Spark: It’s all about transformation and actions

Transformations
- Wrangle with the data
- Consume, and beget, an RDD
- Flock together ... to form daisy chains

But it is actions
- That trigger evaluations
- Providing them potency
- Revealing their expressive power

Frequently asked questions from the previous class survey

- Are the Spark model fitting libraries distributed?
- What if the data is too big to fit in memory of a large, distributed cluster?
Topics covered in this lecture

- Spark APIs
- Resilient Distributed Datasets
- Common Transformations and Actions
Spark APIs

- Spark has two fundamental sets of APIs:
  - The low-level “unstructured” APIs, and
  - The higher-level structured APIs

Structured APIs

- Structured APIs are a tool for manipulating all sorts of data
  - From unstructured log files to semi-structured CSV files and highly structured Parquet files
- Refers to three core types of distributed collection APIs:
  - Datasets
  - DataFrames
  - SQL tables and views
- Majority of the Structured APIs apply to both batch and streaming computation
Spark’s Toolset

- Structured Streaming
- Advanced Analytics
- Libraries & Ecosystem

- Structured APIs
  - Datasets
  - DataFrames
  - SQLs

- Low Level APIs
  - RDDs
  - Distributed variables

Spark has two notions of structured collections: DataFrames and Datasets

- DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns

- Each column:
  - Must have the same number of rows as all the other columns (although you can use null to specify the absence of a value)
  - Has type information that must be consistent for every row in the collection.
DataFrames versus Datasets

- DataFrames are considered “untyped”
- Datasets are considered “typed”

How does Spark view DataFrames and Datasets?

- To Spark, DataFrames and Datasets represent immutable, lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output.
- When we perform an action on a DataFrame, we instruct Spark to perform the actual transformations and return the result.
- These represent plans of how to manipulate rows and columns to compute the user’s desired result.
The DataFrame is the most common Structured API

- Simply represents a table of data with rows and columns
- The list that defines the columns and the types within those columns is called the schema

The DataFrame concept is not unique to Spark

- R and Python both have similar concepts
  - However, Python/R DataFrames (with some exceptions) exist on one machine rather than multiple machines
  - This limits what you can do with a given DataFrame to the resources that exist on that specific machine
- A Spark DataFrame can span thousands of computers
Core Spark Concepts

- Drivers
- SparkContext
- Executors
Spark in a nutshell

- Spark allows users to write a program for the **driver** (or master node) on a cluster computing system that can perform **operations** on data in parallel.
- Spark represents large datasets as **RDDs** which are stored in the executors (or worker nodes).
- The objects that comprise RDDs are called **partitions** and may be (but do not need to be) computed on different nodes of a distributed system.
- The Spark cluster manager handles starting and distributing the Spark executors across a distributed system.

Drivers

- Every Spark application consists of a **driver** program.
- Driver **launches various parallel operations** on the cluster.
- Constituent elements:
  - Application’s main function
  - Defines distributed datasets on the clusters
  - Applies operations to these datasets
SparkContext

- Driver programs access Spark through a SparkContext object
  - Represents a connection to a computing cluster

- Within the shell?
  - Created as the variable `sc`
    - You can even print out `sc` to see the type

- Once you have a SparkContext, you can use it to build RDDs
  - And then run operations on the data ...

Executors

- Driver programs manage a number of nodes, called `executors`

- Executors are responsible for running operations

- For example:
  - If we were running a `count()` operation on cluster
    - Different machines might count lines in different ranges of the file
Components for distributed execution in Spark

Lot of Spark’s API revolves around passing functions to its operators

```python
def hasPython(line):
    return "Python" in line

pythonLines =
    lines.filter(hasPython)

pythonLines =
    lines.filter(line => line.contains("Python"))

Also known as the `lambda` or `=>` syntax
Lot of Spark’s API revolves around passing functions to its operators

JavaRDD<String> pythonLines = lines.filter(new Function<String, Boolean>() {
    Boolean call(String line) {
        return line.contains("Python");
    }
});

JavaRDD<String> pythonLines = lines.filter(line -> line.contains("Python"));

RESILIENT DISTRIBUTED DATASET [RDD]
Resilient Distributed Dataset (RDD)

- RDD is an **immutable distributed collection** of objects
- Each RDD is split into **multiple partitions**
  - Maybe computed on different nodes in the cluster
- Can contain any type of Java, Scala, or Python objects
  - Including user-defined classes

Creation of RDDs

1. Loading an external dataset
2. Distributing a collection of objects via the driver program

```python
>>> lines = sc.textFile("README.md")
```
Once created, RDDs offer two types of operations

- **Transformations**
  - Construct a new RDD from a previous one
  - E.g.: Filtering data that matches a predicate

- **Actions**
  - Compute a result based on an RDD
  - Return result to the driver program or save it in an external storage system (HDFS)

Some more about RDDs

- Although you can define new RDDs anytime
  - Spark computes them in a **lazy fashion**
  - When?
    - The first time they are used in an action

- Loading lazily allows transformations to be performed **before** the action
Lazy loading allows Spark to see the whole chain of transformations

- Allows it to **compute just the data needed** for the result
- Example:
  ```python
  lines = sc.textFile("README.md")
  pythonLines = lines.filter(lambda line: "Python" in line)
  ```
- If Spark were to load and store all lines in the file, as soon as we wrote `lines=sc.textFile()`?
  - Would waste a lot of storage space, since we immediately filter out a lot of lines

RDD and actions

- RDDs are **recomputed** (by default) every time you run an action on them
- If you wanted to **reuse** an RDD?
  - Ask Spark to **persist** it using `RDD.persist()`
  - After computing it the first time, Spark will store RDD contents in memory (**partitioned** across cluster machines)
  - Persisted RDD is used in future actions
Rdds: memory residency and immutability implications

- Spark can keep an RDD loaded in-memory on the executor nodes throughout the life of a Spark application for faster access in repeated computations.
- RDDs are immutable, so transforming an RDD returns a new RDD rather than the existing one.
- Cross-cutting implications?
  - Lazy evaluation, in-memory storage, and immutability allows Spark to be easy-to-use, fault-tolerant, scalable, and efficient.

Every Spark program and shell works as follows

1. **Create** some input RDD from external data.
2. **Transform** them to define new RDDs using transformations like `filter()`.
3. Ask Spark to **persist()** any intermediate RDDs that needs to be reused.
4. **Launch actions** such as `count()`, etc. to kickoff a parallel computation.
   - Computing is optimized and executed by Spark.
A CLOSER LOOK AT RDD OPERATIONS

RDDs support two types of operations

- Transformations
  - Operations that return a new RDD. E.g.: `filter()`

- Actions
  - Operations that return a result to the driver program or write to storage
  - Kicks of a computation. E.g.: `count()`

- Distinguishing aspect?
  - Transformations return RDDs
  - Actions return some other data type
Transformations

- Many transformations are **element-wise**
  - Work on only one element at a time

- Some transformations are not element-wise
  - E.g.: We have a logfile, `log.txt`, with several messages, but we only want to select error messages
    ```python
    inputRDD = sc.textFile("log.txt")
    errorsRDD = inputRDD.filter(lambda x: "error" in x)
    ```

In our previous example ...

- **filter** does not mutate inputRDD
  - Returns a pointer to an entirely new RDD
  - inputRDD can still be reused later in the program

- We could use inputRDD to search for lines with the word “warning”
  - While we are at it, we will use another transformation, `union()`, to print number of lines that contained either
    ```python
    errorsRDD = inputRDD.filter(lambda x: "error" in x)
    warningsRDD = inputRDD.filter(lambda x: "warning" in x)
    badlinesRDD = errorsRDD.union(warningsRDD)
    ```
In our previous example

- Note how `union()` is different from `filter()`
  - Operates on 2 RDDs instead of one

- Transformations can actually operate on any number of RDDs

RDD Lineage graphs

- As new RDDs are derived from each other using transformations, Spark tracks dependencies
  - Lineage graph

- Uses lineage graph to
  - Compute each RDD on demand
  - Recover lost data if part of persistent RDD is lost
RDD lineage graph for our example

inputRDD

errorsRDD

warningsRDD

filter

union

badLinesRDD

Actions

- We can create RDDs from each other using transformations
- At some point, we need to actually do something with the dataset
  - Actions
- Forces evaluations of the transformations required for the RDD they were called on
Let’s try to print information about badlinesRDD

```python
print "Input had " + badLinesRDD.count() + " concerning lines"
print "here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

RDDs also have a `collect` to retrieve the entire RDD

- Useful if program filters RDD to a very small size and you want to deal locally
  - Your entire dataset must fit in memory on a single machine to use `collect()` on it
    - Should NOT be used on large datasets
- In most cases, RDDs **cannot** be `collect()`ed to the driver
  - Common to write data out to a distributed storage system ... HDFS or S3
Lazy Evaluation

- Transformations on RDDs are **lazily evaluated**
  - Spark will not begin to execute until it sees an action

- Uses this to **reduce the number of passes** it has to take over data by grouping operations together

- What does this mean?
  - When you call a transformation on an RDD (for e.g., `map`) the operation is not immediately performed
  - Spark internally records metadata that operation is requested

---

How you should think of RDDs

- Rather than thinking of it as containing specific data
  - Best to think of it as **containing instructions on how to compute the data** that we build through transformations

- Loading data into a RDD is lazily evaluated just as transformations are
COMMON TRANSMFORMATIONS AND ACTIONS

Element-wise transformations: filter()

- Takes in a function and returns an RDD that only has elements that pass the filter() function
Element-wise transformations: `map()`

- Takes in a function and applies it to each element in the RDD
- Result of the function is the new value of each element in the resulting RDD

```
inputRDD = {1, 2, 3, 4}
map x => x*x
Mapped RDD = {1, 4, 9, 16}
filter x => x !=1
Filtered RDD = {2, 3, 4}
```

Things that can be done with `map()`

- Fetch website associated with each URL in collection to just squaring numbers
- `map()`’s return type does not have to be the same as its input type
- Multiple output elements for each input element?
  - Use `flatMap()`

```
lines=sc.parallelize([“hello world”, “hi”])
words=lines.flatMap(lambda line: line.split(” “))
words.first()  # returns hello
```
Difference between `map` and `flatMap`

- `RDD1.map(tokenize)`
  - `mappedRDD` = `{{"coffee", "panda"}, {"happy", "panda"}, {"happiest", "panda", "party"}}`
- `RDD1.flatMap(tokenize)`
  - `flatMappedRDD` = `{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}`

Psuedo set operations

- RDDs support many of the operations of mathematical sets such as union, intersection, etc.
- Even when the RDDs themselves are not properly sets
Some simple set operations

- RDD1 = {coffee, coffee, panda, tiger, tea}
- RDD2 = {coffee, tiger, snake}
- RDD1.distinct() = {coffee, tiger, panda, tea}
- RDD1.union(RDD2) = {coffee, coffee, coffee, panda, tiger, tiger, tea, snake}
- RDD1.intersection(RDD2) = {coffee, tiger}
- RDD1.subtract(RDD2) = {panda, tea}

Cartesian product between two RDDs

- RDD1 = {User1, User2, User3}
- RDD2 = {Venue("Betabrand"), Venue("Asha Tree House"), Venue("Ritual")}
- RDD1.cartesian(RDD2) = {(User1, Venue("Betabrand")), (User1, Venue("Asha Tree House")), (User1, Venue("Ritual")), (User2, Venue("Betabrand")), (User2, Venue("Asha Tree House")), (User2, Venue("Ritual")), (User3, Venue("Betabrand")), (User3, Venue("Asha Tree House")), (User3, Venue("Ritual"))}
Actions on Basic RDDs

- **reduce()**
  - Takes a function that operates on two elements in the RDD; returns an element of the same type
  - E.g., of such an operation? + sums the RDD
  
  ```scala
  sum = rdd.reduce((x,y) => x + y)
  ```

- **fold()** takes a function with the same signature as `reduce()`, but also takes a “zero value” for initial call
  - “Zero value” is the **identity element** for initial call
  - E.g., 0 for +, 1 for *, empty list for concatenation
Both `fold()` and `reduce()` require return type of same type as the RDD elements

- The `aggregate()` removes that constraint
  - For e.g., when computing a running average, maintain both the count so far and the number of elements

**Examples: Basic Actions on RDDs**
Examples: Basic actions on RDDs

- Our RDD contains \(\{1, 2, 3, 3\}\)
- \texttt{collect()}
  - Return all elements from the RDD
  - Invocation: \texttt{rdd.collect()}
  - Result: \(\{1, 2, 3, 3\}\)

Examples: Basic actions on RDDs

- Our RDD contains \(\{1, 2, 3, 3\}\)
- \texttt{count()}
  - Number of elements in the RDD
  - Invocation: \texttt{rdd.count()}
  - Result: 4
Examples: Basic actions on RDDs

- Our RDD contains \{1, 2, 3, 3\}
- **countByValue()**
  - Number of times each element occurs in the RDD
  - **Invocation:** `rdd.countByValue()`
  - **Result:** \{(1,1), (2,1), (3,2)\}

Examples: Basic actions on RDDs

- Our RDD contains \{1, 2, 3, 3\}
- **take(num)**
  - Return `num` elements from the RDD
  - **Invocation:** `rdd.take(2)`
  - **Result:** \{1, 2\}
Examples: Basic actions on RDDs

- Our RDD contains \{1, 2, 3, 3\}

- **reduce(func)**
  - Combine the elements of the RDD together in parallel
  - Invocation: `rdd.reduce((x, y) => x + y)`
  - Result: 9

- **aggregate(zeroValue)(seqOp, combOp)**
  - Similar to `reduce()` but used to return a different type
  - Invocation:
    - `rdd.aggregate((0,0))`
    - `((x,y) => (x._1 + y, x._2 + 1),
      (x,y) => (x._1 + y._1, x._2 + y._2))`
  - Result: (9, 4)
Examples: Basic actions on RDDs

- Our RDD contains \{1, 2, 3, 3\}

- **foreach**\( (\text{func}) \)**
  - Apply the provided function to each element of the RDD
  - **Invocation:** \( \text{rdd.foreach(func)} \)
  - **Result:** Nothing

**PERSISTENCE (CACHING)**
Why persistence?

- Spark RDDs are lazily evaluated, and we may sometimes wish to use the same RDD multiple times.
  - Naively, Spark will **recompute RDD and all of its dependencies** each time we call an action on the RDD.
    - Super expensive for iterative algorithms.
- To avoid recomputing RDD multiple times?
  - Ask Spark to **persist** the data.
  - The nodes that compute the RDD, store the partitions.
  - **E.g.:** `result.persist(StorageLevel.DISK_ONLY)`

Coping with failures

- If a node that has data persisted on it fails?
  - Spark recomputes lost partitions of data when needed.
- Also, replicate data on multiple nodes.
  - To handle node failures without slowdowns.
Persistence Levels for Spark

<table>
<thead>
<tr>
<th>Level</th>
<th>Space Used</th>
<th>CPU time</th>
<th>In Memory</th>
<th>On disk</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>High</td>
<td>Low</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Low</td>
<td>High</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>High</td>
<td>Medium</td>
<td>Some</td>
<td>Some</td>
<td>Spills to disk if there is too much data to fit in memory</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>Low</td>
<td>High</td>
<td>Some</td>
<td>Some</td>
<td>Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory</td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>Low</td>
<td>High</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

What if you attempt to cache too much data that does not fit in memory?

- Spark will **evict old partitions** using a Least Recently Used Cache policy
  - For memory only storage partitions, it will be recomputed the next time they are accessed
  - For memory_and_disk ones? Write them out to disk

- RDDs also come with a method, **unpersist()**
  - Manually remove data elements from the cache
The contents of this slide-set are based on the following references

