빅데이터 분석 및 응용

L06: Spark (2)

Summer 2020

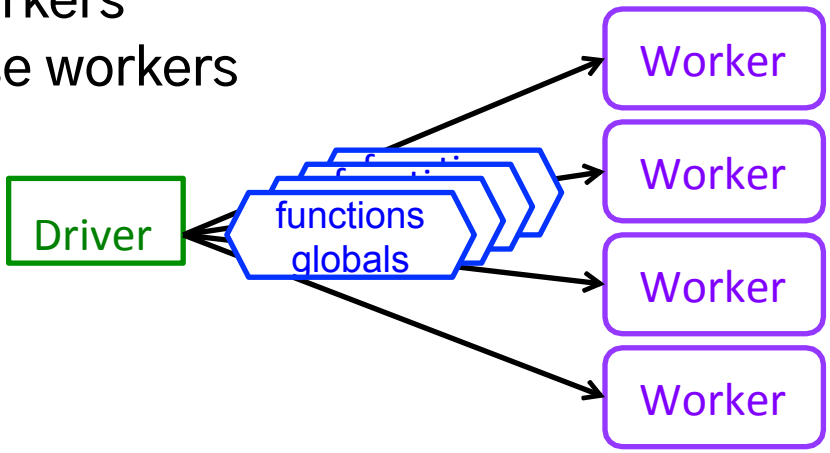
Kookmin University

Outline

- **Closure and Shared Variables**
- RDD Persistence
- More Transformations and Actions

pySpark Closures

- Spark automatically creates closures for:
  - Functions that run on RDDs at workers
  - Any global variables used by those workers



- Closure
  - Variables and methods which must be visible for the executor to perform its computations on the RDD
- One closure per worker
  - Sent for every task
  - No communication between workers
  - Changes to global variables at workers are not sent to driver

Consider These Use Cases

- Iterative or single jobs with large global variables
  - Sending large read-only lookup table to workers
  - Sending large feature vector in an ML algorithm to workers
- Counting events that occur during job execution
  - How many input lines were blank?
  - How many input records were corrupt?

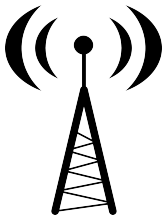
Consider These Use Cases

- Iterative or single jobs with large global variables
  - Sending large read-only lookup table to workers
  - Sending large feature vector in an ML algorithm to workers
- Counting events that occur during job execution
  - How many input lines were blank?
  - How many input records were corrupt?

Problems:

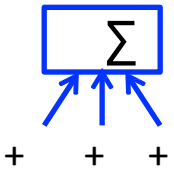
- Closures are (re-)sent with *every* job
- Inefficient to send large data to each worker
- Closures are one way: driver worker

pySpark Shared Variables



Broadcast Variables

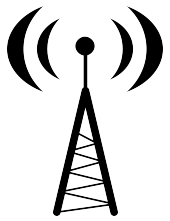
- Efficiently send large, *read-only* value to all workers
- Saved at workers for use in one or more Spark operations
- Like sending a large, read-only lookup table to all the nodes



Accumulators

- Aggregate values from workers back to driver
- Only driver can access value of accumulator
- For tasks, accumulators are write-only
- Use to count errors seen in RDD across workers

Broadcast Variables



- Keep *read-only* variable cached on workers
  - Ship to each worker only once instead of with each task
- Example: efficiently give every worker a large dataset
- Usually distributed using efficient broadcast algorithms

```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At a worker (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]
```

Broadcast Variables Example

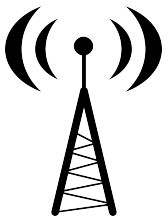
- Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Expensive to send large table
(Re-)sent for every processed file

From: http://shop.oreilly.com/product/0636920028512.do

Broadcast Variables Example



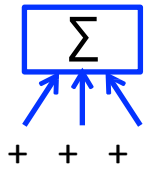
- Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Efficiently sent once to workers

From: http://shop.oreilly.com/product/0636920028512.do

Accumulators



- Variables that can only be “added” to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can read an accumulator’s value, not tasks

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

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```

Accumulators Example

- Counting empty lines

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```

Accumulators

- Tasks at workers cannot access accumulator's values
- Tasks see accumulators as write-only variables
- Accumulators can be used in actions or transformations:
  - Actions: each task's update to accumulator is **applied only once**, in re-execution (of failed or slow tasks)
  - Transformations: **no guarantees** (use only for debugging)
- Types: integers, double, long, float

Outline

- Closure and Shared Variables
- **RDD Persistence**
- More Transformations and Actions

RDD Persistence

- Spark can *persist()* (or *cache()*) a dataset in memory across operations
- Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster
- The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

RDD Persistence

- Storage level: you can do the following by controlling it
  - allows you to persist the dataset on disks
  - allows you to persist it in memory but as serialized Java objects (to save space)
  - allows you to replicate it across nodes
- Specifying storage level
  - `persist()`: takes a `StorageLevel` object as parameter
  - `cache()`: uses the default storage level (`StorageLevel.MEMORY_ONLY`) to store de-serialized objects in memory

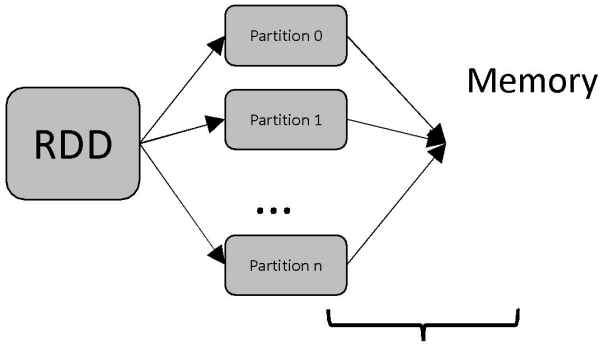
RDD Persistence

- Example in Python

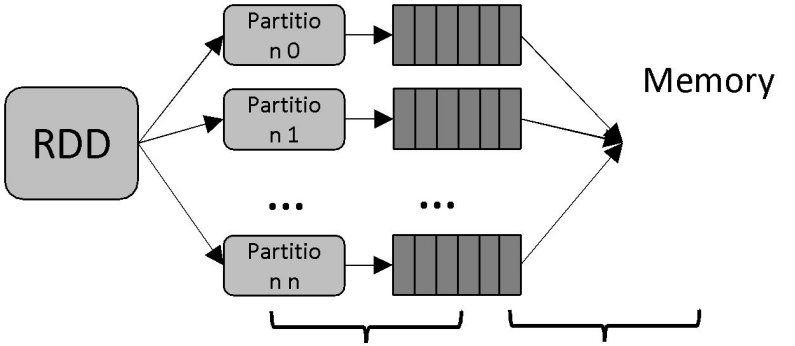
```
from operator import add
f = sc.textFile("README.md")
w = f.flatMap(lambda x: x.split(' '))
     .map(lambda x: (x, 1))
     .cache()
w.reduceByKey(add).collect()
```

RDD Persistence

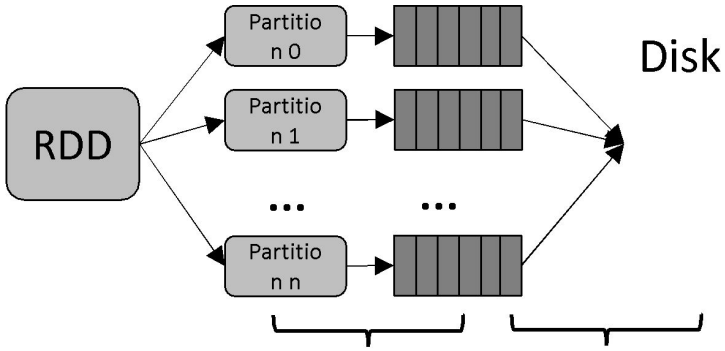
- Storage levels



(a) MEMORY_ONLY



(b) MEMORY_ONLY_SER



(c) DISK_ONLY

RDD Persistence

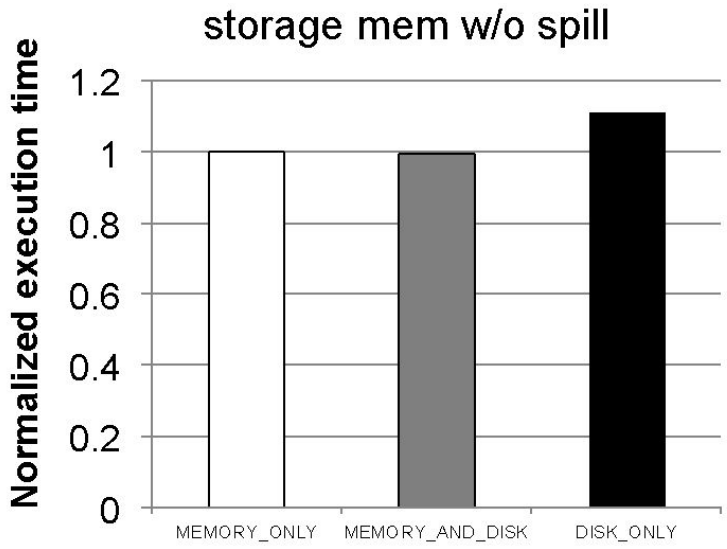
- Storage levels

Storage Level	*Meaning*
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 (Not for Python)	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 (Not for Python)	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.

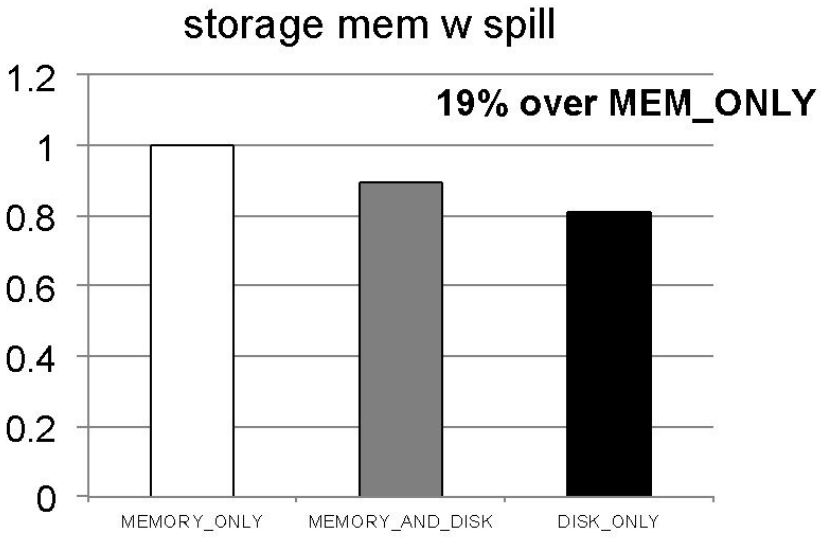
RDD Persistence

- Performance issue of caching
  - In-memory RDD caching is not always effective.
  - For iterative applications, when the cached RDD overflows the (storage) memory, MEMORY_ONLY caching causes significant overhead.

PageRank: 5,500,000 pages



PageRank: 15,000,000 pages



Outline

- Closure and Shared Variables
- RDD Persistence
- **More Transformations and Actions**

More Transformations (1)

- `mapPartitions(`*`func`*`,`
  `preservesPartitioning=False``)`
  - Return a new RDD by applying a function to each partition of this RDD.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

Initialize once for each partition

More Transformations (2)

- `mapPartitionsWithIndex(`*`func`*`, `*`preservesPartitioning=False`*`)`
  - Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(splitIndex, iterator): yield splitIndex
>>> rdd.mapPartitionsWithIndex(f).sum()
6
```

More Transformations (3)

- `sample(withReplacement, fraction, seed=None)`
  - Return a sampled subset of this RDD
  - Parameters
    - *withReplacement*: can elements be sampled multiple times (replaced when sampled out)
    - *fraction*: expected size of the sample as a fraction of this RDD's size without replacement
    - *seed*: seed for the random number generator

```
>>> rdd = sc.parallelize(range(100), 4)
>>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14
True
```

More Transformations (4)

- `aggregate(zeroValue, seqOp, combOp)`
  - Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral "zero value."
  - Useful when we want to return a value of new type after aggregation
  - The first function (seqOp) can return a different result type, U, than the type T of this RDD.
    - Thus, we need one operation for merging a T into an U and one operation for merging two Us.
  - Example for computing (sum, count) of numbers

```
>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
>>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)
(10, 4)
>>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)
(0, 0)
```

More Transformations (5)

- `cartesian(otherDataset)`
  - Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements *(a, b)* where **`a`** is in **`self`** and **`b`** is in **`other`**.

```
>>> rdd = sc.parallelize([1, 2])
>>> sorted(rdd.cartesian(rdd).collect())
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

More Actions (1)

- `takeSample(withReplacement, num, [seed])`
  - Return a fixed-size sampled subset of this RDD.
  - **Note:** This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> rdd = sc.parallelize(range(0, 10))
>>> len(rdd.takeSample(True, 20, 1))
20
>>> len(rdd.takeSample(False, 5, 2))
5
>>> len(rdd.takeSample(False, 15, 3))
10
```

More Actions (2)

- `foreach(f)`
  - Applies a function ***f*** to all elements of this RDD.

```
>>> def f(x): print(x)
>>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)
```

```
1
3
2
4
5
```

Questions?