Spark Concepts

CMPT 732, Fall 2022

Spark

Another way to express computations for a cluster. In contrast to MapReduce:

- Number of stages is up to the programmer (not just "map to key/value pairs" then "group by keys and reduce").
- Spark likes keeping things in memory.
- Runs on YARN or locally (or "standalone" or on Mesos or EC2).

- Written in Scala, which compiles to the JVM.
- APIs for Scala, Python, Java, R. We will use Python. (But Scala is a beautiful language: you should learn it sometime.)
- There is an interactive shell (REPL) where you can experiment with Spark.

Spark abstracts the computation in very different places than MapReduce. The API is *much* more expressive, but exactly what is happening on the cluster can be hard to understand.

Practical result: if you keep thinking about map → shuffle → reduce, you'll find dealing with Spark difficult.

Suggestion: stop thinking (for the moment) about how Spark produces result at all: use the Spark API (smartly) to express the results you want, and let Spark generate them.

If you have written functional code (Haskell, LISP, Scheme, F#, etc), then start with that mindset.

Worry about *how* they are produced as part of optimizing.

An Example

A complete Spark program. Input: file(s) with integers, one-per-line. Output: about 1% of the positive values.

```
from pyspark import SparkContext
sc = SparkContext()
lines = sc.textFile('inputs')
numbers = lines.map(int)
pos_nums = numbers.filter(lambda n: n > 0)
some_pos_nums = pos_nums.sample(fraction=0.01,
        withReplacement=False)
print(some_pos_nums.take(10))
```

RDDs

The basic class that represents data in Spark is the ***Resilient Distributed Dataset***. Basically, a collection of data (rows, elements, or however you think of them).

Can be ***partitioned*** across multiple nodes/processes. Operations can be done on partitions in parallel.

Are ***immutable***: values in a particular RDD can't change (but can be used to compute a new RDD).

Remember that they are just ordered collections (conceptually like lists) of any Python objects.

```
# RDD of strings (lines from file)
lines = sc.textFile('inputs')
# RDD of integers
numbers = lines.map(int)
# RDD of triples of integers
pairs = numbers.map(lambda n: (n, n*n, n*n*n))
```

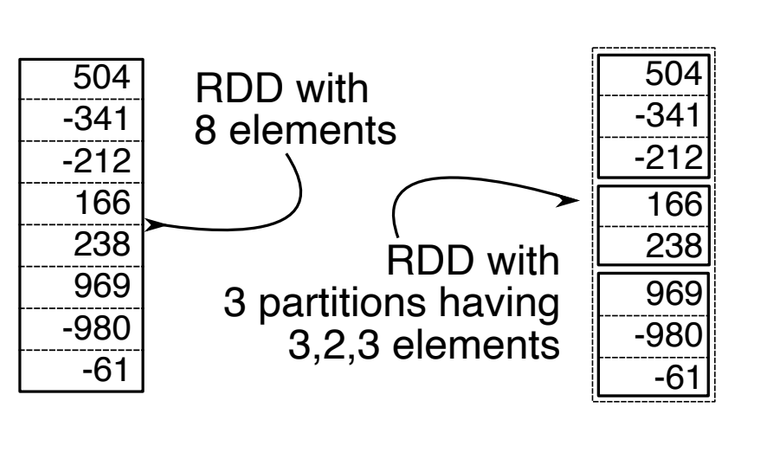
It must be possible to serialize the objects (with Python's pickle module), but that's it.

RDDs are ***partitioned*** into smaller pieces. Each partition can be on a different node.

The number of partitions controls the (maximum) amount of parallelism.

Typically, number of partitions » number of cores. Having hundreds or thousands of partitions is totally normal.

Usually we think of an RDD as a collection of elements. Sometimes we have to think about how it's partitioned.



Which is right? Both.

RDD Operations

Operations that return a new RDD: (transformations)

- `.map(f)`: result of applying `f` to each element.
- `.filter(f)`: elements where `f` returns `True`.

Operations that return a Python value: (actions)

- `.max()`/`.min()`/`.mean()`/`.count()`: largest/smallest/average/number of values.

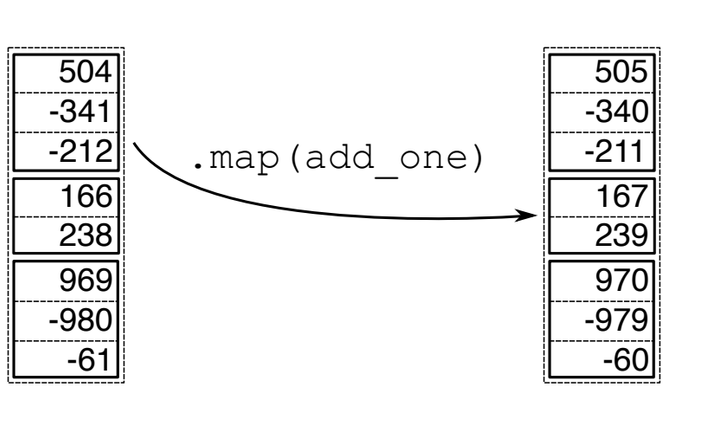
Operations that do something: (actions)

- `.saveAsTextFile(path)`: write one element per line.

There is a rich collection of operations on RDDs.

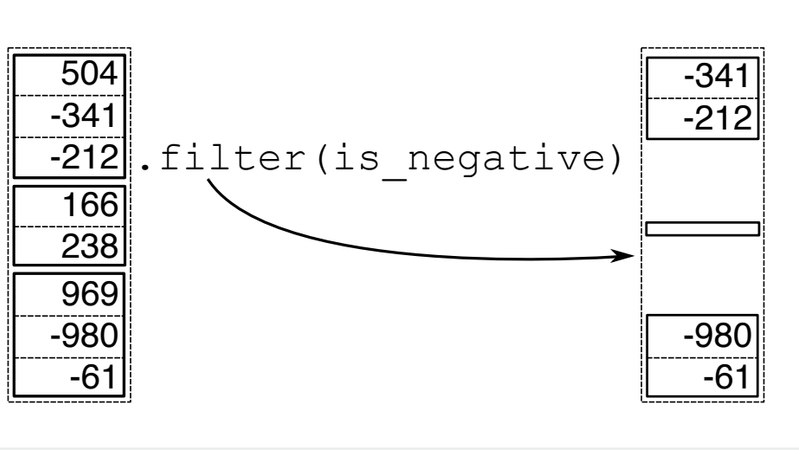
Operations and Partitions

Some transformations inherit the partitioning from the parent, e.g. `.map` and `.filter`.



These can be done completely independently on each partition, in parallel.

This can lead to unbalanced partitions.



Those that create/shuffle RDDs, you can specify/suggest:

```
data = sc.textFile('/some/path', minPartitions=1000)
groups = data.groupBy(lambda line: line[0],
         numPartitions=100)
counts = data.reduceByKey(add_pairs, numPartitions=3)
```

Or you can force the issue (with some cost):

```
data = data.repartition(500)
```

Each partition is stored on one node. That node does calculations on that partition. The partitioning matters because it controls the parallelism.

If you `.coalesce(1)`, you are no longer doing *any* work in parallel. Why use Spark for that?

Having 10^7 elements in 10^6 partitions: probably unnecessary overhead managing the work.

Partitions

We generally only get to see how things were partitioned when writing to a file.

```
values.saveAsTextFile('output')
```

Each partition turns into a file, which allows writing to be done in parallel.

You can usually ignore how your data is partitioned in most of the operations you do.

... until you hit the operation where you can't and everything is a disaster.

Lazy Evaluation

Operations on RDDs are ***lazily evaluated***.

Think of RDD objects in Python as *a plan* for calculations that can be done in the future. As you build an RDD (from input or previous RDDs), the plan (actually a "directed acyclic graph, DAG, of calculations") is constructed.

The plan isn't evaluated until you actually do something with the results...

"Do something with the results" could be:

```
some_pos_nums.collect()
```

`.collect()` turns the RDD into a Python list. Could also be another action:

```
some_pos_nums.saveAsTextFile('output-1')
```

```
s = some_pos_nums.sum()
```

("do something with the results" == "you call an ***action***".)

Chaining Calculations

Because everthing is evaluated lazily, these are identical:

```
lines = sc.textFile('inputs')
numbers = lines.map(int)
pos_nums = numbers.filter(lambda n: n > 0)
some_pos_nums = pos_nums.sample(fraction=0.01,
        withReplacement=False)
print(some_pos_nums.take(5))
```

```
print(sc.textFile('inputs')
    .map(int).filter(lambda n: n > 0)
    .sample(fraction=0.01, withReplacement=False)
    .take(5))
```

Giving an RDD a name in Python isn't meaningful. It's a question of coding style.

Also remember that you can do the exact same calculation in one step:

```
import random
def keep_some_positive(line):
    num = int(line)
    if num < 0 or random.random() < 0.99:
        return []
    else:
        return [num]

some_pos_nums = sc.textFile('inputs').flatMap(keep_some_positive)
print(some_pos_nums.take(5))
```

Speed is within about 1%. Basically the same operations are happening in the same way.

Combining Calculations

```
res1 = input_rdd.map(int)
res2 = res1.filter(lambda n: n > 0)
```

It looks like an RDD with all of the integers (`res1`) must be created before the `.filter()` is applied.

But Spark is free to do the map and filter as one operation, never creating the intermediate RDD (or even `res2`, if more work needs to be done before the end of the stage).

The actual calculation is **not** like this: (pseudocode)

```
for v in partition_of_input_rdd: # res1 = input_rdd.map(int)
    partition_of_res1.append(int(v))
for v in partition_of_res1: # res2 = res1.filter(lambda n: n > 0)
    if v > 0:
        partition_of_res2.append(v)
```

It's much more like:

```
for v in partition_of_input_rdd:
    intermediate_result = int(v)
    if intermediate_result > 0:
        partition_of_res2.append(intermediate_result)
```

The RDD `res1` we imagined never exists.

Or if you prefer, `rdd.map(f).map(g)` is more like,

> For each element, calculate $g(f(x))$ and store the results.

But is **not**,

> For each element, calculate $y = f(x)$ and store the results. Then calculate $g(y)$ and store the results.

Exactly what is happening on the cluster can be hard to understand.

Shuffle Operations

A ***shuffle*** is caused by any all-to-all operation that requires rearranging partitions.

That includes: all of the `.*ByKey()` methods, `.join()`, `.groupBy()`, `.partitionBy()`, `.repartition()`, `.sortBy()`.

The test: if you can look at one partition of the RDD and calculate it's contribution to the result, there's no shuffle.

A shuffle requires moving the RDD's data around between workers: potentially lots of network traffic, serializing/deserializing, etc.

Spark implementations are smart about it. e.g. `.reduceByKey` reduces each partition before shuffling (≈ MapReduce combiner).

Spark lets you shuffle whenever you want, but you should think about whether it's a good idea or not. Also think about how to shrink the data *before* shuffling.

e.g. `.sortBy()`? Megabyte: no problem. Gigabyte: okay. Terabyte: possible but expensive. Petabyte: no.

e.g. `.reduceByKey()` with a small number of unique keys? Almost completely parallel, so okay.

When shuffling, data is naturally repartitioned. There is some default partitioning that's usually reasonable.

```
>>> numbers = sc.textFile('/tmp/inputs').map(int)
>>> numbers.getNumPartitions()
25
>>> numbers.sortBy(lambda n: n).getNumPartitions()
25
```

Sometimes the default might not make sense, depending what you're doing.

```
>>> groups = numbers.groupBy(lambda n: n%5)
>>> groups.getNumPartitions()
25
>>> groups.count()
5
```

i.e. 5 records in 25 partitions.

Most shuffle operations have an argument where you can suggest a number of partitions for the result.

```
>>> groups = numbers.groupBy(lambda n: n%5, numPartitions=2)
>>> groups.getNumPartitions()
2
```

If specifying, give some thought to a sensible number for your data (and subsequent calculations).

Drivers & Executors

The work you do in Spark is divided across two roles: ***driver*** and ***executor***.

The driver runs your "main" logic that builds the RDD descriptions. The executor actually calculates/caches/saves the RDDs.

```
nums = sc.parallelize(range(100000), numSlices=100)
doubled = nums.map(lambda n: n*2)
total = doubled.filter(lambda n: n%4==0) \
        .reduce(lambda a,b: a+b)
print(math.sqrt(total))
```

The `range` function is called in the driver; `sc.parallelize` partitions the result and sends to executors.

The executors calculate `n*2` and `n%4==0` on each element in each partition, and then reduce each partition before sending the result back to the driver.

The driver gets an *integer* in `total` and calculates `math.sqrt`.

When running Spark with `--master=local[*]` (the default with no config), the driver and executors are each a process on your computer.

`--master=yarn --deploy-mode=client` (the default on our cluster): the driver runs **where you run the command**: necessary for `pyspark`; lets you see exceptions; less scalable for many jobs. Executors still out in the cluster.

`--master=yarn --deploy-mode=cluster`: the driver runs **on a cluster node** (the ApplicationMaster); executors run on (other) nodes.

Actions like `.collect()` and `.reduce()` send data from the (nicely parallelized) executors to the (not scalable or parallel) driver. You should think carefully before you do that.

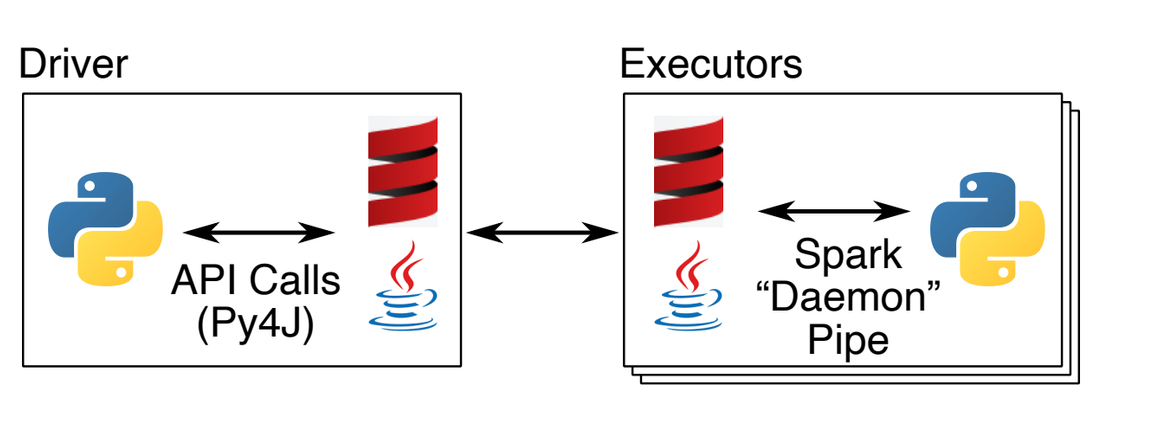
Reducing to an integer: fine. Collecting an RDD that you know has a small number of elements: fine.

Collecting without knowing something very concrete about the size of an RDD: probably dumb.

(Similar advice for `.coalesce(1)`.)

Spark is implemented in Scala on the JVM. You have been writing Python logic that has to run out on the executors.

Your function is sent out to the executors, and the RDD data is piped into it for evaluation.



* * *

Controlling Executors

On YARN, there are some command-line switches to control the number of executors and how many threads each can run:

```
--num-executors=8 --executor-cores=4 --executor-memory=4g
```

(There is also a dynamic allocation mode that is the default on our cluster.)

Spark Web Frontend

Running locally: http://localhost:4040/.

On the cluster: http://localhost:8088/ (forwarded port) → your application → ApplicationMaster. Change "controller.local" to "localhost" in the URL if off campus.

The MapReduce frontend was okay. The Spark frontend is awesome. It lives as long as the shell/application is running.

Spark vs MapReduce

The fundamental abstraction in MapReduce was the input → map → shuffle → reduce → output. We had to make calculation fit that model.

In Spark, the abstractions are in different places, usually for the better. The abstractions in Spark are usually a better fit for what I actually need to *do*.

e.g. `.reduce(f)` and `.reduceByKey(f)` both require that `f` is commutative and associative:

```
f(a,b) == f(b,a)                 or a+b == b+a
f(f(a,b), c) == f(a, f(b,c)) or (a+b)+c == a+(b+c)
```

... and Spark is free to optimize after that (by doing combiner-like things).

All reducers we have written have had those properties anyway, so it's a good trade.

In MapReduce, *everything* had to be key-value pairs. In Spark it's optional.

Want to do key-value operations? Just create an RDD of pairs `(k,v)`.

```
counts = words.map(lambda w: (w, 1))
```

There are several RDD functions that will do key-value operations in this case. Or treat them like any other RDD.

```
counts.reduceByKey(add) # only works on pair RDDs: you get values
counts.map(some_function) # works on any RDD: you get pairs
```

The Spark `.map` method applies a function to each element of an RDD and creates an RDD of the results. An RDD `r` and `r.map(anything)` have the same number of elements.

`.flatMap(f)` assumes `f` returns a list (or iterable or `yield`s several values) and puts all of the elements returned into the new RDD.

```
values = sc.parallelize([1, 2, 3, 4])
values.map(range).collect() \
        == [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3]]
values.flatMap(range).collect() \
        == [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

The `.reduce` method reduces *all* elements in the RDD to one result; `.reduceByKey` reduces the values **over each key**.

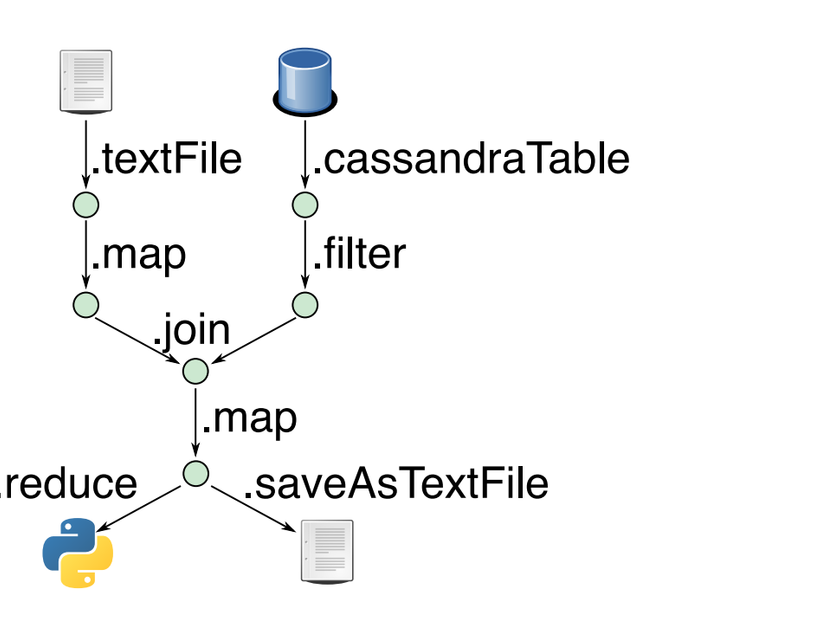
```
def add(a, b):
    return a + b

sc.parallelize([1,2,3,4]).reduce(add) == 10

pairs = sc.parallelize([('a',1), ('a',3), ('b',2)])
pairs.reduceByKey(add).collect() == [('a', 4), ('b', 2)]
pairs.reduce(add) == ('a', 1, 'a', 3, 'b', 2)
```

The MapReduce "reduce" operation is `.reduceByKey`.

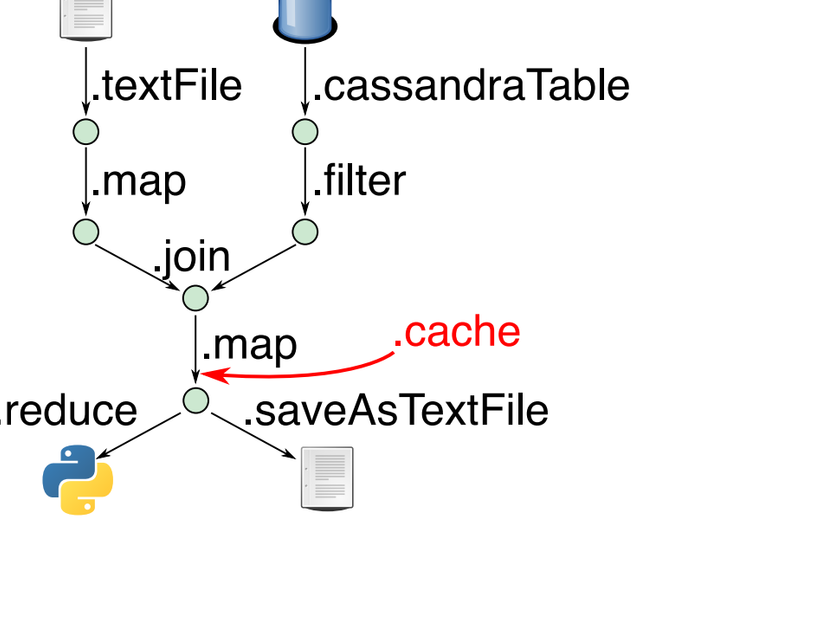
Spark DAG

The (Spark-related) work that the driver does is to indicate the way to calculate each RDD, so that can be done on the executor.

Spark thinks of this as building a ***directed-acyclic graph*** (*DAG*) of RDDs (vertices) and calculations (edges).



* * *



* * *

The DAG is what Spark uses to decide how to calculate your results.

- Non-dependant things in the DAG can be done in parallel.
- Operations in an RDD's "lineage" must be calculated before it (but more details next).
- An action on an RDD means that RDD needs to actually be calculated.
- A branch in the DAG means you should be caching. (Not automatic because Spark doesn't know what you'll do with the RDD later.)

Stages

Most of the RDDs you think you create never actually exist in memory. e.g.

```
rdd1 = sc.textFile(...)
rdd2 = rdd1.map(...)
rdd3 = rdd2.filter(...)
rdd3.saveAsTextFile(...)
```

Here, `rdd1` and `rdd2` never actually have to be created and stored in memory: the individual elements (in each partition) can be created as part of building `rdd3`.

An RDD that must actually be built in memory (***materialized***) marks the end of a ***stage***. Spark calculates a stage as one piece of (parallelized) work.

An action (method that doesn't just build another RDD) definitely ends a stage (and forces it to be materialized).

Caching (or checkpointing) an RDD forces the end of a stage (but doesn't force materialization).

So does any shuffle operation.

Job, Stages, Tasks

As we have seen, a stage is the end of a chain of RDDs that can be calculated as one "unit".

A ***job*** is the collection of stages that must be computed to perform an action.

A ***stage*** is part of the job that can be computed as one "operation" on the collection of partitions.

A ***task*** is the work of computing one stage on one partition. Done by one executor.

RDD Methods

A more complete view of the RDD methods:

- Actions: do not return another RDD. Either return a Python value (`.reduce`, `.count`) or have a side effect (`.saveAsTextFile`).
- Transformations: return a new RDD.
  - Shuffles: all-to-all operations that cause partitions to be rearranged (`.reduceByKey`, `.join`).
  - Non-shuffle: Can be done on existing partitions (`.map`, `.filter`, `.sample`).