CS 378 – Big Data Programming

Lecture 24

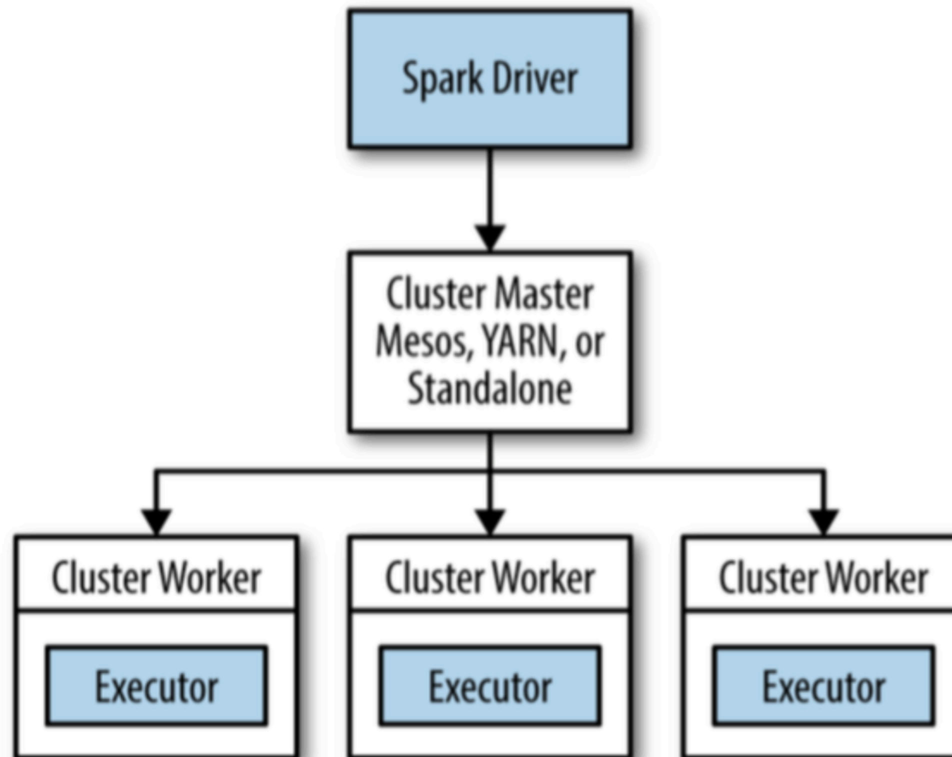
Closures, Caching, Partitions

Review

- Assignment 11: Inverted index in Spark
- Implementation
- Extra credit
  - Approach 1
  - Approach 2

Distributed Spark Application

Learning Spark, Figure 7-1



Distributing a Spark Application

- Spark Driver runs your `main()` method
  - Converts Spark program into tasks
  - Creates an execution plan based on DAG
    - DAG is derived from transformations
  - Performs optimization (like: pipelining map()'s)

- Task are bundled up to be sent to cluster
  - Cluster has multiple task executors

Distributing a Spark Application

- Scheduling individual tasks
  - Executors register with driver
  - Tasks scheduled based on data location
  - Cached data is tracked (for future task scheduling)

- Driver exposes data on task status

Distributing a Spark Application

- With Hadoop the JAR was sent to workers
  - Spark also needs to get the code to workers
- Hadoop has two tasks: map, reduce
  - Instantiation takes place on the workers
- Spark sends *object instances* to workers
  - Individual tasks defined in your Spark code
  - Objects are serialized (we use Java serialization)

Closures

- Functions as *first class objects*
  - Can be passed to a function as an argument
  - Can be returned from a function
  - Can be assigned to variables

- Closures contain free variables that are bound in the lexical environment/scope

Closures

- In Scala, functions as a type are built-in

- In Java, closures are anonymous inner classes
  - Define an object that implements an interface
  - Interface requires implementation of an abstract method
  - In Spark API, that method is `call()`

Closures

- Our Java functions are:
  - Instantiated
  - Sent off to the worker tasks (via serialization)
  - Each task gets its own copy (no communication)

- Non-local references will cause containing object to be serialized as well.
  - Variable value types must be serializable

Closures – Issues in Java

- A function references a method in an enclosing scope
  - Method itself cannot be serialized
  - The entire containing class must be serialized
- Issues
  - This class is not serializable
  - The associated data might be large

Persistence

- Recall that RDDs are recomputed as needed
  - An action initiates evaluation
  - Additional action results in another evaluation

- An RDD can be persisted for efficiency
- Making an RDD persistent:
  - `cache()`
  - `persist(StorageLevel level)`

Persistence Options

From: http://training.databricks.com/workshop/itas_workshop.pdf

transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

Partitioning

- Prudent partitioning can greatly reduce the amount of communication (shuffle)

- If an RDD is scanned only once, no need
- If an RDD is reused multiple times in key-oriented operations
  - Partitioning can improve performance significantly

Partitioning

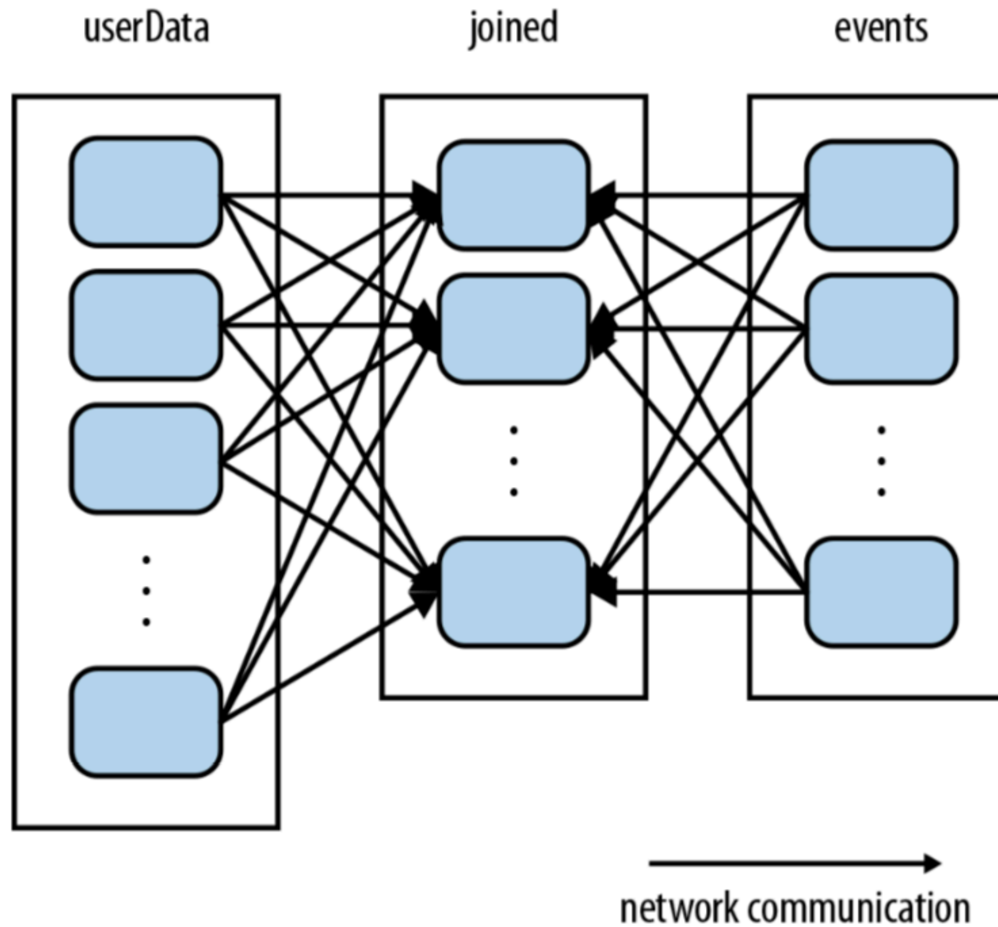
- Partitioning on pair RDDs (key, value)

- Consider an RDD containing user sessions
  – All users over some time period (day or week)
  – We want to merge in the last hour of events

- We'll be joining sessions and events by userID

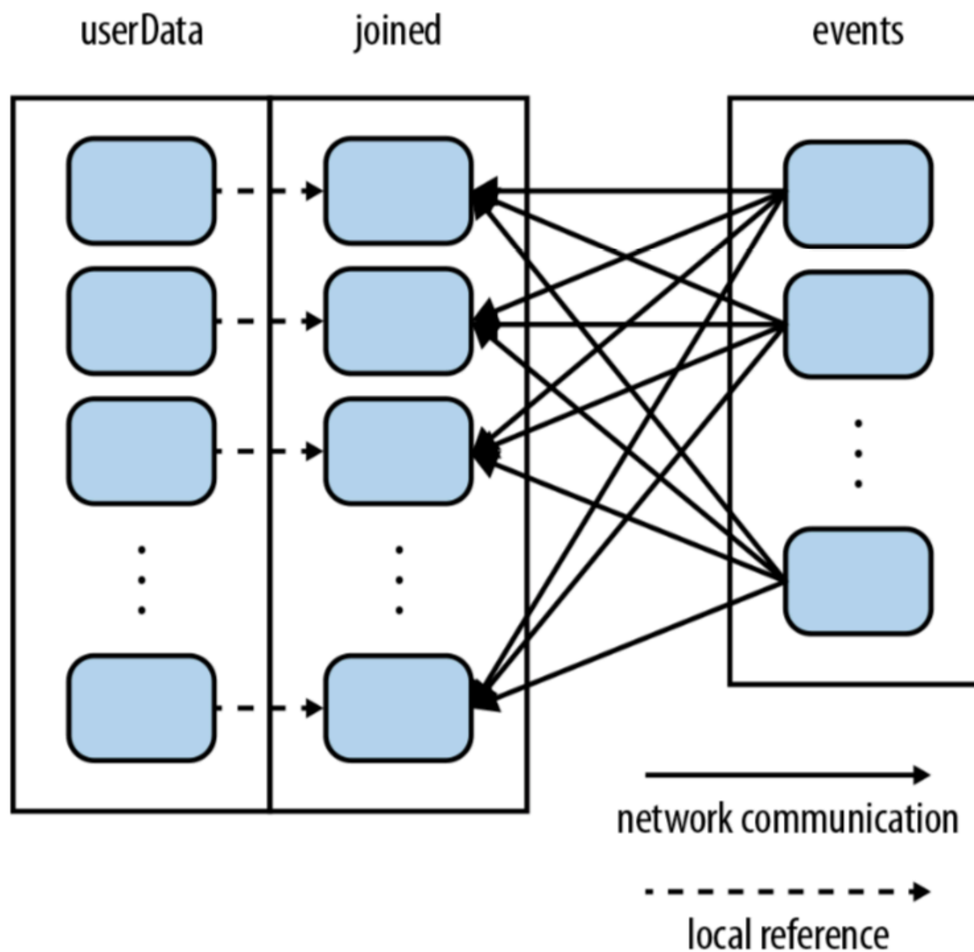
Partitioning

Figure 4-4, from Learning Spark



Partitioning

Figure 4-5, from Learning Spark



Partitioning

- Consider an RDD containing user sessions
  - All users over some time period (day or week)
  - We want to merge events, multiple times

- To set up for this:
  - Create the session RDD
  - Partition (call `partitionBy()`, a transformation)
  - Persist

Assignment 11

- Assignment 11
  - Create user sessions
  - Order events by timestamp
  - Order sessions by user ID, referring domain
  - Partition sessions by referring domain
  - Sample SHOWER sessions (1 in 10)