# Analytics and AI on Cray Systems

## Session III

# Spark on Cray Systems

## Session III

James Maltby, Cray Inc.

# What we will cover….

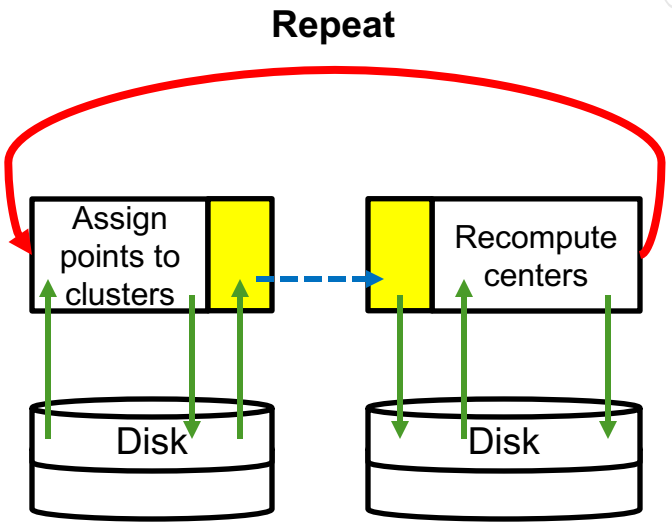
- **Introduction to Spark**
  - History and Background
  - Computation and Communication Model
- **Spark on the XC40**
  - Installation and Configuration
  - Local storage
- **Alchemist: MPI and Spark**
- **BigDL: Deep Learning in Spark**

# In the beginning, there was Hadoop MapReduce…

- **MapReduce: simplified parallel programming model**
  - All computations broken into two parts
    - **Embarassingly parallel map phase**: apply single operation to every key,value-pair, produce new set of key,value-pairs
    - **Combining reduce phase**: Group all values with identical key, performing combining operation to get final value for key
  - Can perform multiple iterations for computations that require
  - I/O intensive
    - Map writes to local storage. Data shuffled to reducer's local storage, reduce reads.
    - Additional I/O between iterations in multi-iteration algorithms (map reads from HDFS, reduce writes to HDFS)
  - Effective model for many data analytics tasks
- **HDFS distributed file system (locality aware – move compute to data)**
- **YARN cluster resource manager**

# Example: K-Means Clustering with MapReduce

- **Initially: Write out random cluster centers**
- **Map:**
  - Read in cluster centers
  - For each data point, compute nearest cluster center and write <key: nearest cluster, value: data point>
- **Reduce:**
  - For each cluster center (key) compute average of datapoints
  - Write out this value as new cluster center
- **Repeat until convergence (clusters don't change)**



# MapReduce Problems

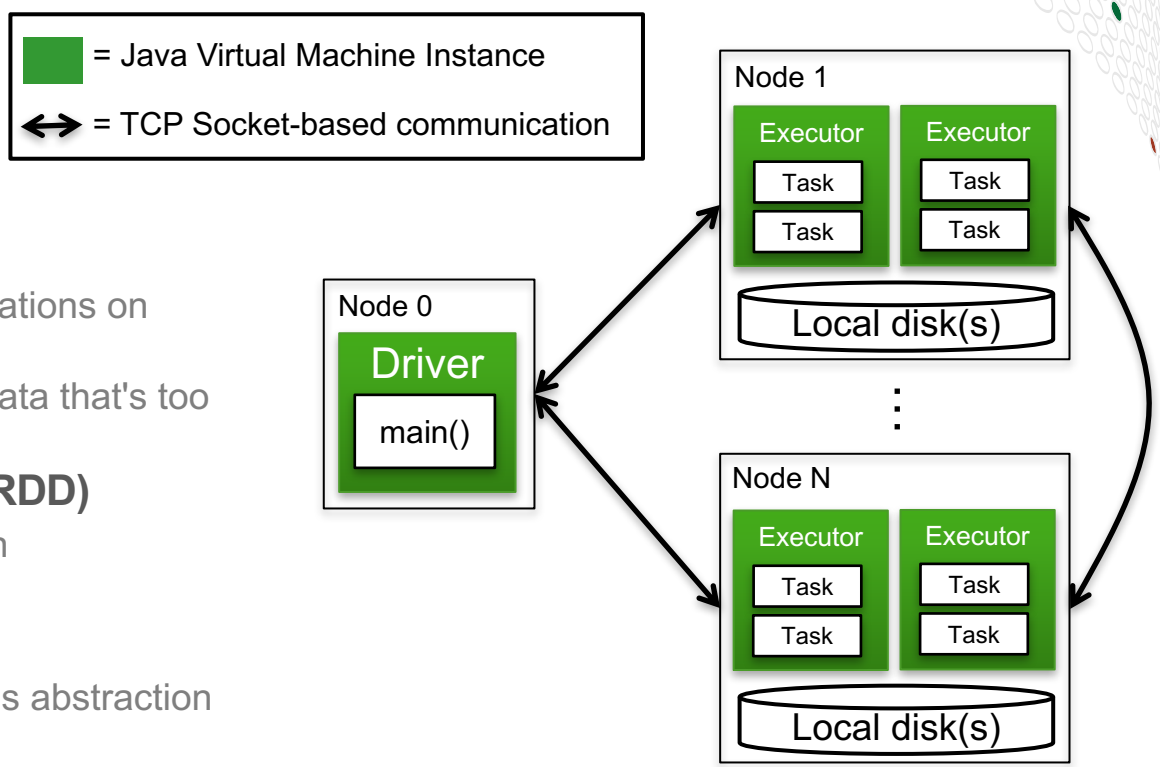
- **Gated on IO bandwidth, possibly interconnect as well**
  - Must write and read between map and reduce phases
  - Multiple iterations must write results in next time (e.g., new cluster centers)
- **No ability to persist reused data**
- **Must re-factor all computations as map then reduce (rinse and repeat?)**

# What is Spark?

- **Newer (2014) analytics framework**
  - Originally from Berkeley AMPLab/BDAS stack, now Apache project
  - Native APIs in Scala.  Java, Python, and R APIs available as well.
  - Many view as successor to Hadoop MapReduce.  Compatible with much of Hadoop Ecosystem.
- **Aims to address some shortcomings of Hadoop MapReduce**
  - More programming flexibility – not constrained to one map, one reduce, write, repeat.
  - Many operations can be pipelined into a single in-memory task
  - Can "persist" intermediate data rather than regenerating every stage

# Spark Execution Model

- **Master-slave parallelism**
- **Driver (master)**
  - Executes main
  - Distributes work to executors
- **Executors (slaves)**
  - Lazily execute tasks (local operations on partitions of the RDD)
  - Rely on local disks for spilling data that's too large, and storing shuffle data
- **Resilient Distributed Dataset (RDD)**
  - Spark's original data abstraction
  - Partitioned amongst executors
  - Fault-tolerant via lineage
  - Dataframes/Datasets extend this abstraction



# RDDs (and DataFrames/DataSets)

- **RDDs are original data abstraction of Spark**
  - DataFrames add structure to RDDs: named columns
  - DataSets add strong typing to columns of DataFrames (Scala and Java only)
  - Both build on the basic idea of RDDs
    - DataFrames were originally called SchemaRDDs
- **RDD data structure contains a description of the data, partitioning, and computation, but not the actual data … why?**
  - Lazy evaluation

# Lazy Evaluation and DAGs

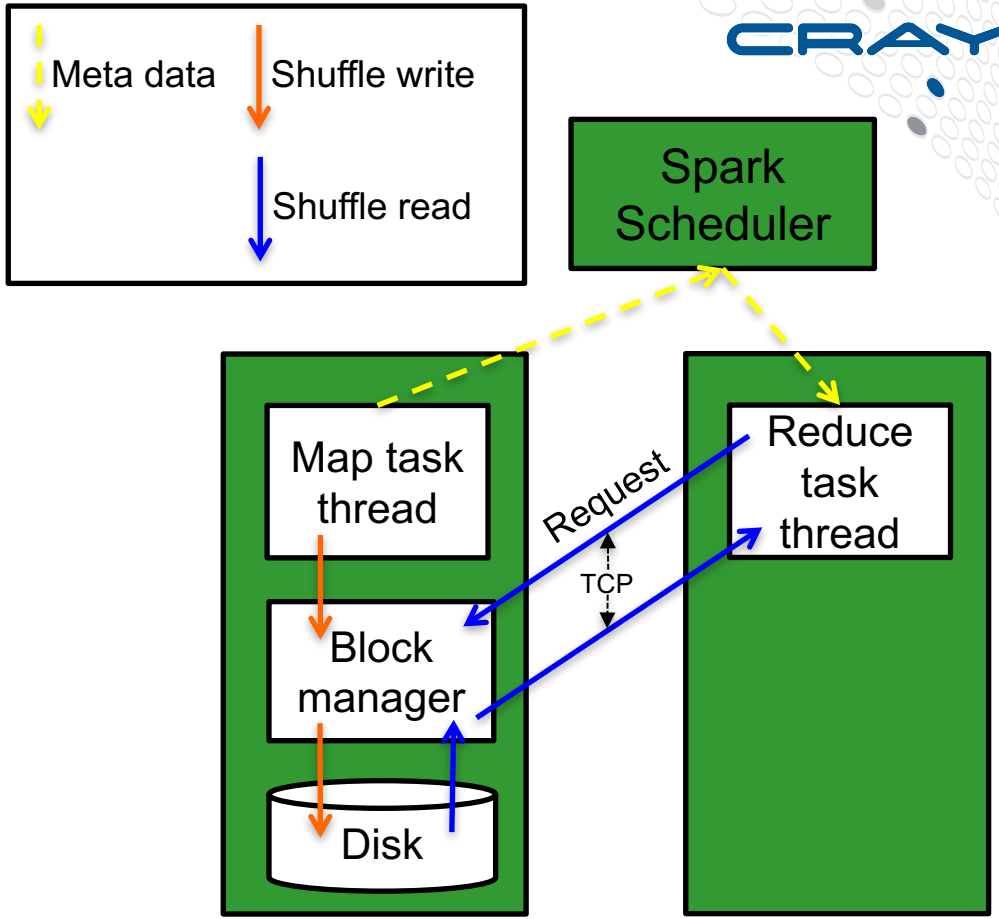
- **Spark is lazily evaluated**
  - Spark operations are only executed when and if needed
  - Needed operations: produce a result for driver, or produce a parent of needed operation (recursive)
- **Spark DAG (Directed Acyclic Graph)**
  - Calls to transformation APIs (operations that produce a new RDD/DataFrame from one or more parents) just add a new node to the DAG, indicating data dependencies (parents) and transformation operation
  - Action APIs (operations that return data) trigger execution of necessary DAG elements
- **Example shortly…**

# Pipelining in Spark

- **If an RDD partition's dependencies are on a single other RDD partition (or on *co-partitioned* data), the operations can be *pipelined* into a single *task***
- **Spark stage: Execution of same task on all partitions**
  - Every stage ends with a *shuffle* (all-to-all communication), an output, or returning data back to the driver.
  - Global barrier between stages. All senders complete shuffle write before receivers request data (shuffle read)

# Spark Communication Model (Shuffles)

- **All data exchanges between executors implemented via *shuffle***
  - Senders ("mappers") send data to block managers; block managers write to disks, tell scheduler *how much* destined for each reducer
  - Barrier until all mappers complete shuffle writes
  - Receivers ("reducers") request data from block managers *that have data for them*; block managers read and send



# Spark Programming Model: Example

Create array of {1, 2, …, 1,000,000}

Partition array into a 40-partition RDD (can also create from file). Executors will execute *tasks* on parititions, so this is also the maximum parallelism.

Spark *transformation* (Create new RDD from old RDD/RDDs)

Spark *action* (return result to driver)

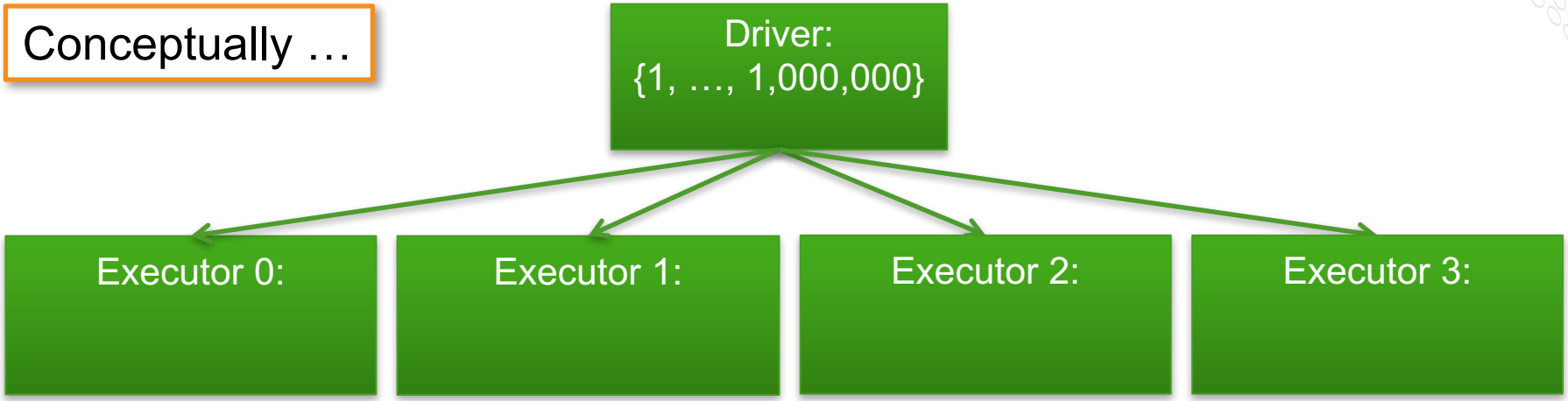
```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 40)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

compute

Lazy Evaluation: No computation until result requested

# Example: Line-by-line

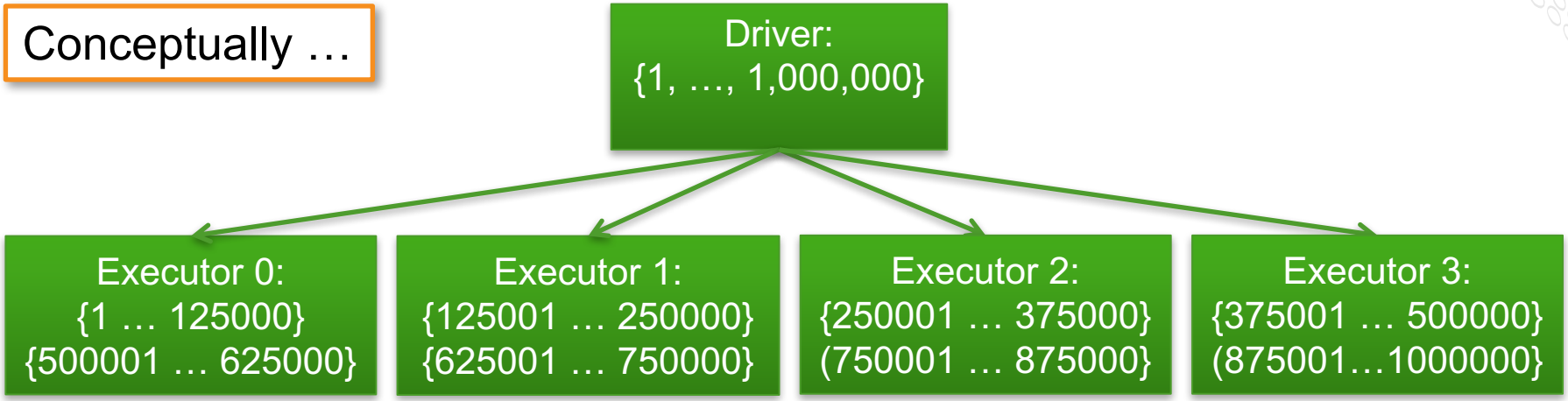
Conceptually …




```
val arr1M = Array.range(1,1000001)
```

# Example: Line-by-line

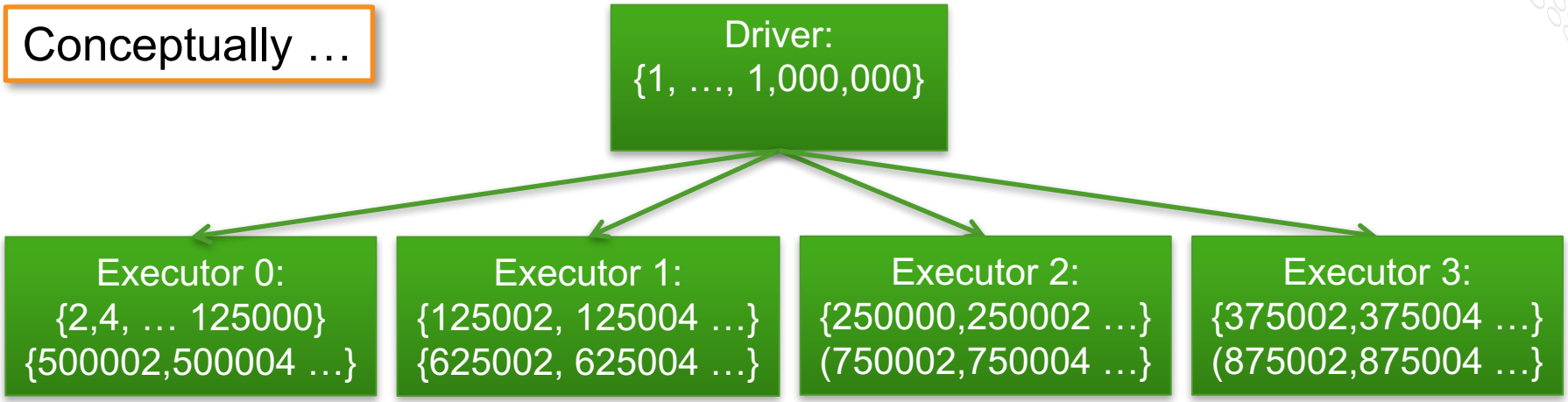
Conceptually …




```
val rdd1M = sc.parallelize(arr1M, 8)
```

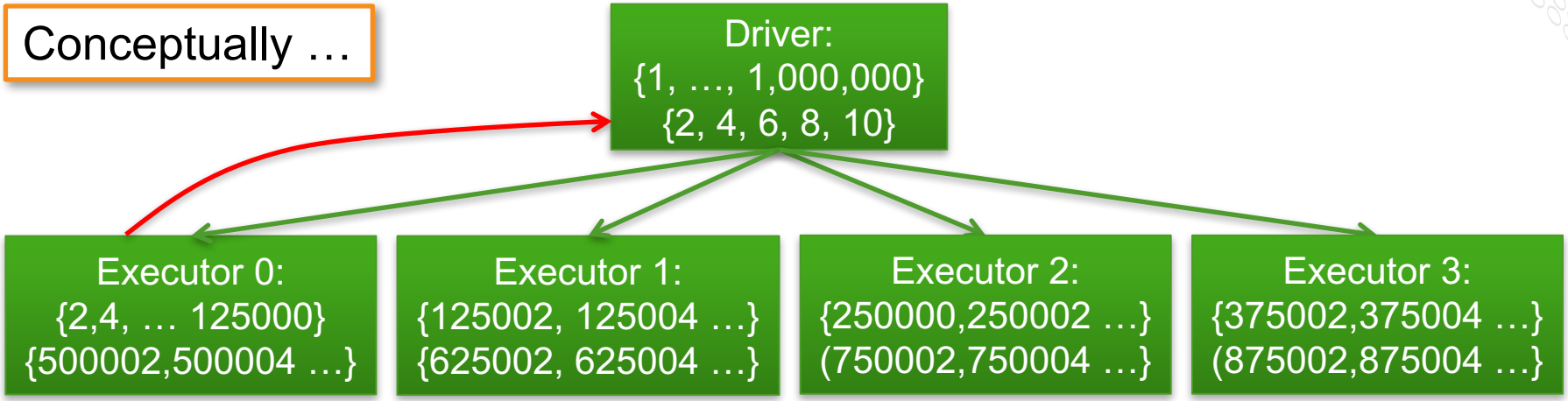
# Example: Line-by-line

Conceptually …




```
val evens = rdd1M.filter(a => a%2==0)
```

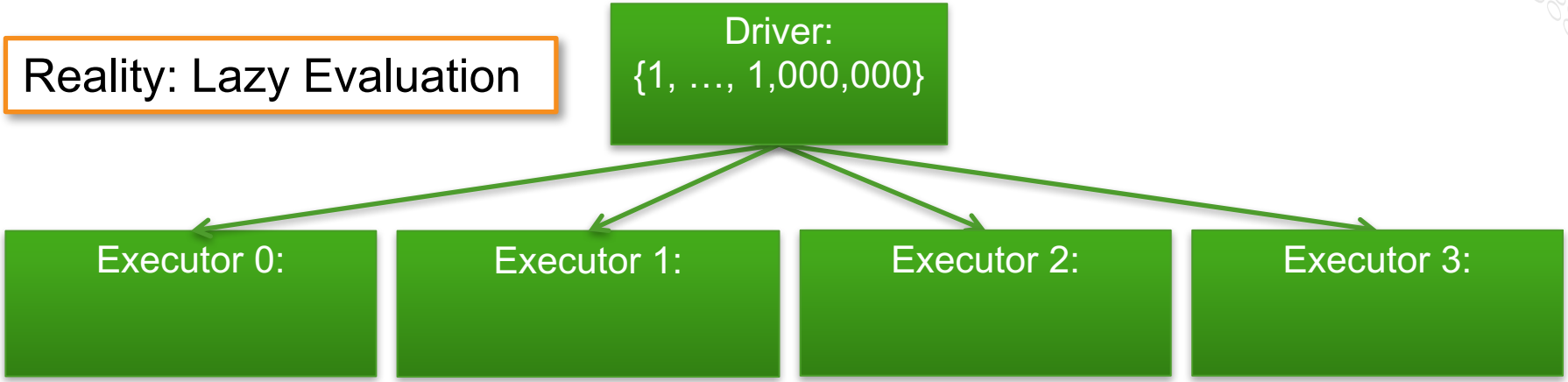
# Example: Line-by-line




```
evens.take(5)
```

# Example: Line-by-line

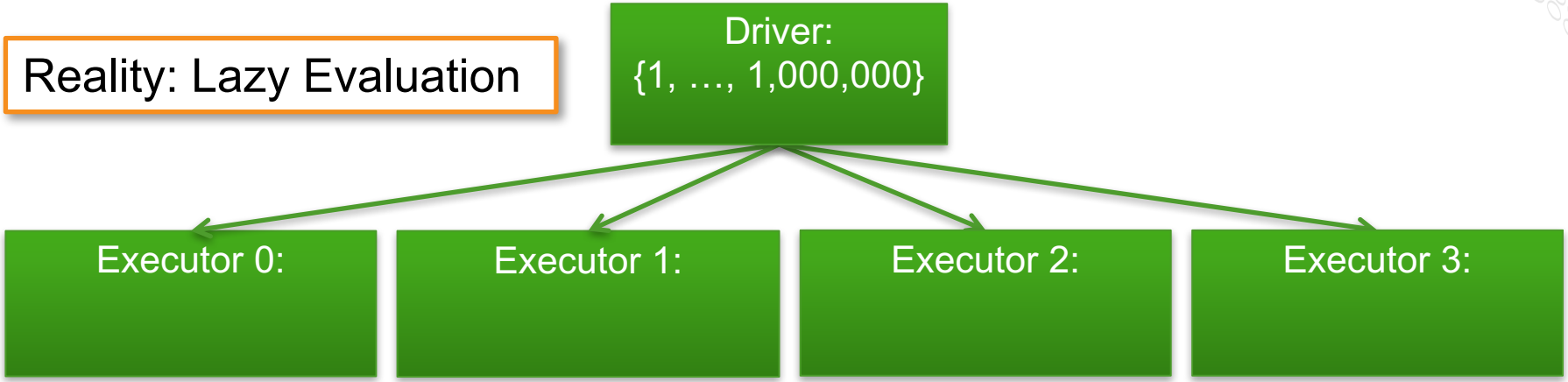
Reality: Lazy Evaluation

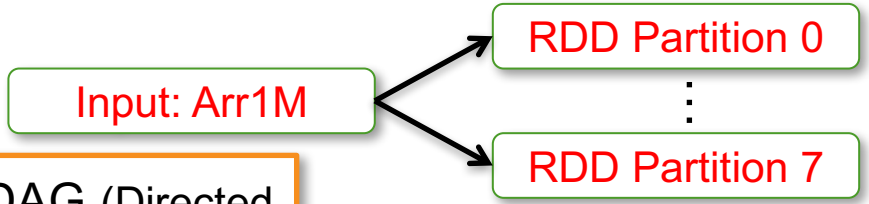



```
val arr1M = Array.range(1,1000001)
```

# Example: Line-by-line

Reality: Lazy Evaluation



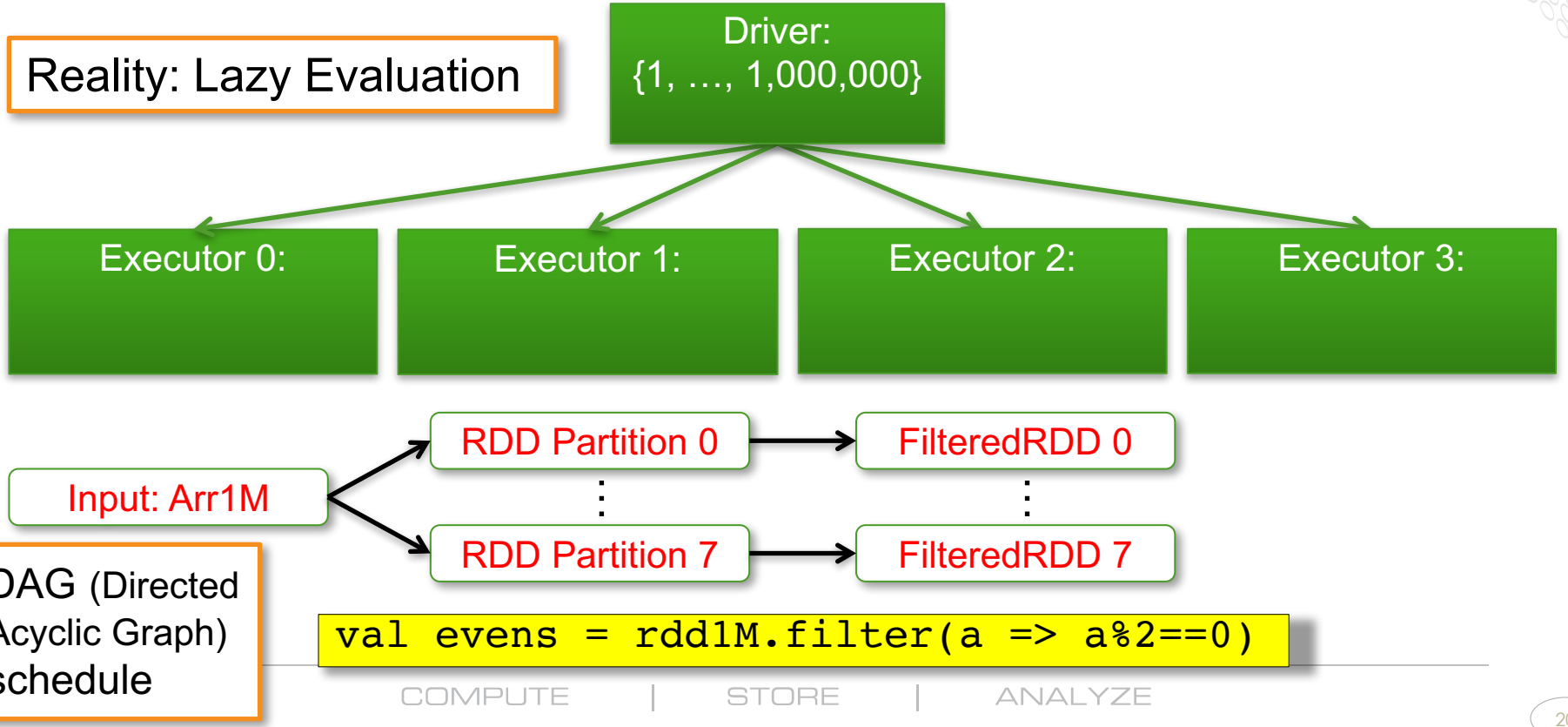




DAG (Directed Acyclic Graph) schedule

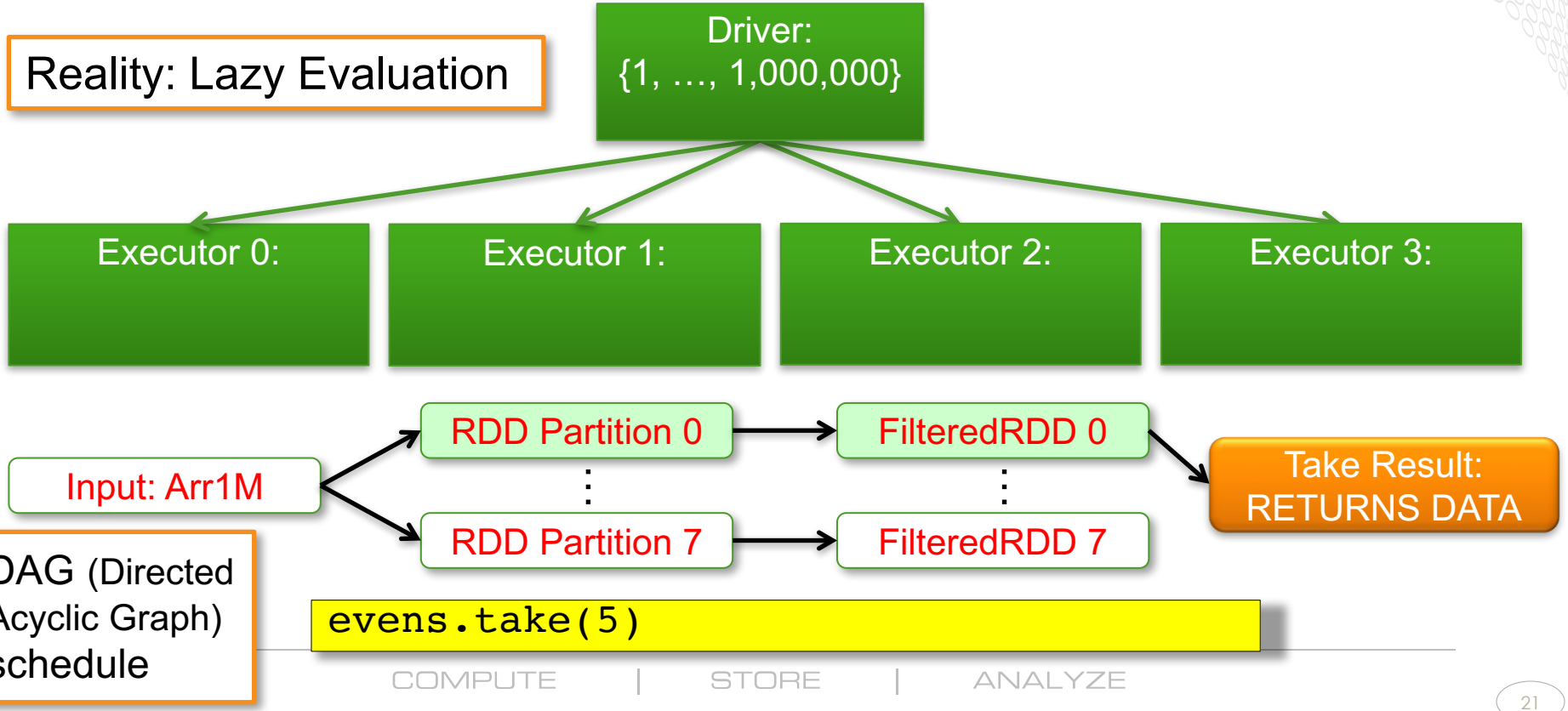
```
val rdd1M = sc.parallelize(arr1M, 8)
```

# Example: Line-by-line

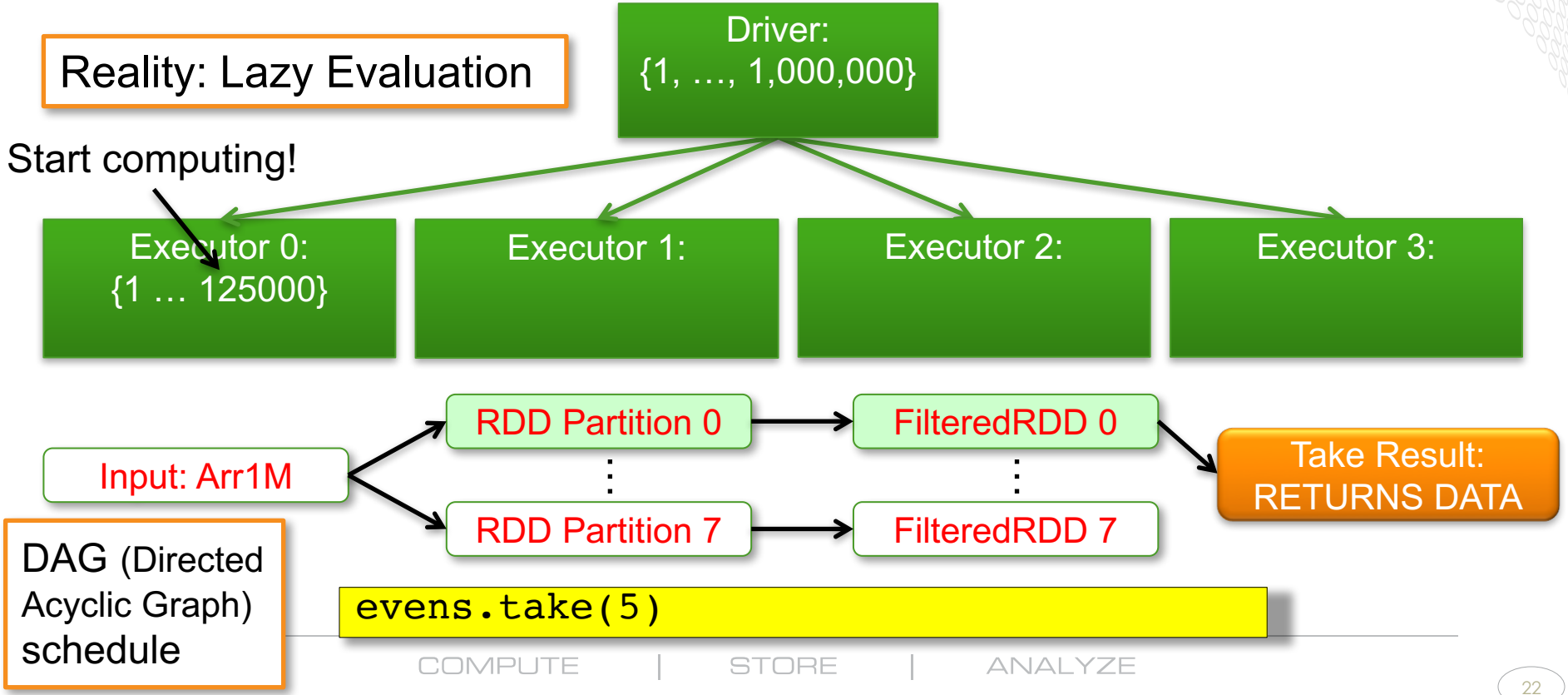
Reality: Lazy Evaluation



# Example: Line-by-line

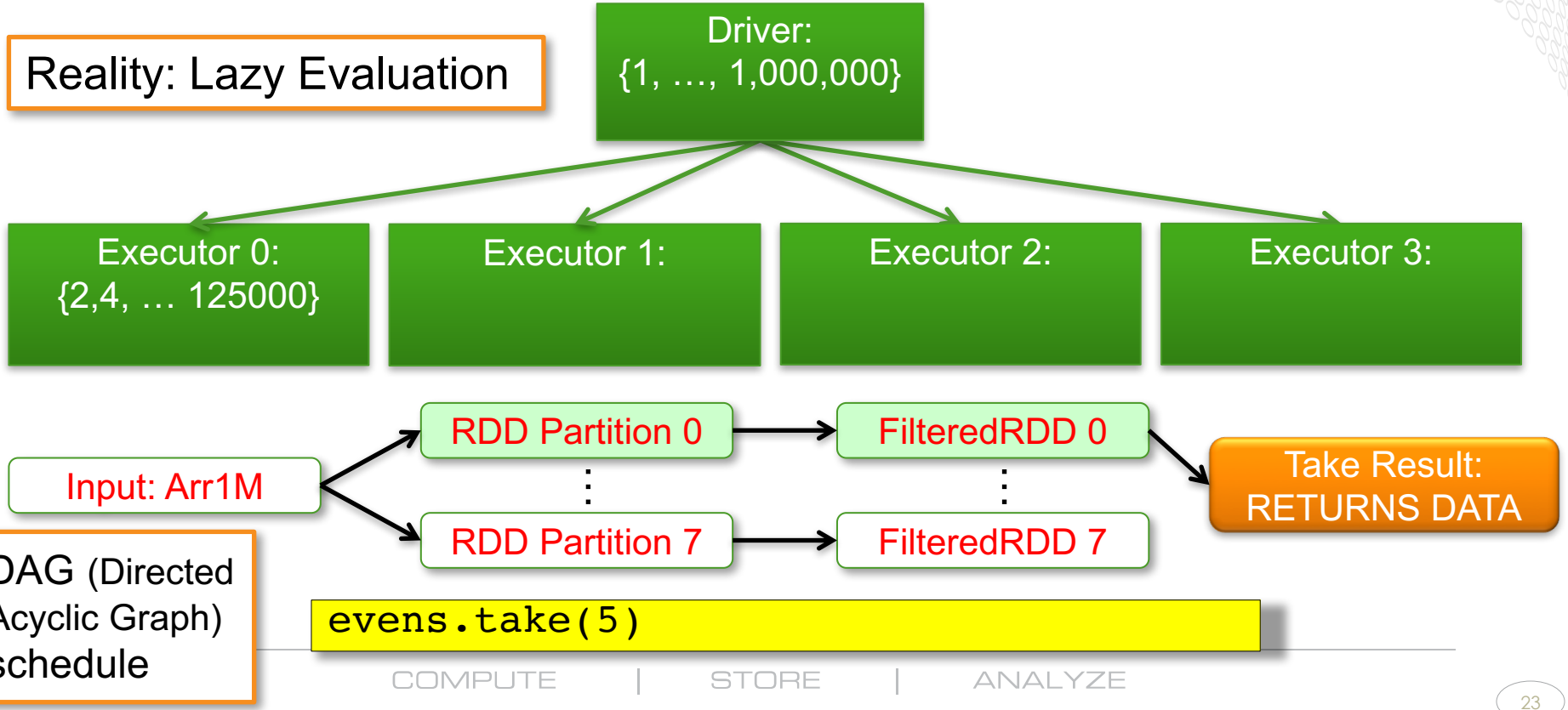


# Example: Line-by-line



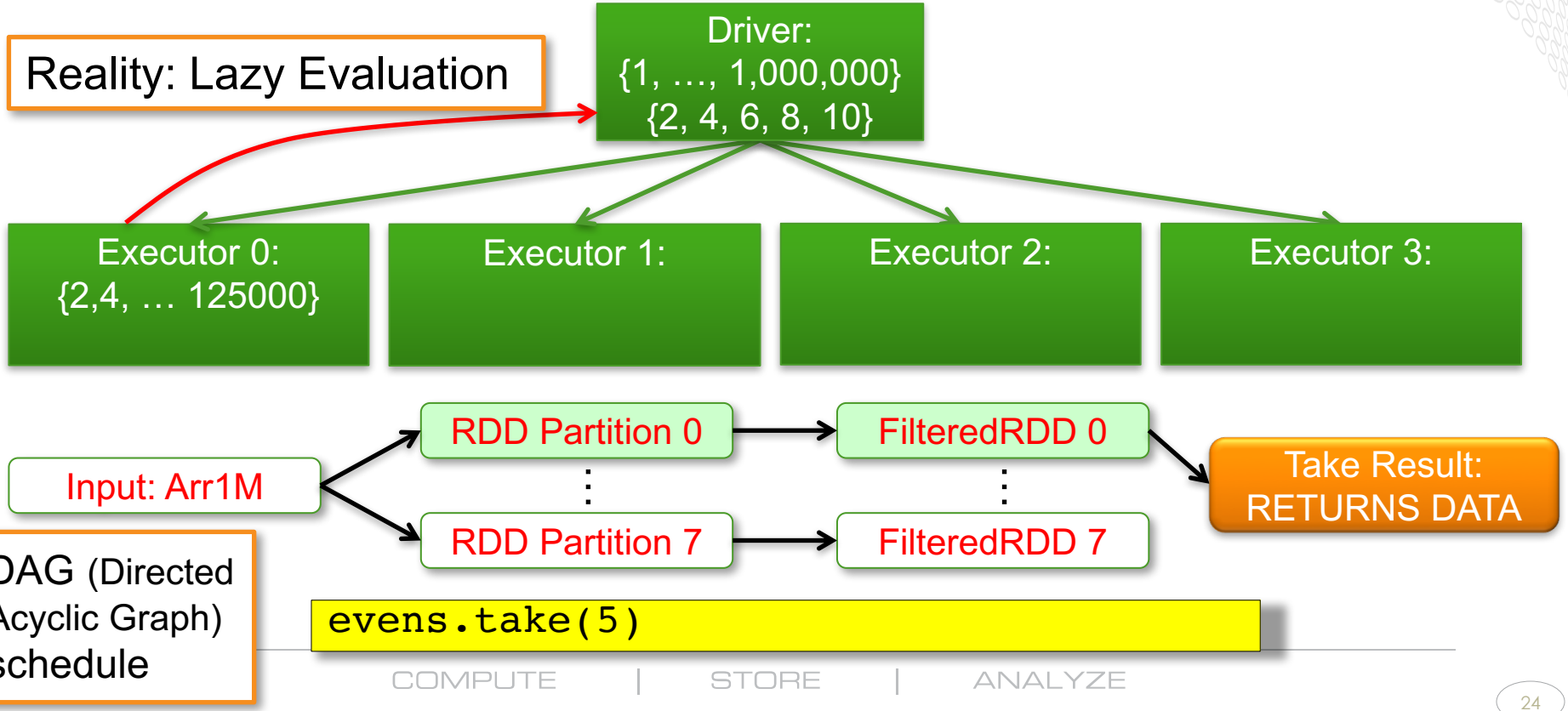
# Example: Line-by-line

Reality: Lazy Evaluation




DAG (Directed Acyclic Graph) schedule

# Example: Line-by-line



# Wait a second …

- **How did Spark know that take() would only require data from one partition?**
  - What if filter() left fewer than 5 elements in the first partition?

# Wait a second …

- **How did Spark know that take() would only require data from one partition?**
  - What if filter() left fewer than 5 elements in the first partition?
- **Answer … It didn't.**
  - Take is typically used to fetch a small initial piece of the data
  - Spark guesses that it will all be available in the first partition
  - If not, tries the first four partitions …
  - Then the first 16 …
  - Etc…

# Modified example

Count returns the total size of an RDD

Reduce performs a reduction over the dataset, combining elements with the argument function.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Imagine we want to perform a number of actions on (i.e., return different data about) our filtered RDD.
- For each action, Spark computes all the DAG steps…

# Modified example

Count returns the total size of an RDD

Reduce performs a reduction over the dataset, combining elements with the argument function.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- **Problem: This means recomputing the filtered "evens" RDD three times – inefficient.**

# Modified example

Persist tells Spark to keep the data in memory even after it is done with the action. Allows future actions to reuse without recomputing. Cache is synonym for default storage level (memory). Can also persist on disk, etc.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
evens.persist()  // or cache()
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- **Problem: This means recomputing the filtered "evens" RDD three times – inefficient.**
- **Solution: Persist the RDD!***

*Relies on immutability of val

# Multi-stage Spark Example: Word Count

Load file

flatMap maps one value to (possibly) many, instead of one-to-one like map

groupByKey combines all key-value pairs with the same key (k, v1), …, (k,vn) into a single key-value pair (k, (v1, …, vn)).

Collect returns all elements to the driver

More efficient: replace group and sum with reduceByKey

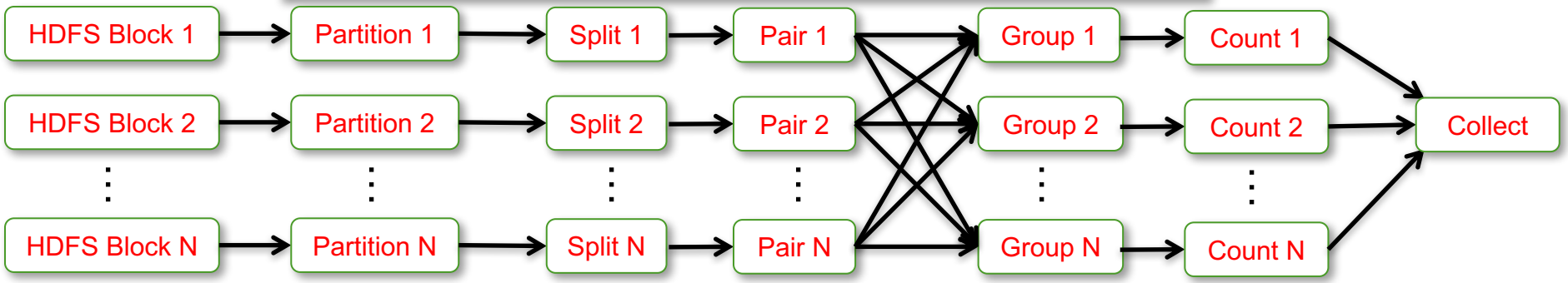
```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a simple example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

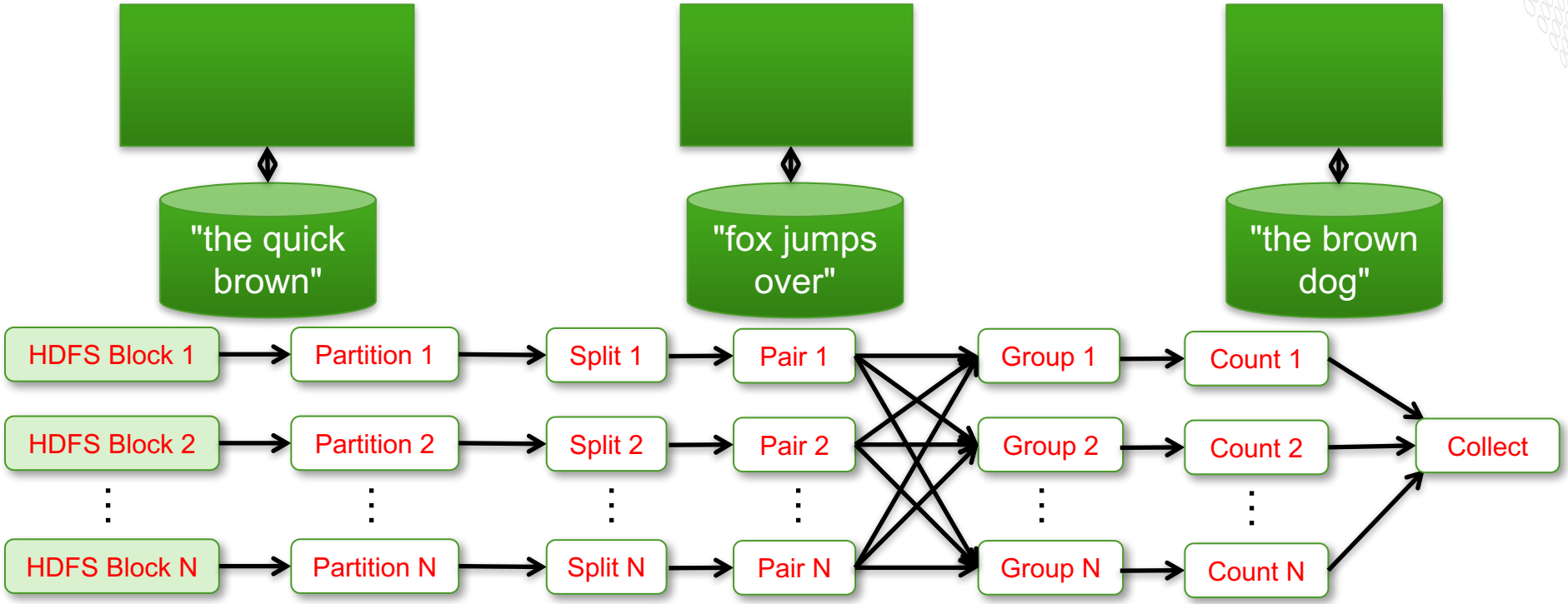
# The Spark DAG

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

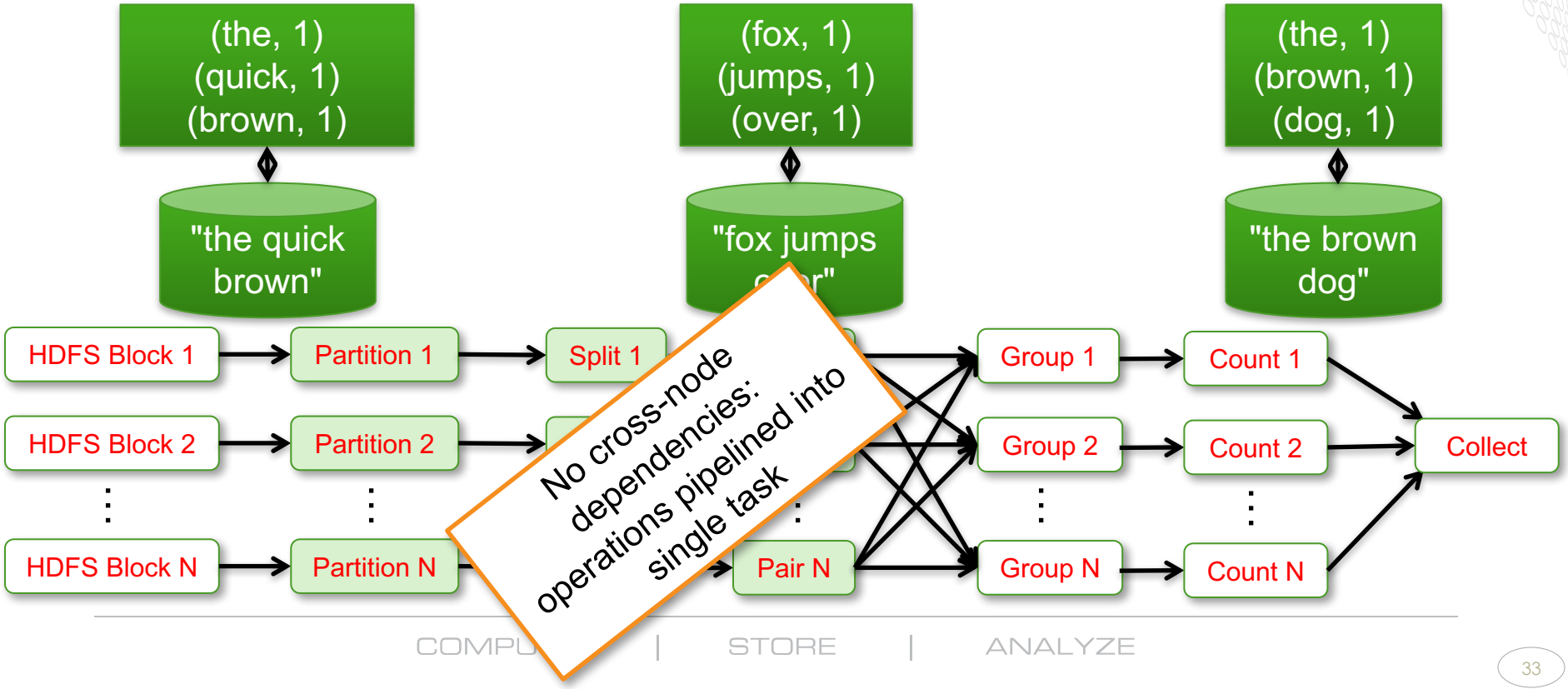
Execute!



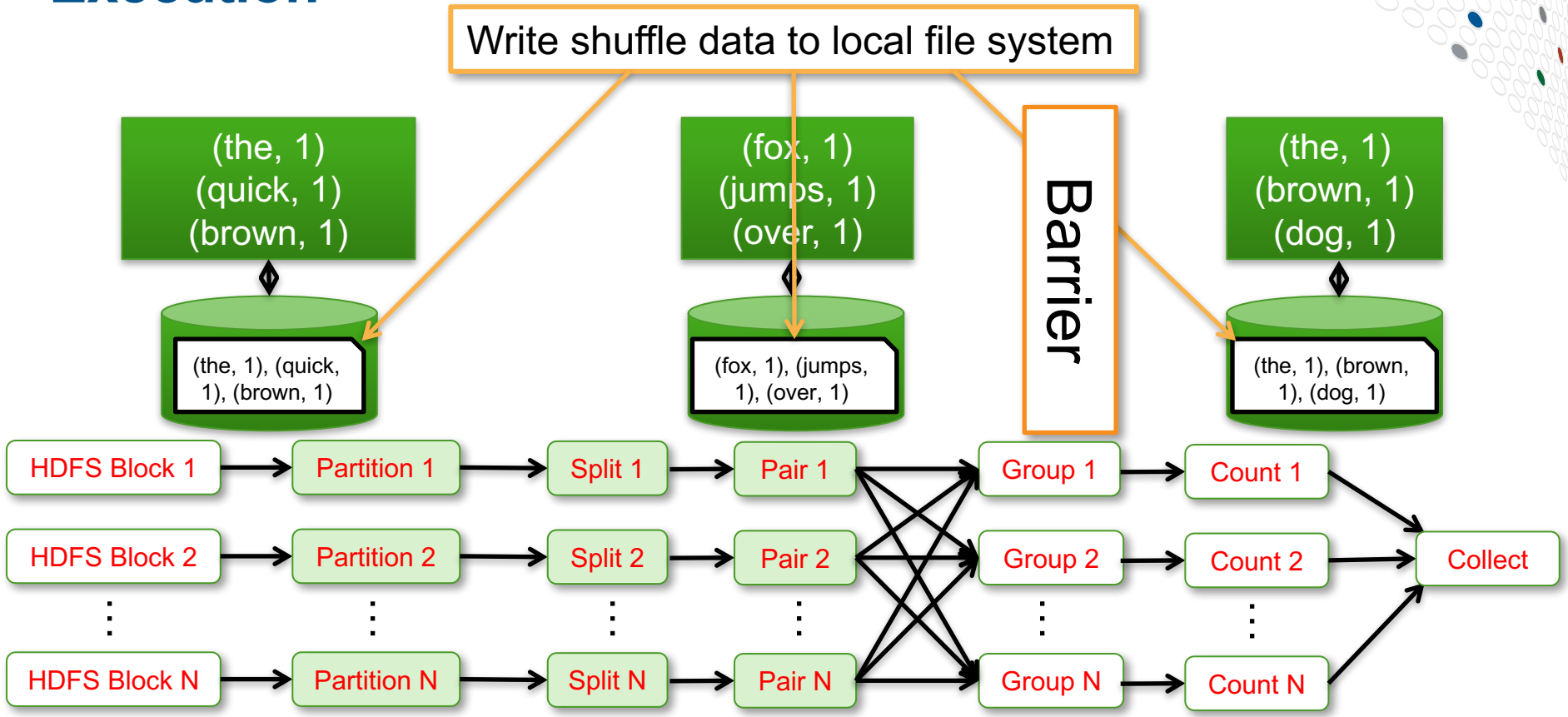
# Execution


"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
Collect
HDFS Block N
Partition N
Split N
Pair N
Group N
Count N

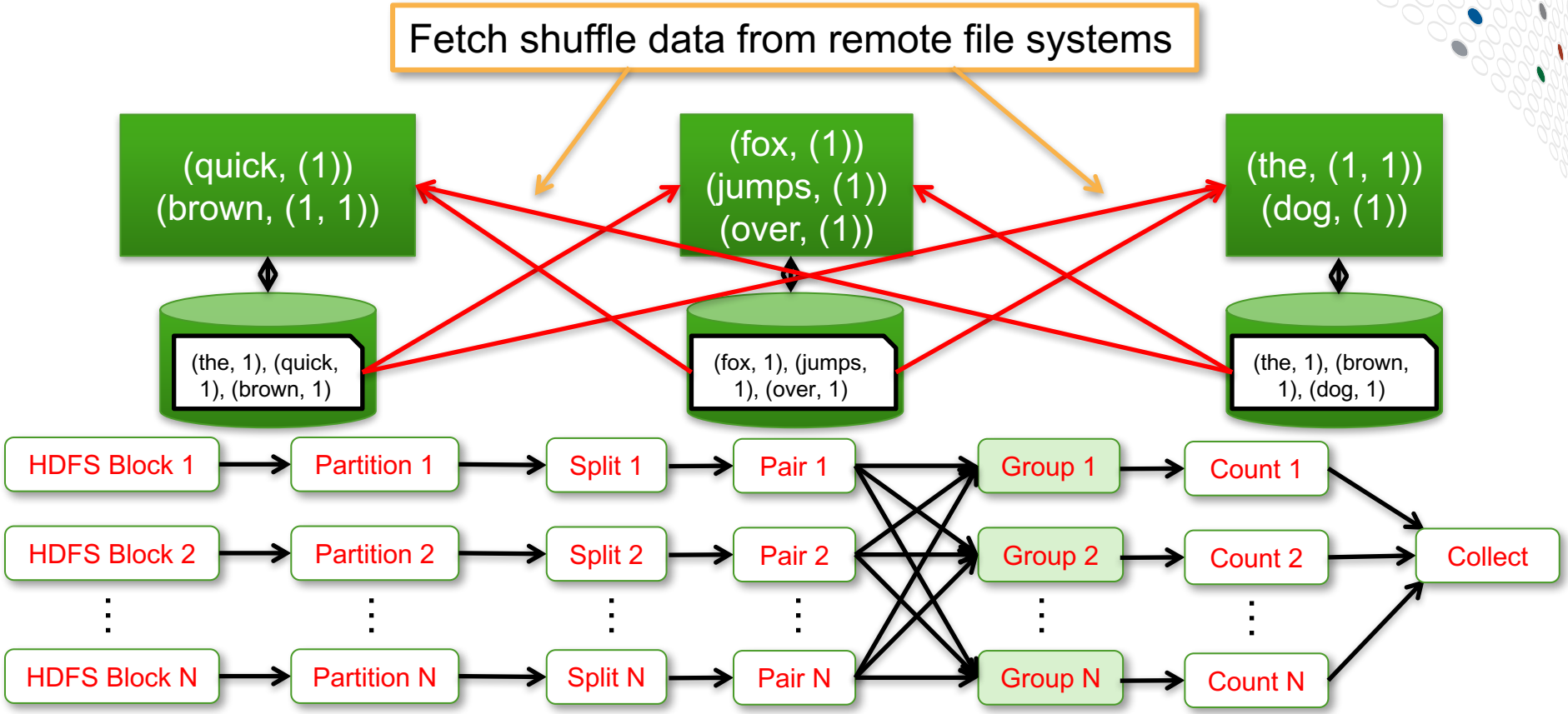
# Execution


(the, 1)
(quick, 1)
(brown, 1)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 1)
(brown, 1)
(dog, 1)
"the quick brown"
"fox jumps
"the brown dog"
HDFS Block 1
HDFS Block 2
HDFS Block N
Partition 1
Partition 2
Partition N
Split 1
Pair N
Group 1
Group 2
Group N
Count 1
Count 2
Count N
Collect
No cross-node dependencies: operations pipelined into single task

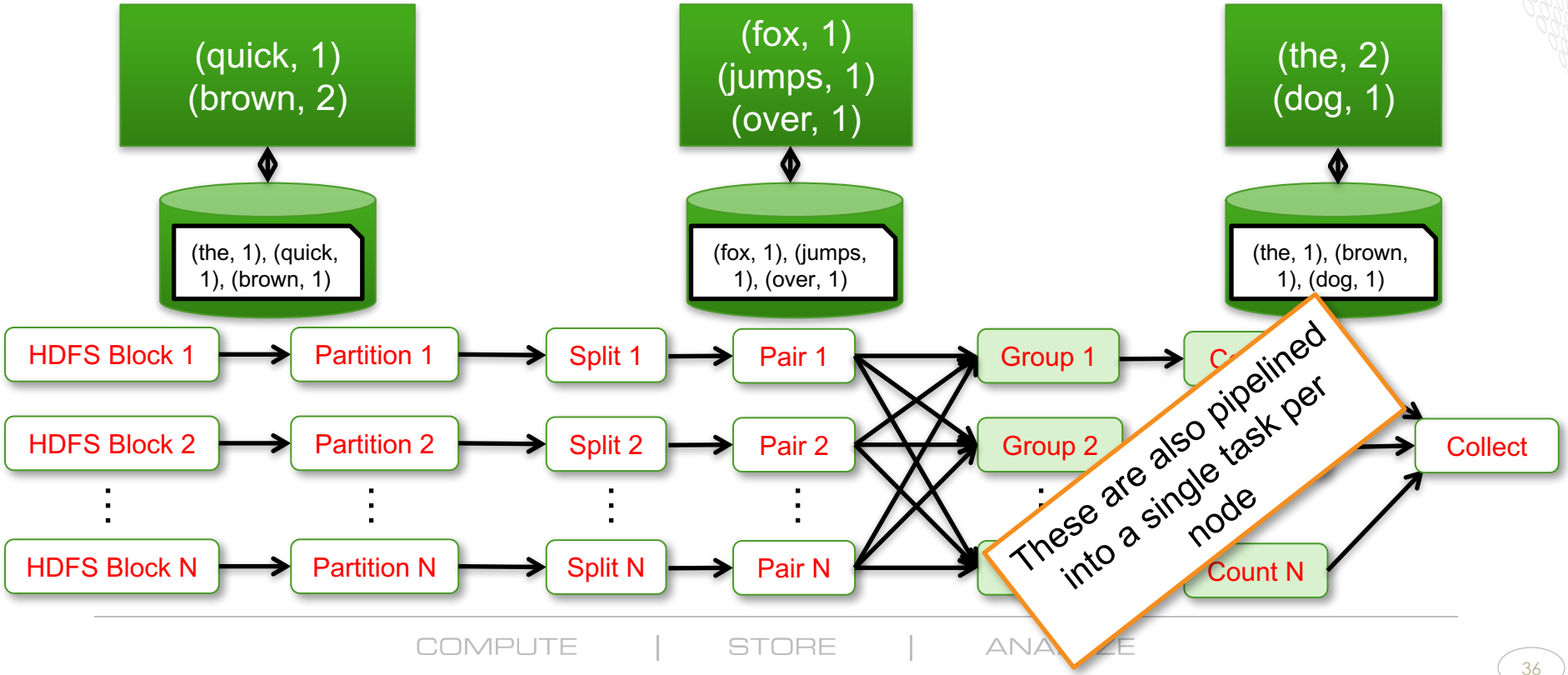
# Execution


Write shuffle data to local file system
(the, 1)
(quick, 1)
(brown, 1)
(fox, 1)
(jumps, 1)
(over, 1)
Barrier
(the, 1)
(brown, 1)
(dog, 1)
(the, 1), (quick, 1), (brown, 1)
(fox, 1), (jumps, 1), (over, 1)
(the, 1), (brown, 1), (dog, 1)
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
Collect
HDFS Block N
Partition N
Split N
Pair N
Group N
Count N

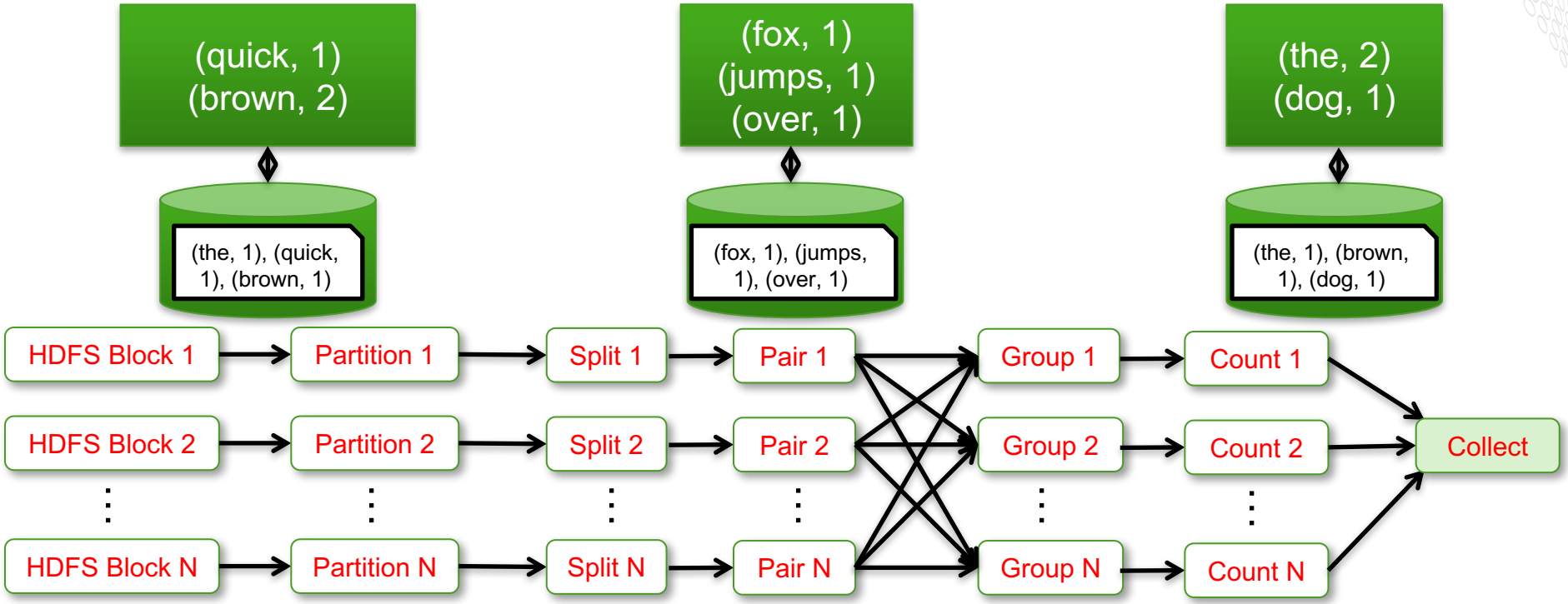
# Execution


Fetch shuffle data from remote file systems
(quick, (1))
(brown, (1, 1))
(fox, (1))
(jumps, (1))
(over, (1))
(the, (1, 1))
(dog, (1))
(the, 1), (quick, 1), (brown, 1)
(fox, 1), (jumps, 1), (over, 1)
(the, 1), (brown, 1), (dog, 1)
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
Collect
HDFS Block N
Partition N
Split N
Pair N
Group N
Count N

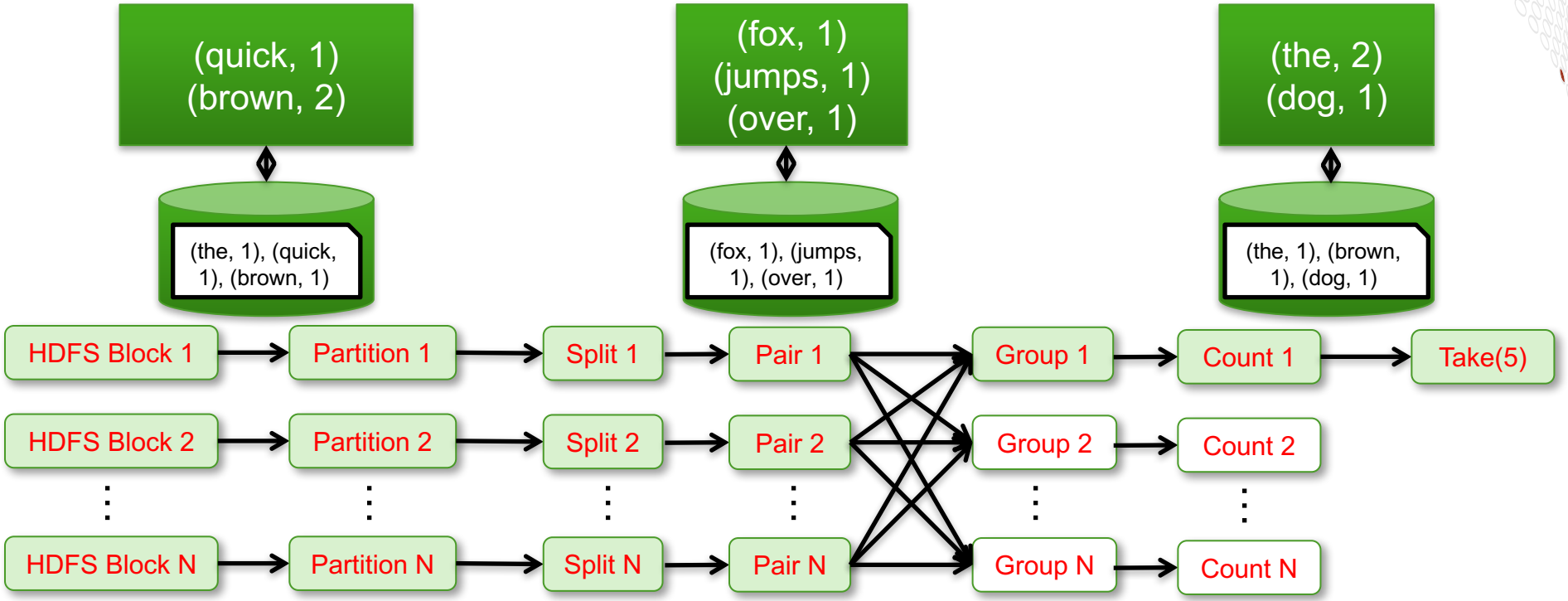
# Execution


(quick, 1)
(brown, 2)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 2)
(dog, 1)
(the, 1), (quick, 1), (brown, 1)
(fox, 1), (jumps, 1), (over, 1)
(the, 1), (brown, 1), (dog, 1)
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
HDFS Block N
Partition N
Split N
Pair N
Count N
Collect
These are also pipelined into a single task per node

# Execution


(quick, 1)
(brown, 2)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 2)
(dog, 1)
(the, 1), (quick, 1), (brown, 1)
(fox, 1), (jumps, 1), (over, 1)
(the, 1), (brown, 1), (dog, 1)
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
Collect
HDFS Block N
Partition N
Split N
Pair N
Group N
Count N

# Execution


(quick, 1)
(brown, 2)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 2)
(dog, 1)
(the, 1), (quick, 1), (brown, 1)
(fox, 1), (jumps, 1), (over, 1)
(the, 1), (brown, 1), (dog, 1)
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
Take(5)
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
HDFS Block N
Partition N
Split N
Pair N
Group N
Count N

# Spark on Cray XC

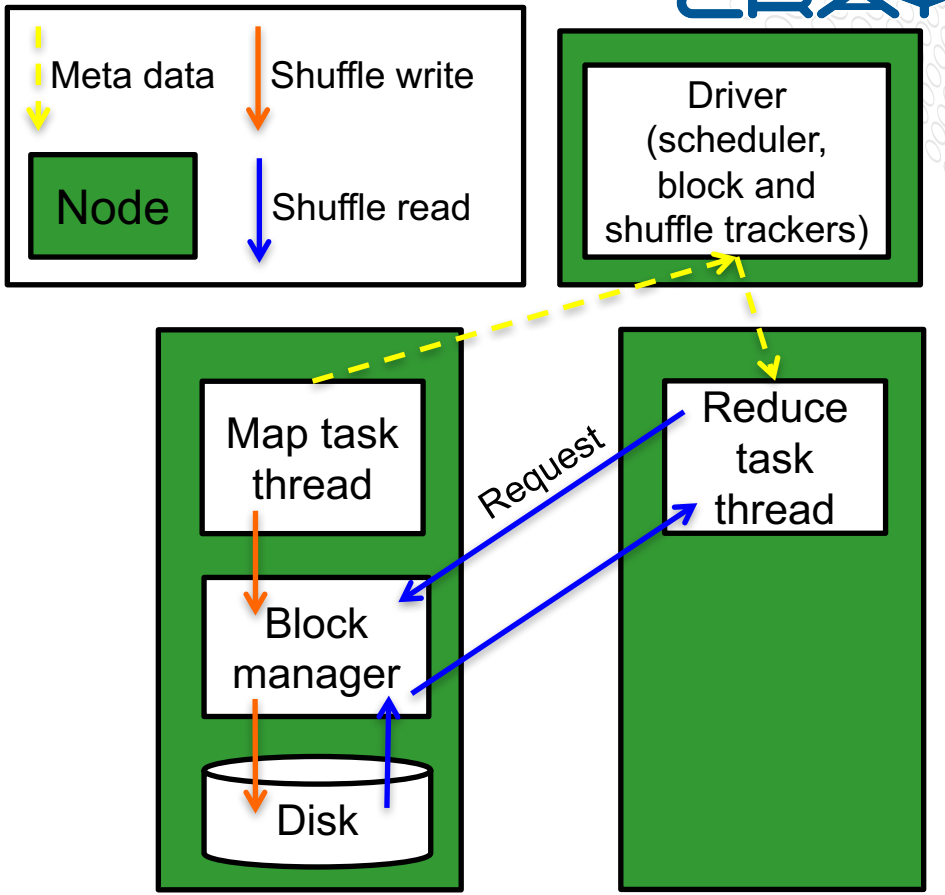
# Spark on XC: Typical Setup Options

- **Cluster Compatibility Mode (CCM) option**
  - Set up and launch standalone Spark cluster in CCM mode; run interactively from Mom node or submit batch script
  - An example recipe can be found in:

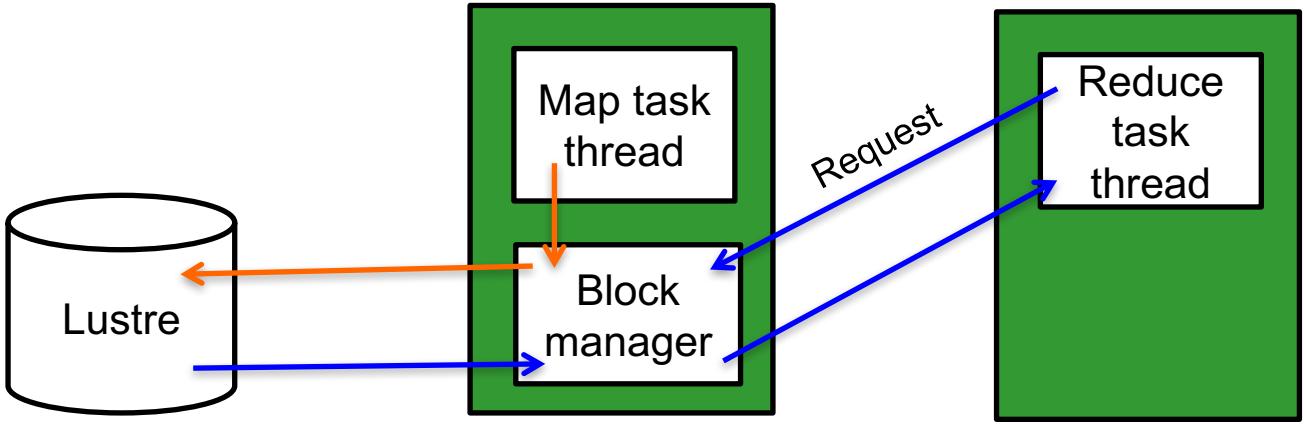
    **"Experiences Running and Optimizing the Berkeley Data Analytics Stack on Cray Platforms", Maschhoff and Ringenburg, CUG 2015**
- **Container option**
  - Shifter container runtime (think "Docker for XC") developed at NERSC
  - Acquire node allocation: run master image on one node, interactive image on another, worker images on rest
  - Cray's Urika-XC analytics suite uses this approach
- **Challenge: Lack of local storage for Spark shuffles and spills.**

# Reminder: Spark Shuffle – Standard Implementation

- Senders ("mappers") send data to block managers; block managers write to **local disks**, tell driver how much destined for each reducer
- Barrier until all mappers complete shuffle writes
- Receivers ("reducers") request data from block managers that have data for them; block managers read from **local disk** and send
- Key assumption: large, fast local block storage device(s) available on executor nodes

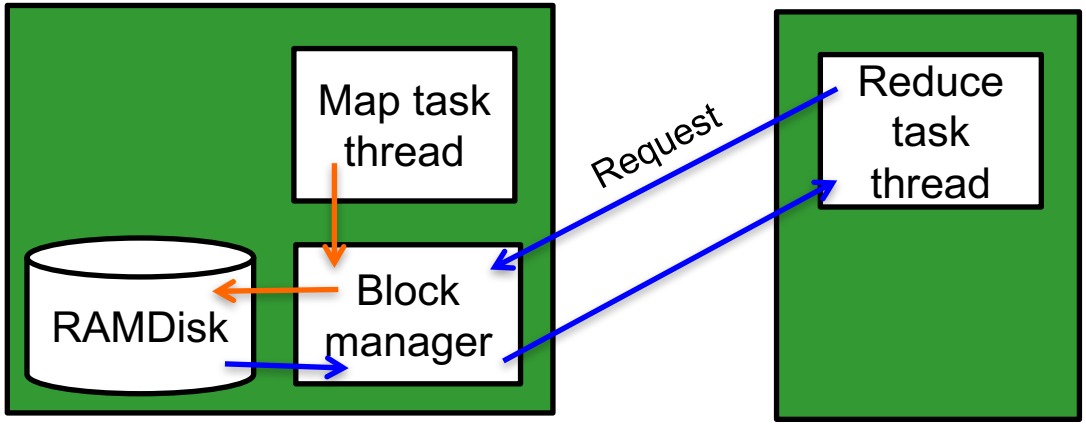


# Shuffle on XC – Version 1



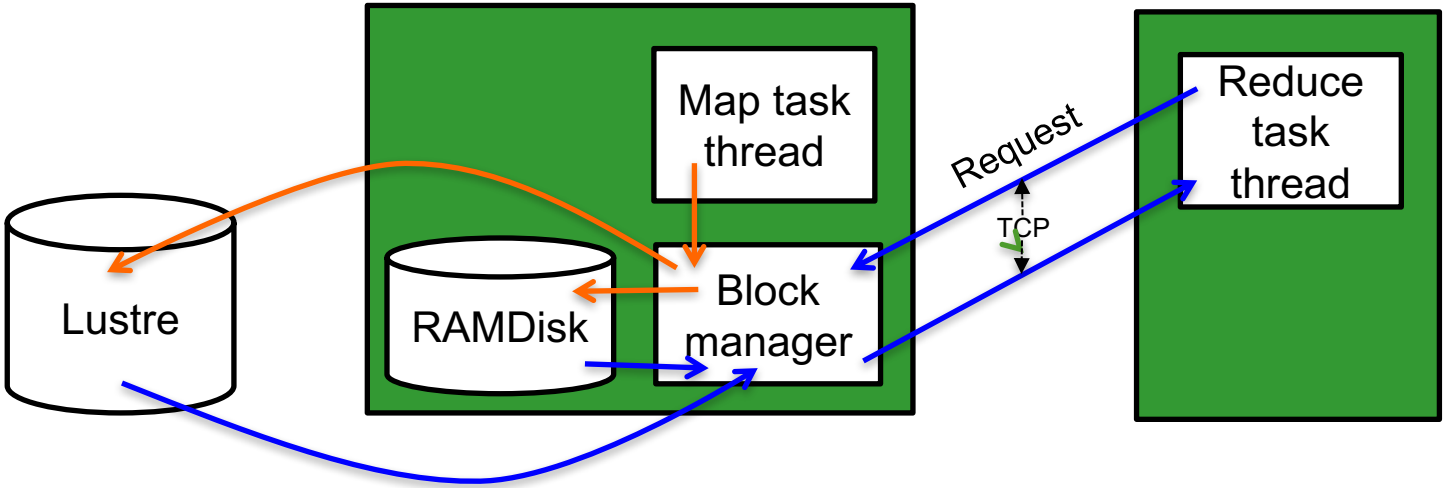

- **Problems: No local disk on standard XC40**
- **First try: Write to Lustre instead**
  - Biggest Issue: Poor file access pattern for lustre (lots of small files, constant opens/closes). Creates a major bottleneck on Lustre Metadata Server (MDS).
  - Issue 2: Unnecessary extra traffic through network

# Shuffle on XC – Version 2



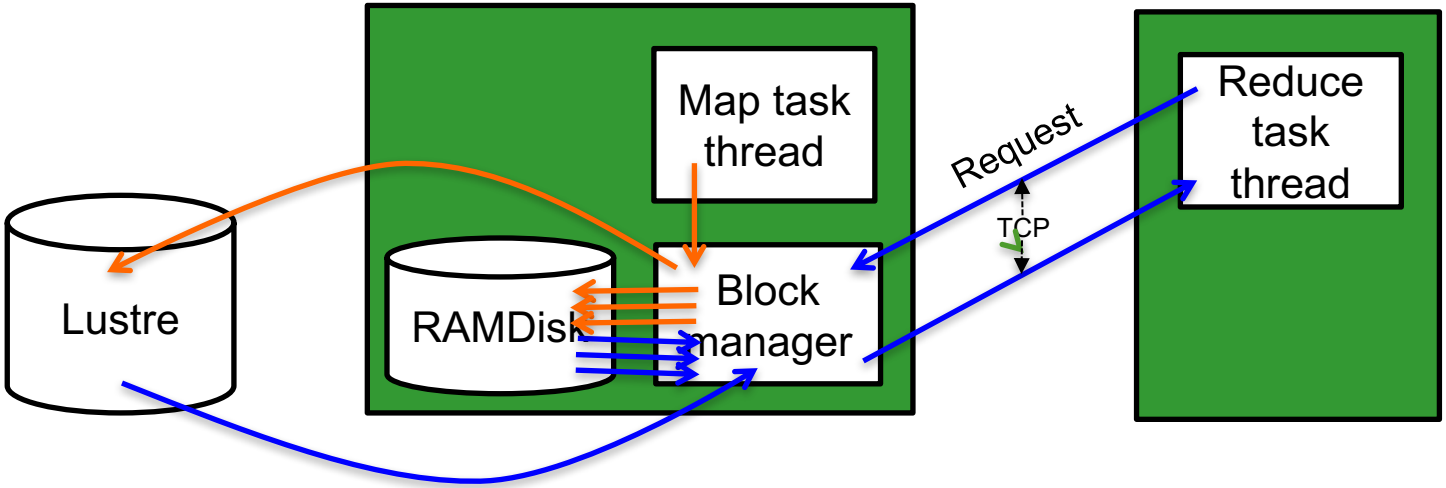

- **Second try: Write to RAMDisk**
  - Much faster, but …
  - Issues: Limited to lessor of: 50% of node DRAM or unused DRAM; Fills up quickly; takes away memory that could otherwise be allocated to Spark
  - Spark behaves unpredictably when it's local scratch space fills up (failures not always simple to diagnose)

# Shuffle on XC – Version 3



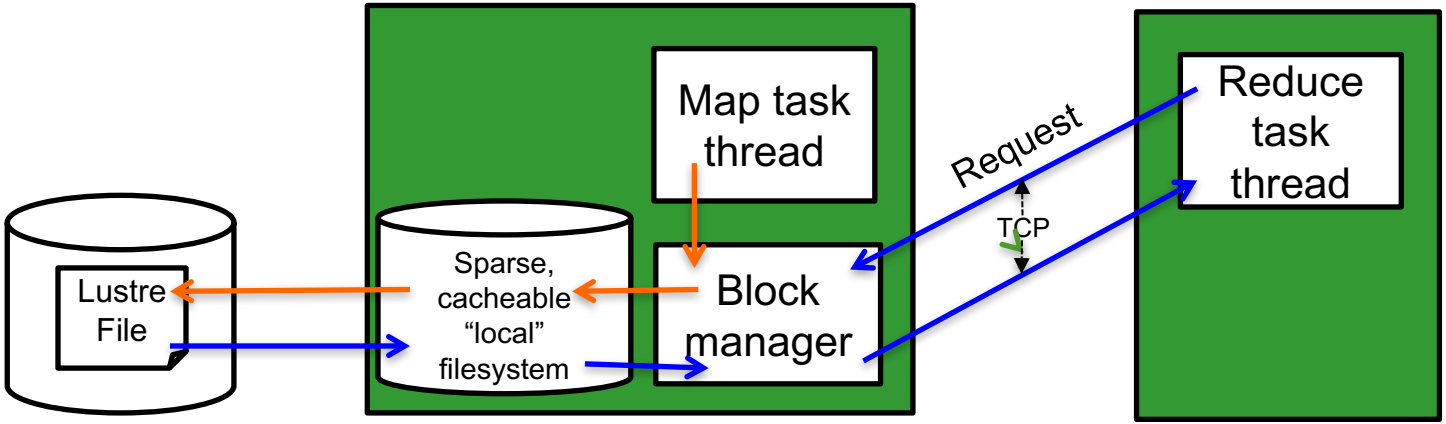

- **Third try: Write to RAMDisk and Lustre**
  - Set local directories to RAMdisk and lustre (can be list)
  - Initially fast and keeps working when RAMDisk full
  - Issues: Slow once RAMDisk fills; Round robin between directories (no bias towards faster RAM)

# Shuffle on XC – Version 3




- **Third try: Write to RAMDisk and Lustre**
  - Set local directories to RAMdisk and lustre (can be list)
  - Initially fast and keeps working when RAMDisk full
  - Issues: Slow once RAMDisk fills; Round robin between directories (no bias towards faster RAM), *but can specify multiple RAM directories*

# Shuffle on XC – with Shifter PerNodeCache



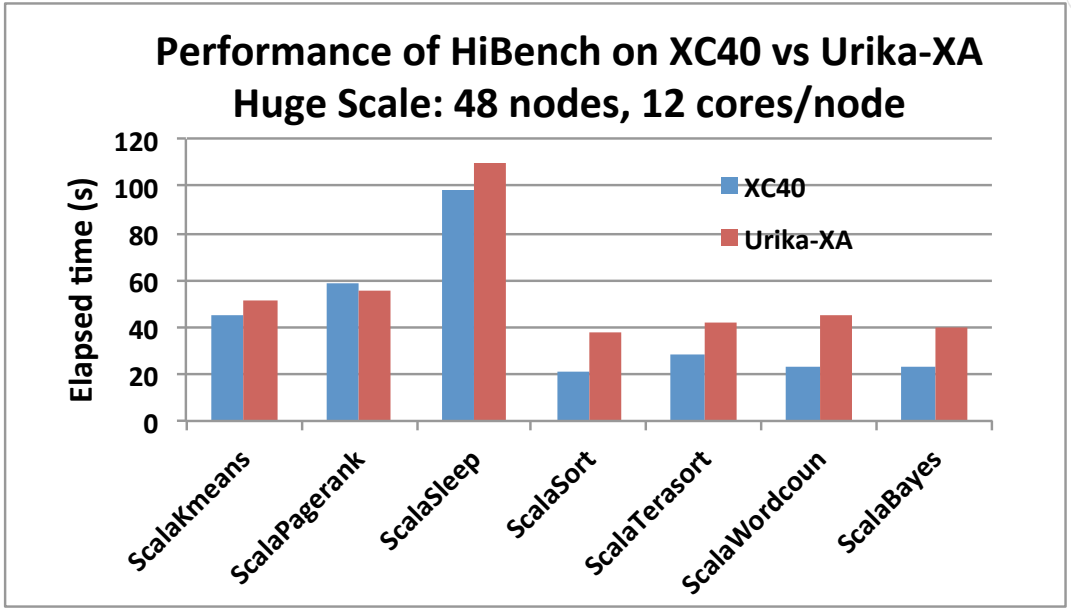

- **Shifter implementation: Per-node loopback file system**
  - NERSC's Shifter containerization (in Cray CLE6) provides optional loopback-mounted per-node temporary filesystem
  - Local to each node – fully cacheable
  - Backed by a single sparse file on Lustre – greatly reduced MDS load, plenty of capacity, doesn't waste space
  - Performance comparable to RAMDisk, without capacity constraints (Chaimov et al, CUG '16)
- **Urika-XC ships as a Shifter image and uses this approach**

# Other Spark Configurations

- **Many config parameters … some of the more relevant:**
  - **spark.shuffle.compress**: Defaults to true. Controls whether shuffle data is compressed. In many cases with fast interconnect, compression and decompression overhead can cost more than the transmission time savings. However, can still be helpful if limited shuffle scratch space.
  - **spark.locality.wait**: Defaults to 3 (seconds). How long to wait for available resources on a node with data locality before trying to execute tasks on another node. Worth playing around with - decrease if seeing a lot of idle executors. Increase if seeing poor locality. (Can check both in history server.) Do not set to 0!

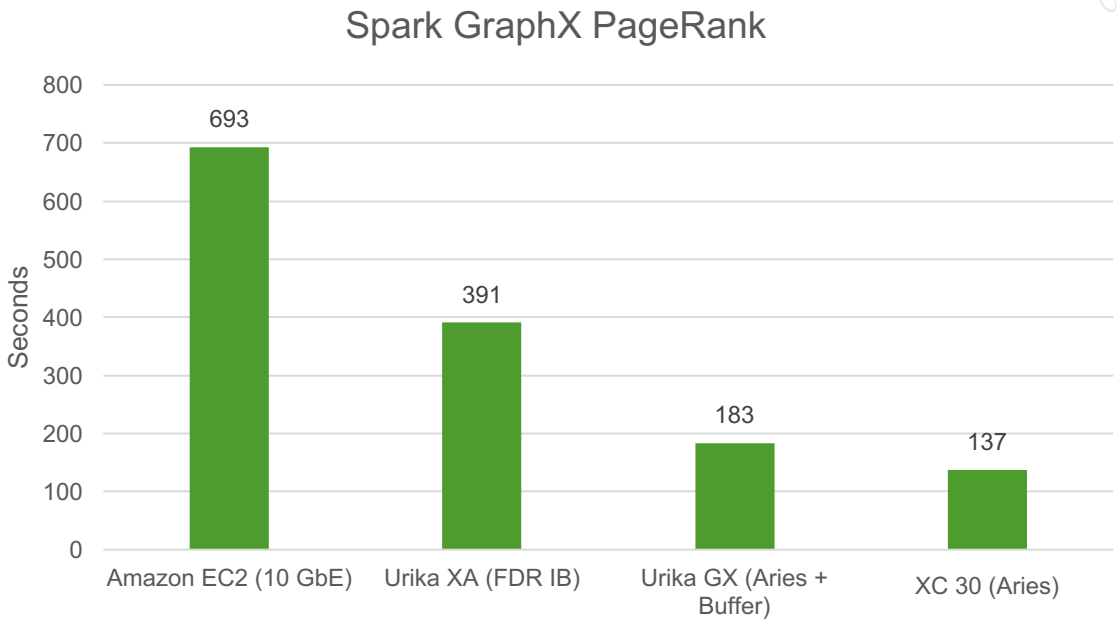
# Spark Performance on XC: HiBench

- **Intel HiBench**
  - Originally MapReduce, Spark added in version 4
- **Compared performance with Urika XA system**
  - XA: FDR Infiniband, XC40: Aries
  - Both: 32 core Haswell nodes
  - XA: 128 GB/node, XC40: 256 GB/node (problems fit in memory on both)
- **Similar performace on Kmeans, PageRank, Sleep**
- **XC40 faster for Sort, TeraSort, Wordcount, Bayes**



# Spark Performance on XC: GraphX

- **GraphX PageRank**
  - 20 iterations on Twitter dataset
  - Interconnect sensitive
- **GX has slightly higher latency and lower peak TCP bandwidth than XC due to buffer chip**



# Demo: PySpark in Jupyter

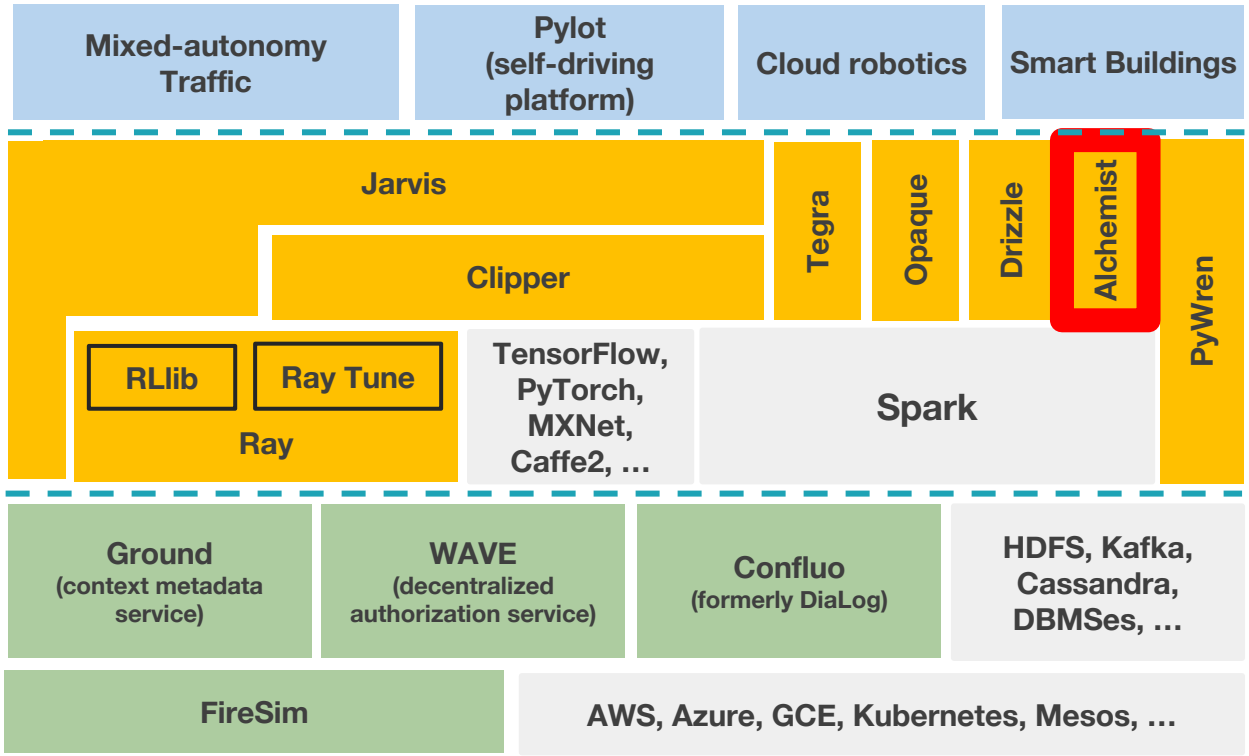
# Alchemist

## An Apache Spark ⇔ MPI Interface

A Collaboration of Cray and the UC Berkeley RiseLab (Alex Gittens, Kai Rothauge, Michael W. Mahoney, Shusen Wang, Jey Kottalam)

Slides courtesy Kai Rothauge

# RISE Stack




Slides courtesy Kai Rothauge, UC Berkeley

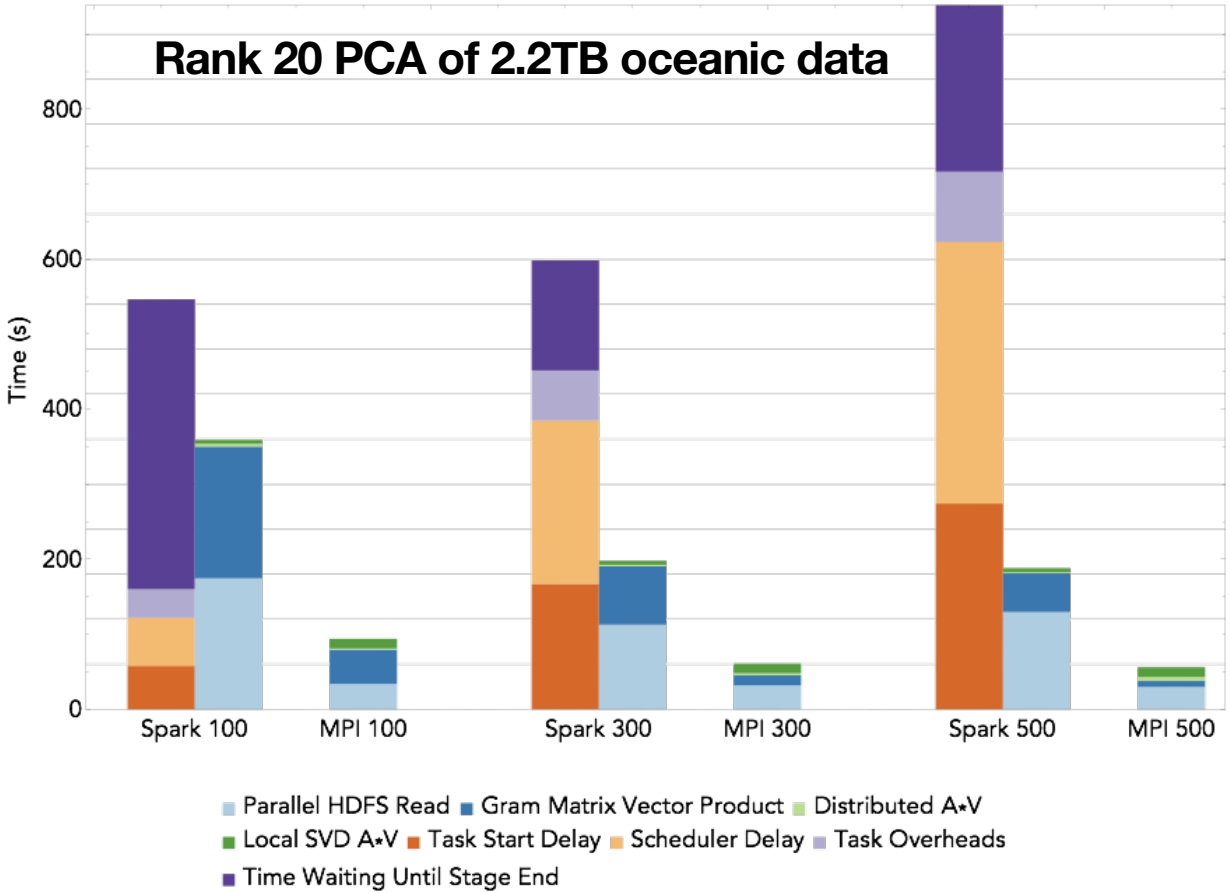
# MPI vs Spark

- Cray and AMPLab performed case study for numerical linear algebra on Spark vs. MPI
- Why do linear algebra in Spark?
  - **Pros**:
    - Faster development, easier reuse
    - One abstract uniform interface (RDD)
    - An entire ecosystem that can be used before and after the NLA computations
    - Spark can take advantage of available local linear algebra codes
    - Automatic fault-tolerance, out-of-core support
  - **Con**:
    - Classical MPI-based linear algebra implementations will be faster and more efficient

Slides courtesy Kai Rothauge, UC Berkeley

# MPI vs Spark

- Performed a case study for numerical linear algebra on Spark vs. MPI:
  - Matrix factorizations considered include Principal Component Analysis (PCA)
  - Data sets include
    - Oceanic data: 2.2 TB
    - Atmospheric data: 16 TB




A. Gittens et al. "Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+MPI using three case studies", 2016 IEEE International Conference on Big Data (Big Data), pages 204–213, Dec 2016.

Slides courtesy Kai Rothauge, UC Berkeley

# MPI vs Spark: Lessons learned

- With favorable data (tall and skinny) and well-adapted algorithms, linear algebra in Spark is 2x-26x slower than MPI when I/O is included
- Spark's overheads are orders of magnitude higher than the actual computations
  - Overheads include time until stage end, scheduler delay, task start delay, executor deserialize time, inefficiencies related to running code via JVM
- **The gaps in performance suggest it may be better to interface with MPI-based codes from Spark**

Slides courtesy Kai Rothauge, UC Berkeley

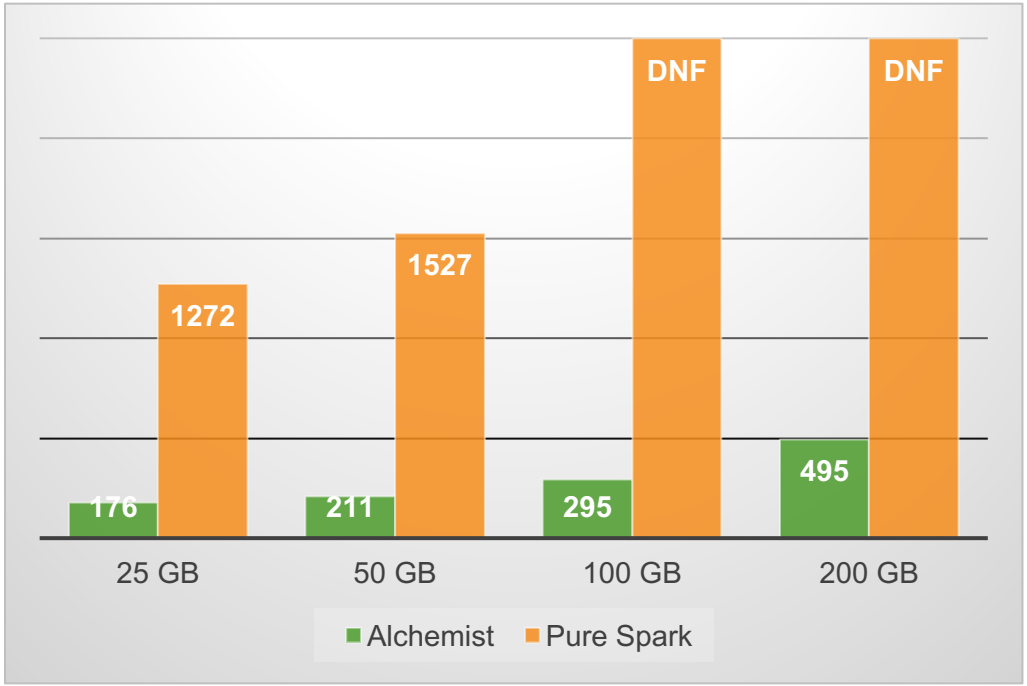
# Alchemist

- Interface between Apache Spark and ***existing*** MPI-based libraries for NLA, ML, etc.
- Design goals include making the system *easy to use*, *efficient*, and *scalable*
- Two main tasks:
  - Send distributed **input** matrices from Spark to MPI-based libraries (**Spark => MPI**)
  - Send distributed **output** matrices back to Spark (**Spark <= MPI**)
- Want as little overhead as possible when transferring data between Spark and a library
- Three possible approaches:
  - File I/O (e.g. HDFS) **too slow!** 
  - Use shared memory buffers, Apache Ignite, Alluxio, etc. **extra copy in memory** 
  - Use in-memory transfer, send data between processes using sockets 

Slides courtesy Kai Rothauge, UC Berkeley

# Truncated SVD Alchemist vs Pure Spark

- Use Alchemist and MLlib to get rank 20 truncated SVD
- Setup:
  - 30 KNL nodes, 96GB DDR4, 16GB MCDRAM
  - Spark: 22 nodes; Alchemist: 8 nodes
  - A: m-by-10K, where m = 5M, 2.5M, 1.25M, 625K, 312.5K
  - Ran jobs for at most 60 minutes (3600 s)
- Alchemist times include data transfer




Slides courtesy Kai Rothauge, UC Berkeley

# Deep Learning in Spark with BigDL

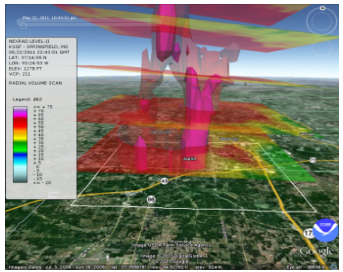
# Motivating Example: Precipitation Nowcasting

- **Problem: Predict precipitation locations and rates at a regional level over a short timeframe**
  - Neighborhood level predictions
  - T+0 – T+6 hours
- **Standard Approach: Numerical Weather Prediction**
  - Physics based simulations
  - High computational cost limits performance and accessibility
- **Cutting edge approach: Deep Learning**
  - Predict rainfall by learning from historical data
  - Heavy computation occurs ahead of time
  - Pre-Trained models can be deployed as soon as data is available

# Data Processing Pipeline

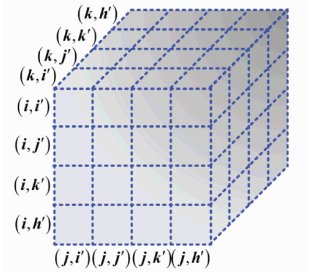

## Data Collection

- Historical Radar Data (NETCDF)
- Geographical Region (Eg:- Seattle)
- Days with over 0.1 inches of precipitation, info from NOAA – NCDC
- Radar scans every 5-10 minutes throughout the day



## Transformation

- Raw radial data structure converted to evenly spaced Cartesian grid (Tensors with float 32)
- Resolution scaling and clipping
- Configure dimensionality
- Sequencing
- 2 channels – Reflectivity, Velocity
- Uses Py-ART package




## Sampling

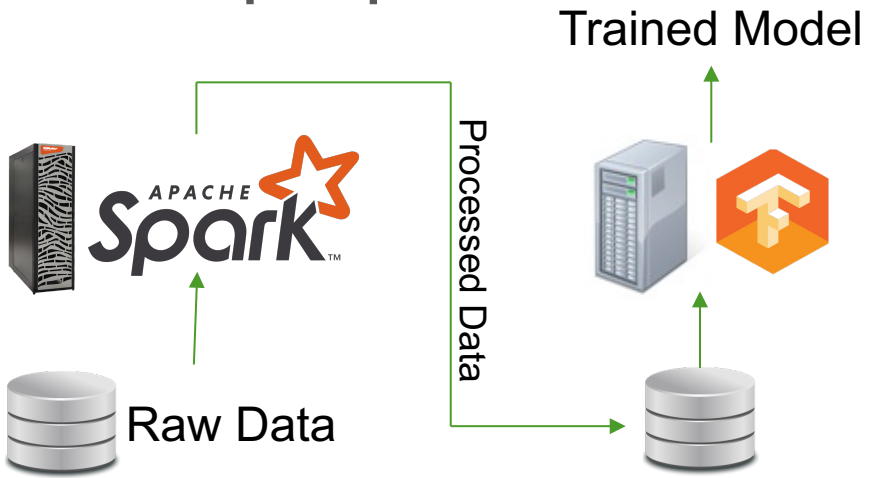
- Time-series
- Inputs and Labels
- Random sampling

intel BigDL

# Initial Implementation: Tensorflow + Spark

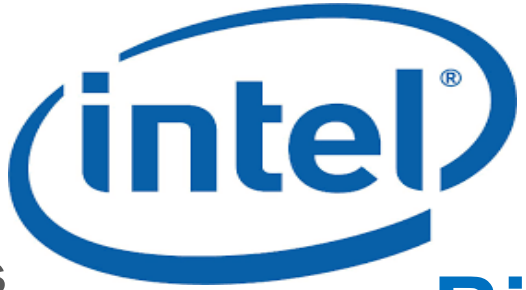
- **Separate workflows – no integration**
  - Forced overhead – data movement
  - Distinct data pipelines
- **Data processing – highly distributed analytics platform**
- **DL Training implementation – dense compute platform**

- **Pro:**
  - Specialized hardware
  - good individual performance
- **Con:**
  - Productivity loss
  - Fragmented workflow



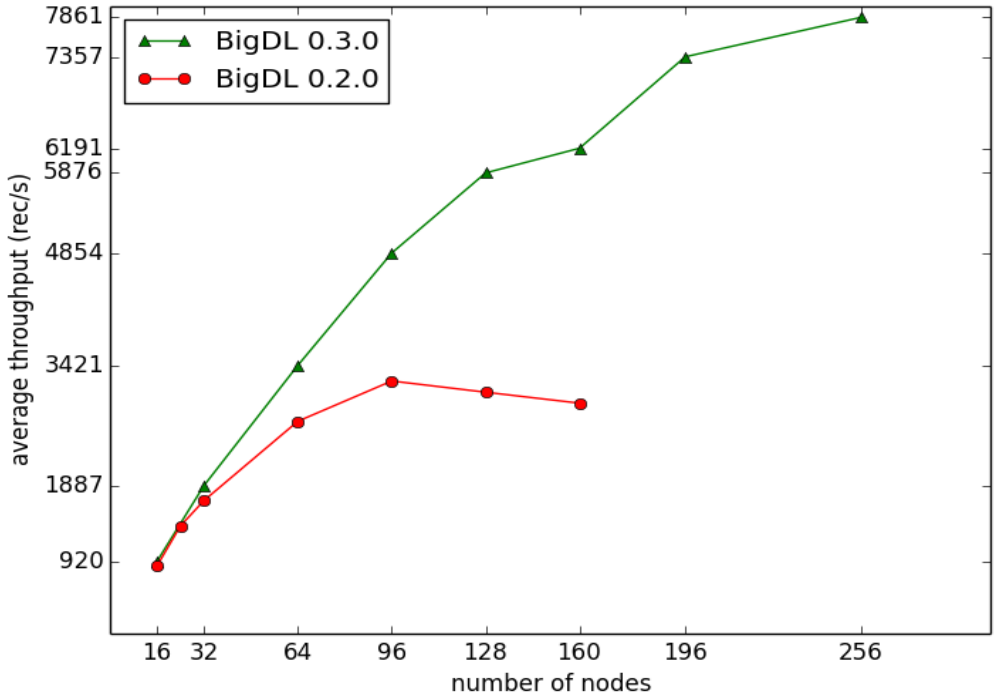
# New Implementation: Intel BigDL

- **Distributed Deep Learning Library**
- **Natively integrated with Spark**
  - Single Spark Context
  - Dataset stays in memory
  - Effortless distributed training
- **Optimized with MKL-DNN libraries**
- **Interface similar to Torch**
  - Stacked NN layers
  - Define a very complex model in very few lines
- **Quickly integrate Deep Learning and Machine Learning into Spark-based data analytics workloads**



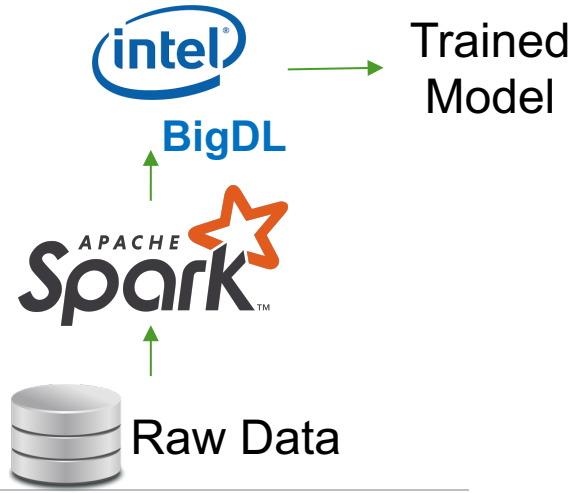

BigDL

# BigDL Training Scaling



# Implementation: BigDL on Spark

- **Singular workflow**
  - Data processing on spark flows directly into the training process with BigDL
- **HPC scale with Urika-XC**
  - High performance compute nodes excel at data analytics
  - MKL, MKL-DNN provide suitable optimization for DL workloads
  - Suite of analytics tools to aid in development
- **Pros:**
  - Single platform
  - Highly productive development environment
  - Effortless distribution
- **Cons:**
  - Less flexible expressive Deep Learning tools
  - Less flexible compute environment



# Demo: BigDL MNIST

# Legal Disclaimer

*Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.*

*Cray Inc. may make changes to specifications and product descriptions at any time, without notice.*

*All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.*

*Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.*

*Cray uses codenames internally to identify products that are in development and not yet publicly announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.*

*Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.*

*The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA and YARCDATA. The following are trademarks of Cray Inc.: CHAPEL, CLUSTER CONNECT, CLUSTERSTOR, CRAYDOC, CRAYPAT, CRAYPORT, DATAWARP, ECOPHLEX, LIBSCI, NODEKARE, REVEAL. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used on this website are the property of their respective owners.*

# Q & A

James Maltby, Cray Inc.
jmaltby@cray.com