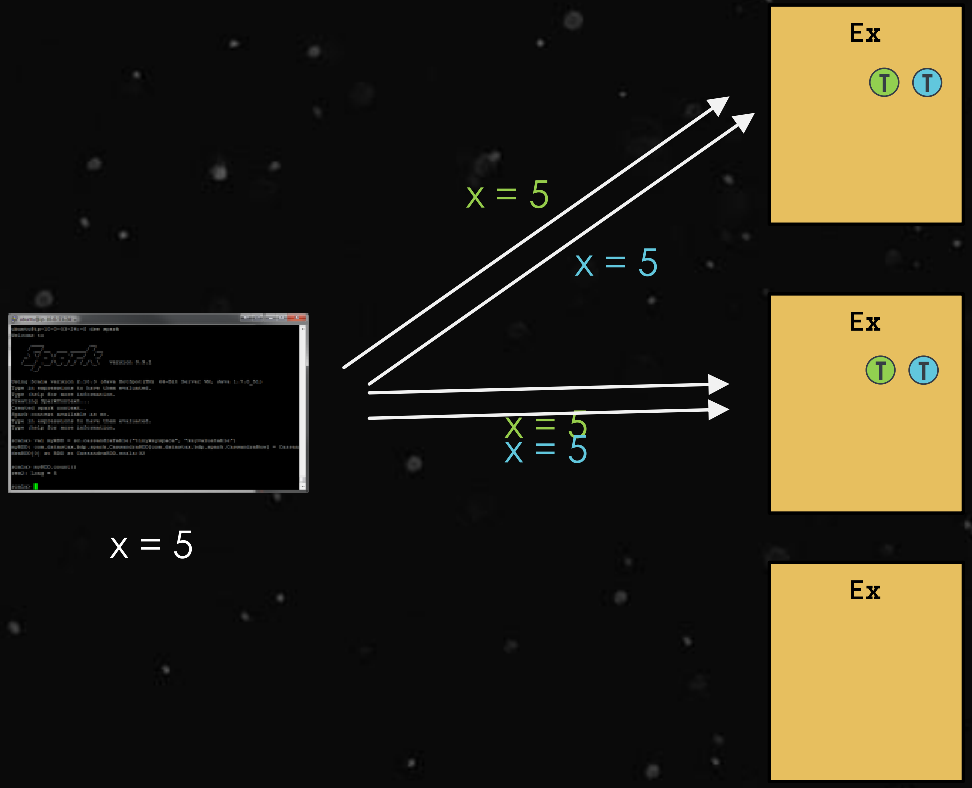
BROADCAST VARIABLES

```
[illegible]
Welcome to
      Spark  version [illegible]

[illegible]
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala> [illegible]
[illegible]
[illegible]

scala> [illegible]
[illegible]

scala>
```

& ACCUMULATORS

databricks


x = 5
x = 5
x = 5
x = 5
x = 5
Ex
T
T
Ex
T
T
Ex

USE CASES:

- Broadcast variables – Send a large read-only lookup table to all the nodes, or send a large feature vector in a ML algorithm to all nodes

- Accumulators – count events that occur during job execution for debugging purposes. Example: How many lines of the input file were blank? Or how many corrupt records were in the input dataset?

Spark supports 2 types of shared variables:

- Broadcast variables – allows your program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. Like sending a large, read-only lookup table to all the nodes.

- Accumulators – allows you to aggregate values from worker nodes back to the driver program. Can be used to count the # of errors seen in an RDD of lines spread across 100s of nodes. Only the driver can access the value of an accumulator, tasks cannot. For tasks, accumulators are write-only.

BROADCAST VARIABLES

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

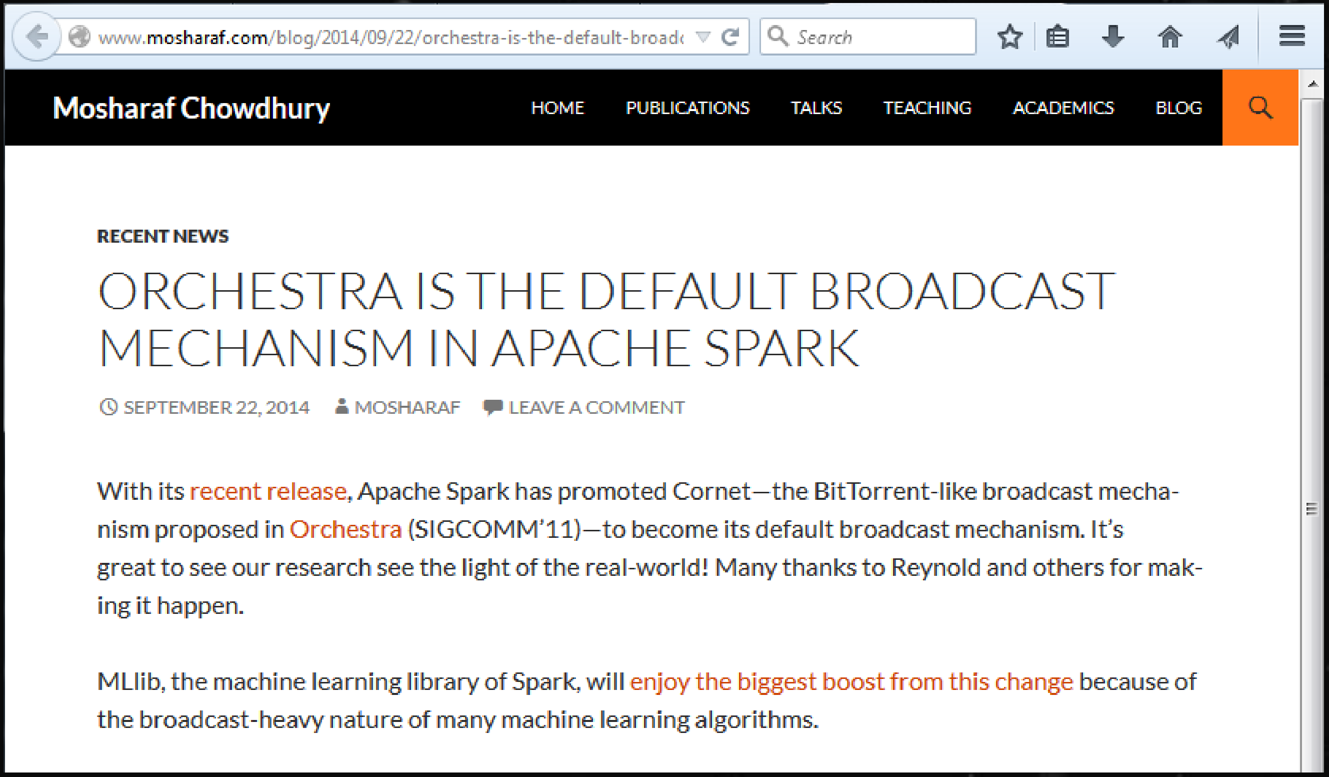
BROADCAST VARIABLES

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```



RECENT NEWS

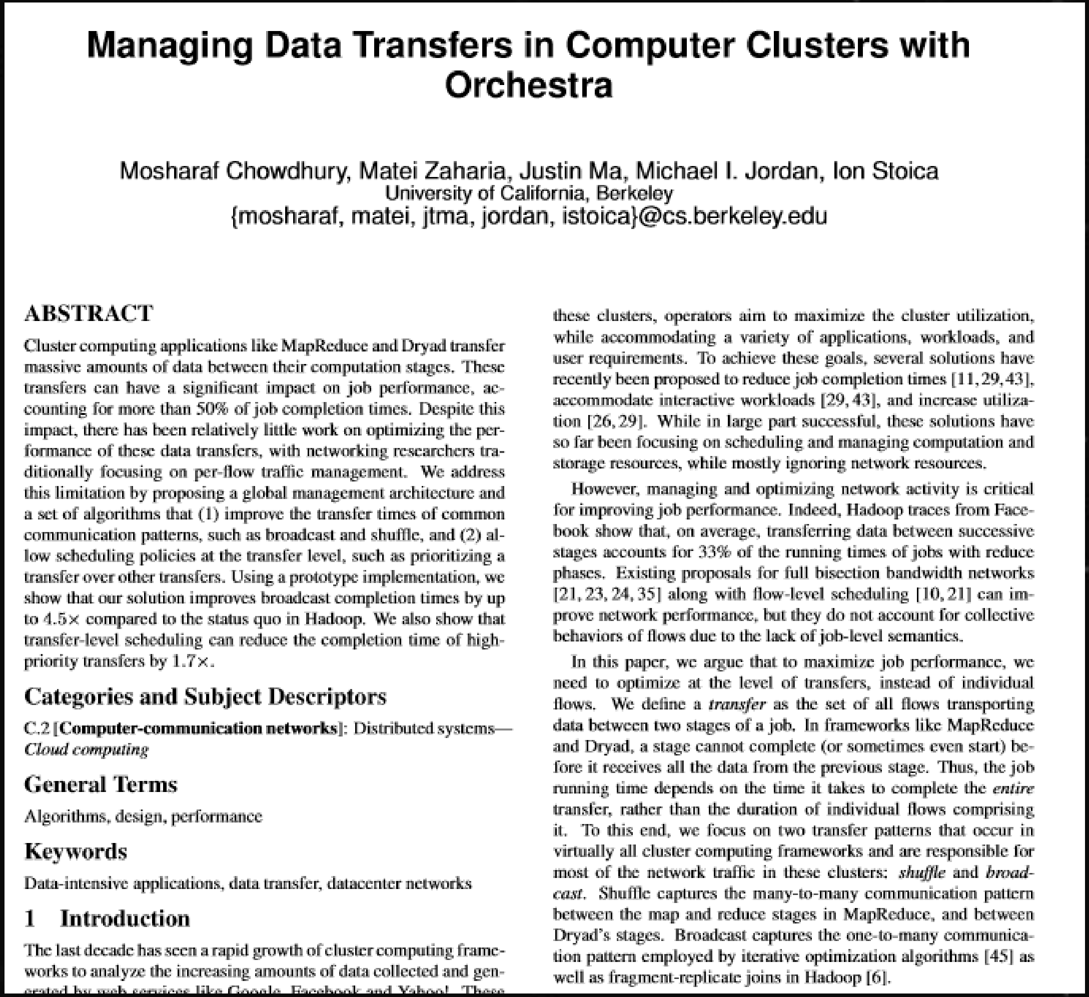
ORCHESTRA IS THE DEFAULT BROADCAST MECHANISM IN APACHE SPARK

SEPTEMBER 22, 2014 MOSHARAF LEAVE A COMMENT

With its recent release, Apache Spark has promoted Cornet—the BitTorrent-like broadcast mechanism proposed in Orchestra (SIGCOMM'11)—to become its default broadcast mechanism. It's great to see our research see the light of the real-world! Many thanks to Reynold and others for making it happen.

MLlib, the machine learning library of Spark, will enjoy the biggest boost from this change because of the broadcast-heavy nature of many machine learning algorithms.

Link



Managing Data Transfers in Computer Clusters with Orchestra

Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, Ion Stoica
University of California, Berkeley
{mosharaf, matei, jtma, jordan, istoica}@cs.berkeley.edu

ABSTRACT

Cluster computing applications like MapReduce and Dryad transfer massive amounts of data between their computation stages. These transfers can have a significant impact on job performance, accounting for more than 50% of job completion times. Despite this impact, there has been relatively little work on optimizing the performance of these data transfers, with networking researchers traditionally focusing on per-flow traffic management. We address this limitation by proposing a global management architecture and a set of algorithms that (1) improve the transfer times of common communication patterns, such as broadcast and shuffle, and (2) allow scheduling policies at the transfer level, such as prioritizing a transfer over other transfers. Using a prototype implementation, we show that our solution improves broadcast completion times by up to 4.5× compared to the status quo in Hadoop. We also show that transfer-level scheduling can reduce the completion time of high-priority transfers by 1.7×.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Distributed systems—*Cloud computing*

General Terms

Algorithms, design, performance

Keywords

Data-intensive applications, data transfer, datacenter networks

1 Introduction

The last decade has seen a rapid growth of cluster computing frameworks to analyze the increasing amounts of data collected and generated by web services like Google, Facebook and Yahoo!. These clusters, operators aim to maximize the cluster utilization, while accommodating a variety of applications, workloads, and user requirements. To achieve these goals, several solutions have recently been proposed to reduce job completion times [11, 29, 43], accommodate interactive workloads [29, 43], and increase utilization [26, 29]. While in large part successful, these solutions have so far been focusing on scheduling and managing computation and storage resources, while mostly ignoring network resources.

However, managing and optimizing network activity is critical for improving job performance. Indeed, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs with reduce phases. Existing proposals for full bisection bandwidth networks [21, 23, 24, 35] along with flow-level scheduling [10, 21] can improve network performance, but they do not account for collective behaviors of flows due to the lack of job-level semantics.

In this paper, we argue that to maximize job performance, we need to optimize at the level of transfers, instead of individual flows. We define a *transfer* as the set of all flows transporting data between two stages of a job. In frameworks like MapReduce and Dryad, a stage cannot complete (or sometimes even start) before it receives all the data from the previous stage. Thus, the job running time depends on the time it takes to complete the *entire* transfer, rather than the duration of individual flows comprising it. To this end, we focus on two transfer patterns that occur in virtually all cluster computing frameworks and are responsible for most of the network traffic in these clusters: *shuffle* and *broadcast*. Shuffle captures the many-to-many communication pattern between the map and reduce stages in MapReduce, and between Dryad's stages. Broadcast captures the one-to-many communication pattern employed by iterative optimization algorithms [45] as well as fragment-replicate joins in Hadoop [6].

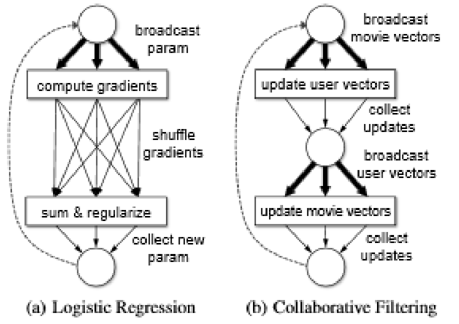


Figure 2: Per-iteration work flow diagrams for our motivating machine learning applications. The circle represents the master node and the boxes represent the set of worker nodes.

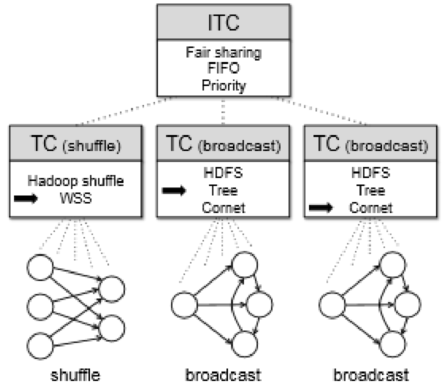
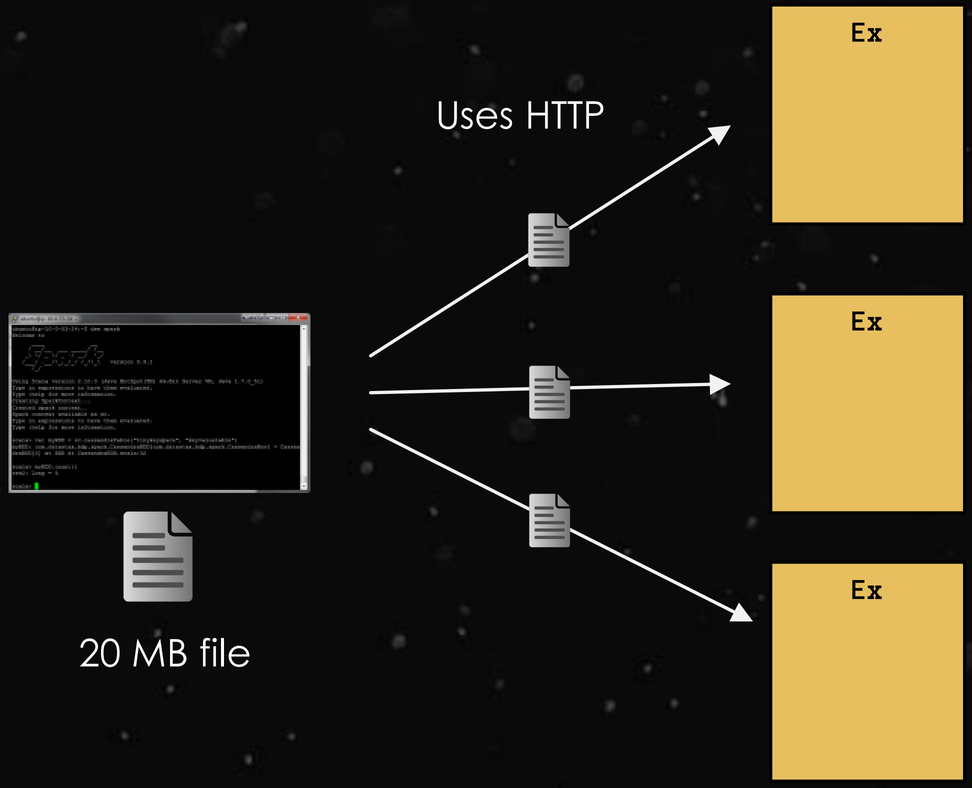
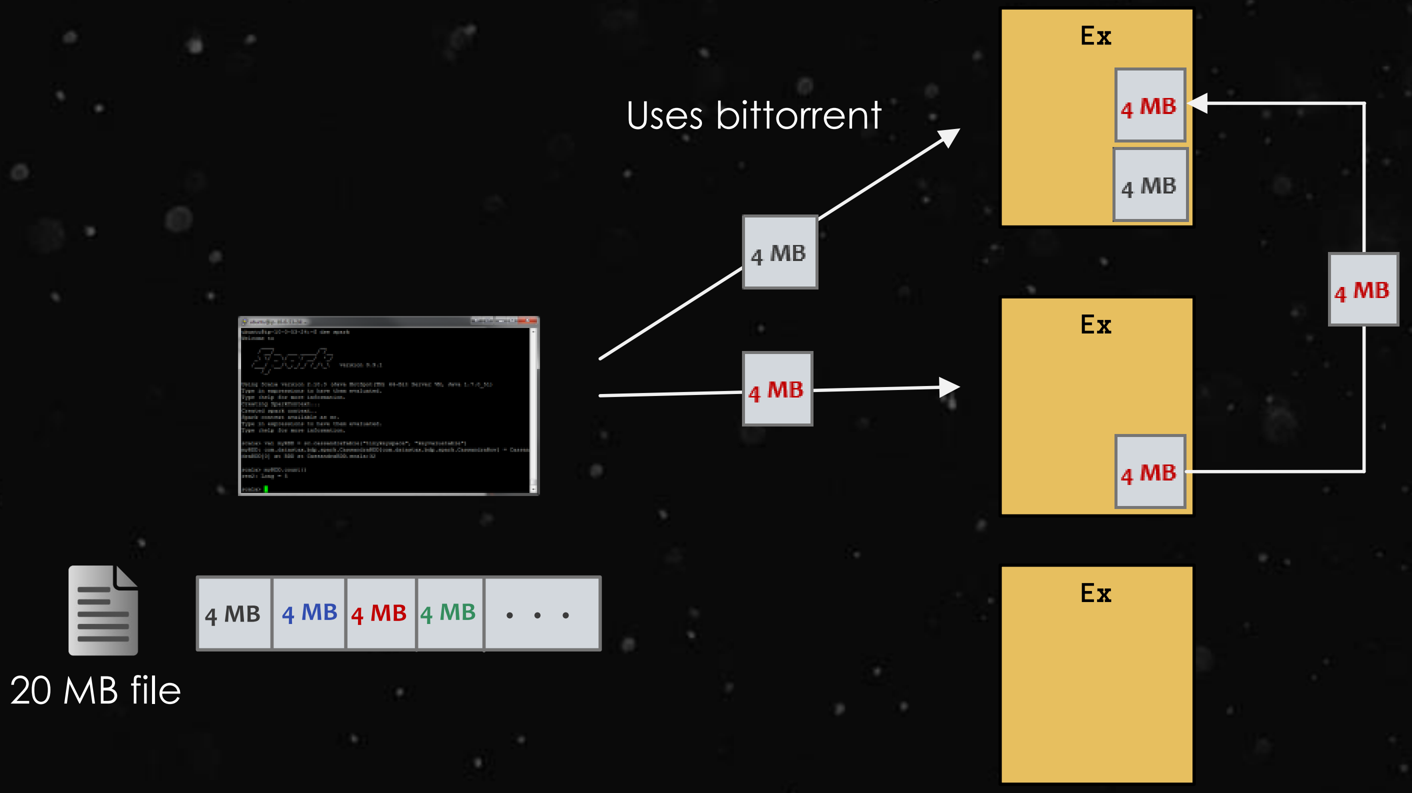


Figure 4: Orchestra architecture. An Inter-Transfer Controller (ITC) manages Transfer Controllers (TCs) for the active transfers. Each TC can choose among multiple transfer mechanisms depending on data size, number of nodes, and other factors. The ITC performs inter-transfer scheduling.

History: OLD TECHNIQUE FOR BROADCAST

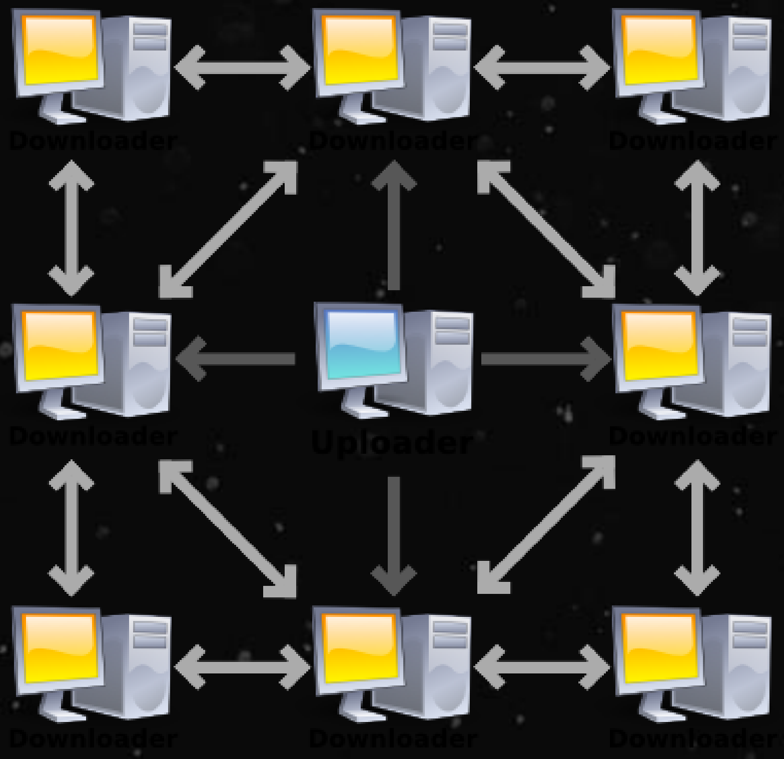


BITTORENT TECHNIQUE FOR BROADCAST



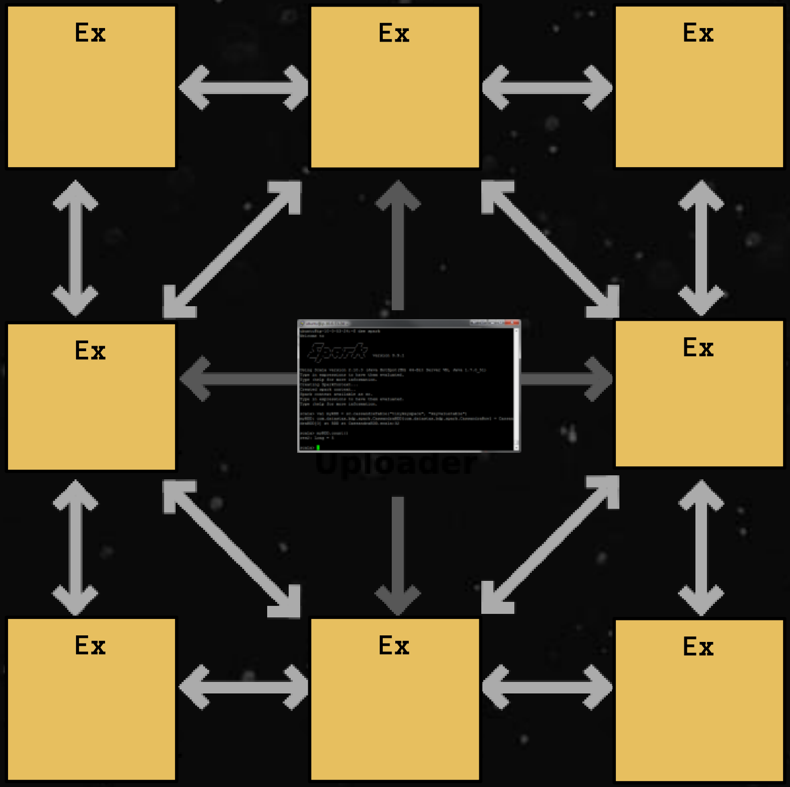
```
[illegible]
Welcome to
[illegible]                 version [illegible]

[illegible]
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala> [illegible]
[illegible]
[illegible]

scala> [illegible]
[illegible]

scala>
```

BITTORENT TECHNIQUE FOR BROADCAST



Source: Scott Martin

BITTORENT TECHNIQUE FOR BROADCAST



```
[illegible]
Welcome to
      Spark   version [illegible]

[illegible]
Type in expressions to have them evaluated.
Type :help for more information.
[illegible]
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala> val myRDD = sc.cassandraTable([illegible])
[illegible]
[illegible]

scala> myRDD.count()
[illegible]

scala>
```

ACCUMULATORS

Accumulators are variables that can only be "added" to through an *associative* operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator's value, not the tasks

ACCUMULATORS

Scala:

```
val accum = sc.accumulator(0)

sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
   global accum
   accum += x

rdd.foreach(f)

accum.value
```