Advanced Spark Programming (1)

Persistence (caching)

Persistence (caching)

1. *RDDs are lazily evaluated*

Persistence (caching)

1. *RDDs are lazily evaluated*
2. *RDD and its dependencies are recomputed on action*

Persistence (caching)

1. *RDDs are lazily evaluated*
2. *RDD and its dependencies are recomputed on action*
3. *We may wish to use the same RDD multiple times.*

Persistence (caching)

1. *RDDs are lazily evaluated*
2. *RDD and its dependencies are recomputed on action*
3. *We may wish to use the same RDD multiple times.*
4. *To avoid re-computing, we can persist the RDD*

Persistence (caching)

1. *If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data when needed.*

Persistence (caching)

1. *If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data when needed.*
2. *We can also replicate our data on multiple nodes if we want to be able to handle node failure without slowdown.*

Persistence (caching) - Example

Persistence (caching) - Example

1. *var nums = sc.parallelize((1 to 100000), 50)*
2.

Persistence (caching) - Example

```
var nums = sc.parallelize((1 to 100000), 50)
def mysum(itr:Iterator[Int]):Iterator[Int] = {
        return Array(itr.sum).toIterator
}

var partitions = nums.mapPartitions(mysum)
```

Persistence (caching) - Example

```
var nums = sc.parallelize((1 to 100000), 50)
def mysum(itr:Iterator[Int]):Iterator[Int] = {
        return Array(itr.sum).toIterator
}

var partitions = nums.mapPartitions(mysum)
def incrByOne(x:Int ):Int = {
    a.  return x+1;
}

var partitions1 = partitions.map(incrByOne)
partitions1.collect()
// say, partitions1 is going to be used very frequently
```

Persistence (caching) - Example

persist()

partitions1.persist()

res21: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at map at <console>:29

partitions1.getStorageLevel

res27: org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true, 1)

Persistence (caching) - Example

persist()

```
//To unpersist an RDD - before repersisting we need to unpersist
partitions1.unpersist()
```

Persistence (caching) - Example

persist()

```
//To unpersist an RDD - before repersisting we need to unpersist
partitions1.unpersist()

import org.apache.spark.storage.StorageLevel

//This persists our RDD into memrory and disk
partitions1.persist(StorageLevel.MEMORY_AND_DISK)
```

Persistence (caching) - Example

persist()

```
//To unpersist an RDD - before repersisting we need to unpersist
partitions1.unpersist()

import org.apache.spark.storage.StorageLevel

//This persists our RDD into memrory and disk
partitions1.persist(StorageLevel.MEMORY_AND_DISK)
partitions1.getStorageLevel
res2: org.apache.spark.storage.StorageLevel = StorageLevel(true, true, false, true, 1)
StorageLevel.MEMORY_AND_DISK
res7: org.apache.spark.storage.StorageLevel = StorageLevel(true, true, false, true, 1)
```

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

```
var mysl = StorageLevel(true, true, false, true, 1)
```

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

```
var mysl = StorageLevel(true, true, false, true, 1)
```

useDisk

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

var mysl = StorageLevel(true, true, false, true, 1)

useDisk **useMemory**

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

var mysl = StorageLevel(true, true, false, true, 1)

useDisk **useMemory** **useOffHeap**

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

var mysl = StorageLevel(true, true, false, true, 1)

useDisk **useMemory** **useOffHeap** **deserialized**

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

var mysl = StorageLevel(true, true, false, true, 1)

useDisk **useMemory** **useOffHeap** **deserialized** **replication**

Persistence - Understanding Storage Level

rdd.persist(storageLevel)

```
var mysl = StorageLevel(true, true, false, true, 1)
```

useDisk **useMemory** **useOffHeap** **deserialized** **replication**

```
partitions1.persist(mysl)
```

Persistence - StorageLevel Shortcuts

*partitions1.persist(StorageLevel.**MEMORY_ONLY**)*

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_ONLY**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✗	✓	✗	✓	1

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.

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_AND_DISK**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✓	✓	⊗	✓	1

Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions on disk that don't fit in memory, and read them from there when they're needed.

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_ONLY_SER**)*

useDisk	useMemory	useOffHeap	deserialized	replication
⊗	✓	⊗	⊗	1

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.

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**MEMORY_AND_DISK_SER**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✓	✓	⊗	⊗	1

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.

Persistence (caching) - StorageLevel

*partitions1.persist(StorageLevel.**DISK_ONLY**)*

useDisk	useMemory	useOffHeap	deserialized	replication
✓	⊗	⊗	⊗	1

Store the RDD partitions only on disk.

Persistence (caching) - StorageLevel

MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.

useDisk	useMemory	useOffHeap	deserialized	replication
-	-	-	-	2

Same as the levels above, but replicate each partition on two cluster nodes.

Persistence (caching) - StorageLevel

OFF_HEAP (experimental)

useDisk	useMemory	useOffHeap	deserialized	replication
⊗	⊗	✓	⊗	1

Store RDD in serialized format in Tachyon.

Which Storage Level to Choose?

Which Storage Level to Choose?

- RDD fits comfortably in memory, use MEMORY_ONLY

Which Storage Level to Choose?

- RDD fits comfortably in memory, use MEMORY_ONLY
- If not, try MEMORY_ONLY_SER

Which Storage Level to Choose?

- RDD fits comfortably in memory, use MEMORY_ONLY
- If not, try MEMORY_ONLY_SER
- Don't persist to disk unless, the computation is really expensive

Which Storage Level to Choose?

- RDD fits comfortably in memory, use MEMORY_ONLY
- If not, try MEMORY_ONLY_SER
- Don't persist to disk unless, the computation is really expensive
- For fast fault recovery, use replicated

Data Partitioning

- Layout data to minimize transfer

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once
- Helpful when dataset is reused multiple times in key-oriented operations

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once
- Helpful when dataset is reused multiple times in key-oriented operations
- Available for k-v rdds

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once
- Helpful when dataset is reused multiple times in key-oriented operations
- Available for k-v rdds
- Causes system to group elements based on key

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once
- Helpful when dataset is reused multiple times in key-oriented operations
- Available for k-v rdds
- Causes system to group elements based on key
- Spark does not give explicit control which worker node has which key

Data Partitioning

- Layout data to minimize transfer
- Not helpful if you need to scan the dataset only once
- Helpful when dataset is reused multiple times in key-oriented operations
- Available for k-v rdds
- Causes system to group elements based on key
- Spark does not give explicit control which worker node has which key
- It lets program control/ensure which set of key will appear together
  - - based on some hash value for e.g.
  - - or you could range-sort

Data Partitioning - Example - Subscriptions

Objective:

Count how many users visited a link that was not one of their subscribed topics.

UserData (subscriptions)

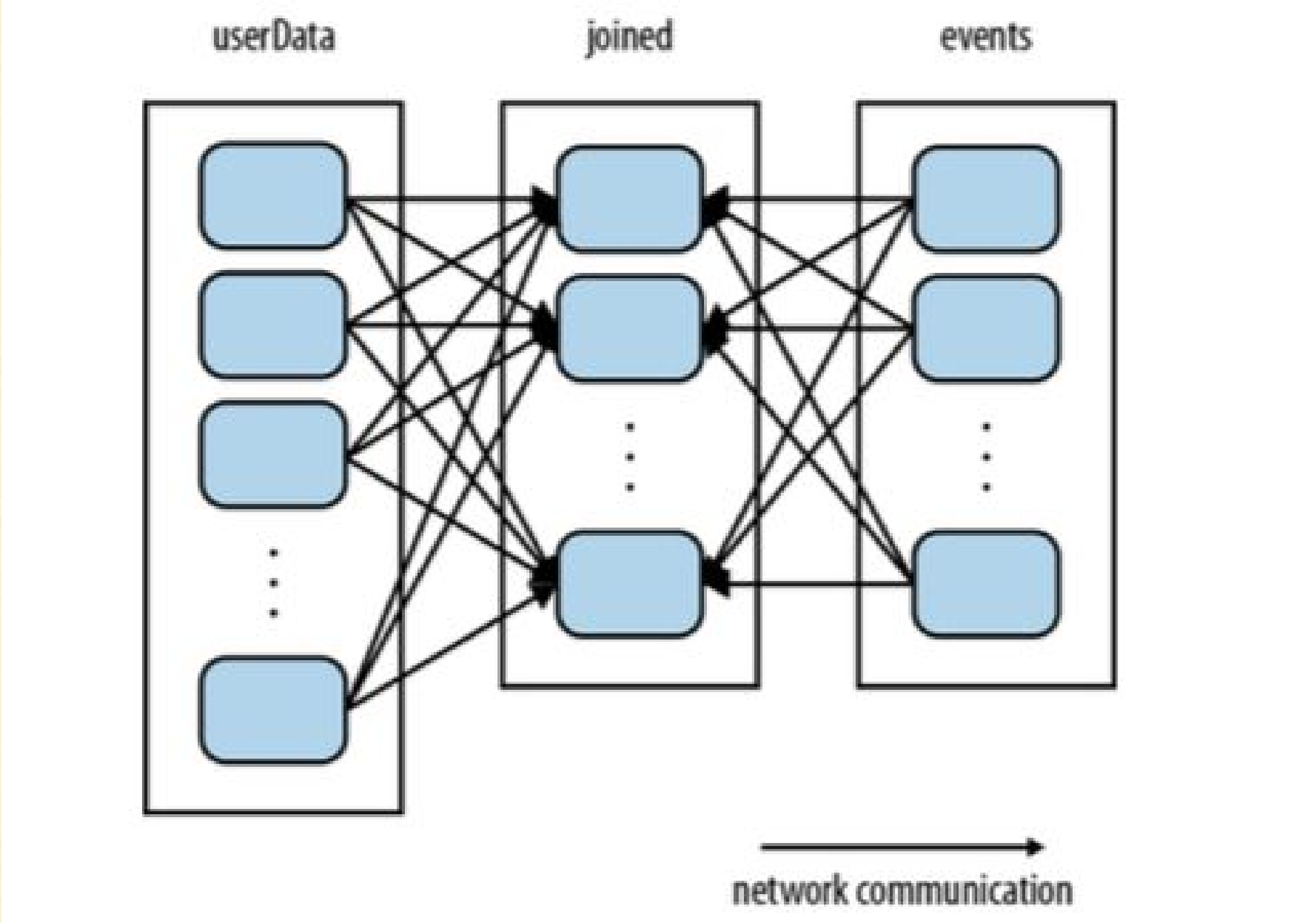
UserID	UserSubscriptionTopicsInfo

Events

UserID	LinkInfo

Data Partitioning - Example

Periodically, we combines UserData with last five mins **events** (smaller)



Code will run fine as is, but it will be inefficient. Lot of network transfer

Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 //.partitionBy( new HashPartitioner( 100))
 .persist()



// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs( logFileName: String) {
  val events = sc.sequenceFile[ UserID, LinkInfo]( logFileName)
  val joined = userData.join( events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
     case (userId, (userInfo, linkInfo)) = > !userInfo.topics.contains( linkInfo.topic)
  }.count()
  println(" Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 .persist()
```

Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 .persist()



// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs( logFileName: String) {
```

Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 .persist()


// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs( logFileName: String) {
  val events = sc.sequenceFile[ UserID, LinkInfo]( logFileName)
```

Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 .persist()


// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs( logFileName: String) {
  val events = sc.sequenceFile[ UserID, LinkInfo]( logFileName)
  val joined = userData.join( events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
```

Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 .persist()


// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs( logFileName: String) {
  val events = sc.sequenceFile[ UserID, LinkInfo]( logFileName)
  val joined = userData.join( events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
     case (userId, (userInfo, linkInfo)) = > !userInfo.topics.contains( linkInfo.topic)
  }.count()
```

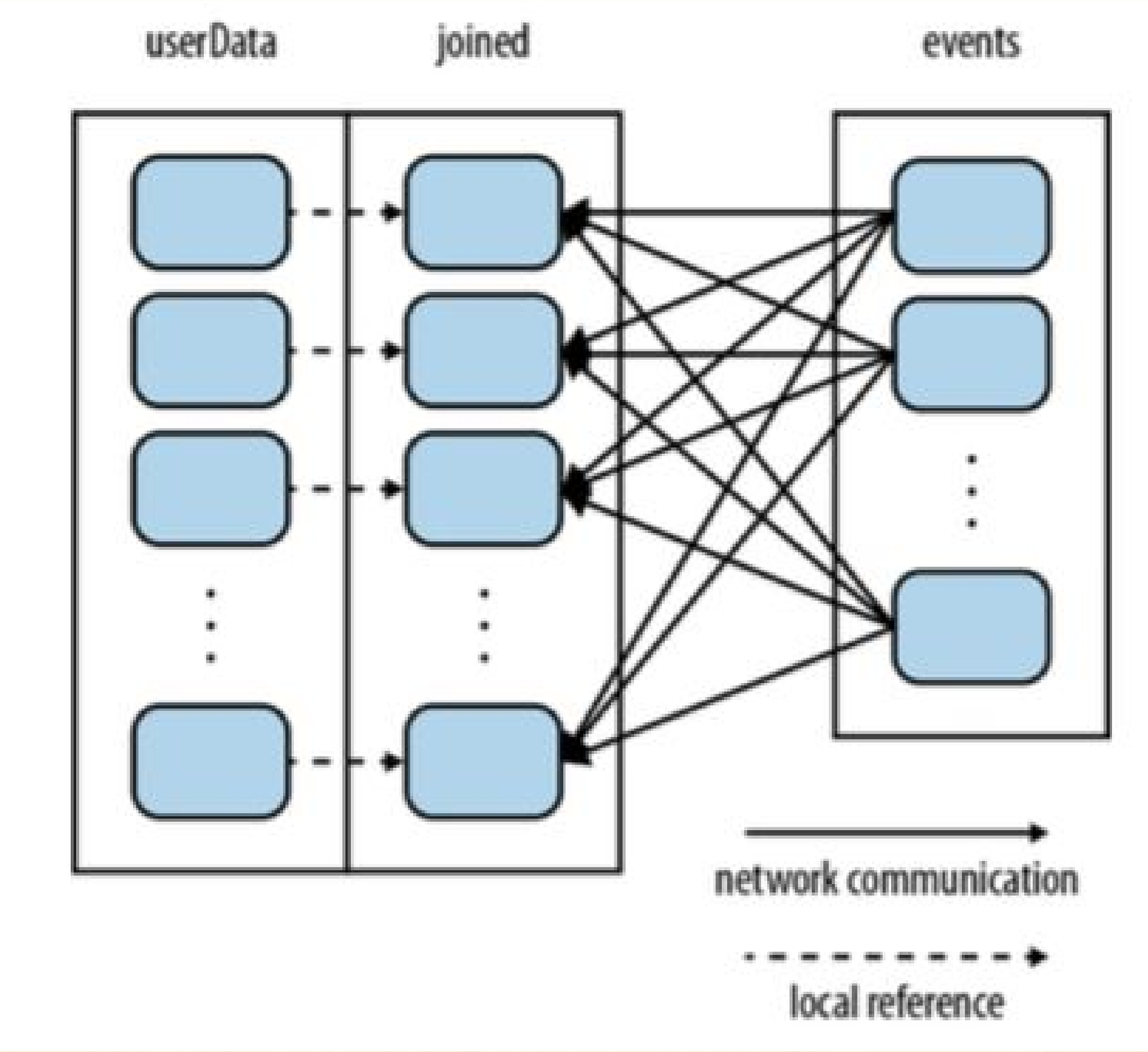
Data Partitioning - Example

```
val sc = new SparkContext(...)
val userData = sc
 .sequenceFile[ UserID, UserInfo](" hdfs://...")
 //.partitionBy( new HashPartitioner( 100))
 .persist()



// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs( logFileName: String) {
  val events = sc.sequenceFile[ UserID, LinkInfo]( logFileName)
  val joined = userData.join( events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
     case (userId, (userInfo, linkInfo)) = > !userInfo.topics.contains( linkInfo.topic)
  }.count()
  println(" Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

Data Partitioning - Example



Data Partitioning - Example

```
import org.apache.spark.HashPartitioner

sc = SparkContext(...)
basedata = sc.sequenceFile("hdfs://...")
userData = basedata.partitionBy(new HashPartitioner(8)).persist()
```

Data Partitioning - Partitioners

1. Reorganize the keys of a PairRDD
2. Can be applied using partionBy
3. Examples
    a. HashPartitioner
    b. RangePartitioner

Data Partitioning - HashPartitioner

- Implements hash-based partitioning using Java's Object.hashCode.
- Does not work if the key is an array

Data Partitioning - HashPartitioner

- Partitions sortable records by range into roughly equal ranges
- The ranges are determined by sampling

Data Re-partitioning - Example

```
scala> var lr = sc.parallelize(1 to 1000, 5)
lr: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[4] at pa

scala> lr.partitions.length
res13: Int = 5

scala> var result = lr.repartition(10)
result: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[8] at r

scala> result.partitions.length
res14: Int = 10
```

Data Partitioning

Operations That Benefit from Partitioning

- cogroup()
- groupWith(), groupByKey(), reduceByKey(),
- join(), leftOuterJoin(), rightOuter Join()
- combineByKey(), and lookup().

Data Partitioning - Creating custom partitioner

Partition 1 Partition 2 Partition 3

[[*(sandeep,1)*], [*(giri,1), (abhishek,1)*], [*(sravani,1), (jude,1)*]]

Partition 1 Partition 2

[[*(giri,1), (abhishek,1), (jude,1)*], [*(sravani,1), (sandeep,1)*]]

Data Partitioning - Creating custom partitioner

```
import org.apache.spark.Partitioner

class TwoPartsPartitioner(override val numPartitions: Int) extends Partitioner {

    def getPartition(key: Any): Int = key match {
        case s: String => {
            if (s(0).toUpper > 'J') 1 else 0
        }
    }

    override def equals(other: Any): Boolean = other.isInstanceOf[TwoPartsPartitioner]
    override def hashCode: Int = 0
}
```

https://gist.github.com/girisandeep/f90e456da6f2381f9c86e8e6bc4e8260

Data Partitioning - Creating custom partitioner

```
var x = sc.parallelize(Array(("sandeep",1),("giri",1),("abhishek",1),("sravani",1),("jude",1)), 3)
x.glom().collect()

//Array(Array((sandeep,1)), Array((giri,1), (abhishek,1)), Array((sravani,1), (jude,1)))
//[ [(sandeep,1)], [(giri,1), (abhishek,1)], [(sravani,1), (jude,1)] ]

var y = x.partitionBy(new TwoPartsPartitioner(2))
y.glom().collect()
//Array(Array((giri,1), (abhishek,1), (jude,1)), Array((sandeep,1), (sravani,1)))
//[ [(giri,1), (abhishek,1), (jude,1)], [(sandeep,1), (sravani,1)] ]
```

https://gist.github.com/girisandeep/f90e456da6f2381f9c86e8e6bc4e8260