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~~
Princeton

amplab
UC BERKELEY

Motivation

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

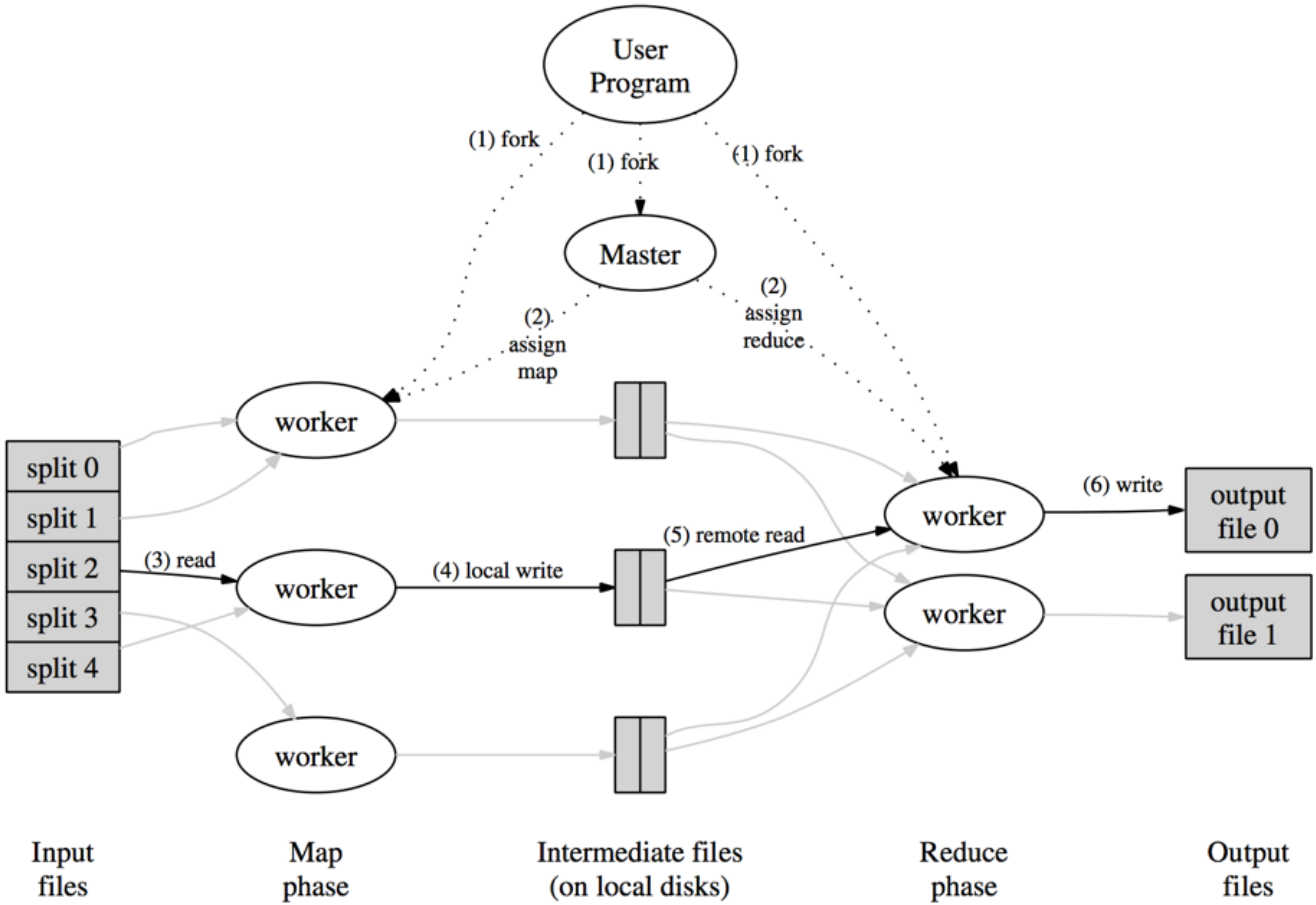
But as soon as it got popular, users wanted more:

» 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)

Map-Reduce System



Original Paper Example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

New Examples

Iteration

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

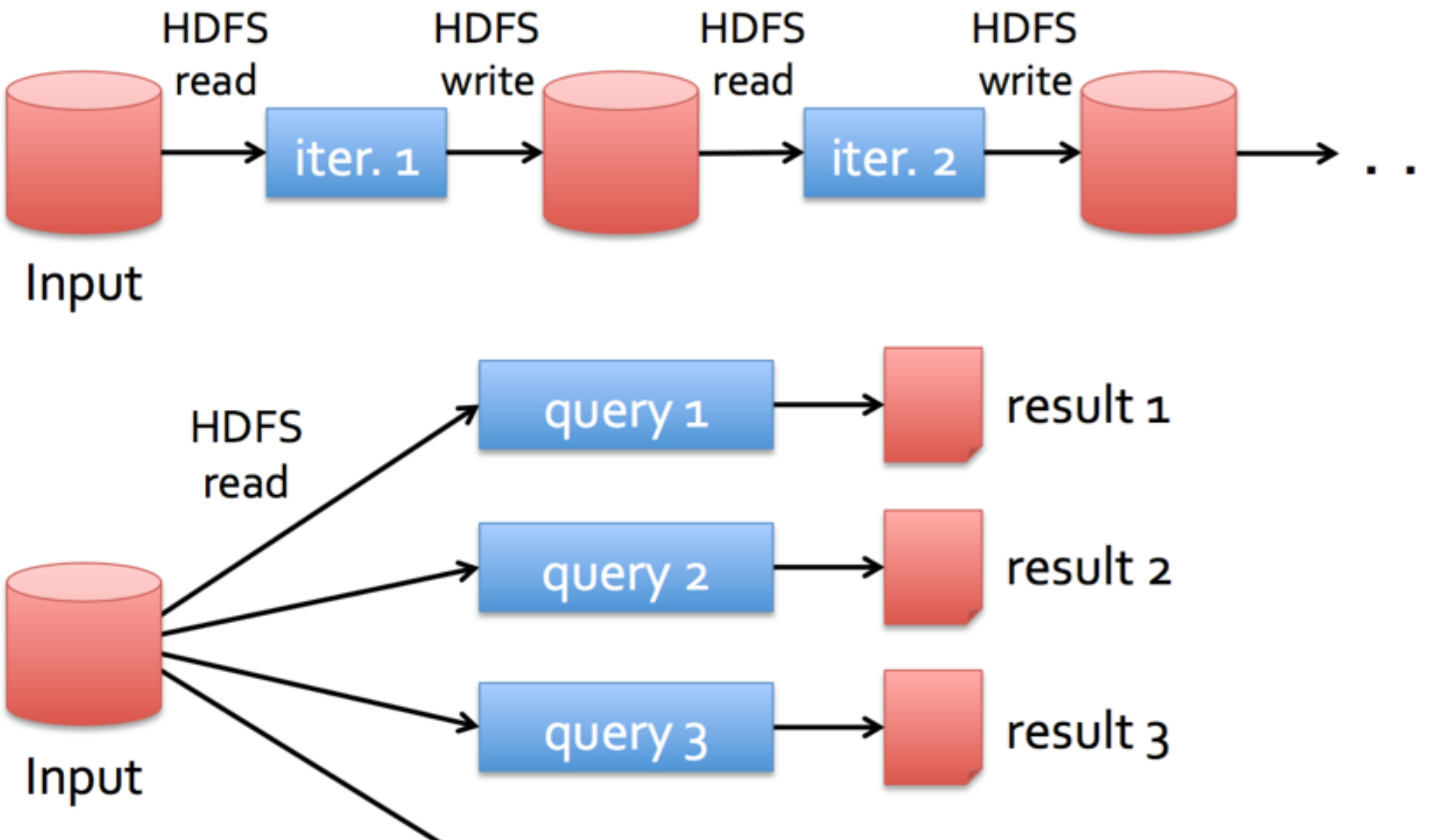
Multiple Queries

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

```
errors.count()
```

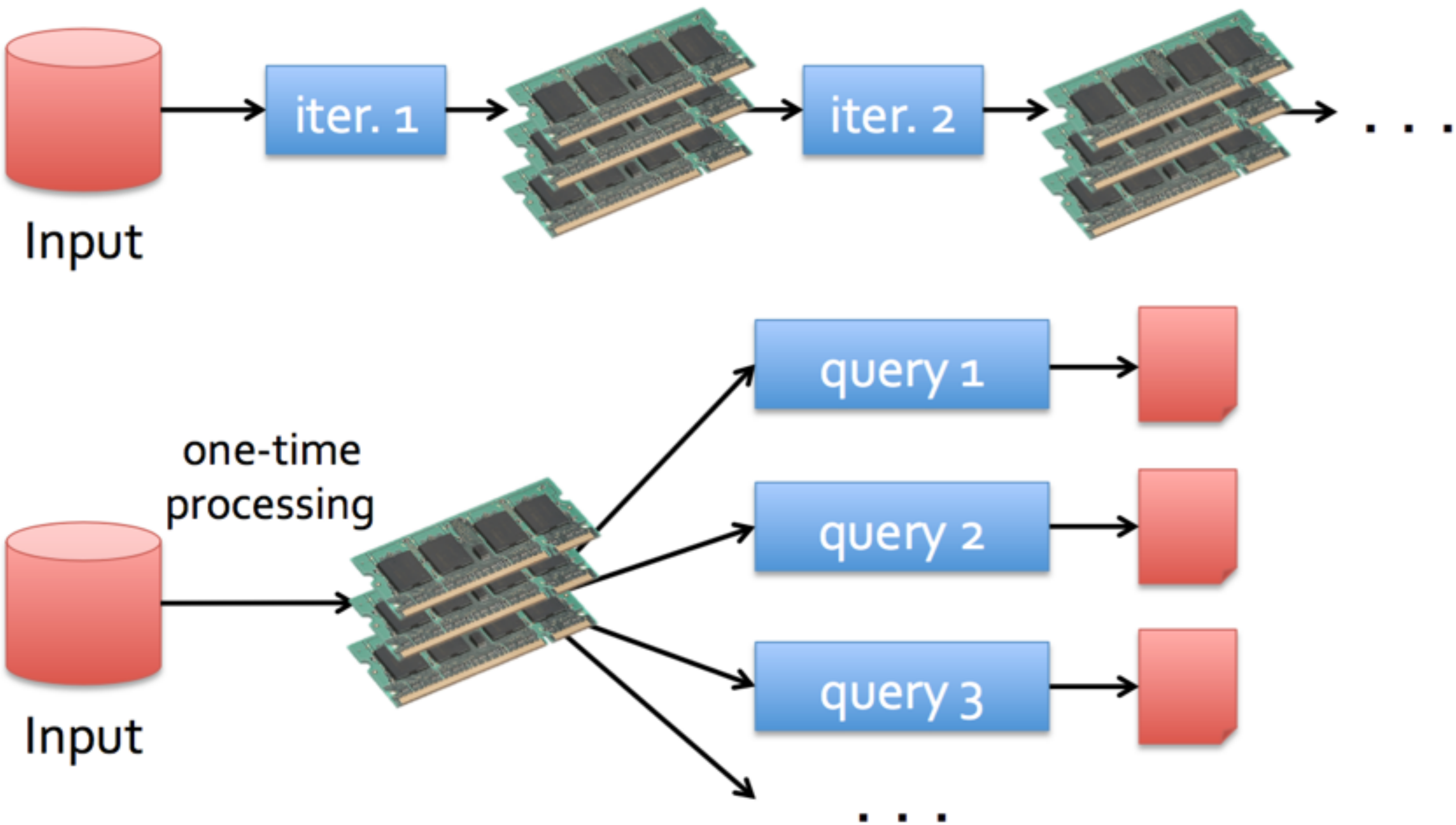
```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

New 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?

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!

Solution: Resilient Distributed Datasets (RDDs)

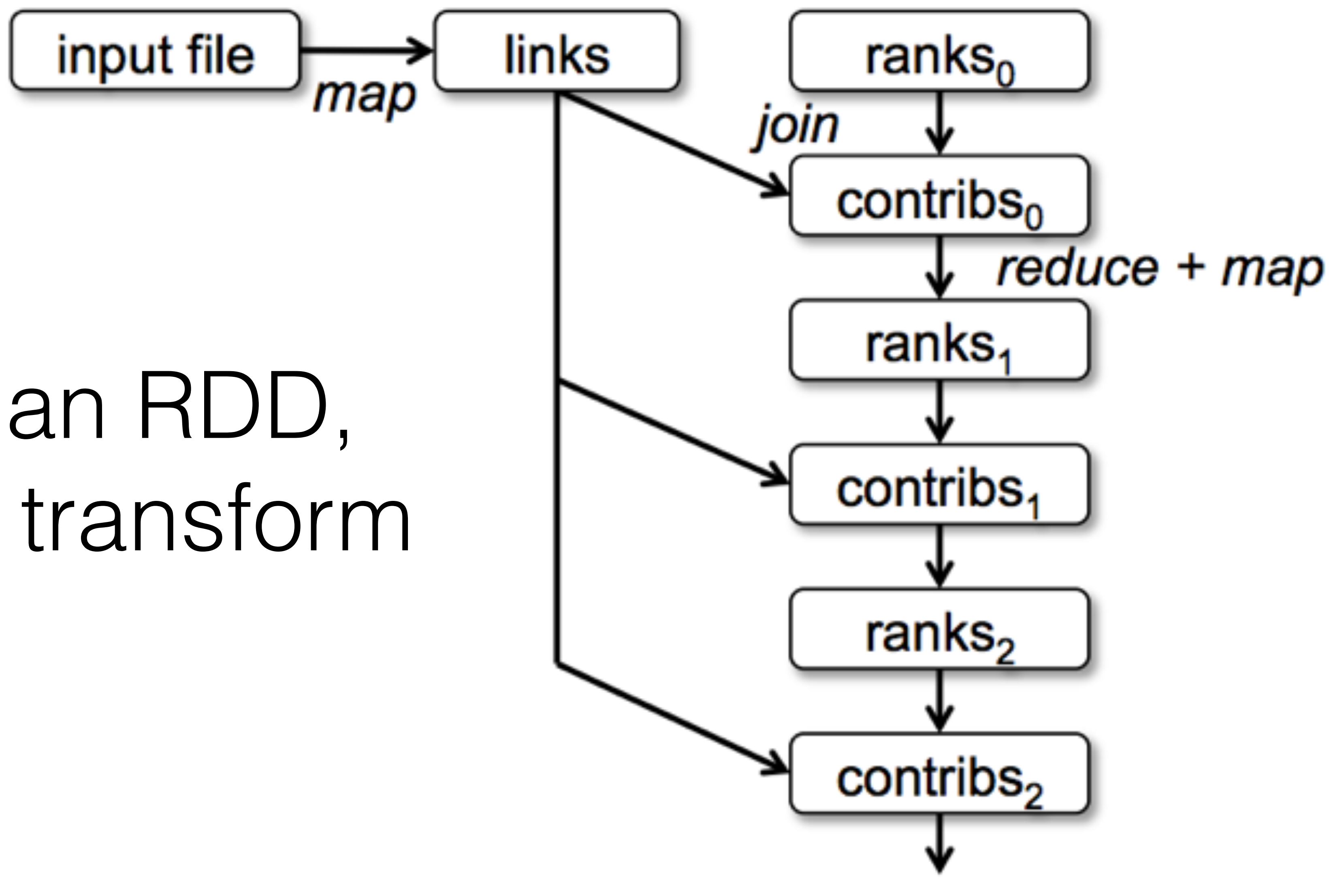
Restricted form of distributed shared memory

- » 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 Lineage Graph



Each Block is an RDD,
each arrow is a transform

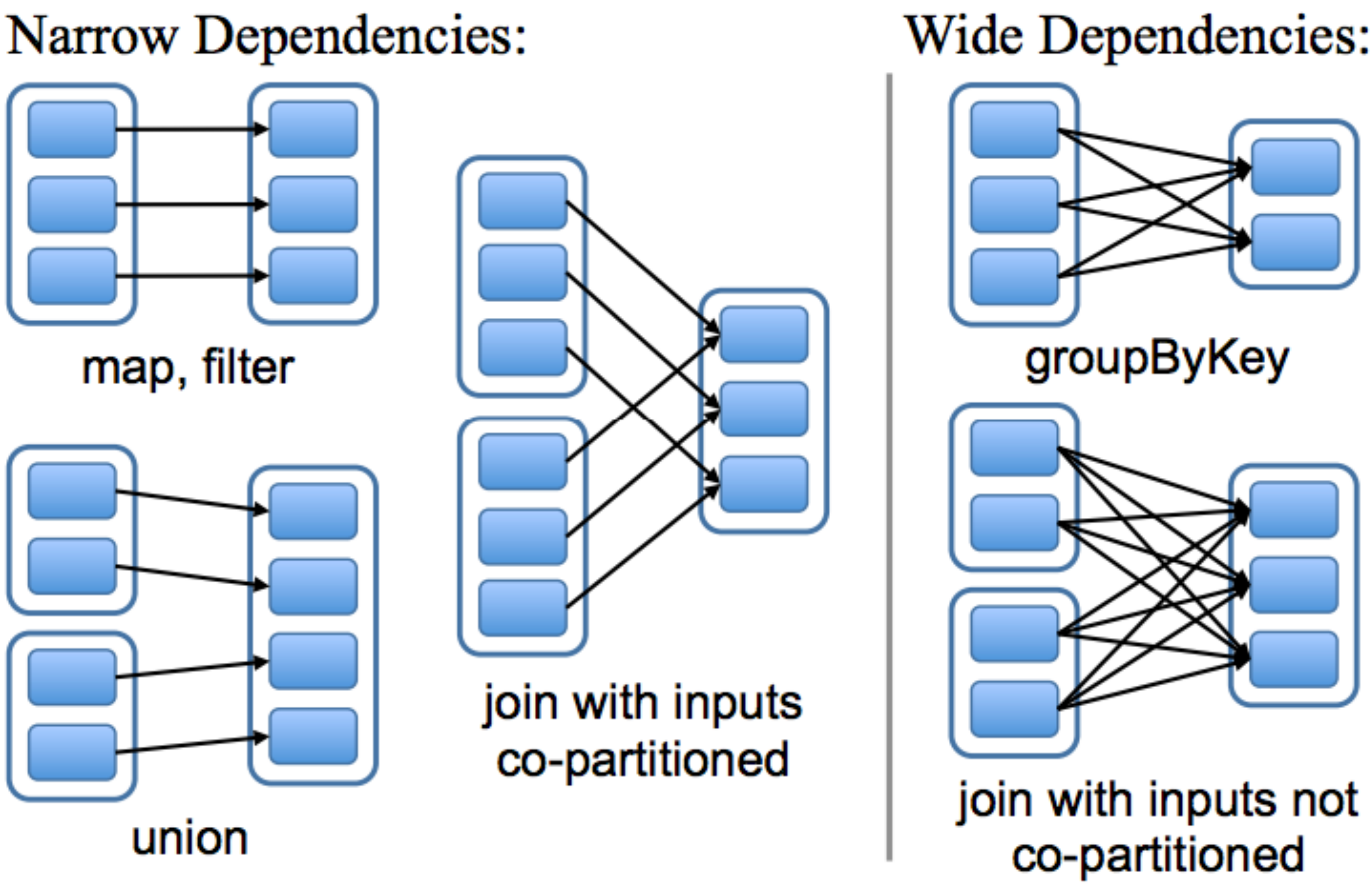
RDD Interface

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Partitioning

- By default, where the data is originally read from a distributed file system.
- Default intermediate partition is **hash(data) mod K**
- User can provide own partitioning function for improved data locality

Partition Dependencies

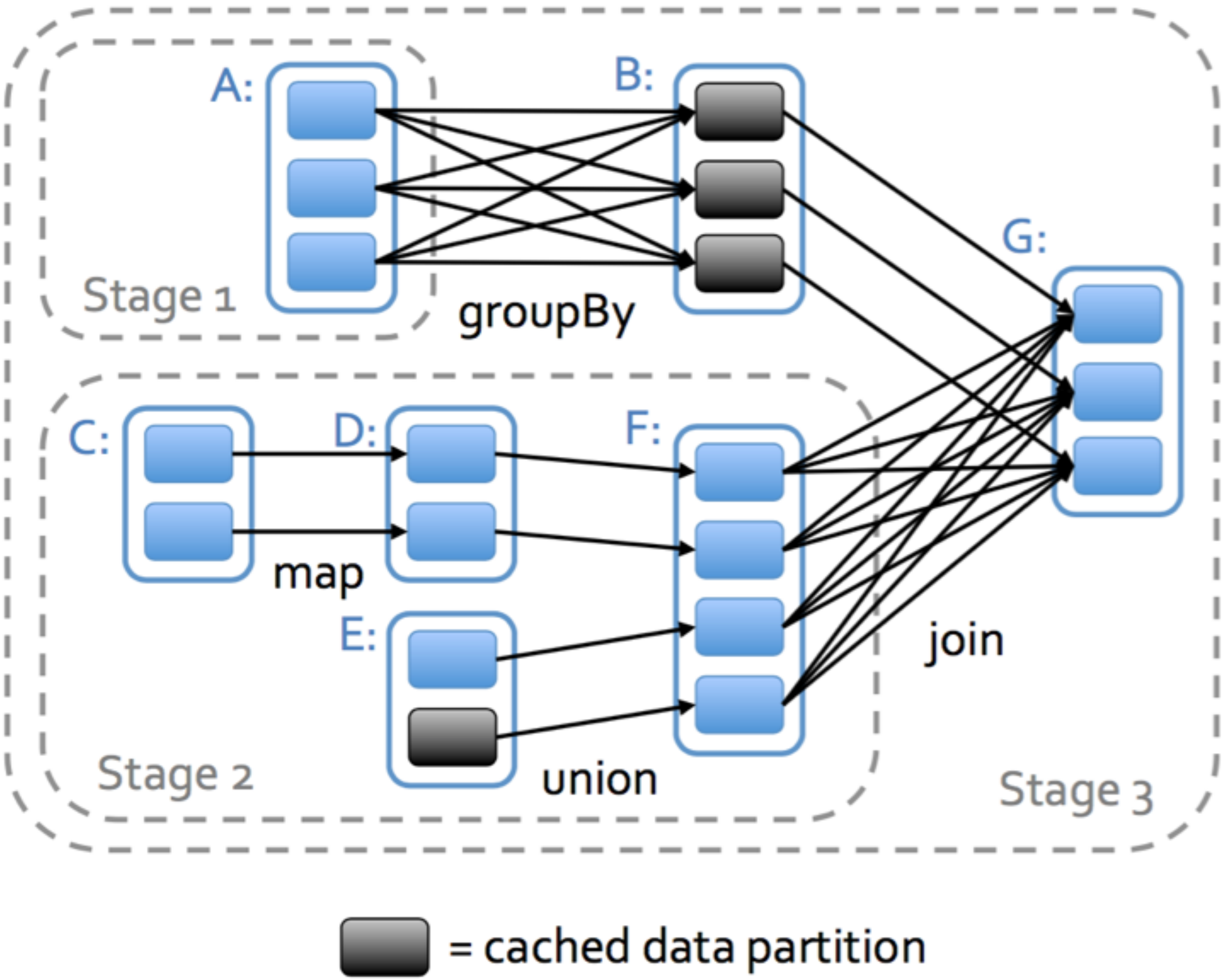


Partition Dependencies

- For narrow dependencies, partitions can be computed on same compute node as their parents

- Wide dependencies require communication and intermediate result storage

Staging



A:
B:
G:
Stage 1
groupBy
C:
D:
F:
map
E:
join
Stage 2
union
Stage 3
= cached data partition

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, …

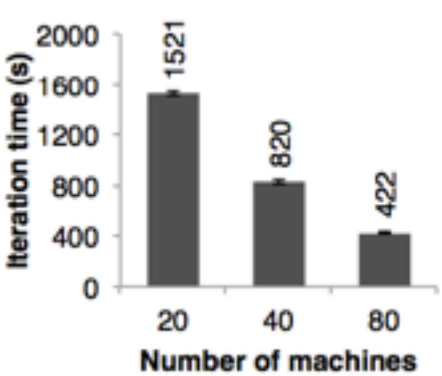
» *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …

Support *new apps* that these models don't

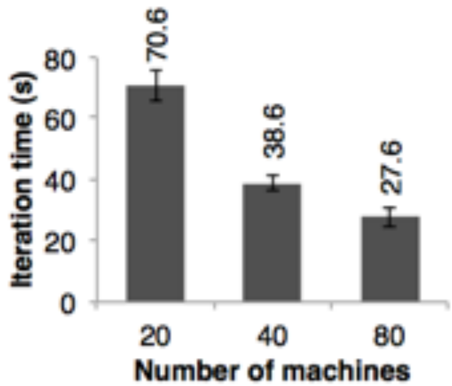
Operations

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	Outputs RDD to a storage system, *e.g.*, HDFS

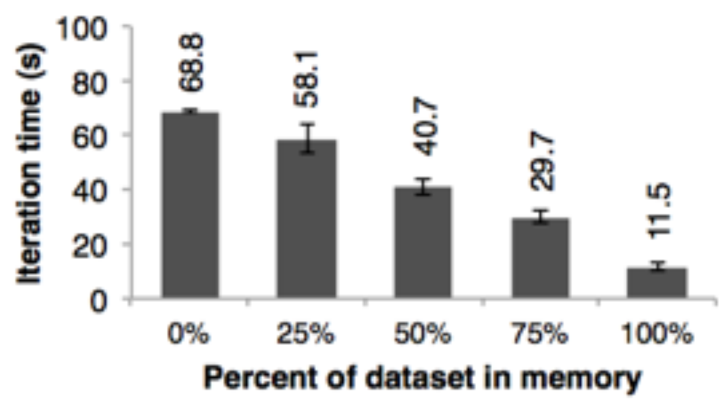
Evaluation

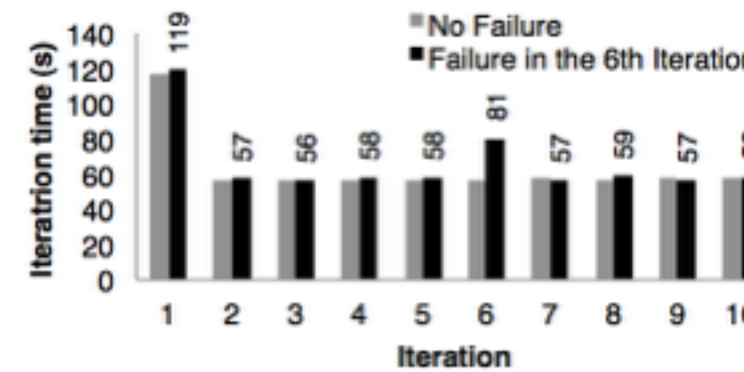


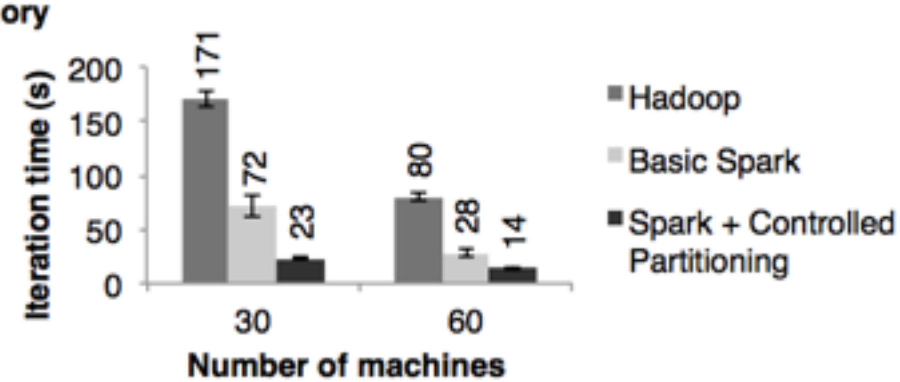
(a) Traffic modeling

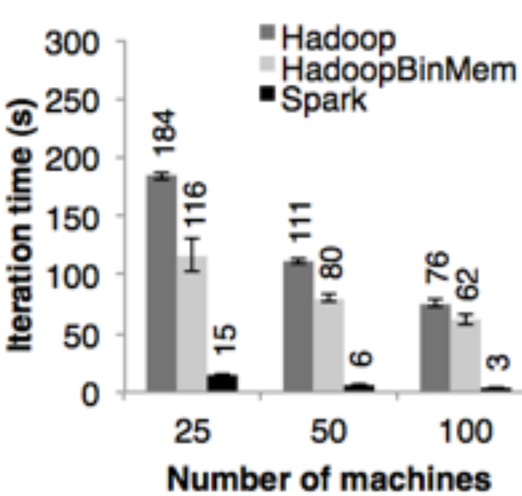


(b) Spam classification

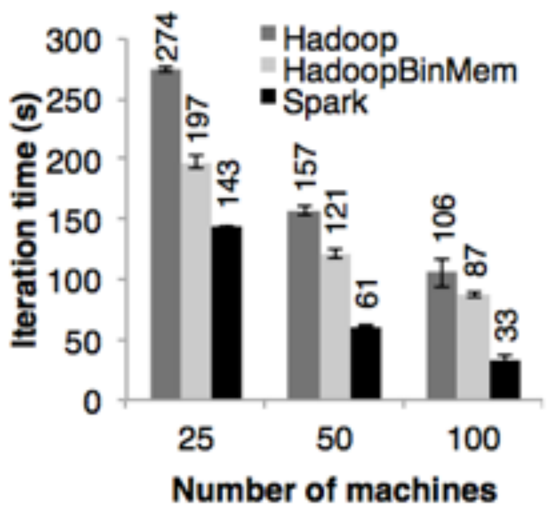




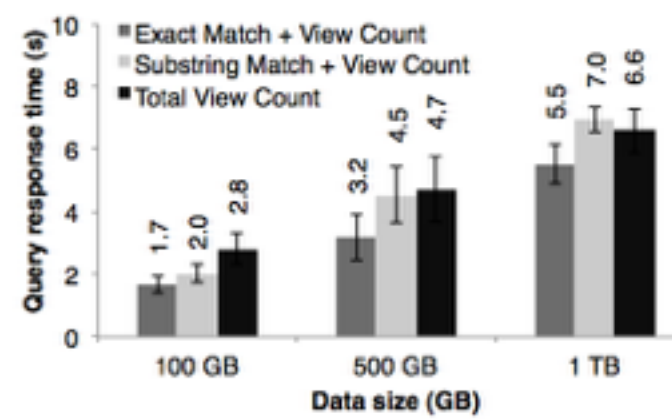


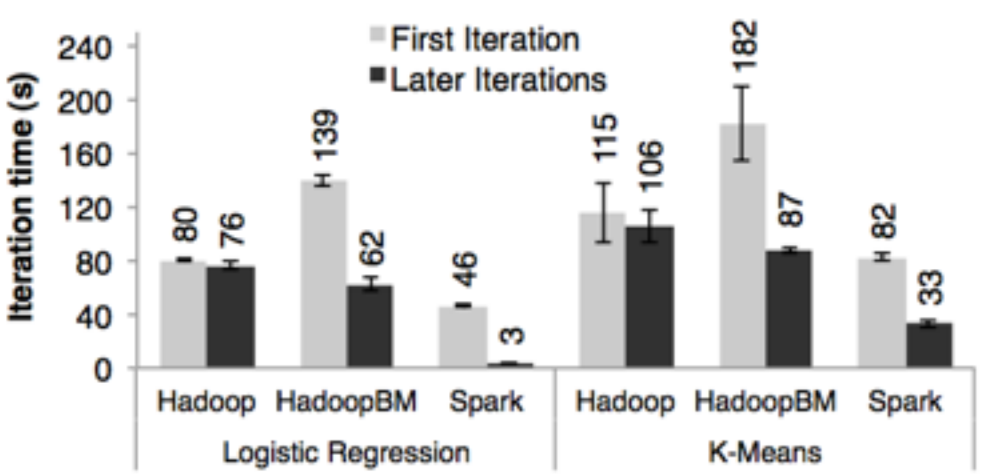


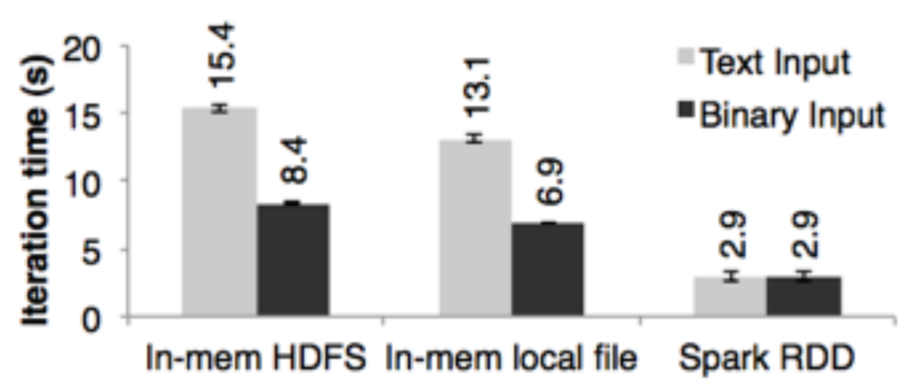
(a) Logistic Regression



(b) K-Means







THANKS