CS 677: Big Data

Spark

Lecture 16

Today's Schedule

- Some Spark Background
- Operations
- Persistence
- Spark Programming Tips

Today's Schedule

- **Some Spark Background**
- Operations
- Persistence
- Spark Programming Tips

Diverging Paths

- MR was the first step on an ongoing journey
- There are two roads to follow to keep improving things in this space:
  - Computation
  - Storage

Storage

- GFS/HDFS were not exactly groundbreaking but made distributed file systems mainstream
- Kicked off research in storage systems / databases:
  - NoSQL! Whoo!
    - Wait a minute…
  - NewSQL?! Hooray!
- The conclusion: we need more than blob storage
  - …users expect database-like properties

Computation

- Why are we writing everything to disk in between phases?
- Machine learning algorithms don't map well to this process
- "MR everything" forces us into some awkward scenarios from a programming perspective
- Even Java isn't great for these types of computational use cases

Now What?

- Google basically no longer uses MapReduce
  - **But** the paradigm itself is still alive and well
- Spark, Flink, etc. have recently become popular
- Hadoop, like many "cool" technologies, was over-prescribed
  - It's still a great option for processing gigantic amounts of data in a batch fashion

Put Away Your Pitchforks...

```
df.rdd
    .filter(
        lambda row: row.geohash.startswith(prefixes))
    .map(
        lambda row: (
            timestamp_to_month(row.Timestamp),
            row.relative_humidity_zerodegc_isotherm))
    .reduceByKey(
        lambda humidity1, humidity2:
            (humidity1 + humidity2) / 2.0)
    .collect()
```

Why Spark

- Spark augments MapReduce paradigm by adding several built-in functions and supporting in-memory computations
- Development is chugging along, whereas Hadoop is more or less in maintenance mode
  - Huge leap in features and speed from 1.0x to 3.0
- Inputs are represented as RDDs, which have two primary operations:
  - Transformations
  - Actions

Today's Schedule

- Some Spark Background
- **Operations**
- Persistence
- Spark Programming Tips

Transformations

- Applied to RDDs to produce new RDD **states**
  - We're modifying the lineage, not doing computations (yet)
- Examples:
  - Splitting each line in the RDD into words
  - Incrementing each number
  - Removing rows that match certain conditions
- Important: transformations are lazy. They are only applied when a terminal **action** is present!

Actions

- Return something to the driver or produce some type of terminal result
- Cause computations to **execute**
- Could be a count of matching records
  - `.count()`
- Or actual row values
  - `.take(50)`
- Or even saving the result of several transformations to HDFS or the local file system

Shuffle

- Many actions will result in shuffle operations
- The mechanism here is very similar to MapReduce
  - In fact, there is a Map and Reduce phase
- Let's say Spark needs to create a new RDD after doing our classic word count job
- It has to do a reduction based on keys (the words) and add up the values (counts)
  - This is an "all-to-all" operation

Transformations

- map (applies a function to each row of the RDD)
- filter (only keeps rows that satisfy a condition)
- sort
- distinct
- join
- intersection / union / cartesian
- group / reduce / aggregate / sort **by key**

Actions

- reduce (apply a reduction. Given two elements, the function supplied should return a single element)
- count (retrieve the number of rows in the RDD)
- take (get the first N rows of the RDD)
- foreach (apply a function to each row)
- collect (transfers the RDD to the driver)
  - **AVOID** if possible!
- Also: saveAsXXXX(…)

Today's Schedule

- Some Spark Background
- Operations
- **Persistence**
- Spark Programming Tips

Persistence [1/2]

- There are two main ways to "checkpoint" RDDs in your Spark jobs
- rdd.cache() – persists the RDD in memory. Good for storing the outcomes of several transformations for further manipulation
  - Fast… but will use memory, of course
- rdd.persist() – the more advanced form of persistence

Persistence [2/2]

- You can pass several options to rdd.persist():
  - MEMORY_ONLY
  - MEMORY_AND_DISK
  - DISK_ONLY
  - OFF_HEAP (experimental)
  - etc

Persistence Alternatives

- saveAsTextFile
- saveAsSequenceFile() (Java + Scala)
- saveAsObjectFile() (Java + Scala)
- saveAsPickleFile (Python)

Today's Schedule

- Some Spark Background
- Operations
- Persistence
- **Spark Programming Tips**

Pitfall 1: The Driver

- Don't put too much strain on your application's driver (Jupyter, ipython, spark shell, etc.)
- If you are constantly transferring data to the driver and processing it there, you're subverting the framework
- One thing I see frequently: .collect(), then iterating through the data, then producing a new RDD
  - **Bad idea!**

Pitfall 2: Caching

- Caching is awesome!
- Except when it isn't.
- If you cache too much data, you'll run out of memory
- You should only cache an RDD if your logic branches from a particular point and you want to do different transformations on it (or iterative processing)

Pitfall 3: Global State

- Just like with MapReduce, MPI, BSP, etc. you have to be careful with global state
- Let's say you pass a function to .map() that operates on a global variable in your code
  - This might work fine on your local machine, but what about when you run on the cluster?
- When distributed, the individual workers don't know about that global state anymore – they have their own copy

Pitfall 4: Magic

- Spark unfortunately is not magic
- It might seem like magic after using MapReduce
- But be careful! It can still crash, run out of memory, and if you use the programming model incorrectly it can be quite slow!