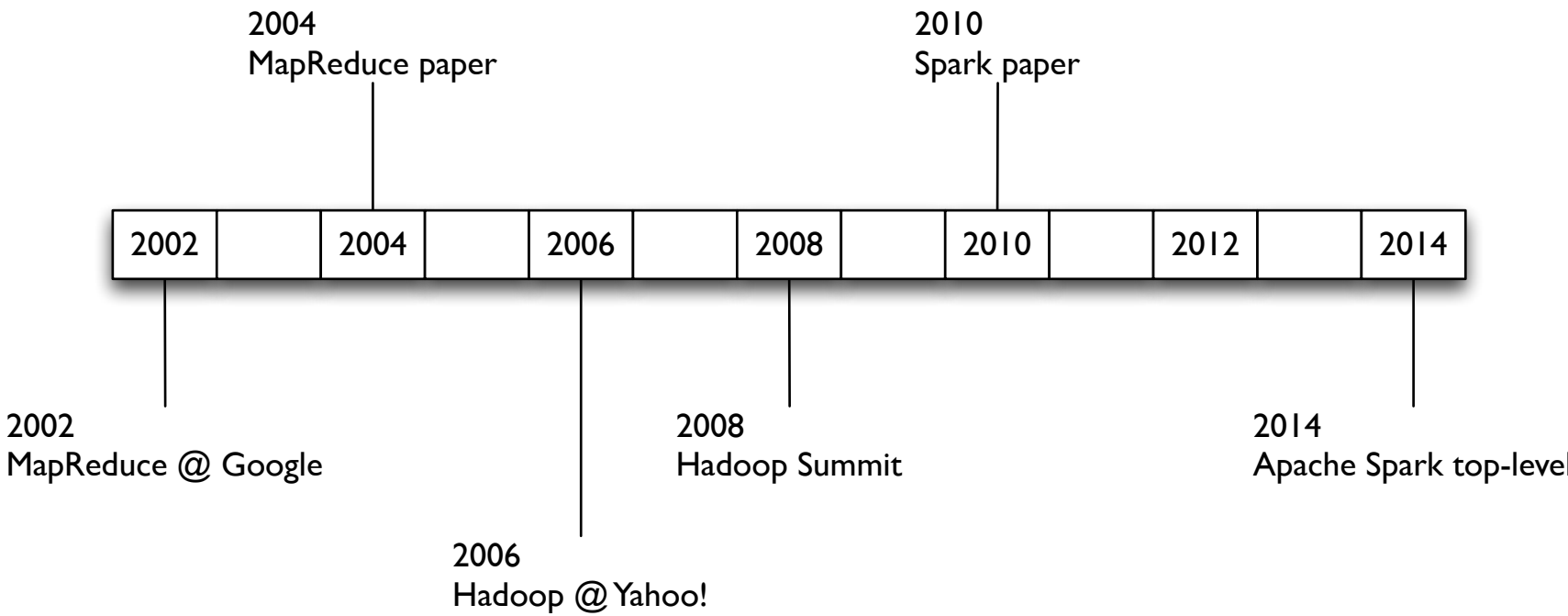
Spark: A Brief History

https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf

A Brief History:



A Brief History: *MapReduce*

circa 1979 – **Stanford**, **MIT**, **CMU**, etc.
set/list operations in LISP, Prolog, etc., for parallel processing
www-formal.stanford.edu/jmc/history/lisp/lisp.htm

circa 2004 – **Google**
MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
research.google.com/archive/mapreduce.html

circa 2006 – **Apache**
Hadoop, originating from the Nutch Project
Doug Cutting
research.yahoo.com/files/cutting.pdf

circa 2008 – **Yahoo**
web scale search indexing
Hadoop Summit, HUG, etc.
developer.yahoo.com/hadoop/

circa 2009 – **Amazon AWS**
Elastic MapReduce
Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.
aws.amazon.com/elasticmapreduce/

A Brief History: *MapReduce*

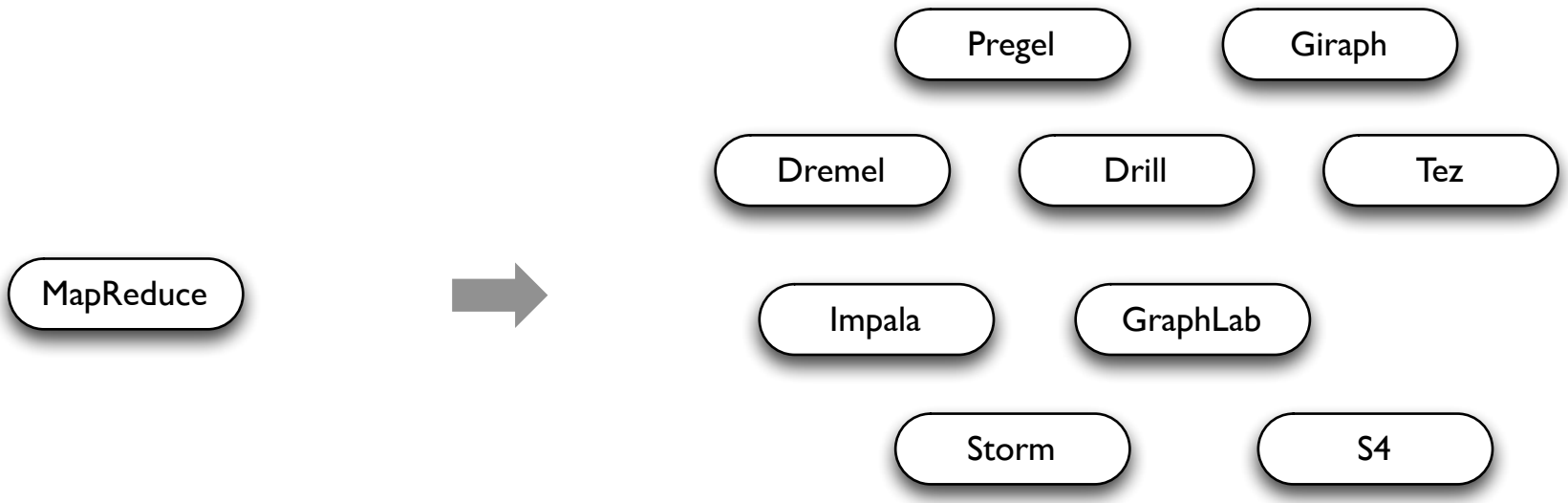
MapReduce use cases showed two major limitations:

1. difficultly of programming directly in MR
2. performance bottlenecks, or batch not fitting the use cases

In short, MR doesn't compose well for large applications

Therefore, people built *specialized systems* as workarounds…

A Brief History: *MapReduce*

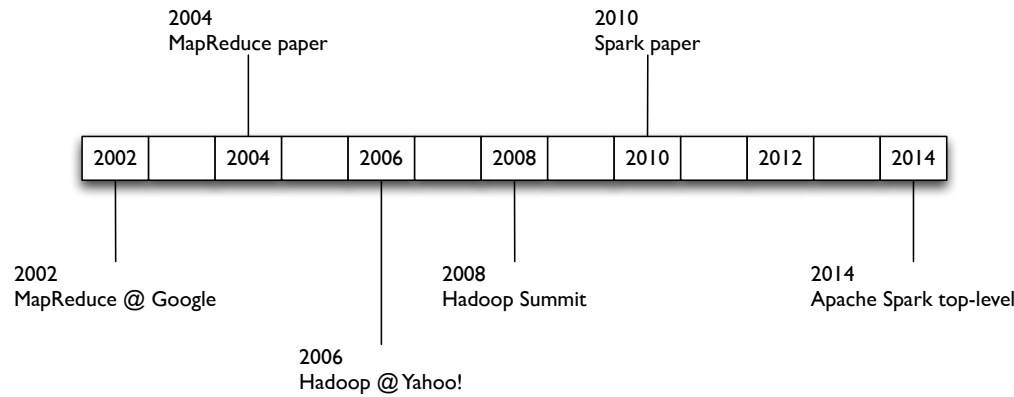


General Batch Processing

Specialized Systems:
iterative, interactive, streaming, graph, etc.

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

A Brief History: *Spark*



Spark: Cluster Computing with Working Sets
Matei Zaharia, Mosharaf Chowdhury,
Michael J. Franklin, Scott Shenker, Ion Stoica
USENIX HotCloud (2010)
people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave,
Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
NSDI (2012)
usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

A Brief History: *Spark*

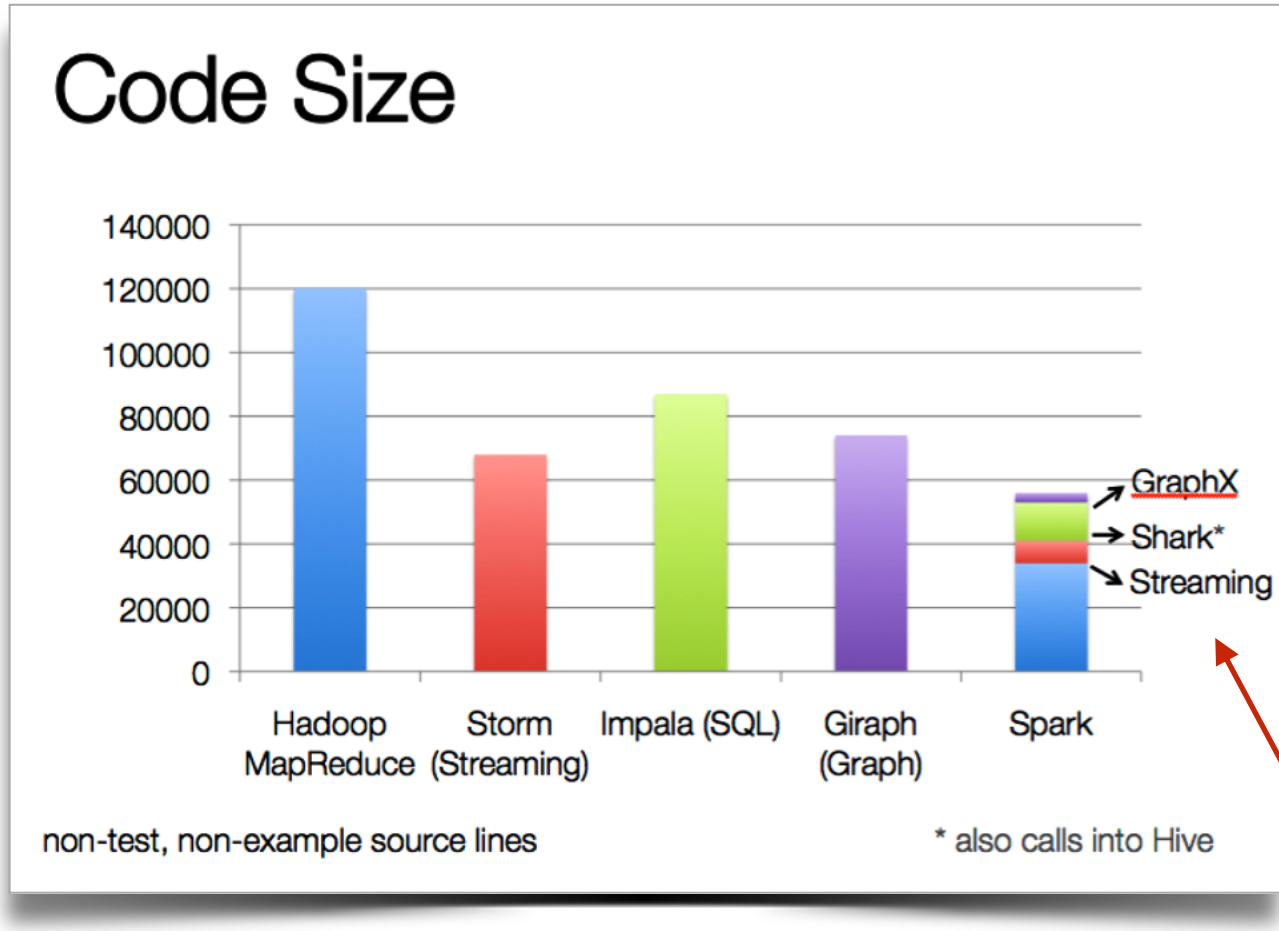
Unlike the various specialized systems, Spark's goal was to *generalize* MapReduce to support new apps within same engine

Two reasonably small additions are enough to express the previous models:

- *fast data sharing*
- *general DAGs*

This allows for an approach which is more efficient for the engine, and much simpler for the end users

A Brief History: *Spark*



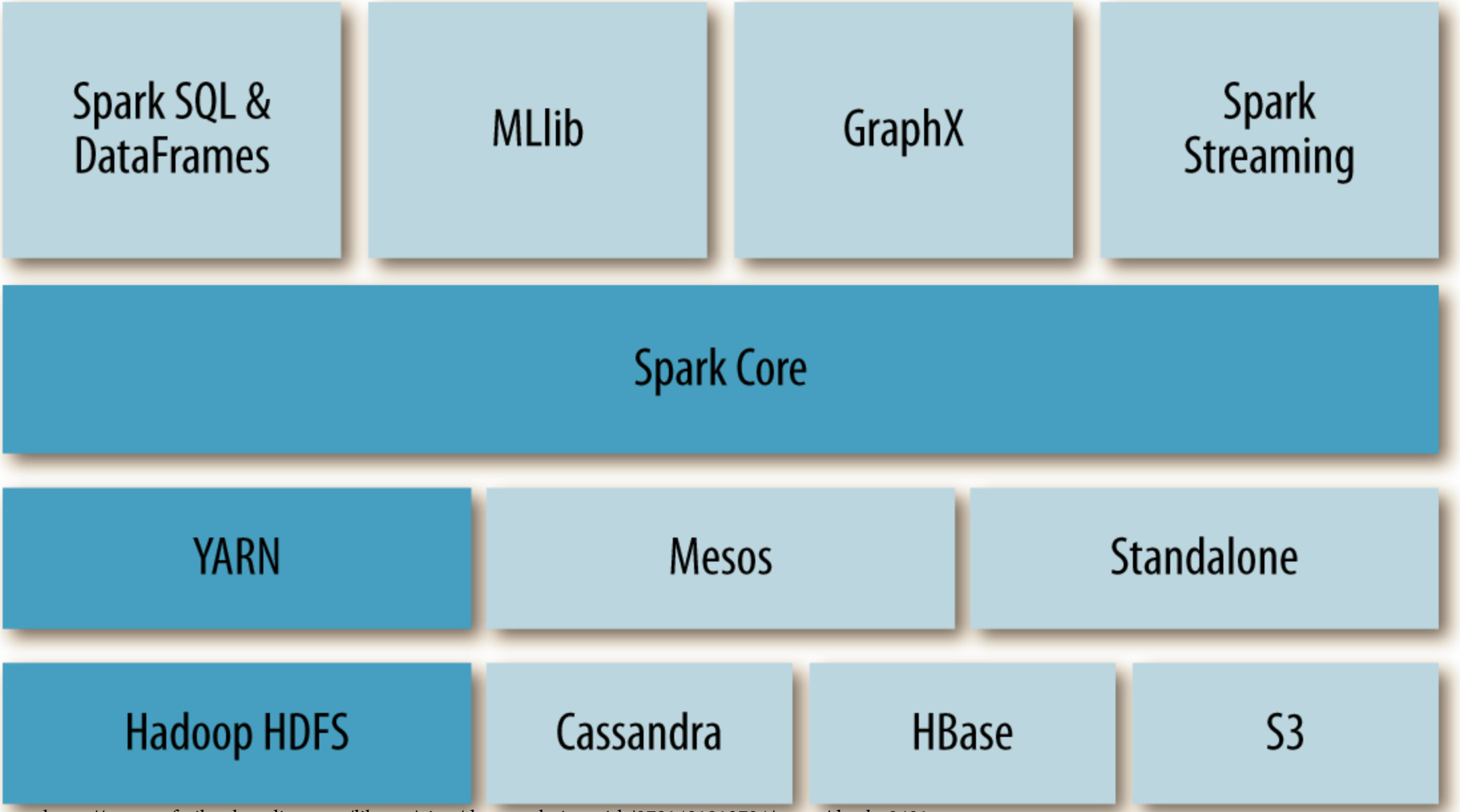
The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

used as libs, instead of specialized systems

A Brief History: *Spark*

Some key points about Spark:

- handles batch, interactive, and real−time within a single framework
- native integration with Java, Python, Scala
- programming at a higher level of abstraction
- more general: map/reduce is just one set of supported constructs



https://www.safaribooksonline.com/library/view/data-analytics-with/9781491913734/assets/dawh_0401.png

Resilient Distributed Datasets

A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, Ion Stoica

UC Berkeley

Motivation

MapReduce greatly simplified "big data" analysis on large, unreliable clusters

But as soon as it got popular, users wanted more:

- More **complex**, multi-stage applications (e.g. iterative machine learning & graph processing)
- More **interactive** ad-hoc queries

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

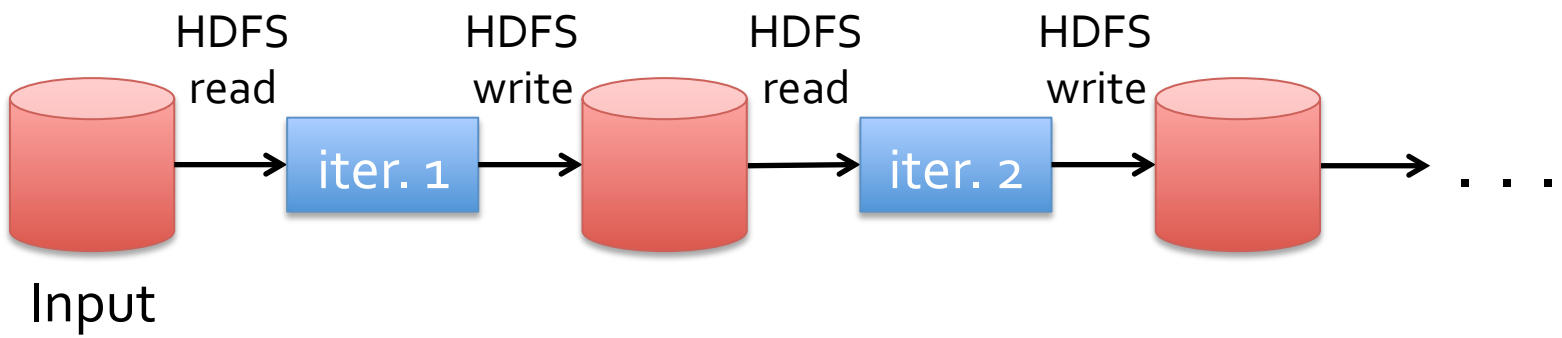
Motivation

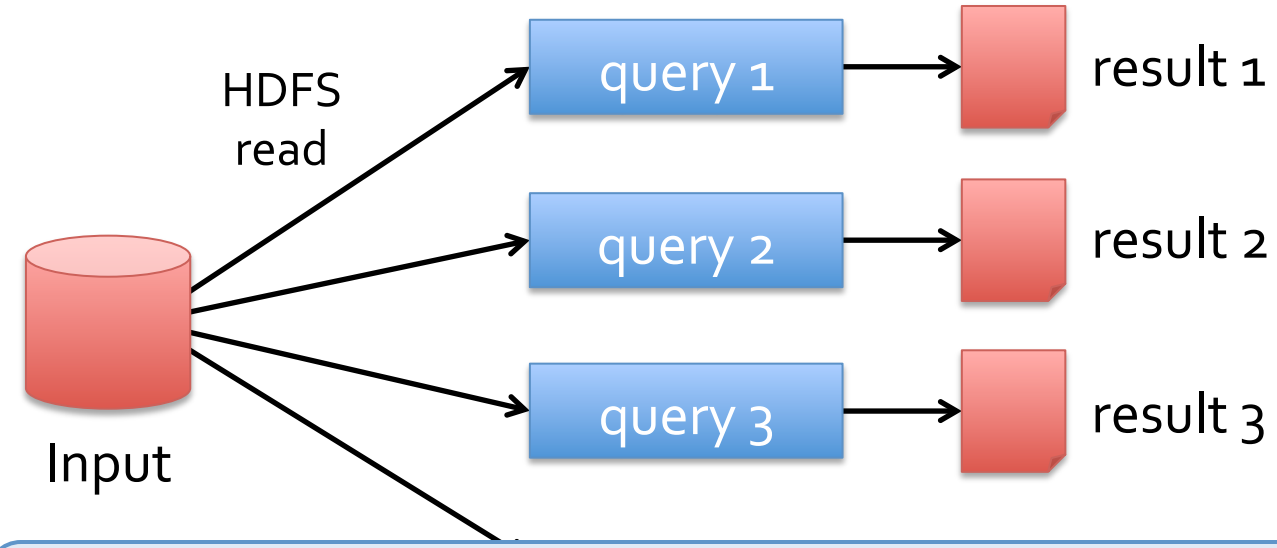
Complex apps and interactive queries both need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

In MapReduce, the only way to share data across jobs is stable storage ➔ slow!

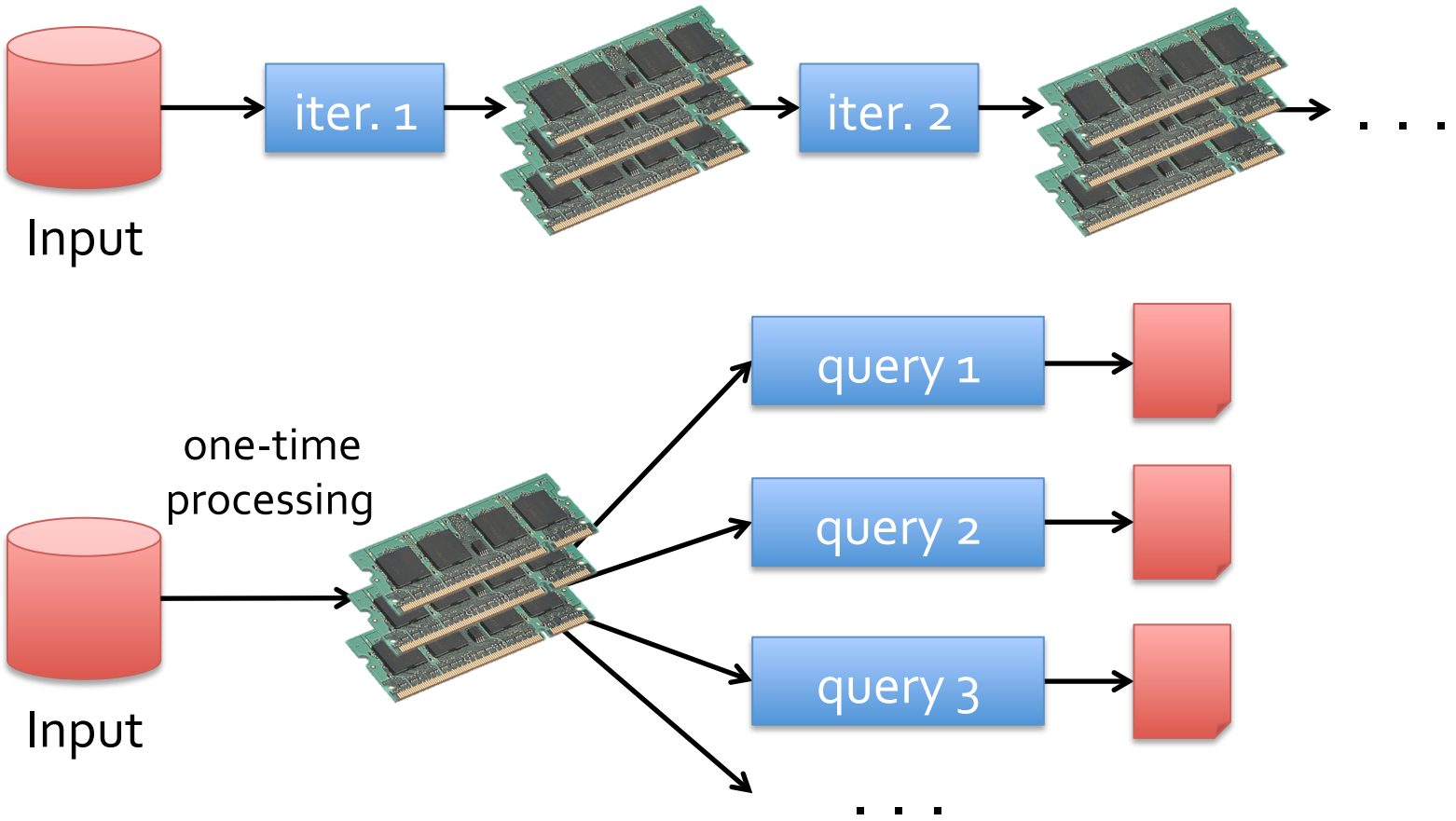
Examples





Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-Memory Data Sharing



10-100× faster than network/disk, but how to get FT?

Challenge

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

Challenge

Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state

- » RAMCloud, databases, distributed mem, Piccolo

Requires replicating data or logs across nodes for fault tolerance

- » Costly for data-intensive apps
- » 10-100x slower than memory write

Solution: Resilient Distributed Datasets (RDDs)

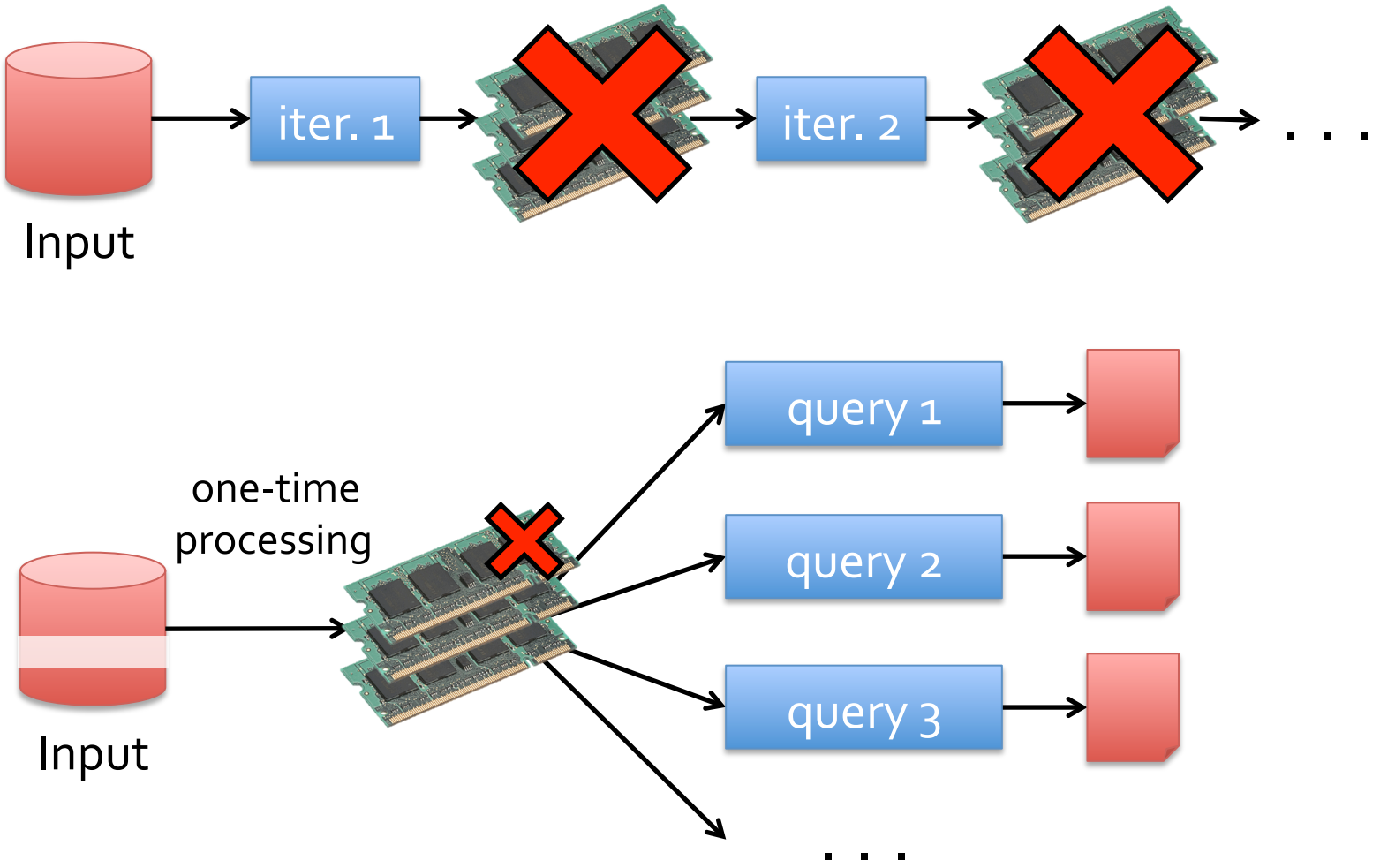
Restricted form of distributed shared memory

- » Immutable, partitioned collections of records
- » Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)

Efficient fault recovery using *lineage*

- » Log one operation to apply to many elements
- » Recompute lost partitions on failure
- » No cost if nothing fails

RDD Recovery



Generality of RDDs

Despite their restrictions, RDDs can express surprisingly many parallel algorithms

» These naturally *apply the same operation to many items*

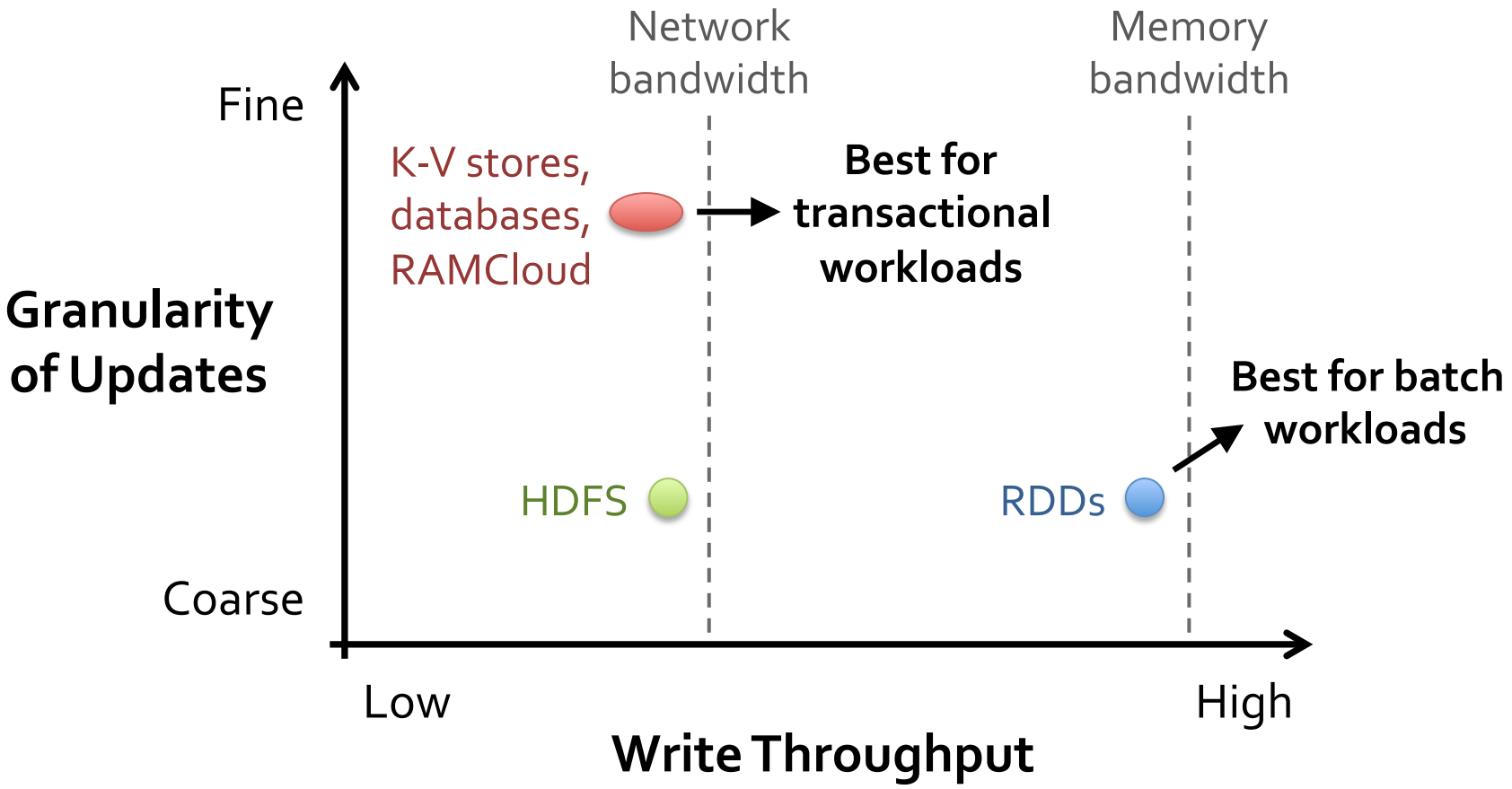
Unify many current programming models

» *Data flow models:* MapReduce, Dryad, SQL, …

» *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …

Support *new apps* that these models don't

Tradeoff Space



Spark Programming Interface

DryadLINQ-like API in the Scala language

Usable interactively from Scala interpreter

Provides:

- Resilient distributed datasets (RDDs)
- Operations on RDDs: *transformations* (build new RDDs), *actions* (compute and output results)
- Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

Spark Operations

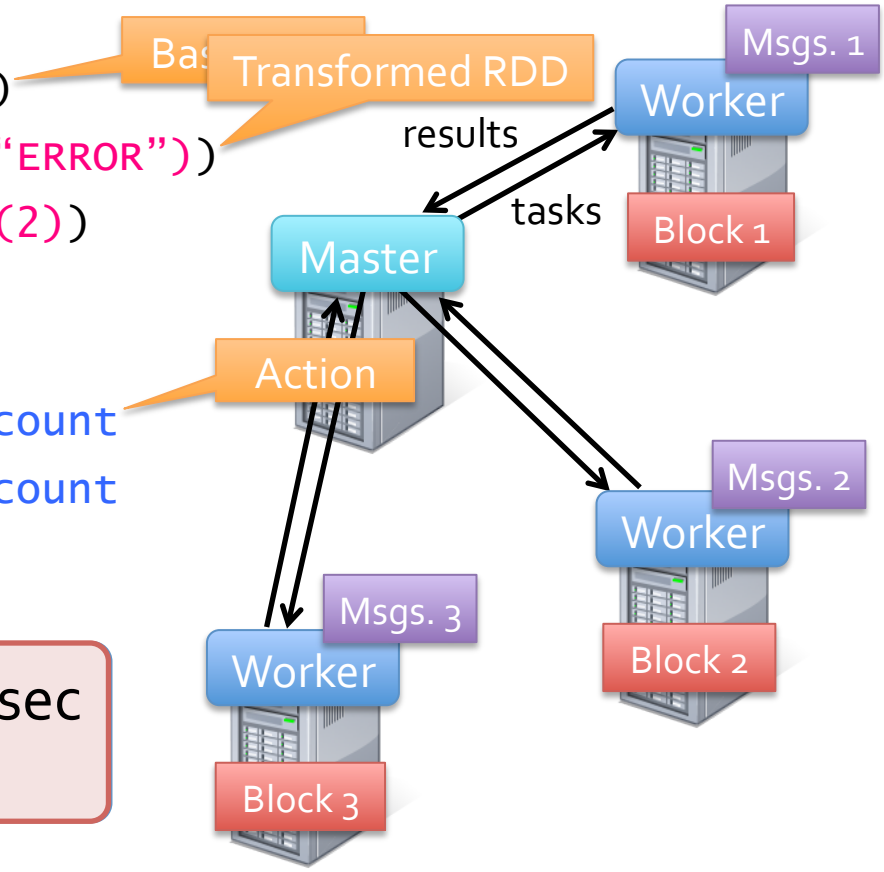
Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```



Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

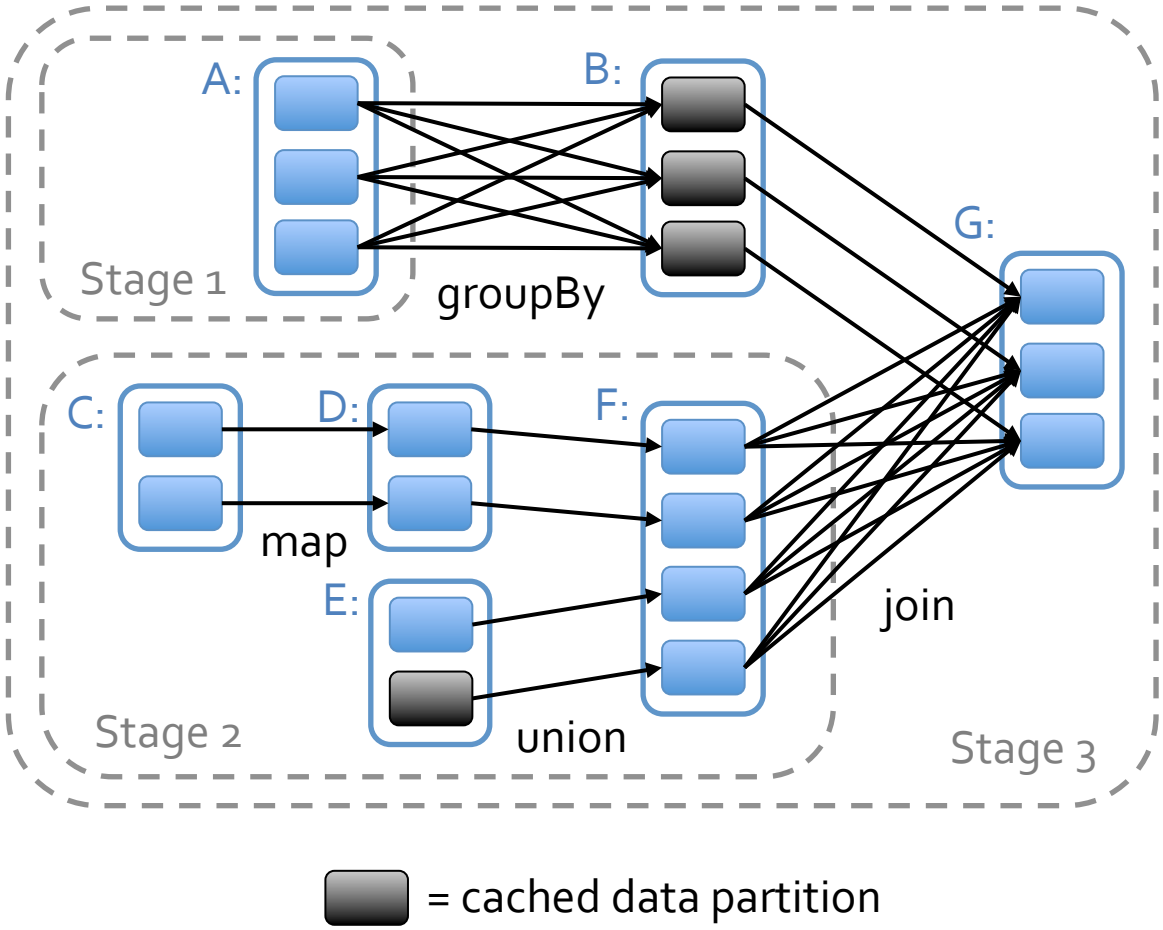
Task Scheduler

Dryad-like DAGs

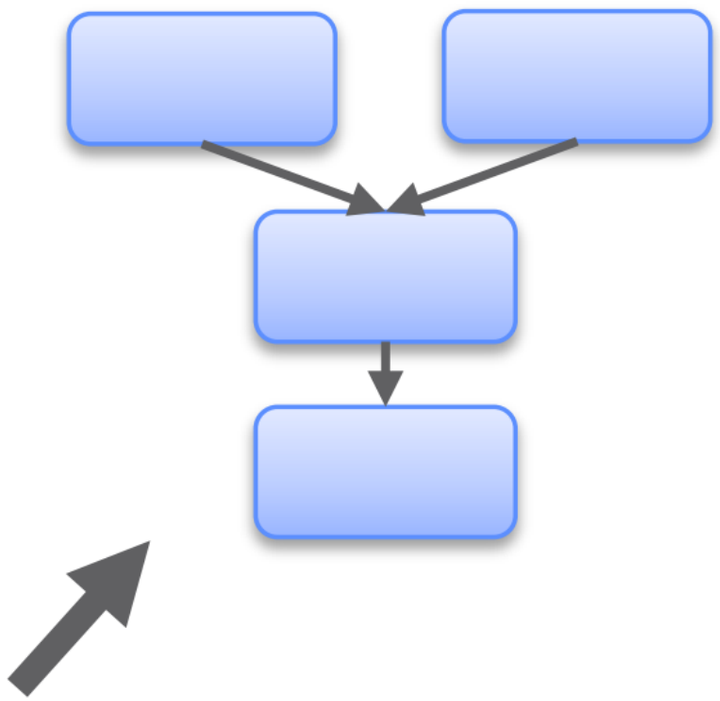
Pipelines functions within a stage

Locality & data reuse aware

Partitioning-aware to avoid shuffles



RDD Objects

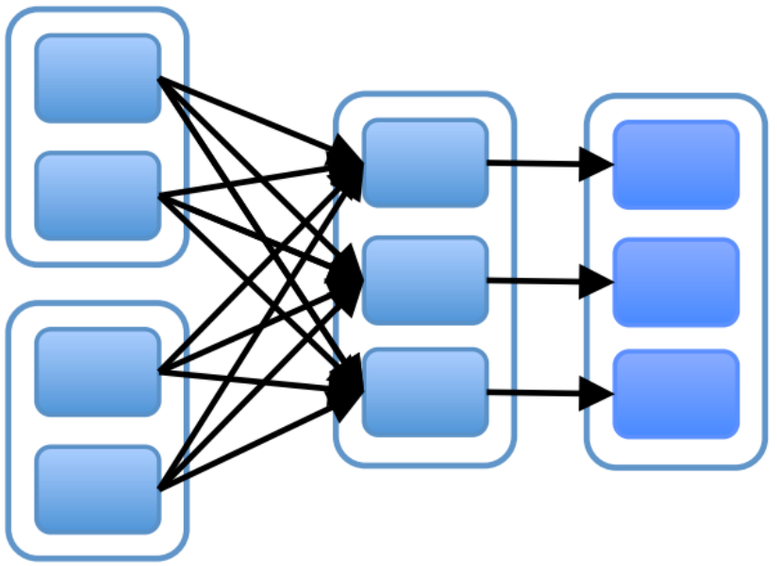


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

build operator DAG

Scheduler (DAGScheduler)

DAG



split graph into *stages* of tasks

submit each stage as ready

https://databricks-training.s3.amazonaws.com/slides/advanced-spark-training.pdf

What is RDD?

Resilient Distributed Dataset

- A big collection of data with following properties
  - Immutable
  - Distributed
  - Lazily evaluated
  - Type inferred
  - Cacheable

https://www.slideshare.net/datamantra/anatomy-of-rdd

Pseudo Monad

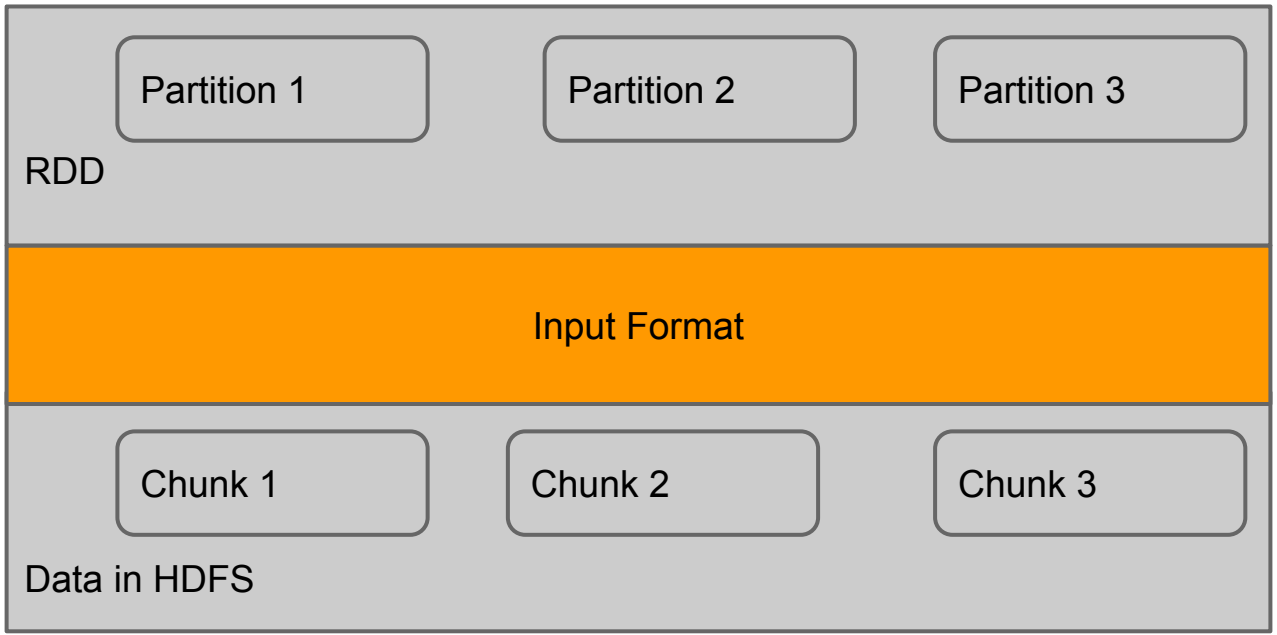
- Wraps iterator + partitions distribution
- Keeps track of history for fault tolerance
- Lazily evaluated, chaining of expressions

https://www.slideshare.net/deanchen11/scala-bay-spark-talk

Partitions

- Logical division of data
- Derived from Hadoop Map/Reduce
- All Input,Intermediate and output data will be represented as partitions
- Partitions are basic unit of parallelism
- RDD data is just collection of partitions

Partition from Input Data



Partition and Immutability

- All partitions are immutable
- Every transformation generates new partition
- Partition immutability driven by underneath storage like HDFS
- Partition immutability allows for fault recovery

Partitions and Distribution

- Partitions derived from HDFS are distributed by default
- Partitions also location aware
- Location awareness of partitions allow for data locality
- For computed data, using caching we can distribute in memory also

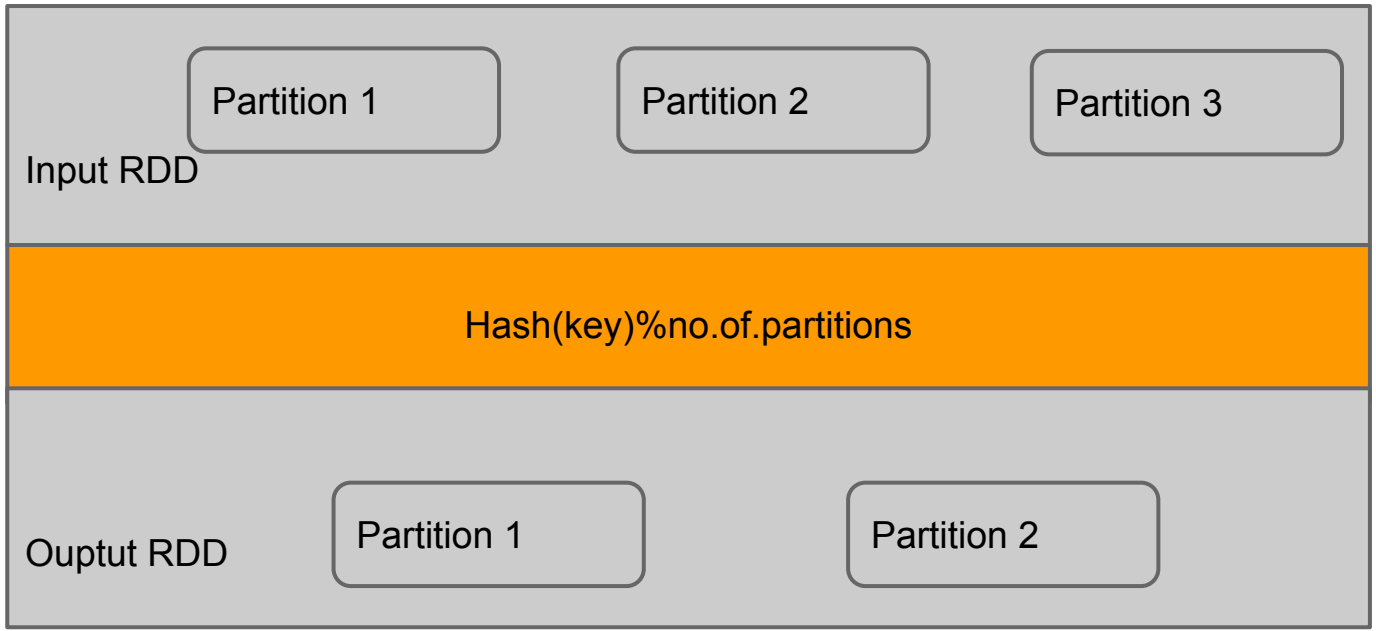
Accessing partitions

- We can access partition together rather single row at a time
- mapPaririton̄s API of RDD allows us that
- Accessing partition at a time allows us to do some partionwise operation which cannot be done by accessing single row.

Partition for transformed Data

- Partitioning will be different for key/value pairs that are generated by shuffle operation
- Partitioning is driven by partitioner specified
- By default HashPartitioner is used
- You can use your own partitioner also

Hash Partitioning



Custom Partitioner

- Partition the data according to your data structure
- Custom partitioning allows control over no of partitions and the distribution of data across when grouping or reducing is done

Look up operation

- Partitioning allows faster lookups
- Lookup operation allows to look up for a given value by specifying the key
- Using partitioner, lookup determines which partition look for
- Then it only need to look in that partition
- If no partition is specified, it will fallback to filter

Parent(Dependency)

- Each RDD has access to it’s parent RDD
- Nil is the value of parent for first RDD
- Before computing it’s value, it always computes it’s parent
- This chain of running allows for laziness

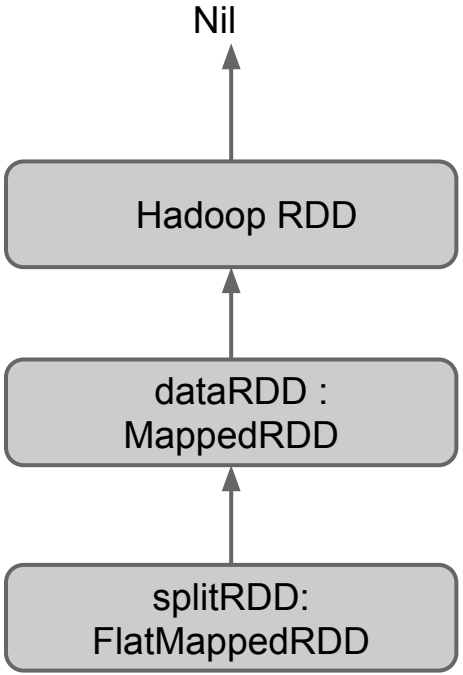
Sub classing

- Each spark operator, creates an instance of specific sub class of RDD
- map operator results in MappedRDD, flatMap in FlatMappedRDD etc
- Subclass allows RDD to remember the operation that is performed in the transformation

RDD transformations

```
val dataRDD =
sc.textFile(args
(1))

val splitRDD =
dataRDD.
flatMap(value =>
value.split(“ “)
```



Compute

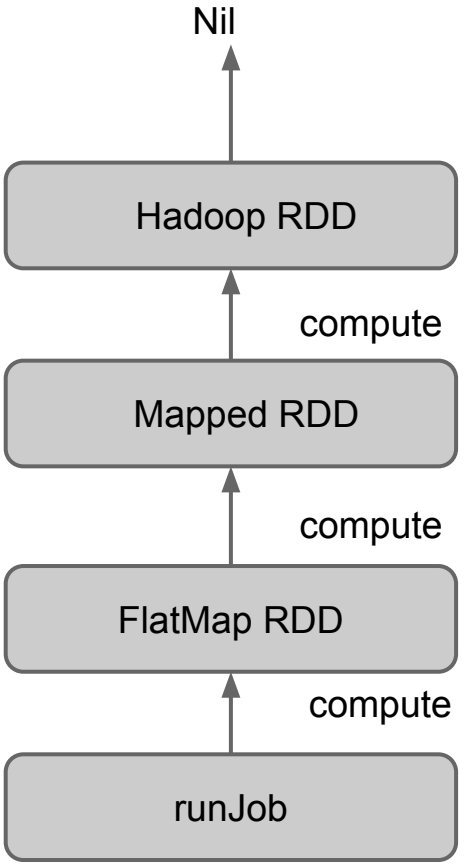
- Compute is the function for evaluation of each partition in RDD
- Compute is an abstract method of RDD
- Each sub class of RDD like MappedRDD, FilteredRDD have to override this method

RDD actions

```
val dataRDD = sc.
textFile(args(1))

val flatMapRDD =
dataRDD.flatMap
(value => value.split(“
“)

flatMapRDD.collect()
```



runJob API

- runJob API of RDD is the api to implement actions
- runJob allows to take each partition and allow you evaluate
- All spark actions internally use runJob api.

Caching

- cache internally uses persist API
- persist sets a specific storage level for a given RDD
- Spark context tracks persistent RDD
- When first evaluates, partition will be put into memory by block manager

Block manager

- Handles all in memory data in spark
- Responsible for
  - Cached Data (BlockRDD)
  - Shuffle Data
  - Broadcast data
- Partition will be stored in Block with id (RDD. id, partition_index)

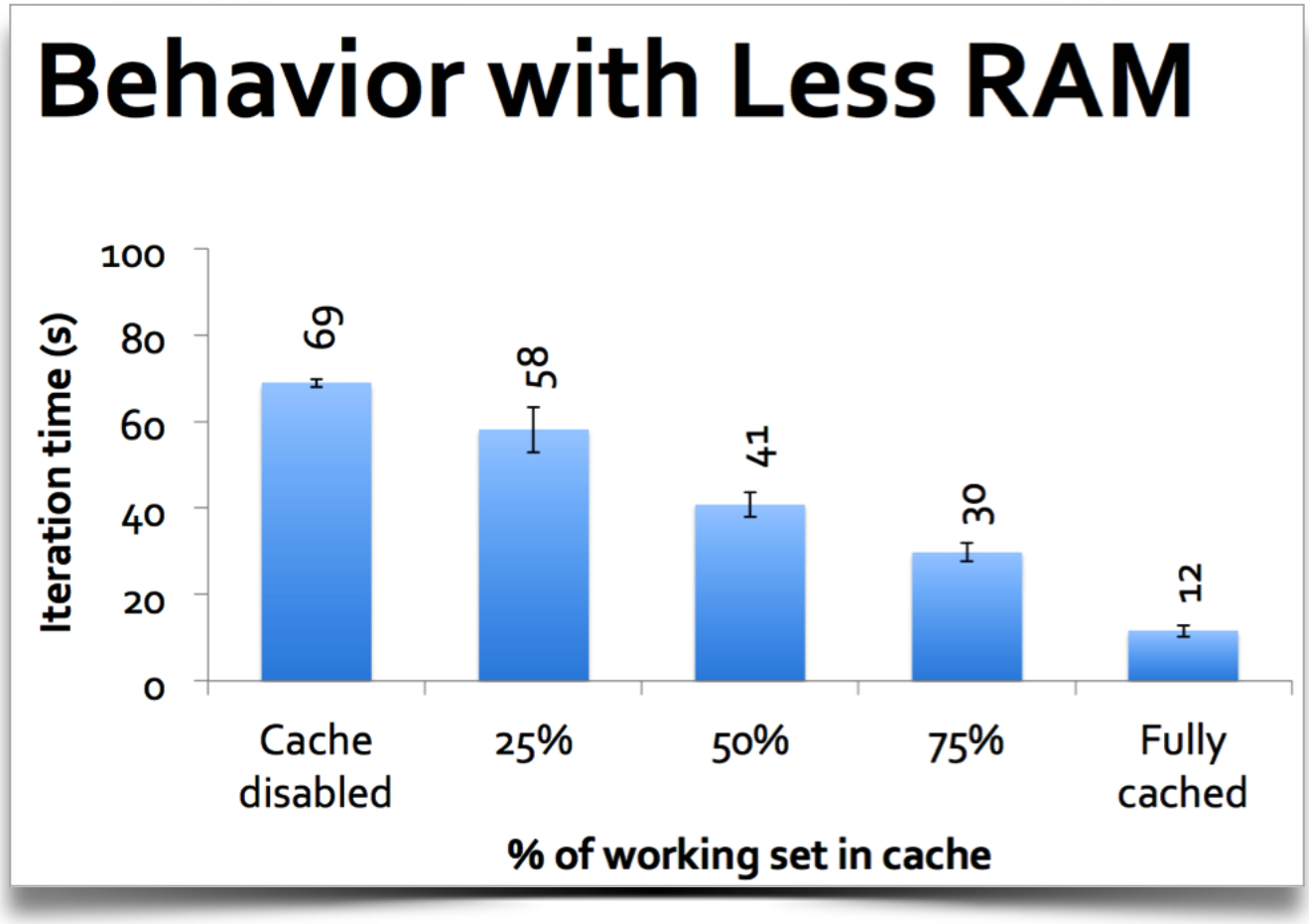
How caching works?

- Partition iterator checks the storage level
- if Storage level is set it calls

  cacheManager.getOrCompute(partition)

- as iterator is run for each RDD evaluation, its transparent to user

A Brief History: *Spark*

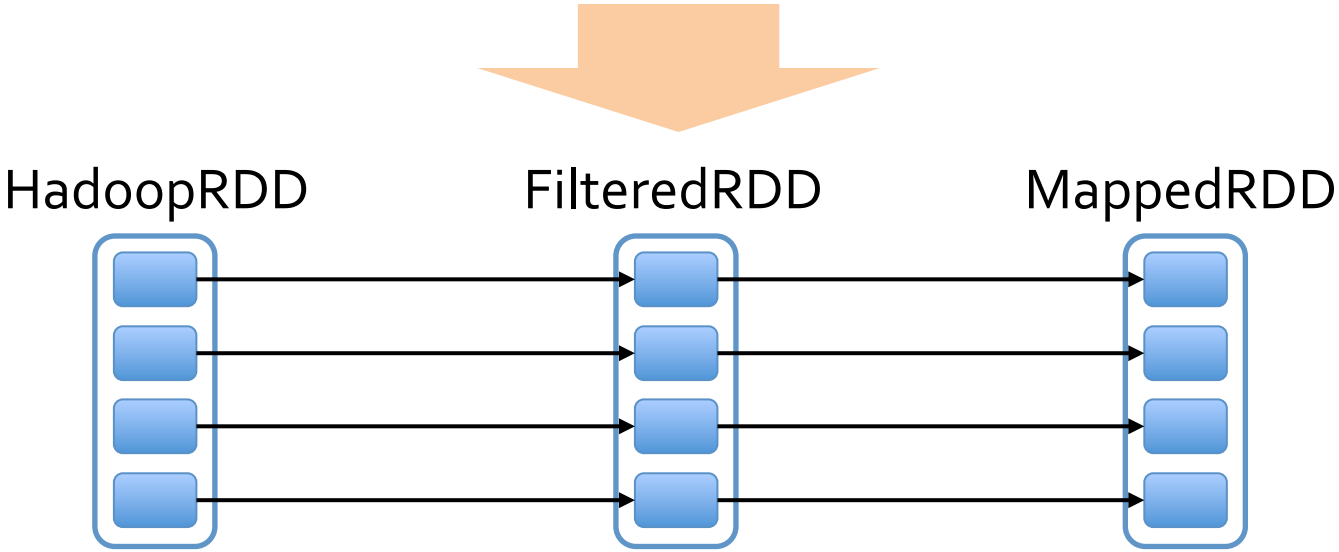


The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

Fault Recovery

RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Fault Recovery Results