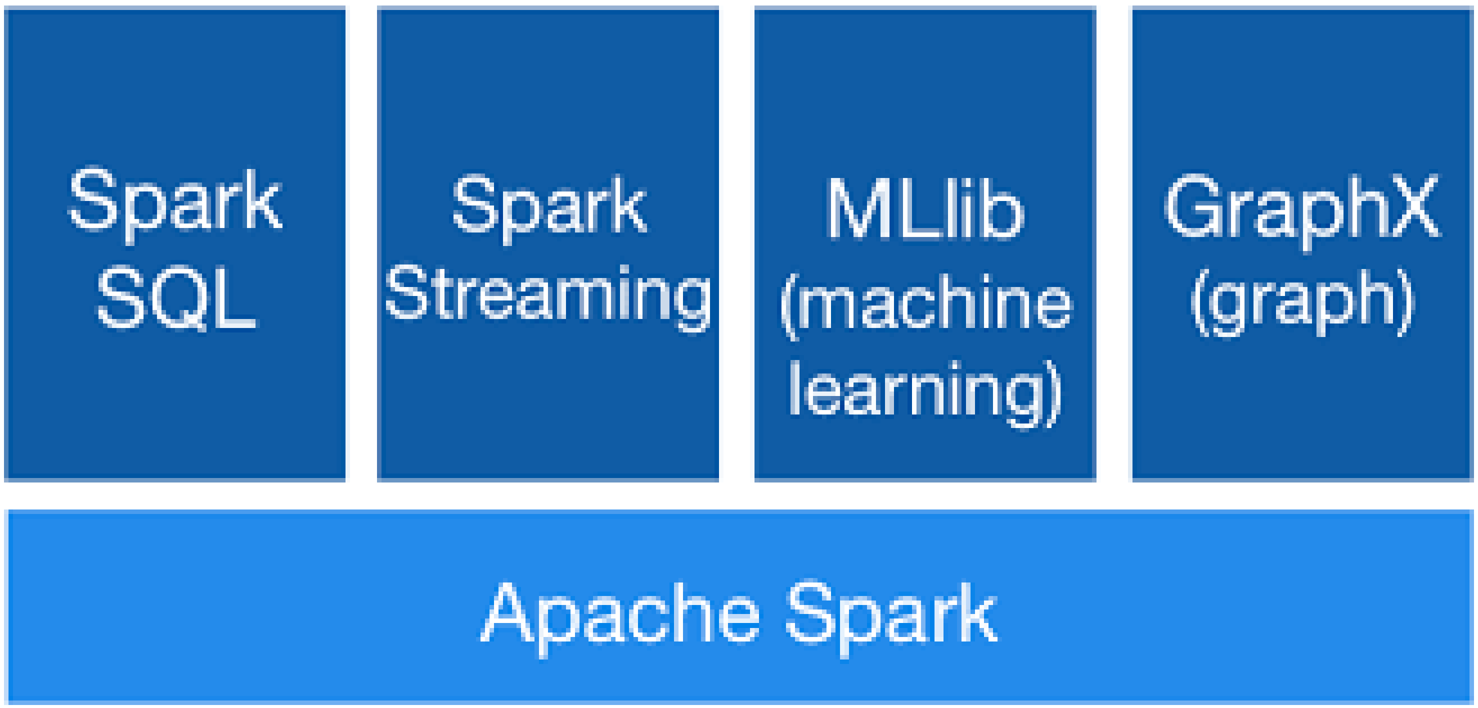
Apache Spark
Part 2

MATERIALS ADOPTED FROM
Berkeley's AMPLab ([Matei Zaharia – PhD student in 2009)](),
open-sourced in 2010 as BSD,
APACHE software foundation in 2013 as SPARK
Company formed DATABRICKS Inc! record in fast sorting in 2014.

Generic Efficient Infrastructure



Motivation : Workloads

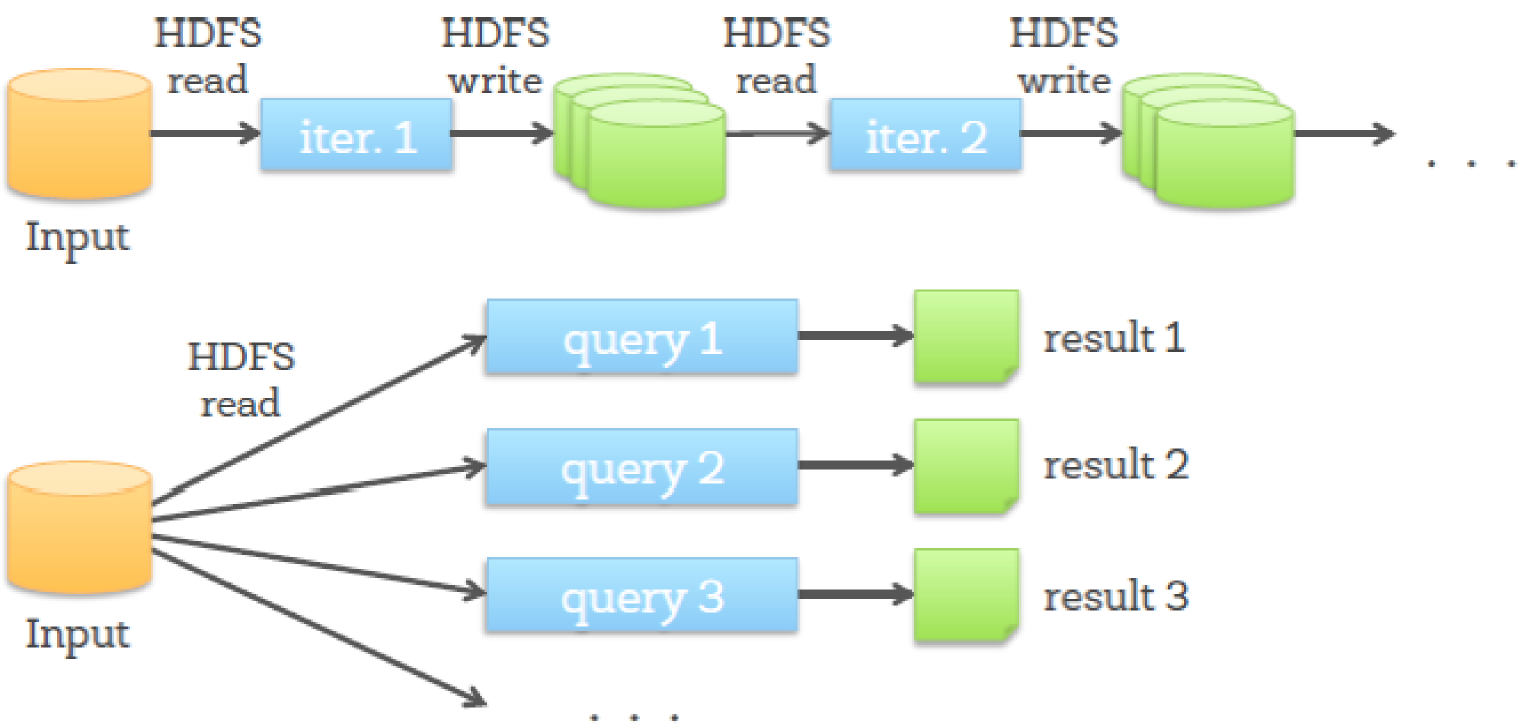
- Complex multi-pass algorithms
- Interactive ad-hoc queries
- Real-time steam processing

All need efficient data sharing and transfer

Motivation: Serving Workloads

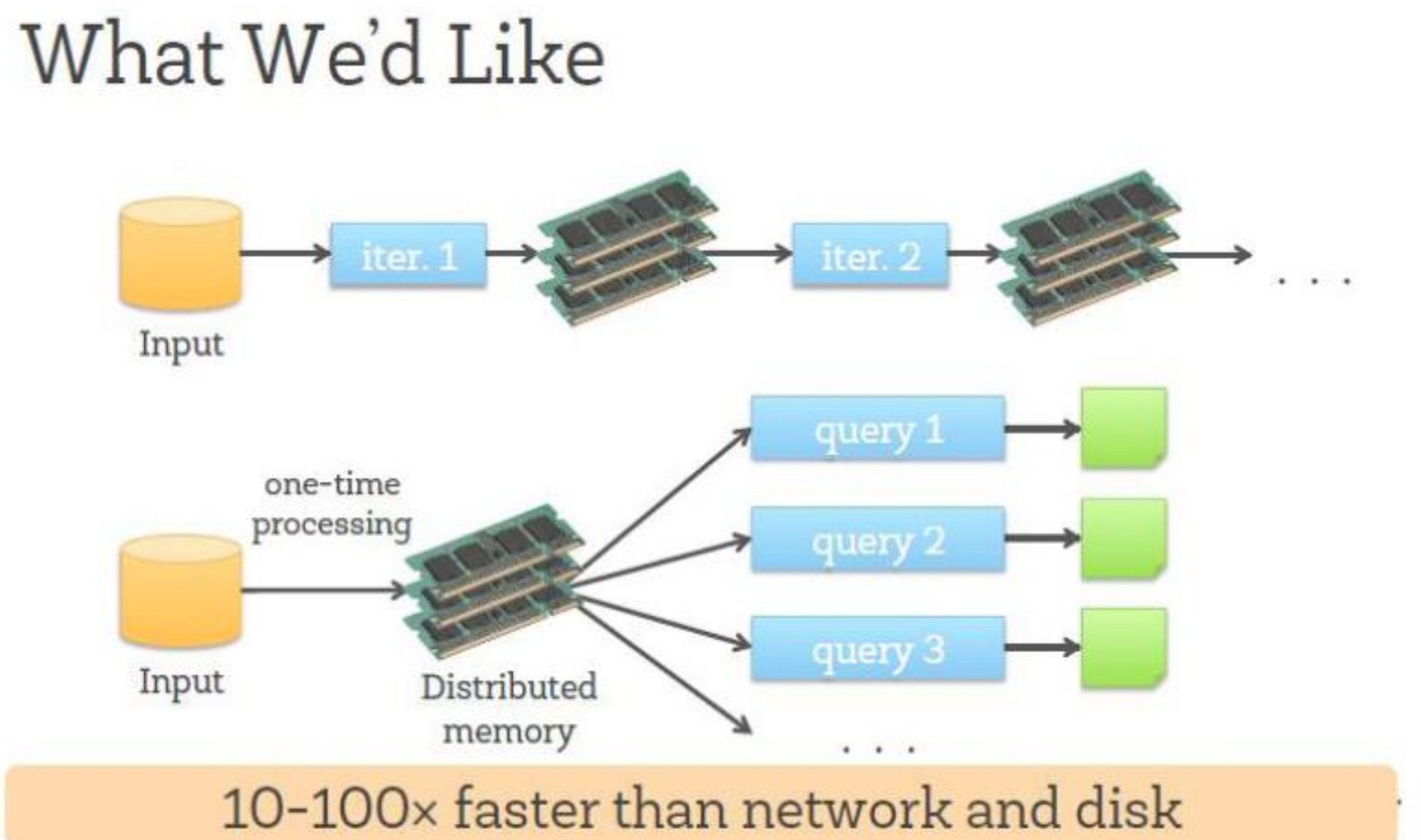
From This …

Data Sharing in MapReduce



Slow due to data replication and disk I/O

Motivation Workloads
Thus To This …



Motivation
From Hardware Side

- RAM is getting much cheaper
- Commodity machines with GBs of RAM
- Large Distributed RAM in the cluster

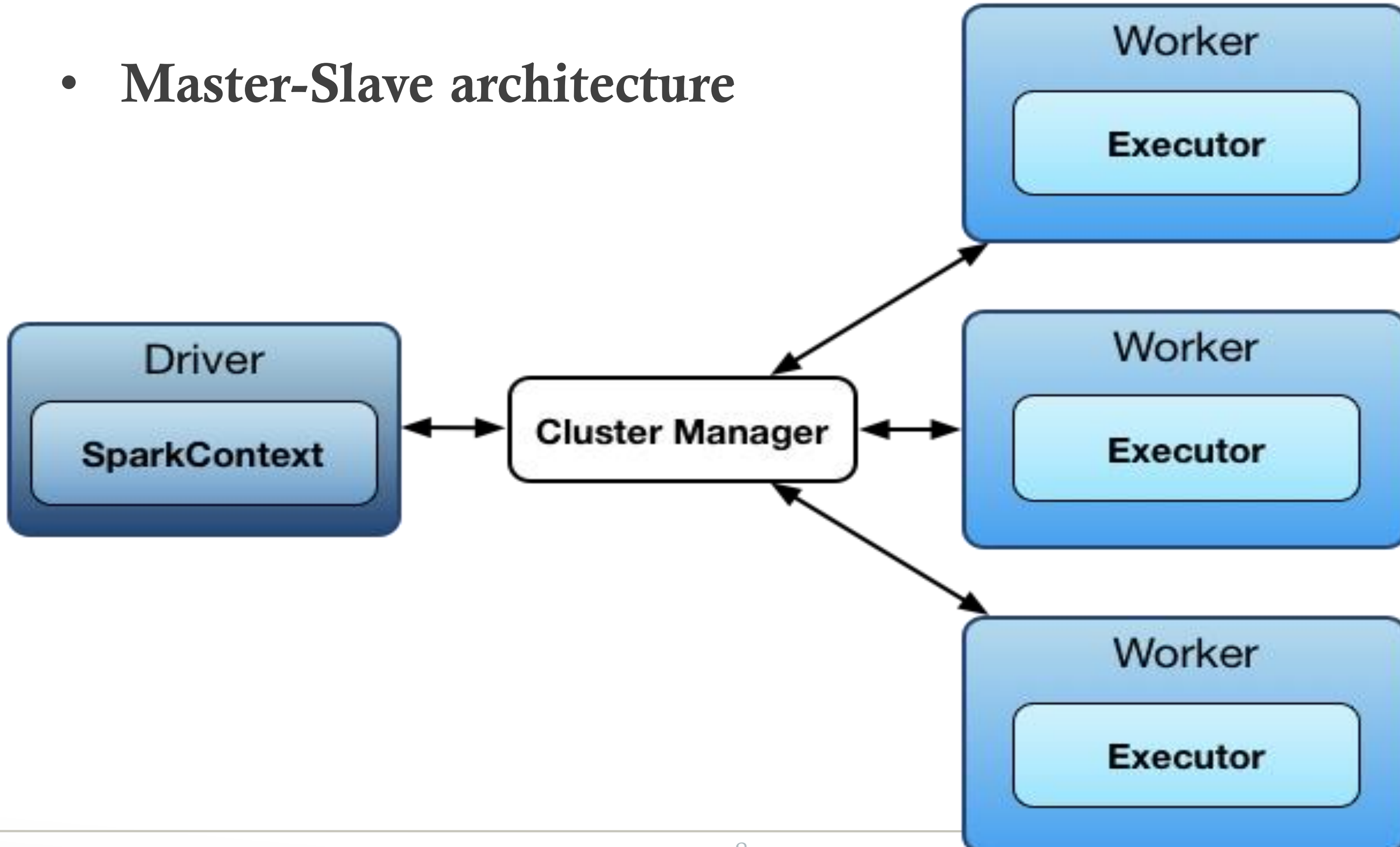
Processing, storage, and data transfer to use RAM, If possible.

Motivation: Overall

- Better support for real-time processing
- Exploit RAM as much as possible
- Large-scale distributed computations

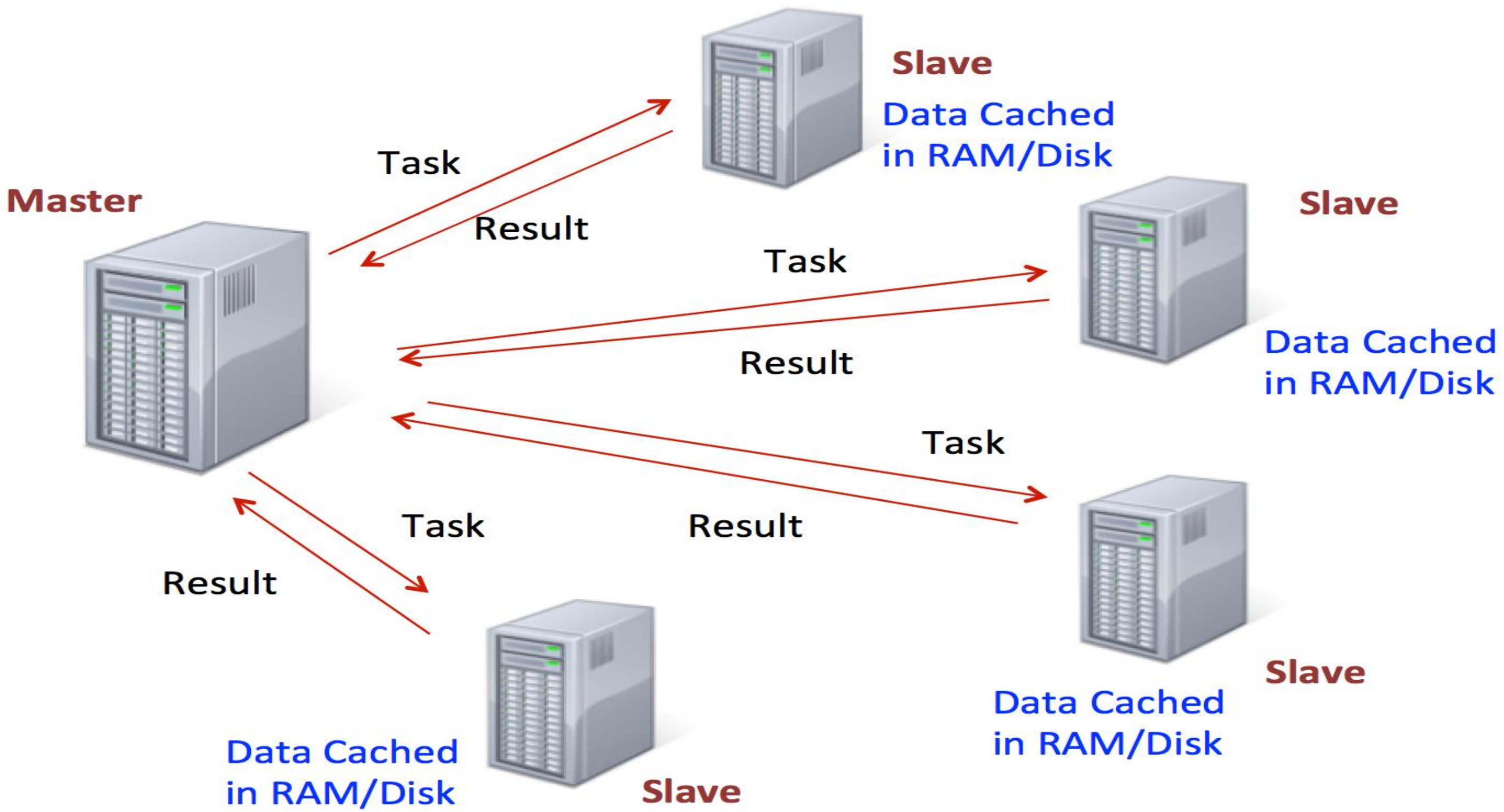
Spark Architecture

- **Master-Slave architecture**



Spark Communication Model

How does Spark execute a job



Spark Programming Model

- High-level coding to build a workflow: **SCALA** as functional prog. language
- Code compiles to distributed parallel operations
- **Two Abstraction Units:**
  - **RDDs: Resilient Distributed Datasets**
  - **Paradigm: Parallel Operations**

Scala

- General purpose programming language (type-safe)
- Combines Object-Oriented and Functional programming
- Features: Concise, logical, and powerful language.
- Compiles to Java bytecode
- Runs on JVM

Comparison

Java	Scala
Complex syntax	Simple syntax
Requires lengthy code.	Shorter code for same purpose.
Rewriting is needed.	Rewriting is not required.
Dynamic in nature	Statically-typed
No assurance of bug-free code	Assurance of lesser defects

Spark RDDs

RDD: Concept

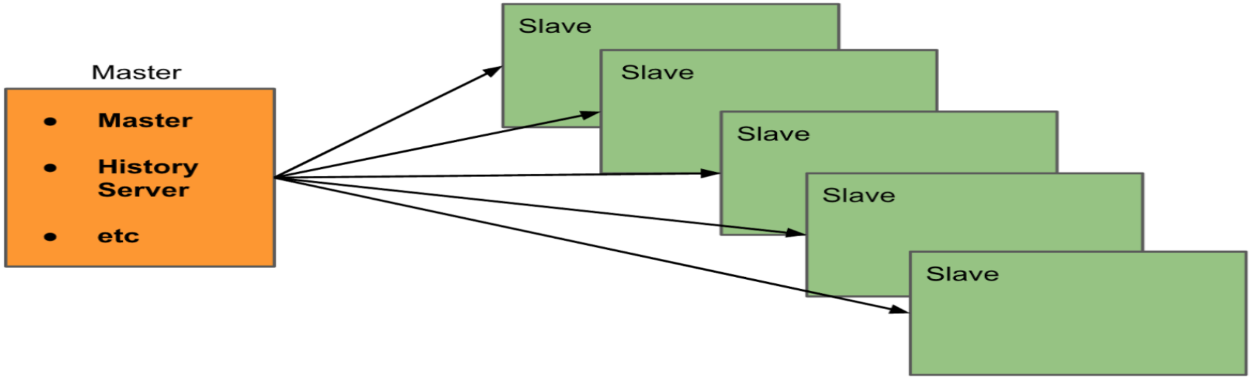
Resilient Distributed Datasets :

- “Collection of objects” (records) that act as one unit
- Stored in main memory (or when needed on disk)
- Parallel operations on top of this “collection”
- Have fault tolerance without replication (lineage)

RDD: Concept

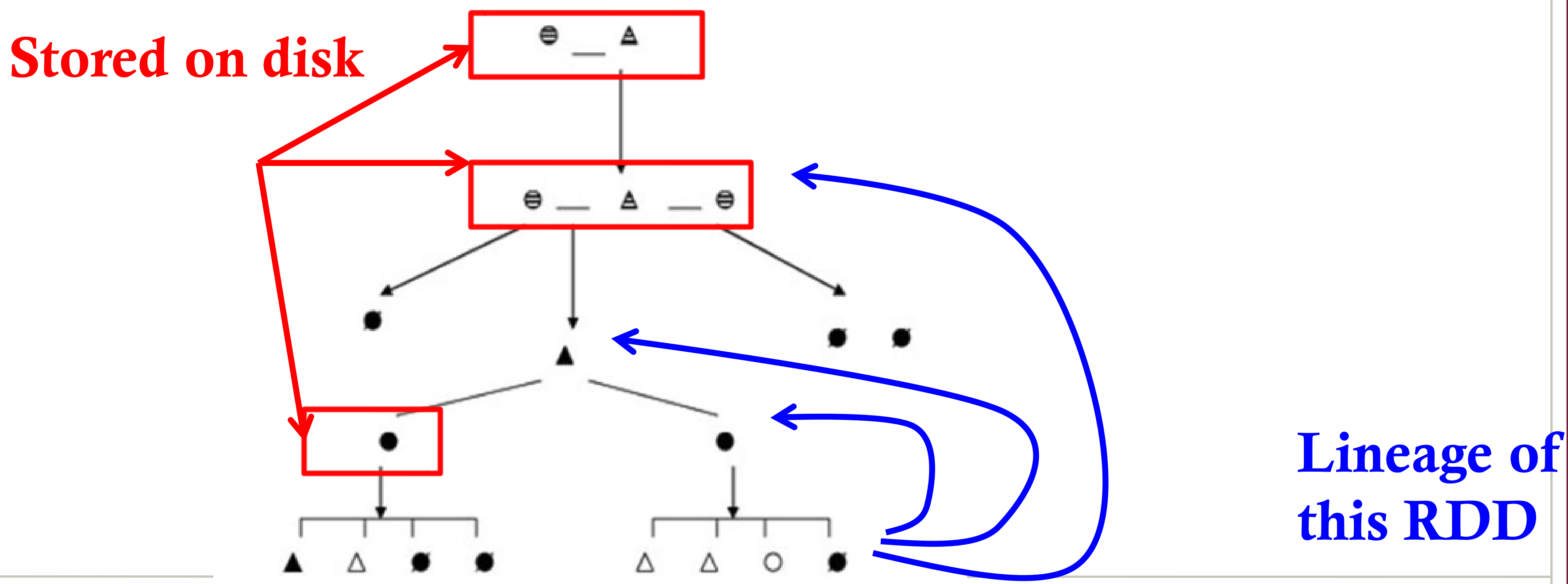
- **RDD is read-only**

- **Distributed either in main memory or disk (automatically decided)**



RDD: Fault Tolerance

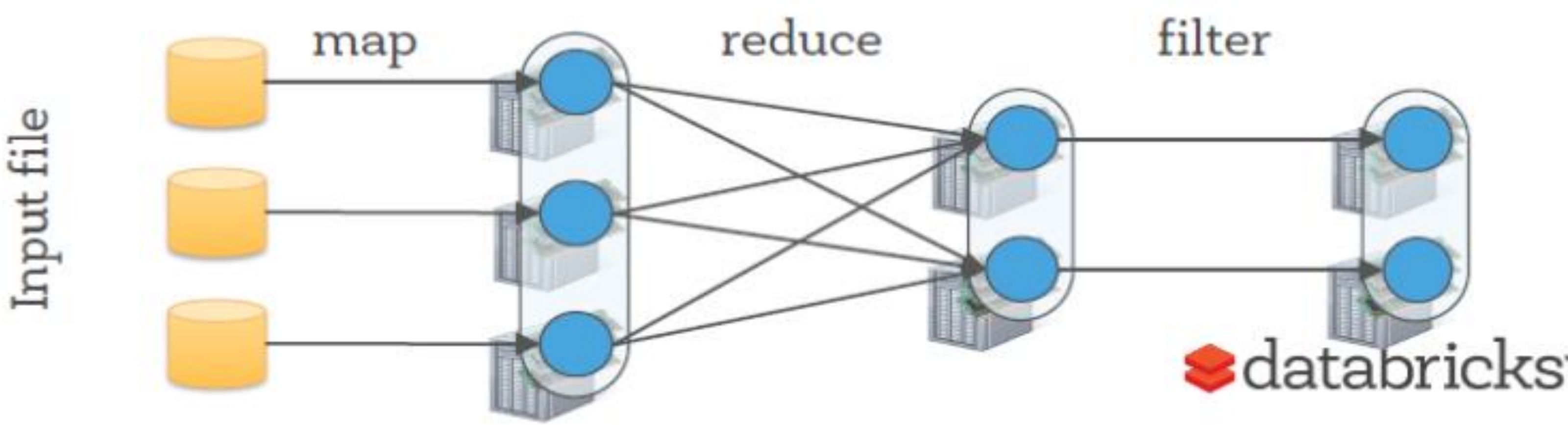
- Not (fully) replicated. But maintain **lineage** (provenance) on how to re-create RDD starting from data in **reliable storage**



RDD: Fault Tolerance

RDDs track *lineage* info to rebuild lost data

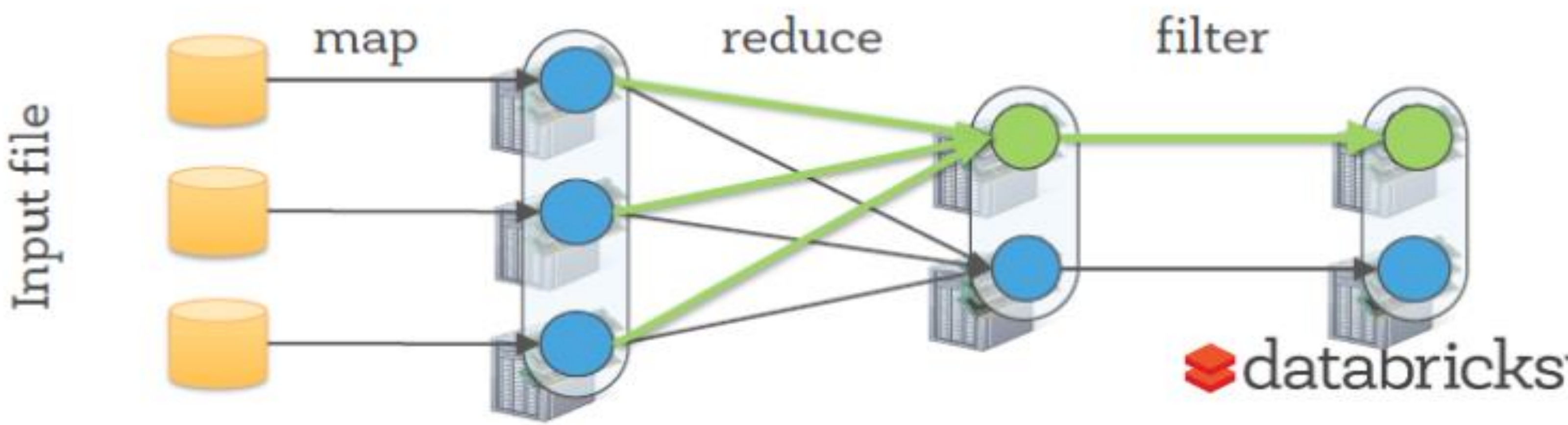
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



RDD: 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)
```



RDD: User Control

Persistence and Partitioning Strategies :

- Indicate which RDDs they will reuse
- Choose a storage strategy for RDD (e.g., hint to keep in-memory storage)
- Request for RDD to be partitioned across machines (i.e., placement optimizations)

RDD: Advantage over MapReduce

- MapReduce:
  - computational power of cluster, but not of distributed main memory
  - **hence, time consuming and slow**

- RDDs in contrast support:
  - in-memory storage
  - in-memory transfer of data

RDD vs. Traditional Shared Memory

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Creating RDDs

- Loading from external dataset (file)
- Creating from another RDD (transformation)
- Parallelizing a centralized collection

Creating RDDs

1. Loading an external dataset

- Most common method for creating RDDs
- Data can be located in any storage system like HDFS, Hbase , Cassandra etc.
- Example:

```
lines = spark.textFile("hdfs://...")
```

Support for HDFS, HBase, Amazon S3, …

RDD: #partitions = #of HDFS blocks

Creating RDDs

2. Creating an RDD from an Existing RDD

- An existing RDD can be used to create a new RDD.
- The Parent RDD remains intact and is not modified.
- The parent RDD can be used for further operations.
- Example

```
errors = lines.filter(_.startsWith("ERROR"))
```

New RDD

Creating RDDs

3. Parallelizing a Central Collection

```
val data = Array(1, 2, 3, 4, 5, 100, 8, 7, ….)

val distData = sc.parallelize(data)
```

SparkContext

```
val data = Array(1, 2, 3, 4, 5, 100, 8, 7, ….)

val distData = sc.parallelize(data, 10)
```

Create 10 partitions

Operations on RDDs

Create new RDD

Return value to caller

No execution is triggered for these operations

Execution is triggered for these operations

- **Transformation Ops.** & **Action Ops.**

Similar to map-side of Hadoop

Similar to reduce-side of Hadoop

Transformation Ops

Transformation Ops

- Operate on one RDD and generate new RDD
- The input RDD is left intact
- Lazy evaluation

- Examples: ***map***, ***filter***, ***join***

Transformation Ops: Example I

Transformations

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))
```

- Original parent RDD is left intact and can be used in future transformations.
- No action takes place, just metadata of `errors` RDD are created.

Transformation Ops: Example II

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
```

Up to Spark to keep it in memory
OR re-compute when needed

```
lineLengths.persist()
```

Ask Spark to keep this RDD in memory

lineLengths.unpersist()

Ask Spark to remove from cache

Action Ops

Action Ops

- Perform a computation on existing RDDs producing a result.
- Result is either:
  - Returned to the Driver Program.
  - Stored in a files system (like HDFS).
- Examples:
  - *count()*
  - *collect()*
  - *reduce()*
  - *save()*

Action Ops: Example I

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

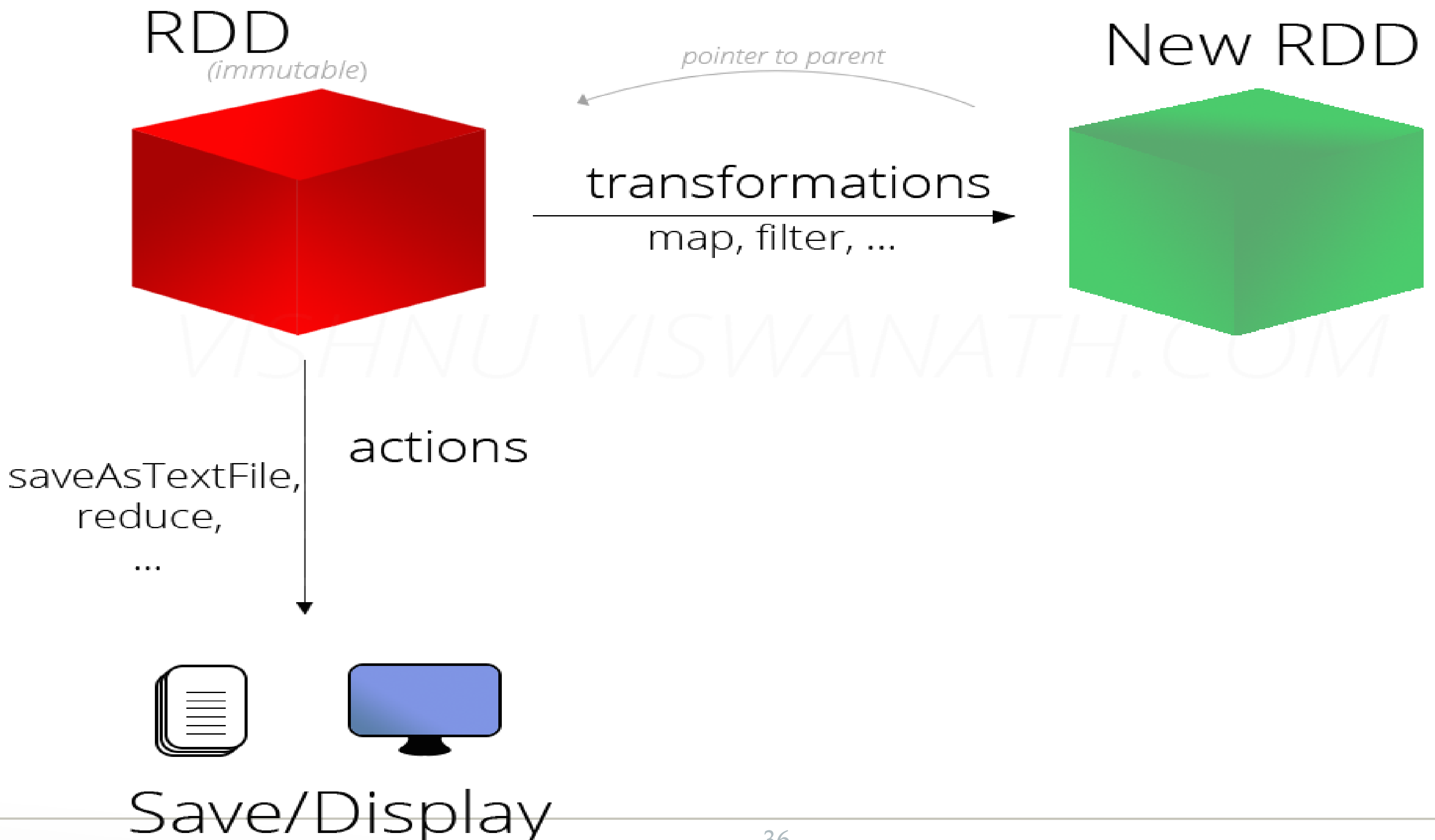
Note: You can apply Reduce op on any type that has + fct.

Action Ops: Example II [Link]

```
val logFile = "hdfs://master.backtobazics.com:9000/user/root/sample.txt"
val lineRDD = sc.textFile(logFile)
//Transformation 1 -> DAG created
//{DAG: Start -> [sc.textFile(logFile)]}

val wordRDD = lineRDD.flatMap(_.split(" "))
//Transformation 2 -> wordRDD DAG updated
//{DAG: Start -> [sc.textFile(logFile)]
//               -> [lineRDD.flatMap(_.split(" "))]}

val filteredWordRDD = wordRDD.filter(_.equalsIgnoreCase("the"))
//Transformation 3 -> filteredWordRDD DAG updated
//{DAG: Start -> [sc.textFile(logFile)]
//               -> [lineRDD.flatMap(_.split(" "))]
//                 -> [wordRDD.filter(_.equalsIgnoreCase("the"))]}


filteredWordRDD.collect
//Action: collect
//Execute DAG & collect result to driver node
```

```scala
val logFile = "hdfs://master.backtobazics.com:9000/user/root/sample.txt"
val lineRDD = sc.textFile(logFile)
//Transformation 1 -> DAG created
//{DAG: Start -> [sc.textFile(logFile)]}

val wordRDD = lineRDD.flatMap(_.split(" "))
//Transformation 2 -> wordRDD DAG updated
//{DAG: Start -> [sc.textFile(logFile)]
//                -> [lineRDD.flatMap(_.split(" "))]}

val filteredWordRDD = wordRDD.filter(_.equalsIgnoreCase("the"))
//Transformation 3 -> filteredWordRDD DAG updated
//{DAG: Start -> [sc.textFile(logFile)]
//                -> [lineRDD.flatMap(_.split(" "))]
//                  -> [wordRDD.filter(_.equalsIgnoreCase("the"))]}


filteredWordRDD.collect
//Action: collect
//Execute DAG & collect result to driver node
```

Transformations vs. Actions



Lazy Evaluation

- Transformation ops follow lazy evaluation
- Results not physically computed right away
- Metadata regarding transformations recorded
- Transformations are implemented only when an action is invoked

Example: Lazy Evaluation

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.count()
```

Execution is triggered here

RDD Fault Tolerance

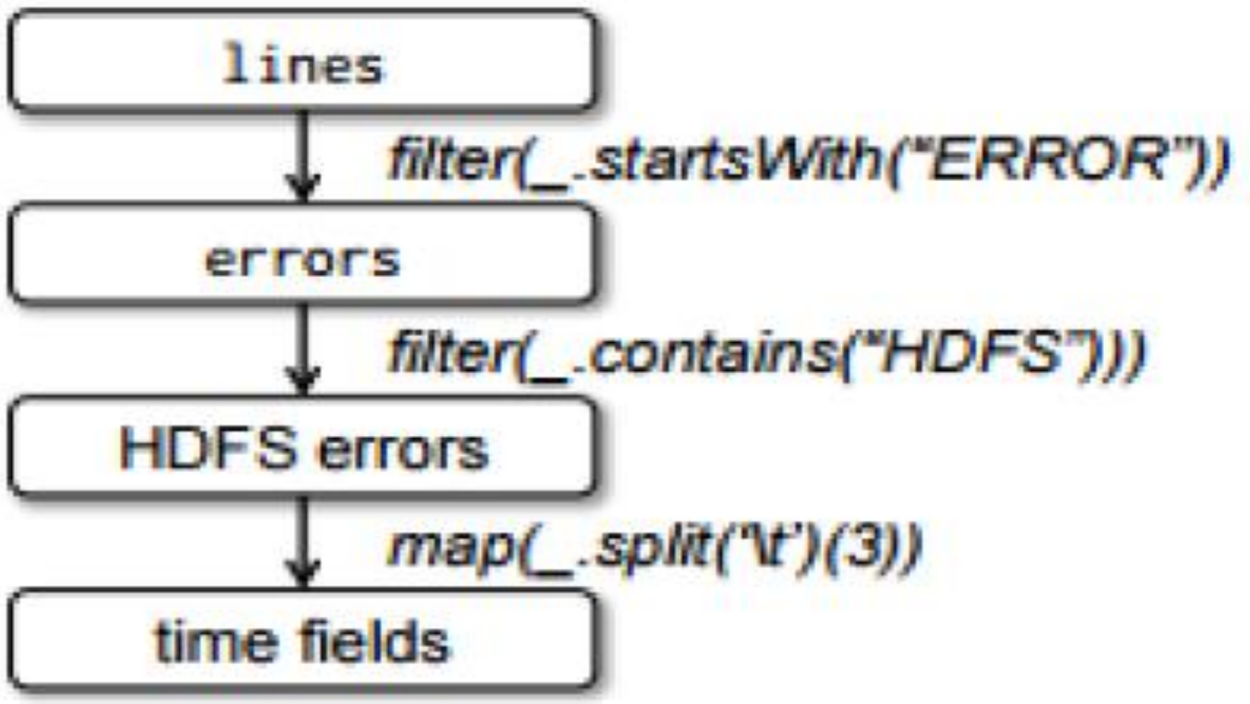
RDD Fault Tolerance

- In-memory RDDs are not replicated
  - RAM is still limited in size (Scarce Resource)

- **Lineage Graph ("Logical Execution Plan")**
  - Directed Acyclic Graph (DAG) produced when Sparkcontext requested to run spark job
  - Maintain dependencies between RDDs
  - Go back to closest disk-based RDD

Lineage Graph

- Not storing the data, but instead stores how it is generated (the processing steps)



Lineage Graph

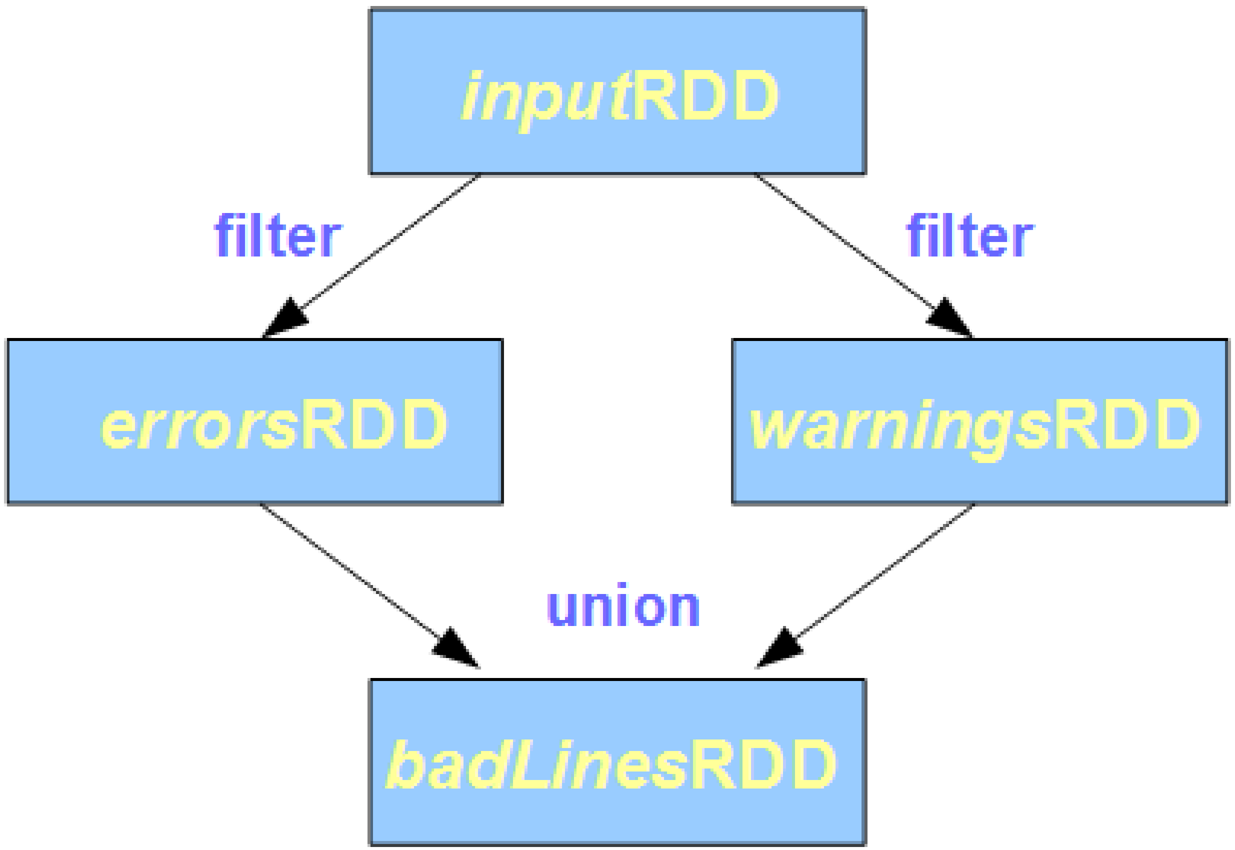


Figure 3-1. RDD lineage graph created during log analysis

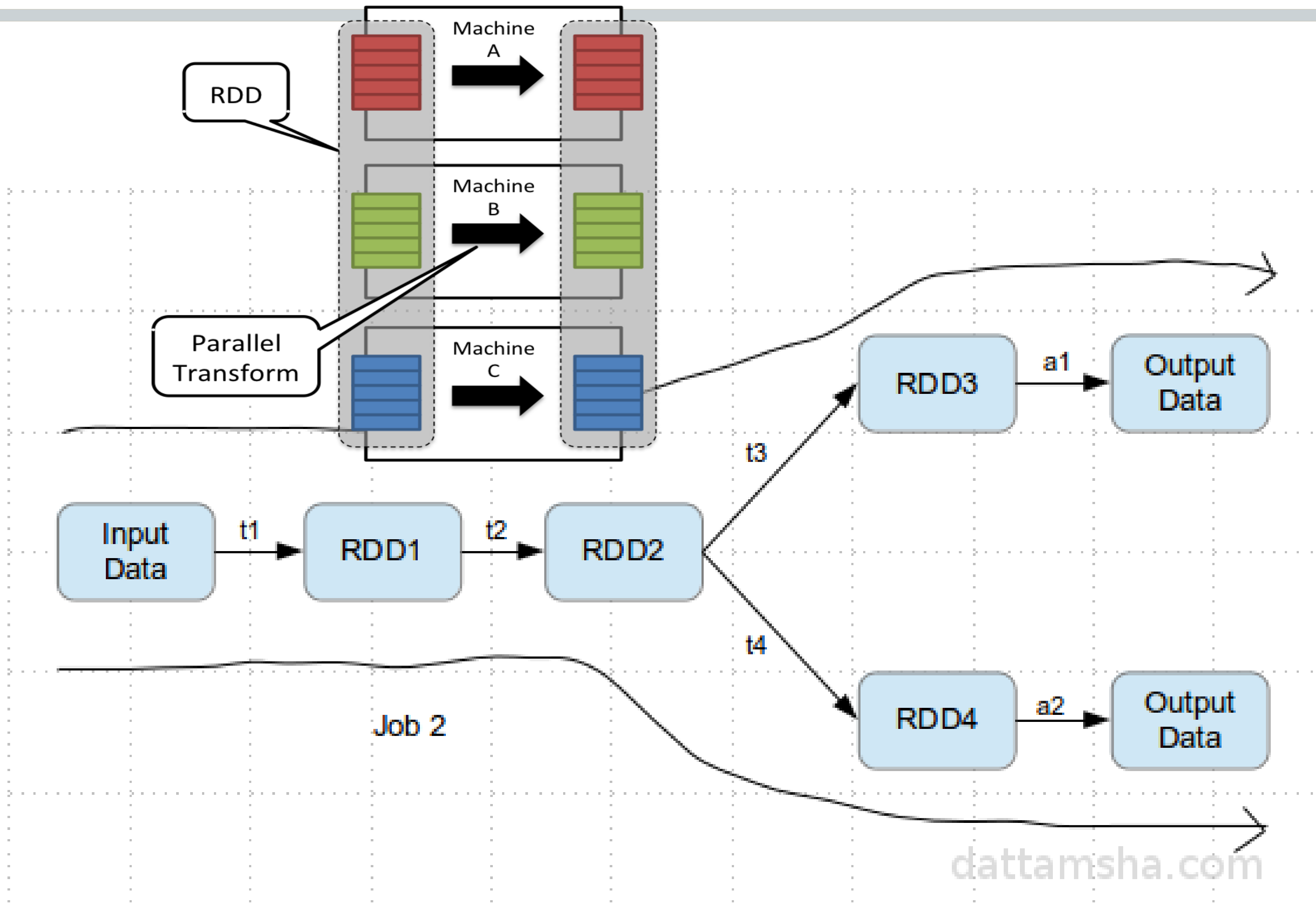
Representation of RDDs

Representation of RDDs

- **Each RDD is divided into:**
  - Multiple partitions
  - Dependencies on parent RDD(s)

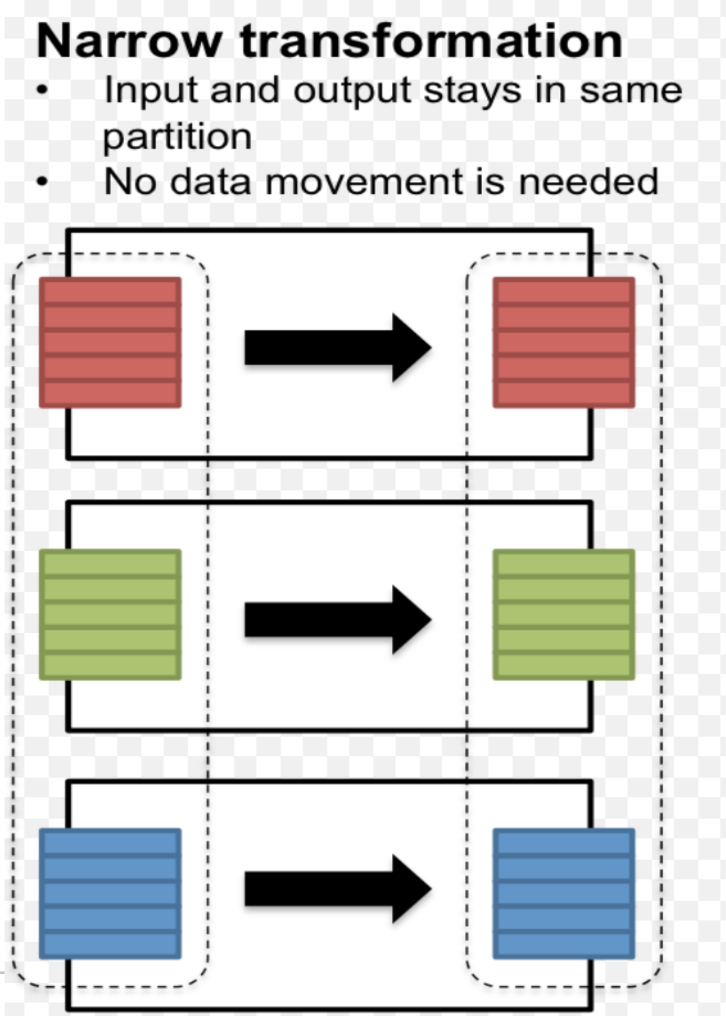
- **Two types of dependencies**
  - Narrow (1 to 1)
  - Wide (many to many)

Representation of RDDs



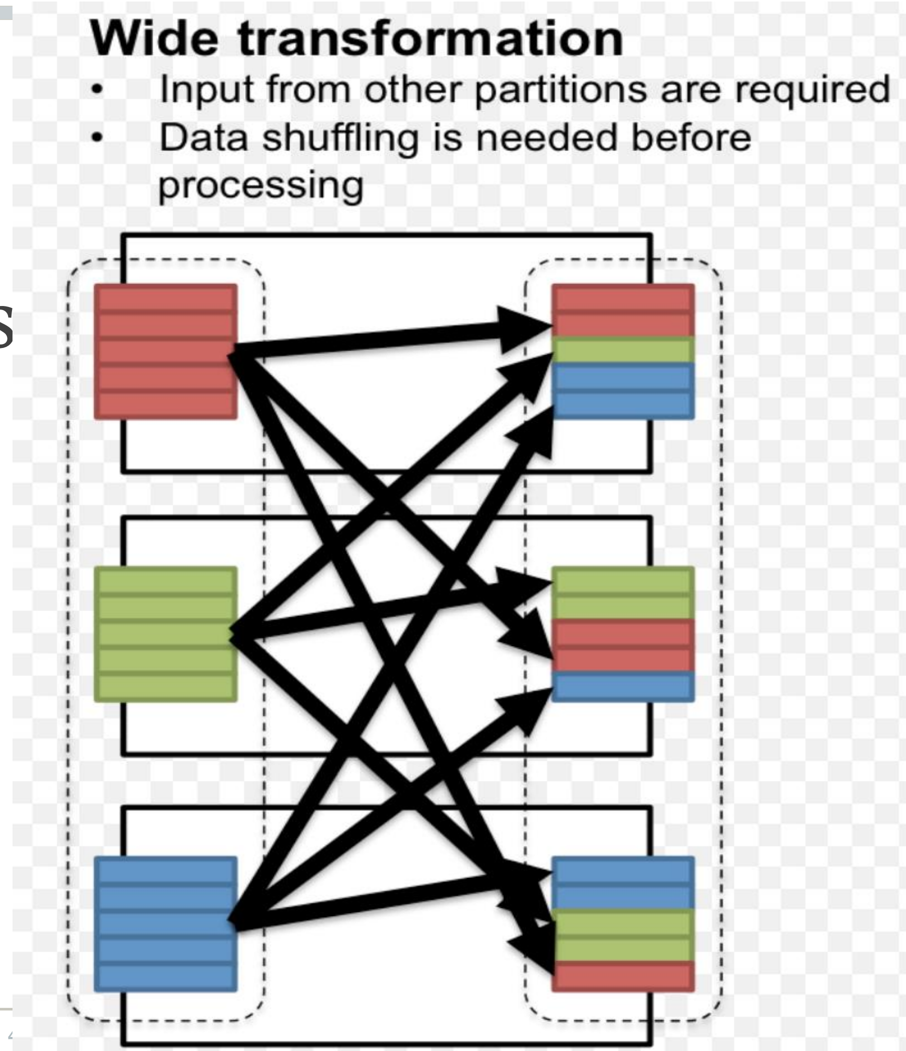
Narrow Dependency

- 1-1 relationship between child-parent partitions
- Example Ops: ***Filter*** & ***Map***
- Relatively cheap process



Wide Dependency

- M-1 or M-M relationship between child-parent partitions
- Example Ops: ***Join*** & ***Grouping***
- More expensive



Advanced Interfaces on RDDs

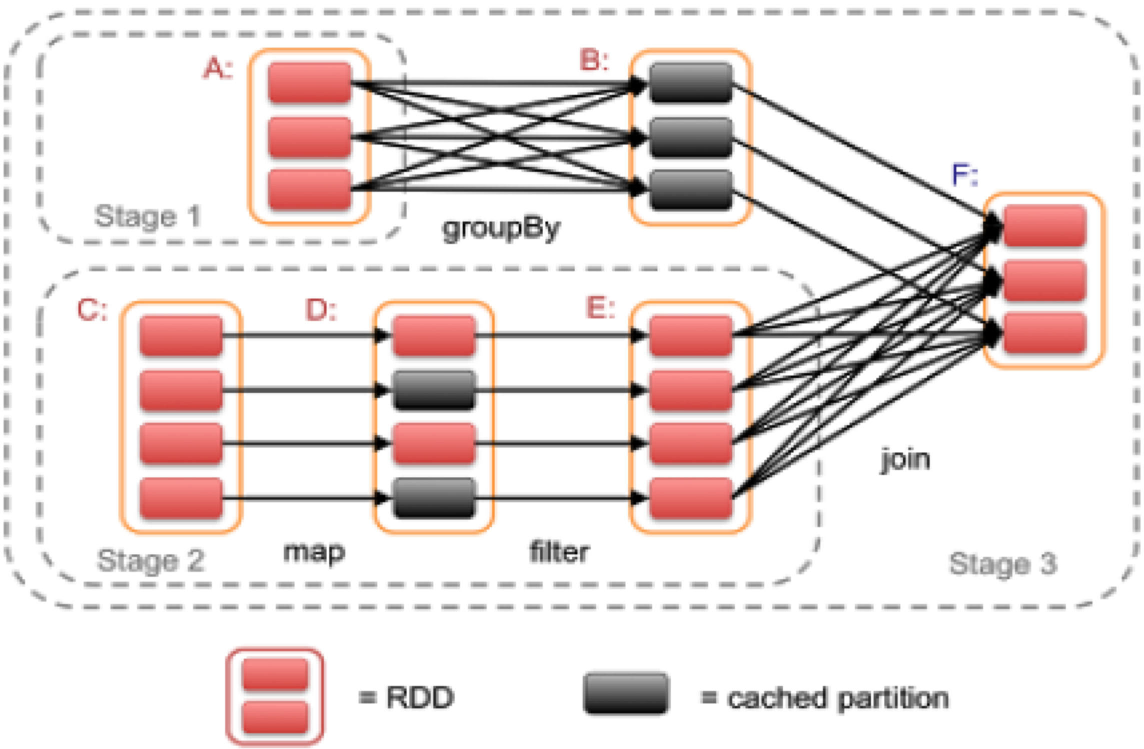
Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent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

Interface used to represent RDD in Spark

Scheduling & Memory Management

Scheduling

- Execution is triggered when an "***Action***" op is invoke
- Scheduler checks the lineage graph to execute



Spark Memory Management

<u>Memory utilization is essential in Spark (Caching):</u>

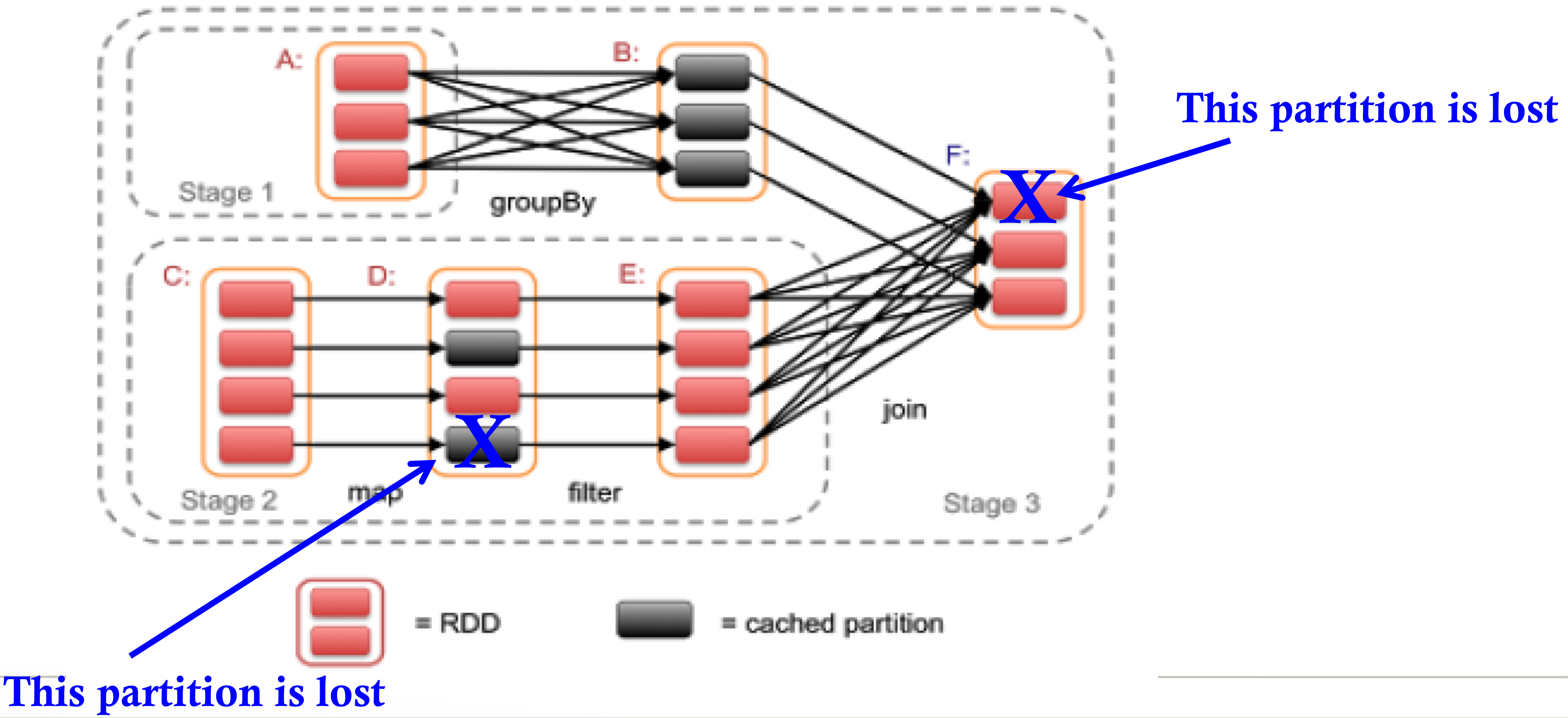
- 1. in-memory storage as (deserialized) java objects
  - - fast
- 2. in-memory storage as serialized data
  - - memory efficient, but not as fast
- 3. on-disk storage (if RDD too large to fit in RAM)
  - - slower and costly

Replacement Policy

- LRU eviction policy at level of RDD partitions is used
- **When a new RDD partition is created**
  - If there is space in memory ➔ Cache it
  - If not ➔ evict one or more partitions from LRU RDD
- Use "*persistence priority*" to prevent eviction of important RDDs

RDD Recovery

- In case of failure and losing an RDD partition



RDD Recovery

- Recovery can be time consuming for RDDs **with long lineage chains**
- Use ***Checkpoint Mechanism*** to make some RDDs persistent:
  - User defined, OR
  - System controlled, OR
  - More intelligent ways, e.g., workload driven

RDD Summary

- Memory Cache RDD is **fault-tolerant:**
  - using sophisticated fine-grained Lineage Mechanism
- Disk Copy RDD is **fault-tolerant:**
  - using HDFS replication

- Aware User: Can manually call **unpersist()**

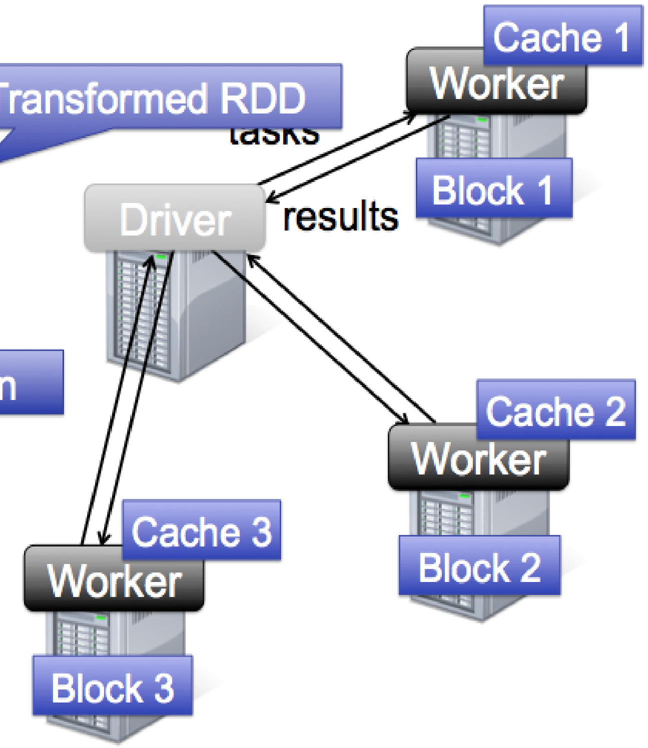
Summary

Example I

Example: Mining Console Logs

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split('\t')[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```

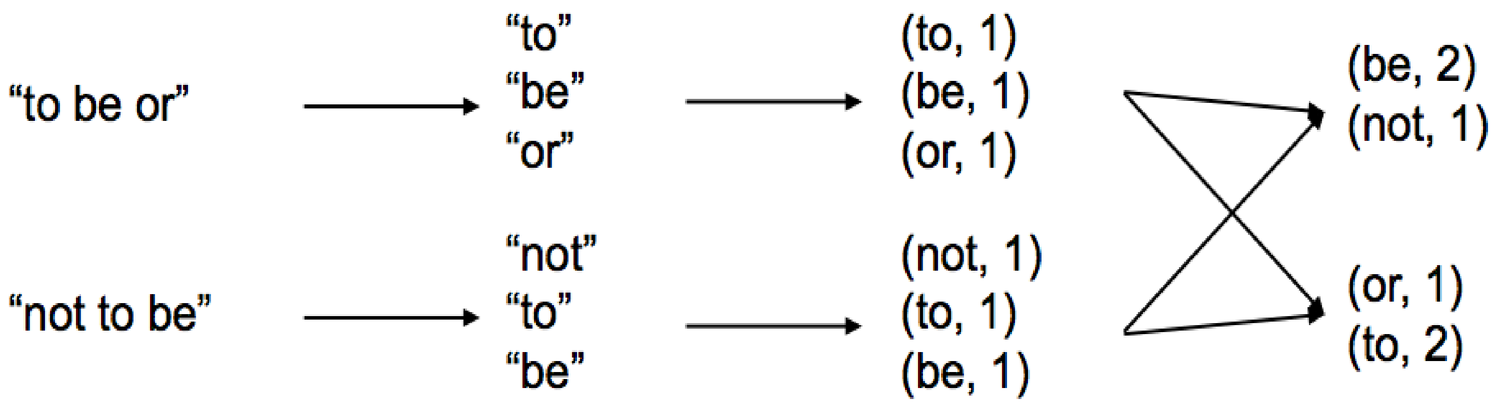


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Example II

Example: Word Count

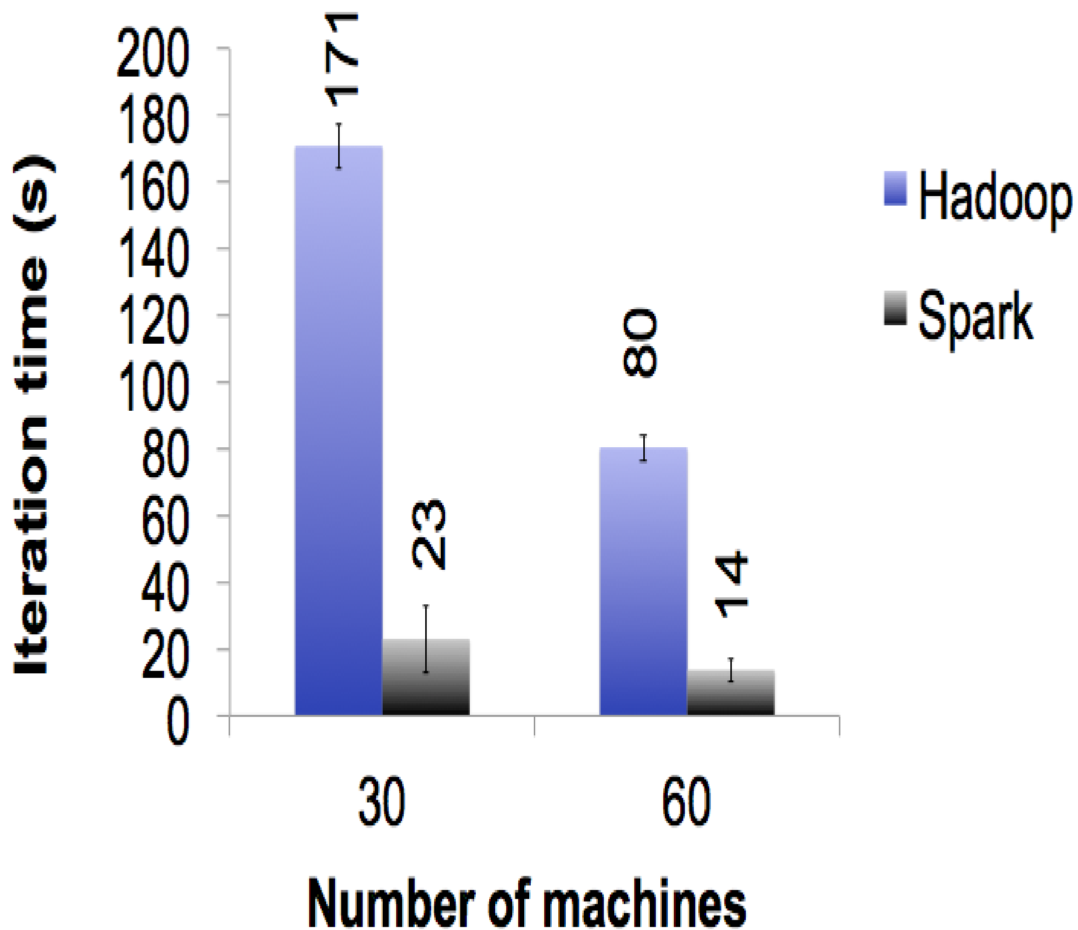
```
lines = sc.textFile("hamlet.txt")

counts = lines.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda x, y: x + y)
```



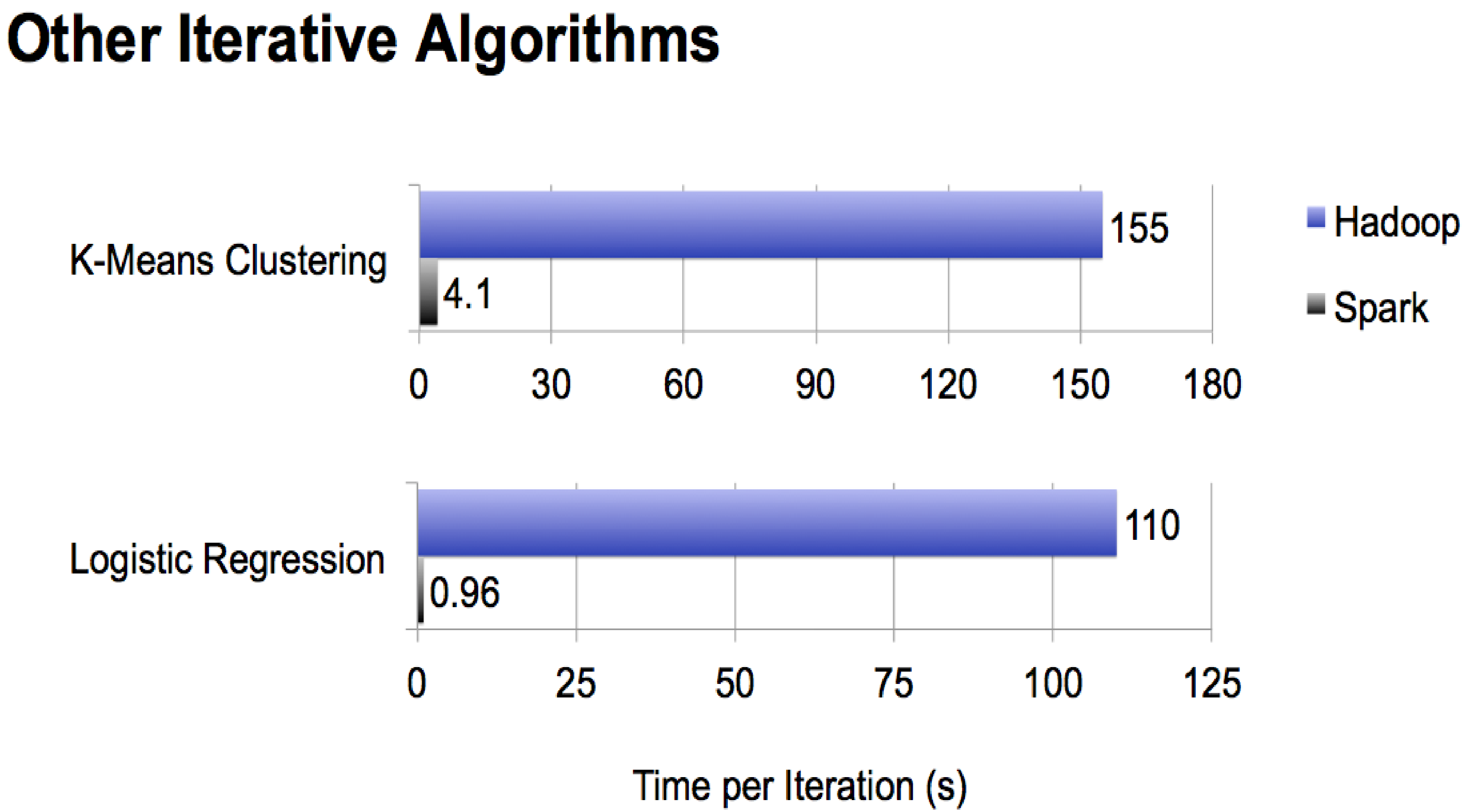
Performance

- Spark is faster than Hadoop 10x – 100x
- Especially for iterative algorithms

PageRank Performance



K-Means & L. Regression



Performance (Revisited)

POSITIVE:

- Spark is faster than Hadoop 10x – 100x
- Especially for iterative algorithms

NEGATIVE:

- Spark needs much more memory than Hadoop
- Spark performance is more sensitive to availability of memory

SAME:

- *Jobs that require one iteration:*
  - No big difference between Spark & Hadoop