CS 378 – Big Data Programming

Lecture 26

Closures, Caching, Partitions

Review

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

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

- 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

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)

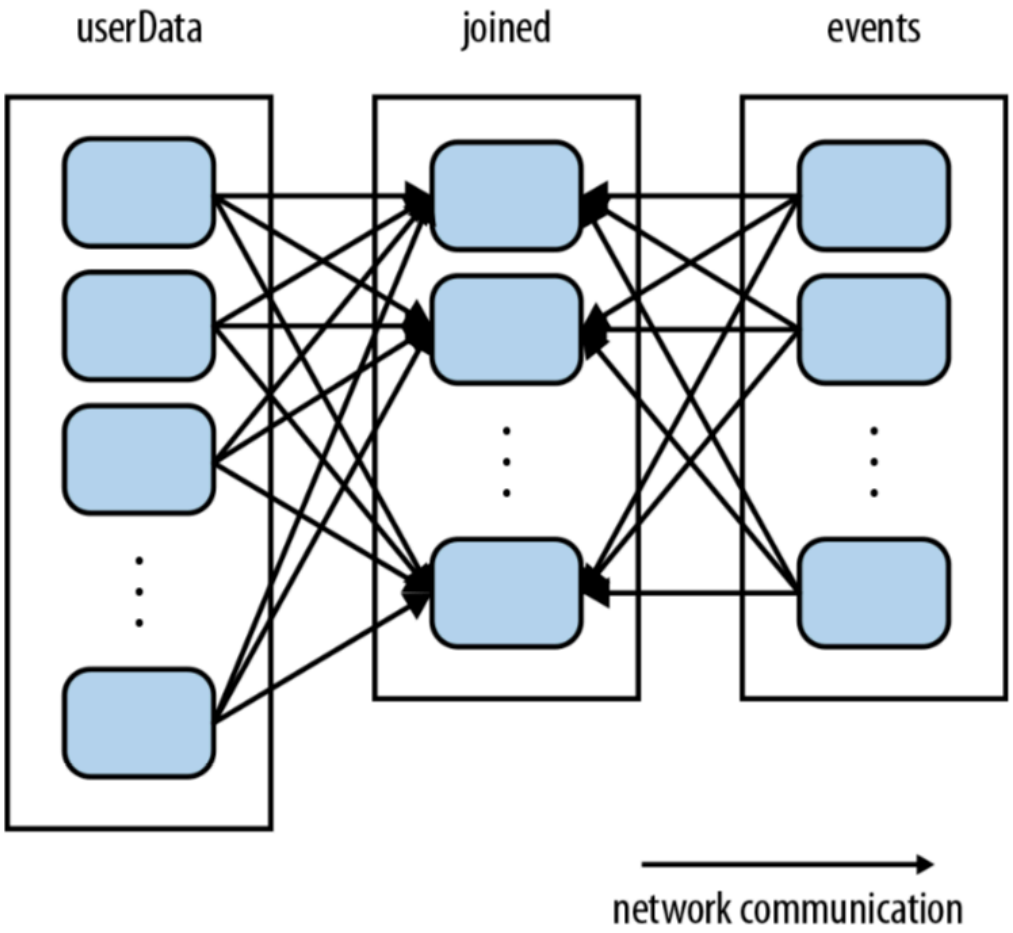
- If an RDD is scanned only once, no need
- If an RDD is reused multiple times in key-oriented operations
  - Partitioning can improve performance significantly

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

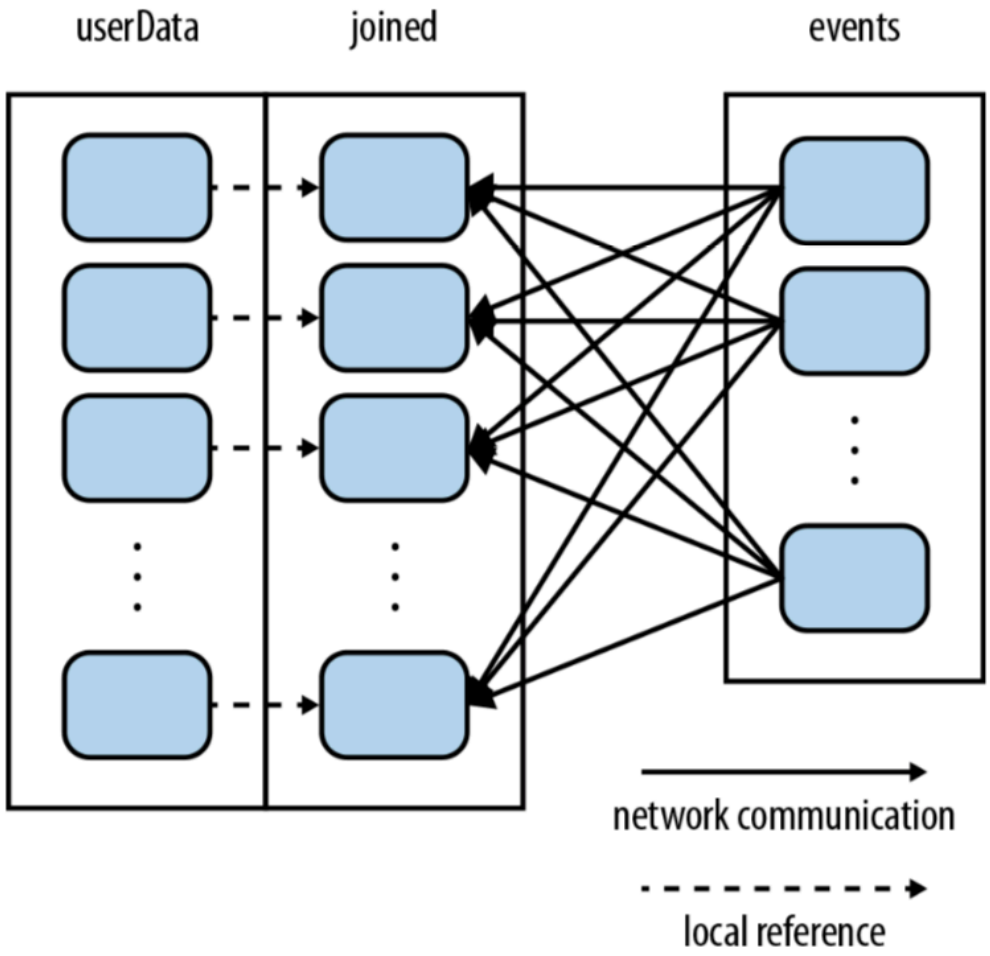
Partitioning

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 (reading from HDFS)
  - Partition (call `partitionBy()`, a transformation)
  - Persist

Partitioning

- Some transformations automatically return an RDD with known partitioning
- `sortByKey()` – range partitioned
- `groupByKey()` – hash partitioned

- Some transformations "forget" parent partitioning
  - `map()`

Benefits of Partitioning

- Many transformations shuffle data across the network
- All these will benefit from partitioning
  - `cogroup()`
  - `groupWith()`
  - `join()`
  - `leftOuterJoin()`
  - `rightOuterJoin()`

Benefits of Partitioning

- And these will benefit from partitioning
  - `groupByKey()`
  - `reduceByKey()`
  - `combineByKey()`
  - `lookup()`

Benefits of Partitioning

- Transformations on a single, partitioned RDD
  - Computed locally on a machine
  - Reduced result is sent to the master machine

- Binary transformations like `cogroup(), join()`
  - Prepartitioning will cause one RDD not to be shuffled
  - If both RDDs have the same partitioner and are on the same machine (e.g., from `mapValues()`)
  - No shuffling will occur

Partitioning

- Which partitioner is set on output?

- Depends on the parent RDDs' partitioners
- By default, hash partitioner
  - Number of partitions is the level of parallelism
- If one parent has an explicit partitioner
  - Use it
- If both have an explicit partitioner, use the first

Partitioning

- To maximize the potential for partitioning-related optimizations, instead of map() use
- `mapValues()`
- `flatMapValues()`

- Why? They preserve the key

Custom Partitioners

- Partitioners used by default:
  - `HashPartitioner`
  - `RangePartitioner`

- Custom partitioner
  - Subclass `Partitioner`
  - Implement the required methods
    - `numPartitions()`
    - `getPartition(key)`
    - `equals()`