Big Data Analytics

Lebanese University

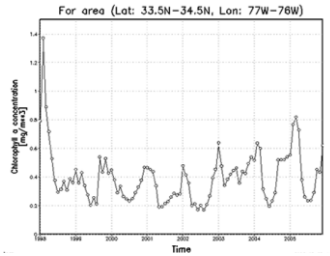
Faculty of Information 1

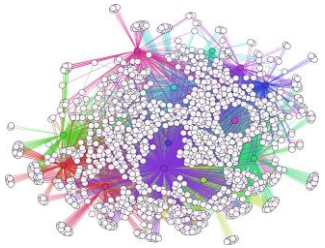
Dr. Hussein Hazimeh

Specialized Systems: Downside

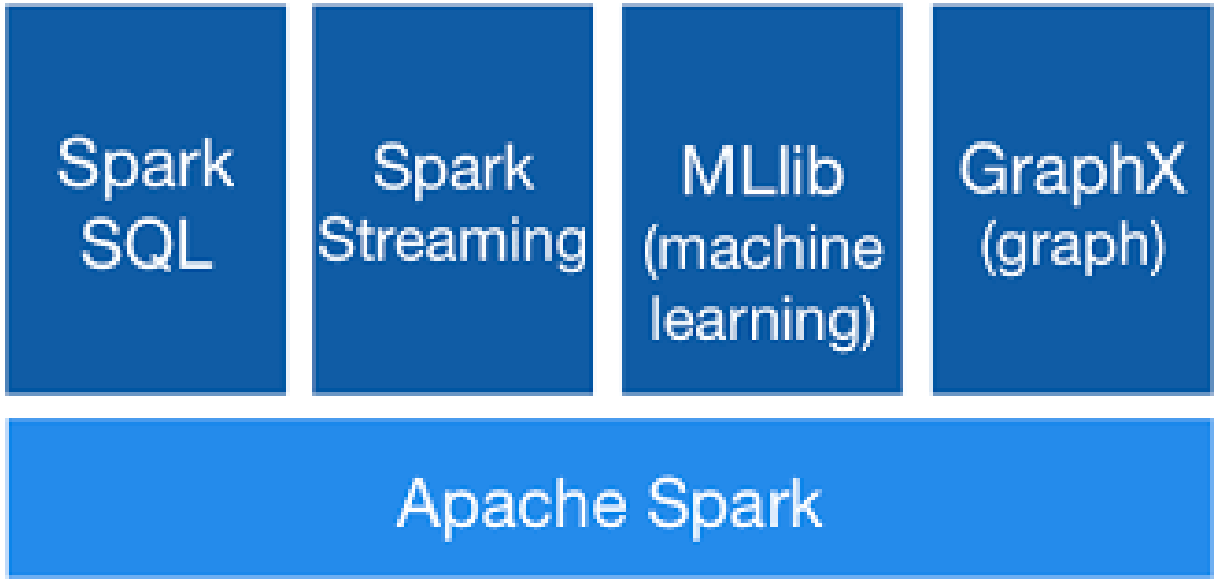
- Problems with Specialized Systems
  - More systems to manage, tune and deploy.
- Can't combine processing types in one application
  - Even though many pipelines need to do this.
  - e.g. load data with SQL, then run machine learning.
- In many pipelines, data exchange between engines is the dominant cost.

One application may need them all



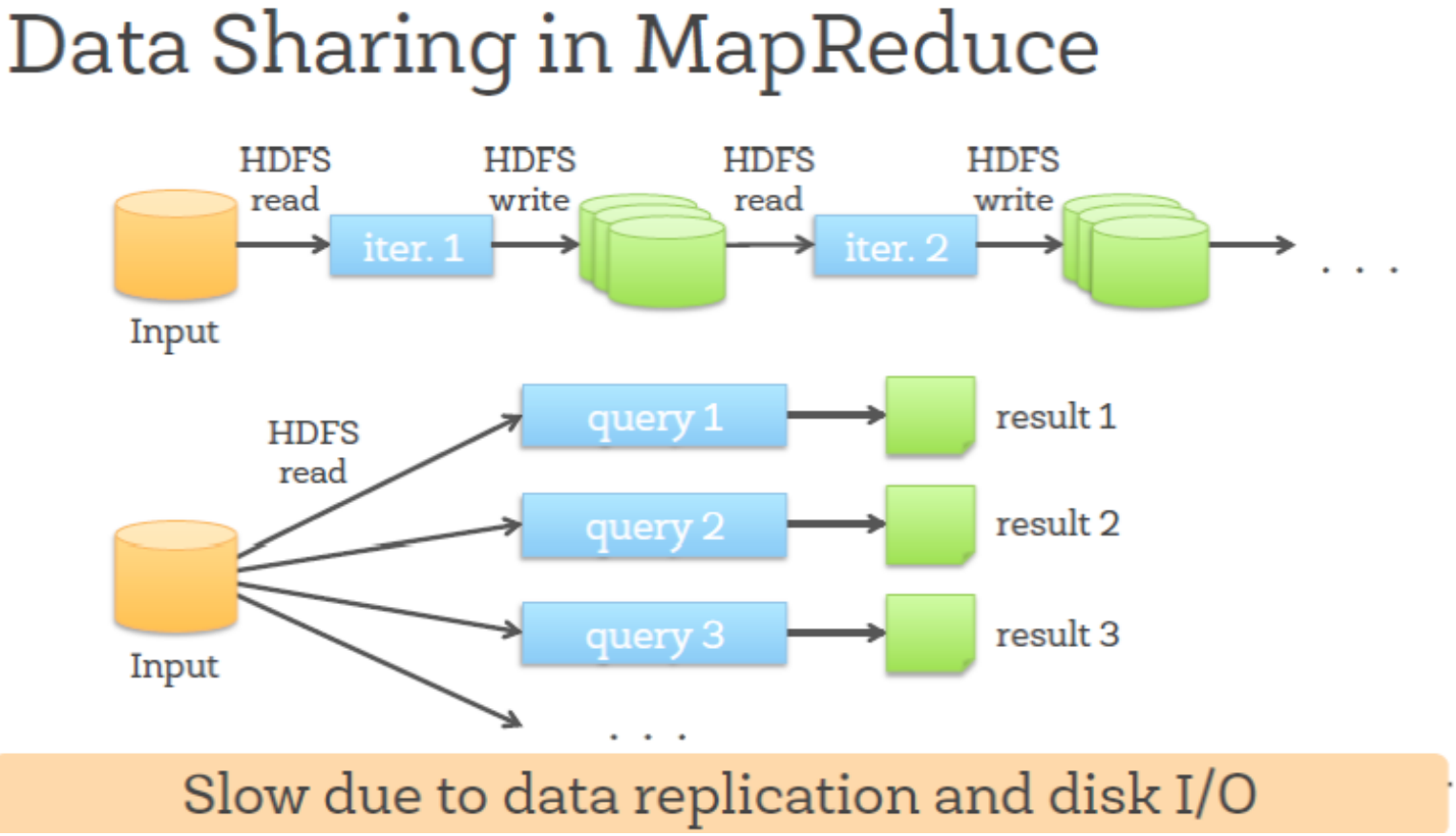


Vision: *Generic Efficient* Infrastructure



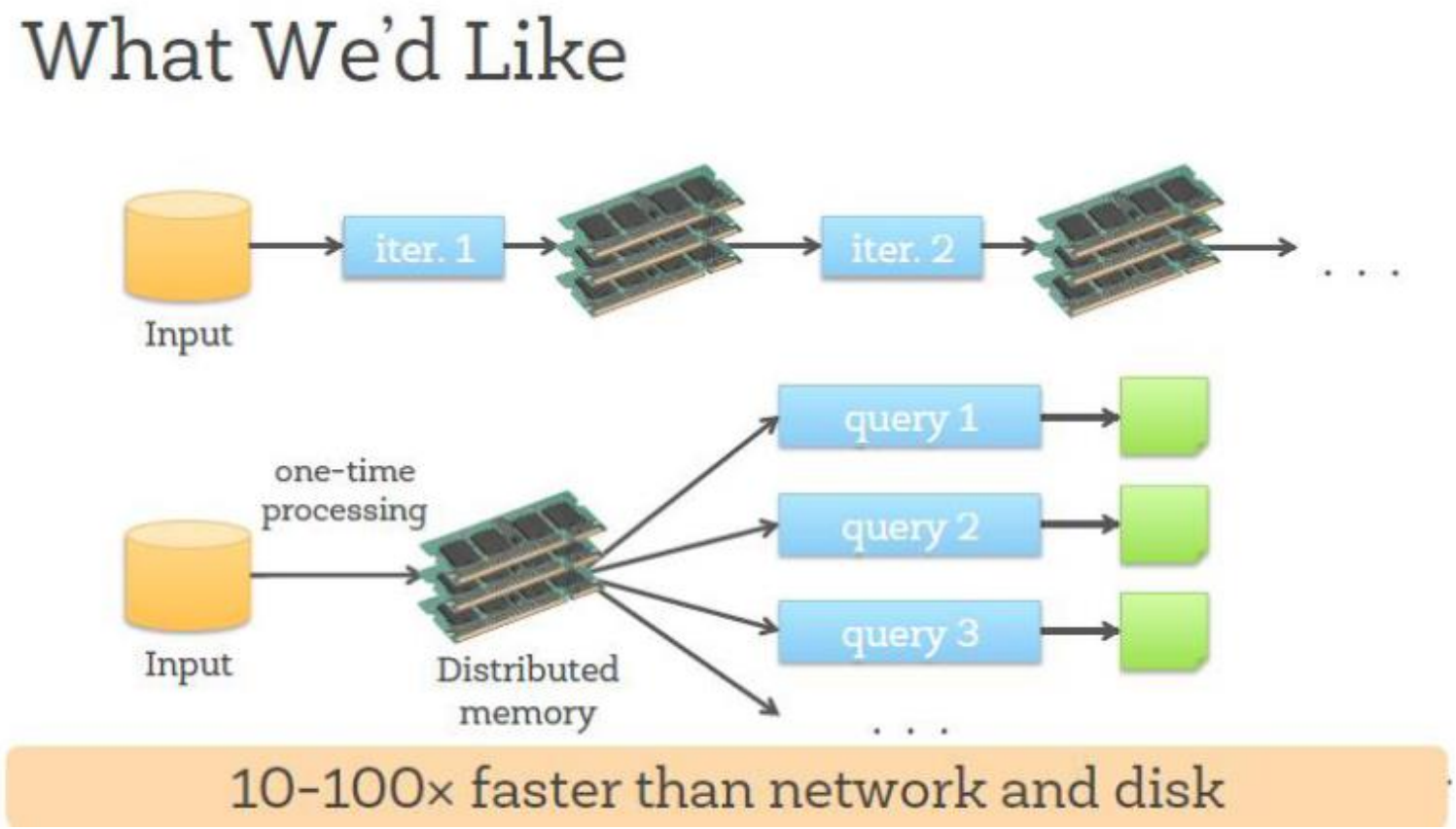
Motivation Workloads

From This …



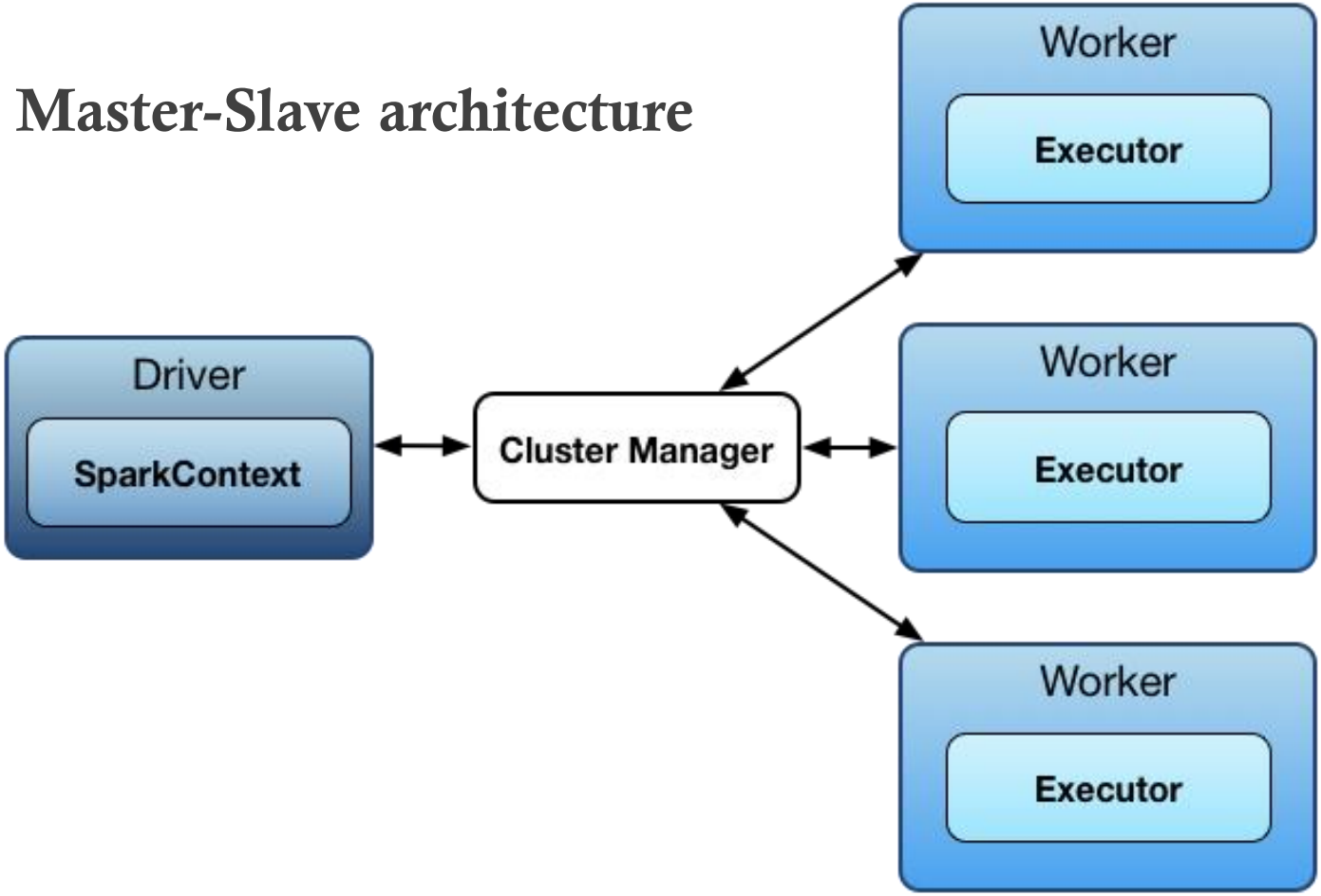
Motivation Workloads

To This …



Spark Architecture

- **Master-Slave architecture**



Spark Programming Model

- High-Level coding to build a workflow (Scala)
- Code compiles to distributed parallel operations
- Two Abstraction Units
  - **RDDs: Resilient Distributed Datasets**
  - **Parallel Operations**

- General purpose programming language
- Combines Object-Oriented and Functional programming
- Compiles to Java bytecode
- Runs on JVM

Spark RDDs

RDD: Concept

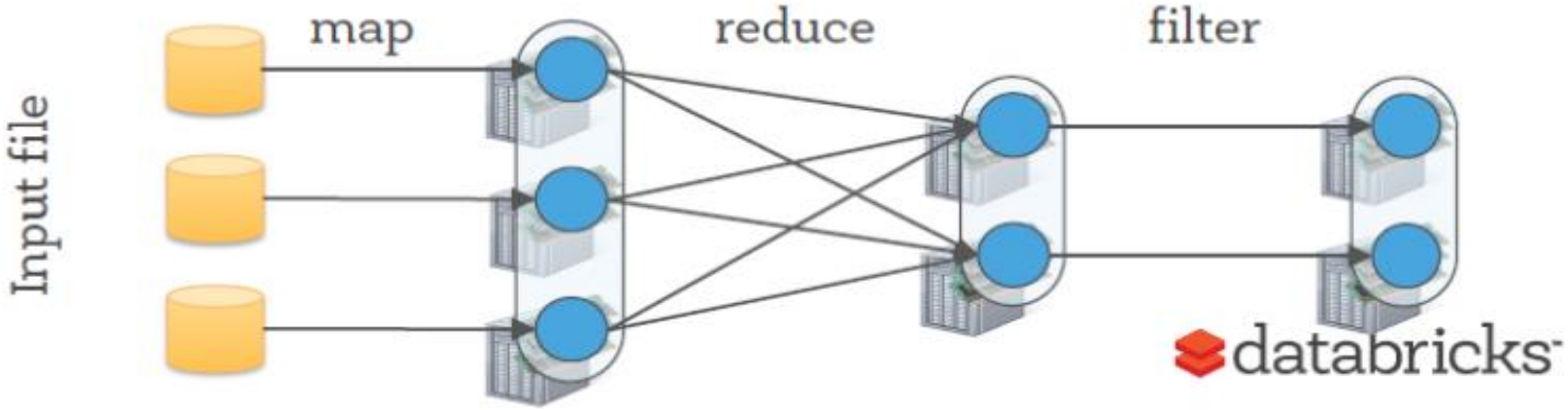
Resilient Distributed Datasets

- Collection of objects (records) that act as one unit
- Stored in main memory or disk
- Parallel operations built on top of them
- Have fault tolerance without replication (lineage)

RDD: Fault Tolerance

RDDs track *lineage* info to rebuild lost data

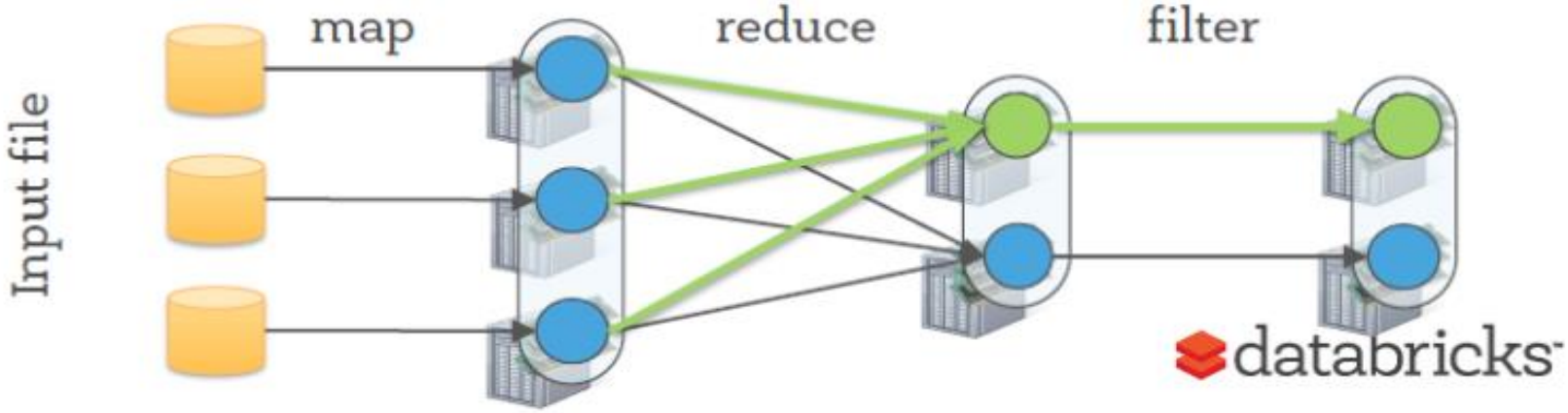
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



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 and choose a storage strategy for them (e.g., in-memory storage)
- Ask that an RDD be partitioned across machines this is useful for placement optimizations.

RDD: Advantage

- MapReduce access the computational power of the cluster, but not distributed memory
  - Time consuming and slow

- RDDs allow in-memory storage and 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

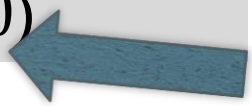
- 3. Parallelizing centralized collection

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

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

```
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

No execution is triggered for these Ops.

Return value to caller

Execution is triggered for these Ops.

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

Similar to map-side of Hadoop

Similar to reduce-side of Hadoop

Transformation Ops

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

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)
```

Action Ops: Example II [Link]

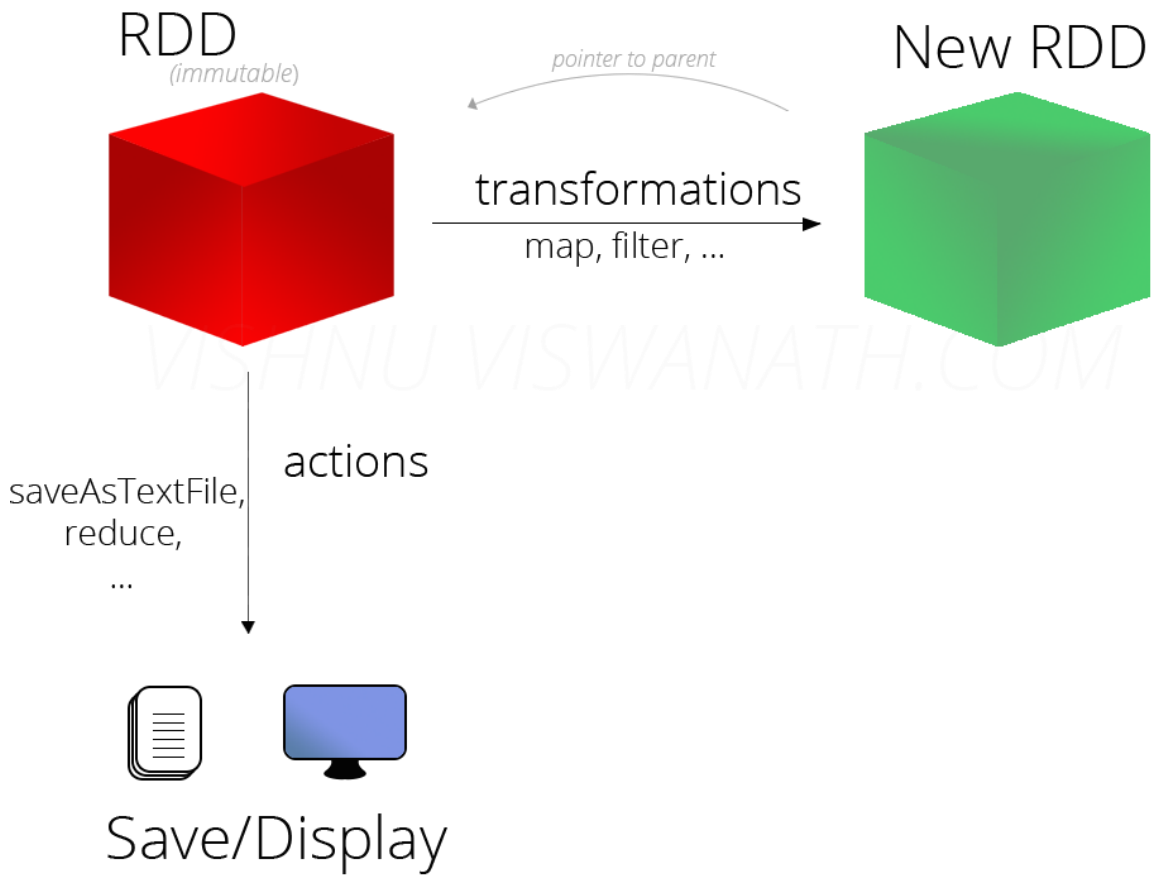
```
val logFile = "hdfs://master.backtobazics.com:9000/user/root/sample.txt"
val lineRDD = sc.textFile(logFile)
//Transformation 1 -> DAG created
//{DAG: Start -> [sc.textFile(logFile)]}

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
```

Transformations vs. Actions



Lazy Evaluation

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

Example

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

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

errors.count()
```

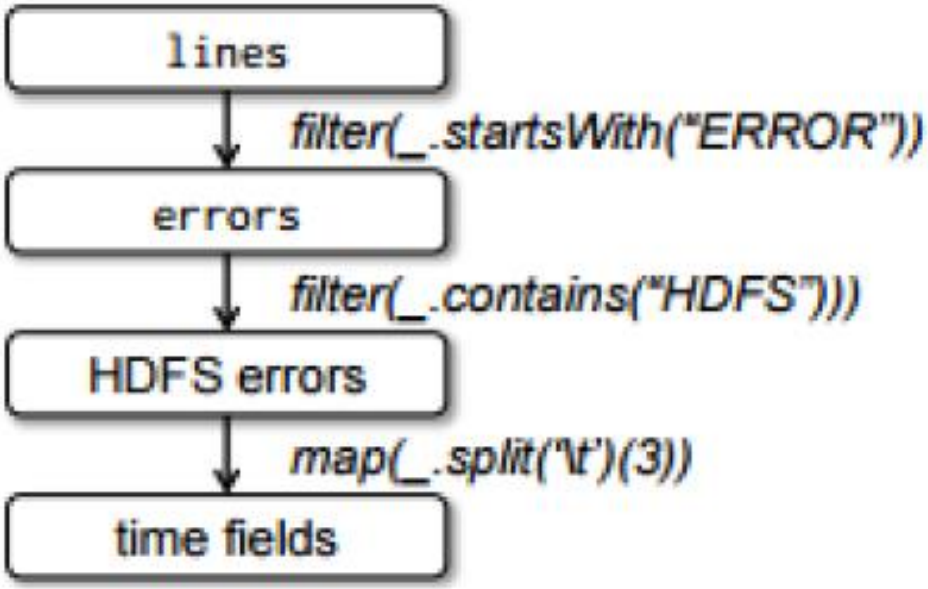
Execution is triggered here

RDD Fault Tolerance

- In-memory RDDs are not replicated
  - RAM is still limited in size (**Scarce Resource**)
- **Lineage Graph**
  - Directed Acyclic Graph (DAG)
- Maintain dependencies between RDDs
- Go back to the closest disk-based RDD

Lineage Graph

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



Lineage Graph

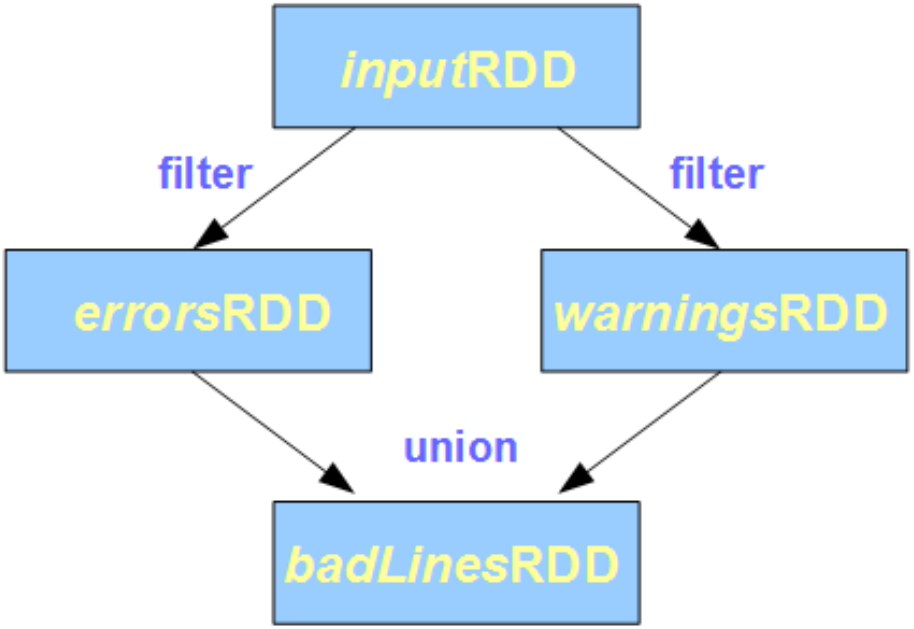


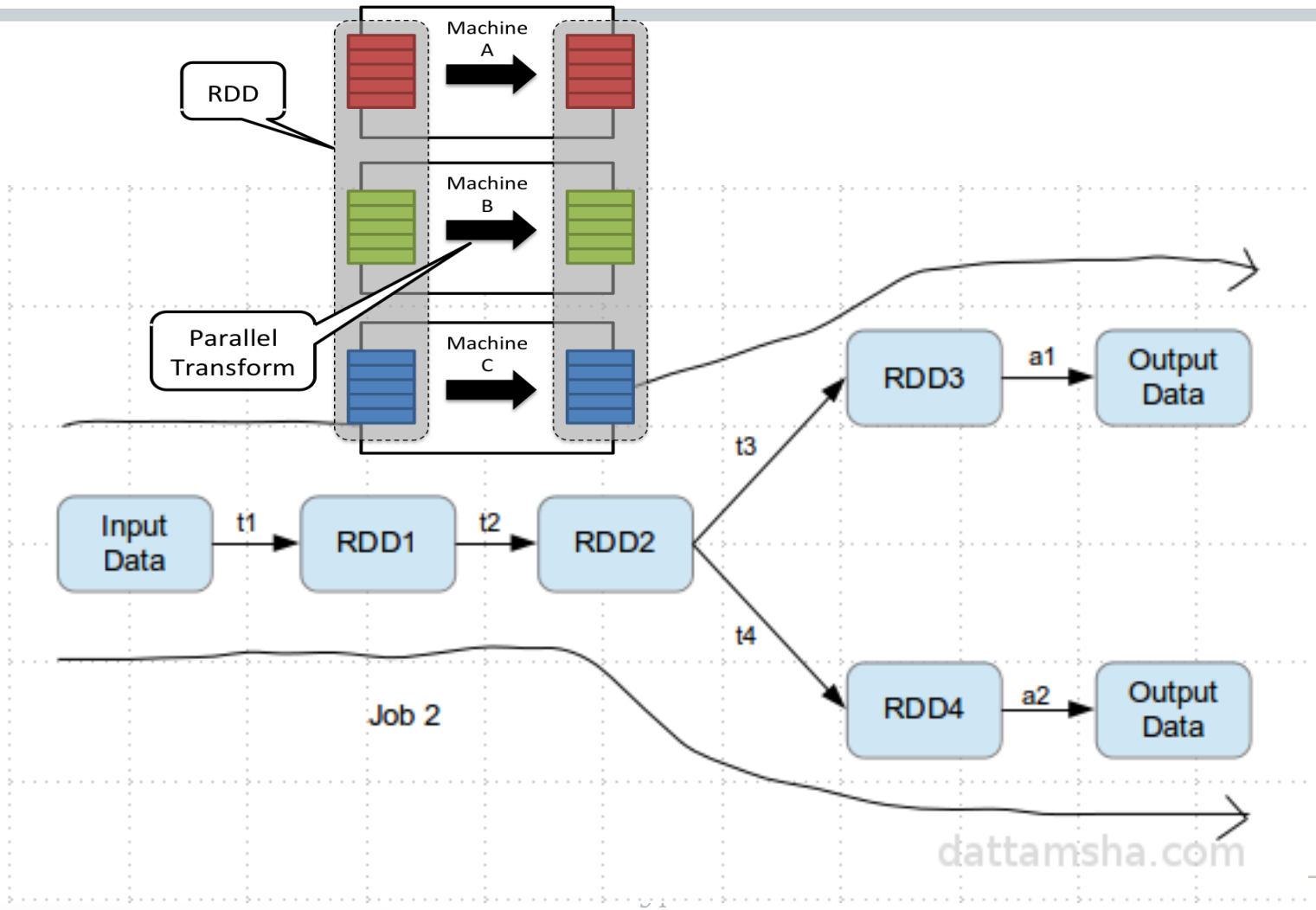
Figure 3-1. RDD lineage graph created during log analysis

Representation of RDDs

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

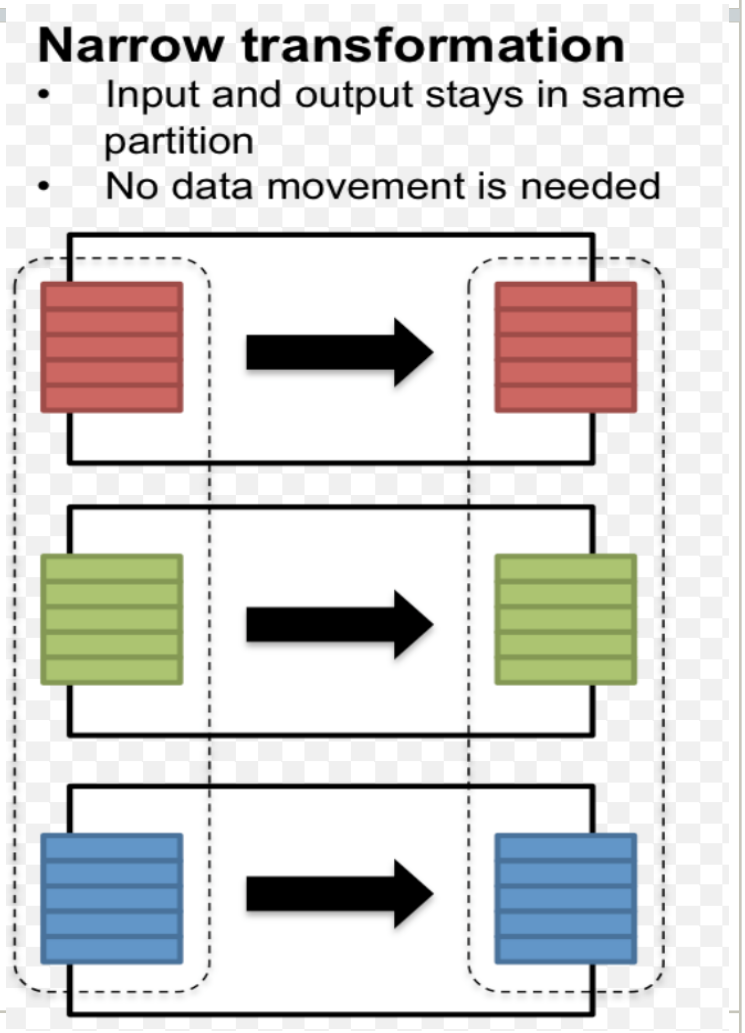
- **Two types of Dependencies**
  - Narrow
  - Wide

Representation of RDDs



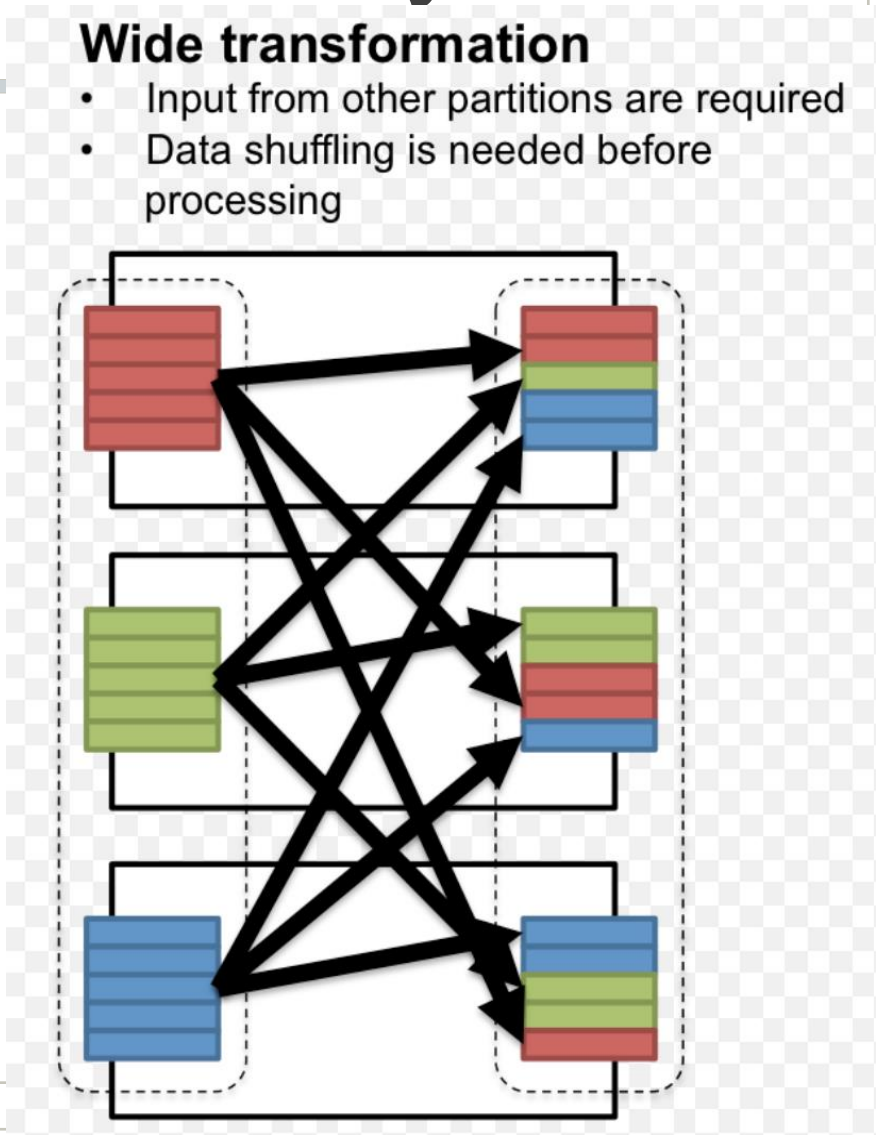
Narrow Dependency

- 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



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 invoked
- Scheduler checks the lineage graph to execute