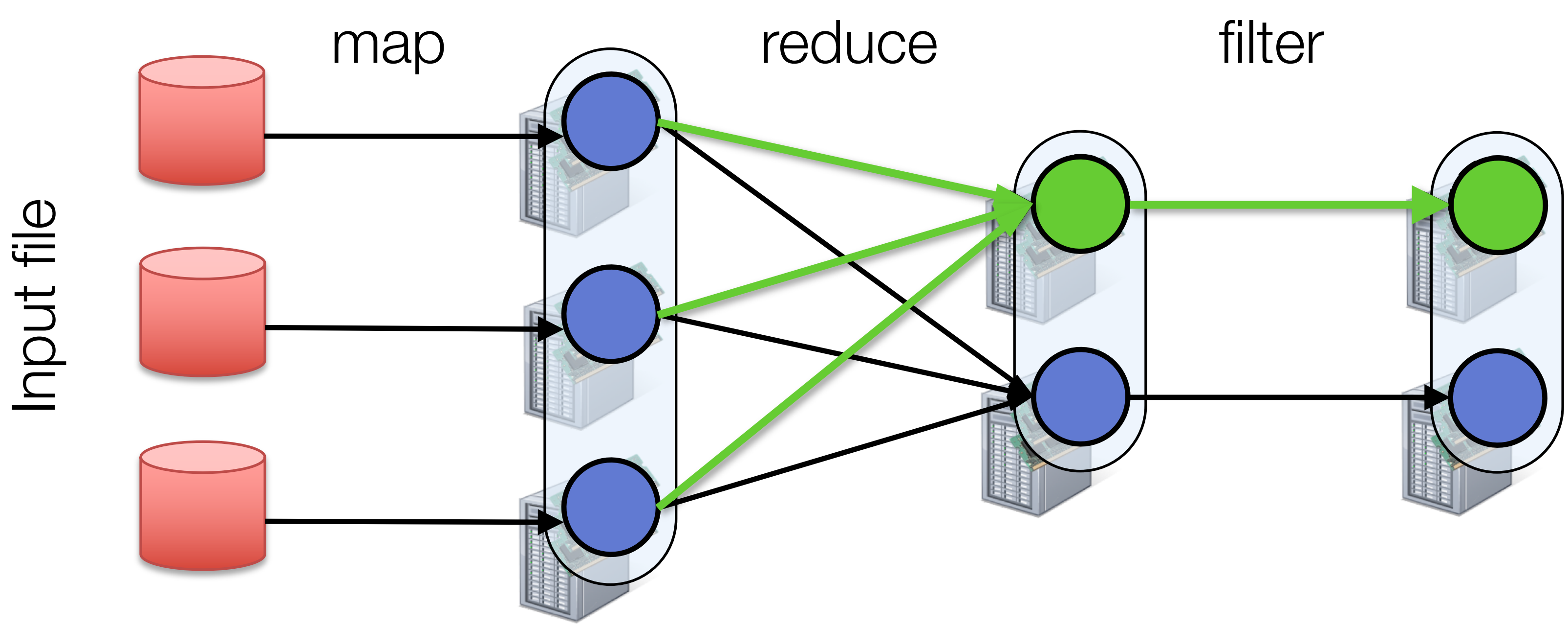
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



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)
```

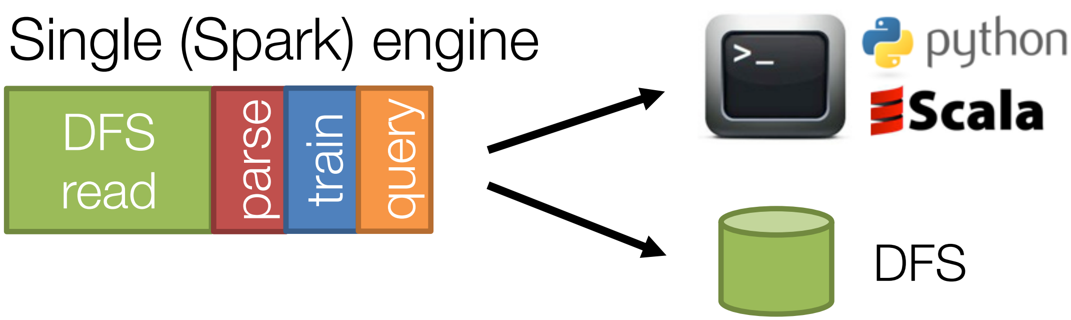


Benefit of a single ecosystem

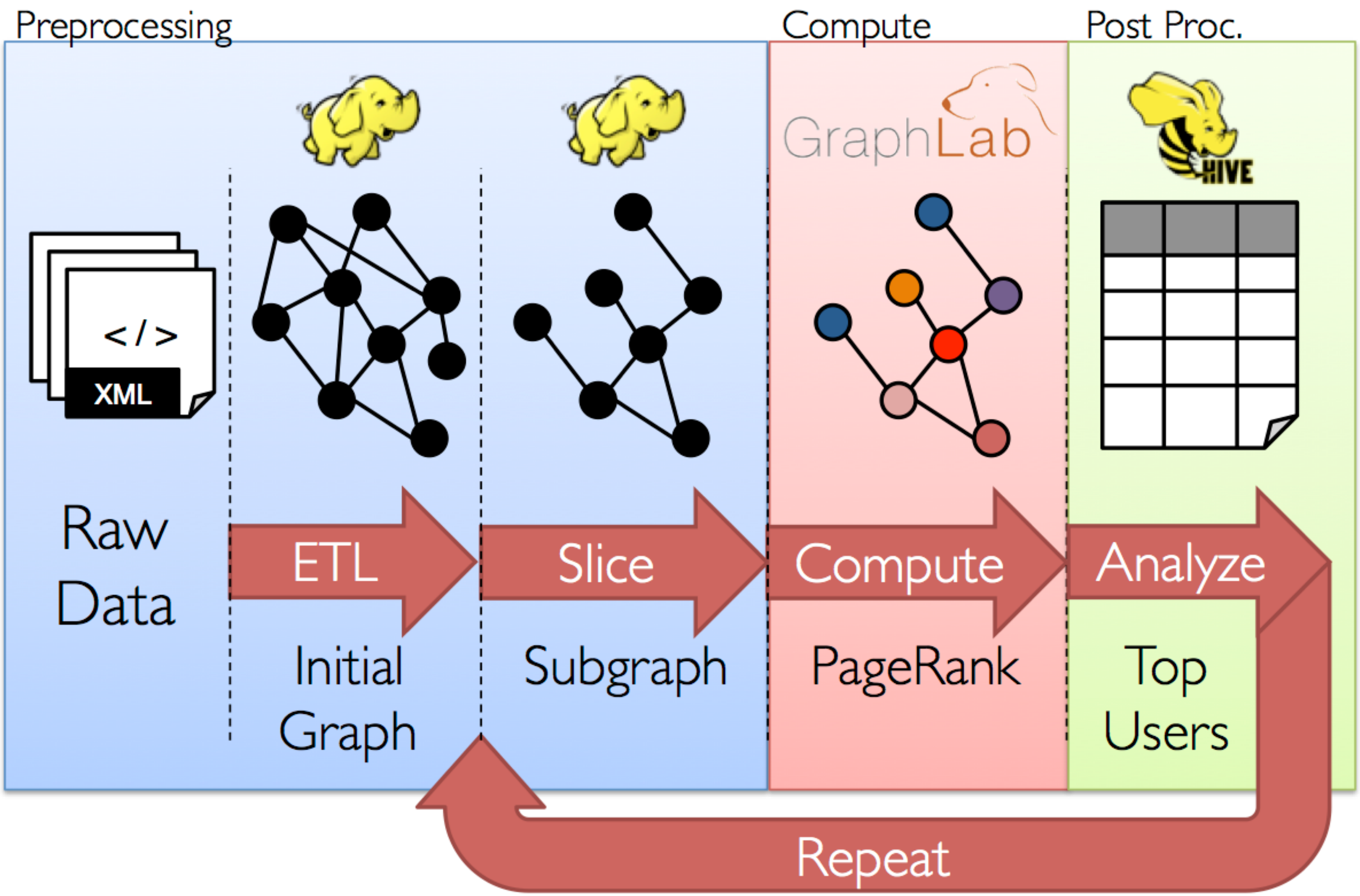
Same engine performs data extraction, model training and interactive queries

Separate engines

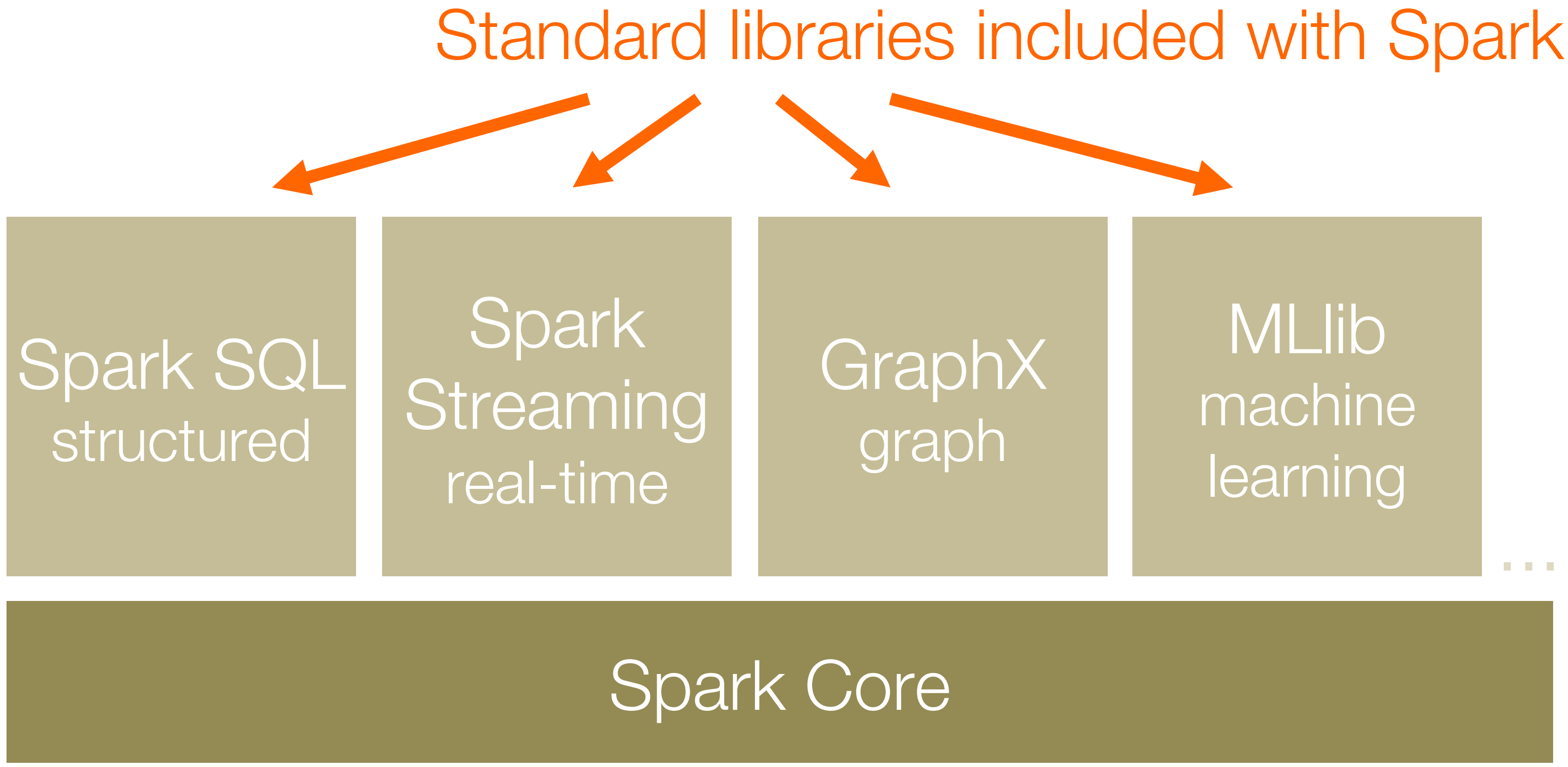
Single (Spark) engine



Example: graph processing



Spark: a general platform



Spark.ML Library (MLlib)

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

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

Spark.GraphX

General graph processing library

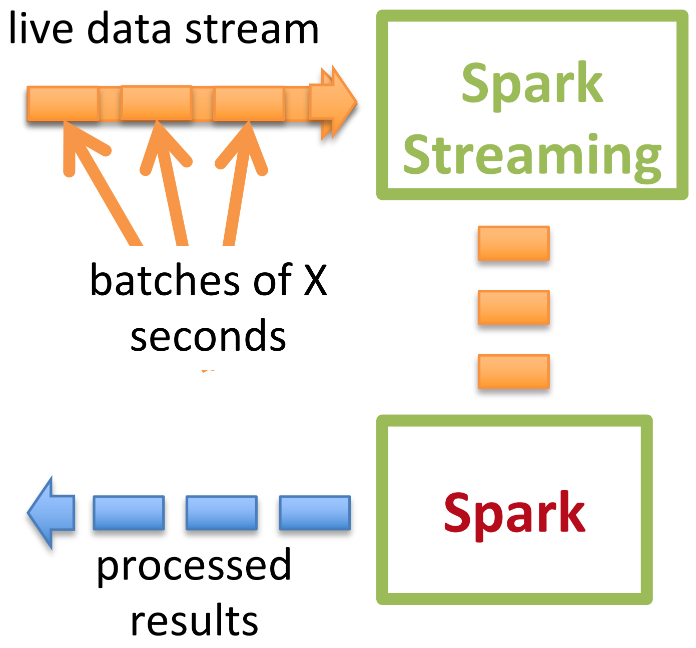
Build graph using RDDs of nodes and edges

Large library of graph algorithms with composable steps

Spark Streaming

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

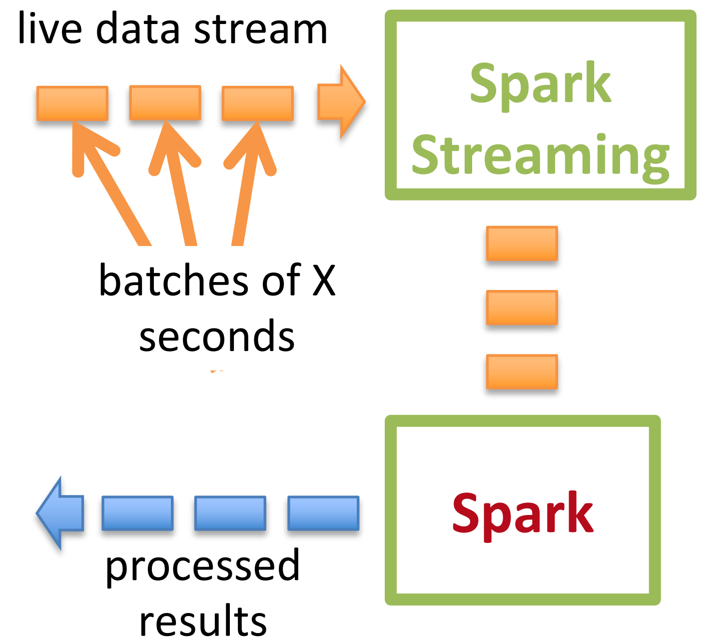
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



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()
```

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
}}
```

May other data-flow systems

Graph Computations: Pregel, GraphLab

SQL based engines: Hive, Pig, …

… data-flow an ideal abstract? Who knows.

Take-aways

Data flow engines are important for distributed processing: simplify life for devs!

MapReduce: batch processing + distinct map and reduce phases. Inefficient and low level.

Spark: RDDs for fault tolerance; ecosystem.