Frequently asked questions from the previous class survey

- If RDDs are recomputed after every action, what's the point of having them immutable?
- Lazy evaluation: Why is this important?
Topics covered in this lecture

- Resilient Distributed Datasets
- Common Transformations and Actions

RESILIENT DISTRIBUTED DATASET [RDD]
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.

April 2, 2019
Professor: SHRIDEEPPALLICKARA

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 log file, `log.txt`, with several messages, but we only want to select error messages
    ```
    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
    ```
    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
   filter
errorsRDD
   filter
  warningsRDD
     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

```scala
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 TRANSFORMATIONS 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: \texttt{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

\begin{center}
\begin{tikzpicture}
  \node[draw, fill=gray!30, minimum width=2cm, minimum height=1.5cm] (input) at (0,0) {inputRDD \{1,2,3,4\}};
  \node[draw, fill=gray!30, minimum width=2cm, minimum height=1.5cm] (map) at (3,0) {mapped RDD \{1,4,9,16\}};
  \node[draw, fill=gray!30, minimum width=2cm, minimum height=1.5cm] (filter) at (3,-1) {filtered RDD \{2,3,4\}};

  \draw[->, thick] (input) -- (map) node[midway, above] {map x \Rightarrow x \times x};
  \draw[->, thick] (map) -- (filter) node[midway, above] {filter x \Rightarrow x \neq 1};
\end{tikzpicture}
\end{center}

Things that can be done with \texttt{map()} 

- Fetch website associated with each URL in collection to just squaring numbers
- \texttt{map()}’s return type does not have to be the same as its input type
- Multiple output elements for each input element?
  - Use \texttt{flatMap()}
    \begin{verbatim}
    lines=sc.parallelize(“hello world”, “hi”)
    words=lines.flatMap(lambda line: line.split(“ ”))
    words.first()  # returns hello
    \end{verbatim}
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, tea, snake}

RDD1.intersection(RDD2)
{coffee, tiger}

RDD1.subtract(RDD2)
{panda, tea}

April 2, 2019

Professor: SHRIDEEP PALICKARA

Cartesian product between two RDDs

RDD1
{User1, User2, User3}

RDD2
{Venue("Betabrand"), Venue("Asha Tree House"), Venue("Ritual")}

cartesian

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"))}
**COMMON ACTIONS**

**April 2, 2019**

### 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

- **collect()**
  - Return all elements from the RDD
  - **Invocation:** `rdd.collect()`
  - **Result:** `{1, 2, 3, 3}`

Example: Basic actions on RDDs

- **count()**
  - Number of elements in the RDD
  - **Invocation:** `rdd.count()`
  - **Result:** `4`
Examples: Basic actions on RDDs [3/7]

- 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 [4/7]

- 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 [5/7]

- 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

Examples: Basic actions on RDDs [6/7]

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

- `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 [7/7]

- Our RDD contains \{1, 2, 3, 3\}
- `foreach(func)`
  - Apply the provided function to each element of the RDD
  - Invocation: `rdd.foreach(func)`
  - Result: Nothing
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>DISKONLY</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
WORKING WITH KEY/VALUE PAIRS

April 2, 2019

RDDs of key/value pairs

- Key/value RDDs are commonly used to perform aggregations
  - Might have to do ETL (Extract, Transform, and Load) to get data into key/value formats

- Advanced feature to control layout of pair RDDs across nodes
  - **Partitioning**
RDDs containing key/value pairs

- Are called **pair RDDs**
- Useful **building block** in many programs
  - Expose operations that allow actions on each key in parallel or regroup data across network
  - `reduceByKey()` to aggregate data separately for each key
  - `join()` to merge two RDDs together by grouping elements of the same key
Pair RDDs

- RDDs that contain **key/value pairs**
- Expose partitions that allow you to act on each key in parallel or regroup data across the network

Creating Pair RDDs

- `pairs=lines.map(lambda x: (x.split(" "))[0], x))`
  - Creates a pairRDD using the first word as the key
- **Java does not have a built-in tuple type**
  - `scala.Tuple2` class
    - `new Tuple2(elem1, elem2)`
The contents of this slide-set are based on the following references
